Lemma. *If $f_1(x)$ and $f_2(x)$ are relatively prime polynomials, there exist polynomials $q_1(x)$ and $q_2(x)$ such that*

$$q_1(x)f_1(x) + q_2(x)f_2(x) = 1,$$

where 1 denotes the constant polynomial with value 1.

Proof. Without loss of generality, assume that the degree of $f_1(x)$ is greater than or equal to the degree of $f_2(x)$. The proof is by mathematical induction on the degree of $f_2(x)$. If $f_2(x)$ has degree 0, then $f_2(x)$ is a nonzero constant c. In this case, we can take $q_1(x) = 0$ and $q_2(x) = 1/c$.

Now suppose that the theorem holds whenever the polynomial of lesser degree has degree less than n for some positive integer n, and suppose that $f_2(x)$ has degree n. By the division algorithm, there exist polynomials $q(x)$ and $r(x)$ such that $r(x)$ has degree less than n and

$$f_1(x) = q(x)f_2(x) + r(x). \tag{5}$$

Since $f_1(x)$ and $f_2(x)$ are relatively prime, $r(x)$ is not the zero polynomial. We claim that $f_2(x)$ and $r(x)$ are relatively prime. Suppose otherwise; then there exists a polynomial $g(x)$ of positive degree that divides both $f_2(x)$ and $r(x)$. Hence, by (5), $g(x)$ also divides $f_1(x)$, contradicting the fact that $f_1(x)$ and $f_2(x)$ are relatively prime. Since $r(x)$ has degree less than n, we may apply the induction hypothesis to $f_2(x)$ and $r(x)$. Thus there exist polynomials $g_1(x)$ and $g_2(x)$ such that

$$g_1(x)f_2(x) + g_2(x)r(x) = 1. \tag{6}$$

Combining (5) and (6), we have

$$\begin{aligned} 1 &= g_1(x)f_2(x) + g_2(x)\left[f_1(x) - q(x)f_2(x)\right] \\ &= g_2(x)f_1(x) + \left[g_1(x) - g_2(x)q(x)\right]f_2(x). \end{aligned}$$

Thus, setting $q_1(x) = g_2(x)$ and $q_2(x) = g_1(x) - g_2(x)q(x)$, we obtain the desired result. ∎

Theorem E.2. *If $f_1(x), f_2(x), \ldots, f_n(x)$ are relatively prime polynomials, there exist polynomials $q_1(x), q_2(x), \ldots, q_n(x)$ such that*

$$q_1(x)f_1(x) + q_2(x)f_2(x) + \cdots + q_n(x)f_n(x) = 1,$$

where 1 denotes the constant polynomial with value 1.

Proof. The proof is by mathematical induction on n, the number of polynomials. The preceding lemma establishes the case $n = 2$.

Now assume the result is true for fewer than n polynomials, for some $n \geq 3$. We will first consider the trivial case where the first $n-1$ polynomials

are relatively prime. By the induction hypothesis, there exist polynomials $q_1(x), q_2(x), \ldots, q_{n-1}(x)$ such that

$$q_1(x)f_1(x) + q_2(x)f_2(x) + \cdots + q_{n-1}(x)f_{n-1}(x) = 1.$$

Setting $q_n(x) = 0$, the zero polynomial, we have

$$q_1(x)f_1(x) + q_2(x)f_2(x) + \cdots + q_n(x)f_n(x) = 1,$$

which proves the result if the first $n-1$ polynomials are relatively prime.

Now suppose that the first $n-1$ polynomials are not relatively prime, and let $g(x)$ be the monic polynomial of maximum positive degree that divides each of these polynomials. For $k = 1, 2, \ldots, n-1$, let $h_k(x)$ be the polynomial defined by $f_k(x) = g(x)h_k(x)$. Then the polynomials $h_1(x), h_2(x), \ldots, h_{n-1}(x)$ are relatively prime. So by the induction hypothesis, there exist polynomials $\phi_1(x), \phi_2(x), \ldots, \phi_{n-1}(x)$ such that

$$\phi_1(x)h_1(x) + \phi_2(x)h_2(x) + \cdots + \phi_{n-1}(x)h_{n-1}(x) = 1.$$

Multiplying both sides of this equation by $g(x)$, we obtain

$$\phi_1(x)f_1(x) + \phi_2(x)f_2(x) + \cdots + \phi_{n-1}(x)f_{n-1}(x) = g(x).$$

Note that $g(x)$ and $f_n(x)$ are relatively prime, and hence there exist polynomials $p(x)$ and $q_n(x)$ such that

$$p(x)g(x) + q_n(x)f_n(x) = 1.$$

Let $q_i(x) = p(x)\phi_i(x)$ for $i = 1, 2, \ldots, n-1$. Then combining the two preceding equations yields

$$q_1(x)f_1(x) + q_2(x)f_2(x) + \cdots + q_{n-1}(x)f_{n-1}(x) + q_n(x)f_n(x) = 1,$$

which completes the induction argument.. ∎

Example 1

Let $f_1(x) = x-1$, $f_2(x) = x^2+x-2$ $f_3(x) = x^2-x-1$, and $f_4(x) = x+1$. As polynomials with real coefficients, $f_1(x), f_2(x), f_3(x)$ and $f_4(x)$ are relatively prime. It is easily verified that the polynomials $q_1(x) = -x^2$, $q_2(x) = x$, $q_3(x) = -3$, and $q_4(x) = x-2$ satisfy

$$q_1(x)f_1(x) + q_2(x)f_2(x) + q_3(x)f_3(x) + q_4(x)f_4(x) = 1,$$

and hence these polynomials satisfy the conclusion of Theorem E.2. ◆

Throughout Chapters 5, 6, and 7, we consider linear operators that are polynomials in a particular operator T and matrices that are polynomials in a particular matrix A. For these operators and matrices, the following notation is convenient.

Definitions. *Let*

$$f(x) = a_0 + a_1(x) + \cdots + a_n x^n$$

be a polynomial with coefficients from a field F. If T is a linear operator on a vector space V over F, we define

$$f(\mathsf{T}) = a_0\mathsf{I} + a_1\mathsf{T} + \cdots + a_n\mathsf{T}^n.$$

Similarly, if A is an $n \times n$ matrix with entries from F, we define

$$f(A) = a_0 I + a_1 A + \cdots + a_n A^n.$$

Example 2

Let T be the linear operator on R^2 defined by $\mathsf{T}(a,b) = (2a+b, a-b)$, and let $f(x) = x^2 + 2x - 3$. It is easily checked that $\mathsf{T}^2(a,b) = (5a+b, a+2b)$; so

$$\begin{aligned} f(\mathsf{T})(a,b) &= (\mathsf{T}^2 + 2\mathsf{T} - 3\mathsf{I})(a,b) \\ &= (5a+b, a+2b) + (4a+2b, 2a-2b) - 3(a,b) \\ &= (6a+3b, 3a-3b). \end{aligned}$$

Similarly, if

$$A = \begin{pmatrix} 2 & 1 \\ 1 & -1 \end{pmatrix},$$

then

$$f(A) = A^2 + 2A - 3I = \begin{pmatrix} 5 & 1 \\ 1 & 2 \end{pmatrix} + 2\begin{pmatrix} 2 & 1 \\ 1 & -1 \end{pmatrix} - 3\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 6 & 3 \\ 3 & -3 \end{pmatrix}. \quad \blacklozenge$$

The next three results use this notation.

Theorem E.3. *Let $f(x)$ be a polynomial with coefficients from a field F, and let T be a linear operator on a vector space V over F. Then the following statements are true.*

(a) *$f(\mathsf{T})$ is a linear operator on V.*

(b) *If β is a finite ordered basis for V and $A = [\mathsf{T}]_\beta$, then $[f(\mathsf{T})]_\beta = f(A)$.*

Proof. Exercise. ∎

Theorem E.4. *Let T be a linear operator on a vector space V over a field F, and let A be a square matrix with entries from F. Then, for any polynomials $f_1(x)$ and $f_2(x)$ with coefficients from F,*

(a) $f_1(\mathsf{T})f_2(\mathsf{T}) = f_2(\mathsf{T})f_1(\mathsf{T})$.

(b) $f_1(A)f_2(A) = f_2(A)f_1(A)$.

Proof. Exercise. ∎

Theorem E.5. *Let* T *be a linear operator on a vector space* V *over a field* F*, and let* A *be an* $n \times n$ *matrix with entries from* F*. If* $f_1(x)$ *and* $f_2(x)$ *are relatively prime polynomials with entries from* F*, then there exist polynomials* $q_1(x)$ *and* $q_2(x)$ *with entries from* F *such that*

(a) $q_1(\mathsf{T})f_1(\mathsf{T}) + q_2(\mathsf{T})f_2(\mathsf{T}) = \mathsf{I}$.

(b) $q_1(A)f_1(A) + q_2(A)f_2(A) = I$.

Proof. Exercise. ∎

In Chapters 5 and 7, we are concerned with determining when a linear operator T on a finite-dimensional vector space can be *diagonalized* and with finding a simple (canonical) representation of T. Both of these problems are affected by the factorization of a certain polynomial determined by T (the *characteristic polynomial* of T). In this setting, particular types of polynomials play an important role.

Definitions. *A polynomial* $f(x)$ *with coefficients from a field* F *is called* ***monic*** *if its leading coefficient is* 1*. If* $f(x)$ *has positive degree and cannot be expressed as a product of polynomials with coefficients from* F *each having positive degree, then* $f(x)$ *is called* ***irreducible****.*

Observe that whether a polynomial is irreducible depends on the field F from which its coefficients come. For example, $f(x) = x^2 + 1$ is irreducible over the field of real numbers, but it is not irreducible over the field of complex numbers since $x^2 + 1 = (x + i)(x - i)$.

Clearly any polynomial of degree 1 is irreducible. Moreover, for polynomials with coefficients from an algebraically closed field, the polynomials of degree 1 are the only irreducible polynomials.

The following facts are easily established.

Theorem E.6. *Let* $\phi(x)$ *and* $f(x)$ *be polynomials. If* $\phi(x)$ *is irreducible and* $\phi(x)$ *does not divide* $f(x)$*, then* $\phi(x)$ *and* $f(x)$ *are relatively prime.*

Proof. Exercise. ∎

Theorem E.7. *Any two distinct irreducible monic polynomials are relatively prime.*

Proof. Exercise. ∎

Theorem E.8. *Let* $f(x)$*,* $g(x)$*, and* $\phi(x)$ *be polynomials. If* $\phi(x)$ *is irreducible and divides the product* $f(x)g(x)$*, then* $\phi(x)$ *divides* $f(x)$ *or* $\phi(x)$ *divides* $g(x)$*.*

Proof. Suppose that $\phi(x)$ does not divide $f(x)$. Then $\phi(x)$ and $f(x)$ are relatively prime by Theorem E.6, and so there exist polynomials $q_1(x)$ and $q_2(x)$ such that

$$1 = q_1(x)\phi(x) + q_2(x)f(x).$$

Multiplying both sides of this equation by $g(x)$ yields

$$g(x) = q_1(x)\phi(x)g(x) + q_2(x)f(x)g(x). \tag{7}$$

Since $\phi(x)$ divides $f(x)g(x)$, there is a polynomial $h(x)$ such that $f(x)g(x) = \phi(x)h(x)$. Thus (7) becomes

$$g(x) = q_1(x)\phi(x)g(x) + q_2(x)\phi(x)h(x) = \phi(x)\left[q_1(x)g(x) + q_2(x)h(x)\right].$$

So $\phi(x)$ divides $g(x)$. ∎

Corollary. *Let $\phi(x), \phi_1(x), \phi_2(x), \ldots, \phi_n(x)$ be irreducible monic polynomials. If $\phi(x)$ divides the product $\phi_1(x)\phi_2(x)\cdots\phi_n(x)$, then $\phi(x) = \phi_i(x)$ for some i $(i = 1, 2, \ldots, n)$.*

Proof. We prove the corollary by mathematical induction on n. For $n = 1$, the result is an immediate consequence of Theorem E.7. Suppose then that for some $n > 1$, the corollary is true for any $n-1$ irreducible monic polynomials, and let $\phi_1(x), \phi_2(x), \ldots, \phi_n(x)$ be n irreducible polynomials. If $\phi(x)$ divides

$$\phi_1(x)\phi_2(x)\cdots\phi_n(x) = \left[\phi_1(x)\phi_2(x)\cdots\phi_{n-1}(x)\right]\phi_n(x),$$

then $\phi(x)$ divides the product $\phi_1(x)\phi_2(x)\cdots\phi_{n-1}(x)$ or $\phi(x)$ divides $\phi_n(x)$ by Theorem E.8. In the first case, $\phi(x) = \phi_i(x)$ for some i $(i = 1, 2, \ldots, n-1)$ by the induction hypothesis; in the second case, $\phi(x) = \phi_n(x)$ by Theorem E.7. ∎

We are now able to establish the unique factorization theorem, which is used throughout Chapters 5 and 7. This result states that every polynomial of positive degree is uniquely expressible as a constant times a product of irreducible monic polynomials.

Theorem E.9 (Unique Factorization Theorem for Polynomials). *For any polynomial $f(x)$ of positive degree, there exist a unique constant c; unique distinct irreducible monic polynomials $\phi_1(x), \phi_2(x), \ldots, \phi_k(x)$; and unique positive integers $n_1, n_2, \ldots, n_k$ such that*

$$f(x) = c[\phi_1(x)]^{n_1}[\phi_2(x)]^{n_2}\cdots[\phi_k(x)]^{n_k}.$$

Proof. We begin by showing the existence of such a factorization using mathematical induction on the degree of $f(x)$. If $f(x)$ is of degree 1, then $f(x) = ax + b$ for some constants a and b with $a \neq 0$. Setting $\phi(x) = x + b/a$,

we have $f(x) = a\phi(x)$. Since $\phi(x)$ is an irreducible monic polynomial, the result is proved in this case. Now suppose that the conclusion is true for any polynomial with positive degree less than some integer $n > 1$, and let $f(x)$ be a polynomial of degree n. Then

$$f(x) = a_n x^n + \cdots + a_1 x + a_0$$

for some constants a_i with $a_n \neq 0$. If $f(x)$ is irreducible, then

$$f(x) = a_n \left(x^n + \frac{a_{n-1}}{a_n} x^{n-1} + \cdots + \frac{a_1}{a_n} + \frac{a_0}{a_n} \right)$$

is a representation of $f(x)$ as a product of a_n and an irreducible monic polynomial. If $f(x)$ is not irreducible, then $f(x) = g(x)h(x)$ for some polynomials $g(x)$ and $h(x)$, each of positive degree less than n. The induction hypothesis guarantees that both $g(x)$ and $h(x)$ factor as products of a constant and powers of distinct irreducible monic polynomials. Consequently $f(x) = g(x)h(x)$ also factors in this way. Thus, in either case, $f(x)$ can be factored as a product of a constant and powers of distinct irreducible monic polynomials.

It remains to establish the uniqueness of such a factorization. Suppose that

$$\begin{aligned} f(x) &= c[\phi_1(x)]^{n_1}[\phi_2(x)]^{n_2} \cdots [\phi_k(x)]^{n_k} \\ &= d[\psi_1(x)]^{m_1}[\psi_2(x)]^{m_2} \cdots [\psi_r(x)]^{m_r}, \end{aligned} \tag{8}$$

where c and d are constants, $\phi_i(x)$ and $\psi_j(x)$ are irreducible monic polynomials, and n_i and m_j are positive integers for $i = 1, 2, \ldots, k$ and $j = 1, 2, \ldots, r$. Clearly both c and d must be the leading coefficient of $f(x)$; hence $c = d$. Dividing by c, we find that (8) becomes

$$[\phi_1(x)]^{n_1}[\phi_2(x)]^{n_2} \cdots [\phi_k(x)]^{n_k} = [\psi_1(x)]^{m_1}[\psi_2(x)]^{m_2} \cdots [\psi_r(x)]^{m_r}. \tag{9}$$

So $\phi_i(x)$ divides the right side of (9) for $i = 1, 2, \ldots, k$. Consequently, by the corollary to Theorem E.8, each $\phi_i(x)$ equals some $\psi_j(x)$, and similarly, each $\psi_j(x)$ equals some $\phi_i(x)$. We conclude that $r = k$ and that, by renumbering if necessary, $\phi_i(x) = \psi_i(x)$ for $i = 1, 2, \ldots, k$. Suppose that $n_i \neq m_i$ for some i. Without loss of generality, we may suppose that $i = 1$ and $n_1 > m_1$. Then by canceling $[\phi_1(x)]^{m_1}$ from both sides of (9), we obtain

$$[\phi_1(x)]^{n_1 - m_1}[\phi_2(x)]^{n_2} \cdots [\phi_k(x)]^{n_k} = [\phi_2(x)]^{m_2} \cdots [\phi_k(x)]^{m_k}. \tag{10}$$

Since $n_1 - m_1 > 0$, $\phi_1(x)$ divides the left side of (10) and hence divides the right side also. So $\phi_1(x) = \phi_i(x)$ for some $i = 2, \ldots, k$ by the corollary to Theorem E.8. But this contradicts that $\phi_1(x), \phi_2(x), \ldots, \phi_k(x)$ are distinct. Hence the factorizations of $f(x)$ in (8) are the same. ∎

It is often useful to regard a polynomial $f(x) = a_n x^n + \cdots + a_1 x + a_0$ with coefficients from a field F as a function $f\colon F \to F$. In this case, the value of f at $c \in F$ is $f(c) = a_n c^n + \cdots + a_1 c + a_0$. Unfortunately, for arbitrary fields there is not a one-to-one correspondence between polynomials and polynomial functions. For example, if $f(x) = x^2$ and $g(x) = x$ are two polynomials over the field Z_2 (defined in Example 4 of Appendix C), then $f(x)$ and $g(x)$ have different degrees and hence are not equal as polynomials. But $f(a) = g(a)$ for all $a \in Z_2$, so that f and g are equal polynomial functions. Our final result shows that this anomaly cannot occur over an infinite field.

Theorem E.10. *Let $f(x)$ and $g(x)$ be polynomials with coefficients from an infinite field F. If $f(a) = g(a)$ for all $a \in F$, then $f(x)$ and $g(x)$ are equal.*

Proof. Suppose that $f(a) = g(a)$ for all $a \in F$. Define $h(x) = f(x) - g(x)$, and suppose that $h(x)$ is of degree $n \geq 1$. It follows from Corollary 2 to Theorem E.1 that $h(x)$ can have at most n zeroes. But

$$h(a) = f(a) - g(a) = 0$$

for every $a \in F$, contradicting the assumption that $h(x)$ has positive degree. Thus $h(x)$ is a constant polynomial, and since $h(a) = 0$ for each $a \in F$, it follows that $h(x)$ is the zero polynomial. Hence $f(x) = g(x)$. ∎

Answers to Selected Exercises

CHAPTER 1

SECTION 1.1

1. Only the pairs in **(b)** and **(c)** are parallel.

2. **(a)** $x = (3, -2, 4) + t(-8, 9, -3)$ **(c)** $x = (3, 7, 2) + t(0, 0, -10)$

3. **(a)** $x = (2, -5, -1) + s(-2, 9, 7) + t(-5, 12, 2)$
(c) $x = (-8, 2, 0) + s(9, 1, 0) + t(14, -7, 0)$

SECTION 1.2

1. **(a)** T **(b)** F **(c)** F **(d)** F **(e)** T **(f)** F
(g) F **(h)** F **(i)** T **(j)** T **(k)** T

3. $M_{13} = 3$, $M_{21} = 4$, and $M_{22} = 5$

4. **(a)** $\begin{pmatrix} 6 & 3 & 2 \\ -4 & 3 & 9 \end{pmatrix}$ **(c)** $\begin{pmatrix} 8 & 20 & -12 \\ 4 & 0 & 28 \end{pmatrix}$
(e) $2x^4 + x^3 + 2x^2 - 2x + 10$ **(g)** $10x^7 - 30x^4 + 40x^2 - 15x$

13. No, (VS 4) fails.

14. Yes

15. No

17. No, (VS 5) fails.

22. 2^{mn}

SECTION 1.3

1. **(a)** F **(b)** F **(c)** T **(d)** F **(e)** T **(f)** F **(g)** F

2. **(a)** $\begin{pmatrix} -4 & 5 \\ 2 & -1 \end{pmatrix}$; the trace is -5 **(c)** $\begin{pmatrix} -3 & 0 & 6 \\ 9 & -2 & 1 \end{pmatrix}$
(e) $\begin{pmatrix} 1 \\ -1 \\ 3 \\ 5 \end{pmatrix}$ **(g)** $\begin{pmatrix} 5 & 6 & 7 \end{pmatrix}$

8. **(a)** Yes **(c)** Yes **(e)** No

11. No, the set is not closed under addition.

15. Yes

SECTION 1.4

1. **(a)** T **(b)** F **(c)** T **(d)** F **(e)** T **(f)** F

2. **(a)** $\{r(1,1,0,0)+s(-3,0,-2,1)+(5,0,4,0)\colon r,s\in R\}$
(c) There are no solutions.
(e) $\{r(10,-3,1,0,0)+s(-3,2,0,1,0)+(-4,3,0,0,5)\colon r,s\in R\}$

3. **(a)** Yes **(c)** No **(e)** No

4. **(a)** Yes **(c)** Yes **(e)** No

5. **(a)** Yes **(c)** No **(e)** Yes **(g)** Yes

SECTION 1.5

1. **(a)** F **(b)** T **(c)** F **(d)** F **(e)** T **(f)** T

2. **(a)** linearly dependent **(c)** linearly independent **(e)** linearly dependent
(g) linearly dependent **(i)** linearly independent

7. $\left\{\begin{pmatrix}1&0\\0&0\end{pmatrix},\begin{pmatrix}0&0\\0&1\end{pmatrix}\right\}$

11. 2^n

SECTION 1.6

1. **(a)** F **(b)** T **(c)** F **(d)** F **(e)** T **(f)** F
(g) F **(h)** T **(i)** F **(j)** T **(k)** T **(l)** T

2. **(a)** Yes **(c)** Yes **(e)** No

3. **(a)** No **(c)** Yes **(e)** No

4. No

5. No

7. $\{u_1,u_2,u_5\}$

9. $(a_1,a_2,a_3,a_4)=a_1u_1+(a_2-a_1)u_2+(a_3-a_2)u_3+(a_4-a_3)u_4$

10. **(a)** $-4x^2-x+8$ **(c)** $-x^3+2x^2+4x-5$

13. $\{(1,1,1)\}$

15. n^2-1

17. $\frac{1}{2}n(n-1)$

26. n

30. $\dim(\mathsf{W}_1)=3$, $\dim(\mathsf{W}_2)=2$, $\dim(\mathsf{W}_1+\mathsf{W}_2)=4$, and $\dim(\mathsf{W}_1\cap\mathsf{W}_2)=1$

SECTION 1.7

1. **(a)** F **(b)** F **(c)** F **(d)** T **(e)** T **(f)** T

CHAPTER 2

SECTION 2.1

1. **(a)** T **(b)** F **(c)** F **(d)** T **(e)** F **(f)** F **(g)** T **(h)** F

2. The nullity is 1, and the rank is 2. T is not one-to-one but is onto.

4. The nullity is 4, and the rank is 2. T is neither one-to-one nor onto.

5. The nullity is 0, and the rank is 3. T is one-to-one but not onto.

10. $\mathsf{T}(2,3) = (5,11)$. T is one-to-one. **12.** No.

SECTION 2.2

1. **(a)** T **(b)** T **(c)** F **(d)** T **(e)** T **(f)** F

2. **(a)** $\begin{pmatrix} 2 & -1 \\ 3 & 4 \\ 1 & 0 \end{pmatrix}$ **(c)** $\begin{pmatrix} 2 & 1 & -3 \end{pmatrix}$ **(d)** $\begin{pmatrix} 0 & 2 & 1 \\ -1 & 4 & 5 \\ 1 & 0 & 1 \end{pmatrix}$

(f) $\begin{pmatrix} 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & \cdots & 1 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 1 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \end{pmatrix}$ **(g)** $\begin{pmatrix} 1 & 0 & \cdots & 0 & 1 \end{pmatrix}$

3. $[\mathsf{T}]_\beta^\gamma = \begin{pmatrix} -\frac{1}{3} & -1 \\ 0 & 1 \\ \frac{2}{3} & 0 \end{pmatrix}$ and $[\mathsf{T}]_\alpha^\gamma = \begin{pmatrix} -\frac{7}{3} & -\frac{11}{3} \\ 2 & 3 \\ \frac{2}{3} & \frac{4}{3} \end{pmatrix}$

5. **(a)** $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ **(b)** $\begin{pmatrix} 0 & 1 & 0 \\ 2 & 2 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 2 \end{pmatrix}$ **(e)** $\begin{pmatrix} 1 \\ -2 \\ 0 \\ 4 \end{pmatrix}$

10. $\begin{pmatrix} 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$

SECTION 2.3

1. **(a)** F **(b)** T **(c)** F **(d)** T **(e)** F **(f)** F
(g) F **(h)** F **(i)** T **(j)** T

2. **(a)** $A(2B+3C) = \begin{pmatrix} 20 & -9 & 18 \\ 5 & 10 & 8 \end{pmatrix}$ and $A(BD) = \begin{pmatrix} 29 \\ -26 \end{pmatrix}$

(b) $A^tB = \begin{pmatrix} 23 & 19 & 0 \\ 26 & -1 & 10 \end{pmatrix}$ and $CB = \begin{pmatrix} 27 & 7 & 9 \end{pmatrix}$

3. **(a)** $[\mathsf{T}]_\beta = \begin{pmatrix} 2 & 3 & 0 \\ 0 & 3 & 6 \\ 0 & 0 & 4 \end{pmatrix}$, $[\mathsf{U}]_\beta^\gamma = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix}$, and $[\mathsf{UT}]_\beta^\gamma = \begin{pmatrix} 2 & 6 & 6 \\ 0 & 0 & 4 \\ 2 & 0 & -6 \end{pmatrix}$

4. **(a)** $\begin{pmatrix} 1 \\ -1 \\ 4 \\ 6 \end{pmatrix}$ **(c)** (5)

12. **(a)** No. **(b)** No.

SECTION 2.4

1. **(a)** F **(b)** T **(c)** F **(d)** F **(e)** T **(f)** F **(g)** T **(h)** T **(i)** T

2. **(a)** No **(b)** No **(c)** Yes **(d)** No **(e)** No **(f)** Yes

3. **(a)** No **(b)** Yes **(c)** Yes **(d)** No

19. **(a)** $[\mathsf{T}]_\beta = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

SECTION 2.5

1. **(a)** F **(b)** T **(c)** T **(d)** F **(e)** T

2. **(a)** $\begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix}$ **(c)** $\begin{pmatrix} 3 & -1 \\ 5 & -2 \end{pmatrix}$

3. **(a)** $\begin{pmatrix} a_2 & b_2 & c_2 \\ a_1 & b_1 & c_1 \\ a_0 & b_0 & c_0 \end{pmatrix}$ **(c)** $\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ -3 & 2 & 1 \end{pmatrix}$ **(e)** $\begin{pmatrix} 5 & -6 & 3 \\ 0 & 4 & -1 \\ 3 & -1 & 2 \end{pmatrix}$

4. $[\mathsf{T}]_{\beta'} = \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 1 & -3 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} = \begin{pmatrix} 8 & 13 \\ -5 & -9 \end{pmatrix}$

5. $[\mathsf{T}]_{\beta'} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{pmatrix}$

6. **(a)** $Q = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$, $[\mathsf{L}_A]_\beta = \begin{pmatrix} 6 & 11 \\ -2 & -4 \end{pmatrix}$

(c) $Q = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 2 \end{pmatrix}$, $[\mathsf{L}_A]_\beta = \begin{pmatrix} 2 & 2 & 2 \\ -2 & -3 & -4 \\ 1 & 1 & 2 \end{pmatrix}$

7. **(a)** $\mathsf{T}(x, y) = \dfrac{1}{1+m^2}((1-m^2)x + 2my, 2mx + (m^2-1)y)$

SECTION 2.6

1. **(a)** F **(b)** T **(c)** T **(d)** T **(e)** F **(f)** T **(g)** T **(h)** F

2. The functions in (a), (c), (e), and (f) are linear functionals.

3. **(a)** $\mathsf{f}_1(x, y, z) = x - \frac{1}{2}y$, $\mathsf{f}_2(x, y, z) = \frac{1}{2}y$, and $\mathsf{f}_3(x, y, z) = -x + z$

5. The basis for V is $\{p_1(x), p_2(x)\}$, where $p_1(x) = 2 - 2x$ and $p_2(x) = -\frac{1}{2} + x$.

7. **(a)** $\mathsf{T}^t(\mathsf{f}) = \mathsf{g}$, where $\mathsf{g}(a + bx) = -3a - 4b$

(b) $[\mathsf{T}^t]_{\gamma^*}^{\beta^*} = \begin{pmatrix} -1 & 1 \\ -2 & 1 \end{pmatrix}$ **(c)** $[\mathsf{T}]_\beta^\gamma = \begin{pmatrix} -1 & -2 \\ 1 & 1 \end{pmatrix}$

SECTION 2.7

1. **(a)** T **(b)** T **(c)** F **(d)** F **(e)** T **(f)** F **(g)** T

2. **(a)** F **(b)** F **(c)** T **(d)** T **(e)** F

3. **(a)** $\{e^{-t}, te^{-t}\}$ **(c)** $\{e^{-t}, te^{-t}, e^{t}, te^{t}\}$ **(e)** $\{e^{-t}, e^{t}\cos 2t, e^{t}\sin 2t\}$

4. **(a)** $\{e^{(1+\sqrt{5})t/2}, e^{(1-\sqrt{5})t/2}\}$ **(c)** $\{1, e^{-4t}, e^{-2t}\}$

CHAPTER 3

SECTION 3.1

1. **(a)** T **(b)** F **(c)** T **(d)** F **(e)** T **(f)** F
(g) T **(h)** F **(i)** T

2. Adding -2 times column 1 to column 2 transforms A into B.

3. **(a)** $\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$ **(c)** $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix}$

SECTION 3.2

1. **(a)** F **(b)** F **(c)** T **(d)** T **(e)** F **(f)** T
(g) T **(h)** T **(i)** T

2. **(a)** 2 **(c)** 2 **(e)** 3 **(g)** 1

4. **(a)** $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$; the rank is 2.

5. **(a)** The rank is 2, and the inverse is $\begin{pmatrix} -1 & 2 \\ 1 & -1 \end{pmatrix}$.

(c) The rank is 2, and so no inverse exists.

(e) The rank is 3, and the inverse is $\begin{pmatrix} \frac{1}{6} & -\frac{1}{3} & \frac{1}{2} \\ \frac{1}{2} & 0 & -\frac{1}{2} \\ -\frac{1}{6} & \frac{1}{3} & \frac{1}{2} \end{pmatrix}$.

(g) The rank is 4, and the inverse is $\begin{pmatrix} -51 & 15 & 7 & 12 \\ 31 & -9 & -4 & -7 \\ -10 & 3 & 1 & 2 \\ -3 & 1 & 1 & 1 \end{pmatrix}$.

6. **(a)** $\mathsf{T}^{-1}(ax^2 + bx + c) = -ax^2 - (4a + b)x - (10a + 2b + c)$

(c) $\mathsf{T}^{-1}(a, b, c) = \left(\frac{1}{6}a - \frac{1}{3}b + \frac{1}{2}c, \frac{1}{2}a - \frac{1}{2}c, -\frac{1}{6} + \frac{1}{3}b + \frac{1}{2}c\right)$

(e) $\mathsf{T}^{-1}(a, b, c) = \left(\frac{1}{2}a - b + \frac{1}{2}c\right)x^2 + \left(-\frac{1}{2}a + \frac{1}{2}c\right)x + b$

7. $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

20. **(a)** $\begin{pmatrix} 1 & 3 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$

SECTION 3.3

1. **(a)** F **(b)** F **(c)** T **(d)** F **(e)** F **(f)** F **(g)** T **(h)** F

2. **(a)** $\left\{\begin{pmatrix} -3 \\ 1 \end{pmatrix}\right\}$ **(c)** $\left\{\begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix}\right\}$

(e) $\left\{\begin{pmatrix} -2 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \\ 0 \\ 1 \end{pmatrix}\right\}$ **(g)** $\left\{\begin{pmatrix} -3 \\ 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \\ 0 \\ 1 \end{pmatrix}\right\}$

3. **(a)** $\left\{\begin{pmatrix} 5 \\ 0 \end{pmatrix} + t\begin{pmatrix} -3 \\ 1 \end{pmatrix} : t \in R\right\}$ **(c)** $\left\{\begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix} + t\begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix} : t \in R\right\}$

(e) $\left\{\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + r\begin{pmatrix} -2 \\ 1 \\ 0 \\ 0 \end{pmatrix} + s\begin{pmatrix} 3 \\ 0 \\ 1 \\ 0 \end{pmatrix} + t\begin{pmatrix} -1 \\ 0 \\ 0 \\ 1 \end{pmatrix} : r, s, t \in R\right\}$

(g) $\left\{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} + r\begin{pmatrix} -3 \\ 1 \\ 1 \\ 0 \end{pmatrix} + s\begin{pmatrix} 1 \\ -1 \\ 0 \\ 1 \end{pmatrix} : r, s \in R\right\}$

4. **(b)** (1) $A^{-1} = \begin{pmatrix} \frac{1}{3} & 0 & \frac{1}{3} \\ \frac{1}{9} & \frac{1}{3} & -\frac{2}{9} \\ -\frac{4}{9} & \frac{2}{3} & -\frac{1}{9} \end{pmatrix}$ (2) $\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \\ -2 \end{pmatrix}$

6. $\mathsf{T}^{-1}\{(1, 11)\} = \left\{\begin{pmatrix} \frac{11}{2} \\ -\frac{9}{2} \\ 0 \end{pmatrix} + t\begin{pmatrix} 1 \\ -1 \\ 2 \end{pmatrix} : t \in R\right\}$

7. The systems in parts (b), (c), and (d) have solutions.

11. The farmer, tailor, and carpenter must have incomes in the proportions $4 : 3 : 4$.

13. There must be 7.8 units of the first commodity and 9.5 units of the second.

SECTION 3.4

1. **(a)** F **(b)** T **(c)** T **(d)** T **(e)** F **(f)** T **(g)** T

2. **(a)** $\begin{pmatrix} 4 \\ -3 \\ -1 \end{pmatrix}$ **(c)** $\begin{pmatrix} 2 \\ 3 \\ -2 \\ -1 \end{pmatrix}$ **(e)** $\left\{ \begin{pmatrix} 4 \\ 0 \\ 1 \\ 0 \end{pmatrix} + r \begin{pmatrix} 4 \\ 1 \\ 0 \\ 0 \end{pmatrix} + s \begin{pmatrix} 1 \\ 0 \\ 2 \\ 1 \end{pmatrix} : r, s \in R \right\}$

(g) $\left\{ \begin{pmatrix} -23 \\ 0 \\ 7 \\ 9 \\ 0 \end{pmatrix} + r \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + s \begin{pmatrix} -23 \\ 0 \\ 6 \\ 9 \\ 1 \end{pmatrix} : r, s \in R \right\}$

(i) $\left\{ \begin{pmatrix} 2 \\ 0 \\ 0 \\ -1 \\ 0 \end{pmatrix} + r \begin{pmatrix} 0 \\ 2 \\ 1 \\ 0 \\ 0 \end{pmatrix} + s \begin{pmatrix} 1 \\ -4 \\ 0 \\ -2 \\ 1 \end{pmatrix} : r, s \in R \right\}$

4. **(a)** $\left\{ \begin{pmatrix} \frac{4}{3} \\ \frac{1}{3} \\ 0 \\ 0 \end{pmatrix} + t \begin{pmatrix} 1 \\ -1 \\ 1 \\ 2 \end{pmatrix} : t \in R \right\}$ **(c)** There are no solutions.

5. $\begin{pmatrix} 1 & 0 & 2 & 1 & 4 \\ -1 & -1 & 3 & -2 & -7 \\ 3 & 1 & 1 & 0 & -9 \end{pmatrix}$

7. $\{u_1, u_2, u_5\}$

11. **(b)** $\{(1,2,1,0,0), (2,1,0,0,0), (1,0,0,1,0), (-2,0,0,0,1)\}$

13. **(b)** $\{(1,0,1,1,1,0), (0,2,1,1,0,0), (1,1,1,0,0,0), (-3,-2,0,0,0,1)\}$

CHAPTER 4

SECTION 4.1

1. **(a)** F **(b)** T **(c)** F **(d)** F **(e)** T

2. **(a)** 30 **(c)** -8

3. **(a)** $-10 + 15i$ **(c)** -24

4. **(a)** 19 **(c)** 14

SECTION 4.2

1. **(a)** F **(b)** T **(c)** T **(d)** T **(e)** F **(f)** F **(g)** F **(h)** T

3. 42 **5.** -12 **7.** -12 **9.** 22 **11.** -3

13. -8 **15.** 0 **17.** -49 **19.** $-28 - i$ **21.** 95

SECTION 4.3

1. **(a)** F **(b)** T **(c)** F **(d)** T **(e)** F **(f)** T **(g)** F **(h)** F

3. $(4, -3, 0)$ **5.** $(-20, -48, -8)$ **7.** $(0, -12, 16)$

24. $t^n + a_{n-1}t^{n-1} + \cdots + a_1 t + a_0$

26. **(a)** $\begin{pmatrix} A_{22} & -A_{12} \\ -A_{21} & A_{11} \end{pmatrix}$ **(c)** $\begin{pmatrix} 10 & 0 & 0 \\ 0 & -20 & 0 \\ 0 & 0 & -8 \end{pmatrix}$

(e) $\begin{pmatrix} -3i & 0 & 0 \\ 4 & -1+i & 0 \\ 10+16i & -5-3i & 3+3i \end{pmatrix}$ **(g)** $\begin{pmatrix} 18 & 28 & -6 \\ -20 & -21 & 37 \\ 48 & 14 & -16 \end{pmatrix}$

SECTION 4.4

1. **(a)** T **(b)** T **(c)** T **(d)** F **(e)** F **(f)** T
(g) T **(h)** F **(i)** T **(j)** T **(k)** T

2. **(a)** 22 **(c)** $2 - 4i$

3. **(a)** -12 **(c)** -12 **(e)** 22 **(g)** -3

4. **(a)** 0 **(c)** -49 **(e)** $-28 - i$ **(g)** 95

SECTION 4.5

1. **(a)** F **(b)** T **(c)** T **(d)** F **(e)** F **(f)** T

3. No **5.** Yes **7.** Yes **9.** No

CHAPTER 5

SECTION 5.1

1. **(a)** F **(b)** T **(c)** T **(d)** F **(e)** F **(f)** F
(g) F **(h)** T **(i)** T **(j)** F **(k)** F

2. **(a)** $11,\ t^2 - 5t + 11$ **(c)** $-2,\ t^4 - 2t^3 + t^2 - 2$

3. **(a)** $[\mathsf{T}]_\beta = \begin{pmatrix} 0 & 2 \\ -1 & 0 \end{pmatrix}$, no **(c)** $[\mathsf{T}]_\beta = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$, yes

(e) $[\mathsf{T}]_\beta = \begin{pmatrix} -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$, no

4. **(a)** The eigenvalues are 4 and -1, a basis of eigenvectors is

$$\left\{ \begin{pmatrix} 2 \\ 3 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right\}, \quad Q = \begin{pmatrix} 2 & 1 \\ 3 & -1 \end{pmatrix}, \quad \text{and} \quad D = \begin{pmatrix} 4 & 0 \\ 0 & -1 \end{pmatrix}.$$

(c) The eigenvalues are 1 and -1, a basis of eigenvectors is

$$\left\{ \begin{pmatrix} 1 \\ 1-i \end{pmatrix}, \begin{pmatrix} 1 \\ -1-i \end{pmatrix} \right\}, \quad Q = \begin{pmatrix} 1 & 1 \\ 1-i & -1-i \end{pmatrix}, \quad \text{and} \quad D = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

5. **(a)** $\lambda = 3, 4$ $\quad \beta = \{(3,5),(1,2)\}$
(b) $\lambda = -1, 1, 2$ $\quad \beta = \{(1,2,0),(1,-1,-1),(2,0,-1)\}$
(f) $\lambda = 1, 3$ $\quad \beta = \{-2+x, -4+x^2, -8+x^3, x\}$
(h) $\lambda = -1,1,1,1$ $\quad \beta = \left\{\begin{pmatrix}-1 & 0\\ 0 & 1\end{pmatrix}, \begin{pmatrix}0 & 1\\ 0 & 0\end{pmatrix}, \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, \begin{pmatrix}0 & 0\\ 1 & 0\end{pmatrix}\right\}$
(i) $\lambda = 1,1,-1,-1$ $\quad \beta = \left\{\begin{pmatrix}1 & 0\\ 1 & 0\end{pmatrix}, \begin{pmatrix}0 & 1\\ 0 & 1\end{pmatrix}, \begin{pmatrix}-1 & 0\\ 1 & 0\end{pmatrix}, \begin{pmatrix}0 & -1\\ 0 & 1\end{pmatrix}\right\}$
(j) $\lambda = -1,1,5$ $\quad \beta = \left\{\begin{pmatrix}0 & 1\\ -1 & 0\end{pmatrix}, \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}, \begin{pmatrix}1 & 0\\ 1 & 0\end{pmatrix}, \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}\right\}$

25. 4

SECTION 5.2

1. **(a)** F **(b)** F **(c)** F **(d)** T **(e)** T **(f)** F
(g) T **(h)** T **(i)** F

2. **(a)** Not diagonalizable **(c)** $Q = \begin{pmatrix}1 & 4\\ 1 & -3\end{pmatrix}$

(e) Not diagonalizable **(g)** $Q = \begin{pmatrix}1 & 1 & 1\\ 2 & -1 & 0\\ -1 & 0 & -1\end{pmatrix}$

3. **(a)** Not diagonalizable **(c)** Not diagonalizable

(d) $\beta = \{x - x^2, 1 - x - x^2, x + x^2\}$ **(e)** $\beta = \{(1,1),(1,-1)\}$

7. $A^n = \frac{1}{3}\begin{pmatrix}5^n + 2(-1)^n & 2(5^n) - 2(-1)^n\\ 5^n - (-1)^n & 2(5^n) + (-1)^n\end{pmatrix}$

15. **(b)** $x(t) = c_1 e^{3t}\begin{pmatrix}-2\\ 1\end{pmatrix} + c_2 e^{-2t}\begin{pmatrix}1\\ -1\end{pmatrix}$

(c) $x(t) = e^t\left[c_1\begin{pmatrix}1\\ 0\\ 0\end{pmatrix} + c_2\begin{pmatrix}0\\ 1\\ 0\end{pmatrix}\right] + c_3 e^{2t}\begin{pmatrix}1\\ 1\\ 1\end{pmatrix}$

SECTION 5.3

1. **(a)** T **(b)** T **(c)** F **(d)** F **(e)** T **(f)** T
(g) T **(h)** F **(i)** F **(j)** T

2. **(a)** $\begin{pmatrix}0 & 0\\ 0 & 0\end{pmatrix}$ **(c)** $\begin{pmatrix}\frac{7}{13} & \frac{7}{13}\\ \frac{6}{13} & \frac{6}{13}\end{pmatrix}$ **(e)** No limit exists.

(g) $\begin{pmatrix}-1 & 0 & -1\\ -4 & 1 & -2\\ 2 & 0 & 2\end{pmatrix}$ **(i)** No limit exists.

6. For those who arrived during the week of June 1, after one week 25% of the patients recovered, 20% were ambulatory, 41% were bedridden, and 14% died. Eventually $\frac{59}{90}$ recover and $\frac{31}{90}$ die.

7. $\frac{3}{7}$.

8. Only the matrices in (a) and (b) are regular transition matrices.

9. **(a)** $\begin{pmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{pmatrix}$ **(c)** No limit exists.

(e) $\begin{pmatrix} 0 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{2} & 0 & 1 \end{pmatrix}$ **(g)** $\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 1 \end{pmatrix}$

10. **(a)** $\begin{pmatrix} 0.225 \\ 0.441 \\ 0.334 \end{pmatrix}$ after two stages and $\begin{pmatrix} 0.20 \\ 0.60 \\ 0.20 \end{pmatrix}$ eventually

(c) $\begin{pmatrix} 0.372 \\ 0.225 \\ 0.403 \end{pmatrix}$ after two stages and $\begin{pmatrix} 0.50 \\ 0.20 \\ 0.30 \end{pmatrix}$ eventually

(e) $\begin{pmatrix} 0.329 \\ 0.334 \\ 0.337 \end{pmatrix}$ after two stages and $\begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{pmatrix}$ eventually

12. $\frac{9}{19}$ new, $\frac{6}{19}$ once-used, and $\frac{4}{19}$ twice-used

13. In 1995, 24% will own large cars, 34% will own intermediate-sized cars, and 42% will own small cars; the corresponding eventual percentages are 10%, 30%, and 60%.

20. $e^O = I$ and $e^I = eI$.

SECTION 5.4

1. **(a)** F **(b)** T **(c)** F **(d)** F **(e)** T **(f)** T **(g)** T

2. The subspaces in (a), (c), and (d) are T-invariant.

6. **(a)** $\left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \\ 2 \\ 2 \end{pmatrix} \right\}$ **(c)** $\left\{ \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right\}$

9. **(a)** $-t(t^2 - 3t + 3)$ **(c)** $1 - t$

10. **(a)** $t(t-1)(t^2 - 3t + 3)$ **(c)** $(t-1)^3(t+1)$

18. **(c)** $A^{-1} = \frac{1}{2} \begin{pmatrix} 2 & -2 & -4 \\ 0 & 1 & 3 \\ 0 & 0 & -2 \end{pmatrix}$

31. **(a)** $t^2 - 6t + 6$ **(c)** $-(t+1)(t^2 - 6t + 6)$

CHAPTER 6

SECTION 6.1

1. **(a)** T **(b)** T **(c)** F **(d)** F **(e)** F **(f)** F **(g)** F **(h)** T

2. $\langle x, y\rangle = 8 + 5i$, $\|x\| = \sqrt{7}$, $\|y\| = \sqrt{14}$, and $\|x + y\| = \sqrt{37}$.

3. $\langle f, g\rangle = 1$, $\|f\| = \frac{\sqrt{3}}{3}$, $\|g\| = \sqrt{\dfrac{e^2 - 1}{2}}$, and $\|f + g\| = \sqrt{\dfrac{11 + 3e^2}{6}}$.

16. **(b)** No

SECTION 6.2

1. **(a)** F **(b)** T **(c)** T **(d)** F **(e)** T **(f)** F **(g)** T

2. For each part the orthonormal basis and the Fourier coefficients are given.

(b) $\left\{\frac{\sqrt{3}}{3}(1,1,1), \frac{\sqrt{6}}{6}(-2,1,1), \frac{\sqrt{2}}{2}(0,-1,1)\right\}$; $\frac{2\sqrt{3}}{3}, -\frac{\sqrt{6}}{6}, \frac{\sqrt{2}}{2}$.

(c) $\{1, 2\sqrt{3}(x - \frac{1}{2}), 6\sqrt{5}(x^2 - x + \frac{1}{6})\}$; $\frac{3}{2}, \frac{\sqrt{3}}{6}, 0$.

(e) $\left\{\frac{1}{5}(2,-1,-2,4), \frac{1}{\sqrt{30}}(-4,2,-3,1), \frac{1}{\sqrt{155}}(-3,4,9,7)\right\}$; $10, 3\sqrt{30}, \sqrt{155}$

(g) $\left\{\dfrac{1}{6}\begin{pmatrix} 3 & 5 \\ -1 & 1 \end{pmatrix}, \dfrac{1}{6\sqrt{2}}\begin{pmatrix} -4 & 4 \\ 6 & -2 \end{pmatrix}, \dfrac{1}{9\sqrt{2}}\begin{pmatrix} 9 & -3 \\ 6 & -6 \end{pmatrix}\right\}$; $24, 6\sqrt{2}, -9\sqrt{2}$

(i) $\left\{\sqrt{\frac{2}{\pi}}\sin t, \sqrt{\frac{2}{\pi}}\cos t, \sqrt{\frac{\pi}{\pi^2-8}}(1 - \frac{4}{\pi}\sin t), \sqrt{\frac{12\pi}{\pi^4-96}}(t + \frac{4}{\pi}\cos t - \frac{\pi}{2})\right\}$;

$\sqrt{\frac{2}{\pi}}(2\pi + 2), -4\sqrt{\frac{2}{\pi}}, \sqrt{\frac{\pi^2-8}{\pi}}(1 + \pi), \sqrt{\frac{\pi^4-96}{3\pi}}$

(k) $\left\{\frac{1}{\sqrt{47}}(-4, 3-2i, i, 1-4i), \frac{1}{\sqrt{60}}(3-i, -5i, -2+4i, 2+i),\right.$

$\left.\frac{1}{\sqrt{1160}}(-17-i, -9+8i, -18+16i, -9+8i)\right\}$;

$\sqrt{47}(-1-i), \sqrt{60}(-1+2i), \sqrt{1160}(1+i)$

(m) $\left\{\dfrac{1}{\sqrt{18}}\begin{pmatrix} -1+i & -i \\ 2-i & 1+3i \end{pmatrix}, \dfrac{1}{\sqrt{246}}\begin{pmatrix} -4i & -11-9i \\ 1+5i & 1-i \end{pmatrix},\right.$

$\left.\dfrac{1}{\sqrt{39063}}\begin{pmatrix} -5-118i & -7-26i \\ -145i & -58 \end{pmatrix}\right\}$; $\sqrt{18}(2+i), \sqrt{246}(-1-i), 0$

4. $S^\perp = \text{span}(\{(i, -\frac{1}{2}(1+i), 1)\})$

5. $S_0^\perp$ is the plane through the origin that is perpendicular to x_0; $S^\perp$ is the line through the origin that is perpendicular to the plane containing x_1 and x_2.

19. **(a)** $\dfrac{1}{17}\begin{pmatrix} 26 \\ 104 \end{pmatrix}$ **(b)** $\dfrac{1}{14}\begin{pmatrix} 29 \\ 17 \\ 40 \end{pmatrix}$

20. **(b)** $\dfrac{1}{\sqrt{14}}$

SECTION 6.3

1. **(a)** T **(b)** F **(c)** F **(d)** T **(e)** F **(f)** T **(g)** T

2. **(a)** $y = (1, -2, 4)$ **(c)** $y = 210x^2 - 204x + 33$

3. **(a)** $\mathsf{T}^*(x) = (11, -12)$ **(c)** $\mathsf{T}^*(f(t)) = 12 + 6t$

14. $\mathsf{T}^*(x) = \langle x, z\rangle\, y$

20. **(a)** The linear function is $y = -2t + 5/2$ with $E = 1$, and the quadratic function is $y = t^2/3 - 4t/3 + 2$ with $E = 0$.

(b) The linear function is $y = 1.25t + 0.55$ with $E = 0.3$, and the quadratic function is $t^2/56 + 15t/14 + 239/280$ with $E = 0.22857$ (approximation).

21. The spring constant is approximately 2.1.

22. **(a)** $x = \frac{2}{7}$, $y = \frac{3}{7}$, $z = \frac{1}{7}$ **(d)** $x = \frac{7}{12}$, $y = \frac{1}{12}$, $z = \frac{1}{4}$, $w = -\frac{1}{12}$

SECTION 6.4

1. **(a)** T **(b)** F **(c)** F **(d)** T **(e)** T **(f)** T **(g)** F **(h)** T

2. **(a)** T is self-adjoint. An orthonormal basis of eigenvectors is

$$\left\{\frac{1}{\sqrt{5}}(1, -2),\ \frac{1}{\sqrt{5}}(2, 1)\right\}, \text{ with corresponding eigenvalues 6 and 1.}$$

(c) T is normal, but not self-adjoint. An orthonormal basis of eigenvectors is

$$\left\{\frac{1}{2}(1+i, \sqrt{2}), \frac{1}{2}(1+i, -\sqrt{2})\right\} \text{ with corresponding eigenvalues}$$

$$2 + \frac{1+i}{\sqrt{2}} \text{ and } 2 - \frac{1+i}{\sqrt{2}}.$$

(e) T is self-adjoint. An orthonormal basis of eigenvectors is

$$\left\{\frac{1}{\sqrt{2}}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}, \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, \frac{1}{\sqrt{2}}\begin{pmatrix}0 & -1\\ 1 & 0\end{pmatrix}, \frac{1}{\sqrt{2}}\begin{pmatrix}-1 & 0\\ 0 & 1\end{pmatrix}\right\}$$

with corresponding eigenvalues 1, 1, −1, 1.

SECTION 6.5

1. **(a)** T **(b)** F **(c)** F **(d)** T **(e)** F **(f)** T
(g) F **(h)** F **(i)** F

2. **(a)** $P = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$ and $D = \begin{pmatrix}3 & 0\\ 0 & -1\end{pmatrix}$

(d) $P = \begin{pmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{3}}\\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{3}}\\ 0 & -\frac{2}{\sqrt{6}} & \frac{1}{\sqrt{3}}\end{pmatrix}$ and $D = \begin{pmatrix}-2 & 0 & 0\\ 0 & -2 & 0\\ 0 & 0 & 4\end{pmatrix}$

4. T_z is normal for all $z \in C$, T_z is self-adjoint if and only if $z \in R$, and T_z is unitary if and only if $|z| = 1$.

5. Only the pair of matrices in (d) are unitarily equivalent.

25. $2(\psi - \phi)$

26. **(a)** $\psi - \dfrac{\phi}{2}$ **(b)** $\psi + \dfrac{\phi}{2}$

27. **(a)** $x = \dfrac{1}{\sqrt{2}}x' + \dfrac{1}{\sqrt{2}}y'$ and $y = \dfrac{1}{\sqrt{2}}x' - \dfrac{1}{\sqrt{2}}y'$

The new quadratic form is $3(x')^2 - (y')^2$.

(c) $x = \dfrac{3}{\sqrt{13}}x' + \dfrac{2}{\sqrt{13}}y'$ and $y = \dfrac{-2}{\sqrt{13}}x' + \dfrac{2}{\sqrt{13}}y'$

The new quadratic form is $5(x')^2 - 8(y')^2$.

29. **(c)** $Q = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{3}} & -\frac{\sqrt{6}}{6} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{3}} & \frac{\sqrt{6}}{6} \\ 0 & \frac{1}{\sqrt{3}} & \frac{\sqrt{6}}{3} \end{pmatrix}$ and $R = \begin{pmatrix} \sqrt{2} & \sqrt{2} & 2\sqrt{2} \\ 0 & \sqrt{3} & \frac{\sqrt{3}}{3} \\ 0 & 0 & \frac{\sqrt{6}}{3} \end{pmatrix}$

(e) $x_1 = 3$, $x_2 = -5$, $x_3 = 4$

SECTION 6.6

1. **(a)** F **(b)** T **(c)** T **(d)** F **(e)** F

2. For $\mathsf{W} = \text{span}(\{(1,2)\})$, $[\mathsf{T}]_\beta = \begin{pmatrix} \frac{1}{5} & \frac{2}{5} \\ \frac{2}{5} & \frac{4}{5} \end{pmatrix}$.

3. (2) **(a)** $\mathsf{T}_1(a,b) = \frac{1}{2}(a+b, a+b)$ and $\mathsf{T}_2(a,b) = \frac{1}{2}(a-b, -a+b)$

(d) $\mathsf{T}_1(a,b,c) = \frac{1}{3}(2a-b-c, -a+2b-c, -a-b+2c)$ and

$\mathsf{T}_2(a,b,c) = \frac{1}{3}(a+b+c, a+b+c, a+b+c)$

SECTION 6.7

1. **(a)** F **(b)** F **(c)** T **(d)** T **(e)** F **(f)** F **(g)** T

2. **(a)** $v_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $v_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, $u_1 = \dfrac{1}{\sqrt{3}}\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$, $u_2 = \dfrac{1}{\sqrt{2}}\begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}$, $u_3 = \dfrac{1}{\sqrt{6}}\begin{pmatrix} 2 \\ -1 \\ -1 \end{pmatrix}$

$\sigma_1 = \sqrt{3}$, $\sigma_2 = \sqrt{2}$

(c) $v_1 = \dfrac{1}{\sqrt{\pi}}\sin x$, $v_2 = \dfrac{1}{\sqrt{\pi}}\cos x$, $v_3 = \dfrac{1}{\sqrt{2\pi}}$

$u_1 = \dfrac{\cos x + 2\sin x}{\sqrt{5\pi}}$, $u_2 = \dfrac{2\cos x - \sin x}{\sqrt{5\pi}}$, $u_3 = \dfrac{1}{\sqrt{2\pi}}$,

$\sigma_1 = \sqrt{5}$, $\sigma_2 = \sqrt{5}$, $\sigma_3 = 2$

3. **(a)** $\begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} \\ -\frac{1}{\sqrt{3}} & 0 & \frac{2}{\sqrt{6}} \end{pmatrix} \begin{pmatrix} \sqrt{6} & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}^*$

(c) $\begin{pmatrix} \frac{2}{\sqrt{10}} & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{10}} \\ \frac{1}{\sqrt{10}} & -\frac{1}{\sqrt{2}} & 0 & -\frac{2}{\sqrt{10}} \\ \frac{1}{\sqrt{10}} & \frac{1}{\sqrt{2}} & 0 & -\frac{2}{\sqrt{10}} \\ \frac{2}{\sqrt{10}} & 0 & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{10}} \end{pmatrix} \begin{pmatrix} \sqrt{5} & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}^*$

(e) $\begin{pmatrix} \frac{1+i}{2} & \frac{1+i}{2} \\ \frac{1-i}{2} & \frac{-1+i}{2} \end{pmatrix} \begin{pmatrix} \sqrt{6} & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \frac{2}{\sqrt{6}} & \frac{1-i}{\sqrt{6}} \\ \frac{1+i}{\sqrt{6}} & -\frac{2}{\sqrt{6}} \end{pmatrix}^*$

4. (a) $WP = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \frac{\sqrt{8}+\sqrt{2}}{2} & \frac{-\sqrt{8}+\sqrt{2}}{2} \\ \frac{-\sqrt{8}+\sqrt{2}}{2} & \frac{\sqrt{8}+\sqrt{2}}{2} \end{pmatrix}$

5. (a) $\mathsf{T}^\dagger(x, y, z) = \left(\dfrac{x+y+z}{3}, \dfrac{y-z}{2}\right)$

(c) $\mathsf{T}^\dagger(a + b\sin x + c\cos x) = \mathsf{T}^{-1}(a + b\sin x + c\cos x) =$

$$\frac{a}{2} + \frac{(2b+c)\sin x + (-b+2c)\cos x}{5}$$

6. (a) $\frac{1}{6}\begin{pmatrix} 1 & 1 & -1 \\ 1 & 1 & -1 \end{pmatrix}$ (c) $\frac{1}{5}\begin{pmatrix} 1 & -2 & 3 & 1 \\ 1 & 3 & -2 & 1 \end{pmatrix}$ (e) $\frac{1}{6}\begin{pmatrix} 1-i & 1+i \\ 1 & i \end{pmatrix}$

7. (a) $\mathsf{Z}_1 = \mathsf{N}(\mathsf{T})^\perp = \mathsf{R}^2$ and $\mathsf{Z}_2 = \mathsf{R}(\mathsf{T}) = \text{span}\{(1,1,1), (0,1,-1)\}$
(c) $\mathsf{Z}_1 = \mathsf{N}(\mathsf{T})^\perp = \mathsf{V}$ and $\mathsf{Z}_2 = \mathsf{R}(\mathsf{T}) = \mathsf{V}$

8. (a) No solution $\frac{1}{2}\begin{pmatrix} 1 \\ 1 \end{pmatrix}$

SECTION 6.8

1. (a) F (b) F (c) T (d) F (e) T (f) F
(g) F (h) F (i) T (j) F

4. (a) Yes (b) No (c) No (d) Yes (e) Yes (f) No

5. (a) $\begin{pmatrix} 0 & 2 & -2 \\ 2 & 0 & -2 \\ 1 & 1 & 0 \end{pmatrix}$ (b) $\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$ (c) $\begin{pmatrix} 0 & 0 & 0 & 0 \\ -1 & 0 & -4 & 0 \\ 0 & 0 & 0 & 0 \\ -2 & 0 & -8 & 0 \end{pmatrix}$

17. (a) and (b) $\left\{ \begin{pmatrix} \frac{2}{\sqrt{5}} \\ -\frac{1}{\sqrt{5}} \end{pmatrix}, \begin{pmatrix} \frac{1}{\sqrt{5}} \\ \frac{2}{\sqrt{5}} \end{pmatrix} \right\}$ (c) $\left\{ \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ -\frac{1}{\sqrt{2}} \end{pmatrix} \right\}$

18. Same as Exercise 17(c)

22. (a) $Q = \begin{pmatrix} 1 & -3 \\ 0 & 1 \end{pmatrix}$ and $D = \begin{pmatrix} 1 & 0 \\ 0 & -7 \end{pmatrix}$

(b) $Q = \begin{pmatrix} 1 & -\frac{1}{2} \\ 1 & \frac{1}{2} \end{pmatrix}$ and $D = \begin{pmatrix} 2 & 0 \\ 0 & -\frac{1}{2} \end{pmatrix}$

(c) $Q = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -0.25 \\ 1 & 0 & 2 \end{pmatrix}$ and $D = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 6.75 \end{pmatrix}$

SECTION 6.9

5. $(B_v)^{-1} = \begin{pmatrix} \frac{1}{\sqrt{1-v^2}} & 0 & \frac{v}{\sqrt{1-v^2}} \\ 0 & 1 & 0 \\ \frac{v}{\sqrt{1-v^2}} & 0 & \frac{1}{\sqrt{1-v^2}} \end{pmatrix}$

SECTION 6.10

1. **(a)** F **(b)** F **(c)** T **(d)** F **(e)** F

2. **(a)** $\sqrt{18}$ **(c)** approximately 2.34

4. **(a)** $\|A\| \approx 84.74$, $\|A^{-1}\| \approx 17.01$, and $\text{cond}(A) \approx 1441$
(b) $\|\tilde{x} - A^{-1}b\| \le \|A^{-1}\| \cdot \|A\tilde{x} - b\| \approx 0.17$ and
$\frac{\|\tilde{x} - A^{-1}b\|}{\|A^{-1}b\|} \le \text{cond}(A)\frac{\|b - A\tilde{x}\|}{\|b\|} \approx \frac{14.41}{\|b\|}$

5. $0.001 \le \frac{\|x - \tilde{x}\|}{\|x\|} \le 10$

6. $R\begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix} = \frac{9}{7}$, $\|B\|_E = 2$, and $\text{cond}(B) = 2$.

SECTION 6.11

1. **(a)** T **(b)** T **(c)** T **(d)** F **(e)** F **(f)** T
(g) T **(h)** F **(i)** T **(j)** F

3. **(b)** $\left\{t\begin{pmatrix} \sqrt{3} \\ 1 \end{pmatrix} : t \in R\right\}$

4. **(b)** $\left\{t\begin{pmatrix} 1 \\ 0 \end{pmatrix} : t \in R\right\}$ if $\phi = 0$ and $\left\{t\begin{pmatrix} \cos\phi + 1 \\ \sin\phi \end{pmatrix} : t \in R\right\}$ if $\phi \ne 0$

CHAPTER 7

SECTION 7.1

1. **(a)** T **(b)** F **(c)** F **(d)** T **(e)** F **(f)** F **(g)** T **(h)** T

2. **(a)** For $\lambda = 2$, $\left\{\begin{pmatrix} -1 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}\right\}$ $J = \begin{pmatrix} 2 & 1 \\ 0 & 2 \end{pmatrix}$

(c) For $\lambda = -1$, $\left\{\begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix}\right\}$ For $\lambda = 2$, $\left\{\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}\right\}$ $J = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix}$

3. **(a)** For $\lambda = 2$, $\{2, -2x, x^2\}$ $J = \begin{pmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix}$

(c) For $\lambda = 1$, $\left\{\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}\right\}$ $J = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

SECTION 7.2

1. **(a)** T **(b)** T **(c)** F **(d)** T **(e)** T **(f)** F **(g)** F **(h)** T

2. $J = \begin{pmatrix} A_1 & O & O \\ O & A_2 & O \\ O & O & A_3 \end{pmatrix}$ where $A_1 = \begin{pmatrix} 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$,

$A_2 = \begin{pmatrix} 4 & 1 & 0 & 0 \\ 0 & 4 & 1 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$ and $A_3 = \begin{pmatrix} -3 & 0 \\ 0 & -3 \end{pmatrix}$

3. **(a)** $-(t-2)^5(t-3)^2$ **(b)**

(c) $\lambda_2 = 3$ **(d)** $p_1 = 3$ and $p_2 = 1$
(e) (i) $\text{rank}(\mathsf{U}_1) = 3$ and $\text{rank}(\mathsf{U}_2) = 0$
(ii) $\text{rank}(\mathsf{U}_1^2) = 1$ and $\text{rank}(\mathsf{U}_2^2) = 0$
(iii) $\text{nullity}(\mathsf{U}_1) = 2$ and $\text{nullity}(\mathsf{U}_2) = 2$
(iv) $\text{nullity}(\mathsf{U}_1^2) = 4$ and $\text{nullity}(\mathsf{U}_2^2) = 2$

4. **(a)** $J = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix}$ and $Q = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 1 & -1 & 0 \end{pmatrix}$

(d) $J = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$ and $Q = \begin{pmatrix} 1 & 0 & 1 & -1 \\ 1 & -1 & 0 & 1 \\ 1 & -2 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$

5. **(a)** $J = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$ and $\beta = \{2e^t, 2te^t, t^2e^t, e^{2t}\}$

(c) $J = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 2 \end{pmatrix}$ and $\beta = \{6x, x^3, 2, x^2\}$

(d) $J = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$ and

$$\beta = \left\{ \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & -1 \\ 0 & 2 \end{pmatrix}, \begin{pmatrix} 1 & -2 \\ 2 & 0 \end{pmatrix} \right\}$$

24. **(a)** $\begin{pmatrix} x \\ y \\ z \end{pmatrix} = e^{2t}\left[(c_1 + c_2 t)\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + c_2\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}\right] + c_3 e^{3t}\begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}$

(b) $\begin{pmatrix} x \\ y \\ z \end{pmatrix} = e^{2t}\left[(c_1 + c_2 t + c_3 t^2)\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + (c_2 + 2c_3 t)\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + 2c_3\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}\right]$

SECTION 7.3

1. **(a)** F **(b)** T **(c)** F **(d)** F **(e)** T **(f)** F **(g)** F **(h)** T **(i)** T

2. **(a)** $(t-1)(t-3)$ **(c)** $(t-1)^2(t-2)$ **(d)** $(t-2)^2$

3. **(a)** $t^2 - 2$ **(c)** $(t-2)^2$ **(d)** $(t-1)(t+1)$

4. For (2), (a); for (3), (a) and (d)

5. The operators are T_0, I, and all operators having both 0 and 1 as eigenvalues.

SECTION 7.4

1. **(a)** T **(b)** F **(c)** F **(d)** T **(e)** T **(f)** F **(g)** T

2. **(a)** $\begin{pmatrix} 0 & 0 & 27 \\ 1 & 0 & -27 \\ 0 & 1 & 9 \end{pmatrix}$ **(b)** $\begin{pmatrix} 0 & -1 \\ 1 & -1 \end{pmatrix}$

(c) $\begin{pmatrix} \frac{1}{2}(-1+i\sqrt{3}) & 0 \\ 0 & \frac{1}{2}(-1-i\sqrt{3}) \end{pmatrix}$ **(e)** $\begin{pmatrix} 0 & -2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -3 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

3. **(a)** $t^2 + 1$ and t^2 $C = \begin{pmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$; $\beta = \{1, x, -2x + x^2, -3x + x^3\}$

(c) $t^2 - t + 1$ $C = \begin{pmatrix} 0 & -1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$

$$\beta = \left\{\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & -1 \end{pmatrix}\right\}$$

Index

Absolute value of a complex number, 558
Absorbing Markov chain, 303
Absorbing state, 303
Addition
  of matrices, 9
Addition of vectors, 6
Additive function, 78
Additive inverse
  of an element of a field, 553
  of a vector, 12
Adjoint
  of a linear operator, 355–358
  of a linear transformation, 364
  of a matrix, 329, 356–358
  uniqueness, 355
Algebraic multiplicity of an eigenvalue, *see* Multiplicity of an eigenvalue
Algebraically closed field, 476, 562
Alternating n-linear function, 239
Angle between two vectors, 202, 333
Annihilator
  of a subset, 127
  of a vector, 518, 522
Approximation property of an orthogonal projection, 396
Area of a parallelogram, 203
Associated quadratic form, 386
Augmented matrix, 160, 174
Auxiliary polynomial, 132, 134, 138–140
Axiom of Choice, 60

Back substitution, 186
Backward pass, 186
Basis, 43–50, 61–62, 191–193
  cyclic, 520
  dual, 121
  Jordan canonical, 477
  ordered, 79
  orthonormal, 339, 344–345, 369
  rational canonical, 520
  standard basis for F^n, 43
  standard basis for $\mathsf{P}_n(F)$, 44
  standard ordered basis for F^n, 79
  standard ordered basis for $\mathsf{P}_n(F)$, 79
  uniqueness of size, 47
Bessel's inequality, 353
Bilinear form, 419–430
  diagonalizable, 425
  diagonalization, 425–432
  index, 441
  invariants, 441
  matrix representation, 421–425
  product with a scalar, 420
  rank, 439
  signature, 441
  sum, 420
  symmetric, 425–427, 430–432
  vector space, 421

Cancellation law for vector addition, 12
Cancellation laws for a field, 554
Canonical form
  Jordan, 477–510
  rational, 520–542
  for a symmetric matrix, 443
Cauchy–Schwarz inequality, 331
Cayley–Hamilton theorem
  for a linear operator, 315
  for a matrix, 316, 374
Chain of sets, 60
Change of coordinate matrix, 112–116
Characteristic of a field, 23, 42, 426, 446, 556
Characteristic polynomial, 370
  of a linear operator, 249
  of a matrix, 248

Characteristic value, *see* Eigenvalue
Characteristic vector, *see* Eigenvector
Classical adjoint
  of an $n \times n$ matrix, 231
  of a 2×2 matrix, 208
Clique, 95, 99
Closed model of a simple economy, 175–177
Closure
  under addition, 17
  under scalar multiplication, 17
Codomain, 551
Coefficient matrix of a system of linear equations, 168
Coefficients
  Fourier, 119
  of a differential equation, 129
  of a linear combination, 25, 44
  of a polynomial, 10
Cofactor, 209, 232
Cofactor expansion, 210, 215, 232
Column of a matrix, 9
Column operation, 148
Column sum of matrices, 294
Column vector, 8
Companion matrix, 520
Complex number, 556–562
  absolute value, 558
  conjugate, 558
  fundamental theorem of algebra, 476, 561
  imaginary part, 556
  real part, 556
Composition
  of functions, 552
  of linear transformations, 87–90
Condition number, 463
Conditioning of a system of linear equations, 459
Congruent matrices, 423, 442, 448
Conic sections, 385–389
Conjugate linear property, 331
Conjugate of a complex number, 558
Conjugate transpose of a matrix, 329, 356–358
Consistent system of linear equations, 169
Consumption matrix, 176
Convergence of matrices, 283–286
Coordinate function, 120–121
Coordinate system
  left-handed, 202
  right-handed, 202
Coordinate vector, 80, 92, 111–112
Corresponding homogeneous system of linear equations, 172
Coset, 24
Cramer's rule, 224
Critical point, 436
Cullen, Charles G., 464
Cycle of generalized eigenvectors, 483–485
  end vector, 483
  initial vector, 483
  length, 483
Cyclic basis, 520
Cyclic subspace, 312–315

Degree of a polynomial, 10
Determinant, 199–244
  area of a parallelogram, 203
  characterization of, 242
  cofactor expansion, 210, 215, 232
  Cramer's rule, 224
  of an identity matrix, 212
  of an invertible matrix, 223
  of a linear operator, 249, 468, 470
  of a matrix transpose, 224
  of an $n \times n$ matrix, 209, 232
  n-dimensional volume, 226
  properties of, 234–236
  of a square matrix, 364, 391
  of a 2×2 matrix, 200
  uniqueness of, 242
  of an upper triangular matrix, 218
  volume of a parallelepiped, 226
  Wronskian, 232
Diagonal entries of a matrix, 8

Diagonal matrix, 19, 98
Diagonalizable bilinear form, 425
Diagonalizable linear operator, 246
Diagonalizable matrix, 246
Diagonalization
  of a bilinear form, 425–432
  problem, 245
  simultaneous, 281, 323, 325, 373, 402
  of a symmetric matrix, 427–430
  test, 268, 490
Diagonalize, 247
Differentiable function, 130
Differential equation, 129
  auxiliary polynomial, 132, 134, 138–140
  coefficients, 129
  homogeneous, 129, 138–140, 517
  linear, 129
  nonhomogeneous, 142
  order, 129
  solution, 129
  solution space, 132, 138–140
  system, 272, 510
Differential operator, 131
  null space, 134–137
  order, 131, 135
Dimension, 47–52, 104, 120, 422
Dimension theorem, 70
Direct sum
  of matrices, 318–319, 491, 539
  of subspaces, 22–23, 58–59, 99, 274–277, 316, 352, 363, 391, 395, 398, 469–471, 488, 539
Disjoint sets, 550
Distance, 338
Division algorithm for polynomials, 562
Domain, 551
Dominance relation, 96, 100
Dot diagram
  of a Jordan canonical form, 492–495
  of a rational canonical form, 529–533
Double dual, 120, 124
Dual basis, 121
Dual space, 120–124

Economics, *see* Leontief, Wassily
Eigenspace
  generalized, 478–485
  of a linear operator or matrix, 264
Eigenvalue
  of a generalized eigenvector, 478
  of a linear operator or matrix, 246, 368–371, 461–464
  multiplicity, 263
Eigenvector
  generalized, 478–485
  of a linear operator or matrix, 246, 368–371
Einstein, Albert, *see* Special theory of relativity
Element, 549
Elementary column operation, 148, 153
Elementary divisor
  of a linear operator, 533
  of a matrix, 535
Elementary matrix, 149–150, 158
Elementary operation, 148
Elementary row operation, 148, 153, 217
Ellipse, *see* Conic sections
Empty set, 549
End vector of a cycle of generalized eigenvectors, 483
Entry of a matrix, 8
Equality
  of functions, 10, 551
  of matrices, 9
  of n-tuples, 8
  of polynomials, 10
  of sets, 549
Equilibrium condition for a simple economy, 176
Equivalence relation, 108, 551
  congruence, 446, 448
  unitary equivalence, 391, 466
Equivalent systems of linear equations, 181–182
Euclidean norm of a matrix, 461–464

Euler's formula, 133
Even function, 15, 22, 353
Exponential function, 133–140
Exponential of a matrix, 310, 509
Extremum, *see* Local extremum

Field, 553–556
  algebraically closed, 476, 562
  cancellation laws, 554
  characteristic, 23, 42, 426, 446, 556
  of complex numbers, 556–562
  product of elements, 553
  of real numbers, 549
  sum of elements, 553
Field of scalars, 6–7, 48
Finite-dimensional vector space, 47–52
Fixed probability vector, 300
Forward pass, 186
Fourier, Jean Baptiste, 345
Fourier coefficients, 119, 345, 397
Frobenius inner product, 330
Function, 551–553
  additive, 78
  alternating n-linear, 239
  codomain of, 551
  composite, 552
  coordinate function, 120–121
  differentiable, 130
  domain of, 551
  equality of, 10, 551
  even, 15, 22, 353
  exponential, 133–140
  image of, 551
  imaginary part of, 130
  inverse, 552
  invertible, 552
  linear, *see* Linear transformation
  n-linear, 238–242
  norm, 338
  odd, 22, 353
  one-to-one, 552
  onto, 552
  polynomial, 10, 52–54, 571
  preimage of, 551
  range of, 551
  real part of, 130
  restriction of, 552
  sum of, 10
  vector space, 10
Fundamental theorem of algebra, 476, 561

Gaussian elimination, 186
  back substitution, 186
  backward pass, 186
  forward pass, 186
General solution of a system of linear equations, 188
Generalized eigenspace, 478–485
Generalized eigenvector, 478–485
Generates, 31
Generator of a cyclic subspace, 312
Geometry, 382–389, 433, 466–471
Gershgorin's circle theorem, 295
Gram–Schmidt process, 342, 393
Gramian matrix, 373

Hardy–Weinberg law, 305
Hermitian operator or matrix, *see* Self-adjoint linear operator or matrix
Hessian matrix, 436
Homogeneous linear differential equation, 129, 138–140, 517
Homogeneous polynomial of degree two, 430
Homogeneous system of linear equations, 170
Hooke's law, 129, 365
Householder operator, 394

Identity element
  in C, 557
  in a field, 553, 554
Identity matrix, 82, 93, 212
Identity transformation, 67
Ill-conditioned system, 459
Image, *see* Range
Image of an element, 551
Imaginary number, 557
Imaginary part
  of a complex number, 556

of a function, 130
Incidence matrix, 94–96, 99
Inconsistent system of linear equations, 169
Index
of a bilinear form, 441
of a matrix, 442
Index set, 550
Infinite-dimensional vector space, 47
Initial probability vector, 291
Initial vector of a cycle of generalized eigenvectors, 483
Inner product, 327–334
Frobenius, 330
on H, 333
standard, 328
Inner product space
complex, 330
H, 330, 341, 345–346, 376, 396
real, 330
Input–output matrix, 176
Intersection of sets, 550
Invariant subspace, 77–78, 311–319
Invariants
of a bilinear form, 441
of a matrix, 442
Inverse
of a function, 552
of a linear transformation, 100–103, 164
of a matrix, 101–103, 107, 160–164
Invertible function, 552
Invertible linear transformation, 100–103
Invertible matrix, 101–103, 112, 223, 463
Irreducible polynomial, 519, 568–571
Isomorphic vector spaces, 103–106
Isomorphism, 103–106, 124, 422

Jordan block, 477
Jordan canonical basis, 477
Jordan canonical form
dot diagram, 492–495
of a linear operator, 477–510
of a matrix, 486
uniqueness, 495

Kernel, *see* Null space
Kronecker delta, 82, 333

Lagrange interpolation formula, 52–54, 126, 399
Lagrange polynomials, 52, 109, 126
Least squares approximation, 358–361
Least squares line, 358
Left shift operator, 76
Left-handed coordinate system, 202
Left-multiplication transformation, 93–94
Legendre polynomials, 344
Length of a cycle of generalized eigenvectors, 483
Length of a vector, *see* Norm
Leontief
closed model, 175–177
open model, 177–178
Leontief, Wassily, 175
Limit of a sequence of matrices, 283–286
Linear combination, 25–27, 29–30, 40
uniqueness of coefficients, 44
Linear dependence, 37–41
Linear differential equation, 129
Linear equations, *see* System of linear equations
Linear functional, 119
Linear independence, 38–41, 60–62, 340
Linear operator, (*see also* Linear transformation), 113
adjoint, 355–358
characteristic polynomial, 249
determinant, 249, 468, 470
diagonalizable, 246
diagonalize, 247
differential, 131
differentiation, 131, 134–137
eigenspace, 264, 398

eigenvalue, 246, 368–371
eigenvector, 246, 368–371
elementary divisor, 533
Householder operator, 394
invariant subspace, 77–78, 311–319
Jordan canonical form, 477–510
left shift, 76
minimal polynomial, 510–515
nilpotent, 506
normal, 367, 398–400
orthogonal, 376–382, 466–471
partial isometry, 391, 402
positive definite, 374–375
positive semidefinite, 374–375
projection, 395–400
projection on a subspace, 86, 118
projection on the x-axis, 66
rational canonical form, 520–542
reflection, 66, 114, 118, 384, 466–471
right shift, 76
rotation, 66, 379, 384, 466–471
self-adjoint, 370, 398–400
simultaneous diagonalization, 281, 402
spectral decomposition, 399
spectrum, 399
unitary, 376–382, 399
Linear space, *see* Vector space
Linear transformation, (*see also* Linear operator), 65
adjoint, 364
composition, 87–90
identity, 67
image, *see* Range
inverse, 100–103, 164
invertible, 100–103
isomorphism, 103–106, 124, 422
kernel, *see* Null space
left-multiplication, 93–94
linear functional, 119
matrix representation, 80, 89–93, 345, 356
null space, 67–69, 134–137
nullity, 70–71
one-to-one, 71
onto, 71
product with a scalar, 82
pseudoinverse, 410
range, 67–69
rank, 70–71, 159
restriction, 77–78
singular value, 404
singular value theorem, 403
sum, 82
transpose, 121, 126, 128
vector space of, 83, 104
zero, 67
Local extremum, 436, 447
Local maximum, 436, 447
Local minimum, 436, 447
Lower triangular matrix, 229

Markov chain, 290, 303
Markov process, 290
Matrix, 8
addition, 9
adjoint, 329, 356–358
augmented, 160, 174
change of coordinate, 112–116
characteristic polynomial, 248
classical adjoint, 208, 231
coefficient, 168
cofactor, 209, 232
column of, 9
column sum, 294
companion, 520
condition number, 463
congruent, 423, 442, 448
conjugate transpose, 329, 356–358
consumption, 176
convergence, 283–286
determinant of, 200, 209, 232, 364, 391
diagonal, 19, 98
diagonal entries of, 8
diagonalizable, 246
diagonalize, 247
direct sum, 318–319, 491, 539
eigenspace, 264
eigenvalue, 246, 461–464

eigenvector, 246
elementary, 149–150, 158
elementary divisor, 535
elementary operations, 148
entry, 8
equality of, 9
Euclidean norm, 461–464
exponential of, 310, 509
Gramian, 373
Hessian, 436
identity, 82
incidence, 94–96, 99
index, 442
input–output, 176
invariants, 442
inverse, 101–103, 107, 160–164
invertible, 101–103, 112, 223, 463
Jordan block, 477
Jordan canonical form, 486
limit of, 283–286
lower triangular, 229
minimal polynomial, 511–515
multiplication with a scalar, 9
nilpotent, 229, 506
norm, 338, 461–464, 509
normal, 367
orthogonal, 229, 379–382
orthogonally equivalent, 381–382
polar decomposition, 408–410
positive definite, 374
positive semidefinite, 374
product, 88–94
product with a scalar, 9
pseudoinverse, 411
rank, 152–159
rational canonical form, 535
reduced row echelon form, 185, 189–190
regular, 293
representation of a bilinear form, 421–425
representation of a linear transformation, 80, 89–93, 345, 356
row of, 8
row sum, 294
scalar, 259
self-adjoint, 370, 461
signature, 442
similarity, 116, 118, 259, 502
simultaneous diagonalization, 281
singular value, 406
singular value decomposition, 407
skew-symmetric, 23, 229, 368
square, 9
stochastic, *see* Transition matrix
submatrix, 230
sum, 9
symmetric, 18, 370, 381, 386, 443
trace, 19, 21, 98, 118, 279, 329, 390
transition, 287–290, 509
transpose, 18, 21, 66, 89, 127, 224, 259, 280
transpose of a matrix inverse, 107
transpose of a product, 89
unitary, 229, 379–382
unitary equivalence, 381–382, 391, 466
upper triangular, 19, 218, 259, 367, 382, 394
Vandermonde, 230
vector space, 9, 329, 422
zero, 9
Maximal element of a family of sets, 59
Maximal linearly independent subset, 60–62
Maximal principle, 60
Member, *see* Element
Michelson–Morley experiment, 448
Minimal polynomial
of a linear operator, 510–515
of a matrix, 511–515
uniqueness, 510
Minimal solution to a system of linear equations, 361–362
Monic polynomial, 568–571
Multiplicative inverse of an element of a field, 553

Multiplicity of an eigenvalue, 263
Multiplicity of an elementary divisor, 533, 535

n-dimensional volume, 226
n-linear function, 238–242
n-tuple, 8
  equality, 8
  scalar multiplication, 8
  sum, 8
  vector space, 8
Nilpotent linear operator, 506
Nilpotent matrix, 229, 506
Nonhomogeneous linear differential equation, 142
Nonhomogeneous system of linear equations, 170
Nonnegative vector, 176
Norm
  Euclidean, 461–464
  of a function, 338
  of a matrix, 338, 461–464, 509
  of a vector, 331–339
Normal equations, 365
Normal linear operator or matrix, 367, 398–400
Normalizing a vector, 333
Null space, 67–69, 134–137
Nullity, 70–71
Numerical methods
  conditioning, 459
  QR factorization, 393–394

Odd function, 22, 353
One-to-one function, 552
One-to-one linear transformation, 71
Onto function, 552
Onto linear transformation, 71
Open model of a simple economy, 177–178
Order
  of a differential equation, 129
  of a differential operator, 131, 135
Ordered basis, 79
Orientation of an ordered basis, 202
Orthogonal complement, 347, 349, 395–398
Orthogonal equivalence of matrices, 381–382
Orthogonal matrix, 229, 379–382
Orthogonal operator, 376–382, 466–471
  on R^2, 384–385
Orthogonal projection, 395–400
Orthogonal projection of a vector, 348
Orthogonal subset, 333, 339
Orthogonal vectors, 333
Orthonormal basis, 339, 344–345, 369
Orthonormal subset, 333

Parallel vectors, 3
Parallelogram
  area of, 203
  law, 2, 335
Parseval's identity, 352
Partial isometry, 391, 402
Pendular motion, 143
Penrose conditions, 418
Periodic motion of a spring, 128, 144
Perpendicular vectors, *see* Orthogonal vectors
Physics
  Hooke's law, 129, 365
  pendular motion, 143
  periodic motion of a spring, 144
  special theory of relativity, 448–456
  spring constant, 365
Polar decomposition of a matrix, 408–410
Polar identities, 336
Polynomial, 10
  annihilator of a vector, 518, 522
  auxiliary, 132, 134, 138–140
  characteristic, 370
  coefficients of, 10
  degree of a, 10
  division algorithm, 562
  equality, 10

function, 10, 52–54, 571
fundamental theorem of algebra, 476, 561
homogeneous of degree two, 430
irreducible, 519, 568–571
Lagrange, 52, 109, 126
Legendre, 344
minimal, 510–515
monic, 568–571
product with a scalar, 11
quotient, 563
relatively prime, 564
remainder, 563
splits, 263, 367, 370
sum, 11
trigonometric, 396
unique factorization theorem, 569
vector space, 11
zero, 10
zero of a, 62, 135, 561, 564
Positive definite matrix, 374
Positive definite operator, 374–375
Positive semidefinite matrix, 374
Positive semidefinite operator, 374–375
Positive vector, 176
Power set, 59
Preimage of an element, 551
Primary decomposition theorem, 539
Principal axis theorem, 387
Probability, *see* Markov chain
Probability vector, 288
fixed, 300
initial, 291
Product
of a bilinear form and a scalar, 420
of complex numbers, 556
of elements of a field, 553
of a linear transformation and scalar, 82
of matrices, 88–94
of a matrix and a scalar, 9
of a vector and a scalar, 7
Projection
on a subspace, 76, 86, 99, 118, 395–400
on the x-axis, 66
orthogonal, 395–400
Proper subset, 549
Proper value, *see* Eigenvalue
Proper vector, *see* Eigenvector
Pseudoinverse
of a linear transformation, 410
of a matrix, 411
Pythagorean theorem, 335

QR factorization, 393–394
Quadratic form, 386, 430–436
Quotient of polynomials, 563
Quotient space, 24, 59, 79, 110, 323

Range, 67–69, 551
Rank
of a bilinear form, 439
of a linear transformation, 70–71, 159
of a matrix, 152–159
Rational canonical basis, 520
Rational canonical form
dot diagram, 529–533
elementary divisor, 533, 535
of a linear operator, 520–542
of a matrix, 535
uniqueness, 533
Rayleigh quotient, 461
Real part
of a complex number, 556
of a function, 130
Reduced row echelon form of a matrix, 185, 189–190
Reflection, 66, 114, 118, 466–471
of R^2, 114, 379–380, 384, 385
Regular transition matrix, 293
Relation on a set, 551
Relative change in a vector, 460
Relatively prime polynomials, 564
Remainder, 563
Replacement theorem, 46–47
Representation of a linear transformation by a matrix, 80
Resolution of the identity operator, 399

Restriction
of a function, 552
of a linear operator on a subspace, 77–78
Right shift operator, 76
Right-handed coordinate system, 202
Rigid motion, 382–384
in the plane, 385
Rotation, 66, 379, 384, 466–471
Row of a matrix, 8
Row operation, 148
Row sum of matrices, 294
Row vector, 8
Rudin, Walter, 561

Saddle point, 437
Scalar, 7
Scalar matrix, 259
Scalar multiplication, 6
Schur's theorem
for a linear operator, 367
for a matrix, 382
Second derivative test, 436–439, 446
Self-adjoint linear operator or matrix, 370, 398–400, 461
Sequence, 11
Set, 549–551
chain, 60
disjoint, 550
element of a, 549
empty, 549
equality of, 549
equivalence relation, 108, 391, 446, 448
equivalence relation on a, 551
indexed, 550
intersection, 550
linearly dependent, 37–41
linearly independent, 38–41
orthogonal, 333, 339
orthonormal, 333
power, 59
proper subset, 549
relation on a, 551
subset, 549
union, 550
Signature
of a bilinear form, 441
of a matrix, 442
Similar matrices, 116, 118, 259, 502
Simpson's rule, 126
Simultaneous diagonalization, 281, 323, 325, 373, 402
Singular value
of a linear transformation, 404
of a matrix, 406
Singular value decomposition of a matrix, 407
Singular value decomposition theorem for matrices, 406
Singular value theorem for linear transformations, 403
Skew-symmetric matrix, 23, 229, 368
Solution
of a differential equation, 129
minimal, 361–362
to a system of linear equations, 169
Solution set of a system of linear equations, 169, 181
Solution space of a homogeneous differential equation, 132, 138–140
Space–time coordinates, 449
Span, 30–31, 35, 341
Special theory of relativity, 448–456
space–time coordinates, 449
time contraction, 455–456
Spectral decomposition, 399
Spectral theorem, 398
Spectrum, 399
Splits, 263, 367, 370
Spring, periodic motion of, 128, 144
Spring constant, 365
Square matrix, 9
Square root of a unitary operator, 390
Standard basis

for F^n, 43
for $\mathsf{P}_n(F)$, 44
Standard inner product on F^n, 328
Standard ordered basis
for F^n, 79
for $\mathsf{P}_n(F)$, 79
Standard representation of a vector space, 105–106
States
absorbing, 303
of a transition matrix, 287
Stationary vector, *see* Fixed probability vector
Statistics, *see* Least squares approximation
Stochastic matrix, *see* Transition matrix
Stochastic process, 290
Submatrix, 230
Subset, 549
linearly dependent, 37–41
linearly independent, 60–62
maximal linearly independent, 60–62
orthogonal, 333, 339
orthogonal complement of a, 347, 349, 395–398
orthonormal, 333
span of a, 30–31, 35, 341
sum, 22
Subspace, 17–20, 50–52
cyclic, 312–315
dimension of a, 50–52
direct sum, 22–23, 58–59, 99, 274–277, 316, 352, 363, 391, 395, 398, 469–471, 488, 539
generated by a set, 31
invariant, 77–78
sum, 274
zero, 17
Sum
of bilinear forms, 420
of complex numbers, 556
of elements of a field, 553
of functions, 10
of linear transformations, 82
of matrices, 9
of n-tuples, 8
of polynomials, 11
of subsets, 22
of vectors, 7
Sum of subspaces, (*see also* Direct sum, of subspaces), 274
Sylvester's law of inertia
for a bilinear form, 440
for a matrix, 442
Symmetric bilinear form, 425–427, 430–432
Symmetric matrix, 18, 370, 381, 386, 443
System of differential equations, 272, 510
System of linear equations, 26–30, 168
augmented matrix, 174
coefficient matrix, 168
consistent, 169
corresponding homogeneous system, 172
equivalent, 181–182
Gaussian elimination, 186
general solution, 188
homogeneous, 170
ill-conditioned, 459
inconsistent, 169
minimal solution, 361–362
nonhomogeneous, 170
solution to, 169
well-conditioned, 459

T-annihilator, 518, 522
T-cyclic basis, 520
T-cyclic subspace, 312–315
T-invariant subspace, 77–78, 311–319
Taylor's theorem, 437
Test for diagonalizability, 490
Time contraction, 455–456
Trace of a matrix, 19, 21, 98, 118, 279, 329, 390
Transition matrix, 287–290, 509
regular, 293
states, 287
Translation, 383

Transpose
of an invertible matrix, 107
of a linear transformation, 121, 126, 128
of a matrix, 18, 21, 66, 89, 127, 224, 259, 280
Trapezoidal rule, 126
Triangle inequality, 331
Trigonometric polynomial, 396
Trivial representation of zero vector, 37–38

Union of sets, 550
Unique factorization theorem for polynomials, 569
Uniqueness
of adjoint, 355
of coefficients of a linear combination, 44
of Jordan canonical form, 495
of minimal polynomial, 510
of rational canonical form, 533
of size of a basis, 47
Unit vector, 333
Unitary equivalence of matrices, 381–382, 391, 466
Unitary matrix, 229, 379–382
Unitary operator, 376–382, 399
Upper triangular matrix, 19, 218, 259, 367, 382, 394

Vandermonde matrix, 230
Vector, 7
additive inverse of a, 12
annihilator of a, 518, 522
column, 8
coordinate, 80, 92, 111–112
fixed probability, 300
Fourier coefficients, 119, 345, 397
initial probability, 291
linear combination, 25
nonnegative, 176
norm, 331–339
normalizing, 333
orthogonal, 333
orthogonal projection of a, 348
parallel, 3
perpendicular, *see* Orthogonal vectors
positive, 176
probability, 288
product with a scalar, 8
Rayleigh quotient, 461
row, 8
sum, 7
unit, 333
zero, 12, 37–38
Vector space, 6
addition, 6
basis, 43–50, 191–193
of bilinear forms, 421
of continuous functions, 18, 67, 119, 329, 343, 353
of cosets, 24, 59, 110, 323
dimension, 47–49, 104, 120, 422
dual, 120–124
finite-dimensional, 47–52
of functions from a set into a field, 10, 110, 127
infinite-dimensional, 47
of infinitely differentiable functions, 131–137, 248, 517
isomorphism, 103–106, 124, 422
of linear transformations, 83, 104
of matrices, 9, 104, 329, 422
of n-tuples, 8
of polynomials, 11, 87, 109
quotient, 24, 59, 79, 110, 323
scalar multiplication, 6
of sequences, 11, 109, 354, 366
subspace, 17–20, 50–52
zero, 15
zero vector of a, 12
Volume of a parallelepiped, 226

Wade, William R., 436
Well-conditioned system, 459
Wilkinson, J. H., 394
Wronskian, 232

Z_2, 16, 42, 426, 554
Zero matrix, 9
Zero of a polynomial, 62, 135, 561, 564
Zero polynomial, 10

Zero subspace, 17
Zero transformation, 67
Zero vector, 12, 37–38
  trivial representation, 37–38
Zero vector space, 15

List of Symbols

A_{ij}	the ij-th entry of the matrix A	page 9
A^{-1}	the inverse of the matrix A	page 101
$A^{\dagger}$	the pseudoinverse of the matrix A	page 411
A^*	the adjoint of the matrix A	page 329
$\tilde{A}_{ij}$	the matrix A with row i and column j deleted	page 209
A^k	the kth power of a square matrix A	page 91
A^t	the transpose of the matrix A	page 18
$(A\|B)$	the matrix A augmented by the matrix B	page 160
$B_1 \oplus \cdots \oplus B_k$	the direct sum of matrices B_1 through B_k	page 318
$\mathcal{B}(\mathsf{V})$	the set of bilinear forms on V	page 419
β^*	the dual basis of β	page 121
β_x	the T-cyclic basis generated by x	page 519
C	the field of complex numbers	page 7
$\mathsf{C}^n(R)$	set of functions f on R with $f^{(n)}$ continuous	page 22
C^∞	set of functions with derivatives of every order	page 131
$\mathsf{C}(R)$	the vector space of continuous functions on R	page 18
$\mathsf{C}([0,1])$	the vector space of continuous functions on $[0,1]$	page 329
C_x	the T-cyclic subspace generated by x	page 518
D	the derivative operator on C^∞	page 131
$\det(A)$	the determinant of the matrix A	page 232
$\det(\mathsf{T})$	the determinant of the linear operator T	page 249
δ_{ij}	the Kronecker delta	page 82
$\dim(\mathsf{V})$	the dimension of V	page 47
e^A	$\lim_{m\to\infty}\left(I + A + \frac{A^2}{2!} + \cdots + \frac{A^m}{m!}\right)$	page 310
e_i	the ith standard vector of F^n	page 43
E^{ij}	the matrix whose only nonzero entry is $E_{ij} = 1$	page 44
E_λ	the eigenspace of T corresponding to λ	page 264

F	a field	page 6
f	a linear functional	page 119
$f(A)$	the polynomial $f(x)$ evaluated at the matrix A	page 560
$f(x)$	a polynomial	page 10
F^n	the set of n-tuples with entries in a field F	page 8
$f(\mathsf{T})$	the polynomial $f(x)$ evaluated at the operator T	page 560
$\mathcal{F}(S, F)$	the set of functions from S to a field F	page 9
H	space of continuous complex functions on $[0, 2\pi]$	page 330
I_n or I	the $n \times n$ identity matrix	page 82
I_V or I	the identity operator on V	page 67
K_λ	generalized eigenspace of T corresponding to λ	page 477
K_ϕ	$\{x\colon (\phi(\mathsf{T}))^p(x) = \mathit{0}$ for some positive integer $p\}$	page 518
L_A	left-multiplication transformation by matrix A	page 93
$\lim\limits_{m\to\infty} A_m$	the limit of a sequence of matrices	page 283
$\mathcal{L}(\mathsf{V})$	the space of linear transformations from V to V	page 83
$\mathcal{L}(\mathsf{V}, \mathsf{W})$	the space of linear transformations from V to W	page 83
$\mathsf{M}_{m\times n}(F)$	the set of $m \times n$ matrices with entries in F	page 9
$\nu(A)$	the column sum of the matrix A	page 294
$\nu_j(A)$	the jth column sum of the matrix A	page 294
$\mathsf{N}(\mathsf{T})$	the null space of T	page 67
nullity(T)	the dimension of the null space of T	page 70
O	the zero matrix	page 9
$\mathsf{P}(F)$	the space of polynomials with coefficients in F	page 11
$\mathsf{P}_n(F)$	the polynomials in $\mathsf{P}(F)$ of degree at most n	page 18
ϕ_β	standard representation with respect to basis β	page 105
$\psi_\beta(H)$	matrix representation of a bilinear form H	page 421
R	the field of real numbers	page 7
rank(A)	the rank of the matrix A	page 152
rank(T)	the rank of the linear transformation T	page 70

CONTINUED ON REAR ENDPAPERS

List of Symbols (continued)

$\rho(A)$	the row sum of the matrix A	page 294
$\rho_i(A)$	the ith row sum of the matrix A	page 294
$\mathsf{R}(\mathsf{T})$	the range of the linear transformation T	page 67
$S_1 + S_2$	the sum of sets S_1 and S_2	page 22
$\text{span}(S)$	the span of the set S	page 30
$S^\perp$	the orthogonal complement of the set S	page 347
$[\mathsf{T}]_\beta$	the matrix representation of T in basis β	page 80
$[\mathsf{T}]_\beta^\gamma$	the matrix representation of T in bases β and γ	page 80
T^{-1}	the inverse of the linear transformation T	page 100
$\mathsf{T}^\dagger$	the pseudoinverse of the linear transformation T	page 410
T^*	the adjoint of the linear operator T	page 355
T^k	the kth iterate of a linear operator T	page 88
T_0	the zero transformation	page 67
T^t	the transpose of the linear transformation T	page 121
T_θ	the rotation transformation by θ	page 66
T_W	the restriction of T to a subspace W	page 312
$\text{tr}(A)$	the trace of the matrix A	page 19
V^*	the dual space of the vector space V	page 120
V/W	the quotient space of V modulo W	page 24
$\mathsf{W}_1 + \cdots + \mathsf{W}_k$	the sum of subspaces W_1 through W_k	page 274
$\sum_{i=1}^k \mathsf{W}_i$	the sum of subspaces W_1 through W_k	page 274
$\mathsf{W}_1 \oplus \mathsf{W}_2$	the direct sum of subspaces W_1 and W_2	page 22
$\mathsf{W}_1 \oplus \cdots \oplus \mathsf{W}_k$	the direct sum of subspaces W_1 through W_k	page 274
$\|x\|$	the norm of the vector x	page 331
$[x]_\beta$	the coordinate vector of x relative to β	page 80
$\langle x, y\rangle$	the inner product of x and y	page 328
Z_2	the field consisting of 0 and 1	page 547
$\overline{z}$	the complex conjugate of z	page 551
$\mathit{0}$	the zero vector	page 7

THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN

Over
1 MILLION
copies sold
worldwide

INTRODUCTION TO ALGORITHMS

FOURTH EDITION

Introduction to Algorithms

Fourth Edition

Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein

Introduction to Algorithms

Fourth Edition

The MIT Press
Cambridge, Massachusetts London, England

© 2022 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

The MIT Press would like to thank the anonymous peer reviewers who provided comments on drafts of this book. The generous work of academic experts is essential for establishing the authority and quality of our publications. We acknowledge with gratitude the contributions of these otherwise uncredited readers.

This book was set in Times Roman and MathTime Professional II by the authors.

Names: Cormen, Thomas H., author. | Leiserson, Charles Eric, author. | Rivest, Ronald L., author. | Stein, Clifford, author.
Title: Introduction to algorithms / Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.
Description: Fourth edition. | Cambridge, Massachusetts : The MIT Press, [2022] | Includes bibliographical references and index.
Identifiers: LCCN 2021037260 | ISBN 9780262046305
Subjects: LCSH: Computer programming. | Computer algorithms.
Classification: LCC QA76.6 .C662 2022 | DDC 005.13--dc23
LC record available at http://lccn.loc.gov/2021037260

10 9 8 7 6 5 4 3 2 1

Contents

Preface *xiii*

I Foundations

Introduction *3*

1 The Role of Algorithms in Computing *5*
1.1 Algorithms *5*
1.2 Algorithms as a technology *12*

2 Getting Started *17*
2.1 Insertion sort *17*
2.2 Analyzing algorithms *25*
2.3 Designing algorithms *34*

3 Characterizing Running Times *49*
3.1 O-notation, Ω-notation, and Θ-notation *50*
3.2 Asymptotic notation: formal definitions *53*
3.3 Standard notations and common functions *63*

4 Divide-and-Conquer *76*
4.1 Multiplying square matrices *80*
4.2 Strassen's algorithm for matrix multiplication *85*
4.3 The substitution method for solving recurrences *90*
4.4 The recursion-tree method for solving recurrences *95*
4.5 The master method for solving recurrences *101*
★ 4.6 Proof of the continuous master theorem *107*
★ 4.7 Akra-Bazzi recurrences *115*

5 Probabilistic Analysis and Randomized Algorithms *126*
5.1 The hiring problem *126*
5.2 Indicator random variables *130*
5.3 Randomized algorithms *134*
★ 5.4 Probabilistic analysis and further uses of indicator random variables *140*

II Sorting and Order Statistics

Introduction *157*

6 Heapsort *161*
6.1 Heaps *161*
6.2 Maintaining the heap property *164*
6.3 Building a heap *167*
6.4 The heapsort algorithm *170*
6.5 Priority queues *172*

7 Quicksort *182*
7.1 Description of quicksort *183*
7.2 Performance of quicksort *187*
7.3 A randomized version of quicksort *191*
7.4 Analysis of quicksort *193*

8 Sorting in Linear Time *205*
8.1 Lower bounds for sorting *205*
8.2 Counting sort *208*
8.3 Radix sort *211*
8.4 Bucket sort *215*

9 Medians and Order Statistics *227*
9.1 Minimum and maximum *228*
9.2 Selection in expected linear time *230*
9.3 Selection in worst-case linear time *236*

III Data Structures

Introduction *249*

10 Elementary Data Structures *252*
10.1 Simple array-based data structures: arrays, matrices, stacks, queues *252*
10.2 Linked lists *258*
10.3 Representing rooted trees *265*

11 Hash Tables *272*
11.1 Direct-address tables *273*
11.2 Hash tables *275*
11.3 Hash functions *282*
11.4 Open addressing *293*
11.5 Practical considerations *301*

12 Binary Search Trees *312*
12.1 What is a binary search tree? *312*
12.2 Querying a binary search tree *316*
12.3 Insertion and deletion *321*

13 Red-Black Trees *331*
13.1 Properties of red-black trees *331*
13.2 Rotations *335*
13.3 Insertion *338*
13.4 Deletion *346*

IV Advanced Design and Analysis Techniques

Introduction *361*

14 Dynamic Programming *362*
14.1 Rod cutting *363*
14.2 Matrix-chain multiplication *373*
14.3 Elements of dynamic programming *382*
14.4 Longest common subsequence *393*
14.5 Optimal binary search trees *400*

15 Greedy Algorithms *417*
15.1 An activity-selection problem *418*
15.2 Elements of the greedy strategy *426*
15.3 Huffman codes *431*
15.4 Offline caching *440*

16 Amortized Analysis *448*
16.1 Aggregate analysis *449*
16.2 The accounting method *453*
16.3 The potential method *456*
16.4 Dynamic tables *460*

V Advanced Data Structures

Introduction *477*

17 Augmenting Data Structures *480*
17.1 Dynamic order statistics *480*
17.2 How to augment a data structure *486*
17.3 Interval trees *489*

18 B-Trees *497*
18.1 Definition of B-trees *501*
18.2 Basic operations on B-trees *504*
18.3 Deleting a key from a B-tree *513*

19 Data Structures for Disjoint Sets *520*
19.1 Disjoint-set operations *520*
19.2 Linked-list representation of disjoint sets *523*
19.3 Disjoint-set forests *527*
★ 19.4 Analysis of union by rank with path compression *531*

VI Graph Algorithms

Introduction *547*

20 Elementary Graph Algorithms *549*
20.1 Representations of graphs *549*
20.2 Breadth-first search *554*
20.3 Depth-first search *563*
20.4 Topological sort *573*
20.5 Strongly connected components *576*

21 Minimum Spanning Trees *585*
21.1 Growing a minimum spanning tree *586*
21.2 The algorithms of Kruskal and Prim *591*

22 Single-Source Shortest Paths *604*
22.1 The Bellman-Ford algorithm *612*
22.2 Single-source shortest paths in directed acyclic graphs *616*
22.3 Dijkstra's algorithm *620*
22.4 Difference constraints and shortest paths *626*
22.5 Proofs of shortest-paths properties *633*

23 All-Pairs Shortest Paths *646*
23.1 Shortest paths and matrix multiplication *648*
23.2 The Floyd-Warshall algorithm *655*
23.3 Johnson's algorithm for sparse graphs *662*

24 Maximum Flow *670*
24.1 Flow networks *671*
24.2 The Ford-Fulkerson method *676*
24.3 Maximum bipartite matching *693*

25 Matchings in Bipartite Graphs *704*
25.1 Maximum bipartite matching (revisited) *705*
25.2 The stable-marriage problem *716*
25.3 The Hungarian algorithm for the assignment problem *723*

VII Selected Topics

Introduction *745*

26 Parallel Algorithms *748*
26.1 The basics of fork-join parallelism *750*
26.2 Parallel matrix multiplication *770*
26.3 Parallel merge sort *775*

27 Online Algorithms *791*
27.1 Waiting for an elevator *792*
27.2 Maintaining a search list *795*
27.3 Online caching *802*

28 Matrix Operations *819*
28.1 Solving systems of linear equations *819*
28.2 Inverting matrices *833*
28.3 Symmetric positive-definite matrices and least-squares approximation *838*

29 Linear Programming *850*
29.1 Linear programming formulations and algorithms *853*
29.2 Formulating problems as linear programs *860*
29.3 Duality *866*

30 Polynomials and the FFT *877*
30.1 Representing polynomials *879*
30.2 The DFT and FFT *885*
30.3 FFT circuits *894*

31 Number-Theoretic Algorithms ***903***
31.1 Elementary number-theoretic notions *904*
31.2 Greatest common divisor *911*
31.3 Modular arithmetic *916*
31.4 Solving modular linear equations *924*
31.5 The Chinese remainder theorem *928*
31.6 Powers of an element *932*
31.7 The RSA public-key cryptosystem *936*
★ 31.8 Primality testing *942*

32 String Matching ***957***
32.1 The naive string-matching algorithm *960*
32.2 The Rabin-Karp algorithm *962*
32.3 String matching with finite automata *967*
★ 32.4 The Knuth-Morris-Pratt algorithm *975*
32.5 Suffix arrays *985*

33 Machine-Learning Algorithms ***1003***
33.1 Clustering *1005*
33.2 Multiplicative-weights algorithms *1015*
33.3 Gradient descent *1022*

34 NP-Completeness ***1042***
34.1 Polynomial time *1048*
34.2 Polynomial-time verification *1056*
34.3 NP-completeness and reducibility *1061*
34.4 NP-completeness proofs *1072*
34.5 NP-complete problems *1080*

35 Approximation Algorithms ***1104***
35.1 The vertex-cover problem *1106*
35.2 The traveling-salesperson problem *1109*
35.3 The set-covering problem *1115*
35.4 Randomization and linear programming *1119*
35.5 The subset-sum problem *1124*

VIII Appendix: Mathematical Background

Introduction ***1139***

A Summations ***1140***
A.1 Summation formulas and properties *1140*
A.2 Bounding summations *1145*

B Sets, Etc. ***1153***
B.1 Sets *1153*
B.2 Relations *1158*
B.3 Functions *1161*
B.4 Graphs *1164*
B.5 Trees *1169*

C Counting and Probability ***1178***
C.1 Counting *1178*
C.2 Probability *1184*
C.3 Discrete random variables *1191*
C.4 The geometric and binomial distributions *1196*
★ C.5 The tails of the binomial distribution *1203*

D Matrices ***1214***
D.1 Matrices and matrix operations *1214*
D.2 Basic matrix properties *1219*

Bibliography ***1227***

Index ***1251***

Preface

Not so long ago, anyone who had heard the word "algorithm" was almost certainly a computer scientist or mathematician. With computers having become prevalent in our modern lives, however, the term is no longer esoteric. If you look around your home, you'll find algorithms running in the most mundane places: your microwave oven, your washing machine, and, of course, your computer. You ask algorithms to make recommendations to you: what music you might like or what route to take when driving. Our society, for better or for worse, asks algorithms to suggest sentences for convicted criminals. You even rely on algorithms to keep you alive, or at least not to kill you: the control systems in your car or in medical equipment.[1] The word "algorithm" appears somewhere in the news seemingly every day.

Therefore, it behooves you to understand algorithms not just as a student or practitioner of computer science, but as a citizen of the world. Once you understand algorithms, you can educate others about what algorithms are, how they operate, and what their limitations are.

This book provides a comprehensive introduction to the modern study of computer algorithms. It presents many algorithms and covers them in considerable depth, yet makes their design accessible to all levels of readers. All the analyses are laid out, some simple, some more involved. We have tried to keep explanations clear without sacrificing depth of coverage or mathematical rigor.

Each chapter presents an algorithm, a design technique, an application area, or a related topic. Algorithms are described in English and in a pseudocode designed to be readable by anyone who has done a little programming. The book contains 231 figures—many with multiple parts—illustrating how the algorithms work. Since we emphasize *efficiency* as a design criterion, we include careful analyses of the running times of the algorithms.

---

[1] To understand many of the ways in which algorithms influence our daily lives, see the book by Fry [162].

The text is intended primarily for use in undergraduate or graduate courses in algorithms or data structures. Because it discusses engineering issues in algorithm design, as well as mathematical aspects, it is equally well suited for self-study by technical professionals.

In this, the fourth edition, we have once again updated the entire book. The changes cover a broad spectrum, including new chapters and sections, color illustrations, and what we hope you'll find to be a more engaging writing style.

To the teacher

We have designed this book to be both versatile and complete. You should find it useful for a variety of courses, from an undergraduate course in data structures up through a graduate course in algorithms. Because we have provided considerably more material than can fit in a typical one-term course, you can select the material that best supports the course you wish to teach.

You should find it easy to organize your course around just the chapters you need. We have made chapters relatively self-contained, so that you need not worry about an unexpected and unnecessary dependence of one chapter on another. Whereas in an undergraduate course, you might use only some sections from a chapter, in a graduate course, you might cover the entire chapter.

We have included 931 exercises and 162 problems. Each section ends with exercises, and each chapter ends with problems. The exercises are generally short questions that test basic mastery of the material. Some are simple self-check thought exercises, but many are substantial and suitable as assigned homework. The problems include more elaborate case studies which often introduce new material. They often consist of several parts that lead the student through the steps required to arrive at a solution.

As with the third edition of this book, we have made publicly available solutions to some, but by no means all, of the problems and exercises. You can find these solutions on our website, http://mitpress.mit.edu/algorithms/. You will want to check this site to see whether it contains the solution to an exercise or problem that you plan to assign. Since the set of solutions that we post might grow over time, we recommend that you check the site each time you teach the course.

We have starred ($\star$) the sections and exercises that are more suitable for graduate students than for undergraduates. A starred section is not necessarily more difficult than an unstarred one, but it may require an understanding of more advanced mathematics. Likewise, starred exercises may require an advanced background or more than average creativity.

To the student

We hope that this textbook provides you with an enjoyable introduction to the field of algorithms. We have attempted to make every algorithm accessible and interesting. To help you when you encounter unfamiliar or difficult algorithms, we describe each one in a step-by-step manner. We also provide careful explanations of the mathematics needed to understand the analysis of the algorithms and supporting figures to help you visualize what is going on.

Since this book is large, your class will probably cover only a portion of its material. Although we hope that you will find this book helpful to you as a course textbook now, we have also tried to make it comprehensive enough to warrant space on your future professional bookshelf.

What are the prerequisites for reading this book?

- You need some programming experience. In particular, you should understand recursive procedures and simple data structures, such as arrays and linked lists (although Section 10.2 covers linked lists and a variant that you may find new).
- You should have some facility with mathematical proofs, and especially proofs by mathematical induction. A few portions of the book rely on some knowledge of elementary calculus. Although this book uses mathematics throughout, Part I and Appendices A–D teach you all the mathematical techniques you will need.

Our website, http://mitpress.mit.edu/algorithms/, links to solutions for some of the problems and exercises. Feel free to check your solutions against ours. We ask, however, that you not send your solutions to us.

To the professional

The wide range of topics in this book makes it an excellent handbook on algorithms. Because each chapter is relatively self-contained, you can focus on the topics most relevant to you.

Since most of the algorithms we discuss have great practical utility, we address implementation concerns and other engineering issues. We often provide practical alternatives to the few algorithms that are primarily of theoretical interest.

If you wish to implement any of the algorithms, you should find the translation of our pseudocode into your favorite programming language to be a fairly straightforward task. We have designed the pseudocode to present each algorithm clearly and succinctly. Consequently, we do not address error handling and other software-engineering issues that require specific assumptions about your programming environment. We attempt to present each algorithm simply and directly without allowing the idiosyncrasies of a particular programming language to obscure its essence. If you are used to 0-origin arrays, you might find our frequent practice of

indexing arrays from 1 a minor stumbling block. You can always either subtract 1 from our indices or just overallocate the array and leave position 0 unused.

We understand that if you are using this book outside of a course, then you might be unable to check your solutions to problems and exercises against solutions provided by an instructor. Our website, http://mitpress.mit.edu/algorithms/, links to solutions for some of the problems and exercises so that you can check your work. Please do not send your solutions to us.

To our colleagues

We have supplied an extensive bibliography and pointers to the current literature. Each chapter ends with a set of chapter notes that give historical details and references. The chapter notes do not provide a complete reference to the whole field of algorithms, however. Though it may be hard to believe for a book of this size, space constraints prevented us from including many interesting algorithms.

Despite myriad requests from students for solutions to problems and exercises, we have adopted the policy of not citing references for them, removing the temptation for students to look up a solution rather than to discover it themselves.

Changes for the fourth edition

As we said about the changes for the second and third editions, depending on how you look at it, the book changed either not much or quite a bit. A quick look at the table of contents shows that most of the third-edition chapters and sections appear in the fourth edition. We removed three chapters and several sections, but we have added three new chapters and several new sections apart from these new chapters.

We kept the hybrid organization from the first three editions. Rather than organizing chapters only by problem domains or only according to techniques, this book incorporates elements of both. It contains technique-based chapters on divide-and-conquer, dynamic programming, greedy algorithms, amortized analysis, augmenting data structures, NP-completeness, and approximation algorithms. But it also has entire parts on sorting, on data structures for dynamic sets, and on algorithms for graph problems. We find that although you need to know how to apply techniques for designing and analyzing algorithms, problems seldom announce to you which techniques are most amenable to solving them.

Some of the changes in the fourth edition apply generally across the book, and some are specific to particular chapters or sections. Here is a summary of the most significant general changes:

- We added 140 new exercises and 22 new problems. We also improved many of the old exercises and problems, often as the result of reader feedback. (Thanks to all readers who made suggestions.)

- We have color! With designers from the MIT Press, we selected a limited palette, devised to convey information and to be pleasing to the eye. (We are delighted to display red-black trees in—get this—red and black!) To enhance readability, defined terms, pseudocode comments, and page numbers in the index are in color.
- Pseudocode procedures appear on a tan background to make them easier to spot, and they do not necessarily appear on the page of their first reference. When they don't, the text directs you to the relevant page. In the same vein, nonlocal references to numbered equations, theorems, lemmas, and corollaries include the page number.
- We removed topics that were rarely taught. We dropped in their entirety the chapters on Fibonacci heaps, van Emde Boas trees, and computational geometry. In addition, the following material was excised: the maximum-subarray problem, implementing pointers and objects, perfect hashing, randomly built binary search trees, matroids, push-relabel algorithms for maximum flow, the iterative fast Fourier transform method, the details of the simplex algorithm for linear programming, and integer factorization. You can find all the removed material on our website, http://mitpress.mit.edu/algorithms/.
- We reviewed the entire book and rewrote sentences, paragraphs, and sections to make the writing clearer, more personal, and gender neutral. For example, the "traveling-salesman problem" in the previous editions is now called the "traveling-salesperson problem." We believe that it is critically important for engineering and science, including our own field of computer science, to be welcoming to everyone. (The one place that stumped us is in Chapter 13, which requires a term for a parent's sibling. Because the English language has no such gender-neutral term, we regretfully stuck with "uncle.")
- The chapter notes, bibliography, and index were updated, reflecting the dramatic growth of the field of algorithms since the third edition.
- We corrected errors, posting most corrections on our website of third-edition errata. Those that were reported while we were in full swing preparing this edition were not posted, but were corrected in this edition. (Thanks again to all readers who helped us identify issues.)

The specific changes for the fourth edition include the following:

- We renamed Chapter 3 and added a section giving an overview of asymptotic notation before delving into the formal definitions.
- Chapter 4 underwent substantial changes to improve its mathematical foundation and make it more robust and intuitive. The notion of an algorithmic recurrence was introduced, and the topic of ignoring floors and ceilings in recur-

rences was addressed more rigorously. The second case of the master theorem incorporates polylogarithmic factors, and a rigorous proof of a "continuous" version of the master theorem is now provided. We also present the powerful and general Akra-Bazzi method (without proof).

- The deterministic order-statistic algorithm in Chapter 9 is slightly different, and the analyses of both the randomized and deterministic order-statistic algorithms have been revamped.
- In addition to stacks and queues, Section 10.1 discusses ways to store arrays and matrices.
- Chapter 11 on hash tables includes a modern treatment of hash functions. It also emphasizes linear probing as an efficient method for resolving collisions when the underlying hardware implements caching to favor local searches.
- To replace the sections on matroids in Chapter 15, we converted a problem in the third edition about offline caching into a full section.
- Section 16.4 now contains a more intuitive explanation of the potential functions to analyze table doubling and halving.
- Chapter 17 on augmenting data structures was relocated from Part III to Part V, reflecting our view that this technique goes beyond basic material.
- Chapter 25 is a new chapter about matchings in bipartite graphs. It presents algorithms to find a matching of maximum cardinality, to solve the stable-marriage problem, and to find a maximum-weight matching (known as the "assignment problem").
- Chapter 26, on task-parallel computing, has been updated with modern terminology, including the name of the chapter.
- Chapter 27, which covers online algorithms, is another new chapter. In an online algorithm, the input arrives over time, rather than being available in its entirety at the start of the algorithm. The chapter describes several examples of online algorithms, including determining how long to wait for an elevator before taking the stairs, maintaining a linked list via the move-to-front heuristic, and evaluating replacement policies for caches.
- In Chapter 29, we removed the detailed presentation of the simplex algorithm, as it was math heavy without really conveying many algorithmic ideas. The chapter now focuses on the key aspect of how to model problems as linear programs, along with the essential duality property of linear programming.
- Section 32.5 adds to the chapter on string matching the simple, yet powerful, structure of suffix arrays.

- Chapter 33, on machine learning, is the third new chapter. It introduces several basic methods used in machine learning: clustering to group similar items together, weighted-majority algorithms, and gradient descent to find the minimizer of a function.
- Section 34.5.6 summarizes strategies for polynomial-time reductions to show that problems are NP-hard.
- The proof of the approximation algorithm for the set-covering problem in Section 35.3 has been revised.

Website

You can use our website, http://mitpress.mit.edu/algorithms/, to obtain supplementary information and to communicate with us. The website links to a list of known errors, material from the third edition that is not included in the fourth edition, solutions to selected exercises and problems, Python implementations of many of the algorithms in this book, a list explaining the corny professor jokes (of course), as well as other content, which we may add to. The website also tells you how to report errors or make suggestions.

How we produced this book

Like the previous three editions, the fourth edition was produced in LaTeX 2_ε. We used the Times font with mathematics typeset using the MathTime Professional II fonts. As in all previous editions, we compiled the index using Windex, a C program that we wrote, and produced the bibliography using BibTeX. The PDF files for this book were created on a MacBook Pro running macOS 10.14.

Our plea to Apple in the preface of the third edition to update MacDraw Pro for macOS 10 went for naught, and so we continued to draw illustrations on pre-Intel Macs running MacDraw Pro under the Classic environment of older versions of macOS 10. Many of the mathematical expressions appearing in illustrations were laid in with the psfrag package for LaTeX 2_ε.

Acknowledgments for the fourth edition

We have been working with the MIT Press since we started writing the first edition in 1987, collaborating with several directors, editors, and production staff. Throughout our association with the MIT Press, their support has always been outstanding. Special thanks to our editors Marie Lee, who put up with us for far too long, and Elizabeth Swayze, who pushed us over the finish line. Thanks also to Director Amy Brand and to Alex Hoopes.

As in the third edition, we were geographically distributed while producing the fourth edition, working in the Dartmouth College Department of Computer Science; the MIT Computer Science and Artificial Intelligence Laboratory and the MIT Department of Electrical Engineering and Computer Science; and the Columbia University Department of Industrial Engineering and Operations Research, Department of Computer Science, and Data Science Institute. During the COVID-19 pandemic, we worked largely from home. We thank our respective universities and colleagues for providing such supportive and stimulating environments. As we complete this book, those of us who are not retired are eager to return to our respective universities now that the pandemic seems to be abating.

Julie Sussman, P.P.A., came to our rescue once again with her technical copyediting under tremendous time pressure. If not for Julie, this book would be riddled with errors (or, let's say, many more errors than it has) and would be far less readable. Julie, we will be forever indebted to you. Errors that remain are the responsibility of the authors (and probably were inserted after Julie read the material).

Dozens of errors in previous editions were corrected in the process of creating this edition. We thank our readers—too many to list them all—who have reported errors and suggested improvements over the years.

We received considerable help in preparing some of the new material in this edition. Neville Campbell (unaffiliated), Bill Kuszmaul of MIT, and Chee Yap of NYU provided valuable advice regarding the treatment of recurrences in Chapter 4. Yan Gu of the University of California, Riverside, provided feedback on parallel algorithms in Chapter 26. Rob Shapire of Microsoft Research altered our approach to the material on machine learning with his detailed comments on Chapter 33. Qi Qi of MIT helped with the analysis of the Monty Hall problem (Problem C-1).

Molly Seaman and Mary Reilly of the MIT Press helped us select the color palette in the illustrations, and Wojciech Jarosz of Dartmouth College suggested design improvements to our newly colored figures. Yichen (Annie) Ke and Linda Xiao, who have since graduated from Dartmouth, aided in colorizing the illustrations, and Linda also produced many of the Python implementations that are available on the book's website.

Finally, we thank our wives—Wendy Leiserson, Gail Rivest, Rebecca Ivry, and the late Nicole Cormen—and our families. The patience and encouragement of those who love us made this project possible. We affectionately dedicate this book to them.

THOMAS H. CORMEN *Lebanon, New Hampshire*
CHARLES E. LEISERSON *Cambridge, Massachusetts*
RONALD L. RIVEST *Cambridge, Massachusetts*
CLIFFORD STEIN *New York, New York*

June, 2021

Part I Foundations

Introduction

When you design and analyze algorithms, you need to be able to describe how they operate and how to design them. You also need some mathematical tools to show that your algorithms do the right thing and do it efficiently. This part will get you started. Later parts of this book will build upon this base.

Chapter 1 provides an overview of algorithms and their place in modern computing systems. This chapter defines what an algorithm is and lists some examples. It also makes a case for considering algorithms as a technology, alongside technologies such as fast hardware, graphical user interfaces, object-oriented systems, and networks.

In Chapter 2, we see our first algorithms, which solve the problem of sorting a sequence of n numbers. They are written in a pseudocode which, although not directly translatable to any conventional programming language, conveys the structure of the algorithm clearly enough that you should be able to implement it in the language of your choice. The sorting algorithms we examine are insertion sort, which uses an incremental approach, and merge sort, which uses a recursive technique known as "divide-and-conquer." Although the time each requires increases with the value of n, the rate of increase differs between the two algorithms. We determine these running times in Chapter 2, and we develop a useful "asymptotic" notation to express them.

Chapter 3 precisely defines asymptotic notation. We'll use asymptotic notation to bound the growth of functions—most often, functions that describe the running time of algorithms—from above and below. The chapter starts by informally defining the most commonly used asymptotic notations and giving an example of how to apply them. It then formally defines five asymptotic notations and presents conventions for how to put them together. The rest of Chapter 3 is primarily a presentation of mathematical notation, more to ensure that your use of notation matches that in this book than to teach you new mathematical concepts.

Chapter 4 delves further into the divide-and-conquer method introduced in Chapter 2. It provides two additional examples of divide-and-conquer algorithms for multiplying square matrices, including Strassen's surprising method. Chapter 4 contains methods for solving recurrences, which are useful for describing the running times of recursive algorithms. In the substitution method, you guess an answer and prove it correct. Recursion trees provide one way to generate a guess. Chapter 4 also presents the powerful technique of the "master method," which you can often use to solve recurrences that arise from divide-and-conquer algorithms. Although the chapter provides a proof of a foundational theorem on which the master theorem depends, you should feel free to employ the master method without delving into the proof. Chapter 4 concludes with some advanced topics.

Chapter 5 introduces probabilistic analysis and randomized algorithms. You typically use probabilistic analysis to determine the running time of an algorithm in cases in which, due to the presence of an inherent probability distribution, the running time may differ on different inputs of the same size. In some cases, you might assume that the inputs conform to a known probability distribution, so that you are averaging the running time over all possible inputs. In other cases, the probability distribution comes not from the inputs but from random choices made during the course of the algorithm. An algorithm whose behavior is determined not only by its input but by the values produced by a random-number generator is a randomized algorithm. You can use randomized algorithms to enforce a probability distribution on the inputs—thereby ensuring that no particular input always causes poor performance—or even to bound the error rate of algorithms that are allowed to produce incorrect results on a limited basis.

Appendices A–D contain other mathematical material that you will find helpful as you read this book. You might have seen much of the material in the appendix chapters before having read this book (although the specific definitions and notational conventions we use may differ in some cases from what you have seen in the past), and so you should think of the appendices as reference material. On the other hand, you probably have not already seen most of the material in Part I. All the chapters in Part I and the appendices are written with a tutorial flavor.

1 The Role of Algorithms in Computing

What are algorithms? Why is the study of algorithms worthwhile? What is the role of algorithms relative to other technologies used in computers? This chapter will answer these questions.

1.1 Algorithms

Informally, an ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output*** in a finite amount of time. An algorithm is thus a sequence of computational steps that transform the input into the output.

You can also view an algorithm as a tool for solving a well-specified ***computational problem***. The statement of the problem specifies in general terms the desired input/output relationship for problem instances, typically of arbitrarily large size. The algorithm describes a specific computational procedure for achieving that input/output relationship for all problem instances.

As an example, suppose that you need to sort a sequence of numbers into monotonically increasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the ***sorting problem***:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Thus, given the input sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$, a correct sorting algorithm returns as output the sequence $\langle 26, 31, 41, 41, 58, 59 \rangle$. Such an input sequence is

called an ***instance*** of the sorting problem. In general, an ***instance of a problem***[1] consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Because many programs use it as an intermediate step, sorting is a fundamental operation in computer science. As a result, you have a large number of good sorting algorithms at your disposal. Which algorithm is best for a given application depends on—among other factors—the number of items to be sorted, the extent to which the items are already somewhat sorted, possible restrictions on the item values, the architecture of the computer, and the kind of storage devices to be used: main memory, disks, or even—archaically—tapes.

An algorithm for a computational problem is ***correct*** if, for every problem instance provided as input, it ***halts***—finishes its computing in finite time—and outputs the correct solution to the problem instance. A correct algorithm ***solves*** the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer. Contrary to what you might expect, incorrect algorithms can sometimes be useful, if you can control their error rate. We'll see an example of an algorithm with a controllable error rate in Chapter 31 when we study algorithms for finding large prime numbers. Ordinarily, however, we'll concern ourselves only with correct algorithms.

An algorithm can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a precise description of the computational procedure to be followed.

What kinds of problems are solved by algorithms?

Sorting is by no means the only computational problem for which algorithms have been developed. (You probably suspected as much when you saw the size of this book.) Practical applications of algorithms are ubiquitous and include the following examples:

- The Human Genome Project has made great progress toward the goals of identifying all the roughly 30,000 genes in human DNA, determining the sequences of the roughly 3 billion chemical base pairs that make up human DNA, storing this information in databases, and developing tools for data analysis. Each of these steps requires sophisticated algorithms. Although the solutions to the various problems involved are beyond the scope of this book, many methods to solve these biological problems use ideas presented here, enabling scientists to accomplish tasks while using resources efficiently. Dynamic programming, as

[1] Sometimes, when the problem context is known, problem instances are themselves simply called "problems."

in Chapter 14, is an important technique for solving several of these biological problems, particularly ones that involve determining similarity between DNA sequences. The savings realized are in time, both human and machine, and in money, as more information can be extracted by laboratory techniques.

- The internet enables people all around the world to quickly access and retrieve large amounts of information. With the aid of clever algorithms, sites on the internet are able to manage and manipulate this large volume of data. Examples of problems that make essential use of algorithms include finding good routes on which the data travels (techniques for solving such problems appear in Chapter 22), and using a search engine to quickly find pages on which particular information resides (related techniques are in Chapters 11 and 32).
- Electronic commerce enables goods and services to be negotiated and exchanged electronically, and it depends on the privacy of personal information such as credit card numbers, passwords, and bank statements. The core technologies used in electronic commerce include public-key cryptography and digital signatures (covered in Chapter 31), which are based on numerical algorithms and number theory.
- Manufacturing and other commercial enterprises often need to allocate scarce resources in the most beneficial way. An oil company might wish to know where to place its wells in order to maximize its expected profit. A political candidate might want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election. An airline might wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that government regulations regarding crew scheduling are met. An internet service provider might wish to determine where to place additional resources in order to serve its customers more effectively. All of these are examples of problems that can be solved by modeling them as linear programs, which Chapter 29 explores.

Although some of the details of these examples are beyond the scope of this book, we do give underlying techniques that apply to these problems and problem areas. We also show how to solve many specific problems, including the following:

- You have a road map on which the distance between each pair of adjacent intersections is marked, and you wish to determine the shortest route from one intersection to another. The number of possible routes can be huge, even if you disallow routes that cross over themselves. How can you choose which of all possible routes is the shortest? You can start by modeling the road map (which is itself a model of the actual roads) as a graph (which we will meet in Part VI and Appendix B). In this graph, you wish to find the shortest path from one vertex to another. Chapter 22 shows how to solve this problem efficiently.

- Given a mechanical design in terms of a library of parts, where each part may include instances of other parts, list the parts in order so that each part appears before any part that uses it. If the design comprises n parts, then there are $n!$ possible orders, where $n!$ denotes the factorial function. Because the factorial function grows faster than even an exponential function, you cannot feasibly generate each possible order and then verify that, within that order, each part appears before the parts using it (unless you have only a few parts). This problem is an instance of topological sorting, and Chapter 20 shows how to solve this problem efficiently.
- A doctor needs to determine whether an image represents a cancerous tumor or a benign one. The doctor has available images of many other tumors, some of which are known to be cancerous and some of which are known to be benign. A cancerous tumor is likely to be more similar to other cancerous tumors than to benign tumors, and a benign tumor is more likely to be similar to other benign tumors. By using a clustering algorithm, as in Chapter 33, the doctor can identify which outcome is more likely.
- You need to compress a large file containing text so that it occupies less space. Many ways to do so are known, including "LZW compression," which looks for repeating character sequences. Chapter 15 studies a different approach, "Huffman coding," which encodes characters by bit sequences of various lengths, with characters occurring more frequently encoded by shorter bit sequences.

These lists are far from exhaustive (as you again have probably surmised from this book's heft), but they exhibit two characteristics common to many interesting algorithmic problems:

1. They have many candidate solutions, the overwhelming majority of which do not solve the problem at hand. Finding one that does, or one that is "best," without explicitly examining each possible solution, can present quite a challenge.
2. They have practical applications. Of the problems in the above list, finding the shortest path provides the easiest examples. A transportation firm, such as a trucking or railroad company, has a financial interest in finding shortest paths through a road or rail network because taking shorter paths results in lower labor and fuel costs. Or a routing node on the internet might need to find the shortest path through the network in order to route a message quickly. Or a person wishing to drive from New York to Boston might want to find driving directions using a navigation app.

Not every problem solved by algorithms has an easily identified set of candidate solutions. For example, given a set of numerical values representing samples of a signal taken at regular time intervals, the discrete Fourier transform converts

the time domain to the frequency domain. That is, it approximates the signal as a weighted sum of sinusoids, producing the strength of various frequencies which, when summed, approximate the sampled signal. In addition to lying at the heart of signal processing, discrete Fourier transforms have applications in data compression and multiplying large polynomials and integers. Chapter 30 gives an efficient algorithm, the fast Fourier transform (commonly called the FFT), for this problem. The chapter also sketches out the design of a hardware FFT circuit.

Data structures

This book also presents several data structures. A ***data structure*** is a way to store and organize data in order to facilitate access and modifications. Using the appropriate data structure or structures is an important part of algorithm design. No single data structure works well for all purposes, and so you should know the strengths and limitations of several of them.

Technique

Although you can use this book as a "cookbook" for algorithms, you might someday encounter a problem for which you cannot readily find a published algorithm (many of the exercises and problems in this book, for example). This book will teach you techniques of algorithm design and analysis so that you can develop algorithms on your own, show that they give the correct answer, and analyze their efficiency. Different chapters address different aspects of algorithmic problem solving. Some chapters address specific problems, such as finding medians and order statistics in Chapter 9, computing minimum spanning trees in Chapter 21, and determining a maximum flow in a network in Chapter 24. Other chapters introduce techniques, such as divide-and-conquer in Chapters 2 and 4, dynamic programming in Chapter 14, and amortized analysis in Chapter 16.

Hard problems

Most of this book is about efficient algorithms. Our usual measure of efficiency is speed: how long does an algorithm take to produce its result? There are some problems, however, for which we know of no algorithm that runs in a reasonable amount of time. Chapter 34 studies an interesting subset of these problems, which are known as NP-complete.

Why are NP-complete problems interesting? First, although no efficient algorithm for an NP-complete problem has ever been found, nobody has ever proven that an efficient algorithm for one cannot exist. In other words, no one knows whether efficient algorithms exist for NP-complete problems. Second, the set of

NP-complete problems has the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them. This relationship among the NP-complete problems makes the lack of efficient solutions all the more tantalizing. Third, several NP-complete problems are similar, but not identical, to problems for which we do know of efficient algorithms. Computer scientists are intrigued by how a small change to the problem statement can cause a big change to the efficiency of the best known algorithm.

You should know about NP-complete problems because some of them arise surprisingly often in real applications. If you are called upon to produce an efficient algorithm for an NP-complete problem, you are likely to spend a lot of time in a fruitless search. If, instead, you can show that the problem is NP-complete, you can spend your time developing an efficient approximation algorithm, that is, an algorithm that gives a good, but not necessarily the best possible, solution.

As a concrete example, consider a delivery company with a central depot. Each day, it loads up delivery trucks at the depot and sends them around to deliver goods to several addresses. At the end of the day, each truck must end up back at the depot so that it is ready to be loaded for the next day. To reduce costs, the company wants to select an order of delivery stops that yields the lowest overall distance traveled by each truck. This problem is the well-known "traveling-salesperson problem," and it is NP-complete.[2] It has no known efficient algorithm. Under certain assumptions, however, we know of efficient algorithms that compute overall distances close to the smallest possible. Chapter 35 discusses such "approximation algorithms."

Alternative computing models

For many years, we could count on processor clock speeds increasing at a steady rate. Physical limitations present a fundamental roadblock to ever-increasing clock speeds, however: because power density increases superlinearly with clock speed, chips run the risk of melting once their clock speeds become high enough. In order to perform more computations per second, therefore, chips are being designed to contain not just one but several processing "cores." We can liken these multicore computers to several sequential computers on a single chip. In other words, they are a type of "parallel computer." In order to elicit the best performance from multicore computers, we need to design algorithms with parallelism in mind. Chapter 26 presents a model for "task-parallel" algorithms, which take advantage of multiple processing cores. This model has advantages from both theoretical and

[2] To be precise, only decision problems—those with a "yes/no" answer—can be NP-complete. The decision version of the traveling salesperson problem asks whether there exists an order of stops whose distance totals at most a given amount.

practical standpoints, and many modern parallel-programming platforms embrace something similar to this model of parallelism.

Most of the examples in this book assume that all of the input data are available when an algorithm begins running. Much of the work in algorithm design makes the same assumption. For many important real-world examples, however, the input actually arrives over time, and the algorithm must decide how to proceed without knowing what data will arrive in the future. In a data center, jobs are constantly arriving and departing, and a scheduling algorithm must decide when and where to run a job, without knowing what jobs will be arriving in the future. Traffic must be routed in the internet based on the current state, without knowing about where traffic will arrive in the future. Hospital emergency rooms make triage decisions about which patients to treat first without knowing when other patients will be arriving in the future and what treatments they will need. Algorithms that receive their input over time, rather than having all the input present at the start, are ***online algorithms***, which Chapter 27 examines.

Exercises

1.1-1
Describe your own real-world example that requires sorting. Describe one that requires finding the shortest distance between two points.

1.1-2
Other than speed, what other measures of efficiency might you need to consider in a real-world setting?

1.1-3
Select a data structure that you have seen, and discuss its strengths and limitations.

1.1-4
How are the shortest-path and traveling-salesperson problems given above similar? How are they different?

1.1-5
Suggest a real-world problem in which only the best solution will do. Then come up with one in which "approximately" the best solution is good enough.

1.1-6
Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time.

1.2 Algorithms as a technology

If computers were infinitely fast and computer memory were free, would you have any reason to study algorithms? The answer is yes, if for no other reason than that you would still like to be certain that your solution method terminates and does so with the correct answer.

If computers were infinitely fast, any correct method for solving a problem would do. You would probably want your implementation to be within the bounds of good software engineering practice (for example, your implementation should be well designed and documented), but you would most often use whichever method was the easiest to implement.

Of course, computers may be fast, but they are not infinitely fast. Computing time is therefore a bounded resource, which makes it precious. Although the saying goes, "Time is money," time is even more valuable than money: you can get back money after you spend it, but once time is spent, you can never get it back. Memory may be inexpensive, but it is neither infinite nor free. You should choose algorithms that use the resources of time and space efficiently.

Efficiency

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, Chapter 2 introduces two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We'll see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. For example, when n is 1000, $\lg n$ is approximately 10, and when n is 1,000,000, $\lg n$ is approximately only 20. Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ versus n more than compensates for the difference in constant factors. No matter how much smaller c_1 is than c_2, there is always a crossover point beyond which merge sort is faster.

For a concrete example, let us pit a faster computer (computer A) running insertion sort against a slower computer (computer B) running merge sort. They each must sort an array of 10 million numbers. (Although 10 million numbers might seem like a lot, if the numbers are eight-byte integers, then the input occupies about 80 megabytes, which fits in the memory of even an inexpensive laptop computer many times over.) Suppose that computer A executes 10 billion instructions per second (faster than any single sequential computer at the time of this writing) and computer B executes only 10 million instructions per second (much slower than most contemporary computers), so that computer A is 1000 times faster than computer B in raw computing power. To make the difference even more dramatic, suppose that the world's craftiest programmer codes insertion sort in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Suppose further that just an average programmer implements merge sort, using a high-level language with an inefficient compiler, with the resulting code taking $50\,n \lg n$ instructions. To sort 10 million numbers, computer A takes

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20{,}000 \text{ seconds (more than 5.5 hours)} ,$$

while computer B takes

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ seconds (under 20 minutes)} .$$

By using an algorithm whose running time grows more slowly, even with a poor compiler, computer B runs more than 17 times faster than computer A! The advantage of merge sort is even more pronounced when sorting 100 million numbers: where insertion sort takes more than 23 days, merge sort takes under four hours. Although 100 million might seem like a large number, there are more than 100 million web searches every half hour, more than 100 million emails sent every minute, and some of the smallest galaxies (known as ultra-compact dwarf galaxies) contain about 100 million stars. In general, as the problem size increases, so does the relative advantage of merge sort.

Algorithms and other technologies

The example above shows that you should consider algorithms, like computer hardware, as a ***technology***. Total system performance depends on choosing efficient algorithms as much as on choosing fast hardware. Just as rapid advances are being made in other computer technologies, they are being made in algorithms as well.

You might wonder whether algorithms are truly that important on contemporary computers in light of other advanced technologies, such as

- advanced computer architectures and fabrication technologies,
- easy-to-use, intuitive, graphical user interfaces (GUIs),
- object-oriented systems,
- integrated web technologies,
- fast networking, both wired and wireless,
- machine learning,
- and mobile devices.

The answer is yes. Although some applications do not explicitly require algorithmic content at the application level (such as some simple, web-based applications), many do. For example, consider a web-based service that determines how to travel from one location to another. Its implementation would rely on fast hardware, a graphical user interface, wide-area networking, and also possibly on object orientation. It would also require algorithms for operations such as finding routes (probably using a shortest-path algorithm), rendering maps, and interpolating addresses.

Moreover, even an application that does not require algorithmic content at the application level relies heavily upon algorithms. Does the application rely on fast hardware? The hardware design used algorithms. Does the application rely on graphical user interfaces? The design of any GUI relies on algorithms. Does the application rely on networking? Routing in networks relies heavily on algorithms. Was the application written in a language other than machine code? Then it was processed by a compiler, interpreter, or assembler, all of which make extensive use of algorithms. Algorithms are at the core of most technologies used in contemporary computers.

Machine learning can be thought of as a method for performing algorithmic tasks without explicitly designing an algorithm, but instead inferring patterns from data and thereby automatically learning a solution. At first glance, machine learning, which automates the process of algorithmic design, may seem to make learning about algorithms obsolete. The opposite is true, however. Machine learning is itself a collection of algorithms, just under a different name. Furthermore, it currently seems that the successes of machine learning are mainly for problems for which we, as humans, do not really understand what the right algorithm is. Prominent examples include computer vision and automatic language translation. For algorithmic problems that humans understand well, such as most of the problems in this book, efficient algorithms designed to solve a specific problem are typically more successful than machine-learning approaches.

Data science is an interdisciplinary field with the goal of extracting knowledge and insights from structured and unstructured data. Data science uses methods

from statistics, computer science, and optimization. The design and analysis of algorithms is fundamental to the field. The core techniques of data science, which overlap significantly with those in machine learning, include many of the algorithms in this book.

Furthermore, with the ever-increasing capacities of computers, we use them to solve larger problems than ever before. As we saw in the above comparison between insertion sort and merge sort, it is at larger problem sizes that the differences in efficiency between algorithms become particularly prominent.

Having a solid base of algorithmic knowledge and technique is one characteristic that defines the truly skilled programmer. With modern computing technology, you can accomplish some tasks without knowing much about algorithms, but with a good background in algorithms, you can do much, much more.

Exercises

1.2-1
Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

1.2-2
Suppose that for inputs of size n on a particular computer, insertion sort runs in $8n^2$ steps and merge sort runs in $64\,n \lg n$ steps. For which values of n does insertion sort beat merge sort?

1.2-3
What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

Problems

1-1 Comparison of running times
For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

Chapter notes

There are many excellent texts on the general topic of algorithms, including those by Aho, Hopcroft, and Ullman [5, 6], Dasgupta, Papadimitriou, and Vazirani [107], Edmonds [133], Erickson [135], Goodrich and Tamassia [195, 196], Kleinberg and Tardos [257], Knuth [259, 260, 261, 262, 263], Levitin [298], Louridas [305], Mehlhorn and Sanders [325], Mitzenmacher and Upfal [331], Neapolitan [342], Roughgarden [385, 386, 387, 388], Sanders, Mehlhorn, Dietzfelbinger, and Dementiev [393], Sedgewick and Wayne [402], Skiena [414], Soltys-Kulinicz [419], Wilf [455], and Williamson and Shmoys [459]. Some of the more practical aspects of algorithm design are discussed by Bentley [49, 50, 51], Bhargava [54], Kochenderfer and Wheeler [268], and McGeoch [321]. Surveys of the field of algorithms can also be found in books by Atallah and Blanton [27, 28] and Mehta and Sahhi [326]. For less technical material, see the books by Christian and Griffiths [92], Cormen [104], Erwig [136], MacCormick [307], and Vöcking et al. [448]. Overviews of the algorithms used in computational biology can be found in books by Jones and Pevzner [240], Elloumi and Zomaya [134], and Marchisio [315].

2 Getting Started

This chapter will familiarize you with the framework we'll use throughout the book to think about the design and analysis of algorithms. It is self-contained, but it does include several references to material that will be introduced in Chapters 3 and 4. (It also contains several summations, which Appendix A shows how to solve.)

We'll begin by examining the insertion sort algorithm to solve the sorting problem introduced in Chapter 1. We'll specify algorithms using a pseudocode that should be understandable to you if you have done computer programming. We'll see why insertion sort correctly sorts and analyze its running time. The analysis introduces a notation that describes how running time increases with the number of items to be sorted. Following a discussion of insertion sort, we'll use a method called divide-and-conquer to develop a sorting algorithm called merge sort. We'll end with an analysis of merge sort's running time.

2.1 Insertion sort

Our first algorithm, insertion sort, solves the ***sorting problem*** introduced in Chapter 1:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

The numbers to be sorted are also known as the ***keys***. Although the problem is conceptually about sorting a sequence, the input comes in the form of an array with n elements. When we want to sort numbers, it's often because they are the keys associated with other data, which we call ***satellite data***. Together, a key and satellite data form a ***record***. For example, consider a spreadsheet containing student records with many associated pieces of data such as age, grade-point average, and number of courses taken. Any one of these quantities could be a key, but when the

spreadsheet sorts, it moves the associated record (the satellite data) with the key. When describing a sorting algorithm, we focus on the keys, but it is important to remember that there usually is associated satellite data.

In this book, we'll typically describe algorithms as procedures written in a ***pseudocode*** that is similar in many respects to C, C++, Java, Python,[1] or JavaScript. (Apologies if we've omitted your favorite programming language. We can't list them all.) If you have been introduced to any of these languages, you should have little trouble understanding algorithms "coded" in pseudocode. What separates pseudocode from real code is that in pseudocode, we employ whatever expressive method is most clear and concise to specify a given algorithm. Sometimes the clearest method is English, so do not be surprised if you come across an English phrase or sentence embedded within a section that looks more like real code. Another difference between pseudocode and real code is that pseudocode often ignores aspects of software engineering—such as data abstraction, modularity, and error handling—in order to convey the essence of the algorithm more concisely.

We start with ***insertion sort***, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way you might sort a hand of playing cards. Start with an empty left hand and the cards in a pile on the table. Pick up the first card in the pile and hold it with your left hand. Then, with your right hand, remove one card at a time from the pile, and insert it into the correct position in your left hand. As Figure 2.1 illustrates, you find the correct position for a card by comparing it with each of the cards already in your left hand, starting at the right and moving left. As soon as you see a card in your left hand whose value is less than or equal to the card you're holding in your right hand, insert the card that you're holding in your right hand just to the right of this card in your left hand. If all the cards in your left hand have values greater than the card in your right hand, then place this card as the leftmost card in your left hand. At all times, the cards held in your left hand are sorted, and these cards were originally the top cards of the pile on the table.

The pseudocode for insertion sort is given as the procedure INSERTION-SORT on the facing page. It takes two parameters: an array A containing the values to be sorted and the number n of values of sort. The values occupy positions $A[1]$ through $A[n]$ of the array, which we denote by $A[1:n]$. When the INSERTION-SORT procedure is finished, array $A[1:n]$ contains the original values, but in sorted order.

---

[1] If you're familiar with only Python, you can think of arrays as similar to Python lists.

Figure 2.1 Sorting a hand of cards using insertion sort.

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

Loop invariants and the correctness of insertion sort

Figure 2.2 shows how this algorithm works for an array A that starts out with the sequence $\langle 5, 2, 4, 6, 1, 3 \rangle$. The index i indicates the "current card" being inserted into the hand. At the beginning of each iteration of the **for** loop, which is indexed by i, the ***subarray*** (a contiguous portion of the array) consisting of elements $A[1 : i - 1]$ (that is, $A[1]$ through $A[i - 1]$) constitutes the currently sorted hand, and the remaining subarray $A[i + 1 : n]$ (elements $A[i + 1]$ through $A[n]$) corresponds to the pile of cards still on the table. In fact, elements $A[1 : i - 1]$ are the elements *originally* in positions 1 through $i - 1$, but now in sorted order. We state these properties of $A[1 : i - 1]$ formally as a ***loop invariant***:

Figure 2.2 The operation of INSERTION-SORT(A, n), where A initially contains the sequence $\langle 5, 2, 4, 6, 1, 3 \rangle$ and $n = 6$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the blue rectangle holds the key taken from $A[i]$, which is compared with the values in tan rectangles to its left in the test of line 5. Orange arrows show array values moved one position to the right in line 6, and blue arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

> At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1 : i - 1]$ consists of the elements originally in $A[1 : i - 1]$, but in sorted order.

Loop invariants help us understand why an algorithm is correct. When you're using a loop invariant, you need to show three things:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: The loop terminates, and when it terminates, the invariant—usually along with the reason that the loop terminated—gives us a useful property that helps show that the algorithm is correct.

When the first two properties hold, the loop invariant is true prior to every iteration of the loop. (Of course, you are free to use established facts other than the loop invariant itself to prove that the loop invariant remains true before each iteration.) A loop-invariant proof is a form of mathematical induction, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration corresponds to the base case, and showing that the invariant holds from iteration to iteration corresponds to the inductive step.

The third property is perhaps the most important one, since you are using the loop invariant to show correctness. Typically, you use the loop invariant along with the condition that caused the loop to terminate. Mathematical induction typically applies the inductive step infinitely, but in a loop invariant the "induction" stops when the loop terminates.

Let's see how these properties hold for insertion sort.

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $i = 2$.[2] The subarray $A[1:i-1]$ consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (after all, how could a subarray with just one value not be sorted?), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the **for** loop works by moving the values in $A[i-1]$, $A[i-2]$, $A[i-3]$, and so on by one position to the right until it finds the proper position for $A[i]$ (lines 4–7), at which point it inserts the value of $A[i]$ (line 8). The subarray $A[1:i]$ then consists of the elements originally in $A[1:i]$, but in sorted order. ***Incrementing*** i (increasing its value by 1) for the next iteration of the **for** loop then preserves the loop invariant.

A more formal treatment of the second property would require us to state and show a loop invariant for the **while** loop of lines 5–7. Let's not get bogged down in such formalism just yet. Instead, we'll rely on our informal analysis to show that the second property holds for the outer loop.

Termination: Finally, we examine loop termination. The loop variable i starts at 2 and increases by 1 in each iteration. Once i's value exceeds n in line 1, the loop terminates. That is, the loop terminates once i equals $n + 1$. Substituting $n + 1$ for i in the wording of the loop invariant yields that the subarray $A[1:n]$ consists of the elements originally in $A[1:n]$, but in sorted order. Hence, the algorithm is correct.

This method of loop invariants is used to show correctness in various places throughout this book.

Pseudocode conventions

We use the following conventions in our pseudocode.

- Indentation indicates block structure. For example, the body of the **for** loop that begins on line 1 consists of lines 2–8, and the body of the **while** loop that

[2] When the loop is a **for** loop, the loop-invariant check just prior to the first iteration occurs immediately after the initial assignment to the loop-counter variable and just before the first test in the loop header. In the case of INSERTION-SORT, this time is after assigning 2 to the variable i but before the first test of whether $i \le n$.

begins on line 5 contains lines 6–7 but not line 8. Our indentation style applies to **if-else** statements[3] as well. Using indentation instead of textual indicators of block structure, such as **begin** and **end** statements or curly braces, reduces clutter while preserving, or even enhancing, clarity.[4]

- The looping constructs **while**, **for**, and **repeat-until** and the **if-else** conditional construct have interpretations similar to those in C, C++, Java, Python, and JavaScript.[5] In this book, the loop counter retains its value after the loop is exited, unlike some situations that arise in C++ and Java. Thus, immediately after a **for** loop, the loop counter's value is the value that first exceeded the **for** loop bound.[6] We used this property in our correctness argument for insertion sort. The **for** loop header in line 1 is **for** $i = 2$ **to** n, and so when this loop terminates, i equals $n+1$. We use the keyword **to** when a **for** loop increments its loop counter in each iteration, and we use the keyword **downto** when a **for** loop ***decrements*** its loop counter (reduces its value by 1 in each iteration). When the loop counter changes by an amount greater than 1, the amount of change follows the optional keyword **by**.
- The symbol "**//**" indicates that the remainder of the line is a comment.
- Variables (such as i, j, and *key*) are local to the given procedure. We won't use global variables without explicit indication.
- We access array elements by specifying the array name followed by the index in square brackets. For example, $A[i]$ indicates the ith element of the array A.

  Although many programming languages enforce 0-origin indexing for arrays (0 is the smallest valid index), we choose whichever indexing scheme is clearest for human readers to understand. Because people usually start counting at 1, not 0, most—but not all—of the arrays in this book use 1-origin indexing. To be

---

[3] In an **if-else** statement, we indent **else** at the same level as its matching **if**. The first executable line of an **else** clause appears on the same line as the keyword **else**. For multiway tests, we use **elseif** for tests after the first one. When it is the first line in an **else** clause, an **if** statement appears on the line following **else** so that you do not misconstrue it as **elseif**.

[4] Each pseudocode procedure in this book appears on one page so that you do not need to discern levels of indentation in pseudocode that is split across pages.

[5] Most block-structured languages have equivalent constructs, though the exact syntax may differ. Python lacks **repeat-until** loops, and its **for** loops operate differently from the **for** loops in this book. Think of the pseudocode line "**for** $i = 1$ **to** n" as equivalent to "for i in range(1, n+1)" in Python.

[6] In Python, the loop counter retains its value after the loop is exited, but the value it retains is the value it had during the final iteration of the **for** loop, rather than the value that exceeded the loop bound. That is because a Python **for** loop iterates through a list, which may contain nonnumeric values.

clear about whether a particular algorithm assumes 0-origin or 1-origin indexing, we'll specify the bounds of the arrays explicitly. If you are implementing an algorithm that we specify using 1-origin indexing, but you're writing in a programming language that enforces 0-origin indexing (such as C, C++, Java, Python, or JavaScript), then give yourself credit for being able to adjust. You can either always subtract 1 from each index or allocate each array with one extra position and just ignore position 0.

The notation ":" denotes a subarray. Thus, $A[i : j]$ indicates the subarray of A consisting of the elements $A[i], A[i + 1], \ldots, A[j]$.[7] We also use this notation to indicate the bounds of an array, as we did earlier when discussing the array $A[1 : n]$.

- We typically organize compound data into ***objects***, which are composed of ***attributes***. We access a particular attribute using the syntax found in many object-oriented programming languages: the object name, followed by a dot, followed by the attribute name. For example, if an object x has attribute f, we denote this attribute by $x.f$.

  We treat a variable representing an array or object as a pointer (known as a reference in some programming languages) to the data representing the array or object. For all attributes f of an object x, setting $y = x$ causes $y.f$ to equal $x.f$. Moreover, if we now set $x.f = 3$, then afterward not only does $x.f$ equal 3, but $y.f$ equals 3 as well. In other words, x and y point to the same object after the assignment $y = x$. This way of treating arrays and objects is consistent with most contemporary programming languages.

  Our attribute notation can "cascade." For example, suppose that the attribute f is itself a pointer to some type of object that has an attribute g. Then the notation $x.f.g$ is implicitly parenthesized as $(x.f).g$. In other words, if we had assigned $y = x.f$, then $x.f.g$ is the same as $y.g$.

  Sometimes a pointer refers to no object at all. In this case, we give it the special value NIL.

- We pass parameters to a procedure ***by value***: the called procedure receives its own copy of the parameters, and if it assigns a value to a parameter, the change is *not* seen by the calling procedure. When objects are passed, the pointer to the data representing the object is copied, but the object's attributes are not. For example, if x is a parameter of a called procedure, the assignment $x = y$ within

[7] If you're used to programming in Python, bear in mind that in this book, the subarray $A[i : j]$ includes the element $A[j]$. In Python, the last element of $A[i : j]$ is $A[j - 1]$. Python allows negative indices, which count from the back end of the list. This book does not use negative array indices.

the called procedure is not visible to the calling procedure. The assignment $x.f = 3$, however, is visible if the calling procedure has a pointer to the same object as x. Similarly, arrays are passed by pointer, so that a pointer to the array is passed, rather than the entire array, and changes to individual array elements are visible to the calling procedure. Again, most contemporary programming languages work this way.

- A **return** statement immediately transfers control back to the point of call in the calling procedure. Most **return** statements also take a value to pass back to the caller. Our pseudocode differs from many programming languages in that we allow multiple values to be returned in a single **return** statement without having to create objects to package them together.[8]
- The boolean operators "and" and "or" are ***short circuiting***. That is, evaluate the expression "x and y" by first evaluating x. If x evaluates to FALSE, then the entire expression cannot evaluate to TRUE, and therefore y is not evaluated. If, on the other hand, x evaluates to TRUE, y must be evaluated to determine the value of the entire expression. Similarly, in the expression "x or y" the expression y is evaluated only if x evaluates to FALSE. Short-circuiting operators allow us to write boolean expressions such as "$x \neq$ NIL and $x.f = y$" without worrying about what happens upon evaluating $x.f$ when x is NIL.
- The keyword **error** indicates that an error occurred because conditions were wrong for the procedure to have been called, and the procedure immediately terminates. The calling procedure is responsible for handling the error, and so we do not specify what action to take.

Exercises

2.1-1

Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on an array initially containing the sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2

Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array $A[1:n]$. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUM-ARRAY procedure returns the sum of the numbers in $A[1:n]$.

---

[8] Python's tuple notation allows **return** statements to return multiple values without creating objects from a programmer-defined class.

```
SUM-ARRAY(A, n)
sum = 0
for i = 1 to n
    sum = sum + A[i]
return sum
```

2.1-3
Rewrite the INSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

2.1-4
Consider the ***searching problem***:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ stored in array $A[1:n]$ and a value x.

Output: An index i such that x equals $A[i]$ or the special value NIL if x does not appear in A.

Write pseudocode for ***linear search***, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

2.1-5
Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays $A[0:n-1]$ and $B[0:n-1]$, where each element is either 0 or 1, $a = \sum_{i=0}^{n-1} A[i] \cdot 2^i$, and $b = \sum_{i=0}^{n-1} B[i] \cdot 2^i$. The sum $c = a + b$ of the two integers should be stored in binary form in an $(n+1)$-element array $C[0:n]$, where $c = \sum_{i=0}^{n} C[i] \cdot 2^i$. Write a procedure ADD-BINARY-INTEGERS that takes as input arrays A and B, along with the length n, and returns array C holding the sum.

2.2 Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. You might consider resources such as memory, communication bandwidth, or energy consumption. Most often, however, you'll want to measure computational time. If you analyze several candidate algorithms for a problem,

you can identify the most efficient one. There might be more than just one viable candidate, but you can often rule out several inferior algorithms in the process.

Before you can analyze an algorithm, you need a model of the technology that it runs on, including the resources of that technology and a way to express their costs. Most of this book assumes a generic one-processor, ***random-access machine (RAM)*** model of computation as the implementation technology, with the understanding that algorithms are implemented as computer programs. In the RAM model, instructions execute one after another, with no concurrent operations. The RAM model assumes that each instruction takes the same amount of time as any other instruction and that each data access—using the value of a variable or storing into a variable—takes the same amount of time as any other data access. In other words, in the RAM model each instruction or data access takes a constant amount of time—even indexing into an array.[9]

Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and yield little insight into algorithm design and analysis. Yet we must be careful not to abuse the RAM model. For example, what if a RAM had an instruction that sorts? Then you could sort in just one step. Such a RAM would be unrealistic, since such instructions do not appear in real computers. Our guide, therefore, is how real computers are designed. The RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return).

The data types in the RAM model are integer, floating point (for storing real-number approximations), and character. Real computers do not usually have a separate data type for the boolean values TRUE and FALSE. Instead, they often test whether an integer value is 0 (FALSE) or nonzero (TRUE), as in C. Although we typically do not concern ourselves with precision for floating-point values in this book (many numbers cannot be represented exactly in floating point), precision is crucial for most applications. We also assume that each word of data has a limit on the number of bits. For example, when working with inputs of size n, we typically

---

[9] We assume that each element of a given array occupies the same number of bytes and that the elements of a given array are stored in contiguous memory locations. For example, if array $A[1:n]$ starts at memory address 1000 and each element occupies four bytes, then element $A[i]$ is at address $1000 + 4(i - 1)$. In general, computing the address in memory of a particular array element requires at most one subtraction (no subtraction for a 0-origin array), one multiplication (often implemented as a shift operation if the element size is an exact power of 2), and one addition. Furthermore, for code that iterates through the elements of an array in order, an optimizing compiler can generate the address of each element using just one addition, by adding the element size to the address of the preceding element.

assume that integers are represented by $c \log_2 n$ bits for some constant $c \geq 1$. We require $c \geq 1$ so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily. (If the word size could grow arbitrarily, we could store huge amounts of data in one word and operate on it all in constant time—an unrealistic scenario.)

Real computers contain instructions not listed above, and such instructions represent a gray area in the RAM model. For example, is exponentiation a constant-time instruction? In the general case, no: to compute x^n when x and n are general integers typically takes time logarithmic in n (see equation (31.34) on page 934), and you must worry about whether the result fits into a computer word. If n is an exact power of 2, however, exponentiation can usually be viewed as a constant-time operation. Many computers have a "shift left" instruction, which in constant time shifts the bits of an integer by n positions to the left. In most computers, shifting the bits of an integer by 1 position to the left is equivalent to multiplying by 2, so that shifting the bits by n positions to the left is equivalent to multiplying by 2^n. Therefore, such computers can compute 2^n in 1 constant-time instruction by shifting the integer 1 by n positions to the left, as long as n is no more than the number of bits in a computer word. We'll try to avoid such gray areas in the RAM model and treat computing 2^n and multiplying by 2^n as constant-time operations when the result is small enough to fit in a computer word.

The RAM model does not account for the memory hierarchy that is common in contemporary computers. It models neither caches nor virtual memory. Several other computational models attempt to account for memory-hierarchy effects, which are sometimes significant in real programs on real machines. Section 11.5 and a handful of problems in this book examine memory-hierarchy effects, but for the most part, the analyses in this book do not consider them. Models that include the memory hierarchy are quite a bit more complex than the RAM model, and so they can be difficult to work with. Moreover, RAM-model analyses are usually excellent predictors of performance on actual machines.

Although it is often straightforward to analyze an algorithm in the RAM model, sometimes it can be quite a challenge. You might need to employ mathematical tools such as combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. Because an algorithm might behave differently for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

Analysis of insertion sort

How long does the INSERTION-SORT procedure take? One way to tell would be for you to run it on your computer and time how long it takes to run. Of course, you'd

first have to implement it in a real programming language, since you cannot run our pseudocode directly. What would such a timing test tell you? You would find out how long insertion sort takes to run on your particular computer, on that particular input, under the particular implementation that you created, with the particular compiler or interpreter that you ran, with the particular libraries that you linked in, and with the particular background tasks that were running on your computer concurrently with your timing test (such as checking for incoming information over a network). If you run insertion sort again on your computer with the same input, you might even get a different timing result. From running just one implementation of insertion sort on just one computer and on just one input, what would you be able to determine about insertion sort's running time if you were to give it a different input, if you were to run it on a different computer, or if you were to implement it in a different programming language? Not much. We need a way to predict, given a new input, how long insertion sort will take.

Instead of timing a run, or even several runs, of insertion sort, we can determine how long it takes by analyzing the algorithm itself. We'll examine how many times it executes each line of pseudocode and how long each line of pseudocode takes to run. We'll first come up with a precise but complicated formula for the running time. Then, we'll distill the important part of the formula using a convenient notation that can help us compare the running times of different algorithms for the same problem.

How do we analyze insertion sort? First, let's acknowledge that the running time depends on the input. You shouldn't be terribly surprised that sorting a thousand numbers takes longer than sorting three numbers. Moreover, insertion sort can take different amounts of time to sort two input arrays of the same size, depending on how nearly sorted they already are. Even though the running time can depend on many features of the input, we'll focus on the one that has been shown to have the greatest effect, namely the size of the input, and describe the running time of a program as a function of the size of its input. To do so, we need to define the terms "running time" and "input size" more carefully. We also need to be clear about whether we are discussing the running time for an input that elicits the worst-case behavior, the best-case behavior, or some other case.

The best notion for ***input size*** depends on the problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the *number of items in the input*—for example, the number n of items being sorted. For many other problems, such as multiplying two integers, the best measure of input size is the *total number of bits* needed to represent the input in ordinary binary notation. Sometimes it is more appropriate to describe the size of the input with more than just one number. For example, if the input to an algorithm is a graph, we usually characterize the input size by both the number

of vertices and the number of edges in the graph. We'll indicate which input size measure is being used with each problem we study.

The ***running time*** of an algorithm on a particular input is the number of instructions and data accesses executed. How we account for these costs should be independent of any particular computer, but within the framework of the RAM model. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line might take more or less time than another line, but we'll assume that each execution of the kth line takes c_k time, where c_k is a constant. This viewpoint is in keeping with the RAM model, and it also reflects how the pseudocode would be implemented on most actual computers.[10]

Let's analyze the INSERTION-SORT procedure. As promised, we'll start by devising a precise formula that uses the input size and all the statement costs c_k. This formula turns out to be messy, however. We'll then switch to a simpler notation that is more concise and easier to use. This simpler notation makes clear how to compare the running times of algorithms, especially as the size of the input increases.

To analyze the INSERTION-SORT procedure, let's view it on the following page with the time cost of each statement and the number of times each statement is executed. For each $i = 2, 3, \ldots, n$, let t_i denote the number of times the **while** loop test in line 5 is executed for that value of i. When a **for** or **while** loop exits in the usual way—because the test in the loop header comes up FALSE—the test is executed one time more than the loop body. Because comments are not executable statements, assume that they take no time.

The running time of the algorithm is the sum of running times for each statement executed. A statement that takes c_k steps to execute and executes m times contributes $c_k m$ to the total running time.[11] We usually denote the running time of an algorithm on an input of size n by $T(n)$. To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of the *cost* and *times* columns, obtaining

---

[10] There are some subtleties here. Computational steps that we specify in English are often variants of a procedure that requires more than just a constant amount of time. For example, in the RADIX-SORT procedure on page 213, one line reads "use a stable sort to sort array A on digit i," which, as we shall see, takes more than a constant amount of time. Also, although a statement that calls a subroutine takes only constant time, the subroutine itself, once invoked, may take more. That is, we separate the process of ***calling*** the subroutine—passing parameters to it, etc.—from the process of ***executing*** the subroutine.

[11] This characteristic does not necessarily hold for a resource such as memory. A statement that references m words of memory and is executed n times does not necessarily reference mn distinct words of memory.

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $\quad key = A[i]$	c_2	$n-1$
3 $\quad$ // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $\quad j = i - 1$	c_4	$n-1$
5 $\quad$ **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $\quad\quad A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7 $\quad\quad j = j - 1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
8 $\quad A[j+1] = key$	c_8	$n-1$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n}(t_i - 1) \\ &\quad + c_7 \sum_{i=2}^{n}(t_i - 1) + c_8(n-1) \,. \end{aligned}$$

Even for inputs of a given size, an algorithm's running time may depend on *which* input of that size is given. For example, in INSERTION-SORT, the best case occurs when the array is already sorted. In this case, each time that line 5 executes, the value of *key*—the value originally in $A[i]$—is already greater than or equal to all values in $A[1:i-1]$, so that the **while** loop of lines 5–7 always exits upon the first test in line 5. Therefore, we have that $t_i = 1$ for $i = 2, 3, \ldots, n$, and the best-case running time is given by

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \,. \end{aligned} \tag{2.1}$$

We can express this running time as $an + b$ for *constants* a and b that depend on the statement costs c_k (where $a = c_1 + c_2 + c_4 + c_5 + c_8$ and $b = c_2 + c_4 + c_5 + c_8$). The running time is thus a ***linear function*** of n.

The worst case arises when the array is in reverse sorted order—that is, it starts out in decreasing order. The procedure must compare each element $A[i]$ with each element in the entire sorted subarray $A[1:i-1]$, and so $t_i = i$ for $i = 2, 3, \ldots, n$. (The procedure finds that $A[j] > key$ every time in line 5, and the **while** loop exits only when j reaches 0.) Noting that

$$\begin{aligned} \sum_{i=2}^{n} i &= \left(\sum_{i=1}^{n} i \right) - 1 \\ &= \frac{n(n+1)}{2} - 1 \quad \text{(by equation (A.2) on page 1141)} \end{aligned}$$

and

$$\begin{aligned}\sum_{i=2}^{n}(i-1) &= \sum_{i=1}^{n-1} i \\ &= \frac{n(n-1)}{2} \quad \text{(again, by equation (A.2))} ,\end{aligned}$$

we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned}T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \qquad (2.2)\end{aligned}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_k (now, $a = c_5/2 + c_6/2 + c_7/2$, $b = c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8$, and $c = -(c_2 + c_4 + c_5 + c_8)$). The running time is thus a ***quadratic function*** of n.

Typically, as in insertion sort, the running time of an algorithm is fixed for a given input, although we'll also see some interesting "randomized" algorithms whose behavior can vary even for a fixed input.

Worst-case and average-case analysis

Our analysis of insertion sort looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. For the remainder of this book, though, we'll usually (but not always) concentrate on finding only the ***worst-case running time***, that is, the longest running time for *any* input of size n. Why? Here are three reasons:

- The worst-case running time of an algorithm gives an upper bound on the running time for *any* input. If you know it, then you have a guarantee that the algorithm never takes any longer. You need not make some educated guess about the running time and hope that it never gets much worse. This feature is especially important for real-time computing, in which operations must complete by a deadline.
- For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm's worst case often occurs when the information is not present in the database. In some applications, searches for absent information may be frequent.

- The "average case" is often roughly as bad as the worst case. Suppose that you run insertion sort on an array of n randomly chosen numbers. How long does it take to determine where in subarray $A[1:i-1]$ to insert element $A[i]$? On average, half the elements in $A[1:i-1]$ are less than $A[i]$, and half the elements are greater. On average, therefore, $A[i]$ is compared with just half of the subarray $A[1:i-1]$, and so t_i is about $i/2$. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

In some particular cases, we'll be interested in the ***average-case*** running time of an algorithm. We'll see the technique of ***probabilistic analysis*** applied to various algorithms throughout this book. The scope of average-case analysis is limited, because it may not be apparent what constitutes an "average" input for a particular problem. Often, we'll assume that all inputs of a given size are equally likely. In practice, this assumption may be violated, but we can sometimes use a ***randomized algorithm***, which makes random choices, to allow a probabilistic analysis and yield an ***expected*** running time. We explore randomized algorithms more in Chapter 5 and in several other subsequent chapters.

Order of growth

In order to ease our analysis of the INSERTION-SORT procedure, we used some simplifying abstractions. First, we ignored the actual cost of each statement, using the constants c_k to represent these costs. Still, the best-case and worst-case running times in equations (2.1) and (2.2) are rather unwieldy. The constants in these expressions give us more detail than we really need. That's why we also expressed the best-case running time as $an + b$ for constants a and b that depend on the statement costs c_k and why we expressed the worst-case running time as $an^2 + bn + c$ for constants a, b, and c that depend on the statement costs. We thus ignored not only the actual statement costs, but also the abstract costs c_k.

Let's now make one more simplifying abstraction: it is the ***rate of growth***, or ***order of growth***, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large values of n. We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. For insertion sort's worst-case running time, when we ignore the lower-order terms and the leading term's constant coefficient, only the factor of n^2 from the leading term remains. That factor, n^2, is by far the most important part of the running time. For example, suppose that an algorithm implemented on a particular machine takes $n^2/100 + 100n + 17$ microseconds on an input of size n. Although the coefficients of $1/100$ for the n^2 term and 100 for the n term differ by four orders of magnitude, the $n^2/100$ term domi-

nates the $100n$ term once n exceeds 10,000. Although 10,000 might seem large, it is smaller than the population of an average town. Many real-world problems have much larger input sizes.

To highlight the order of growth of the running time, we have a special notation that uses the Greek letter Θ (theta). We write that insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of n-squared" or just "theta n-squared"). We also write that insertion sort has a best-case running time of $\Theta(n)$ ("theta of n" or "theta n"). For now, think of Θ-notation as saying "roughly proportional when n is large," so that $\Theta(n^2)$ means "roughly proportional to n^2 when n is large" and $\Theta(n)$ means "roughly proportional to n when n is large" We'll use Θ-notation informally in this chapter and define it precisely in Chapter 3.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth. Due to constant factors and lower-order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth. But on large enough inputs, an algorithm whose worst-case running time is $\Theta(n^2)$, for example, takes less time in the worst case than an algorithm whose worst-case running time is $\Theta(n^3)$. Regardless of the constants hidden by the Θ-notation, there is always some number, say n_0, such that for all input sizes $n \geq n_0$, the $\Theta(n^2)$ algorithm beats the $\Theta(n^3)$ algorithm in the worst case.

Exercises

2.2-1

Express the function $n^3/1000 + 100n^2 - 100n + 3$ in terms of Θ-notation.

2.2-2

Consider sorting n numbers stored in array $A[1:n]$ by first finding the smallest element of $A[1:n]$ and exchanging it with the element in $A[1]$. Then find the smallest element of $A[2:n]$, and exchange it with $A[2]$. Then find the smallest element of $A[3:n]$, and exchange it with $A[3]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the worst-case running time of selection sort in Θ-notation. Is the best-case running time any better?

2.2-3

Consider linear search again (see Exercise 2.1-4). How many elements of the input array need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case?

Using Θ-notation, give the average-case and worst-case running times of linear search. Justify your answers.

2.2-4
How can you modify any sorting algorithm to have a good best-case running time?

2.3 Designing algorithms

You can choose from a wide range of algorithm design techniques. Insertion sort uses the ***incremental*** method: for each element $A[i]$, insert it into its proper place in the subarray $A[1:i]$, having already sorted the subarray $A[1:i-1]$.

This section examines another design method, known as "divide-and-conquer," which we explore in more detail in Chapter 4. We'll use divide-and-conquer to design a sorting algorithm whose worst-case running time is much less than that of insertion sort. One advantage of using an algorithm that follows the divide-and-conquer method is that analyzing its running time is often straightforward, using techniques that we'll explore in Chapter 4.

2.3.1 The divide-and-conquer method

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they ***recurse*** (call themselves) one or more times to handle closely related subproblems. These algorithms typically follow the ***divide-and-conquer*** method: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

In the divide-and-conquer method, if the problem is small enough—the ***base case***—you just solve it directly without recursing. Otherwise—the ***recursive case***—you perform three characteristic steps:

Divide the problem into one or more subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

Combine the subproblem solutions to form a solution to the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer method. In each step, it sorts a subarray $A[p:r]$, starting with the entire array $A[1:n]$ and recursing down to smaller and smaller subarrays. Here is how merge sort operates:

Divide the subarray $A[p:r]$ to be sorted into two adjacent subarrays, each of half the size. To do so, compute the midpoint q of $A[p:r]$ (taking the average of p and r), and divide $A[p:r]$ into subarrays $A[p:q]$ and $A[q+1:r]$.

Conquer by sorting each of the two subarrays $A[p:q]$ and $A[q+1:r]$ recursively using merge sort.

Combine by merging the two sorted subarrays $A[p:q]$ and $A[q+1:r]$ back into $A[p:r]$, producing the sorted answer.

The recursion "bottoms out"—it reaches the base case—when the subarray $A[p:r]$ to be sorted has just 1 element, that is, when p equals r. As we noted in the initialization argument for INSERTION-SORT's loop invariant, a subarray comprising just a single element is always sorted.

The key operation of the merge sort algorithm occurs in the "combine" step, which merges two adjacent, sorted subarrays. The merge operation is performed by the auxiliary procedure MERGE(A, p, q, r) on the following page, where A is an array and p, q, and r are indices into the array such that $p \le q < r$. The procedure assumes that the adjacent subarrays $A[p:q]$ and $A[q+1:r]$ were already recursively sorted. It ***merges*** the two sorted subarrays to form a single sorted subarray that replaces the current subarray $A[p:r]$.

To understand how the MERGE procedure works, let's return to our card-playing motif. Suppose that you have two piles of cards face up on a table. Each pile is sorted, with the smallest-value cards on top. You wish to merge the two piles into a single sorted output pile, which is to be face down on the table. The basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile—which exposes a new top card—and placing this card face down onto the output pile. Repeat this step until one input pile is empty, at which time you can just take the remaining input pile and flip over the entire pile, placing it face down onto the output pile.

Let's think about how long it takes to merge two sorted piles of cards. Each basic step takes constant time, since you are comparing just the two top cards. If the two sorted piles that you start with each have $n/2$ cards, then the number of basic steps is at least $n/2$ (since in whichever pile was emptied, every card was found to be smaller than some card from the other pile) and at most n (actually, at most $n-1$, since after $n-1$ basic steps, one of the piles must be empty). With each basic step taking constant time and the total number of basic steps being between $n/2$ and n, we can say that merging takes time roughly proportional to n. That is, merging takes $\Theta(n)$ time.

In detail, the MERGE procedure works as follows. It copies the two subarrays $A[p:q]$ and $A[q+1:r]$ into temporary arrays L and R ("left" and "right"), and then it merges the values in L and R back into $A[p:r]$. Lines 1 and 2 compute the lengths n_L and n_R of the subarrays $A[p:q]$ and $A[q+1:r]$, respectively. Then

```
MERGE(A, p, q, r)
1  n_L = q - p + 1          // length of A[p : q]
2  n_R = r - q              // length of A[q + 1 : r]
3  let L[0 : n_L - 1] and R[0 : n_R - 1] be new arrays
4  for i = 0 to n_L - 1     // copy A[p : q] into L[0 : n_L - 1]
5      L[i] = A[p + i]
6  for j = 0 to n_R - 1     // copy A[q + 1 : r] into R[0 : n_R - 1]
7      R[j] = A[q + j + 1]
8  i = 0                    // i indexes the smallest remaining element in L
9  j = 0                    // j indexes the smallest remaining element in R
10 k = p                    // k indexes the location in A to fill
11 // As long as each of the arrays L and R contains an unmerged element,
   //     copy the smallest unmerged element back into A[p : r].
12 while i < n_L and j < n_R
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
18     k = k + 1
19 // Having gone through one of L and R entirely, copy the
   //     remainder of the other to the end of A[p : r].
20 while i < n_L
21     A[k] = L[i]
22     i = i + 1
23     k = k + 1
24 while j < n_R
25     A[k] = R[j]
26     j = j + 1
27     k = k + 1
```

line 3 creates arrays $L[0:n_L - 1]$ and $R[0:n_R - 1]$ with respective lengths n_L and n_R.[12] The **for** loop of lines 4–5 copies the subarray $A[p:q]$ into L, and the **for** loop of lines 6–7 copies the subarray $A[q+1:r]$ into R.

Lines 8–18, illustrated in Figure 2.3, perform the basic steps. The **while** loop of lines 12–18 repeatedly identifies the smallest value in L and R that has yet to

[12] This procedure is the rare case that uses both 1-origin indexing (for array A) and 0-origin indexing (for arrays L and R). Using 0-origin indexing for L and R makes for a simpler loop invariant in Exercise 2.3-3.

Figure 2.3 The operation of the **while** loop in lines 8–18 in the call MERGE$(A, 9, 12, 16)$, when the subarray $A[9:16]$ contains the values $\langle 2, 4, 6, 7, 1, 2, 3, 5\rangle$. After allocating and copying into the arrays L and R, the array L contains $\langle 2, 4, 6, 7\rangle$, and the array R contains $\langle 1, 2, 3, 5\rangle$. Tan positions in A contain their final values, and tan positions in L and R contain values that have yet to be copied back into A. Taken together, the tan positions always comprise the values originally in $A[9:16]$. Blue positions in A contain values that will be copied over, and dark positions in L and R contain values that have already been copied back into A. **(a)–(g)** The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–18. At the point in part (g), all values in R have been copied back into A (indicated by j equaling the length of R), and so the **while** loop in lines 12–18 terminates. **(h)** The arrays and indices at termination. The **while** loops of lines 20–23 and 24–27 copied back into A the remaining values in L and R, which are the largest values originally in $A[9:16]$. Here, lines 20–23 copied $L[2:3]$ into $A[15:16]$, and because all values in R had already been copied back into A, the **while** loop of lines 24–27 iterated 0 times. At this point, the subarray in $A[9:16]$ is sorted.

be copied back into $A[p:r]$ and copies it back in. As the comments indicate, the index k gives the position of A that is being filled in, and the indices i and j give the positions in L and R, respectively, of the smallest remaining values. Eventually, either all of L or all of R is copied back into $A[p:r]$, and this loop terminates. If the loop terminates because all of R has been copied back, that is, because j equals n_R, then i is still less than n_L, so that some of L has yet to be copied back, and these values are the greatest in both L and R. In this case, the **while** loop of lines 20–23 copies these remaining values of L into the last few positions of $A[p:r]$. Because j equals n_R, the **while** loop of lines 24–27 iterates 0 times. If instead the **while** loop of lines 12–18 terminates because i equals n_L, then all of L has already been copied back into $A[p:r]$, and the **while** loop of lines 24–27 copies the remaining values of R back into the end of $A[p:r]$.

To see that the MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$,[13] observe that each of lines 1–3 and 8–10 takes constant time, and the **for** loops of lines 4–7 take $\Theta(n_L + n_R) = \Theta(n)$ time.[14] To account for the three **while** loops of lines 12–18, 20–23, and 24–27, observe that each iteration of these loops copies exactly one value from L or R back into A and that every value is copied back into A exactly once. Therefore, these three loops together make a total of n iterations. Since each iteration of each of the three loops takes constant time, the total time spent in these three loops is $\Theta(n)$.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT(A, p, r) on the facing page sorts the elements in the subarray $A[p:r]$. If p equals r, the subarray has just 1 element and is therefore already sorted. Otherwise, we must have $p < r$, and MERGE-SORT runs the divide, conquer, and combine steps. The divide step simply computes an index q that partitions $A[p:r]$ into two adjacent subarrays: $A[p:q]$, containing $\lceil n/2 \rceil$ elements, and $A[q+1:r]$, containing $\lfloor n/2 \rfloor$ elements.[15] The initial call MERGE-SORT$(A, 1, n)$ sorts the entire array $A[1:n]$.

Figure 2.4 illustrates the operation of the procedure for $n = 8$, showing also the sequence of divide and merge steps. The algorithm recursively divides the array down to 1-element subarrays. The combine steps merge pairs of 1-element subar-

---

[13] If you're wondering where the "+1" comes from, imagine that $r = p + 1$. Then the subarray $A[p:r]$ consists of two elements, and $r - p + 1 = 2$.

[14] Chapter 3 shows how to formally interpret equations containing Θ-notation.

[15] The expression $\lceil x \rceil$ denotes the least integer greater than or equal to x, and $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x. These notations are defined in Section 3.3. The easiest way to verify that setting q to $\lfloor (p+r)/2 \rfloor$ yields subarrays $A[p:q]$ and $A[q+1:r]$ of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively, is to examine the four cases that arise depending on whether each of p and r is odd or even.

```
MERGE-SORT(A, p, r)
if p ≥ r                          // zero or one element?
    return
q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
MERGE-SORT(A, p, q)               // recursively sort A[p : q]
MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
// Merge A[p : q] and A[q + 1 : r] into A[p : r].
MERGE(A, p, q, r)
```

rays to form sorted subarrays of length 2, merges those to form sorted subarrays of length 4, and merges those to form the final sorted subarray of length 8. If n is not an exact power of 2, then some divide steps create subarrays whose lengths differ by 1. (For example, when dividing a subarray of length 7, one subarray has length 4 and the other has length 3.) Regardless of the lengths of the two subarrays being merged, the time to merge a total of n items is $\Theta(n)$.

2.3.2 Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call, you can often describe its running time by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size n in terms of the running time of the same algorithm on smaller inputs. You can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic method. As we did for insertion sort, let $T(n)$ be the worst-case running time on a problem of size n. If the problem size is small enough, say $n < n_0$ for some constant $n_0 > 0$, the straightforward solution takes constant time, which we write as $\Theta(1)$.[16] Suppose that the division of the problem yields a subproblems, each with size n/b, that is, $1/b$ the size of the original. For merge sort, both a and b are 2, but we'll see other divide-and-conquer algorithms in which $a \neq b$. It takes $T(n/b)$ time to solve one subproblem of size n/b, and so it takes $aT(n/b)$ time to solve all a of them. If it takes $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

[16] If you're wondering where $\Theta(1)$ comes from, think of it this way. When we say that $n^2/100$ is $\Theta(n^2)$, we are ignoring the coefficient $1/100$ of the factor n^2. Likewise, when we say that a constant c is $\Theta(1)$, we are ignoring the coefficient c of the factor 1 (which you can also think of as n^0).

Figure 2.4 The operation of merge sort on the array A with length 8 that initially contains the sequence $\langle 12, 3, 7, 9, 14, 6, 11, 2 \rangle$. The indices p, q, and r into each subarray appear above their values. Numbers in italics indicate the order in which the MERGE-SORT and MERGE procedures are called following the initial call of MERGE-SORT$(A, 1, 8)$.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 \,, \\ D(n) + aT(n/b) + C(n) & \text{otherwise} \,. \end{cases}$$

Chapter 4 shows how to solve common recurrences of this form.

Sometimes, the n/b size of the divide step isn't an integer. For example, the MERGE-SORT procedure divides a problem of size n into subproblems of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Since the difference between $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ is at most 1,

which for large n is much smaller than the effect of dividing n by 2, we'll squint a little and just call them both size $n/2$. As Chapter 4 will discuss, this simplification of ignoring floors and ceilings does not generally affect the order of growth of a solution to a divide-and-conquer recurrence.

Another convention we'll adopt is to omit a statement of the base cases of the recurrence, which we'll also discuss in more detail in Chapter 4. The reason is that the base cases are pretty much always $T(n) = \Theta(1)$ if $n < n_0$ for some constant $n_0 > 0$. That's because the running time of an algorithm on an input of constant size is constant. We save ourselves a lot of extra writing by adopting this convention.

Analysis of merge sort

Here's how to set up the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

Conquer: Recursively solving two subproblems, each of size $n/2$, contributes $2T(n/2)$ to the running time (ignoring the floors and ceilings, as we discussed).

Combine: Since the MERGE procedure on an n-element subarray takes $\Theta(n)$ time, we have $C(n) = \Theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of n. That is, it is roughly proportional to n when n is large, and so merge sort's dividing and combining times together are $\Theta(n)$. Adding $\Theta(n)$ to the $2T(n/2)$ term from the conquer step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = 2T(n/2) + \Theta(n) . \tag{2.3}$$

Chapter 4 presents the "master theorem," which shows that $T(n) = \Theta(n \lg n)$.[17] Compared with insertion sort, whose worst-case running time is $\Theta(n^2)$, merge sort trades away a factor of n for a factor of $\lg n$. Because the logarithm function grows more slowly than any linear function, that's a good trade. For large enough inputs, merge sort, with its $\Theta(n \lg n)$ worst-case running time, outperforms insertion sort, whose worst-case running time is $\Theta(n^2)$.

---

[17] The notation $\lg n$ stands for $\log_2 n$, although the base of the logarithm doesn't matter here, but as computer scientists, we like logarithms base 2. Section 3.3 discusses other standard notation.

We do not need the master theorem, however, to understand intuitively why the solution to recurrence (2.3) is $T(n) = \Theta(n \lg n)$. For simplicity, assume that n is an exact power of 2 and that the implicit base case is $n = 1$. Then recurrence (2.3) is essentially

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 , \\ 2T(n/2) + c_2 n & \text{if } n > 1 , \end{cases} \tag{2.4}$$

where the constant $c_1 > 0$ represents the time required to solve a problem of size 1, and $c_2 > 0$ is the time per array element of the divide and combine steps.[18]

Figure 2.5 illustrates one way of figuring out the solution to recurrence (2.4). Part (a) of the figure shows $T(n)$, which part (b) expands into an equivalent tree representing the recurrence. The $c_2 n$ term denotes the cost of dividing and combining at the top level of recursion, and the two subtrees of the root are the two smaller recurrences $T(n/2)$. Part (c) shows this process carried one step further by expanding $T(n/2)$. The cost for dividing and combining at each of the two nodes at the second level of recursion is $c_2 n/2$. Continue to expand each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1, each with a cost of c_1. Part (d) shows the resulting ***recursion tree***.

Next, add the costs across each level of the tree. The top level has total cost $c_2 n$, the next level down has total cost $c_2(n/2) + c_2(n/2) = c_2 n$, the level after that has total cost $c_2(n/4) + c_2(n/4) + c_2(n/4) + c_2(n/4) = c_2 n$, and so on. Each level has twice as many nodes as the level above, but each node contributes only half the cost of a node from the level above. From one level to the next, doubling and halving cancel each other out, so that the cost across each level is the same: $c_2 n$. In general, the level that is i levels below the top has 2^i nodes, each contributing a cost of $c_2(n/2^i)$, so that the ith level below the top has total cost $2^i \cdot c_2(n/2^i) = c_2 n$. The bottom level has n nodes, each contributing a cost of c_1, for a total cost of $c_1 n$.

The total number of levels of the recursion tree in Figure 2.5 is $\lg n + 1$, where n is the number of leaves, corresponding to the input size. An informal inductive argument justifies this claim. The base case occurs when $n = 1$, in which case the tree has only 1 level. Since $\lg 1 = 0$, we have that $\lg n + 1$ gives the correct number of levels. Now assume as an inductive hypothesis that the number of levels of a recursion tree with 2^i leaves is $\lg 2^i + 1 = i + 1$ (since for any value of i, we have that $\lg 2^i = i$). Because we assume that the input size is an exact power of 2, the next input size to consider is 2^{i+1}. A tree with $n = 2^{i+1}$ leaves has 1 more

[18] It is unlikely that c_1 is exactly the time to solve problems of size 1 and that $c_2 n$ is exactly the time of the divide and combine steps. We'll look more closely at bounding recurrences in Chapter 4, where we'll be more careful about this kind of detail.

Figure 2.5 How to construct a recursion tree for the recurrence (2.4). Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels. Each level above the leaves contributes a total cost of c_2n, and the leaf level contributes c_1n. The total cost, therefore, is $c_2n \lg n + c_1n = \Theta(n \lg n)$.

level than a tree with 2^i leaves, and so the total number of levels is $(i + 1) + 1 = \lg 2^{i+1} + 1$.

To compute the total cost represented by the recurrence (2.4), simply add up the costs of all the levels. The recursion tree has $\lg n + 1$ levels. The levels above the leaves each cost $c_2 n$, and the leaf level costs $c_1 n$, for a total cost of $c_2 n \lg n + c_1 n = \Theta(n \lg n)$.

Exercises

2.3-1

Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence $\langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

2.3-2

The test in line 1 of the MERGE-SORT procedure reads "**if** $p \geq r$" rather than "**if** $p \neq r$." If MERGE-SORT is called with $p > r$, then the subarray $A[p : r]$ is empty. Argue that as long as the initial call of MERGE-SORT$(A, 1, n)$ has $n \geq 1$, the test "**if** $p \neq r$" suffices to ensure that no recursive call has $p > r$.

2.3-3

State a loop invariant for the **while** loop of lines 12–18 of the MERGE procedure. Show how to use it, along with the **while** loops of lines 20–23 and 24–27, to prove that the MERGE procedure is correct.

2.3-4

Use mathematical induction to show that when $n \geq 2$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2 , \\ 2T(n/2) + n & \text{if } n > 2 \end{cases}$$

is $T(n) = n \lg n$.

2.3-5

You can also think of insertion sort as a recursive algorithm. In order to sort $A[1 : n]$, recursively sort the subarray $A[1 : n - 1]$ and then insert $A[n]$ into the sorted subarray $A[1 : n - 1]$. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

2.3-6

Referring back to the searching problem (see Exercise 2.1-4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further

consideration. The ***binary search*** algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

2.3-7
The **while** loop of lines 5–7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1 : j - 1]$. What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

2.3-8
Describe an algorithm that, given a set S of n integers and another integer x, determines whether S contains two elements that sum to exactly x. Your algorithm should take $\Theta(n \lg n)$ time in the worst case.

Problems

2-1 Insertion sort on small arrays in merge sort
Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus it makes sense to ***coarsen*** the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

a. Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.

b. Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time.

c. Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?

d. How should you choose k in practice?

2-2 *Correctness of bubblesort*

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order. The procedure BUBBLESORT sorts array $A[1:n]$.

```
BUBBLESORT(A, n)
1  for i = 1 to n − 1
2      for j = n downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

a. Let A' denote the array A after BUBBLESORT(A, n) is executed. To prove that BUBBLESORT is correct, you need to prove that it terminates and that

$$A'[1] \le A'[2] \le \cdots \le A'[n] \,. \tag{2.5}$$

In order to show that BUBBLESORT actually sorts, what else do you need to prove?

The next two parts prove inequality (2.5).

b. State precisely a loop invariant for the **for** loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop-invariant proof presented in this chapter.

c. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1–4 that allows you to prove inequality (2.5). Your proof should use the structure of the loop-invariant proof presented in this chapter.

d. What is the worst-case running time of BUBBLESORT? How does it compare with the running time of INSERTION-SORT?

2-3 *Correctness of Horner's rule*

You are given the coefficents $a_0, a_1, a_2, \ldots, a_n$ of a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \,, \end{aligned}$$

and you want to evaluate this polynomial for a given value of x. ***Horner's rule*** says to evaluate the polynomial according to this parenthesization:

$$P(x) = a_0 + x\Big(a_1 + x\left(a_2 + \cdots + x(a_{n-1} + xa_n)\cdots\right)\Big) .$$

The procedure HORNER implements Horner's rule to evaluate $P(x)$, given the coefficients $a_0, a_1, a_2, \ldots, a_n$ in an array $A[0:n]$ and the value of x.

```
HORNER(A, n, x)
1  p = 0
2  for i = n downto 0
3      p = A[i] + x · p
4  return p
```

a. In terms of Θ-notation, what is the running time of this procedure?

b. Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare with HORNER?

c. Consider the following loop invariant for the procedure HORNER:

> At the start of each iteration of the **for** loop of lines 2–3,
>
> $$p = \sum_{k=0}^{n-(i+1)} A[k+i+1] \cdot x^k .$$

Interpret a summation with no terms as equaling 0. Following the structure of the loop-invariant proof presented in this chapter, use this loop invariant to show that, at termination, $p = \sum_{k=0}^{n} A[k] \cdot x^k$.

2-4 Inversions

Let $A[1:n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A.

a. List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

b. What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

d. Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint:* Modify merge sort.)

Chapter notes

In 1968, Knuth published the first of three volumes with the general title *The Art of Computer Programming* [259, 260, 261]. The first volume ushered in the modern study of computer algorithms with a focus on the analysis of running time. The full series remains an engaging and worthwhile reference for many of the topics presented here. According to Knuth, the word "algorithm" is derived from the name "al-Khowârizmî," a ninth-century Persian mathematician.

Aho, Hopcroft, and Ullman [5] advocated the asymptotic analysis of algorithms —using notations that Chapter 3 introduces, including Θ-notation—as a means of comparing relative performance. They also popularized the use of recurrence relations to describe the running times of recursive algorithms.

Knuth [261] provides an encyclopedic treatment of many sorting algorithms. His comparison of sorting algorithms (page 381) includes exact step-counting analyses, like the one we performed here for insertion sort. Knuth's discussion of insertion sort encompasses several variations of the algorithm. The most important of these is Shell's sort, introduced by D. L. Shell, which uses insertion sort on periodic subarrays of the input to produce a faster sorting algorithm.

Merge sort is also described by Knuth. He mentions that a mechanical collator capable of merging two decks of punched cards in a single pass was invented in 1938. J. von Neumann, one of the pioneers of computer science, apparently wrote a program for merge sort on the EDVAC computer in 1945.

The early history of proving programs correct is described by Gries [200], who credits P. Naur with the first article in this field. Gries attributes loop invariants to R. W. Floyd. The textbook by Mitchell [329] is a good reference on how to prove programs correct.

3 Characterizing Running Times

The order of growth of the running time of an algorithm, defined in Chapter 2, gives a simple way to characterize the algorithm's efficiency and also allows us to compare it with alternative algorithms. Once the input size n becomes large enough, merge sort, with its $\Theta(n \lg n)$ worst-case running time, beats insertion sort, whose worst-case running time is $\Theta(n^2)$. Although we can sometimes determine the exact running time of an algorithm, as we did for insertion sort in Chapter 2, the extra precision is rarely worth the effort of computing it. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.

When we look at input sizes large enough to make relevant only the order of growth of the running time, we are studying the ***asymptotic*** efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input *in the limit*, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient is the best choice for all but very small inputs.

This chapter gives several standard methods for simplifying the asymptotic analysis of algorithms. The next section presents informally the three most commonly used types of "asymptotic notation," of which we have already seen an example in Θ-notation. It also shows one way to use these asymptotic notations to reason about the worst-case running time of insertion sort. Then we look at asymptotic notations more formally and present several notational conventions used throughout this book. The last section reviews the behavior of functions that commonly arise when analyzing algorithms.

3.1 O-notation, Ω-notation, and Θ-notation

When we analyzed the worst-case running time of insertion sort in Chapter 2, we started with the complicated expression

$$\left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ - (c_2 + c_4 + c_5 + c_8) \; .$$

We then discarded the lower-order terms $(c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)n$ and $c_2 + c_4 + c_5 + c_8$, and we also ignored the coefficient $c_5/2 + c_6/2 + c_7/2$ of n^2. That left just the factor n^2, which we put into Θ-notation as $\Theta(n^2)$. We use this style to characterize running times of algorithms: discard the lower-order terms and the coefficient of the leading term, and use a notation that focuses on the rate of growth of the running time.

Θ-notation is not the only such "asymptotic notation." In this section, we'll see other forms of asymptotic notation as well. We start with intuitive looks at these notations, revisiting insertion sort to see how we can apply them. In the next section, we'll see the formal definitions of our asymptotic notations, along with conventions for using them.

Before we get into specifics, bear in mind that the asymptotic notations we'll see are designed so that they characterize functions in general. It so happens that the functions we are most interested in denote the running times of algorithms. But asymptotic notation can apply to functions that characterize some other aspect of algorithms (the amount of space they use, for example), or even to functions that have nothing whatsoever to do with algorithms.

O-notation

O-notation characterizes an *upper bound* on the asymptotic behavior of a function. In other words, it says that a function grows *no faster* than a certain rate, based on the highest-order term. Consider, for example, the function $7n^3 + 100n^2 - 20n + 6$. Its highest-order term is $7n^3$, and so we say that this function's rate of growth is n^3. Because this function grows no faster than n^3, we can write that it is $O(n^3)$. You might be surprised that we can also write that the function $7n^3 + 100n^2 - 20n + 6$ is $O(n^4)$. Why? Because the function grows more slowly than n^4, we are correct in saying that it grows no faster. As you might have guessed, this function is also $O(n^5)$, $O(n^6)$, and so on. More generally, it is $O(n^c)$ for any constant $c \geq 3$.

Ω-notation

Ω-notation characterizes a *lower bound* on the asymptotic behavior of a function. In other words, it says that a function grows *at least as fast* as a certain rate, based —as in O-notation—on the highest-order term. Because the highest-order term in the function $7n^3 + 100n^2 - 20n + 6$ grows at least as fast as n^3, this function is $\Omega(n^3)$. This function is also $\Omega(n^2)$ and $\Omega(n)$. More generally, it is $\Omega(n^c)$ for any constant $c \leq 3$.

Θ-notation

Θ-notation characterizes a *tight bound* on the asymptotic behavior of a function. It says that a function grows *precisely* at a certain rate, based—once again—on the highest-order term. Put another way, Θ-notation characterizes the rate of growth of the function to within a constant factor from above and to within a constant factor from below. These two constant factors need not be equal.

If you can show that a function is both $O(f(n))$ and $\Omega(f(n))$ for some function $f(n)$, then you have shown that the function is $\Theta(f(n))$. (The next section states this fact as a theorem.) For example, since the function $7n^3+100n^2-20n+6$ is both $O(n^3)$ and $\Omega(n^3)$, it is also $\Theta(n^3)$.

Example: Insertion sort

Let's revisit insertion sort and see how to work with asymptotic notation to characterize its $\Theta(n^2)$ worst-case running time without evaluating summations as we did in Chapter 2. Here is the INSERTION-SORT procedure once again:

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

What can we observe about how the pseudocode operates? The procedure has nested loops. The outer loop is a **for** loop that runs $n - 1$ times, regardless of the values being sorted. The inner loop is a **while** loop, but the number of iterations it makes depends on the values being sorted. The loop variable j starts at $i - 1$

Figure 3.1 The $\Omega(n^2)$ lower bound for insertion sort. If the first $n/3$ positions contain the $n/3$ largest values, each of these values must move through each of the middle $n/3$ positions, one position at a time, to end up somewhere in the last $n/3$ positions. Since each of $n/3$ values moves through at least each of $n/3$ positions, the time taken in this case is at least proportional to $(n/3)(n/3) = n^2/9$, or $\Omega(n^2)$.

and decreases by 1 in each iteration until either it reaches 0 or $A[j] \le key$. For a given value of i, the **while** loop might iterate 0 times, $i - 1$ times, or anywhere in between. The body of the **while** loop (lines 6–7) takes constant time per iteration of the **while** loop.

These observations suffice to deduce an $O(n^2)$ running time for any case of INSERTION-SORT, giving us a blanket statement that covers all inputs. The running time is dominated by the inner loop. Because each of the $n - 1$ iterations of the outer loop causes the inner loop to iterate at most $i - 1$ times, and because i is at most n, the total number of iterations of the inner loop is at most $(n-1)(n-1)$, which is less than n^2. Since each iteration of the inner loop takes constant time, the total time spent in the inner loop is at most a constant times n^2, or $O(n^2)$.

With a little creativity, we can also see that the worst-case running time of INSERTION-SORT is $\Omega(n^2)$. By saying that the worst-case running time of an algorithm is $\Omega(n^2)$, we mean that for every input size n above a certain threshold, there is at least one input of size n for which the algorithm takes at least cn^2 time, for some positive constant c. It does not necessarily mean that the algorithm takes at least cn^2 time for all inputs.

Let's now see why the worst-case running time of INSERTION-SORT is $\Omega(n^2)$. For a value to end up to the right of where it started, it must have been moved in line 6. In fact, for a value to end up k positions to the right of where it started, line 6 must have executed k times. As Figure 3.1 shows, let's assume that n is a multiple of 3 so that we can divide the array A into groups of $n/3$ positions. Suppose that in the input to INSERTION-SORT, the $n/3$ largest values occupy the first $n/3$ array positions $A[1:n/3]$. (It does not matter what relative order they have within the first $n/3$ positions.) Once the array has been sorted, each of these $n/3$ values ends up somewhere in the last $n/3$ positions $A[2n/3+1:n]$. For that to happen, each of these $n/3$ values must pass through each of the middle $n/3$ positions $A[n/3+1:2n/3]$. Each of these $n/3$ values passes through these middle

$n/3$ positions one position at a time, by at least $n/3$ executions of line 6. Because at least $n/3$ values have to pass through at least $n/3$ positions, the time taken by INSERTION-SORT in the worst case is at least proportional to $(n/3)(n/3) = n^2/9$, which is $\Omega(n^2)$.

Because we have shown that INSERTION-SORT runs in $O(n^2)$ time in all cases and that there is an input that makes it take $\Omega(n^2)$ time, we can conclude that the worst-case running time of INSERTION-SORT is $\Theta(n^2)$. It does not matter that the constant factors for upper and lower bounds might differ. What matters is that we have characterized the worst-case running time to within constant factors (discounting lower-order terms). This argument does not show that INSERTION-SORT runs in $\Theta(n^2)$ time in *all* cases. Indeed, we saw in Chapter 2 that the best-case running time is $\Theta(n)$.

Exercises

3.1-1
Modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3.

3.1-2
Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm from Exercise 2.2-2.

3.1-3
Suppose that α is a fraction in the range $0 < \alpha < 1$. Show how to generalize the lower-bound argument for insertion sort to consider an input in which the αn largest values start in the first αn positions. What additional restriction do you need to put on α? What value of α maximizes the number of times that the αn largest values must pass through each of the middle $(1 - 2\alpha)n$ array positions?

3.2 Asymptotic notation: formal definitions

Having seen asymptotic notation informally, let's get more formal. The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are typically the set $\mathbb{N}$ of natural numbers or the set $\mathbb{R}$ of real numbers. Such notations are convenient for describing a running-time function $T(n)$. This section defines the basic asymptotic notations and also introduces some common "proper" notational abuses.

Figure 3.2 Graphic examples of the O, Ω, and Θ notations. In each part, the value of n_0 shown is the minimum possible value, but any greater value also works. **(a)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(b)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$. **(c)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive.

O-notation

As we saw in Section 3.1, O-notation describes an ***asymptotic upper bound***. We use O-notation to give an upper bound on a function, to within a constant factor.

Here is the formal definition of O-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the *set of functions*

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}\ .^{1}$$

A function $f(n)$ belongs to the set $O(g(n))$ if there exists a positive constant c such that $f(n) \le cg(n)$ for sufficiently large n. Figure 3.2(a) shows the intuition behind O-notation. For all values n at and to the right of n_0, the value of the function $f(n)$ is on or below $cg(n)$.

The definition of $O(g(n))$ requires that every function $f(n)$ in the set $O(g(n))$ be ***asymptotically nonnegative***: $f(n)$ must be nonnegative whenever n is sufficiently large. (An ***asymptotically positive*** function is one that is positive for all

[1] Within set notation, a colon means "such that."

sufficiently large n.) Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $O(g(n))$ is empty. We therefore assume that every function used within O-notation is asymptotically nonnegative. This assumption holds for the other asymptotic notations defined in this chapter as well.

You might be surprised that we define O-notation in terms of sets. Indeed, you might expect that we would write "$f(n) \in O(g(n))$" to indicate that $f(n)$ belongs to the set $O(g(n))$. Instead, we usually write "$f(n) = O(g(n))$" and say "$f(n)$ is big-oh of $g(n)$" to express the same notion. Although it may seem confusing at first to abuse equality in this way, we'll see later in this section that doing so has its advantages.

Let's explore an example of how to use the formal definition of O-notation to justify our practice of discarding lower-order terms and ignoring the constant coefficient of the highest-order term. We'll show that $4n^2 + 100n + 500 = O(n^2)$, even though the lower-order terms have much larger coefficients than the leading term. We need to find positive constants c and n_0 such that $4n^2 + 100n + 500 \le cn^2$ for all $n \ge n_0$. Dividing both sides by n^2 gives $4 + 100/n + 500/n^2 \le c$. This inequality is satisfied for many choices of c and n_0. For example, if we choose $n_0 = 1$, then this inequality holds for $c = 604$. If we choose $n_0 = 10$, then $c = 19$ works, and choosing $n_0 = 100$ allows us to use $c = 5.05$.

We can also use the formal definition of O-notation to show that the function $n^3 - 100n^2$ does not belong to the set $O(n^2)$, even though the coefficient of n^2 is a large negative number. If we had $n^3 - 100n^2 = O(n^2)$, then there would be positive constants c and n_0 such that $n^3 - 100n^2 \le cn^2$ for all $n \ge n_0$. Again, we divide both sides by n^2, giving $n - 100 \le c$. Regardless of what value we choose for the constant c, this inequality does not hold for any value of $n > c + 100$.

Ω-notation

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$

Figure 3.2(b) shows the intuition behind Ω-notation. For all values n at or to the right of n_0, the value of $f(n)$ is on or above $cg(n)$.

We've already shown that $4n^2 + 100n + 500 = O(n^2)$. Now let's show that $4n^2 + 100n + 500 = \Omega(n^2)$. We need to find positive constants c and n_0 such that $4n^2 + 100n + 500 \ge cn^2$ for all $n \ge n_0$. As before, we divide both sides by n^2,

giving $4 + 100/n + 500/n^2 \geq c$. This inequality holds when n_0 is any positive integer and $c = 4$.

What if we had subtracted the lower-order terms from the $4n^2$ term instead of adding them? What if we had a small coefficient for the n^2 term? The function would still be $\Omega(n^2)$. For example, let's show that $n^2/100 - 100n - 500 = \Omega(n^2)$. Dividing by n^2 gives $1/100 - 100/n - 500/n^2 \geq c$. We can choose any value for n_0 that is at least 10,005 and find a positive value for c. For example, when $n_0 = 10{,}005$, we can choose $c = 2.49 \times 10^{-9}$. Yes, that's a tiny value for c, but it is positive. If we select a larger value for n_0, we can also increase c. For example, if $n_0 = 100{,}000$, then we can choose $c = 0.0089$. The higher the value of n_0, the closer to the coefficient $1/100$ we can choose c.

Θ-notation

We use Θ-notation for ***asymptotically tight bounds***. For a given function $g(n)$, we denote by $\Theta(g(n))$ ("theta of g of n") the set of functions

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\} .$$

Figure 3.2(c) shows the intuition behind Θ-notation. For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within constant factors.

The definitions of O-, Ω-, and Θ-notations lead to the following theorem, whose proof we leave as Exercise 3.2-4.

Theorem 3.1
For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■

We typically apply Theorem 3.1 to prove asymptotically tight bounds from asymptotic upper and lower bounds.

Asymptotic notation and running times

When you use asymptotic notation to characterize an algorithm's running time, make sure that the asymptotic notation you use is as precise as possible without overstating which running time it applies to. Here are some examples of using asymptotic notation properly and improperly to characterize running times.

Let's start with insertion sort. We can correctly say that insertion sort's worst-case running time is $O(n^2)$, $\Omega(n^2)$, and—due to Theorem 3.1—$\Theta(n^2)$. Although

all three ways to characterize the worst-case running times are correct, the $\Theta(n^2)$ bound is the most precise and hence the most preferred. We can also correctly say that insertion sort's best-case running time is $O(n)$, $\Omega(n)$, and $\Theta(n)$, again with $\Theta(n)$ the most precise and therefore the most preferred.

Here is what we *cannot* correctly say: insertion sort's running time is $\Theta(n^2)$. That is an overstatement because by omitting "worst-case" from the statement, we're left with a blanket statement covering all cases. The error here is that insertion sort does not run in $\Theta(n^2)$ time in all cases since, as we've seen, it runs in $\Theta(n)$ time in the best case. We can correctly say that insertion sort's running time is $O(n^2)$, however, because in all cases, its running time grows no faster than n^2. When we say $O(n^2)$ instead of $\Theta(n^2)$, there is no problem in having cases whose running time grows more slowly than n^2. Likewise, we cannot correctly say that insertion sort's running time is $\Theta(n)$, but we can say that its running time is $\Omega(n)$.

How about merge sort? Since merge sort runs in $\Theta(n \lg n)$ time in all cases, we can just say that its running time is $\Theta(n \lg n)$ without specifying worst-case, best-case, or any other case.

People occasionally conflate O-notation with Θ-notation by mistakenly using O-notation to indicate an asymptotically tight bound. They say things like "an $O(n \lg n)$-time algorithm runs faster than an $O(n^2)$-time algorithm." Maybe it does, maybe it doesn't. Since O-notation denotes only an asymptotic upper bound, that so-called $O(n^2)$-time algorithm might actually run in $\Theta(n)$ time. You should be careful to choose the appropriate asymptotic notation. If you want to indicate an asymptotically tight bound, use Θ-notation.

We typically use asymptotic notation to provide the simplest and most precise bounds possible. For example, if an algorithm has a running time of $3n^2 + 20n$ in all cases, we use asymptotic notation to write that its running time is $\Theta(n^2)$. Strictly speaking, we are also correct in writing that the running time is $O(n^3)$ or $\Theta(3n^2 + 20n)$. Neither of these expressions is as useful as writing $\Theta(n^2)$ in this case, however: $O(n^3)$ is less precise than $\Theta(n^2)$ if the running time is $3n^2 + 20n$, and $\Theta(3n^2 + 20n)$ introduces complexity that obscures the order of growth. By writing the simplest and most precise bound, such as $\Theta(n^2)$, we can categorize and compare different algorithms. Throughout the book, you will see asymptotic running times that are almost always based on polynomials and logarithms: functions such as n, $n \lg^2 n$, $n^2 \lg n$, or $n^{1/2}$. You will also see some other functions, such as exponentials, $\lg \lg n$, and $\lg^* n$ (see Section 3.3). It is usually fairly easy to compare the rates of growth of these functions. Problem 3-3 gives you good practice.

Asymptotic notation in equations and inequalities

Although we formally define asymptotic notation in terms of sets, we use the equal sign (=) instead of the set membership sign ($\in$) within formulas. For example, we wrote that $4n^2 + 100n + 500 = O(n^2)$. We might also write $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. How do we interpret such formulas?

When the asymptotic notation stands alone (that is, not within a larger formula) on the right-hand side of an equation (or inequality), as in $4n^2 + 100n + 500 = O(n^2)$, the equal sign means set membership: $4n^2 + 100n + 500 \in O(n^2)$. In general, however, when asymptotic notation appears in a formula, we interpret it as standing for some anonymous function that we do not care to name. For example, the formula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$, where $f(n) \in \Theta(n)$. In this case, we let $f(n) = 3n + 1$, which indeed belongs to $\Theta(n)$.

Using asymptotic notation in this manner can help eliminate inessential detail and clutter in an equation. For example, in Chapter 2 we expressed the worst-case running time of merge sort as the recurrence

$$T(n) = 2T(n/2) + \Theta(n) .$$

If we are interested only in the asymptotic behavior of $T(n)$, there is no point in specifying all the lower-order terms exactly, because they are all understood to be included in the anonymous function denoted by the term $\Theta(n)$.

The number of anonymous functions in an expression is understood to be equal to the number of times the asymptotic notation appears. For example, in the expression

$$\sum_{i=1}^{n} O(i) ,$$

there is only a single anonymous function (a function of i). This expression is thus *not* the same as $O(1) + O(2) + \cdots + O(n)$, which doesn't really have a clean interpretation.

In some cases, asymptotic notation appears on the left-hand side of an equation, as in

$$2n^2 + \Theta(n) = \Theta(n^2) .$$

Interpret such equations using the following rule: *No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid*. Thus, our example means that for *any* function $f(n) \in \Theta(n)$, there is *some* function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$ for all n. In other words, the right-hand side of an equation provides a coarser level of detail than the left-hand side.

We can chain together a number of such relationships, as in

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) \ . \end{aligned}$$

By the rules above, interpret each equation separately. The first equation says that there is *some* function $f(n) \in \Theta(n)$ such that $2n^2+3n+1 = 2n^2 + f(n)$ for all n. The second equation says that for *any* function $g(n) \in \Theta(n)$ (such as the $f(n)$ just mentioned), there is *some* function $h(n) \in \Theta(n^2)$ such that $2n^2 + g(n) = h(n)$ for all n. This interpretation implies that $2n^2 + 3n + 1 = \Theta(n^2)$, which is what the chaining of equations intuitively says.

Proper abuses of asymptotic notation

Besides the abuse of equality to mean set membership, which we now see has a precise mathematical interpretation, another abuse of asymptotic notation occurs when the variable tending toward ∞ must be inferred from context. For example, when we say $O(g(n))$, we can assume that we're interested in the growth of $g(n)$ as n grows, and if we say $O(g(m))$ we're talking about the growth of $g(m)$ as m grows. The free variable in the expression indicates what variable is going to ∞.

The most common situation requiring contextual knowledge of which variable tends to ∞ occurs when the function inside the asymptotic notation is a constant, as in the expression $O(1)$. We cannot infer from the expression which variable is going to ∞, because no variable appears there. The context must disambiguate. For example, if the equation using asymptotic notation is $f(n) = O(1)$, it's apparent that the variable we're interested in is n. Knowing from context that the variable of interest is n, however, allows us to make perfect sense of the expression by using the formal definition of O-notation: the expression $f(n) = O(1)$ means that the function $f(n)$ is bounded from above by a constant as n goes to ∞. Technically, it might be less ambiguous if we explicitly indicated the variable tending to ∞ in the asymptotic notation itself, but that would clutter the notation. Instead, we simply ensure that the context makes it clear which variable (or variables) tend to ∞.

When the function inside the asymptotic notation is bounded by a positive constant, as in $T(n) = O(1)$, we often abuse asymptotic notation in yet another way, especially when stating recurrences. We may write something like $T(n) = O(1)$ for $n < 3$. According to the formal definition of O-notation, this statement is meaningless, because the definition only says that $T(n)$ is bounded above by a positive constant c for $n \geq n_0$ for some $n_0 > 0$. The value of $T(n)$ for $n < n_0$ need not be so bounded. Thus, in the example $T(n) = O(1)$ for $n < 3$, we cannot infer any constraint on $T(n)$ when $n < 3$, because it might be that $n_0 > 3$.

What is conventionally meant when we say $T(n) = O(1)$ for $n < 3$ is that there exists a positive constant c such that $T(n) \leq c$ for $n < 3$. This convention saves

us the trouble of naming the bounding constant, allowing it to remain anonymous while we focus on more important variables in an analysis. Similar abuses occur with the other asymptotic notations. For example, $T(n) = \Theta(1)$ for $n < 3$ means that $T(n)$ is bounded above and below by positive constants when $n < 3$.

Occasionally, the function describing an algorithm's running time may not be defined for certain input sizes, for example, when an algorithm assumes that the input size is an exact power of 2. We still use asymptotic notation to describe the growth of the running time, understanding that any constraints apply only when the function is defined. For example, suppose that $f(n)$ is defined only on a subset of the natural or nonnegative real numbers. Then $f(n) = O(g(n))$ means that the bound $0 \le T(n) \le cg(n)$ in the definition of O-notation holds for all $n \ge n_0$ over the domain of $f(n)$, that is, where $f(n)$ is defined. This abuse is rarely pointed out, since what is meant is generally clear from context.

In mathematics, it's okay—and often desirable—to abuse a notation, as long as we don't misuse it. If we understand precisely what is meant by the abuse and don't draw incorrect conclusions, it can simplify our mathematical language, contribute to our higher-level understanding, and help us focus on what really matters.

o-notation

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight. We formally define $o(g(n))$ ("little-oh of g of n") as the set

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\} .$$

For example, $2n = o(n^2)$, but $2n^2 \ne o(n^2)$.

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \le f(n) \le cg(n)$ holds for *some* constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \le f(n) < cg(n)$ holds for *all* constants $c > 0$. Intuitively, in o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n gets large:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 .$$

Some authors use this limit as a definition of the o-notation, but the definition in this book also restricts the anonymous functions to be asymptotically nonnegative.

ω-notation

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight. One way to define it is by

$$f(n) \in \omega(g(n)) \text{ if and only if } g(n) \in o(f(n)) .$$

Formally, however, we define $\omega(g(n))$ ("little-omega of g of n") as the set

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0\} .$$

Where the definition of o-notation says that $f(n) < cg(n)$, the definition of ω-notation says the opposite: that $cg(n) < f(n)$. For examples of ω-notation, we have $n^2/2 = \omega(n)$, but $n^2/2 \ne \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty ,$$

if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n gets large.

Comparing functions

Many of the relational properties of real numbers apply to asymptotic comparisons as well. For the following, assume that $f(n)$ and $g(n)$ are asymptotically positive.

Transitivity:

$$\begin{aligned}
f(n) &= \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n)) ,\\
f(n) &= O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n)) ,\\
f(n) &= \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n)) ,\\
f(n) &= o(g(n)) \text{ and } g(n) = o(h(n)) \quad \text{imply} \quad f(n) = o(h(n)) ,\\
f(n) &= \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \quad \text{imply} \quad f(n) = \omega(h(n)) .
\end{aligned}$$

Reflexivity:

$$\begin{aligned}
f(n) &= \Theta(f(n)) ,\\
f(n) &= O(f(n)) ,\\
f(n) &= \Omega(f(n)) .
\end{aligned}$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

Transpose symmetry:

$$f(n) = O(g(n)) \quad \text{if and only if} \quad g(n) = \Omega(f(n)),$$
$$f(n) = o(g(n)) \quad \text{if and only if} \quad g(n) = \omega(f(n)) .$$

Because these properties hold for asymptotic notations, we can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

$$f(n) = O(g(n)) \quad \text{is like} \quad a \le b ,$$
$$f(n) = \Omega(g(n)) \quad \text{is like} \quad a \ge b ,$$
$$f(n) = \Theta(g(n)) \quad \text{is like} \quad a = b ,$$
$$f(n) = o(g(n)) \quad \text{is like} \quad a < b ,$$
$$f(n) = \omega(g(n)) \quad \text{is like} \quad a > b .$$

We say that $f(n)$ is ***asymptotically smaller*** than $g(n)$ if $f(n) = o(g(n))$, and $f(n)$ is ***asymptotically larger*** than $g(n)$ if $f(n) = \omega(g(n))$.

One property of real numbers, however, does not carry over to asymptotic notation:

Trichotomy: For any two real numbers a and b, exactly one of the following must hold: $a < b$, $a = b$, or $a > b$.

Although any two real numbers can be compared, not all functions are asymptotically comparable. That is, for two functions $f(n)$ and $g(n)$, it may be the case that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds. For example, we cannot compare the functions n and $n^{1+\sin n}$ using asymptotic notation, since the value of the exponent in $n^{1+\sin n}$ oscillates between 0 and 2, taking on all values in between.

Exercises

3.2-1
Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$.

3.2-2
Explain why the statement, "The running time of algorithm A is at least $O(n^2)$," is meaningless.

3.2-3
Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

3.2-4
Prove Theorem 3.1.

3.2-5
Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

3.2-6
Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

3.2-7
We can extend our notation to the case of two parameters n and m that can go to ∞ independently at different rates. For a given function $g(n,m)$, we denote by $O(g(n,m))$ the set of functions

$$\begin{aligned} O(g(n,m)) = \{ f(n,m) : \; & \text{there exist positive constants } c, n_0, \text{ and } m_0 \\ & \text{such that } 0 \le f(n,m) \le c\,g(n,m) \\ & \text{for all } n \ge n_0 \text{ or } m \ge m_0 \} \; . \end{aligned}$$

Give corresponding definitions for $\Omega(g(n,m))$ and $\Theta(g(n,m))$.

3.3 Standard notations and common functions

This section reviews some standard mathematical functions and notations and explores the relationships among them. It also illustrates the use of the asymptotic notations.

Monotonicity

A function $f(n)$ is ***monotonically increasing*** if $m \le n$ implies $f(m) \le f(n)$. Similarly, it is ***monotonically decreasing*** if $m \le n$ implies $f(m) \ge f(n)$. A function $f(n)$ is ***strictly increasing*** if $m < n$ implies $f(m) < f(n)$ and ***strictly decreasing*** if $m < n$ implies $f(m) > f(n)$.

Floors and ceilings

For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the floor of x") and the least integer greater than or equal to x by $\lceil x \rceil$ (read "the ceiling of x"). The floor function is monotonically increasing, as is the ceiling function.

Floors and ceilings obey the following properties. For any integer n, we have

$$\lfloor n \rfloor = n = \lceil n \rceil \; . \tag{3.1}$$

For all real x, we have

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1 . \tag{3.2}$$

We also have

$$-\lfloor x \rfloor = \lceil -x \rceil , \tag{3.3}$$

or equivalently,

$$-\lceil x \rceil = \lfloor -x \rfloor . \tag{3.4}$$

For any real number $x \ge 0$ and integers $a, b > 0$, we have

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil , \tag{3.5}$$

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor , \tag{3.6}$$

$$\left\lceil \frac{a}{b} \right\rceil \le \frac{a + (b-1)}{b} , \tag{3.7}$$

$$\left\lfloor \frac{a}{b} \right\rfloor \ge \frac{a - (b-1)}{b} . \tag{3.8}$$

For any integer n and real number x, we have

$$\lfloor n + x \rfloor = n + \lfloor x \rfloor , \tag{3.9}$$

$$\lceil n + x \rceil = n + \lceil x \rceil . \tag{3.10}$$

Modular arithmetic

For any integer a and any positive integer n, the value $a \bmod n$ is the ***remainder*** (or ***residue***) of the quotient a/n:

$$a \bmod n = a - n \lfloor a/n \rfloor . \tag{3.11}$$

It follows that

$$0 \le a \bmod n < n , \tag{3.12}$$

even when a is negative.

Given a well-defined notion of the remainder of one integer when divided by another, it is convenient to provide special notation to indicate equality of remainders. If $(a \bmod n) = (b \bmod n)$, we write $a = b \pmod{n}$ and say that a is ***equivalent*** to b, modulo n. In other words, $a = b \pmod{n}$ if a and b have the same remainder when divided by n. Equivalently, $a = b \pmod{n}$ if and only if n is a divisor of $b - a$. We write $a \ne b \pmod{n}$ if a is not equivalent to b, modulo n.

Polynomials

Given a nonnegative integer d, a ***polynomial in n of degree d*** is a function $p(n)$ of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

where the constants $a_0, a_1, \ldots, a_d$ are the ***coefficients*** of the polynomial and $a_d \neq 0$. A polynomial is asymptotically positive if and only if $a_d > 0$. For an asymptotically positive polynomial $p(n)$ of degree d, we have $p(n) = \Theta(n^d)$. For any real constant $a \geq 0$, the function n^a is monotonically increasing, and for any real constant $a \leq 0$, the function n^a is monotonically decreasing. We say that a function $f(n)$ is ***polynomially bounded*** if $f(n) = O(n^k)$ for some constant k.

Exponentials

For all real $a > 0$, m, and n, we have the following identities:

$$\begin{aligned}
a^0 &= 1 , \\
a^1 &= a , \\
a^{-1} &= 1/a, \\
(a^m)^n &= a^{mn} , \\
(a^m)^n &= (a^n)^m , \\
a^m a^n &= a^{m+n} .
\end{aligned}$$

For all n and $a \geq 1$, the function a^n is monotonically increasing in n. When convenient, we assume that $0^0 = 1$.

We can relate the rates of growth of polynomials and exponentials by the following fact. For all real constants $a > 1$ and b, we have

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0 ,$$

from which we can conclude that

$$n^b = o(a^n) . \tag{3.13}$$

Thus, any exponential function with a base strictly greater than 1 grows faster than any polynomial function.

Using e to denote $2.71828\ldots$, the base of the natural-logarithm function, we have for all real x,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} ,$$

where "!" denotes the factorial function defined later in this section. For all real x, we have the inequality

$$1 + x \le e^x \ , \tag{3.14}$$

where equality holds only when $x = 0$. When $|x| \le 1$, we have the approximation

$$1 + x \le e^x \le 1 + x + x^2 \ . \tag{3.15}$$

When $x \to 0$, the approximation of e^x by $1 + x$ is quite good:

$$e^x = 1 + x + \Theta(x^2) \ .$$

(In this equation, the asymptotic notation is used to describe the limiting behavior as $x \to 0$ rather than as $x \to \infty$.) We have for all x,

$$\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x \ . \tag{3.16}$$

Logarithms

We use the following notations:

$$\begin{aligned}
\lg n &= \log_2 n && \text{(binary logarithm)} \ , \\
\ln n &= \log_e n && \text{(natural logarithm)} \ , \\
\lg^k n &= (\lg n)^k && \text{(exponentiation)} \ , \\
\lg\lg n &= \lg(\lg n) && \text{(composition)} \ .
\end{aligned}$$

We adopt the following notational convention: in the absence of parentheses, *a logarithm function applies only to the next term in the formula*, so that $\lg n + 1$ means $(\lg n) + 1$ and not $\lg(n + 1)$.

For any constant $b > 1$, the function $\log_b n$ is undefined if $n \le 0$, strictly increasing if $n > 0$, negative if $0 < n < 1$, positive if $n > 1$, and 0 if $n = 1$. For all real $a > 0$, $b > 0$, $c > 0$, and n, we have

$$a = b^{\log_b a} \ , \tag{3.17}$$

$$\log_c(ab) = \log_c a + \log_c b \ , \tag{3.18}$$

$$\log_b a^n = n \log_b a \ ,$$

$$\log_b a = \frac{\log_c a}{\log_c b} \ , \tag{3.19}$$

$$\log_b(1/a) = -\log_b a \ , \tag{3.20}$$

$$\log_b a = \frac{1}{\log_a b} \ ,$$

$$a^{\log_b c} = c^{\log_b a} \ , \tag{3.21}$$

where, in each equation above, logarithm bases are not 1.

By equation (3.19), changing the base of a logarithm from one constant to another changes the value of the logarithm by only a constant factor. Consequently, we often use the notation "$\lg n$" when we don't care about constant factors, such as in O-notation. Computer scientists find 2 to be the most natural base for logarithms because so many algorithms and data structures involve splitting a problem into two parts.

There is a simple series expansion for $\ln(1+x)$ when $|x| < 1$:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots . \tag{3.22}$$

We also have the following inequalities for $x > -1$:

$$\frac{x}{1+x} \le \ln(1+x) \le x , \tag{3.23}$$

where equality holds only for $x = 0$.

We say that a function $f(n)$ is ***polylogarithmically bounded*** if $f(n) = O(\lg^k n)$ for some constant k. We can relate the growth of polynomials and polylogarithms by substituting $\lg n$ for n and 2^a for a in equation (3.13). For all real constants $a > 0$ and b, we have

$$\lg^b n = o(n^a) . \tag{3.24}$$

Thus, any positive polynomial function grows faster than any polylogarithmic function.

Factorials

The notation $n!$ (read "n factorial") is defined for integers $n \ge 0$ as

$$n! = \begin{cases} 1 & \text{if } n = 0 , \\ n \cdot (n-1)! & \text{if } n > 0 . \end{cases}$$

Thus, $n! = 1 \cdot 2 \cdot 3 \cdots n$.

A weak upper bound on the factorial function is $n! \le n^n$, since each of the n terms in the factorial product is at most n. ***Stirling's approximation***,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) , \tag{3.25}$$

where e is the base of the natural logarithm, gives us a tighter upper bound, and a lower bound as well. Exercise 3.3-4 asks you to prove the three facts

$$n! = o(n^n) , \tag{3.26}$$

$$n! = \omega(2^n) , \tag{3.27}$$

$$\lg(n!) = \Theta(n \lg n) , \tag{3.28}$$

where Stirling's approximation is helpful in proving equation (3.28). The following equation also holds for all $n \geq 1$:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \tag{3.29}$$

where

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n} .$$

Functional iteration

We use the notation $f^{(i)}(n)$ to denote the function $f(n)$ iteratively applied i times to an initial value of n. Formally, let $f(n)$ be a function over the reals. For nonnegative integers i, we recursively define

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 , \\ f(f^{(i-1)}(n)) & \text{if } i > 0 . \end{cases} \tag{3.30}$$

For example, if $f(n) = 2n$, then $f^{(i)}(n) = 2^i n$.

The iterated logarithm function

We use the notation $\lg^* n$ (read "log star of n") to denote the iterated logarithm, defined as follows. Let $\lg^{(i)} n$ be as defined above, with $f(n) = \lg n$. Because the logarithm of a nonpositive number is undefined, $\lg^{(i)} n$ is defined only if $\lg^{(i-1)} n > 0$. Be sure to distinguish $\lg^{(i)} n$ (the logarithm function applied i times in succession, starting with argument n) from $\lg^i n$ (the logarithm of n raised to the ith power). Then we define the iterated logarithm function as

$$\lg^* n = \min\left\{i \geq 0 : \lg^{(i)} n \leq 1\right\} .$$

The iterated logarithm is a *very* slowly growing function:

$$\begin{aligned} \lg^* 2 &= 1 , \\ \lg^* 4 &= 2 , \\ \lg^* 16 &= 3 , \\ \lg^* 65536 &= 4 , \\ \lg^*(2^{65536}) &= 5 . \end{aligned}$$

Since the number of atoms in the observable universe is estimated to be about 10^{80}, which is much less than $2^{65536} = 10^{65536/\lg 10} \approx 10^{19,728}$, we rarely encounter an input size n for which $\lg^* n > 5$.

Fibonacci numbers

We define the ***Fibonacci numbers*** F_i, for $i \geq 0$, as follows:

$$F_i = \begin{cases} 0 & \text{if } i = 0 , \\ 1 & \text{if } i = 1 , \\ F_{i-1} + F_{i-2} & \text{if } i \geq 2 . \end{cases} \tag{3.31}$$

Thus, after the first two, each Fibonacci number is the sum of the two previous ones, yielding the sequence

0,1,1,2,3,5,8,13,21,34,55,. . ..

Fibonacci numbers are related to the ***golden ratio*** ϕ and its conjugate $\widehat{\phi}$, which are the two roots of the equation

$$x^2 = x + 1 .$$

As Exercise 3.3-7 asks you to prove, the golden ratio is given by

$$\begin{aligned} \phi &= \frac{1 + \sqrt{5}}{2} \\ &= 1.61803 \ldots , \end{aligned} \tag{3.32}$$

and its conjugate, by

$$\begin{aligned} \widehat{\phi} &= \frac{1 - \sqrt{5}}{2} \\ &= -.61803 \ldots . \end{aligned} \tag{3.33}$$

Specifically, we have

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} ,$$

which can be proved by induction (Exercise 3.3-8). Since $\left|\widehat{\phi}\right| < 1$, we have

$$\begin{aligned} \frac{\left|\widehat{\phi}^i\right|}{\sqrt{5}} &< \frac{1}{\sqrt{5}} \\ &< \frac{1}{2} , \end{aligned}$$

which implies that

$$F_i = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor , \tag{3.34}$$

which is to say that the ith Fibonacci number F_i is equal to $\phi^i / \sqrt{5}$ rounded to the nearest integer. Thus, Fibonacci numbers grow exponentially.

Exercises

3.3-1
Show that if $f(n)$ and $g(n)$ are monotonically increasing functions, then so are the functions $f(n) + g(n)$ and $f(g(n))$, and if $f(n)$ and $g(n)$ are in addition nonnegative, then $f(n) \cdot g(n)$ is monotonically increasing.

3.3-2
Prove that $\lfloor \alpha n \rfloor + \lceil (1 - \alpha)n \rceil = n$ for any integer n and real number α in the range $0 \le \alpha \le 1$.

3.3-3
Use equation (3.14) or other means to show that $(n + o(n))^k = \Theta(n^k)$ for any real constant k. Conclude that $\lceil n \rceil^k = \Theta(n^k)$ and $\lfloor n \rfloor^k = \Theta(n^k)$.

3.3-4
Prove the following:

a. Equation (3.21).

b. Equations (3.26)–(3.28).

c. $\lg(\Theta(n)) = \Theta(\lg n)$.

★ ***3.3-5***
Is the function $\lceil \lg n \rceil!$ polynomially bounded? Is the function $\lceil \lg \lg n \rceil!$ polynomially bounded?

★ ***3.3-6***
Which is asymptotically larger: $\lg(\lg^* n)$ or $\lg^*(\lg n)$?

3.3-7
Show that the golden ratio ϕ and its conjugate $\hat{\phi}$ both satisfy the equation $x^2 = x + 1$.

3.3-8
Prove by induction that the ith Fibonacci number satisfies the equation

$$F_i = (\phi^i - \hat{\phi}^i)/\sqrt{5} ,$$

where ϕ is the golden ratio and $\hat{\phi}$ is its conjugate.

3.3-9
Show that $k \lg k = \Theta(n)$ implies $k = \Theta(n/\lg n)$.

Problems

3-1 Asymptotic behavior of polynomials

Let

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

where $a_d > 0$, be a degree-d polynomial in n, and let k be a constant. Use the definitions of the asymptotic notations to prove the following properties.

a. If $k \geq d$, then $p(n) = O(n^k)$.

b. If $k \leq d$, then $p(n) = \Omega(n^k)$.

c. If $k = d$, then $p(n) = \Theta(n^k)$.

d. If $k > d$, then $p(n) = o(n^k)$.

e. If $k < d$, then $p(n) = \omega(n^k)$.

3-2 Relative asymptotic growths

Indicate, for each pair of expressions (A, B) in the table below whether A is O, o, Ω, ω, or Θ of B. Assume that $k \geq 1$, $\epsilon > 0$, and $c > 1$ are constants. Write your answer in the form of the table with "yes" or "no" written in each box.

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg c}$	$c^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

3-3 Ordering by asymptotic growth rates

a. Rank the following functions by order of growth. That is, find an arrangement $g_1, g_2, \ldots, g_{30}$ of the functions satisfying $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, $\ldots$, $g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes such that functions $f(n)$ and $g(n)$ belong to the same class if and only if $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(3/2)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

b. Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$.

3-4 Asymptotic notation properties

Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures.

a. $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

b. $f(n) + g(n) = \Theta(\min\{f(n), g(n)\})$.

c. $f(n) = O(g(n))$ implies $\lg f(n) = O(\lg g(n))$, where $\lg g(n) \geq 1$ and $f(n) \geq 1$ for all sufficiently large n.

d. $f(n) = O(g(n))$ implies $2^{f(n)} = O\left(2^{g(n)}\right)$.

e. $f(n) = O\left((f(n))^2\right)$.

f. $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$.

g. $f(n) = \Theta(f(n/2))$.

h. $f(n) + o(f(n)) = \Theta(f(n))$.

3-5 Manipulating asymptotic notation

Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove the following identities:

a. $\Theta(\Theta(f(n))) = \Theta(f(n))$.

b. $\Theta(f(n)) + O(f(n)) = \Theta(f(n))$.

c. $\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$.

d. $\Theta(f(n)) \cdot \Theta(g(n)) = \Theta(f(n) \cdot g(n))$.

e. Argue that for any real constants $a_1, b_1 > 0$ and integer constants k_1, k_2, the following asymptotic bound holds:

$$(a_1 n)^{k_1} \lg^{k_2}(a_2 n) = \Theta(n^{k_1} \lg^{k_2} n) .$$

★ ***f.*** Prove that for $S \subseteq \mathbb{Z}$, we have

$$\sum_{k \in S} \Theta(f(k)) = \Theta\left(\sum_{k \in S} f(k)\right) ,$$

assuming that both sums converge.

★ ***g.*** Show that for $S \subseteq \mathbb{Z}$, the following asymptotic bound does not necessarily hold, even assuming that both products converge, by giving a counterexample:

$$\prod_{k \in S} \Theta(f(k)) = \Theta\left(\prod_{k \in S} f(k)\right) .$$

3-6 Variations on O and Ω

Some authors define Ω-notation in a slightly different way than this textbook does. We'll use the nomenclature $\overset{\infty}{\Omega}$ (read "omega infinity") for this alternative definition. We say that $f(n) = \overset{\infty}{\Omega}(g(n))$ if there exists a positive constant c such that $f(n) \geq cg(n) \geq 0$ for infinitely many integers n.

a. Show that for any two asymptotically nonnegative functions $f(n)$ and $g(n)$, we have $f(n) = O(g(n))$ or $f(n) = \overset{\infty}{\Omega}(g(n))$ (or both).

b. Show that there exist two asymptotically nonnegative functions $f(n)$ and $g(n)$ for which neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds.

c. Describe the potential advantages and disadvantages of using $\overset{\infty}{\Omega}$-notation instead of Ω-notation to characterize the running times of programs.

Some authors also define O in a slightly different manner. We'll use O' for the alternative definition: $f(n) = O'(g(n))$ if and only if $|f(n)| = O(g(n))$.

d. What happens to each direction of the "if and only if" in Theorem 3.1 on page 56 if we substitute O' for O but still use Ω?

Some authors define $\widetilde{O}$ (read "soft-oh") to mean O with logarithmic factors ignored:

$$\widetilde{O}(g(n)) = \{f(n) : \text{ there exist positive constants } c, k, \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n)\lg^k(n) \text{ for all } n \ge n_0\} .$$

e. Define $\widetilde{\Omega}$ and $\widetilde{\Theta}$ in a similar manner. Prove the corresponding analog to Theorem 3.1.

3-7 Iterated functions

We can apply the iteration operator * used in the $\lg^*$ function to any monotonically increasing function $f(n)$ over the reals. For a given constant $c \in \mathbb{R}$, we define the iterated function f_c^* by

$$f_c^*(n) = \min\left\{i \ge 0 : f^{(i)}(n) \le c\right\} ,$$

which need not be well defined in all cases. In other words, the quantity $f_c^*(n)$ is the minimum number of iterated applications of the function f required to reduce its argument down to c or less.

For each of the functions $f(n)$ and constants c in the table below, give as tight a bound as possible on $f_c^*(n)$. If there is no i such that $f^{(i)}(n) \le c$, write "undefined" as your answer.

	$f(n)$	c	$f_c^*(n)$
a.	$n-1$	0	
b.	$\lg n$	1	
c.	$n/2$	1	
d.	$n/2$	2	
e.	$\sqrt{n}$	2	
f.	$\sqrt{n}$	1	
g.	$n^{1/3}$	2	

Chapter notes

Knuth [259] traces the origin of the O-notation to a number-theory text by P. Bachmann in 1892. The o-notation was invented by E. Landau in 1909 for his discussion of the distribution of prime numbers. The Ω and Θ notations were advocated by Knuth [265] to correct the popular, but technically sloppy, practice in the literature of using O-notation for both upper and lower bounds. As noted earlier in this chapter, many people continue to use the O-notation where the Θ-notation is more technically precise. The soft-oh notation $\widetilde{O}$ in Problem 3-6 was introduced

by Babai, Luks, and Seress [31], although it was originally written as $O\sim$. Some authors now define $\widetilde{O}(g(n))$ as ignoring factors that are logarithmic in $g(n)$, rather than in n. With this definition, we can say that $n2^n = \widetilde{O}(2^n)$, but with the definition in Problem 3-6, this statement is not true. Further discussion of the history and development of asymptotic notations appears in works by Knuth [259, 265] and Brassard and Bratley [70].

Not all authors define the asymptotic notations in the same way, although the various definitions agree in most common situations. Some of the alternative definitions encompass functions that are not asymptotically nonnegative, as long as their absolute values are appropriately bounded.

Equation (3.29) is due to Robbins [381]. Other properties of elementary mathematical functions can be found in any good mathematical reference, such as Abramowitz and Stegun [1] or Zwillinger [468], or in a calculus book, such as Apostol [19] or Thomas et al. [433]. Knuth [259] and Graham, Knuth, and Patashnik [199] contain a wealth of material on discrete mathematics as used in computer science.

4 Divide-and-Conquer

The divide-and-conquer method is a powerful strategy for designing asymptotically efficient algorithms. We saw an example of divide-and-conquer in Section 2.3.1 when learning about merge sort. In this chapter, we'll explore applications of the divide-and-conquer method and acquire valuable mathematical tools that you can use to solve the recurrences that arise when analyzing divide-and-conquer algorithms.

Recall that for divide-and-conquer, you solve a given problem (instance) recursively. If the problem is small enough—the ***base case***—you just solve it directly without recursing. Otherwise—the ***recursive case***—you perform three characteristic steps:

Divide the problem into one or more subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

Combine the subproblem solutions to form a solution to the original problem.

A divide-and-conquer algorithm breaks down a large problem into smaller subproblems, which themselves may be broken down into even smaller subproblems, and so forth. The recursion ***bottoms out*** when it reaches a base case and the subproblem is small enough to solve directly without further recursing.

Recurrences

To analyze recursive divide-and-conquer algorithms, we'll need some mathematical tools. A ***recurrence*** is an equation that describes a function in terms of its value on other, typically smaller, arguments. Recurrences go hand in hand with the divide-and-conquer method because they give us a natural way to characterize the running times of recursive algorithms mathematically. You saw an example of a recurrence in Section 2.3.2 when we analyzed the worst-case running time of merge sort.

For the divide-and-conquer matrix-multiplication algorithms presented in Sections 4.1 and 4.2, we'll derive recurrences that describe their worst-case running times. To understand why these two divide-and-conquer algorithms perform the way they do, you'll need to learn how to solve the recurrences that describe their running times. Sections 4.3–4.7 teach several methods for solving recurrences. These sections also explore the mathematics behind recurrences, which can give you stronger intuition for designing your own divide-and-conquer algorithms.

We want to get to the algorithms as soon as possible. So, let's just cover a few recurrence basics now, and then we'll look more deeply at recurrences, especially how to solve them, after we see the matrix-multiplication examples.

The general form of a recurrence is an equation or inequality that describes a function over the integers or reals using the function itself. It contains two or more cases, depending on the argument. If a case involves the recursive invocation of the function on different (usually smaller) inputs, it is a ***recursive case***. If a case does not involve a recursive invocation, it is a ***base case***. There may be zero, one, or many functions that satisfy the statement of the recurrence. The recurrence is ***well defined*** if there is at least one function that satisfies it, and ***ill defined*** otherwise.

Algorithmic recurrences

We'll be particularly interested in recurrences that describe the running times of divide-and-conquer algorithms. A recurrence $T(n)$ is ***algorithmic*** if, for every sufficiently large ***threshold*** constant $n_0 > 0$, the following two properties hold:

1. For all $n < n_0$, we have $T(n) = \Theta(1)$.
2. For all $n \geq n_0$, every path of recursion terminates in a defined base case within a finite number of recursive invocations.

Similar to how we sometimes abuse asymptotic notation (see page 60), when a function is not defined for all arguments, we understand that this definition is constrained to values of n for which $T(n)$ is defined.

Why would a recurrence $T(n)$ that represents a (correct) divide-and-conquer algorithm's worst-case running time satisfy these properties for all sufficiently large threshold constants? The first property says that there exist constants c_1, c_2 such that $0 < c_1 \leq T(n) \leq c_2$ for $n < n_0$. For every legal input, the algorithm must output the solution to the problem it's solving in finite time (see Section 1.1). Thus we can let c_1 be the minimum amount of time to call and return from a procedure, which must be positive, because machine instructions need to be executed to invoke a procedure. The running time of the algorithm may not be defined for some values of n if there are no legal inputs of that size, but it must be defined for at least one, or else the "algorithm" doesn't solve any problem. Thus we can let c_2 be the algorithm's maximum running time on any input of size $n < n_0$, where n_0 is

sufficiently large that the algorithm solves at least one problem of size less than n_0. The maximum is well defined, since there are at most a finite number of inputs of size less than n_0, and there is at least one if n_0 is sufficiently large. Consequently, $T(n)$ satisfies the first property. If the second property fails to hold for $T(n)$, then the algorithm isn't correct, because it would end up in an infinite recursive loop or otherwise fail to compute a solution. Thus, it stands to reason that a recurrence for the worst-case running time of a correct divide-and-conquer algorithm would be algorithmic.

Conventions for recurrences

We adopt the following convention:

> *Whenever a recurrence is stated without an explicit base case, we assume that the recurrence is algorithmic.*

That means you're free to pick any sufficiently large threshold constant n_0 for the range of base cases where $T(n) = \Theta(1)$. Interestingly, the asymptotic solutions of most algorithmic recurrences you're likely to see when analyzing algorithms don't depend on the choice of threshold constant, as long as it's large enough to make the recurrence well defined.

Asymptotic solutions of algorithmic divide-and-conquer recurrences also don't tend to change when we drop any floors or ceilings in a recurrence defined on the integers to convert it to a recurrence defined on the reals. Section 4.7 gives a sufficient condition for ignoring floors and ceilings that applies to most of the divide-and-conquer recurrences you're likely to see. Consequently, we'll frequently state algorithmic recurrences without floors and ceilings. Doing so generally simplifies the statement of the recurrences, as well as any math that we do with them.

You may sometimes see recurrences that are not equations, but rather inequalities, such as $T(n) \le 2T(n/2) + \Theta(n)$. Because such a recurrence states only an upper bound on $T(n)$, we express its solution using O-notation rather than Θ-notation. Similarly, if the inequality is reversed to $T(n) \ge 2T(n/2) + \Theta(n)$, then, because the recurrence gives only a lower bound on $T(n)$, we use Ω-notation in its solution.

Divide-and-conquer and recurrences

This chapter illustrates the divide-and-conquer method by presenting and using recurrences to analyze two divide-and-conquer algorithms for multiplying $n \times n$ matrices. Section 4.1 presents a simple divide-and-conquer algorithm that solves a matrix-multiplication problem of size n by breaking it into four subproblems of size $n/2$, which it then solves recursively. The running time of the algorithm can be characterized by the recurrence

$$T(n) = 8T(n/2) + \Theta(1) ,$$

which turns out to have the solution $T(n) = \Theta(n^3)$. Although this divide-and-conquer algorithm is no faster than the straightforward method that uses a triply nested loop, it leads to an asymptotically faster divide-and-conquer algorithm due to V. Strassen, which we'll explore in Section 4.2. Strassen's remarkable algorithm divides a problem of size n into seven subproblems of size $n/2$ which it solves recursively. The running time of Strassen's algorithm can be described by the recurrence

$$T(n) = 7T(n/2) + \Theta(n^2) ,$$

which has the solution $T(n) = \Theta(n^{\lg 7}) = O(n^{2.81})$. Strassen's algorithm beats the straightforward looping method asymptotically.

These two divide-and-conquer algorithms both break a problem of size n into several subproblems of size $n/2$. Although it is common when using divide-and-conquer for all the subproblems to have the same size, that isn't always the case. Sometimes it's productive to divide a problem of size n into subproblems of different sizes, and then the recurrence describing the running time reflects the irregularity. For example, consider a divide-and-conquer algorithm that divides a problem of size n into one subproblem of size $n/3$ and another of size $2n/3$, taking $\Theta(n)$ time to divide the problem and combine the solutions to the subproblems. Then the algorithm's running time can be described by the recurrence

$$T(n) = T(n/3) + T(2n/3) + \Theta(n) ,$$

which turns out to have solution $T(n) = \Theta(n \lg n)$. We'll even see an algorithm in Chapter 9 that solves a problem of size n by recursively solving a subproblem of size $n/5$ and another of size $7n/10$, taking $\Theta(n)$ time for the divide and combine steps. Its performance satisfies the recurrence

$$T(n) = T(n/5) + T(7n/10) + \Theta(n) ,$$

which has solution $T(n) = \Theta(n)$.

Although divide-and-conquer algorithms usually create subproblems with sizes a constant fraction of the original problem size, that's not always the case. For example, a recursive version of linear search (see Exercise 2.1-4) creates just one subproblem, with one element less than the original problem. Each recursive call takes constant time plus the time to recursively solve a subproblem with one less element, leading to the recurrence

$$T(n) = T(n-1) + \Theta(1) ,$$

which has solution $T(n) = \Theta(n)$. Nevertheless, the vast majority of efficient divide-and-conquer algorithms solve subproblems that are a constant fraction of the size of the original problem, which is where we'll focus our efforts.

Solving recurrences

After learning about divide-and-conquer algorithms for matrix multiplication in Sections 4.1 and 4.2, we'll explore several mathematical tools for solving recurrences—that is, for obtaining asymptotic Θ-, O-, or Ω-bounds on their solutions. We want simple-to-use tools that can handle the most commonly occurring situations. But we also want general tools that work, perhaps with a little more effort, for less common cases. This chapter offers four methods for solving recurrences:

- In the ***substitution method*** (Section 4.3), you guess the form of a bound and then use mathematical induction to prove your guess correct and solve for constants. This method is perhaps the most robust method for solving recurrences, but it also requires you to make a good guess and to produce an inductive proof.
- The ***recursion-tree method*** (Section 4.4) models the recurrence as a tree whose nodes represent the costs incurred at various levels of the recursion. To solve the recurrence, you determine the costs at each level and add them up, perhaps using techniques for bounding summations from Section A.2. Even if you don't use this method to formally prove a bound, it can be helpful in guessing the form of the bound for use in the substitution method.
- The ***master method*** (Sections 4.5 and 4.6) is the easiest method, when it applies. It provides bounds for recurrences of the form

  $$T(n) = aT(n/b) + f(n) ,$$

  where $a > 0$ and $b > 1$ are constants and $f(n)$ is a given "driving" function. This type of recurrence tends to arise more frequently in the study of algorithms than any other. It characterizes a divide-and-conquer algorithm that creates a subproblems, each of which is $1/b$ times the size of the original problem, using $f(n)$ time for the divide and combine steps. To apply the master method, you need to memorize three cases, but once you do, you can easily determine asymptotic bounds on running times for many divide-and-conquer algorithms.
- The ***Akra-Bazzi method*** (Section 4.7) is a general method for solving divide-and-conquer recurrences. Although it involves calculus, it can be used to attack more complicated recurrences than those addressed by the master method.

4.1 Multiplying square matrices

We can use the divide-and-conquer method to multiply square matrices. If you've seen matrices before, then you probably know how to multiply them. (Otherwise,

you should read Section D.1.) Let $A = (a_{ik})$ and $B = (b_{jk})$ be square $n \times n$ matrices. The matrix product $C = A \cdot B$ is also an $n \times n$ matrix, where for $i, j = 1, 2, \ldots, n$, the (i, j) entry of C is given by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \ . \tag{4.1}$$

Generally, we'll assume that the matrices are ***dense***, meaning that most of the n^2 entries are not 0, as opposed to ***sparse***, where most of the n^2 entries are 0 and the nonzero entries can be stored more compactly than in an $n \times n$ array.

Computing the matrix C requires computing n^2 matrix entries, each of which is the sum of n pairwise products of input elements from A and B. The MATRIX-MULTIPLY procedure implements this strategy in a straightforward manner, and it generalizes the problem slightly. It takes as input three $n \times n$ matrices A, B, and C, and it adds the matrix product $A \cdot B$ to C, storing the result in C. Thus, it computes $C = C + A \cdot B$, instead of just $C = A \cdot B$. If only the product $A \cdot B$ is needed, just initialize all n^2 entries of C to 0 before calling the procedure, which takes an additional $\Theta(n^2)$ time. We'll see that the cost of matrix multiplication asymptotically dominates this initialization cost.

```
MATRIX-MULTIPLY(A, B, C, n)
1  for i = 1 to n                    // compute entries in each of n rows
2      for j = 1 to n                // compute n entries in row i
3          for k = 1 to n
4              c_ij = c_ij + a_ik · b_kj   // add in another term of equation (4.1)
```

The pseudocode for MATRIX-MULTIPLY works as follows. The **for** loop of lines 1–4 computes the entries of each row i, and within a given row i, the **for** loop of lines 2–4 computes each of the entries c_{ij} for each column j. Each iteration of the **for** loop of lines 3–4 adds in one more term of equation (4.1).

Because each of the triply nested **for** loops runs for exactly n iterations, and each execution of line 4 takes constant time, the MATRIX-MULTIPLY procedure operates in $\Theta(n^3)$ time. Even if we add in the $\Theta(n^2)$ time for initializing C to 0, the running time is still $\Theta(n^3)$.

A simple divide-and-conquer algorithm

Let's see how to compute the matrix product $A \cdot B$ using divide-and-conquer. For $n > 1$, the divide step partitions the $n \times n$ matrices into four $n/2 \times n/2$ submatrices. We'll assume that n is an exact power of 2, so that as the algorithm recurses, we are guaranteed that the submatrix dimensions are integer. (Exercise 4.1-1 asks you

to relax this assumption.) As with MATRIX-MULTIPLY, we'll actually compute $C = C + A \cdot B$. But to simplify the math behind the algorithm, let's assume that C has been initialized to the zero matrix, so that we are indeed computing $C = A \cdot B$.

The divide step views each of the $n \times n$ matrices A, B, and C as four $n/2 \times n/2$ submatrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}. \tag{4.2}$$

Then we can write the matrix product as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{4.3}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix}, \tag{4.4}$$

which corresponds to the equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}\,, \tag{4.5}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}\,, \tag{4.6}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}\,, \tag{4.7}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}\,. \tag{4.8}$$

Equations (4.5)–(4.8) involve eight $n/2 \times n/2$ multiplications and four additions of $n/2 \times n/2$ submatrices.

As we look to transform these equations to an algorithm that can be described with pseudocode, or even implemented for real, there are two common approaches for implementing the matrix partitioning.

One strategy is to allocate temporary storage to hold A's four submatrices A_{11}, A_{12}, A_{21}, and A_{22} and B's four submatrices B_{11}, B_{12}, B_{21}, and B_{22}. Then copy each element in A and B to its corresponding location in the appropriate submatrix. After the recursive conquer step, copy the elements in each of C's four submatrices C_{11}, C_{12}, C_{21}, and C_{22} to their corresponding locations in C. This approach takes $\Theta(n^2)$ time, since $3n^2$ elements are copied.

The second approach uses index calculations and is faster and more practical. A submatrix can be specified within a matrix by indicating where within the matrix the submatrix lies without touching any matrix elements. Partitioning a matrix (or recursively, a submatrix) only involves arithmetic on this location information, which has constant size independent of the size of the matrix. Changes to the submatrix elements update the original matrix, since they occupy the same storage.

Going forward, we'll assume that index calculations are used and that partitioning can be performed in $\Theta(1)$ time. Exercise 4.1-3 asks you to show that it makes no difference to the overall asymptotic running time of matrix multiplication, however, whether the partitioning of matrices uses the first method of copying or the

second method of index calculation. But for other divide-and-conquer matrix calculations, such as matrix addition, it can make a difference, as Exercise 4.1-4 asks you to show.

The procedure MATRIX-MULTIPLY-RECURSIVE uses equations (4.5)–(4.8) to implement a divide-and-conquer strategy for square-matrix multiplication. Like MATRIX-MULTIPLY, the procedure MATRIX-MULTIPLY-RECURSIVE computes $C = C + A \cdot B$ since, if necessary, C can be initialized to 0 before the procedure is called in order to compute only $C = A \cdot B$.

```
MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
1  if n == 1
2  // Base case.
3      c11 = c11 + a11 · b11
4      return
5  // Divide.
6  partition A, B, and C into n/2 × n/2 submatrices
       A11, A12, A21, A22; B11, B12, B21, B22;
       and C11, C12, C21, C22; respectively
7  // Conquer.
8  MATRIX-MULTIPLY-RECURSIVE(A11, B11, C11, n/2)
9  MATRIX-MULTIPLY-RECURSIVE(A11, B12, C12, n/2)
10 MATRIX-MULTIPLY-RECURSIVE(A21, B11, C21, n/2)
11 MATRIX-MULTIPLY-RECURSIVE(A21, B12, C22, n/2)
12 MATRIX-MULTIPLY-RECURSIVE(A12, B21, C11, n/2)
13 MATRIX-MULTIPLY-RECURSIVE(A12, B22, C12, n/2)
14 MATRIX-MULTIPLY-RECURSIVE(A22, B21, C21, n/2)
15 MATRIX-MULTIPLY-RECURSIVE(A22, B22, C22, n/2)
```

As we walk through the pseudocode, we'll derive a recurrence to characterize its running time. Let $T(n)$ be the worst-case time to multiply two $n \times n$ matrices using this procedure.

In the base case, when $n = 1$, line 3 performs just the one scalar multiplication and one addition, which means that $T(1) = \Theta(1)$. As is our convention for constant base cases, we can omit this base case in the statement of the recurrence.

The recursive case occurs when $n > 1$. As discussed, we'll use index calculations to partition the matrices in line 6, taking $\Theta(1)$ time. Lines 8–15 recursively call MATRIX-MULTIPLY-RECURSIVE a total of eight times. The first four recursive calls compute the first terms of equations (4.5)–(4.8), and the subsequent four recursive calls compute and add in the second terms. Each recursive call adds the product of a submatrix of A and a submatrix of B to the appropriate submatrix

of C in place, thanks to index calculations. Because each recursive call multiplies two $n/2 \times n/2$ matrices, thereby contributing $T(n/2)$ to the overall running time, the time taken by all eight recursive calls is $8T(n/2)$. There is no combine step, because the matrix C is updated in place. The total time for the recursive case, therefore, is the sum of the partitioning time and the time for all the recursive calls, or $\Theta(1) + 8T(n/2)$.

Thus, omitting the statement of the base case, our recurrence for the running time of MATRIX-MULTIPLY-RECURSIVE is

$$T(n) = 8T(n/2) + \Theta(1) . \tag{4.9}$$

As we'll see from the master method in Section 4.5, recurrence (4.9) has the solution $T(n) = \Theta(n^3)$, which means that it has the same asymptotic running time as the straightforward MATRIX-MULTIPLY procedure.

Why is the $\Theta(n^3)$ solution to this recurrence so much larger than the $\Theta(n \lg n)$ solution to the merge-sort recurrence (2.3) on page 41? After all, the recurrence for merge sort contains a $\Theta(n)$ term, whereas the recurrence for recursive matrix multiplication contains only a $\Theta(1)$ term.

Let's think about what the recursion tree for recurrence (4.9) would look like as compared with the recursion tree for merge sort, illustrated in Figure 2.5 on page 43. The factor of 2 in the merge-sort recurrence determines how many children each tree node has, which in turn determines how many terms contribute to the sum at each level of the tree. In comparison, for the recurrence (4.9) for MATRIX-MULTIPLY-RECURSIVE, each internal node in the recursion tree has eight children, not two, leading to a "bushier" recursion tree with many more leaves, despite the fact that the internal nodes are each much smaller. Consequently, the solution to recurrence (4.9) grows much more quickly than the solution to recurrence (2.3), which is borne out in the actual solutions: $\Theta(n^3)$ versus $\Theta(n \lg n)$.

Exercises

Note: You may wish to read Section 4.5 before attempting some of these exercises.

4.1-1

Generalize MATRIX-MULTIPLY-RECURSIVE to multiply $n \times n$ matrices for which n is not necessarily an exact power of 2. Give a recurrence describing its running time. Argue that it runs in $\Theta(n^3)$ time in the worst case.

4.1-2

How quickly can you multiply a $kn \times n$ matrix (kn rows and n columns) by an $n \times kn$ matrix, where $k \geq 1$, using MATRIX-MULTIPLY-RECURSIVE as a subroutine? Answer the same question for multiplying an $n \times kn$ matrix by a $kn \times n$ matrix. Which is asymptotically faster, and by how much?

4.1-3

Suppose that instead of partitioning matrices by index calculation in MATRIX-MULTIPLY-RECURSIVE, you copy the appropriate elements of A, B, and C into separate $n/2 \times n/2$ submatrices $A_{11}, A_{12}, A_{21}, A_{22}$; $B_{11}, B_{12}, B_{21}, B_{22}$; and C_{11}, C_{12}, C_{21}, C_{22}, respectively. After the recursive calls, you copy the results from C_{11}, C_{12}, C_{21}, and C_{22} back into the appropriate places in C. How does recurrence (4.9) change, and what is its solution?

4.1-4

Write pseudocode for a divide-and-conquer algorithm MATRIX-ADD-RECURSIVE that sums two $n \times n$ matrices A and B by partitioning each of them into four $n/2 \times n/2$ submatrices and then recursively summing corresponding pairs of submatrices. Assume that matrix partitioning uses $\Theta(1)$-time index calculations. Write a recurrence for the worst-case running time of MATRIX-ADD-RECURSIVE, and solve your recurrence. What happens if you use $\Theta(n^2)$-time copying to implement the partitioning instead of index calculations?

4.2 Strassen's algorithm for matrix multiplication

You might find it hard to imagine that any matrix multiplication algorithm could take less than $\Theta(n^3)$ time, since the natural definition of matrix multiplication requires n^3 scalar multiplications. Indeed, many mathematicians presumed that it was not possible to multiply matrices in $o(n^3)$ time until 1969, when V. Strassen [424] published a remarkable recursive algorithm for multiplying $n \times n$ matrices. Strassen's algorithm runs in $\Theta(n^{\lg 7})$ time. Since $\lg 7 = 2.8073549\ldots$, Strassen's algorithm runs in $O(n^{2.81})$ time, which is asymptotically better than the $\Theta(n^3)$ MATRIX-MULTIPLY and MATRIX-MULTIPLY-RECURSIVE procedures.

The key to Strassen's method is to use the divide-and-conquer idea from the MATRIX-MULTIPLY-RECURSIVE procedure, but make the recursion tree less bushy. We'll actually increase the work for each divide and combine step by a constant factor, but the reduction in bushiness will pay off. We won't reduce the bushiness from the eight-way branching of recurrence (4.9) all the way down to the two-way branching of recurrence (2.3), but we'll improve it just a little, and that will make a big difference. Instead of performing eight recursive multiplications of $n/2 \times n/2$ matrices, Strassen's algorithm performs only seven. The cost of eliminating one matrix multiplication is several new additions and subtractions of $n/2 \times n/2$ matrices, but still only a constant number. Rather than saying "additions and subtractions" everywhere, we'll adopt the common terminology of call-

ing them both "additions" because subtraction is structurally the same computation as addition, except for a change of sign.

To get an inkling how the number of multiplications might be reduced, as well as why reducing the number of multiplications might be desirable for matrix calculations, suppose that you have two numbers x and y, and you want to calculate the quantity $x^2 - y^2$. The straightforward calculation requires two multiplications to square x and y, followed by one subtraction (which you can think of as a "negative addition"). But let's recall the old algebra trick $x^2 - y^2 = x^2 - xy + xy - y^2 = x(x - y) + y(x - y) = (x + y)(x - y)$. Using this formulation of the desired quantity, you could instead compute the sum $x + y$ and the difference $x - y$ and then multiply them, requiring only a single multiplication and two additions. At the cost of an extra addition, only one multiplication is needed to compute an expression that looks as if it requires two. If x and y are scalars, there's not much difference: both approaches require three scalar operations. If x and y are large matrices, however, the cost of multiplying outweighs the cost of adding, in which case the second method outperforms the first, although not asymptotically.

Strassen's strategy for reducing the number of matrix multiplications at the expense of more matrix additions is not at all obvious—perhaps the biggest understatement in this book! As with MATRIX-MULTIPLY-RECURSIVE, Strassen's algorithm uses the divide-and-conquer method to compute $C = C + A \cdot B$, where A, B, and C are all $n \times n$ matrices and n is an exact power of 2. Strassen's algorithm computes the four submatrices C_{11}, C_{12}, C_{21}, and C_{22} of C from equations (4.5)–(4.8) on page 82 in four steps. We'll analyze costs as we go along to develop a recurrence $T(n)$ for the overall running time. Let's see how it works:

1. If $n = 1$, the matrices each contain a single element. Perform a single scalar multiplication and a single scalar addition, as in line 3 of MATRIX-MULTIPLY-RECURSIVE, taking $\Theta(1)$ time, and return. Otherwise, partition the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.2). This step takes $\Theta(1)$ time by index calculation, just as in MATRIX-MULTIPLY-RECURSIVE.

2. Create $n/2 \times n/2$ matrices $S_1, S_2, \ldots, S_{10}$, each of which is the sum or difference of two submatrices from step 1. Create and zero the entries of seven $n/2 \times n/2$ matrices $P_1, P_2, \ldots, P_7$ to hold seven $n/2 \times n/2$ matrix products. All 17 matrices can be created, and the P_i initialized, in $\Theta(n^2)$ time.

3. Using the submatrices from step 1 and the matrices $S_1, S_2, \ldots, S_{10}$ created in step 2, recursively compute each of the seven matrix products $P_1, P_2, \ldots, P_7$, taking $7T(n/2)$ time.

4. Update the four submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding or subtracting various P_i matrices, which takes $\Theta(n^2)$ time.

We'll see the details of steps 2–4 in a moment, but we already have enough information to set up a recurrence for the running time of Strassen's method. As is common, the base case in step 1 takes $\Theta(1)$ time, which we'll omit when stating the recurrence. When $n > 1$, steps 1, 2, and 4 take a total of $\Theta(n^2)$ time, and step 3 requires seven multiplications of $n/2 \times n/2$ matrices. Hence, we obtain the following recurrence for the running time of Strassen's algorithm:

$$T(n) = 7T(n/2) + \Theta(n^2) . \tag{4.10}$$

Compared with MATRIX-MULTIPLY-RECURSIVE, we have traded off one recursive submatrix multiplication for a constant number of submatrix additions. Once you understand recurrences and their solutions, you'll be able to see why this tradeoff actually leads to a lower asymptotic running time. By the master method in Section 4.5, recurrence (4.10) has the solution $T(n) = \Theta(n^{\lg 7}) = O(n^{2.81})$, beating the $\Theta(n^3)$-time algorithms.

Now, let's delve into the details. Step 2 creates the following 10 matrices:

$$\begin{aligned}
S_1 &= B_{12} - B_{22} , \\
S_2 &= A_{11} + A_{12} , \\
S_3 &= A_{21} + A_{22} , \\
S_4 &= B_{21} - B_{11} , \\
S_5 &= A_{11} + A_{22} , \\
S_6 &= B_{11} + B_{22} , \\
S_7 &= A_{12} - A_{22} , \\
S_8 &= B_{21} + B_{22} , \\
S_9 &= A_{11} - A_{21} , \\
S_{10} &= B_{11} + B_{12} .
\end{aligned}$$

This step adds or subtracts $n/2 \times n/2$ matrices 10 times, taking $\Theta(n^2)$ time.

Step 3 recursively multiplies $n/2 \times n/2$ matrices 7 times to compute the following $n/2 \times n/2$ matrices, each of which is the sum or difference of products of A and B submatrices:

$$\begin{aligned}
P_1 &= A_{11} \cdot S_1 \;\; (= A_{11} \cdot B_{12} - A_{11} \cdot B_{22}) , \\
P_2 &= S_2 \cdot B_{22} \;\; (= A_{11} \cdot B_{22} + A_{12} \cdot B_{22}) , \\
P_3 &= S_3 \cdot B_{11} \;\; (= A_{21} \cdot B_{11} + A_{22} \cdot B_{11}) , \\
P_4 &= A_{22} \cdot S_4 \;\; (= A_{22} \cdot B_{21} - A_{22} \cdot B_{11}) , \\
P_5 &= S_5 \cdot S_6 \;\; (= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}) , \\
P_6 &= S_7 \cdot S_8 \;\; (= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}) , \\
P_7 &= S_9 \cdot S_{10} \;\; (= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}) .
\end{aligned}$$

The only multiplications that the algorithm performs are those in the middle column of these equations. The right-hand column just shows what these products equal in terms of the original submatrices created in step 1, but the terms are never explicitly calculated by the algorithm.

Step 4 adds to and subtracts from the four $n/2 \times n/2$ submatrices of the product C the various P_i matrices created in step 3. We start with

$$C_{11} = C_{11} + P_5 + P_4 - P_2 + P_6 .$$

Expanding the calculation on the right-hand side, with the expansion of each P_i on its own line and vertically aligning terms that cancel out, we see that the update to C_{11} equals

$$\begin{array}{llllll}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & \\
 & & - A_{22} \cdot B_{11} & & + A_{22} \cdot B_{21} & \\
 & - A_{11} \cdot B_{22} & & & & - A_{12} \cdot B_{22} \\
 & & & - A_{22} \cdot B_{22} & - A_{22} \cdot B_{21} & + A_{12} \cdot B_{22} + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} & & & & & + A_{12} \cdot B_{21} ,
\end{array}$$

which corresponds to equation (4.5). Similarly, setting

$$C_{12} = C_{12} + P_1 + P_2$$

means that the update to C_{12} equals

$$\begin{array}{lll}
A_{11} \cdot B_{12} & - A_{11} \cdot B_{22} & \\
 & + A_{11} \cdot B_{22} & + A_{12} \cdot B_{22} \\
\hline
A_{11} \cdot B_{12} & & + A_{12} \cdot B_{22} ,
\end{array}$$

corresponding to equation (4.6). Setting

$$C_{21} = C_{21} + P_3 + P_4$$

means that the update to C_{21} equals

$$\begin{array}{lll}
A_{21} \cdot B_{11} & + A_{22} \cdot B_{11} & \\
 & - A_{22} \cdot B_{11} & + A_{22} \cdot B_{21} \\
\hline
A_{21} \cdot B_{11} & & + A_{22} \cdot B_{21} ,
\end{array}$$

corresponding to equation (4.7). Finally, setting

$$C_{22} = C_{22} + P_5 + P_1 - P_3 - P_7$$

means that the update to C_{22} equals

$$
\begin{array}{lll}
A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} & & \\
\qquad\qquad - A_{11} \cdot B_{22} & & + A_{11} \cdot B_{12} \\
\qquad\qquad\qquad - A_{22} \cdot B_{11} & & \qquad\qquad - A_{21} \cdot B_{11} \\
- A_{11} \cdot B_{11} & & - A_{11} \cdot B_{12} + A_{21} \cdot B_{11} + A_{21} \cdot B_{12} \\
\hline
\qquad\qquad\qquad\qquad A_{22} \cdot B_{22} & & \qquad\qquad\qquad + A_{21} \cdot B_{12} \,,
\end{array}
$$

which corresponds to equation (4.8). Altogether, since we add or subtract $n/2 \times n/2$ matrices 12 times in step 4, this step indeed takes $\Theta(n^2)$ time.

We can see that Strassen's remarkable algorithm, comprising steps 1–4, produces the correct matrix product using 7 submatrix multiplications and 18 submatrix additions. We can also see that recurrence (4.10) characterizes its running time. Since Section 4.5 shows that this recurrence has the solution $T(n) = \Theta(n^{\lg 7}) = o(n^3)$, Strassen's method asymptotically beats the $\Theta(n^3)$ MATRIX-MULTIPLY and MATRIX-MULTIPLY-RECURSIVE procedures.

Exercises

Note: You may wish to read Section 4.5 before attempting some of these exercises.

4.2-1
Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix} .$$

Show your work.

4.2-2
Write pseudocode for Strassen's algorithm.

4.2-3
What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in $o(n^{\lg 7})$ time? What is the running time of this algorithm?

4.2-4
V. Pan discovered a way of multiplying 68×68 matrices using 132,464 multiplications, a way of multiplying 70×70 matrices using 143,640 multiplications, and a way of multiplying 72×72 matrices using 155,424 multiplications. Which method yields the best asymptotic running time when used in a divide-and-conquer matrix-multiplication algorithm? How does it compare with Strassen's algorithm?

4.2-5
Show how to multiply the complex numbers $a + bi$ and $c + di$ using only three multiplications of real numbers. The algorithm should take a, b, c, and d as input and produce the real component $ac - bd$ and the imaginary component $ad + bc$ separately.

4.2-6
Suppose that you have a $\Theta(n^\alpha)$-time algorithm for squaring $n \times n$ matrices, where $\alpha \geq 2$. Show how to use that algorithm to multiply two different $n \times n$ matrices in $\Theta(n^\alpha)$ time.

4.3 The substitution method for solving recurrences

Now that you have seen how recurrences characterize the running times of divide-and-conquer algorithms, let's learn how to solve them. We start in this section with the ***substitution method***, which is the most general of the four methods in this chapter. The substitution method comprises two steps:

1. Guess the form of the solution using symbolic constants.
2. Use mathematical induction to show that the solution works, and find the constants.

To apply the inductive hypothesis, you substitute the guessed solution for the function on smaller values—hence the name "substitution method." This method is powerful, but you must guess the form of the answer. Although generating a good guess might seem difficult, a little practice can quickly improve your intuition.

You can use the substitution method to establish either an upper or a lower bound on a recurrence. It's usually best not to try to do both at the same time. That is, rather than trying to prove a Θ-bound directly, first prove an O-bound, and then prove an Ω-bound. Together, they give you a Θ-bound (Theorem 3.1 on page 56).

As an example of the substitution method, let's determine an asymptotic upper bound on the recurrence:

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n) . \tag{4.11}$$

This recurrence is similar to recurrence (2.3) on page 41 for merge sort, except for the floor function, which ensures that $T(n)$ is defined over the integers. Let's guess that the asymptotic upper bound is the same—$T(n) = O(n \lg n)$—and use the substitution method to prove it.

We'll adopt the inductive hypothesis that $T(n) \leq cn \lg n$ for all $n \geq n_0$, where we'll choose the specific constants $c > 0$ and $n_0 > 0$ later, after we see what

constraints they need to obey. If we can establish this inductive hypothesis, we can conclude that $T(n) = O(n \lg n)$. It would be dangerous to use $T(n) = O(n \lg n)$ as the inductive hypothesis because the constants matter, as we'll see in a moment in our discussion of pitfalls.

Assume by induction that this bound holds for all numbers at least as big as n_0 and less than n. In particular, therefore, if $n \geq 2n_0$, it holds for $\lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Substituting into recurrence (4.11)—hence the name "substitution" method—yields

$$\begin{aligned}
T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + \Theta(n) \\
&\leq 2(c(n/2) \lg(n/2)) + \Theta(n) \\
&= cn \lg(n/2) + \Theta(n) \\
&= cn \lg n - cn \lg 2 + \Theta(n) \\
&= cn \lg n - cn + \Theta(n) \\
&\leq cn \lg n ,
\end{aligned}$$

where the last step holds if we constrain the constants n_0 and c to be sufficiently large that for $n \geq 2n_0$, the quantity cn dominates the anonymous function hidden by the $\Theta(n)$ term.

We've shown that the inductive hypothesis holds for the inductive case, but we also need to prove that the inductive hypothesis holds for the base cases of the induction, that is, that $T(n) \leq cn \lg n$ when $n_0 \leq n < 2n_0$. As long as $n_0 > 1$ (a new constraint on n_0), we have $\lg n > 0$, which implies that $n \lg n > 0$. So let's pick $n_0 = 2$. Since the base case of recurrence (4.11) is not stated explicitly, by our convention, $T(n)$ is algorithmic, which means that $T(2)$ and $T(3)$ are constant (as they should be if they describe the worst-case running time of any real program on inputs of size 2 or 3). Picking $c = \max\{T(2), T(3)\}$ yields $T(2) \leq c < (2 \lg 2)c$ and $T(3) \leq c < (3 \lg 3)c$, establishing the inductive hypothesis for the base cases.

Thus, we have $T(n) \leq cn \lg n$ for all $n \geq 2$, which implies that the solution to recurrence (4.11) is $T(n) = O(n \lg n)$.

In the algorithms literature, people rarely carry out their substitution proofs to this level of detail, especially in their treatment of base cases. The reason is that for most algorithmic divide-and-conquer recurrences, the base cases are all handled in pretty much the same way. You ground the induction on a range of values from a convenient positive constant n_0 up to some constant $n_0' > n_0$ such that for $n \geq n_0'$, the recurrence always bottoms out in a constant-sized base case between n_0 and n_0'. (This example used $n_0' = 2n_0$.) Then, it's usually apparent, without spelling out the details, that with a suitably large choice of the leading constant (such as c for this example), the inductive hypothesis can be made to hold for all the values in the range from n_0 to n_0'.

Making a good guess

Unfortunately, there is no general way to correctly guess the tightest asymptotic solution to an arbitrary recurrence. Making a good guess takes experience and, occasionally, creativity. Fortunately, learning some recurrence-solving heuristics, as well as playing around with recurrences to gain experience, can help you become a good guesser. You can also use recursion trees, which we'll see in Section 4.4, to help generate good guesses.

If a recurrence is similar to one you've seen before, then guessing a similar solution is reasonable. As an example, consider the recurrence

$$T(n) = 2T(n/2 + 17) + \Theta(n) ,$$

defined on the reals. This recurrence looks somewhat like the merge-sort recurrence (2.3), but it's more complicated because of the added "17" in the argument to T on the right-hand side. Intuitively, however, this additional term shouldn't substantially affect the solution to the recurrence. When n is large, the relative difference between $n/2$ and $n/2 + 17$ is not that large: both cut n nearly in half. Consequently, it makes sense to guess that $T(n) = O(n \lg n)$, which you can verify is correct using the substitution method (see Exercise 4.3-1).

Another way to make a good guess is to determine loose upper and lower bounds on the recurrence and then reduce your range of uncertainty. For example, you might start with a lower bound of $T(n) = \Omega(n)$ for recurrence (4.11), since the recurrence includes the term $\Theta(n)$, and you can prove an initial upper bound of $T(n) = O(n^2)$. Then split your time between trying to lower the upper bound and trying to raise the lower bound until you converge on the correct, asymptotically tight solution, which in this case is $T(n) = \Theta(n \lg n)$.

A trick of the trade: subtracting a low-order term

Sometimes, you might correctly guess a tight asymptotic bound on the solution of a recurrence, but somehow the math fails to work out in the induction proof. The problem frequently turns out to be that the inductive assumption is not strong enough. The trick to resolving this problem is to revise your guess by *subtracting* a lower-order term when you hit such a snag. The math then often goes through.

Consider the recurrence

$$T(n) = 2T(n/2) + \Theta(1) \tag{4.12}$$

defined on the reals. Let's guess that the solution is $T(n) = O(n)$ and try to show that $T(n) \le cn$ for $n \ge n_0$, where we choose the constants $c, n_0 > 0$ suitably. Substituting our guess into the recurrence, we obtain

$$\begin{aligned} T(n) &\le 2(c(n/2)) + \Theta(1) \\ &= cn + \Theta(1) , \end{aligned}$$

which, unfortunately, does not imply that $T(n) \le cn$ for *any* choice of c. We might be tempted to try a larger guess, say $T(n) = O(n^2)$. Although this larger guess works, it provides only a loose upper bound. It turns out that our original guess of $T(n) = O(n)$ is correct and tight. In order to show that it is correct, however, we must strengthen our inductive hypothesis.

Intuitively, our guess is nearly right: we are off only by $\Theta(1)$, a lower-order term. Nevertheless, mathematical induction requires us to prove the *exact* form of the inductive hypothesis. Let's try our trick of subtracting a lower-order term from our previous guess: $T(n) \le cn - d$, where $d \ge 0$ is a constant. We now have

$$\begin{aligned} T(n) &\le 2(c(n/2) - d) + \Theta(1) \\ &= cn - 2d + \Theta(1) \\ &\le cn - d - (d - \Theta(1)) \\ &\le cn - d \end{aligned}$$

as long as we choose d to be larger than the anonymous upper-bound constant hidden by the Θ-notation. Subtracting a lower-order term works! Of course, we must not forget to handle the base case, which is to choose the constant c large enough that $cn - d$ dominates the implicit base cases.

You might find the idea of subtracting a lower-order term to be counterintuitive. After all, if the math doesn't work out, shouldn't you increase your guess? Not necessarily! When the recurrence contains more than one recursive invocation (recurrence (4.12) contains two), if you add a lower-order term to the guess, then you end up adding it once for each of the recursive invocations. Doing so takes you even further away from the inductive hypothesis. On the other hand, if you subtract a lower-order term from the guess, then you get to subtract it once for each of the recursive invocations. In the above example, we subtracted the constant d twice because the coefficient of $T(n/2)$ is 2. We ended up with the inequality $T(n) \le cn - d - (d - \Theta(1))$, and we readily found a suitable value for d.

Avoiding pitfalls

Avoid using asymptotic notation in the inductive hypothesis for the substitution method because it's error prone. For example, for recurrence (4.11), we can falsely "prove" that $T(n) = O(n)$ if we unwisely adopt $T(n) = O(n)$ as our inductive hypothesis:

$$\begin{aligned} T(n) &\le 2 \cdot O(\lfloor n/2 \rfloor) + \Theta(n) \\ &= 2 \cdot O(n) + \Theta(n) \\ &= O(n) . \qquad \Longleftarrow \textit{wrong!} \end{aligned}$$

The problem with this reasoning is that the constant hidden by the O-notation changes. We can expose the fallacy by repeating the "proof" using an explicit constant. For the inductive hypothesis, assume that $T(n) \le cn$ for all $n \ge n_0$, where $c, n_0 > 0$ are constants. Repeating the first two steps in the inequality chain yields

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor) + \Theta(n) \\ &\le cn + \Theta(n) . \end{aligned}$$

Now, indeed $cn + \Theta(n) = O(n)$, but the constant hidden by the O-notation must be larger than c because the anonymous function hidden by the $\Theta(n)$ is asymptotically positive. We cannot take the third step to conclude that $cn + \Theta(n) \le cn$, thus exposing the fallacy.

When using the substitution method, or more generally mathematical induction, you must be careful that the constants hidden by any asymptotic notation are the same constants throughout the proof. Consequently, it's best to avoid asymptotic notation in your inductive hypothesis and to name constants explicitly.

Here's another fallacious use of the substitution method to show that the solution to recurrence (4.11) is $T(n) = O(n)$. We guess $T(n) \le cn$ and then argue

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor) + \Theta(n) \\ &\le cn + \Theta(n) \\ &= O(n) , \qquad \Longleftarrow \textit{wrong!} \end{aligned}$$

since c is a positive constant. The mistake stems from the difference between our goal—to prove that $T(n) = O(n)$—and our inductive hypothesis—to prove that $T(n) \le cn$. When using the substitution method, or in any inductive proof, you must prove the *exact* statement of the inductive hypothesis. In this case, we must explicitly prove that $T(n) \le cn$ to show that $T(n) = O(n)$.

Exercises

4.3-1

Use the substitution method to show that each of the following recurrences defined on the reals has the asymptotic solution specified:

a. $T(n) = T(n-1) + n$ has solution $T(n) = O(n^2)$.

b. $T(n) = T(n/2) + \Theta(1)$ has solution $T(n) = O(\lg n)$.

c. $T(n) = 2T(n/2) + n$ has solution $T(n) = \Theta(n \lg n)$.

d. $T(n) = 2T(n/2 + 17) + n$ has solution $T(n) = O(n \lg n)$.

e. $T(n) = 2T(n/3) + \Theta(n)$ has solution $T(n) = \Theta(n)$.

f. $T(n) = 4T(n/2) + \Theta(n)$ has solution $T(n) = \Theta(n^2)$.

4.3-2
The solution to the recurrence $T(n) = 4T(n/2) + n$ turns out to be $T(n) = \Theta(n^2)$. Show that a substitution proof with the assumption $T(n) \le cn^2$ fails. Then show how to subtract a lower-order term to make a substitution proof work.

4.3-3
The recurrence $T(n) = 2T(n-1) + 1$ has the solution $T(n) = O(2^n)$. Show that a substitution proof fails with the assumption $T(n) \le c2^n$, where $c > 0$ is constant. Then show how to subtract a lower-order term to make a substitution proof work.

4.4 The recursion-tree method for solving recurrences

Although you can use the substitution method to prove that a solution to a recurrence is correct, you might have trouble coming up with a good guess. Drawing out a recursion tree, as we did in our analysis of the merge-sort recurrence in Section 2.3.2, can help. In a ***recursion tree***, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. You typically sum the costs within each level of the tree to obtain the per-level costs, and then you sum all the per-level costs to determine the total cost of all levels of the recursion. Sometimes, however, adding up the total cost takes more creativity.

A recursion tree is best used to generate intuition for a good guess, which you can then verify by the substitution method. If you are meticulous when drawing out a recursion tree and summing the costs, however, you can use a recursion tree as a direct proof of a solution to a recurrence. But if you use it only to generate a good guess, you can often tolerate a small amount of "sloppiness," which can simplify the math. When you verify your guess with the substitution method later on, your math should be precise. This section demonstrates how you can use recursion trees to solve recurrences, generate good guesses, and gain intuition for recurrences.

An illustrative example

Let's see how a recursion tree can provide a good guess for an upper-bound solution to the recurrence

$$T(n) = 3T(n/4) + \Theta(n^2) \; . \tag{4.13}$$

Figure 4.1 shows how to derive the recursion tree for $T(n) = 3T(n/4) + cn^2$, where the constant $c > 0$ is the upper-bound constant in the $\Theta(n^2)$ term. Part (a) of the figure shows $T(n)$, which part (b) expands into an equivalent tree representing the recurrence. The cn^2 term at the root represents the cost at the top level of recursion, and the three subtrees of the root represent the costs incurred by the

Figure 4.1 Constructing a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in **(d)** has height $\log_4 n$.

subproblems of size $n/4$. Part (c) shows this process carried one step further by expanding each node with cost $T(n/4)$ from part (b). The cost for each of the three children of the root is $c(n/4)^2$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence.

Because subproblem sizes decrease by a factor of 4 every time we go down one level, the recursion must eventually bottom out in a base case where $n < n_0$. By convention, the base case is $T(n) = \Theta(1)$ for $n < n_0$, where $n_0 > 0$ is any threshold constant sufficiently large that the recurrence is well defined. For the purpose of intuition, however, let's simplify the math a little. Let's assume that n is an exact power of 4 and that the base case is $T(1) = \Theta(1)$. As it turns out, these assumptions don't affect the asymptotic solution.

What's the height of the recursion tree? The subproblem size for a node at depth i is $n/4^i$. As we descend the tree from the root, the subproblem size hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$. Thus, the tree has internal nodes at depths $0, 1, 2, \ldots, \log_4 n - 1$ and leaves at depth $\log_4 n$.

Part (d) of Figure 4.1 shows the cost at each level of the tree. Each level has three times as many nodes as the level above, and so the number of nodes at depth i is 3^i. Because subproblem sizes reduce by a factor of 4 for each level further from the root, each internal node at depth $i = 0, 1, 2, \ldots, \log_4 n - 1$ has a cost of $c(n/4^i)^2$. Multiplying, we see that the total cost of all nodes at a given depth i is $3^i c(n/4^i)^2 = (3/16)^i cn^2$. The bottom level, at depth $\log_4 n$, contains $3^{\log_4 n} = n^{\log_4 3}$ leaves (using equation (3.21) on page 66). Each leaf contributes $\Theta(1)$, leading to a total leaf cost of $\Theta(n^{\log_4 3})$.

Now we add up the costs over all levels to determine the cost for the entire tree:

$$\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) && \text{(by equation (A.7) on page 1142)} \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) && (\Theta(n^{\log_4 3}) = O(n^{0.8}) = O(n^2)) .
\end{aligned}$$

We've derived the guess of $T(n) = O(n^2)$ for the original recurrence. In this example, the coefficients of cn^2 form a decreasing geometric series. By equation (A.7), the sum of these coefficients is bounded from above by the constant $16/13$. Since

the root's contribution to the total cost is cn^2, the cost of the root dominates the total cost of the tree.

In fact, if $O(n^2)$ is indeed an upper bound for the recurrence (as we'll verify in a moment), then it must be a tight bound. Why? The first recursive call contributes a cost of $\Theta(n^2)$, and so $\Omega(n^2)$ must be a lower bound for the recurrence.

Let's now use the substitution method to verify that our guess is correct, namely, that $T(n) = O(n^2)$ is an upper bound for the recurrence $T(n) = 3T(n/4)+\Theta(n^2)$. We want to show that $T(n) \le dn^2$ for some constant $d > 0$. Using the same constant $c > 0$ as before, we have

$$\begin{aligned} T(n) &\le 3T(n/4) + cn^2 \\ &\le 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}\,dn^2 + cn^2 \\ &\le dn^2 , \end{aligned}$$

where the last step holds if we choose $d \ge (16/13)c$.

For the base case of the induction, let $n_0 > 0$ be a sufficiently large threshold constant that the recurrence is well defined when $T(n) = \Theta(1)$ for $n < n_0$. We can pick d large enough that d dominates the constant hidden by the Θ, in which case $dn^2 \ge d \ge T(n)$ for $1 \le n < n_0$, completing the proof of the base case.

The substitution proof we just saw involves two named constants, c and d. We named c and used it to stand for the upper-bound constant hidden and guaranteed to exist by the Θ-notation. We cannot pick c arbitrarily—it's given to us—although, for any such c, any constant $c' \ge c$ also suffices. We also named d, but we were free to choose any value for it that fit our needs. In this example, the value of d happened to depend on the value of c, which is fine, since d is constant if c is constant.

An irregular example

Let's find an asymptotic upper bound for another, more irregular, example. Figure 4.2 shows the recursion tree for the recurrence

$$T(n) = T(n/3) + T(2n/3) + \Theta(n) . \tag{4.14}$$

This recursion tree is unbalanced, with different root-to-leaf paths having different lengths. Going left at any node produces a subproblem of one-third the size, and going right produces a subproblem of two-thirds the size. Let $n_0 > 0$ be the implicit threshold constant such that $T(n) = \Theta(1)$ for $0 < n < n_0$, and let c represent the upper-bound constant hidden by the $\Theta(n)$ term for $n \ge n_0$. There are actually two n_0 constants here—one for the threshold in the recurrence, and the other for the threshold in the Θ-notation, so we'll let n_0 be the larger of the two constants.

Figure 4.2 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

The height of the tree runs down the right edge of the tree, corresponding to subproblems of sizes $n, (2/3)n, (4/9)n, \ldots, \Theta(1)$ with costs bounded by $cn, c(2n/3), c(4n/9), \ldots, \Theta(1)$, respectively. We hit the rightmost leaf when $(2/3)^h n < n_0 \le (2/3)^{h-1} n$, which happens when $h = \lfloor \log_{3/2}(n/n_0) \rfloor + 1$ since, applying the floor bounds in equation (3.2) on page 64 with $x = \log_{3/2}(n/n_0)$, we have $(2/3)^h n = (2/3)^{\lfloor x \rfloor + 1} n < (2/3)^x n = (n_0/n)n = n_0$ and $(2/3)^{h-1} n = (2/3)^{\lfloor x \rfloor} n > (2/3)^x n = (n_0/n)n = n_0$. Thus, the height of the tree is $h = \Theta(\lg n)$.

We're now in a position to understand the upper bound. Let's postpone dealing with the leaves for a moment. Summing the costs of internal nodes across each level, we have at most cn per level times the $\Theta(\lg n)$ tree height for a total cost of $O(n \lg n)$ for all internal nodes.

It remains to deal with the leaves of the recursion tree, which represent base cases, each costing $\Theta(1)$. How many leaves are there? It's tempting to upper-bound their number by the number of leaves in a complete binary tree of height $h = \lfloor \log_{3/2}(n/n_0) \rfloor + 1$, since the recursion tree is contained within such a complete binary tree. But this approach turns out to give us a poor bound. The complete binary tree has 1 node at the root, 2 nodes at depth 1, and generally 2^k nodes at depth k. Since the height is $h = \lfloor \log_{3/2} n \rfloor + 1$, there are

$2^h = 2^{\lfloor \log_{3/2} n \rfloor + 1} \le 2n^{\log_{3/2} 2}$ leaves in the complete binary tree, which is an upper bound on the number of leaves in the recursion tree. Because the cost of each leaf is $\Theta(1)$, this analysis says that the total cost of all leaves in the recursion tree is $O(n^{\log_{3/2} 2}) = O(n^{1.71})$, which is an asymptotically greater bound than the $O(n \lg n)$ cost of all internal nodes. In fact, as we're about to see, this bound is not tight. The cost of all leaves in the recursion tree is $O(n)$—asymptotically *less* than $O(n \lg n)$. In other words, the cost of the internal nodes dominates the cost of the leaves, not vice versa.

Rather than analyzing the leaves, we could quit right now and prove by substitution that $T(n) = \Theta(n \lg n)$. This approach works (see Exercise 4.4-3), but it's instructive to understand how many leaves this recursion tree has. You may see recurrences for which the cost of leaves dominates the cost of internal nodes, and then you'll be in better shape if you've had some experience analyzing the number of leaves.

To figure out how many leaves there really are, let's write a recurrence $L(n)$ for the number of leaves in the recursion tree for $T(n)$. Since all the leaves in $T(n)$ belong either to the left subtree or the right subtree of the root, we have

$$L(n) = \begin{cases} 1 & \text{if } n < n_0 , \\ L(n/3) + L(2n/3) & \text{if } n \ge n_0 . \end{cases} \tag{4.15}$$

This recurrence is similar to recurrence (4.14), but it's missing the $\Theta(n)$ term, and it contains an explicit base case. Because this recurrence omits the $\Theta(n)$ term, it is much easier to solve. Let's apply the substitution method to show that it has solution $L(n) = O(n)$. Using the inductive hypothesis $L(n) \le dn$ for some constant $d > 0$, and assuming that the inductive hypothesis holds for all values less than n, we have

$$\begin{aligned} L(n) &= L(n/3) + L(2n/3) \\ &\le dn/3 + 2(dn)/3 \\ &\le dn , \end{aligned}$$

which holds for any $d > 0$. We can now choose d large enough to handle the base case $L(n) = 1$ for $0 < n < n_0$, for which $d = 1$ suffices, thereby completing the substitution method for the upper bound on leaves. (Exercise 4.4-2 asks you to prove that $L(n) = \Theta(n)$.)

Returning to recurrence (4.14) for $T(n)$, it now becomes apparent that the total cost of leaves over all levels must be $L(n) \cdot \Theta(1) = \Theta(n)$. Since we have derived the bound of $O(n \lg n)$ on the cost of the internal nodes, it follows that the solution to recurrence (4.14) is $T(n) = O(n \lg n) + \Theta(n) = O(n \lg n)$. (Exercise 4.4-3 asks you to prove that $T(n) = \Theta(n \lg n)$.)

It's wise to verify any bound obtained with a recursion tree by using the substitution method, especially if you've made simplifying assumptions. But another

strategy altogether is to use more-powerful mathematics, typically in the form of the master method in the next section (which unfortunately doesn't apply to recurrence (4.14)) or the Akra-Bazzi method (which does, but requires calculus). Even if you use a powerful method, a recursion tree can improve your intuition for what's going on beneath the heavy math.

Exercises

4.4-1
For each of the following recurrences, sketch its recursion tree, and guess a good asymptotic upper bound on its solution. Then use the substitution method to verify your answer.

a. $T(n) = T(n/2) + n^3$.

b. $T(n) = 4T(n/3) + n$.

c. $T(n) = 4T(n/2) + n$.

d. $T(n) = 3T(n-1) + 1$.

4.4-2
Use the substitution method to prove that recurrence (4.15) has the asymptotic lower bound $L(n) = \Omega(n)$. Conclude that $L(n) = \Theta(n)$.

4.4-3
Use the substitution method to prove that recurrence (4.14) has the solution $T(n) = \Omega(n \lg n)$. Conclude that $T(n) = \Theta(n \lg n)$.

4.4-4
Use a recursion tree to justify a good guess for the solution to the recurrence $T(n) = T(\alpha n) + T((1-\alpha)n) + \Theta(n)$, where α is a constant in the range $0 < \alpha < 1$.

4.5 The master method for solving recurrences

The master method provides a "cookbook" method for solving algorithmic recurrences of the form

$$T(n) = aT(n/b) + f(n) , \tag{4.16}$$

where $a > 0$ and $b > 1$ are constants. We call $f(n)$ a ***driving function***, and we call a recurrence of this general form a ***master recurrence***. To use the master method, you need to memorize three cases, but then you'll be able to solve many master recurrences quite easily.

A master recurrence describes the running time of a divide-and-conquer algorithm that divides a problem of size n into a subproblems, each of size $n/b < n$. The algorithm solves the a subproblems recursively, each in $T(n/b)$ time. The driving function $f(n)$ encompasses the cost of dividing the problem before the recursion, as well as the cost of combining the results of the recursive solutions to subproblems. For example, the recurrence arising from Strassen's algorithm is a master recurrence with $a = 7$, $b = 2$, and driving function $f(n) = \Theta(n^2)$.

As we have mentioned, in solving a recurrence that describes the running time of an algorithm, one technicality that we'd often prefer to ignore is the requirement that the input size n be an integer. For example, we saw that the running time of merge sort can be described by recurrence (2.3), $T(n) = 2T(n/2) + \Theta(n)$, on page 41. But if n is an odd number, we really don't have two problems of exactly half the size. Rather, to ensure that the problem sizes are integers, we round one subproblem down to size $\lfloor n/2 \rfloor$ and the other up to size $\lceil n/2 \rceil$, so the true recurrence is $T(n) = T(\lceil n/2 \rceil + T(\lfloor n/2 \rfloor) + \Theta(n)$. But this floors-and-ceilings recurrence is longer to write and messier to deal with than recurrence (2.3), which is defined on the reals. We'd rather not worry about floors and ceilings, if we don't have to, especially since the two recurrences have the same $\Theta(n \lg n)$ solution.

The master method allows you to state a master recurrence without floors and ceilings and implicitly infer them. No matter how the arguments are rounded up or down to the nearest integer, the asymptotic bounds that it provides remain the same. Moreover, as we'll see in Section 4.6, if you define your master recurrence on the reals, without implicit floors and ceilings, the asymptotic bounds still don't change. Thus you can ignore floors and ceilings for master recurrences. Section 4.7 gives sufficient conditions for ignoring floors and ceilings in more general divide-and-conquer recurrences.

The master theorem

The master method depends upon the following theorem.

Theorem 4.1 (Master theorem)
Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by

$$T(n) = aT(n/b) + f(n) , \tag{4.17}$$

where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a' \geq 0$ and $a'' \geq 0$ satisfying $a = a' + a''$. Then the asymptotic behavior of $T(n)$ can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Before applying the master theorem to some examples, let's spend a few moments to understand broadly what it says. The function $n^{\log_b a}$ is called the ***watershed function***. In each of the three cases, we compare the driving function $f(n)$ to the watershed function $n^{\log_b a}$. Intuitively, if the watershed function grows asymptotically faster than the driving function, then case 1 applies. Case 2 applies if the two functions grow at nearly the same asymptotic rate. Case 3 is the "opposite" of case 1, where the driving function grows asymptotically faster than the watershed function. But the technical details matter.

In case 1, not only must the watershed function grow asymptotically faster than the driving function, it must grow *polynomially* faster. That is, the watershed function $n^{\log_b a}$ must be asymptotically larger than the driving function $f(n)$ by at least a factor of $\Theta(n^\epsilon)$ for some constant $\epsilon > 0$. The master theorem then says that the solution is $T(n) = \Theta(n^{\log_b a})$. In this case, if we look at the recursion tree for the recurrence, the cost per level grows at least geometrically from root to leaves, and the total cost of leaves dominates the total cost of the internal nodes.

In case 2, the watershed and driving functions grow at nearly the same asymptotic rate. But more specifically, the driving function grows faster than the watershed function by a factor of $\Theta(\lg^k n)$, where $k \geq 0$. The master theorem says that we tack on an extra $\lg n$ factor to $f(n)$, yielding the solution $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. In this case, each level of the recursion tree costs approximately the same—$\Theta(n^{\log_b a} \lg^k n)$—and there are $\Theta(\lg n)$ levels. In practice, the most common situation for case 2 occurs when $k = 0$, in which case the watershed and driving functions have the same asymptotic growth, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n)$.

Case 3 mirrors case 1. Not only must the driving function grow asymptotically faster than the watershed function, it must grow *polynomially* faster. That is, the driving function $f(n)$ must be asymptotically larger than the watershed function $n^{\log_b a}$ by at least a factor of $\Theta(n^\epsilon)$ for some constant $\epsilon > 0$. Moreover, the driving function must satisfy the regularity condition that $af(n/b) \leq cf(n)$. This condition is satisfied by most of the polynomially bounded functions that you're likely to encounter when applying case 3. The regularity condition might not be satisfied

if the driving function grows slowly in local areas, yet relatively quickly overall. (Exercise 4.5-5 gives an example of such a function.) For case 3, the master theorem says that the solution is $T(n) = \Theta(f(n))$. If we look at the recursion tree, the cost per level drops at least geometrically from the root to the leaves, and the root cost dominates the cost of all other nodes.

It's worth looking again at the requirement that there be polynomial separation between the watershed function and the driving function for either case 1 or case 3 to apply. The separation doesn't need to be much, but it must be there, and it must grow polynomially. For example, for the recurrence $T(n) = 4T(n/2) + n^{1.99}$ (admittedly not a recurrence you're likely to see when analyzing an algorithm), the watershed function is $n^{\log_b a} = n^2$. Hence the driving function $f(n) = n^{1.99}$ is polynomially smaller by a factor of $n^{0.01}$. Thus case 1 applies with $\epsilon = 0.01$.

Using the master method

To use the master method, you determine which case (if any) of the master theorem applies and write down the answer.

As a first example, consider the recurrence $T(n) = 9T(n/3) + n$. For this recurrence, we have $a = 9$ and $b = 3$, which implies that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = n = O(n^{2-\epsilon})$ for any constant $\epsilon \le 1$, we can apply case 1 of the master theorem to conclude that the solution is $T(n) = \Theta(n^2)$.

Now consider the recurrence $T(n) = T(2n/3) + 1$, which has $a = 1$ and $b = 3/2$, which means that the watershed function is $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies since $f(n) = 1 = \Theta(n^{\log_b a} \lg^0 n) = \Theta(1)$. The solution to the recurrence is $T(n) = \Theta(\lg n)$.

For the recurrence $T(n) = 3T(n/4) + n \lg n$, we have $a = 3$ and $b = 4$, which means that $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = n \lg n = \Omega(n^{\log_4 3 + \epsilon})$, where ϵ can be as large as approximately 0.2, case 3 applies as long as the regularity condition holds for $f(n)$. It does, because for sufficiently large n, we have that $af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n \lg n = cf(n)$ for $c = 3/4$. By case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

Next, let's look at the recurrence $T(n) = 2T(n/2) + n \lg n$, where we have $a = 2$, $b = 2$, and $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies since $f(n) = n \lg n = \Theta(n^{\log_b a} \lg^1 n)$. We conclude that the solution is $T(n) = \Theta(n \lg^2 n)$.

We can use the master method to solve the recurrences we saw in Sections 2.3.2, 4.1, and 4.2.

Recurrence (2.3), $T(n) = 2T(n/2) + \Theta(n)$, on page 41, characterizes the running time of merge sort. Since $a = 2$ and $b = 2$, the watershed function is $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies because $f(n) = \Theta(n)$, and the solution is $T(n) = \Theta(n \lg n)$.

Recurrence (4.9), $T(n) = 8T(n/2) + \Theta(1)$, on page 84, describes the running time of the simple recursive algorithm for matrix multiplication. We have $a = 8$ and $b = 2$, which means that the watershed function is $n^{\log_b a} = n^{\log_2 8} = n^3$. Since n^3 is polynomially larger than the driving function $f(n) = \Theta(1)$—indeed, we have $f(n) = O(n^{3-\epsilon})$ for any positive $\epsilon < 3$—case 1 applies. We conclude that $T(n) = \Theta(n^3)$.

Finally, recurrence (4.10), $T(n) = 7T(n/2) + \Theta(n^2)$, on page 87, arose from the analysis of Strassen's algorithm for matrix multiplication. For this recurrence, we have $a = 7$ and $b = 2$, and the watershed function is $n^{\log_b a} = n^{\lg 7}$. Observing that $\lg 7 = 2.807355\ldots$, we can let $\epsilon = 0.8$ and bound the driving function $f(n) = \Theta(n^2) = O(n^{\lg 7 - \epsilon})$. Case 1 applies with solution $T(n) = \Theta(n^{\lg 7})$.

When the master method doesn't apply

There are situations where you can't use the master theorem. For example, it can be that the watershed function and the driving function cannot be asymptotically compared. We might have that $f(n) \gg n^{\log_b a}$ for an infinite number of values of n but also that $f(n) \ll n^{\log_b a}$ for an infinite number of different values of n. As a practical matter, however, most of the driving functions that arise in the study of algorithms can be meaningfully compared with the watershed function. If you encounter a master recurrence for which that's not the case, you'll have to resort to substitution or other methods.

Even when the relative growths of the driving and watershed functions can be compared, the master theorem does not cover all the possibilities. There is a gap between cases 1 and 2 when $f(n) = o(n^{\log_b a})$, yet the watershed function does not grow polynomially faster than the driving function. Similarly, there is a gap between cases 2 and 3 when $f(n) = \omega(n^{\log_b a})$ and the driving function grows more than polylogarithmically faster than the watershed function, but it does not grow polynomially faster. If the driving function falls into one of these gaps, or if the regularity condition in case 3 fails to hold, you'll need to use something other than the master method to solve the recurrence.

As an example of a driving function falling into a gap, consider the recurrence $T(n) = 2T(n/2) + n/\lg n$. Since $a = 2$ and $b = 2$, the watershed function is $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. The driving function is $n/\lg n = o(n)$, which means that it grows asymptotically more slowly than the watershed function n. But $n/\lg n$ grows only *logarithmically* slower than n, not *polynomially* slower. More precisely, equation (3.24) on page 67 says that $\lg n = o(n^\epsilon)$ for any constant $\epsilon > 0$, which means that $1/\lg n = \omega(n^{-\epsilon})$ and $n/\lg n = \omega(n^{1-\epsilon}) = \omega(n^{\log_b a - \epsilon})$. Thus no constant $\epsilon > 0$ exists such that $n/\lg n = O(n^{\log_b a - \epsilon})$, which is required for case 1 to apply. Case 2 fails to apply as well, since $n/\lg n = \Theta(n^{\log_b a} \lg^k n)$, where $k = -1$, but k must be nonnegative for case 2 to apply.

To solve this kind of recurrence, you must use another method, such as the substitution method (Section 4.3) or the Akra-Bazzi method (Section 4.7). (Exercise 4.6-3 asks you to show that the answer is $\Theta(n \lg \lg n)$.) Although the master theorem doesn't handle this particular recurrence, it does handle the overwhelming majority of recurrences that tend to arise in practice.

Exercises

4.5-1

Use the master method to give tight asymptotic bounds for the following recurrences.

a. $T(n) = 2T(n/4) + 1$.

b. $T(n) = 2T(n/4) + \sqrt{n}$.

c. $T(n) = 2T(n/4) + \sqrt{n} \lg^2 n$.

d. $T(n) = 2T(n/4) + n$.

e. $T(n) = 2T(n/4) + n^2$.

4.5-2

Professor Caesar wants to develop a matrix-multiplication algorithm that is asymptotically faster than Strassen's algorithm. His algorithm will use the divide-and-conquer method, dividing each matrix into $n/4 \times n/4$ submatrices, and the divide and combine steps together will take $\Theta(n^2)$ time. Suppose that the professor's algorithm creates a recursive subproblems of size $n/4$. What is the largest integer value of a for which his algorithm could possibly run asymptotically faster than Strassen's?

4.5-3

Use the master method to show that the solution to the binary-search recurrence $T(n) = T(n/2) + \Theta(1)$ is $T(n) = \Theta(\lg n)$. (See Exercise 2.3-6 for a description of binary search.)

4.5-4

Consider the function $f(n) = \lg n$. Argue that although $f(n/2) < f(n)$, the regularity condition $af(n/b) \le cf(n)$ with $a = 1$ and $b = 2$ does not hold for any constant $c < 1$. Argue further that for any $\epsilon > 0$, the condition in case 3 that $f(n) = \Omega(n^{\log_b a + \epsilon})$ does not hold.

4.5-5

Show that for suitable constants a, b, and ϵ, the function $f(n) = 2^{\lceil \lg n \rceil}$ satisfies all the conditions in case 3 of the master theorem except the regularity condition.

★ 4.6 Proof of the continuous master theorem

Proving the master theorem (Theorem 4.1) in its full generality, especially dealing with the knotty technical issue of floors and ceilings, is beyond the scope of this book. This section, however, states and proves a variant of the master theorem, called the ***continuous master theorem***[1] in which the master recurrence (4.17) is defined over sufficiently large positive real numbers. The proof of this version, uncomplicated by floors and ceilings, contains the main ideas needed to understand how master recurrences behave. Section 4.7 discusses floors and ceilings in divide-and-conquer recurrences at greater length, presenting sufficient conditions for them not to affect the asymptotic solutions.

Of course, since you need not understand the proof of the master theorem in order to apply the master method, you may choose to skip this section. But if you wish to study more-advanced algorithms beyond the scope of this textbook, you may appreciate a better understanding of the underlying mathematics, which the proof of the continuous master theorem provides.

Although we usually assume that recurrences are algorithmic and don't require an explicit statement of a base case, we must be much more careful for proofs that justify the practice. The lemmas and theorem in this section explicitly state the base cases, because the inductive proofs require mathematical grounding. It is common in the world of mathematics to be extraordinarily careful proving theorems that justify acting more casually in practice.

The proof of the continuous master theorem involves two lemmas. Lemma 4.2 uses a slightly simplified master recurrence with a threshold constant of $n_0 = 1$, rather than the more general $n_0 > 0$ threshold constant implied by the unstated base case. The lemma employs a recursion tree to reduce the solution of the simplified master recurrence to that of evaluating a summation. Lemma 4.3 then provides asymptotic bounds for the summation, mirroring the three cases of the master theorem. Finally, the continuous master theorem itself (Theorem 4.4) gives asymptotic bounds for master recurrences, while generalizing to an arbitrary threshold constant $n_0 > 0$ as implied by the unstated base case.

[1] This terminology does not mean that either $T(n)$ or $f(n)$ need be continuous, only that the domain of $T(n)$ is the real numbers, as opposed to integers.

Some of the proofs use the properties described in Problem 3-5 on pages 72–73 to combine and simplify complicated asymptotic expressions. Although Problem 3-5 addresses only Θ-notation, the properties enumerated there can be extended to O-notation and Ω-notation as well.

Here's the first lemma.

Lemma 4.2

Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a function defined over real numbers $n \geq 1$. Then the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } 0 \leq n < 1 , \\ aT(n/b) + f(n) & \text{if } n \geq 1 \end{cases}$$

has solution

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) . \tag{4.18}$$

Proof Consider the recursion tree in Figure 4.3. Let's look first at its internal nodes. The root of the tree has cost $f(n)$, and it has a children, each with cost $f(n/b)$. (It is convenient to think of a as being an integer, especially when visualizing the recursion tree, but the mathematics does not require it.) Each of these children has a children, making a^2 nodes at depth 2, and each of the a children has cost $f(n/b^2)$. In general, there are a^j nodes at depth j, and each node has cost $f(n/b^j)$.

Now, let's move on to understanding the leaves. The tree grows downward until n/b^j becomes less than 1. Thus, the tree has height $\lfloor \log_b n \rfloor + 1$, because $n/b^{\lfloor \log_b n \rfloor} \geq n/b^{\log_b n} = 1$ and $n/b^{\lfloor \log_b n \rfloor + 1} < n/b^{\log_b n} = 1$. Since, as we have observed, the number of nodes at depth j is a^j and all the leaves are at depth $\lfloor \log_b n \rfloor + 1$, the tree contains $a^{\lfloor \log_b n \rfloor + 1}$ leaves. Using the identity (3.21) on page 66, we have $a^{\lfloor \log_b n \rfloor + 1} \leq a^{\log_b n + 1} = an^{\log_b a} = O(n^{\log_b a})$, since a is constant, and $a^{\lfloor \log_b n \rfloor + 1} \geq a^{\log_b n} = n^{\log_b a} = \Omega(n^{\log_b a})$. Consequently, the total number of leaves is $\Theta(n^{\log_b a})$—asymptotically, the watershed function.

We are now in a position to derive equation (4.18) by summing the costs of the nodes at each depth in the tree, as shown in the figure. The first term in the equation is the total costs of the leaves. Since each leaf is at depth $\lfloor \log_b n \rfloor + 1$ and $n/b^{\lfloor \log_b n \rfloor + 1} < 1$, the base case of the recurrence gives the cost of a leaf: $T(n/b^{\lfloor \log_b n \rfloor + 1}) = \Theta(1)$. Hence the cost of all $\Theta(n^{\log_b a})$ leaves is $\Theta(n^{\log_b a}) \cdot \Theta(1) = \Theta(n^{\log_b a})$ by Problem 3-5(d). The second term in equation (4.18) is the cost of the internal nodes, which, in the underlying divide-and-conquer algorithm, represents the costs of dividing problems into subproblems and

Figure 4.3 The recursion tree generated by $T(n) = aT(n/b) + f(n)$. The tree is a complete a-ary tree with $a^{\lfloor \log_b n \rfloor + 1}$ leaves and height $\lfloor \log_b n \rfloor + 1$. The cost of the nodes at each depth is shown at the right, and their sum is given in equation (4.18).

then recombining the subproblems. Since the cost for all the internal nodes at depth j is $a^j f(n/b^j)$, the total cost of all internal nodes is

$$\sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) \; . \qquad \blacksquare$$

As we'll see, the three cases of the master theorem depend on the distribution of the total cost across levels of the recursion tree:

Case 1: The costs increase geometrically from the root to the leaves, growing by a constant factor with each level.

Case 2: The costs depend on the value of k in the theorem. With $k = 0$, the costs are equal for each level; with $k = 1$, the costs grow linearly from the root to the leaves; with $k = 2$, the growth is quadratic; and in general, the costs grow polynomially in k.

Case 3: The costs decrease geometrically from the root to the leaves, shrinking by a constant factor with each level.

The summation in equation (4.18) describes the cost of the dividing and combining steps in the underlying divide-and-conquer algorithm. The next lemma provides asymptotic bounds on the summation's growth.

Lemma 4.3
Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a function defined over real numbers $n \geq 1$. Then the asymptotic behavior of the function

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) , \tag{4.19}$$

defined for $n \geq 1$, can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $g(n) = O(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $g(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant c in the range $0 < c < 1$ such that $0 < af(n/b) \leq cf(n)$ for all $n \geq 1$, then $g(n) = \Theta(f(n))$.

Proof For case 1, we have $f(n) = O(n^{\log_b a - \epsilon})$, which implies that $f(n/b^j) = O((n/b^j)^{\log_b a - \epsilon})$. Substituting into equation (4.19) yields

$$\begin{aligned}
g(n) &= \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j O\left(\left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right) \\
&= O\left(\sum_{j=0}^{\lfloor \log_b n \rfloor} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right) && \text{(by Problem 3-5(c), repeatedly)} \\
&= O\left(n^{\log_b a - \epsilon} \sum_{j=0}^{\lfloor \log_b n \rfloor} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j\right) \\
&= O\left(n^{\log_b a - \epsilon} \sum_{j=0}^{\lfloor \log_b n \rfloor} (b^\epsilon)^j\right) && \text{(by equation (3.17) on page 66)} \\
&= O\left(n^{\log_b a - \epsilon} \left(\frac{b^{\epsilon(\lfloor \log_b n \rfloor + 1)} - 1}{b^\epsilon - 1}\right)\right) && \text{(by equation (A.6) on page 1142)} ,
\end{aligned}$$

the last series being geometric. Since b and ϵ are constants, the $b^\epsilon - 1$ denominator doesn't affect the asymptotic growth of $g(n)$, and neither does the -1 in

the numerator. Since $b^{\epsilon(\lfloor \log_b n \rfloor + 1)} \le (b^{\log_b n + 1})^\epsilon = b^\epsilon n^\epsilon = O(n^\epsilon)$, we obtain $g(n) = O(n^{\log_b a - \epsilon} \cdot O(n^\epsilon)) = O(n^{\log_b a})$, thereby proving case 1.

Case 2 assumes that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, from which we can conclude that $f(n/b^j) = \Theta((n/b^j)^{\log_b a} \lg^k (n/b^j))$. Substituting into equation (4.19) and repeatedly applying Problem 3-5(c) yields

$$\begin{aligned}
g(n) &= \Theta\left(\sum_{j=0}^{\lfloor \log_b n \rfloor} a^j \left(\frac{n}{b^j}\right)^{\log_b a} \lg^k \left(\frac{n}{b^j}\right)\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \frac{a^j}{b^{j \log_b a}} \lg^k \left(\frac{n}{b^j}\right)\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \lg^k \left(\frac{n}{b^j}\right)\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \left(\frac{\log_b (n/b^j)}{\log_b 2}\right)^k\right) && \text{(by equation (3.19) on page 66)} \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \left(\frac{\log_b n - j}{\log_b 2}\right)^k\right) && \text{(by equations (3.17), (3.18), and (3.20))} \\
&= \Theta\left(\frac{n^{\log_b a}}{\log_b^k 2} \sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k\right) && (b > 1 \text{ and } k \text{ are constants}) .
\end{aligned}$$

The summation within the Θ-notation can be bounded from above as follows:

$$\begin{aligned}
\sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k &\le \sum_{j=0}^{\lfloor \log_b n \rfloor} (\lfloor \log_b n \rfloor + 1 - j)^k \\
&= \sum_{j=1}^{\lfloor \log_b n \rfloor + 1} j^k && \text{(reindexing—pages 1143–1144)} \\
&= O((\lfloor \log_b n \rfloor + 1)^{k+1}) && \text{(by Exercise A.1-5 on page 1144)} \\
&= O(\log_b^{k+1} n) && \text{(by Exercise 3.3-3 on page 70)} .
\end{aligned}$$

Exercise 4.6-1 asks you to show that the summation can similarly be bounded from below by $\Omega(\log_b^{k+1} n)$. Since we have tight upper and lower bounds, the summation is $\Theta(\log_b^{k+1} n)$, from which we can conclude that $g(n) = \Theta\left(n^{\log_b a} \log_b^{k+1} n\right)$, thereby completing the proof of case 2.

For case 3, observe that $f(n)$ appears in the definition (4.19) of $g(n)$ (when $j = 0$) and that all terms of $g(n)$ are positive. Therefore, we must have $g(n) = \Omega(f(n))$, and it only remains to prove that $g(n) = O(f(n))$. Performing j iterations of the inequality $af(n/b) \le cf(n)$ yields $a^j f(n/b^j) \le c^j f(n)$. Substituting into equation (4.19), we obtain

$$\begin{aligned}
g(n) &= \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) \\
&\le \sum_{j=0}^{\lfloor \log_b n \rfloor} c^j f(n) \\
&\le f(n) \sum_{j=0}^{\infty} c^j \\
&= f(n) \left(\frac{1}{1-c} \right) \qquad \text{(by equation (A.7) on page 1142 since } |c| < 1) \\
&= O(f(n)) .
\end{aligned}$$

Thus, we can conclude that $g(n) = \Theta(f(n))$. With case 3 proved, the entire proof of the lemma is complete. ■

We can now state and prove the continuous master theorem.

Theorem 4.4 (Continuous master theorem)
Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the algorithmic recurrence $T(n)$ on the positive real numbers by

$$T(n) = aT(n/b) + f(n) .$$

Then the asymptotic behavior of $T(n)$ can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \ge 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the regularity condition $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Proof The idea is to bound the summation (4.18) from Lemma 4.2 by applying Lemma 4.3. But we must account for Lemma 4.2 using a base case for $0 < n < 1$,

whereas this theorem uses an implicit base case for $0 < n < n_0$, where $n_0 > 0$ is an arbitrary threshold constant. Since the recurrence is algorithmic, we can assume that $f(n)$ is defined for $n \geq n_0$.

For $n > 0$, let us define two auxiliary functions $T'(n) = T(n_0 n)$ and $f'(n) = f(n_0 n)$. We have

$$\begin{aligned} T'(n) &= T(n_0 n) \\ &= \begin{cases} \Theta(1) & \text{if } n_0 n < n_0 , \\ aT(n_0 n/b) + f(n_0 n) & \text{if } n_0 n \geq n_0 \end{cases} \\ &= \begin{cases} \Theta(1) & \text{if } n < 1 , \\ aT'(n/b) + f'(n) & \text{if } n \geq 1 . \end{cases} \end{aligned} \tag{4.20}$$

We have obtained a recurrence for $T'(n)$ that satisfies the conditions of Lemma 4.2, and by that lemma, the solution is

$$T'(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f'(n/b^j) . \tag{4.21}$$

To solve $T'(n)$, we first need to bound $f'(n)$. Let's examine the individual cases in the theorem.

The condition for case 1 is $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$. We have

$$\begin{aligned} f'(n) &= f(n_0 n) \\ &= O((n_0 n)^{\log_b a - \epsilon}) \\ &= O(n^{\log_b a - \epsilon}) , \end{aligned}$$

since a, b, n_0, and ϵ are all constant. The function $f'(n)$ satisfies the conditions of case 1 of Lemma 4.3, and the summation in equation (4.18) of Lemma 4.2 evaluates to $O(n^{\log_b a})$. Because a, b and n_0 are all constants, we have

$$\begin{aligned} T(n) &= T'(n/n_0) \\ &= \Theta((n/n_0)^{\log_b a}) + O((n/n_0)^{\log_b a}) \\ &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}) \qquad \text{(by Problem 3-5(b))} , \end{aligned}$$

thereby completing case 1 of the theorem.

The condition for case 2 is $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$. We have

$$\begin{aligned} f'(n) &= f(n_0 n) \\ &= \Theta((n_0 n)^{\log_b a} \lg^k (n_0 n)) \\ &= \Theta(n^{\log_b a} \lg^k n) \qquad \text{(by eliminating the constant terms)} . \end{aligned}$$

Similar to the proof of case 1, the function $f'(n)$ satisfies the conditions of case 2 of Lemma 4.3. The summation in equation (4.18) of Lemma 4.2 is therefore $\Theta(n^{\log_b a} \lg^{k+1} n)$, which implies that

$$\begin{aligned} T(n) &= T'(n/n_0) \\ &= \Theta((n/n_0)^{\log_b a}) + \Theta((n/n_0)^{\log_b a} \lg^{k+1}(n/n_0)) \\ &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg^{k+1} n) \\ &= \Theta(n^{\log_b a} \lg^{k+1} n) \qquad \text{(by Problem 3-5(c))} , \end{aligned}$$

which proves case 2 of the theorem.

Finally, the condition for case 3 is $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ additionally satisfies the regularity condition $af(n/b) \le cf(n)$ for all $n \ge n_0$ and some constants $c < 1$ and $n_0 > 1$. The first part of case 3 is like case 1:

$$\begin{aligned} f'(n) &= f(n_0 n) \\ &= \Omega((n_0 n)^{\log_b a+\epsilon}) \\ &= \Omega(n^{\log_b a+\epsilon}) . \end{aligned}$$

Using the definition of $f'(n)$ and the fact that $n_0 n \ge n_0$ for all $n \ge 1$, we have for $n \ge 1$ that

$$\begin{aligned} af'(n/b) &= af(n_0 n/b) \\ &\le cf(n_0 n) \\ &= cf'(n) . \end{aligned}$$

Thus $f'(n)$ satisfies the requirements for case 3 of Lemma 4.3, and the summation in equation (4.18) of Lemma 4.2 evaluates to $\Theta(f'(n))$, yielding

$$\begin{aligned} T(n) &= T'(n/n_0) \\ &= \Theta((n/n_0)^{\log_b a}) + \Theta(f'(n/n_0)) \\ &= \Theta(f'(n/n_0)) \\ &= \Theta(f(n)) , \end{aligned}$$

which completes the proof of case 3 of the theorem and thus the whole theorem. ■

Exercises

4.6-1

Show that $\sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k = \Omega(\log_b^{k+1} n)$.

★ ***4.6-2***

Show that case 3 of the master theorem is overstated (which is also why case 3 of Lemma 4.3 does not require that $f(n) = \Omega(n^{\log_b a+\epsilon})$) in the sense that the

regularity condition $af(n/b) \le cf(n)$ for some constant $c < 1$ implies that there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$.

★ ***4.6-3***
For $f(n) = \Theta(n^{\log_b a}/\lg n)$, prove that the summation in equation (4.19) has solution $g(n) = \Theta(n^{\log_b a} \lg\lg n)$. Conclude that a master recurrence $T(n)$ using $f(n)$ as its driving function has solution $T(n) = \Theta(n^{\log_b a} \lg\lg n)$.

★ 4.7 Akra-Bazzi recurrences

This section provides an overview of two advanced topics related to divide-and-conquer recurrences. The first deals with technicalities arising from the use of floors and ceilings, and the second discusses the Akra-Bazzi method, which involves a little calculus, for solving complicated divide-and-conquer recurrences.

In particular, we'll look at the class of algorithmic divide-and-conquer recurrences originally studied by M. Akra and L. Bazzi [13]. These ***Akra-Bazzi*** recurrences take the form

$$T(n) = f(n) + \sum_{i=1}^{k} a_i T(n/b_i) , \tag{4.22}$$

where k is a positive integer; all the constants $a_1, a_2, \ldots, a_k \in \mathbb{R}$ are strictly positive; all the constants $b_1, b_2, \ldots, b_k \in \mathbb{R}$ are strictly greater than 1; and the driving function $f(n)$ is defined on sufficiently large nonnegative reals and is itself nonnegative.

Akra-Bazzi recurrences generalize the class of recurrences addressed by the master theorem. Whereas master recurrences characterize the running times of divide-and-conquer algorithms that break a problem into equal-sized subproblems (modulo floors and ceilings), Akra-Bazzi recurrences can describe the running time of divide-and-conquer algorithms that break a problem into different-sized subproblems. The master theorem, however, allows you to ignore floors and ceilings, but the Akra-Bazzi method for solving Akra-Bazzi recurrences needs an additional requirement to deal with floors and ceilings.

But before diving into the Akra-Bazzi method itself, let's understand the limitations involved in ignoring floors and ceilings in Akra-Bazzi recurrences. As you're aware, algorithms generally deal with integer-sized inputs. The mathematics for recurrences is often easier with real numbers, however, than with integers, where we must cope with floors and ceilings to ensure that terms are well defined. The difference may not seem to be much—especially because that's often the truth with recurrences—but to be mathematically correct, we must be careful with our

assumptions. Since our end goal is to understand algorithms and not the vagaries of mathematical corner cases, we'd like to be casual yet rigorous. How can we treat floors and ceilings casually while still ensuring rigor?

From a mathematical point of view, the difficulty in dealing with floors and ceilings is that some driving functions can be really, really weird. So it's not okay in general to ignore floors and ceilings in Akra-Bazzi recurrences. Fortunately, most of the driving functions we encounter in the study of algorithms behave nicely, and floors and ceilings don't make a difference.

The polynomial-growth condition

If the driving function $f(n)$ in equation (4.22) is well behaved in the following sense, it's okay to drop floors and ceilings.

> A function $f(n)$ defined on all sufficiently large positive reals satisfies the ***polynomial-growth condition*** if there exists a constant $\hat{n} > 0$ such that the following holds: for every constant $\phi \geq 1$, there exists a constant $d > 1$ (depending on ϕ) such that $f(n)/d \leq f(\psi n) \leq df(n)$ for all $1 \leq \psi \leq \phi$ and $n \geq \hat{n}$.

This definition may be one of the hardest in this textbook to get your head around. To a first order, it says that $f(n)$ satisfies the property that $f(\Theta(n)) = \Theta(f(n))$, although the polynomial-growth condition is actually somewhat stronger (see Exercise 4.7-4). The definition also implies that $f(n)$ is asymptotically positive (see Exercise 4.7-3).

Examples of functions that satisfy the polynomial-growth condition include any function of the form $f(n) = \Theta(n^\alpha \lg^\beta n \lg\lg^\gamma n)$, where α, β, and γ are constants. Most of the polynomially bounded functions used in this book satisfy the condition. Exponentials and superexponentials do not (see Exercise 4.7-2, for example), and there also exist polynomially bounded functions that do not.

Floors and ceilings in "nice" recurrences

When the driving function in an Akra-Bazzi recurrence satisfies the polynomial-growth condition, floors and ceilings don't change the asymptotic behavior of the solution. The following theorem, which is presented without proof, formalizes this notion.

Theorem 4.5

Let $T(n)$ be a function defined on the nonnegative reals that satisfies recurrence (4.22), where $f(n)$ satisfies the polynomial-growth condition. Let $T'(n)$ be another function defined on the natural numbers also satisfying recurrence (4.22),

except that each $T(n/b_i)$ is replaced either with $T(\lceil n/b_i \rceil)$ or with $T(\lfloor n/b_i \rfloor)$. Then we have $T'(n) = \Theta(T(n))$. ■

Floors and ceilings represent a minor perturbation to the arguments in the recursion. By inequality (3.2) on page 64, they perturb an argument by at most 1. But much larger perturbations are tolerable. As long as the driving function $f(n)$ in recurrence (4.22) satisfies the polynomial-growth condition, it turns out that replacing any term $T(n/b_i)$ with $T(n/b_i + h_i(n))$, where $|h_i(n)| = O(n/\lg^{1+\epsilon} n)$ for some constant $\epsilon > 0$ and sufficiently large n, leaves the asymptotic solution unaffected. Thus, the divide step in a divide-and-conquer algorithm can be moderately coarse without affecting the solution to its running-time recurrence.

The Akra-Bazzi method

The Akra-Bazzi method, not surprisingly, was developed to solve Akra-Bazzi recurrences (4.22), which by dint of Theorem 4.5, applies in the presence of floors and ceilings or even larger perturbations, as just discussed. The method involves first determining the unique real number p such that $\sum_{i=1}^{k} a_i/b_i^p = 1$. Such a p always exists, because when $p \to -\infty$, the sum goes to ∞; it decreases as p increases; and when $p \to \infty$, it goes to 0. The Akra-Bazzi method then gives the solution to the recurrence as

$$T(n) = \Theta\left(n^p \left(1 + \int_1^n \frac{f(x)}{x^{p+1}}\, dx\right)\right) . \tag{4.23}$$

As an example, consider the recurrence

$$T(n) = T(n/5) + T(7n/10) + n . \tag{4.24}$$

We'll see the similar recurrence (9.1) on page 240 when we study an algorithm for selecting the ith smallest element from a set of n numbers. This recurrence has the form of equation (4.22), where $a_1 = a_2 = 1$, $b_1 = 5$, $b_2 = 10/7$, and $f(n) = n$. To solve it, the Akra-Bazzi method says that we should determine the unique p satisfying

$$\left(\frac{1}{5}\right)^p + \left(\frac{7}{10}\right)^p = 1 .$$

Solving for p is kind of messy—it turns out that $p = 0.83978\ldots$—but we can solve the recurrence without actually knowing the exact value for p. Observe that $(1/5)^0 + (7/10)^0 = 2$ and $(1/5)^1 + (7/10)^1 = 9/10$, and thus p lies in the range $0 < p < 1$. That turns out to be sufficient for the Akra-Bazzi method to give us the solution. We'll use the fact from calculus that if $k \neq -1$, then $\int x^k dx = x^{k+1}/(k+1)$, which we'll apply with $k = -p \neq -1$. The Akra-Bazzi

solution (4.23) gives us

$$\begin{aligned}
T(n) &= \Theta\left(n^p\left(1+\int_1^n \frac{f(x)}{x^{p+1}}\,dx\right)\right) \\
&= \Theta\left(n^p\left(1+\int_1^n x^{-p}\,dx\right)\right) \\
&= \Theta\left(n^p\left(1+\left[\frac{x^{1-p}}{1-p}\right]_1^n\right)\right) \\
&= \Theta\left(n^p\left(1+\left(\frac{n^{1-p}}{1-p}-\frac{1}{1-p}\right)\right)\right) \\
&= \Theta\left(n^p\cdot\Theta(n^{1-p})\right) && \text{(because } 1-p \text{ is a positive constant)} \\
&= \Theta(n) && \text{(by Problem 3-5(d))} .
\end{aligned}$$

Although the Akra-Bazzi method is more general than the master theorem, it requires calculus and sometimes a bit more reasoning. You also must ensure that your driving function satisfies the polynomial-growth condition if you want to ignore floors and ceilings, although that's rarely a problem. When it applies, the master method is much simpler to use, but only when subproblem sizes are more or less equal. They are both good tools for your algorithmic toolkit.

Exercises

★ ***4.7-1***

Consider an Akra-Bazzi recurrence $T(n)$ on the reals as given in recurrence (4.22), and define $T'(n)$ as

$$T'(n) = cf(n) + \sum_{i=1}^{k} a_i T'(n/b_i) ,$$

where $c > 0$ is constant. Prove that whatever the implicit initial conditions for $T(n)$ might be, there exist initial conditions for $T'(n)$ such that $T'(n) = cT(n)$ for all $n > 0$. Conclude that we can drop the asymptotics on a driving function in any Akra-Bazzi recurrence without affecting its asymptotic solution.

4.7-2

Show that $f(n) = n^2$ satisfies the polynomial-growth condition but that $f(n) = 2^n$ does not.

4.7-3

Let $f(n)$ be a function that satisfies the polynomial-growth condition. Prove that $f(n)$ is asymptotically positive, that is, there exists a constant $n_0 \geq 0$ such that $f(n) \geq 0$ for all $n \geq n_0$.

★ ***4.7-4***
Give an example of a function $f(n)$ that does not satisfy the polynomial-growth condition but for which $f(\Theta(n)) = \Theta(f(n))$.

4.7-5
Use the Akra-Bazzi method to solve the following recurrences.

a. $T(n) = T(n/2) + T(n/3) + T(n/6) + n \lg n$.

b. $T(n) = 3T(n/3) + 8T(n/4) + n^2/\lg n$.

c. $T(n) = (2/3)T(n/3) + (1/3)T(2n/3) + \lg n$.

d. $T(n) = (1/3)T(n/3) + 1/n$.

e. $T(n) = 3T(n/3) + 3T(2n/3) + n^2$.

★ ***4.7-6***
Use the Akra-Bazzi method to prove the continuous master theorem.

Problems

4-1 Recurrence examples
Give asymptotically tight upper and lower bounds for $T(n)$ in each of the following algorithmic recurrences. Justify your answers.

a. $T(n) = 2T(n/2) + n^3$.

b. $T(n) = T(8n/11) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 4T(n/2) + n^2 \lg n$.

e. $T(n) = 8T(n/3) + n^2$.

f. $T(n) = 7T(n/2) + n^2 \lg n$.

g. $T(n) = 2T(n/4) + \sqrt{n}$.

h. $T(n) = T(n-2) + n^2$.

4-2 Parameter-passing costs

Throughout this book, we assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three parameter-passing strategies:

1. Arrays are passed by pointer. Time $= \Theta(1)$.
2. Arrays are passed by copying. Time $= \Theta(N)$, where N is the size of the array.
3. Arrays are passed by copying only the subrange that might be accessed by the called procedure. Time $= \Theta(n)$ if the subarray contains n elements.

Consider the following three algorithms:

a. The recursive binary-search algorithm for finding a number in a sorted array (see Exercise 2.3-6).

b. The MERGE-SORT procedure from Section 2.3.1.

c. The MATRIX-MULTIPLY-RECURSIVE procedure from Section 4.1.

Give nine recurrences $T_{a1}(N,n), T_{a2}(N,n), \ldots, T_{c3}(N,n)$ for the worst-case running times of each of the three algorithms above when arrays and matrices are passed using each of the three parameter-passing strategies above. Solve your recurrences, giving tight asymptotic bounds.

4-3 Solving recurrences with a change of variables

Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one you have seen before. Let's solve the recurrence

$$T(n) = 2T\left(\sqrt{n}\right) + \Theta(\lg n) \tag{4.25}$$

by using the change-of-variables method.

a. Define $m = \lg n$ and $S(m) = T(2^m)$. Rewrite recurrence (4.25) in terms of m and $S(m)$.

b. Solve your recurrence for $S(m)$.

c. Use your solution for $S(m)$ to conclude that $T(n) = \Theta(\lg n \lg\lg n)$.

d. Sketch the recursion tree for recurrence (4.25), and use it to explain intuitively why the solution is $T(n) = \Theta(\lg n \lg\lg n)$.

Solve the following recurrences by changing variables:

e. $T(n) = 2T(\sqrt{n}) + \Theta(1)$.

f. $T(n) = 3T(\sqrt[3]{n}) + \Theta(n)$.

4-4 More recurrence examples

Give asymptotically tight upper and lower bounds for $T(n)$ in each of the following recurrences. Justify your answers.

a. $T(n) = 5T(n/3) + n \lg n$.

b. $T(n) = 3T(n/3) + n/\lg n$.

c. $T(n) = 8T(n/2) + n^3\sqrt{n}$.

d. $T(n) = 2T(n/2 - 2) + n/2$.

e. $T(n) = 2T(n/2) + n/\lg n$.

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

g. $T(n) = T(n-1) + 1/n$.

h. $T(n) = T(n-1) + \lg n$.

i. $T(n) = T(n-2) + 1/\lg n$.

j. $T(n) = \sqrt{n}\,T(\sqrt{n}) + n$.

4-5 Fibonacci numbers

This problem develops properties of the Fibonacci numbers, which are defined by recurrence (3.31) on page 69. We'll explore the technique of generating functions to solve the Fibonacci recurrence. Define the ***generating function*** (or ***formal power series***) $\mathcal{F}$ as

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \cdots ,\end{aligned}$$

where F_i is the ith Fibonacci number.

a. Show that $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

b. Show that

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1 - z - z^2} \\ &= \frac{z}{(1 - \phi z)(1 - \widehat{\phi} z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \phi z} - \frac{1}{1 - \widehat{\phi} z}\right) ,\end{aligned}$$

where ϕ is the golden ratio, and $\widehat{\phi}$ is its conjugate (see page 69).

c. Show that

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \widehat{\phi}^i) z^i .$$

You may use without proof the generating-function version of equation (A.7) on page 1142, $\sum_{k=0}^{\infty} x^k = 1/(1 - x)$. Because this equation involves a generating function, x is a formal variable, not a real-valued variable, so that you don't have to worry about convergence of the summation or about the requirement in equation (A.7) that $|x| < 1$, which doesn't make sense here.

d. Use part (c) to prove that $F_i = \phi^i / \sqrt{5}$ for $i > 0$, rounded to the nearest integer. (*Hint:* Observe that $\left|\widehat{\phi}\right| < 1$.)

e. Prove that $F_{i+2} \geq \phi^i$ for $i \geq 0$.

4-6 Chip testing

Professor Diogenes has n supposedly identical integrated-circuit chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the professor cannot trust the answer of a bad chip. Thus, the four possible outcomes of a test are as follows:

Chip A says	Chip B says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

a. Show that if at least $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

Now you will design an algorithm to identify which chips are good and which are bad, assuming that more than $n/2$ of the chips are good. First, you will determine how to identify one good chip.

b. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size. That is, show how to use $\lfloor n/2 \rfloor$ pairwise tests to obtain a set with at most $\lceil n/2 \rceil$ chips that still has the property that more than half of the chips are good.

c. Show how to apply the solution to part (b) recursively to identify one good chip. Give and solve the recurrence that describes the number of tests needed to identify one good chip.

You have now determined how to identify one good chip.

d. Show how to identify all the good chips with an additional $\Theta(n)$ pairwise tests.

4-7 Monge arrays

An $m \times n$ array A of real numbers is a ***Monge array*** if for all i, j, k, and l such that $1 \le i < k \le m$ and $1 \le j < l \le n$, we have

$$A[i, j] + A[k, l] \le A[i, l] + A[k, j] .$$

In other words, whenever we pick two rows and two columns of a Monge array and consider the four elements at the intersections of the rows and the columns, the sum of the upper-left and lower-right elements is less than or equal to the sum of the lower-left and upper-right elements. For example, the following array is Monge:

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

a. Prove that an array is Monge if and only if for all $i = 1, 2, \ldots, m - 1$ and $j = 1, 2, \ldots, n - 1$, we have

$$A[i, j] + A[i + 1, j + 1] \le A[i, j + 1] + A[i + 1, j] .$$

(*Hint:* For the "if" part, use induction separately on rows and columns.)

b. The following array is not Monge. Change one element in order to make it Monge. (*Hint:* Use part (a).)

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

c. Let $f(i)$ be the index of the column containing the leftmost minimum element of row i. Prove that $f(1) \le f(2) \le \cdots \le f(m)$ for any $m \times n$ Monge array.

d. Here is a description of a divide-and-conquer algorithm that computes the leftmost minimum element in each row of an $m \times n$ Monge array A:

> Construct a submatrix A' of A consisting of the even-numbered rows of A. Recursively determine the leftmost minimum for each row of A'. Then compute the leftmost minimum in the odd-numbered rows of A.

Explain how to compute the leftmost minimum in the odd-numbered rows of A (given that the leftmost minimum of the even-numbered rows is known) in $O(m + n)$ time.

e. Write the recurrence for the running time of the algorithm in part (d). Show that its solution is $O(m + n \log m)$.

Chapter notes

Divide-and-conquer as a technique for designing algorithms dates back at least to 1962 in an article by Karatsuba and Ofman [242], but it might have been used well before then. According to Heideman, Johnson, and Burrus [211], C. F. Gauss devised the first fast Fourier transform algorithm in 1805, and Gauss's formulation breaks the problem into smaller subproblems whose solutions are combined.

Strassen's algorithm [424] caused much excitement when it appeared in 1969. Before then, few imagined the possibility of an algorithm asymptotically faster than the basic MATRIX-MULTIPLY procedure. Shortly thereafter, S. Winograd reduced the number of submatrix additions from 18 to 15 while still using seven submatrix multiplications. This improvement, which Winograd apparently never published (and which is frequently miscited in the literature), may enhance the practicality of the method, but it does not affect its asymptotic performance. Probert [368] described Winograd's algorithm and showed that with seven multiplications, 15 additions is the minimum possible.

Strassen's $\Theta(n^{\lg 7}) = O(n^{2.81})$ bound for matrix multiplication held until 1987, when Coppersmith and Winograd [103] made a significant advance, improving the

bound to $O(n^{2.376})$ time with a mathematically sophisticated but wildly impractical algorithm based on tensor products. It took approximately 25 years before the asymptotic upper bound was again improved. In 2012 Vassilevska Williams [445] improved it to $O(n^{2.37287})$, and two years later Le Gall [278] achieved $O(n^{2.37286})$, both of them using mathematically fascinating but impractical algorithms. The best lower bound to date is just the obvious $\Omega(n^2)$ bound (obvious because any algorithm for matrix multiplication must fill in the n^2 elements of the product matrix).

The performance of MATRIX-MULTIPLY-RECURSIVE can be improved in practice by coarsening the leaves of the recursion. It also exhibits better cache behavior than MATRIX-MULTIPLY, although MATRIX-MULTIPLY can be improved by "tiling." Leiserson et al. [293] conducted a performance-engineering study of matrix multiplication in which a parallel and vectorized divide-and-conquer algorithm achieved the highest performance. Strassen's algorithm can be practical for large dense matrices, although large matrices tend to be sparse, and sparse methods can be much faster. When using limited-precision floating-point values, Strassen's algorithm produces larger numerical errors than the $\Theta(n^3)$ algorithms do, although Higham [215] demonstrated that Strassen's algorithm is amply accurate for some applications.

Recurrences were studied as early as 1202 by Leonardo Bonacci [66], also known as Fibonacci, for whom the Fibonacci numbers are named, although Indian mathematicians had discovered Fibonacci numbers centuries before. The French mathematician De Moivre [108] introduced the method of generating functions with which he studied Fibonacci numbers (see Problem 4-5). Knuth [259] and Liu [302] are good resources for learning the method of generating functions.

Aho, Hopcroft, and Ullman [5, 6] offered one of the first general methods for solving recurrences arising from the analysis of divide-and-conquer algorithms. The master method was adapted from Bentley, Haken, and Saxe [52]. The Akra-Bazzi method is due (unsurprisingly) to Akra and Bazzi [13]. Divide-and-conquer recurrences have been studied by many researchers, including Campbell [79], Graham, Knuth, and Patashnik [199], Kuszmaul and Leiserson [274], Leighton [287], Purdom and Brown [371], Roura [389], Verma [447], and Yap [462].

The issue of floors and ceilings in divide-and-conquer recurrences, including a theorem similar to Theorem 4.5, was studied by Leighton [287]. Leighton proposed a version of the polynomial-growth condition. Campbell [79] removed several limitations in Leighton's statement of it and showed that there were polynomially bounded functions that do not satisfy Leighton's condition. Campbell also carefully studied many other technical issues, including the well-definedness of divide-and-conquer recurrences. Kuszmaul and Leiserson [274] provided a proof of Theorem 4.5 that does not involve calculus or other higher math. Both Campbell and Leighton explored the perturbations of arguments beyond simple floors and ceilings.

5 Probabilistic Analysis and Randomized Algorithms

This chapter introduces probabilistic analysis and randomized algorithms. If you are unfamiliar with the basics of probability theory, you should read Sections C.1–C.4 of Appendix C, which review this material. We'll revisit probabilistic analysis and randomized algorithms several times throughout this book.

5.1 The hiring problem

Suppose that you need to hire a new office assistant. Your previous attempts at hiring have been unsuccessful, and you decide to use an employment agency. The employment agency sends you one candidate each day. You interview that person and then decide either to hire that person or not. You must pay the employment agency a small fee to interview an applicant. To actually hire an applicant is more costly, however, since you must fire your current office assistant and also pay a substantial hiring fee to the employment agency. You are committed to having, at all times, the best possible person for the job. Therefore, you decide that, after interviewing each applicant, if that applicant is better qualified than the current office assistant, you will fire the current office assistant and hire the new applicant. You are willing to pay the resulting price of this strategy, but you wish to estimate what that price will be.

The procedure HIRE-ASSISTANT on the facing page expresses this strategy for hiring in pseudocode. The candidates for the office assistant job are numbered 1 through n and interviewed in that order. The procedure assumes that after interviewing candidate i, you can determine whether candidate i is the best candidate you have seen so far. It starts by creating a dummy candidate, numbered 0, who is less qualified than each of the other candidates.

The cost model for this problem differs from the model described in Chapter 2. We focus not on the running time of HIRE-ASSISTANT, but instead on the fees paid for interviewing and hiring. On the surface, analyzing the cost of this algorithm

```
HIRE-ASSISTANT(n)
best = 0            // candidate 0 is a least-qualified dummy candidate
for i = 1 to n
    interview candidate i
    if candidate i is better than candidate best
        best = i
        hire candidate i
```

may seem very different from analyzing the running time of, say, merge sort. The analytical techniques used, however, are identical whether we are analyzing cost or running time. In either case, we are counting the number of times certain basic operations are executed.

Interviewing has a low cost, say c_i, whereas hiring is expensive, costing c_h. Letting m be the number of people hired, the total cost associated with this algorithm is $O(c_i n + c_h m)$. No matter how many people you hire, you always interview n candidates and thus always incur the cost $c_i n$ associated with interviewing. We therefore concentrate on analyzing $c_h m$, the hiring cost. This quantity depends on the order in which you interview candidates.

This scenario serves as a model for a common computational paradigm. Algorithms often need to find the maximum or minimum value in a sequence by examining each element of the sequence and maintaining a current "winner." The hiring problem models how often a procedure updates its notion of which element is currently winning.

Worst-case analysis

In the worst case, you actually hire every candidate that you interview. This situation occurs if the candidates come in strictly increasing order of quality, in which case you hire n times, for a total hiring cost of $O(c_h n)$.

Of course, the candidates do not always come in increasing order of quality. In fact, you have no idea about the order in which they arrive, nor do you have any control over this order. Therefore, it is natural to ask what we expect to happen in a typical or average case.

Probabilistic analysis

Probabilistic analysis is the use of probability in the analysis of problems. Most commonly, we use probabilistic analysis to analyze the running time of an algorithm. Sometimes we use it to analyze other quantities, such as the hiring cost in

procedure HIRE-ASSISTANT. In order to perform a probabilistic analysis, we must use knowledge of, or make assumptions about, the distribution of the inputs. Then we analyze our algorithm, computing an average-case running time, where we take the average, or expected value, over the distribution of the possible inputs. When reporting such a running time, we refer to it as the ***average-case running time***.

You must be careful in deciding on the distribution of inputs. For some problems, you may reasonably assume something about the set of all possible inputs, and then you can use probabilistic analysis as a technique for designing an efficient algorithm and as a means for gaining insight into a problem. For other problems, you cannot characterize a reasonable input distribution, and in these cases you cannot use probabilistic analysis.

For the hiring problem, we can assume that the applicants come in a random order. What does that mean for this problem? We assume that you can compare any two candidates and decide which one is better qualified, which is to say that there is a total order on the candidates. (See Section B.2 for the definition of a total order.) Thus, you can rank each candidate with a unique number from 1 through n, using $rank(i)$ to denote the rank of applicant i, and adopt the convention that a higher rank corresponds to a better qualified applicant. The ordered list $\langle rank(1), rank(2), \ldots, rank(n) \rangle$ is a permutation of the list $\langle 1, 2, \ldots, n \rangle$. Saying that the applicants come in a random order is equivalent to saying that this list of ranks is equally likely to be any one of the $n!$ permutations of the numbers 1 through n. Alternatively, we say that the ranks form a ***uniform random permutation***, that is, each of the possible $n!$ permutations appears with equal probability.

Section 5.2 contains a probabilistic analysis of the hiring problem.

Randomized algorithms

In order to use probabilistic analysis, you need to know something about the distribution of the inputs. In many cases, you know little about the input distribution. Even if you do know something about the distribution, you might not be able to model this knowledge computationally. Yet, probability and randomness often serve as tools for algorithm design and analysis, by making part of the algorithm behave randomly.

In the hiring problem, it may seem as if the candidates are being presented to you in a random order, but you have no way of knowing whether they really are. Thus, in order to develop a randomized algorithm for the hiring problem, you need greater control over the order in which you'll interview the candidates. We will, therefore, change the model slightly. The employment agency sends you a list of the n candidates in advance. On each day, you choose, randomly, which candidate to interview. Although you know nothing about the candidates (besides their names), we have made a significant change. Instead of accepting the order given

to you by the employment agency and hoping that it's random, you have instead gained control of the process and enforced a random order.

More generally, we call an algorithm ***randomized*** if its behavior is determined not only by its input but also by values produced by a ***random-number generator***. We assume that we have at our disposal a random-number generator RANDOM. A call to RANDOM(a, b) returns an integer between a and b, inclusive, with each such integer being equally likely. For example, RANDOM$(0, 1)$ produces 0 with probability $1/2$, and it produces 1 with probability $1/2$. A call to RANDOM$(3, 7)$ returns any one of 3, 4, 5, 6, or 7, each with probability $1/5$. Each integer returned by RANDOM is independent of the integers returned on previous calls. You may imagine RANDOM as rolling a $(b - a + 1)$-sided die to obtain its output. (In practice, most programming environments offer a ***pseudorandom-number generator***: a deterministic algorithm returning numbers that "look" statistically random.)

When analyzing the running time of a randomized algorithm, we take the expectation of the running time over the distribution of values returned by the random number generator. We distinguish these algorithms from those in which the input is random by referring to the running time of a randomized algorithm as an ***expected running time***. In general, we discuss the average-case running time when the probability distribution is over the inputs to the algorithm, and we discuss the expected running time when the algorithm itself makes random choices.

Exercises

5.1-1

Show that the assumption that you are always able to determine which candidate is best, in line 4 of procedure HIRE-ASSISTANT, implies that you know a total order on the ranks of the candidates.

★ ***5.1-2***

Describe an implementation of the procedure RANDOM(a, b) that makes calls only to RANDOM$(0, 1)$. What is the expected running time of your procedure, as a function of a and b?

★ ***5.1-3***

You wish to implement a program that outputs 0 with probability $1/2$ and 1 with probability $1/2$. At your disposal is a procedure BIASED-RANDOM that outputs either 0 or 1, but it outputs 1 with some probability p and 0 with probability $1 - p$, where $0 < p < 1$. You do not know what p is. Give an algorithm that uses BIASED-RANDOM as a subroutine, and returns an unbiased answer, returning 0 with probability $1/2$ and 1 with probability $1/2$. What is the expected running time of your algorithm as a function of p?

5.2 Indicator random variables

In order to analyze many algorithms, including the hiring problem, we use indicator random variables. Indicator random variables provide a convenient method for converting between probabilities and expectations. Given a sample space S and an event A, the ***indicator random variable*** $\mathrm{I}\{A\}$ associated with event A is defined as

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs ,} \\ 0 & \text{if } A \text{ does not occur .} \end{cases} \tag{5.1}$$

As a simple example, let us determine the expected number of heads obtained when flipping a fair coin. The sample space for a single coin flip is $S = \{H, T\}$, with $\Pr\{H\} = \Pr\{T\} = 1/2$. We can then define an indicator random variable X_H, associated with the coin coming up heads, which is the event H. This variable counts the number of heads obtained in this flip, and it is 1 if the coin comes up heads and 0 otherwise. We write

$$\begin{aligned} X_H &= \mathrm{I}\{H\} \\ &= \begin{cases} 1 & \text{if } H \text{ occurs ,} \\ 0 & \text{if } T \text{ occurs .} \end{cases} \end{aligned}$$

The expected number of heads obtained in one flip of the coin is simply the expected value of our indicator variable X_H:

$$\begin{aligned} \mathrm{E}[X_H] &= \mathrm{E}[\mathrm{I}\{H\}] \\ &= 1 \cdot \Pr\{H\} + 0 \cdot \Pr\{T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 \,. \end{aligned}$$

Thus the expected number of heads obtained by one flip of a fair coin is $1/2$. As the following lemma shows, the expected value of an indicator random variable associated with an event A is equal to the probability that A occurs.

Lemma 5.1
Given a sample space S and an event A in the sample space S, let $X_A = \mathrm{I}\{A\}$. Then $\mathrm{E}[X_A] = \Pr\{A\}$.

Proof By the definition of an indicator random variable from equation (5.1) and the definition of expected value, we have

$$\begin{aligned} \mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\overline{A}\} \\ &= \Pr\{A\} \,, \end{aligned}$$

where $\overline{A}$ denotes $S - A$, the complement of A. ■

Although indicator random variables may seem cumbersome for an application such as counting the expected number of heads on a flip of a single coin, they are useful for analyzing situations that perform repeated random trials. In Appendix C, for example, indicator random variables provide a simple way to determine the expected number of heads in n coin flips. One option is to consider separately the probability of obtaining 0 heads, 1 head, 2 heads, etc. to arrive at the result of equation (C.41) on page 1199. Alternatively, we can employ the simpler method proposed in equation (C.42), which uses indicator random variables implicitly. Making this argument more explicit, let X_i be the indicator random variable associated with the event in which the ith flip comes up heads: $X_i = \mathrm{I}\{\text{the } i\text{th flip results in the event } H\}$. Let X be the random variable denoting the total number of heads in the n coin flips, so that

$$X = \sum_{i=1}^{n} X_i \ .$$

In order to compute the expected number of heads, take the expectation of both sides of the above equation to obtain

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \ . \tag{5.2}$$

By Lemma 5.1, the expectation of each of the random variables is $\mathrm{E}[X_i] = 1/2$ for $i = 1, 2, \ldots, n$. Then we can compute the sum of the expectations: $\sum_{i=1}^{n} \mathrm{E}[X_i] = n/2$. But equation (5.2) calls for the expectation of the sum, not the sum of the expectations. How can we resolve this conundrum? Linearity of expectation, equation (C.24) on page 1192, to the rescue: *the expectation of the sum always equals the sum of the expectations*. Linearity of expectation applies even when there is dependence among the random variables. Combining indicator random variables with linearity of expectation gives us a powerful technique to compute expected values when multiple events occur. We now can compute the expected number of heads:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} 1/2 \\
&= n/2 \ .
\end{aligned}$$

Thus, compared with the method used in equation (C.41), indicator random variables greatly simplify the calculation. We use indicator random variables throughout this book.

Analysis of the hiring problem using indicator random variables

Returning to the hiring problem, we now wish to compute the expected number of times that you hire a new office assistant. In order to use a probabilistic analysis, let's assume that the candidates arrive in a random order, as discussed in Section 5.1. (We'll see in Section 5.3 how to remove this assumption.) Let X be the random variable whose value equals the number of times you hire a new office assistant. We could then apply the definition of expected value from equation (C.23) on page 1192 to obtain

$$\mathrm{E}[X] = \sum_{x=1}^{n} x \Pr\{X = x\} ,$$

but this calculation would be cumbersome. Instead, let's simplify the calculation by using indicator random variables.

To use indicator random variables, instead of computing $\mathrm{E}[X]$ by defining just one variable denoting the number of times you hire a new office assistant, think of the process of hiring as repeated random trials and define n variables indicating whether each particular candidate is hired. In particular, let X_i be the indicator random variable associated with the event in which the ith candidate is hired. Thus,

$$\begin{aligned} X_i &= \mathrm{I}\{\text{candidate } i \text{ is hired}\} \\ &= \begin{cases} 1 & \text{if candidate } i \text{ is hired} , \\ 0 & \text{if candidate } i \text{ is not hired} , \end{cases} \end{aligned}$$

and

$$X = X_1 + X_2 + \cdots + X_n . \tag{5.3}$$

Lemma 5.1 gives

$$\mathrm{E}[X_i] = \Pr\{\text{candidate } i \text{ is hired}\} ,$$

and we must therefore compute the probability that lines 5–6 of HIRE-ASSISTANT are executed.

Candidate i is hired, in line 6, exactly when candidate i is better than each of candidates 1 through $i - 1$. Because we have assumed that the candidates arrive in a random order, the first i candidates have appeared in a random order. Any one of these first i candidates is equally likely to be the best qualified so far. Candidate i has a probability of $1/i$ of being better qualified than candidates 1 through $i - 1$ and thus a probability of $1/i$ of being hired. By Lemma 5.1, we conclude that

$$\mathrm{E}[X_i] = 1/i \ . \tag{5.4}$$

Now we can compute $\mathrm{E}[X]$:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] && \text{(by equation (5.3))} && (5.5)\\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] && \text{(by equation (C.24), linearity of expectation)}\\
&= \sum_{i=1}^{n} \frac{1}{i} && \text{(by equation (5.4))}\\
&= \ln n + O(1) && \text{(by equation (A.9), the harmonic series)} \ . && (5.6)
\end{aligned}$$

Even though you interview n people, you actually hire only approximately $\ln n$ of them, on average. We summarize this result in the following lemma.

Lemma 5.2
Assuming that the candidates are presented in a random order, algorithm HIRE-ASSISTANT has an average-case total hiring cost of $O(c_h \ln n)$.

Proof The bound follows immediately from our definition of the hiring cost and equation (5.6), which shows that the expected number of hires is approximately $\ln n$. ■

The average-case hiring cost is a significant improvement over the worst-case hiring cost of $O(c_h n)$.

Exercises

5.2-1
In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you hire exactly one time? What is the probability that you hire exactly n times?

5.2-2
In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you hire exactly twice?

5.2-3
Use indicator random variables to compute the expected value of the sum of n dice.

5.2-4

This exercise asks you to (partly) verify that linearity of expectation holds even if the random variables are not independent. Consider two 6-sided dice that are rolled independently. What is the expected value of the sum? Now consider the case where the first die is rolled normally and then the second die is set equal to the value shown on the first die. What is the expected value of the sum? Now consider the case where the first die is rolled normally and the second die is set equal to 7 minus the value of the first die. What is the expected value of the sum?

5.2-5

Use indicator random variables to solve the following problem, which is known as the ***hat-check problem***. Each of n customers gives a hat to a hat-check person at a restaurant. The hat-check person gives the hats back to the customers in a random order. What is the expected number of customers who get back their own hat?

5.2-6

Let $A[1:n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A. (See Problem 2-4 on page 47 for more on inversions.) Suppose that the elements of A form a uniform random permutation of $\langle 1, 2, \ldots, n \rangle$. Use indicator random variables to compute the expected number of inversions.

5.3 Randomized algorithms

In the previous section, we showed how knowing a distribution on the inputs can help us to analyze the average-case behavior of an algorithm. What if you do not know the distribution? Then you cannot perform an average-case analysis. As mentioned in Section 5.1, however, you might be able to use a randomized algorithm.

For a problem such as the hiring problem, in which it is helpful to assume that all permutations of the input are equally likely, a probabilistic analysis can guide us when developing a randomized algorithm. Instead of *assuming* a distribution of inputs, we *impose* a distribution. In particular, before running the algorithm, let's randomly permute the candidates in order to enforce the property that every permutation is equally likely. Although we have modified the algorithm, we still expect to hire a new office assistant approximately $\ln n$ times. But now we expect this to be the case for *any* input, rather than for inputs drawn from a particular distribution.

Let us further explore the distinction between probabilistic analysis and randomized algorithms. In Section 5.2, we claimed that, assuming that the candidates

arrive in a random order, the expected number of times you hire a new office assistant is about $\ln n$. This algorithm is deterministic: for any particular input, the number of times a new office assistant is hired is always the same. Furthermore, the number of times you hire a new office assistant differs for different inputs, and it depends on the ranks of the various candidates. Since this number depends only on the ranks of the candidates, to represent a particular input, we can just list, in order, the ranks $\langle rank(1), rank(2), \ldots, rank(n)\rangle$ of the candidates. Given the rank list $A_1 = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle$, a new office assistant is always hired 10 times, since each successive candidate is better than the previous one, and lines 5–6 of HIRE-ASSISTANT are executed in each iteration. Given the list of ranks $A_2 = \langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1\rangle$, a new office assistant is hired only once, in the first iteration. Given a list of ranks $A_3 = \langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6\rangle$, a new office assistant is hired three times, upon interviewing the candidates with ranks 5, 8, and 10. Recalling that the cost of our algorithm depends on how many times you hire a new office assistant, we see that there are expensive inputs such as A_1, inexpensive inputs such as A_2, and moderately expensive inputs such as A_3.

Consider, on the other hand, the randomized algorithm that first permutes the list of candidates and then determines the best candidate. In this case, we randomize in the algorithm, not in the input distribution. Given a particular input, say A_3 above, we cannot say how many times the maximum is updated, because this quantity differs with each run of the algorithm. The first time you run the algorithm on A_3, it might produce the permutation A_1 and perform 10 updates. But the second time you run the algorithm, it might produce the permutation A_2 and perform only one update. The third time you run the algorithm, it might perform some other number of updates. Each time you run the algorithm, its execution depends on the random choices made and is likely to differ from the previous execution of the algorithm. For this algorithm and many other randomized algorithms, *no particular input elicits its worst-case behavior*. Even your worst enemy cannot produce a bad input array, since the random permutation makes the input order irrelevant. The randomized algorithm performs badly only if the random-number generator produces an "unlucky" permutation.

For the hiring problem, the only change needed in the code is to randomly permute the array, as done in the RANDOMIZED-HIRE-ASSISTANT procedure. This simple change creates a randomized algorithm whose performance matches that obtained by assuming that the candidates were presented in a random order.

```
RANDOMIZED-HIRE-ASSISTANT(n)
1  randomly permute the list of candidates
2  HIRE-ASSISTANT(n)
```

Lemma 5.3
The expected hiring cost of the procedure RANDOMIZED-HIRE-ASSISTANT is $O(c_h \ln n)$.

Proof Permuting the input array achieves a situation identical to that of the probabilistic analysis of HIRE-ASSISTANT in Section 5.2. ■

By carefully comparing Lemmas 5.2 and 5.3, you can see the difference between probabilistic analysis and randomized algorithms. Lemma 5.2 makes an assumption about the input. Lemma 5.3 makes no such assumption, although randomizing the input takes some additional time. To remain consistent with our terminology, we couched Lemma 5.2 in terms of the average-case hiring cost and Lemma 5.3 in terms of the expected hiring cost. In the remainder of this section, we discuss some issues involved in randomly permuting inputs.

Randomly permuting arrays

Many randomized algorithms randomize the input by permuting a given input array. We'll see elsewhere in this book other ways to randomize an algorithm, but now, let's see how we can randomly permute an array of n elements. The goal is to produce a ***uniform random permutation***, that is, a permutation that is as likely as any other permutation. Since there are $n!$ possible permutations, we want the probability that any particular permutation is produced to be $1/n!$.

You might think that to prove that a permutation is a uniform random permutation, it suffices to show that, for each element $A[i]$, the probability that the element winds up in position j is $1/n$. Exercise 5.3-4 shows that this weaker condition is, in fact, insufficient.

Our method to generate a random permutation permutes the array ***in place***: at most a constant number of elements of the input array are ever stored outside the array. The procedure RANDOMLY-PERMUTE permutes an array $A[1:n]$ in place in $\Theta(n)$ time. In its ith iteration, it chooses the element $A[i]$ randomly from among elements $A[i]$ through $A[n]$. After the ith iteration, $A[i]$ is never altered.

```
RANDOMLY-PERMUTE(A, n)
for i = 1 to n
    swap A[i] with A[RANDOM(i, n)]
```

We use a loop invariant to show that procedure RANDOMLY-PERMUTE produces a uniform random permutation. A ***k-permutation*** on a set of n elements is a se-

quence containing k of the n elements, with no repetitions. (See page 1180 in Appendix C.) There are $n!/(n-k)!$ such possible k-permutations.

Lemma 5.4
Procedure RANDOMLY-PERMUTE computes a uniform random permutation.

Proof We use the following loop invariant:

> Just prior to the ith iteration of the **for** loop of lines 1–2, for each possible $(i-1)$-permutation of the n elements, the subarray $A[1:i-1]$ contains this $(i-1)$-permutation with probability $(n-i+1)!/n!$.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, that the loop terminates, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Consider the situation just before the first loop iteration, so that $i = 1$. The loop invariant says that for each possible 0-permutation, the subarray $A[1:0]$ contains this 0-permutation with probability $(n-i+1)!/n! = n!/n! = 1$. The subarray $A[1:0]$ is an empty subarray, and a 0-permutation has no elements. Thus, $A[1:0]$ contains any 0-permutation with probability 1, and the loop invariant holds prior to the first iteration.

Maintenance: By the loop invariant, we assume that just before the ith iteration, each possible $(i-1)$-permutation appears in the subarray $A[1:i-1]$ with probability $(n-i+1)!/n!$. We shall show that after the ith iteration, each possible i-permutation appears in the subarray $A[1:i]$ with probability $(n-i)!/n!$. Incrementing i for the next iteration then maintains the loop invariant.

Let us examine the ith iteration. Consider a particular i-permutation, and denote the elements in it by $\langle x_1, x_2, \ldots, x_i \rangle$. This permutation consists of an $(i-1)$-permutation $\langle x_1, \ldots, x_{i-1} \rangle$ followed by the value x_i that the algorithm places in $A[i]$. Let E_1 denote the event in which the first $i-1$ iterations have created the particular $(i-1)$-permutation $\langle x_1, \ldots, x_{i-1} \rangle$ in $A[1:i-1]$. By the loop invariant, $\Pr\{E_1\} = (n-i+1)!/n!$. Let E_2 be the event that the ith iteration puts x_i in position $A[i]$. The i-permutation $\langle x_1, \ldots, x_i \rangle$ appears in $A[1:i]$ precisely when both E_1 and E_2 occur, and so we wish to compute $\Pr\{E_2 \cap E_1\}$. Using equation (C.16) on page 1187, we have

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\} \ .$$

The probability $\Pr\{E_2 \mid E_1\}$ equals $1/(n-i+1)$ because in line 2 the algorithm chooses x_i randomly from the $n-i+1$ values in positions $A[i:n]$. Thus, we have

$$\begin{aligned}\Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\}\Pr\{E_1\} \\ &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\ &= \frac{(n-i)!}{n!} .\end{aligned}$$

Termination: The loop terminates, since it is a **for** loop iterating n times. At termination, $i = n + 1$, and we have that the subarray $A[1:n]$ is a given n-permutation with probability $(n-(n+1)+1)!/n! = 0!/n! = 1/n!$.

Thus, RANDOMLY-PERMUTE produces a uniform random permutation. ■

A randomized algorithm is often the simplest and most efficient way to solve a problem.

Exercises

5.3-1

Professor Marceau objects to the loop invariant used in the proof of Lemma 5.4. He questions whether it holds prior to the first iteration. He reasons that we could just as easily declare that an empty subarray contains no 0-permutations. Therefore, the probability that an empty subarray contains a 0-permutation should be 0, thus invalidating the loop invariant prior to the first iteration. Rewrite the procedure RANDOMLY-PERMUTE so that its associated loop invariant applies to a nonempty subarray prior to the first iteration, and modify the proof of Lemma 5.4 for your procedure.

5.3-2

Professor Kelp decides to write a procedure that produces at random any permutation except the ***identity permutation***, in which every element ends up where it started. He proposes the procedure PERMUTE-WITHOUT-IDENTITY. Does this procedure do what Professor Kelp intends?

```
PERMUTE-WITHOUT-IDENTITY(A, n)
for i = 1 to n - 1
    swap A[i] with A[RANDOM(i + 1, n)]
```

5.3-3

Consider the PERMUTE-WITH-ALL procedure on the facing page, which instead of swapping element $A[i]$ with a random element from the subarray $A[i:n]$, swaps it with a random element from anywhere in the array. Does PERMUTE-WITH-ALL produce a uniform random permutation? Why or why not?

```
PERMUTE-WITH-ALL(A, n)
for i = 1 to n
    swap A[i] with A[RANDOM(1, n)]
```

5.3-4

Professor Knievel suggests the procedure PERMUTE-BY-CYCLE to generate a uniform random permutation. Show that each element $A[i]$ has a $1/n$ probability of winding up in any particular position in B. Then show that Professor Knievel is mistaken by showing that the resulting permutation is not uniformly random.

```
PERMUTE-BY-CYCLE(A, n)
let B[1 : n] be a new array
offset = RANDOM(1, n)
for i = 1 to n
    dest = i + offset
    if dest > n
        dest = dest − n
    B[dest] = A[i]
return B
```

5.3-5

Professor Gallup wants to create a ***random sample*** of the set $\{1, 2, 3, \ldots, n\}$, that is, an m-element subset S, where $0 \le m \le n$, such that each m-subset is equally likely to be created. One way is to set $A[i] = i$, for $i = 1, 2, 3, \ldots, n$, call RANDOMLY-PERMUTE(A), and then take just the first m array elements. This method makes n calls to the RANDOM procedure. In Professor Gallup's application, n is much larger than m, and so the professor wants to create a random sample with fewer calls to RANDOM.

```
RANDOM-SAMPLE(m, n)
S = ∅
for k = n − m + 1 to n          // iterates m times
    i = RANDOM(1, k)
    if i ∈ S
        S = S ∪ {k}
    else S = S ∪ {i}
return S
```

Show that the procedure RANDOM-SAMPLE on the previous page returns a random m-subset S of $\{1, 2, 3, \ldots, n\}$, in which each m-subset is equally likely, while making only m calls to RANDOM.

★ 5.4 Probabilistic analysis and further uses of indicator random variables

This advanced section further illustrates probabilistic analysis by way of four examples. The first determines the probability that in a room of k people, two of them share the same birthday. The second example examines what happens when randomly tossing balls into bins. The third investigates "streaks" of consecutive heads when flipping coins. The final example analyzes a variant of the hiring problem in which you have to make decisions without actually interviewing all the candidates.

5.4.1 The birthday paradox

Our first example is the ***birthday paradox***. How many people must there be in a room before there is a 50% chance that two of them were born on the same day of the year? The answer is surprisingly few. The paradox is that it is in fact far fewer than the number of days in a year, or even half the number of days in a year, as we shall see.

To answer this question, we index the people in the room with the integers $1, 2, \ldots, k$, where k is the number of people in the room. We ignore the issue of leap years and assume that all years have $n = 365$ days. For $i = 1, 2, \ldots, k$, let b_i be the day of the year on which person i's birthday falls, where $1 \le b_i \le n$. We also assume that birthdays are uniformly distributed across the n days of the year, so that $\Pr\{b_i = r\} = 1/n$ for $i = 1, 2, \ldots, k$ and $r = 1, 2, \ldots, n$.

The probability that two given people, say i and j, have matching birthdays depends on whether the random selection of birthdays is independent. We assume from now on that birthdays are independent, so that the probability that i's birthday and j's birthday both fall on day r is

$$\begin{aligned}\Pr\{b_i = r \text{ and } b_j = r\} &= \Pr\{b_i = r\}\Pr\{b_j = r\} \\ &= \frac{1}{n^2}\,.\end{aligned}$$

Thus, the probability that they both fall on the same day is

$$\Pr\{b_i = b_j\} = \sum_{r=1}^{n} \Pr\{b_i = r \text{ and } b_j = r\}$$

$$= \sum_{r=1}^{n} \frac{1}{n^2}$$
$$= \frac{1}{n} . \tag{5.7}$$

More intuitively, once b_i is chosen, the probability that b_j is chosen to be the same day is $1/n$. As long as the birthdays are independent, the probability that i and j have the same birthday is the same as the probability that the birthday of one of them falls on a given day.

We can analyze the probability of at least 2 out of k people having matching birthdays by looking at the complementary event. The probability that at least two of the birthdays match is 1 minus the probability that all the birthdays are different. The event B_k that k people have distinct birthdays is

$$B_k = \bigcap_{i=1}^{k} A_i ,$$

where A_i is the event that person i's birthday is different from person j's for all $j < i$. Since we can write $B_k = A_k \cap B_{k-1}$, we obtain from equation (C.18) on page 1189 the recurrence

$$\Pr\{B_k\} = \Pr\{B_{k-1}\}\Pr\{A_k \mid B_{k-1}\} , \tag{5.8}$$

where we take $\Pr\{B_1\} = \Pr\{A_1\} = 1$ as an initial condition. In other words, the probability that $b_1, b_2, \ldots, b_k$ are distinct birthdays equals the probability that $b_1, b_2, \ldots, b_{k-1}$ are distinct birthdays multiplied by the probability that $b_k \neq b_i$ for $i = 1, 2, \ldots, k-1$, given that $b_1, b_2, \ldots, b_{k-1}$ are distinct.

If $b_1, b_2, \ldots, b_{k-1}$ are distinct, the conditional probability that $b_k \neq b_i$ for $i = 1, 2, \ldots, k-1$ is $\Pr\{A_k \mid B_{k-1}\} = (n-k+1)/n$, since out of the n days, $n-(k-1)$ days are not taken. We iteratively apply the recurrence (5.8) to obtain

$$\begin{aligned}
\Pr\{B_k\} &= \Pr\{B_{k-1}\}\Pr\{A_k \mid B_{k-1}\} \\
&= \Pr\{B_{k-2}\}\Pr\{A_{k-1} \mid B_{k-2}\}\Pr\{A_k \mid B_{k-1}\} \\
&\;\;\vdots \\
&= \Pr\{B_1\}\Pr\{A_2 \mid B_1\}\Pr\{A_3 \mid B_2\}\cdots\Pr\{A_k \mid B_{k-1}\} \\
&= 1 \cdot \left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\cdots\left(\frac{n-k+1}{n}\right) \\
&= 1 \cdot \left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{k-1}{n}\right) .
\end{aligned}$$

Inequality (3.14) on page 66, $1 + x \leq e^x$, gives us

$$\begin{aligned}
\Pr\{B_k\} &\le e^{-1/n} e^{-2/n} \cdots e^{-(k-1)/n} \\
&= e^{-\sum_{i=1}^{k-1} i/n} \\
&= e^{-k(k-1)/2n} \\
&\le \frac{1}{2}
\end{aligned}$$

when $-k(k-1)/2n \le \ln(1/2)$. The probability that all k birthdays are distinct is at most $1/2$ when $k(k-1) \ge 2n \ln 2$ or, solving the quadratic equation, when $k \ge (1 + \sqrt{1 + (8 \ln 2)n})/2$. For $n = 365$, we must have $k \ge 23$. Thus, if at least 23 people are in a room, the probability is at least $1/2$ that at least two people have the same birthday. Since a year on Mars is 669 Martian days long, it takes 31 Martians to get the same effect.

An analysis using indicator random variables

Indicator random variables afford a simpler but approximate analysis of the birthday paradox. For each pair (i, j) of the k people in the room, define the indicator random variable X_{ij}, for $1 \le i < j \le k$, by

$$\begin{aligned}
X_{ij} &= \mathrm{I}\{\text{person } i \text{ and person } j \text{ have the same birthday}\} \\
&= \begin{cases} 1 & \text{if person } i \text{ and person } j \text{ have the same birthday} , \\ 0 & \text{otherwise} . \end{cases}
\end{aligned}$$

By equation (5.7), the probability that two people have matching birthdays is $1/n$, and thus by Lemma 5.1 on page 130, we have

$$\begin{aligned}
\mathrm{E}[X_{ij}] &= \Pr\{\text{person } i \text{ and person } j \text{ have the same birthday}\} \\
&= 1/n .
\end{aligned}$$

Letting X be the random variable that counts the number of pairs of individuals having the same birthday, we have

$$X = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} X_{ij} .$$

Taking expectations of both sides and applying linearity of expectation, we obtain

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} X_{ij}\right] \\
&= \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \mathrm{E}[X_{ij}]
\end{aligned}$$

$$= \binom{k}{2}\frac{1}{n}$$
$$= \frac{k(k-1)}{2n} .$$

When $k(k-1) \geq 2n$, therefore, the expected number of pairs of people with the same birthday is at least 1. Thus, if we have at least $\sqrt{2n}+1$ individuals in a room, we can expect at least two to have the same birthday. For $n = 365$, if $k = 28$, the expected number of pairs with the same birthday is $(28 \cdot 27)/(2 \cdot 365) \approx 1.0356$. Thus, with at least 28 people, we expect to find at least one matching pair of birthdays. On Mars, with 669 days per year, we need at least 38 Martians.

The first analysis, which used only probabilities, determined the number of people required for the probability to exceed $1/2$ that a matching pair of birthdays exists, and the second analysis, which used indicator random variables, determined the number such that the expected number of matching birthdays is 1. Although the exact numbers of people differ for the two situations, they are the same asymptotically: $\Theta(\sqrt{n})$.

5.4.2 Balls and bins

Consider a process in which you randomly toss identical balls into b bins, numbered $1, 2, \ldots, b$. The tosses are independent, and on each toss the ball is equally likely to end up in any bin. The probability that a tossed ball lands in any given bin is $1/b$. If we view the ball-tossing process as a sequence of Bernoulli trials (see Appendix C.4), where success means that the ball falls in the given bin, then each trial has a probability $1/b$ of success. This model is particularly useful for analyzing hashing (see Chapter 11), and we can answer a variety of interesting questions about the ball-tossing process. (Problem C-2 asks additional questions about balls and bins.)

- *How many balls fall in a given bin?* The number of balls that fall in a given bin follows the binomial distribution $b(k; n, 1/b)$. If you toss n balls, equation (C.41) on page 1199 tells us that the expected number of balls that fall in the given bin is n/b.
- *How many balls must you toss, on the average, until a given bin contains a ball?* The number of tosses until the given bin receives a ball follows the geometric distribution with probability $1/b$ and, by equation (C.36) on page 1197, the expected number of tosses until success is $1/(1/b) = b$.
- *How many balls must you toss until every bin contains at least one ball?* Let us call a toss in which a ball falls into an empty bin a "hit." We want to know the expected number n of tosses required to get b hits.

Using the hits, we can partition the n tosses into stages. The ith stage consists of the tosses after the $(i-1)$st hit up to and including the ith hit. The first stage consists of the first toss, since you are guaranteed to have a hit when all bins are empty. For each toss during the ith stage, $i-1$ bins contain balls and $b-i+1$ bins are empty. Thus, for each toss in the ith stage, the probability of obtaining a hit is $(b-i+1)/b$.

Let n_i denote the number of tosses in the ith stage. The number of tosses required to get b hits is $n = \sum_{i=1}^{b} n_i$. Each random variable n_i has a geometric distribution with probability of success $(b-i+1)/b$ and thus, by equation (C.36), we have

$$\mathrm{E}[n_i] = \frac{b}{b-i+1} .$$

By linearity of expectation, we have

$$\begin{aligned}
\mathrm{E}[n] &= \mathrm{E}\left[\sum_{i=1}^{b} n_i\right] \\
&= \sum_{i=1}^{b} \mathrm{E}[n_i] \\
&= \sum_{i=1}^{b} \frac{b}{b-i+1} \\
&= b \sum_{i=1}^{b} \frac{1}{i} && \text{(by equation (A.14) on page 1144)} \\
&= b(\ln b + O(1)) && \text{(by equation (A.9) on page 1142)} .
\end{aligned}$$

It therefore takes approximately $b \ln b$ tosses before we can expect that every bin has a ball. This problem is also known as the ***coupon collector's problem***, which says that if you are trying to collect each of b different coupons, then you should expect to acquire approximately $b \ln b$ randomly obtained coupons in order to succeed.

5.4.3 Streaks

Suppose that you flip a fair coin n times. What is the longest streak of consecutive heads that you expect to see? We'll prove upper and lower bounds separately to show that the answer is $\Theta(\lg n)$.

We first prove that the expected length of the longest streak of heads is $O(\lg n)$. The probability that each coin flip is a head is $1/2$. Let A_{ik} be the event that a streak of heads of length at least k begins with the ith coin flip or, more precisely, the event that the k consecutive coin flips $i, i+1, \ldots, i+k-1$ yield only heads, where $1 \le k \le n$ and $1 \le i \le n-k+1$. Since coin flips are mutually independent, for any given event A_{ik}, the probability that all k flips are heads is

$$\Pr\{A_{ik}\} = \frac{1}{2^k} . \tag{5.9}$$

For $k = 2\lceil \lg n \rceil$,

$$\begin{aligned}
\Pr\{A_{i,2\lceil \lg n \rceil}\} &= \frac{1}{2^{2\lceil \lg n \rceil}} \\
&\le \frac{1}{2^{2 \lg n}} \\
&= \frac{1}{n^2} ,
\end{aligned}$$

and thus the probability that a streak of heads of length at least $2\lceil \lg n \rceil$ begins in position i is quite small. There are at most $n - 2\lceil \lg n \rceil + 1$ positions where such a streak can begin. The probability that a streak of heads of length at least $2\lceil \lg n \rceil$ begins anywhere is therefore

$$\begin{aligned}
&\Pr\left\{ \bigcup_{i=1}^{n-2\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil} \right\} \\
&\quad \le \sum_{i=1}^{n-2\lceil \lg n \rceil+1} \Pr\{A_{i,2\lceil \lg n \rceil}\} \quad \text{(by Boole's inequality (C.21) on page 1190)} \\
&\quad \le \sum_{i=1}^{n-2\lceil \lg n \rceil+1} \frac{1}{n^2} \\
&\quad < \sum_{i=1}^{n} \frac{1}{n^2} \\
&\quad = \frac{1}{n} .
\end{aligned} \tag{5.10}$$

We can use inequality (5.10) to bound the length of the longest streak. For $j = 0, 1, 2, \ldots, n$, let L_j be the event that the longest streak of heads has length exactly j, and let L be the length of the longest streak. By the definition of expected value, we have

$$\mathrm{E}[L] = \sum_{j=0}^{n} j \Pr\{L_j\} . \tag{5.11}$$

We could try to evaluate this sum using upper bounds on each $\Pr\{L_j\}$ similar to those computed in inequality (5.10). Unfortunately, this method yields weak bounds. We can use some intuition gained by the above analysis to obtain a good bound, however. For no individual term in the summation in equation (5.11) are both the factors j and $\Pr\{L_j\}$ large. Why? When $j \geq 2\lceil \lg n \rceil$, then $\Pr\{L_j\}$ is very small, and when $j < 2\lceil \lg n \rceil$, then j is fairly small. More precisely, since the events L_j for $j = 0, 1, \ldots, n$ are disjoint, the probability that a streak of heads of length at least $2\lceil \lg n \rceil$ begins anywhere is $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\}$. Inequality (5.10) tells us that the probability that a streak of heads of length at least $2\lceil \lg n \rceil$ begins anywhere is less than $1/n$, which means that $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\} < 1/n$. Also, noting that $\sum_{j=0}^{n} \Pr\{L_j\} = 1$, we have that $\sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\{L_j\} \leq 1$. Thus, we obtain

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{2\lceil \lg n \rceil - 1} j \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} j \Pr\{L_j\} \\
&< \sum_{j=0}^{2\lceil \lg n \rceil - 1} (2\lceil \lg n \rceil) \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} n \Pr\{L_j\} \\
&= 2\lceil \lg n \rceil \sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\{L_j\} + n \sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\} \\
&< 2\lceil \lg n \rceil \cdot 1 + n \cdot \frac{1}{n} \\
&= O(\lg n) .
\end{aligned}$$

The probability that a streak of heads exceeds $r\lceil \lg n \rceil$ flips diminishes quickly with r. Let's get a rough bound on the probability that a streak of at least $r\lceil \lg n \rceil$ heads occurs, for $r \geq 1$. The probability that a streak of at least $r\lceil \lg n \rceil$ heads starts in position i is

$$\begin{aligned}
\Pr\{A_{i,r\lceil \lg n \rceil}\} &= \frac{1}{2^{r\lceil \lg n \rceil}} \\
&\leq \frac{1}{n^r} .
\end{aligned}$$

A streak of at least $r\lceil \lg n \rceil$ heads cannot start in the last $n - r\lceil \lg n \rceil + 1$ flips, but let's overestimate the probability of such a streak by allowing it to start anywhere within the n coin flips. Then the probability that a streak of at least $r\lceil \lg n \rceil$ heads

occurs is at most

$$\begin{aligned}
\Pr\left\{\bigcup_{i=1}^{n} A_{i,r\lceil \lg n\rceil}\right\} &\le \sum_{i=1}^{n} \Pr\{A_{i,r\lceil \lg n\rceil}\} \qquad \text{(by Boole's inequality (C.21))} \\
&\le \sum_{i=1}^{n} \frac{1}{n^r} \\
&= \frac{1}{n^{r-1}} .
\end{aligned}$$

Equivalently, the probability is at least $1 - 1/n^{r-1}$ that the longest streak has length less than $r\lceil \lg n\rceil$.

As an example, during $n = 1000$ coin flips, the probability of encountering a streak of at least $2\lceil \lg n\rceil = 20$ heads is at most $1/n = 1/1000$. The chance of a streak of at least $3\lceil \lg n\rceil = 30$ heads is at most $1/n^2 = 1/1{,}000{,}000$.

Let's now prove a complementary lower bound: the expected length of the longest streak of heads in n coin flips is $\Omega(\lg n)$. To prove this bound, we look for streaks of length s by partitioning the n flips into approximately n/s groups of s flips each. If we choose $s = \lfloor (\lg n)/2\rfloor$, we'll see that it is likely that at least one of these groups comes up all heads, which means that it's likely that the longest streak has length at least $s = \Omega(\lg n)$. We'll then show that the longest streak has expected length $\Omega(\lg n)$.

Let's partition the n coin flips into at least $\lfloor n/\lfloor (\lg n)/2\rfloor\rfloor$ groups of $\lfloor (\lg n)/2\rfloor$ consecutive flips and bound the probability that no group comes up all heads. By equation (5.9), the probability that the group starting in position i comes up all heads is

$$\begin{aligned}
\Pr\{A_{i,\lfloor (\lg n)/2\rfloor}\} &= \frac{1}{2^{\lfloor (\lg n)/2\rfloor}} \\
&\ge \frac{1}{\sqrt{n}} .
\end{aligned}$$

The probability that a streak of heads of length at least $\lfloor (\lg n)/2\rfloor$ does not begin in position i is therefore at most $1 - 1/\sqrt{n}$. Since the $\lfloor n/\lfloor (\lg n)/2\rfloor\rfloor$ groups are formed from mutually exclusive, independent coin flips, the probability that every one of these groups *fails* to be a streak of length $\lfloor (\lg n)/2\rfloor$ is at most

$$\begin{aligned}
\left(1 - 1/\sqrt{n}\right)^{\lfloor n/\lfloor (\lg n)/2\rfloor\rfloor} &\le \left(1 - 1/\sqrt{n}\right)^{n/\lfloor (\lg n)/2\rfloor - 1} \\
&\le \left(1 - 1/\sqrt{n}\right)^{2n/\lg n - 1} \\
&\le e^{-(2n/\lg n - 1)/\sqrt{n}} \\
&= O(e^{-\ln n}) \\
&= O(1/n) . \qquad (5.12)
\end{aligned}$$

For this argument, we used inequality (3.14), $1 + x \le e^x$, on page 66 and the fact, which you may verify, that $(2n/\lg n - 1)/\sqrt{n} \ge \ln n$ for sufficiently large n.

We want to bound the probability that the longest streak equals or exceeds $\lfloor (\lg n)/2 \rfloor$. To do so, let L be the event that the longest streak of heads equals or exceeds $s = \lfloor (\lg n)/2 \rfloor$. Let $\overline{L}$ be the complementary event, that the longest streak of heads is strictly less than s, so that $\Pr\{L\} + \Pr\{\overline{L}\} = 1$. Let F be the event that every group of s flips fails to be a streak of s heads. By inequality (5.12), we have $\Pr\{F\} = O(1/n)$. If the longest streak of heads is less than s, then certainly every group of s flips fails to be a streak of s heads, which means that event $\overline{L}$ implies event F. Of course, event F could occur even if event $\overline{L}$ does not (for example, if a streak of s or more heads crosses over the boundary between two groups), and so we have $\Pr\{\overline{L}\} \le \Pr\{F\} = O(1/n)$. Since $\Pr\{L\} + \Pr\{\overline{L}\} = 1$, we have that

$$\begin{aligned}
\Pr\{L\} &= 1 - \Pr\{\overline{L}\} \\
&\ge 1 - \Pr\{F\} \\
&= 1 - O(1/n) .
\end{aligned}$$

That is, the probability that the longest streak equals or exceeds $\lfloor (\lg n)/2 \rfloor$ is

$$\sum_{j=\lfloor (\lg n)/2 \rfloor}^{n} \Pr\{L_j\} \ge 1 - O(1/n) . \tag{5.13}$$

We can now calculate a lower bound on the expected length of the longest streak, beginning with equation (5.11) and proceeding in a manner similar to our analysis of the upper bound:

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor - 1} j \Pr\{L_j\} + \sum_{j=\lfloor (\lg n)/2 \rfloor}^{n} j \Pr\{L_j\} \\
&\ge \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor - 1} 0 \cdot \Pr\{L_j\} + \sum_{j=\lfloor (\lg n)/2 \rfloor}^{n} \lfloor (\lg n)/2 \rfloor \Pr\{L_j\} \\
&= 0 \cdot \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor - 1} \Pr\{L_j\} + \lfloor (\lg n)/2 \rfloor \sum_{j=\lfloor (\lg n)/2 \rfloor}^{n} \Pr\{L_j\} \\
&\ge 0 + \lfloor (\lg n)/2 \rfloor (1 - O(1/n)) \qquad \text{(by inequality (5.13))} \\
&= \Omega(\lg n) .
\end{aligned}$$

As with the birthday paradox, we can obtain a simpler, but approximate, analysis using indicator random variables. Instead of determining the expected length of the longest streak, we'll find the expected number of streaks with at least a given length. Let $X_{ik} = \mathrm{I}\{A_{ik}\}$ be the indicator random variable associated with a streak of heads of length at least k beginning with the ith coin flip. To count the total number of such streaks, define

$$X_k = \sum_{i=1}^{n-k+1} X_{ik} .$$

Taking expectations and using linearity of expectation, we have

$$\begin{aligned}
\mathrm{E}[X_k] &= \mathrm{E}\left[\sum_{i=1}^{n-k+1} X_{ik}\right] \\
&= \sum_{i=1}^{n-k+1} \mathrm{E}[X_{ik}] \\
&= \sum_{i=1}^{n-k+1} \Pr\{A_{ik}\} \\
&= \sum_{i=1}^{n-k+1} \frac{1}{2^k} \\
&= \frac{n-k+1}{2^k} .
\end{aligned}$$

By plugging in various values for k, we can calculate the expected number of streaks of length at least k. If this expected number is large (much greater than 1), then we expect many streaks of length k to occur, and the probability that one occurs is high. If this expected number is small (much less than 1), then we expect to see few streaks of length k, and the probability that one occurs is low. If $k = c \lg n$, for some positive constant c, we obtain

$$\begin{aligned}
\mathrm{E}[X_{c \lg n}] &= \frac{n - c \lg n + 1}{2^{c \lg n}} \\
&= \frac{n - c \lg n + 1}{n^c} \\
&= \frac{1}{n^{c-1}} - \frac{(c \lg n - 1)/n}{n^{c-1}} \\
&= \Theta(1/n^{c-1}) .
\end{aligned}$$

If c is large, the expected number of streaks of length $c \lg n$ is small, and we conclude that they are unlikely to occur. On the other hand, if $c = 1/2$, then we

obtain $\mathrm{E}[X_{(1/2)\lg n}] = \Theta(1/n^{1/2-1}) = \Theta(n^{1/2})$, and we expect there to be numerous streaks of length $(1/2)\lg n$. Therefore, one streak of such a length is likely to occur. We can conclude that the expected length of the longest streak is $\Theta(\lg n)$.

5.4.4 The online hiring problem

As a final example, let's consider a variant of the hiring problem. Suppose now that you do not wish to interview all the candidates in order to find the best one. You also want to avoid hiring and firing as you find better and better applicants. Instead, you are willing to settle for a candidate who is close to the best, in exchange for hiring exactly once. You must obey one company requirement: after each interview you must either immediately offer the position to the applicant or immediately reject the applicant. What is the trade-off between minimizing the amount of interviewing and maximizing the quality of the candidate hired?

We can model this problem in the following way. After meeting an applicant, you are able to give each one a score. Let $score(i)$ denote the score you give to the ith applicant, and assume that no two applicants receive the same score. After you have seen j applicants, you know which of the j has the highest score, but you do not know whether any of the remaining $n - j$ applicants will receive a higher score. You decide to adopt the strategy of selecting a positive integer $k < n$, interviewing and then rejecting the first k applicants, and hiring the first applicant thereafter who has a higher score than all preceding applicants. If it turns out that the best-qualified applicant was among the first k interviewed, then you hire the nth applicant—the last one interviewed. We formalize this strategy in the procedure ONLINE-MAXIMUM(k, n), which returns the index of the candidate you wish to hire.

```
ONLINE-MAXIMUM(k, n)
best-score = -∞
for i = 1 to k
    if score(i) > best-score
        best-score = score(i)
for i = k + 1 to n
    if score(i) > best-score
        return i
return n
```

If we determine, for each possible value of k, the probability that you hire the most qualified applicant, then you can choose the best possible k and implement the strategy with that value. For the moment, assume that k is fixed. Let

$M(j) = \max\{score(i) : 1 \le i \le j\}$ denote the maximum score among applicants 1 through j. Let S be the event that you succeed in choosing the best-qualified applicant, and let S_i be the event that you succeed when the best-qualified applicant is the ith one interviewed. Since the various S_i are disjoint, we have that $\Pr\{S\} = \sum_{i=1}^{n} \Pr\{S_i\}$. Noting that you never succeed when the best-qualified applicant is one of the first k, we have that $\Pr\{S_i\} = 0$ for $i = 1, 2, \ldots, k$. Thus, we obtain

$$\Pr\{S\} = \sum_{i=k+1}^{n} \Pr\{S_i\} \ . \tag{5.14}$$

We now compute $\Pr\{S_i\}$. In order to succeed when the best-qualified applicant is the ith one, two things must happen. First, the best-qualified applicant must be in position i, an event which we denote by B_i. Second, the algorithm must not select any of the applicants in positions $k+1$ through $i-1$, which happens only if, for each j such that $k+1 \le j \le i-1$, line 6 finds that $score(j) < best\text{-}score$. (Because scores are unique, we can ignore the possibility of $score(j) = best\text{-}score$.) In other words, all of the values $score(k+1)$ through $score(i-1)$ must be less than $M(k)$. If any are greater than $M(k)$, the algorithm instead returns the index of the first one that is greater. We use O_i to denote the event that none of the applicants in position $k+1$ through $i-1$ are chosen. Fortunately, the two events B_i and O_i are independent. The event O_i depends only on the relative ordering of the values in positions 1 through $i-1$, whereas B_i depends only on whether the value in position i is greater than the values in all other positions. The ordering of the values in positions 1 through $i-1$ does not affect whether the value in position i is greater than all of them, and the value in position i does not affect the ordering of the values in positions 1 through $i-1$. Thus, we can apply equation (C.17) on page 1188 to obtain

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\}\Pr\{O_i\} \ .$$

We have $\Pr\{B_i\} = 1/n$ since the maximum is equally likely to be in any one of the n positions. For event O_i to occur, the maximum value in positions 1 through $i-1$, which is equally likely to be in any of these $i-1$ positions, must be in one of the first k positions. Consequently, $\Pr\{O_i\} = k/(i-1)$ and $\Pr\{S_i\} = k/(n(i-1))$. Using equation (5.14), we have

$$\begin{aligned}\Pr\{S\} &= \sum_{i=k+1}^{n} \Pr\{S_i\} \\ &= \sum_{i=k+1}^{n} \frac{k}{n(i-1)}\end{aligned}$$

$$= \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i-1}$$

$$= \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i} .$$

We approximate by integrals to bound this summation from above and below. By the inequalities (A.19) on page 1150, we have

$$\int_{k}^{n} \frac{1}{x}\, dx \le \sum_{i=k}^{n-1} \frac{1}{i} \le \int_{k-1}^{n-1} \frac{1}{x}\, dx .$$

Evaluating these definite integrals gives us the bounds

$$\frac{k}{n}(\ln n - \ln k) \le \Pr\{S\} \le \frac{k}{n}(\ln(n-1) - \ln(k-1)) ,$$

which provide a rather tight bound for $\Pr\{S\}$. Because you wish to maximize your probability of success, let us focus on choosing the value of k that maximizes the lower bound on $\Pr\{S\}$. (Besides, the lower-bound expression is easier to maximize than the upper-bound expression.) Differentiating the expression $(k/n)(\ln n - \ln k)$ with respect to k, we obtain

$$\frac{1}{n}(\ln n - \ln k - 1) .$$

Setting this derivative equal to 0, we see that you maximize the lower bound on the probability when $\ln k = \ln n - 1 = \ln(n/e)$ or, equivalently, when $k = n/e$. Thus, if you implement our strategy with $k = n/e$, you succeed in hiring the best-qualified applicant with probability at least $1/e$.

Exercises

5.4-1

How many people must there be in a room before the probability that someone has the same birthday as you do is at least $1/2$? How many people must there be before the probability that at least two people have a birthday on July 4 is greater than $1/2$?

5.4-2

How many people must there be in a room before the probability that two people have the same birthday is at least 0.99? For that many people, what is the expected number of pairs of people who have the same birthday?

5.4-3
You toss balls into b bins until some bin contains two balls. Each toss is independent, and each ball is equally likely to end up in any bin. What is the expected number of ball tosses?

★ ***5.4-4***
For the analysis of the birthday paradox, is it important that the birthdays be mutually independent, or is pairwise independence sufficient? Justify your answer.

★ ***5.4-5***
How many people should be invited to a party in order to make it likely that there are *three* people with the same birthday?

★ ***5.4-6***
What is the probability that a k-string (defined on page 1179) over a set of size n forms a k-permutation? How does this question relate to the birthday paradox?

★ ***5.4-7***
You toss n balls into n bins, where each toss is independent and the ball is equally likely to end up in any bin. What is the expected number of empty bins? What is the expected number of bins with exactly one ball?

★ ***5.4-8***
Sharpen the lower bound on streak length by showing that in n flips of a fair coin, the probability is at least $1 - 1/n$ that a streak of length $\lg n - 2 \lg \lg n$ consecutive heads occurs.

Problems

5-1 Probabilistic counting
With a b-bit counter, we can ordinarily only count up to $2^b - 1$. With R. Morris's ***probabilistic counting***, we can count up to a much larger value at the expense of some loss of precision.

We let a counter value of i represent a count of n_i for $i = 0, 1, \ldots, 2^b - 1$, where the n_i form an increasing sequence of nonnegative values. We assume that the initial value of the counter is 0, representing a count of $n_0 = 0$. The INCREMENT operation works on a counter containing the value i in a probabilistic manner. If $i = 2^b - 1$, then the operation reports an overflow error. Otherwise, the INCREMENT operation increases the counter by 1 with probability $1/(n_{i+1} - n_i)$, and it leaves the counter unchanged with probability $1 - 1/(n_{i+1} - n_i)$.

If we select $n_i = i$ for all $i \geq 0$, then the counter is an ordinary one. More interesting situations arise if we select, say, $n_i = 2^{i-1}$ for $i > 0$ or $n_i = F_i$ (the ith Fibonacci number—see equation (3.31) on page 69).

For this problem, assume that n_{2^b-1} is large enough that the probability of an overflow error is negligible.

a. Show that the expected value represented by the counter after n INCREMENT operations have been performed is exactly n.

b. The analysis of the variance of the count represented by the counter depends on the sequence of the n_i. Let us consider a simple case: $n_i = 100i$ for all $i \geq 0$. Estimate the variance in the value represented by the register after n INCREMENT operations have been performed.

5-2 Searching an unsorted array

This problem examines three algorithms for searching for a value x in an unsorted array A consisting of n elements.

Consider the following randomized strategy: pick a random index i into A. If $A[i] = x$, then terminate; otherwise, continue the search by picking a new random index into A. Continue picking random indices into A until you find an index j such that $A[j] = x$ or until every element of A has been checked. This strategy may examine a given element more than once, because it picks from the whole set of indices each time.

a. Write pseudocode for a procedure RANDOM-SEARCH to implement the strategy above. Be sure that your algorithm terminates when all indices into A have been picked.

b. Suppose that there is exactly one index i such that $A[i] = x$. What is the expected number of indices into A that must be picked before x is found and RANDOM-SEARCH terminates?

c. Generalizing your solution to part (b), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the expected number of indices into A that must be picked before x is found and RANDOM-SEARCH terminates? Your answer should be a function of n and k.

d. Suppose that there are no indices i such that $A[i] = x$. What is the expected number of indices into A that must be picked before all elements of A have been checked and RANDOM-SEARCH terminates?

Now consider a deterministic linear search algorithm. The algorithm, which we call DETERMINISTIC-SEARCH, searches A for x in order, considering $A[1]$, $A[2]$,

$A[3], \ldots, A[n]$ until either it finds $A[i] = x$ or it reaches the end of the array. Assume that all possible permutations of the input array are equally likely.

e. Suppose that there is exactly one index i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH?

f. Generalizing your solution to part (e), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH? Your answer should be a function of n and k.

g. Suppose that there are no indices i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH?

Finally, consider a randomized algorithm SCRAMBLE-SEARCH that first randomly permutes the input array and then runs the deterministic linear search given above on the resulting permuted array.

h. Letting k be the number of indices i such that $A[i] = x$, give the worst-case and expected running times of SCRAMBLE-SEARCH for the cases in which $k = 0$ and $k = 1$. Generalize your solution to handle the case in which $k \geq 1$.

i. Which of the three searching algorithms would you use? Explain your answer.

Chapter notes

Bollobás [65], Hofri [223], and Spencer [420] contain a wealth of advanced probabilistic techniques. The advantages of randomized algorithms are discussed and surveyed by Karp [249] and Rabin [372]. The textbook by Motwani and Raghavan [336] gives an extensive treatment of randomized algorithms.

The RANDOMLY-PERMUTE procedure is by Durstenfeld [128], based on an earlier procedure by Fisher and Yates [143, p. 34].

Several variants of the hiring problem have been widely studied. These problems are more commonly referred to as "secretary problems." Examples of work in this area are the paper by Ajtai, Meggido, and Waarts [11] and another by Kleinberg [258], which ties the secretary problem to online ad auctions.

Part II Sorting and Order Statistics

Introduction

This part presents several algorithms that solve the following ***sorting problem***:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

The input sequence is usually an n-element array, although it may be represented in some other fashion, such as a linked list.

The structure of the data

In practice, the numbers to be sorted are rarely isolated values. Each is usually part of a collection of data called a ***record***. Each record contains a ***key***, which is the value to be sorted. The remainder of the record consists of ***satellite data***, which are usually carried around with the key. In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, it often pays to permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

In a sense, it is these implementation details that distinguish an algorithm from a full-blown program. A sorting algorithm describes the *method* to determine the sorted order, regardless of whether what's being sorted are individual numbers or large records containing many bytes of satellite data. Thus, when focusing on the problem of sorting, we typically assume that the input consists only of numbers. Translating an algorithm for sorting numbers into a program for sorting records is conceptually straightforward, although in a given engineering situation other subtleties may make the actual programming task a challenge.

Why sorting?

Many computer scientists consider sorting to be the most fundamental problem in the study of algorithms. There are several reasons:

- Sometimes an application inherently needs to sort information. For example, in order to prepare customer statements, banks need to sort checks by check number.
- Algorithms often use sorting as a key subroutine. For example, a program that renders graphical objects which are layered on top of each other might have to sort the objects according to an "above" relation so that it can draw these objects from bottom to top. We will see numerous algorithms in this text that use sorting as a subroutine.
- We can draw from among a wide variety of sorting algorithms, and they employ a rich set of techniques. In fact, many important techniques used throughout algorithm design appear in sorting algorithms that have been developed over the years. In this way, sorting is also a problem of historical interest.
- We can prove a nontrivial lower bound for sorting (as we'll do in Chapter 8). Since the best upper bounds match the lower bound asymptotically, we can conclude that certain of our sorting algorithms are asymptotically optimal. Moreover, we can use the lower bound for sorting to prove lower bounds for various other problems.
- Many engineering issues come to the fore when implementing sorting algorithms. The fastest sorting program for a particular situation may depend on many factors, such as prior knowledge about the keys and satellite data, the memory hierarchy (caches and virtual memory) of the host computer, and the software environment. Many of these issues are best dealt with at the algorithmic level, rather than by "tweaking" the code.

Sorting algorithms

We introduced two algorithms that sort n real numbers in Chapter 2. Insertion sort takes $\Theta(n^2)$ time in the worst case. Because its inner loops are tight, however, it is a fast sorting algorithm for small input sizes. Moreover, unlike merge sort, it sorts ***in place***, meaning that at most a constant number of elements of the input array are ever stored outside the array, which can be advantageous for space efficiency. Merge sort has a better asymptotic running time, $\Theta(n \lg n)$, but the MERGE procedure it uses does not operate in place. (We'll see a parallelized version of merge sort in Section 26.3.)

This part introduces two more algorithms that sort arbitrary real numbers. Heapsort, presented in Chapter 6, sorts n numbers in place in $O(n \lg n)$ time. It uses an important data structure, called a heap, which can also implement a priority queue.

Quicksort, in Chapter 7, also sorts n numbers in place, but its worst-case running time is $\Theta(n^2)$. Its expected running time is $\Theta(n \lg n)$, however, and it generally outperforms heapsort in practice. Like insertion sort, quicksort has tight code, and so the hidden constant factor in its running time is small. It is a popular algorithm for sorting large arrays.

Insertion sort, merge sort, heapsort, and quicksort are all comparison sorts: they determine the sorted order of an input array by comparing elements. Chapter 8 begins by introducing the decision-tree model in order to study the performance limitations of comparison sorts. Using this model, we prove a lower bound of $\Omega(n \lg n)$ on the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

Chapter 8 then goes on to show that we might be able to beat this lower bound of $\Omega(n \lg n)$ if an algorithm can gather information about the sorted order of the input by means other than comparing elements. The counting sort algorithm, for example, assumes that the input numbers belong to the set $\{0, 1, \ldots, k\}$. By using array indexing as a tool for determining relative order, counting sort can sort n numbers in $\Theta(k + n)$ time. Thus, when $k = O(n)$, counting sort runs in time that is linear in the size of the input array. A related algorithm, radix sort, can be used to extend the range of counting sort. If there are n integers to sort, each integer has d digits, and each digit can take on up to k possible values, then radix sort can sort the numbers in $\Theta(d(n + k))$ time. When d is a constant and k is $O(n)$, radix sort runs in linear time. A third algorithm, bucket sort, requires knowledge of the probabilistic distribution of numbers in the input array. It can sort n real numbers uniformly distributed in the half-open interval $[0, 1)$ in average-case $O(n)$ time.

The table on the following page summarizes the running times of the sorting algorithms from Chapters 2 and 6–8. As usual, n denotes the number of items to sort. For counting sort, the items to sort are integers in the set $\{0, 1, \ldots, k\}$. For radix sort, each item is a d-digit number, where each digit takes on k possible values. For bucket sort, we assume that the keys are real numbers uniformly distributed in the half-open interval $[0, 1)$. The rightmost column gives the average-case or expected running time, indicating which one it gives when it differs from the worst-case running time. We omit the average-case running time of heapsort because we do not analyze it in this book.

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

Order statistics

The ith order statistic of a set of n numbers is the ith smallest number in the set. You can, of course, select the ith order statistic by sorting the input and indexing the ith element of the output. With no assumptions about the input distribution, this method runs in $\Omega(n \lg n)$ time, as the lower bound proved in Chapter 8 shows.

Chapter 9 shows how to find the ith smallest element in $O(n)$ time, even when the elements are arbitrary real numbers. We present a randomized algorithm with tight pseudocode that runs in $\Theta(n^2)$ time in the worst case, but whose expected running time is $O(n)$. We also give a more complicated algorithm that runs in $O(n)$ worst-case time.

Background

Although most of this part does not rely on difficult mathematics, some sections do require mathematical sophistication. In particular, analyses of quicksort, bucket sort, and the order-statistic algorithm use probability, which is reviewed in Appendix C, and the material on probabilistic analysis and randomized algorithms in Chapter 5.

6 Heapsort

This chapter introduces another sorting algorithm: heapsort. Like merge sort, but unlike insertion sort, heapsort's running time is $O(n \lg n)$. Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time. Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Heapsort also introduces another algorithm design technique: using a data structure, in this case one we call a "heap," to manage information. Not only is the heap data structure useful for heapsort, but it also makes an efficient priority queue. The heap data structure will reappear in algorithms in later chapters.

The term "heap" was originally coined in the context of heapsort, but it has since come to refer to "garbage-collected storage," such as the programming languages Java and Python provide. Please don't be confused. The heap data structure is *not* garbage-collected storage. This book is consistent in using the term "heap" to refer to the data structure, not the storage class.

6.1 Heaps

The ***(binary) heap*** data structure is an array object that we can view as a nearly complete binary tree (see Section B.5.3), as shown in Figure 6.1. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array $A[1:n]$ that represents a heap is an object with an attribute $A.\mathit{heap\text{-}size}$, which represents how many elements in the heap are stored within array A. That is, although $A[1:n]$ may contain numbers, only the elements in $A[1:A.\mathit{heap\text{-}size}]$, where $0 \le A.\mathit{heap\text{-}size} \le n$, are valid elements of the heap. If $A.\mathit{heap\text{-}size} = 0$, then the heap is empty. The root of the tree is $A[1]$, and given the index i of a node,

Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships, with parents always to the left of their children. The tree has height 3, and the node at index 4 (with value 8) has height 1.

there's a simple way to compute the indices of its parent, left child, and right child with the one-line procedures PARENT, LEFT, and RIGHT.

```
PARENT(i)
return ⌊i/2⌋

LEFT(i)
return 2i

RIGHT(i)
return 2i + 1
```

On most computers, the LEFT procedure can compute $2i$ in one instruction by simply shifting the binary representation of i left by one bit position. Similarly, the RIGHT procedure can quickly compute $2i + 1$ by shifting the binary representation of i left by one bit position and then adding 1. The PARENT procedure can compute $\lfloor i/2 \rfloor$ by shifting i right one bit position. Good implementations of heapsort often implement these procedures as macros or inline procedures.

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a ***heap property***, the specifics of which depend on the kind of heap. In a ***max-heap***, the ***max-heap property*** is that for every node i other than the root,

$$A[\text{PARENT}(i)] \geq A[i] ,$$

that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself. A ***min-heap*** is organized in the opposite way: the ***min-heap property*** is that for every node i other than the root,

$$A[\text{PARENT}(i)] \le A[i] .$$

The smallest element in a min-heap is at the root.

The heapsort algorithm uses max-heaps. Min-heaps commonly implement priority queues, which we discuss in Section 6.5. We'll be precise in specifying whether we need a max-heap or a min-heap for any particular application, and when properties apply to either max-heaps or min-heaps, we just use the term "heap."

Viewing a heap as a tree, we define the ***height*** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\lg n)$ (see Exercise 6.1-2). As we'll see, the basic operations on heaps run in time at most proportional to the height of the tree and thus take $O(\lg n)$ time. The remainder of this chapter presents some basic procedures and shows how they are used in a sorting algorithm and a priority-queue data structure.

- The MAX-HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The HEAPSORT procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The procedures MAX-HEAP-INSERT, MAX-HEAP-EXTRACT-MAX, MAX-HEAP-INCREASE-KEY, and MAX-HEAP-MAXIMUM allow the heap data structure to implement a priority queue. They run in $O(\lg n)$ time plus the time for mapping between objects being inserted into the priority queue and indices in the heap.

Exercises

6.1-1

What are the minimum and maximum numbers of elements in a heap of height h?

6.1-2

Show that an n-element heap has height $\lfloor \lg n \rfloor$.

6.1-3
Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

6.1-4
Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

6.1-5
At which levels in a max-heap might the kth largest element reside, for $2 \leq k \leq \lfloor n/2 \rfloor$, assuming that all elements are distinct?

6.1-6
Is an array that is in sorted order a min-heap?

6.1-7
Is the array with values $\langle 33, 19, 20, 15, 13, 10, 2, 13, 16, 12 \rangle$ a max-heap?

6.1-8
Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2 Maintaining the heap property

The procedure MAX-HEAPIFY on the facing page maintains the max-heap property. Its inputs are an array A with the *heap-size* attribute and an index i into the array. When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that $A[i]$ might be smaller than its children, thus violating the max-heap property. MAX-HEAPIFY lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

Figure 6.2 illustrates the action of MAX-HEAPIFY. Each step determines the largest of the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$ and stores the index of the largest element in *largest*. If $A[i]$ is largest, then the subtree rooted at node i is already a max-heap and nothing else needs to be done. Otherwise, one of the two children contains the largest element. Positions i and *largest* swap their contents, which causes node i and its children to satisfy the max-heap property. The node indexed by *largest*, however, just had its value decreased, and thus the subtree rooted at *largest* might violate the max-heap property. Consequently, MAX-HEAPIFY calls itself recursively on that subtree.

Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. The node that potentially violates the max-heap property is shown in blue. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After $A[4]$ and $A[9]$ are swapped, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

To analyze MAX-HEAPIFY, let $T(n)$ be the worst-case running time that the procedure takes on a subtree of size at most n. For a tree rooted at a given node i, the running time is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$, plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children's subtrees each have size at most $2n/3$ (see Exercise 6.2-2), and therefore we can describe the running time of MAX-HEAPIFY by the recurrence

$$T(n) \le T(2n/3) + \Theta(1) . \tag{6.1}$$

The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1 on page 102), is $T(n) = O(\lg n)$. Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height h as $O(h)$.

Exercises

6.2-1
Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

6.2-2
Show that each child of the root of an n-node heap is the root of a subtree containing at most $2n/3$ nodes. What is the smallest constant α such that each subtree has at most αn nodes? How does that affect the recurrence (6.1) and its solution?

6.2-3
Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare with that of MAX-HEAPIFY?

6.2-4
What is the effect of calling MAX-HEAPIFY(A, i) when the element $A[i]$ is larger than its children?

6.2-5
What is the effect of calling MAX-HEAPIFY(A, i) for $i > A.\mathit{heap\text{-}size}/2$?

6.2-6
The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, for which some compilers might produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

6.2-7
Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (*Hint:* For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

6.3 Building a heap

The procedure BUILD-MAX-HEAP converts an array $A[1:n]$ into a max-heap by calling MAX-HEAPIFY in a bottom-up manner. Exercise 6.1-8 says that the elements in the subarray $A[\lfloor n/2 \rfloor + 1 : n]$ are all leaves of the tree, and so each is a 1-element heap to begin with. BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one. Figure 6.3 shows an example of the action of BUILD-MAX-HEAP.

```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–3, each node $i + 1$, $i + 2, \ldots, n$ is the root of a max-heap.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, that the loop terminates, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1$, $i + 2, \ldots, n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. The node indexed by i in each iteration is shown in blue. **(a)** A 10-element input array A and the binary tree it represents. The loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

Termination: The loop makes exactly $\lfloor n/2 \rfloor$ iterations, and so it terminates. At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.

We can compute a simple upper bound on the running time of BUILD-MAX-HEAP as follows. Each call to MAX-HEAPIFY costs $O(\lg n)$ time, and BUILD-MAX-HEAP makes $O(n)$ such calls. Thus, the running time is $O(n \lg n)$. This upper bound, though correct, is not as tight as it can be.

We can derive a tighter asymptotic bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and that the heights of most nodes are small. Our tighter analysis relies on the properties that an n-element heap has height $\lfloor \lg n \rfloor$ (see Exercise 6.1-2) and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h (see Exercise 6.3-4).

The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$. Letting c be the constant implicit in the asymptotic notation, we can express the total cost of BUILD-MAX-HEAP as being bounded from above by $\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil ch$. As Exercise 6.3-2 shows, we have $\lceil n/2^{h+1} \rceil \geq 1/2$ for $0 \leq h \leq \lfloor \lg n \rfloor$. Since $\lceil x \rceil \leq 2x$ for any $x \geq 1/2$, we have $\lceil n/2^{h+1} \rceil \leq n/2^h$. We thus obtain

$$
\begin{aligned}
\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch & \\
&\leq \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch \\
&= cn \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \\
&\leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} \\
&\leq cn \cdot \frac{1/2}{(1-1/2)^2} \quad \text{(by equation (A.11) on page 1142 with } x = 1/2) \\
&= O(n) \,.
\end{aligned}
$$

Hence, we can build a max-heap from an unordered array in linear time.

To build a min-heap, use the procedure BUILD-MIN-HEAP, which is the same as BUILD-MAX-HEAP but with the call to MAX-HEAPIFY in line 3 replaced by a call to MIN-HEAPIFY (see Exercise 6.2-3). BUILD-MIN-HEAP produces a min-heap from an unordered linear array in linear time.

Exercises

6.3-1
Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2
Show that $\lceil n/2^{h+1} \rceil \geq 1/2$ for $0 \leq h \leq \lfloor \lg n \rfloor$.

6.3-3
Why does the loop index i in line 2 of BUILD-MAX-HEAP decrease from $\lfloor n/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor n/2 \rfloor$?

6.3-4
Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

6.4 The heapsort algorithm

The heapsort algorithm, given by the procedure HEAPSORT, starts by calling the BUILD-MAX-HEAP procedure to build a max-heap on the input array $A[1:n]$. Since the maximum element of the array is stored at the root $A[1]$, HEAPSORT can place it into its correct final position by exchanging it with $A[n]$. If the procedure then discards node n from the heap—and it can do so by simply decrementing $A.heap\text{-}size$—the children of the root remain max-heaps, but the new root element might violate the max-heap property. To restore the max-heap property, the procedure just calls MAX-HEAPIFY$(A, 1)$, which leaves a max-heap in $A[1:n-1]$. The HEAPSORT procedure then repeats this process for the max-heap of size $n-1$ down to a heap of size 2. (See Exercise 6.4-2 for a precise loop invariant.)

```
HEAPSORT(A, n)
1  BUILD-MAX-HEAP(A, n)
2  for i = n downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size - 1
5      MAX-HEAPIFY(A, 1)
```

Figure 6.4 shows an example of the operation of HEAPSORT after line 1 has built the initial max-heap. The figure shows the max-heap before the first iteration of the **for** loop of lines 2–5 and after each iteration.

Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only blue nodes remain in the heap. Tan nodes contain the largest values in the array, in sorted order. **(k)** The resulting sorted array A.

The HEAPSORT procedure takes $O(n \lg n)$ time, since the call to BUILD-MAX-HEAP takes $O(n)$ time and each of the $n-1$ calls to MAX-HEAPIFY takes $O(\lg n)$ time.

Exercises

6.4-1
Using Figure 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

6.4-2
Argue the correctness of HEAPSORT using the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–5, the subarray $A[1:i]$ is a max-heap containing the i smallest elements of $A[1:n]$, and the subarray $A[i+1:n]$ contains the $n-i$ largest elements of $A[1:n]$, sorted.

6.4-3
What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? How about if the array is already sorted in decreasing order?

6.4-4
Show that the worst-case running time of HEAPSORT is $\Omega(n \lg n)$.

★ ***6.4-5***
Show that when all the elements of A are distinct, the best-case running time of HEAPSORT is $\Omega(n \lg n)$.

6.5 Priority queues

In Chapter 8, we will see that any comparison-based sorting algorithm requires $\Omega(n \lg n)$ comparisons and hence $\Omega(n \lg n)$ time. Therefore, heapsort is asymptotically optimal among comparison-based sorting algorithms. Yet, a good implementation of quicksort, presented in Chapter 7, usually beats it in practice. Nevertheless, the heap data structure itself has many uses. In this section, we present one of the most popular applications of a heap: as an efficient priority queue. As with heaps, priority queues come in two forms: max-priority queues and min-priority queues. We'll focus here on how to implement max-priority queues, which are in turn based on max-heaps. Exercise 6.5-3 asks you to write the procedures for min-priority queues.

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations:

INSERT(S, x, k) inserts the element x with key k into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

MAXIMUM(S) returns the element of S with the largest key.

EXTRACT-MAX(S) removes and returns the element of S with the largest key.

INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Among their other applications, you can use max-priority queues to schedule jobs on a computer shared among multiple users. The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending by calling EXTRACT-MAX. The scheduler can add a new job to the queue at any time by calling INSERT.

Alternatively, a ***min-priority queue*** supports the operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY. A min-priority queue can be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. The simulation program calls EXTRACT-MIN at each step to choose the next event to simulate. As new events are produced, the simulator inserts them into the min-priority queue by calling INSERT. We'll see other uses for min-priority queues, highlighting the DECREASE-KEY operation, in Chapters 21 and 22.

When you use a heap to implement a priority queue within a given application, elements of the priority queue correspond to objects in the application. Each object contains a key. If the priority queue is implemented by a heap, you need to determine which application object corresponds to a given heap element, and vice versa. Because the heap elements are stored in an array, you need a way to map application objects to and from array indices.

One way to map between application objects and heap elements uses ***handles***, which are additional information stored in the objects and heap elements that give enough information to perform the mapping. Handles are often implemented to be opaque to the surrounding code, thereby maintaining an abstraction barrier between the application and the priority queue. For example, the handle within an application object might contain the corresponding index into the heap array. But since only the code for the priority queue accesses this index, the index is entirely hidden from the application code. Because heap elements change locations within

the array during heap operations, an actual implementation of the priority queue, upon relocating a heap element, must also update the array indices in the corresponding handles. Conversely, each element in the heap might contain a pointer to the corresponding application object, but the heap element knows this pointer as only an opaque handle and the application maps this handle to an application object. Typically, the worst-case overhead for maintaining handles is $O(1)$ per access.

As an alternative to incorporating handles in application objects, you can store within the priority queue a mapping from application objects to array indices in the heap. The advantage of doing so is that the mapping is contained entirely within the priority queue, so that the application objects need no further embellishment. The disadvantage lies in the additional cost of establishing and maintaining the mapping. One option for the mapping is a hash table (see Chapter 11).[1] The added expected time for a hash table to map an object to an array index is just $O(1)$, though the worst-case time can be as bad as $\Theta(n)$.

Let's see how to implement the operations of a max-priority queue using a max-heap. In the previous sections, we treated the array elements as the keys to be sorted, implicitly assuming that any satellite data moved with the corresponding keys. When a heap implements a priority queue, we instead treat each array element as a pointer to an object in the priority queue, so that the object is analogous to the satellite data when sorting. We further assume that each such object has an attribute *key*, which determines where in the heap the object belongs. For a heap implemented by an array A, we refer to $A[i].key$.

The procedure MAX-HEAP-MAXIMUM on the facing page implements the MAXIMUM operation in $\Theta(1)$ time, and MAX-HEAP-EXTRACT-MAX implements the operation EXTRACT-MAX. MAX-HEAP-EXTRACT-MAX is similar to the **for** loop body (lines 3–5) of the HEAPSORT procedure. We implicitly assume that MAX-HEAPIFY compares priority-queue objects based on their *key* attributes. We also assume that when MAX-HEAPIFY exchanges elements in the array, it is exchanging pointers and also that it updates the mapping between objects and array indices. The running time of MAX-HEAP-EXTRACT-MAX is $O(\lg n)$, since it performs only a constant amount of work on top of the $O(\lg n)$ time for MAX-HEAPIFY, plus whatever overhead is incurred within MAX-HEAPIFY for mapping priority-queue objects to array indices.

The procedure MAX-HEAP-INCREASE-KEY on page 176 implements the INCREASE-KEY operation. It first verifies that the new key k will not cause the key in the object x to decrease, and if there is no problem, it gives x the new key value. The procedure then finds the index i in the array corresponding to object x,

[1] In Python, dictionaries are implemented with hash tables.

```
MAX-HEAP-MAXIMUM(A)
if A.heap-size < 1
    error "heap underflow"
return A[1]

MAX-HEAP-EXTRACT-MAX(A)
max = MAX-HEAP-MAXIMUM(A)
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

so that $A[i]$ is x. Because increasing the key of $A[i]$ might violate the max-heap property, the procedure then, in a manner reminiscent of the insertion loop (lines 5–7) of INSERTION-SORT on page 19, traverses a simple path from this node toward the root to find a proper place for the newly increased key. As MAX-HEAP-INCREASE-KEY traverses this path, it repeatedly compares an element's key to that of its parent, exchanging pointers and continuing if the element's key is larger, and terminating if the element's key is smaller, since the max-heap property now holds. (See Exercise 6.5-7 for a precise loop invariant.) Like MAX-HEAPIFY when used in a priority queue, MAX-HEAP-INCREASE-KEY updates the information that maps objects to array indices when array elements are exchanged. Figure 6.5 shows an example of a MAX-HEAP-INCREASE-KEY operation. In addition to the overhead for mapping priority queue objects to array indices, the running time of MAX-HEAP-INCREASE-KEY on an n-element heap is $O(\lg n)$, since the path traced from the node updated in line 3 to the root has length $O(\lg n)$.

The procedure MAX-HEAP-INSERT on the next page implements the INSERT operation. It takes as inputs the array A implementing the max-heap, the new object x to be inserted into the max-heap, and the size n of array A. The procedure first verifies that the array has room for the new element. It then expands the max-heap by adding to the tree a new leaf whose key is $-\infty$. Then it calls MAX-HEAP-INCREASE-KEY to set the key of this new element to its correct value and maintain the max-heap property. The running time of MAX-HEAP-INSERT on an n-element heap is $O(\lg n)$ plus the overhead for mapping priority queue objects to indices.

In summary, a heap can support any priority-queue operation on a set of size n in $O(\lg n)$ time, plus the overhead for mapping priority queue objects to array indices.

```
MAX-HEAP-INCREASE-KEY(A, x, k)
if k < x.key
    error "new key is smaller than current key"
x.key = k
find the index i in array A where object x occurs
while i > 1 and A[PARENT(i)].key < A[i].key
    exchange A[i] with A[PARENT(i)], updating the information that maps
        priority queue objects to array indices
    i = PARENT(i)

MAX-HEAP-INSERT(A, x, n)
if A.heap-size == n
    error "heap overflow"
A.heap-size = A.heap-size + 1
k = x.key
x.key = -∞
A[A.heap-size] = x
map x to index heap-size in the array
MAX-HEAP-INCREASE-KEY(A, x, k)
```

Exercises

6.5-1

Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-EXTRACT-MAX on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-2

Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-INSERT$(A, 10)$ on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-3

Write pseudocode to implement a min-priority queue with a min-heap by writing the procedures MIN-HEAP-MINIMUM, MIN-HEAP-EXTRACT-MIN, MIN-HEAP-DECREASE-KEY, and MIN-HEAP-INSERT.

6.5-4

Write pseudocode for the procedure MAX-HEAP-DECREASE-KEY(A, x, k) in a max-heap. What is the running time of your procedure?

Figure 6.5 The operation of MAX-HEAP-INCREASE-KEY. Only the key of each element in the priority queue is shown. The node indexed by i in each iteration is shown in blue. **(a)** The max-heap of Figure 6.4(a) with i indexing the node whose key is about to be increased. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 5–7, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

6.5-5

Why does MAX-HEAP-INSERT bother setting the key of the inserted object to $-\infty$ in line 5 given that line 8 will set the object's key to the desired value?

6.5-6

Professor Uriah suggests replacing the **while** loop of lines 5–7 in MAX-HEAP-INCREASE-KEY by a call to MAX-HEAPIFY. Explain the flaw in the professor's idea.

6.5-7

Argue the correctness of MAX-HEAP-INCREASE-KEY using the following loop invariant:

At the start of each iteration of the **while** loop of lines 5–7:

a. If both nodes PARENT(i) and LEFT(i) exist, then $A[\text{PARENT}(i)].key \geq A[\text{LEFT}(i)].key$.
b. If both nodes PARENT(i) and RIGHT(i) exist, then $A[\text{PARENT}(i)].key \geq A[\text{RIGHT}(i)].key$.
c. The subarray $A[1 : A.heap\text{-}size]$ satisfies the max-heap property, except that there may be one violation, which is that $A[i].key$ may be greater than $A[\text{PARENT}(i)].key$.

You may assume that the subarray $A[1 : A.heap\text{-}size]$ satisfies the max-heap property at the time MAX-HEAP-INCREASE-KEY is called.

6.5-8
Each exchange operation on line 6 of MAX-HEAP-INCREASE-KEY typically requires three assignments, not counting the updating of the mapping from objects to array indices. Show how to use the idea of the inner loop of INSERTION-SORT to reduce the three assignments to just one assignment.

6.5-9
Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue. (Queues and stacks are defined in Section 10.1.3.)

6.5-10
The operation MAX-HEAP-DELETE(A, x) deletes the object x from max-heap A. Give an implementation of MAX-HEAP-DELETE for an n-element max-heap that runs in $O(\lg n)$ time plus the overhead for mapping priority queue objects to array indices.

6.5-11
Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (*Hint:* Use a min-heap for k-way merging.)

Problems

6-1 Building a heap using insertion
One way to build a heap is by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the procedure BUILD-MAX-HEAP′ on the facing page. It assumes that the objects being inserted are just the heap elements.

```
BUILD-MAX-HEAP′(A, n)
A.heap-size = 1
for i = 2 to n
    MAX-HEAP-INSERT(A, A[i], n)
```

a. Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

b. Show that in the worst case, BUILD-MAX-HEAP′ requires $\Theta(n \lg n)$ time to build an n-element heap.

6-2 Analysis of d-ary heaps

A ***d-ary heap*** is like a binary heap, but (with one possible exception) nonleaf nodes have d children instead of two children. In all parts of this problem, assume that the time to maintain the mapping between objects and heap elements is $O(1)$ per operation.

a. Describe how to represent a d-ary heap in an array.

b. Using Θ-notation, express the height of a d-ary heap of n elements in terms of n and d.

c. Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.

d. Give an efficient implementation of INCREASE-KEY in a d-ary max-heap. Analyze its running time in terms of d and n.

e. Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.

6-3 Young tableaus

An $m \times n$ ***Young tableau*** is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be ∞, which we treat as nonexistent elements. Thus, a Young tableau can be used to hold $r \le mn$ finite numbers.

a. Draw a 4×4 Young tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

b. Argue that an $m \times n$ Young tableau Y is empty if $Y[1,1] = \infty$. Argue that Y is full (contains mn elements) if $Y[m,n] < \infty$.

c. Give an algorithm to implement EXTRACT-MIN on a nonempty $m \times n$ Young tableau that runs in $O(m + n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m-1) \times n$ or an $m \times (n-1)$ subproblem. (*Hint:* Think about MAX-HEAPIFY.) Explain why your implementation of EXTRACT-MIN runs in $O(m + n)$ time.

d. Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m + n)$ time.

e. Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort n^2 numbers in $O(n^3)$ time.

f. Give an $O(m + n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

Chapter notes

The heapsort algorithm was invented by Williams [456], who also described how to implement a priority queue with a heap. The BUILD-MAX-HEAP procedure was suggested by Floyd [145]. Schaffer and Sedgewick [395] showed that in the best case, the number of times elements move in the heap during heapsort is approximately $(n/2) \lg n$ and that the average number of moves is approximately $n \lg n$.

We use min-heaps to implement min-priority queues in Chapters 15, 21, and 22. Other, more complicated, data structures give better time bounds for certain min-priority queue operations. Fredman and Tarjan [156] developed Fibonacci heaps, which support INSERT and DECREASE-KEY in $O(1)$ amortized time (see Chapter 16). That is, the average worst-case running time for these operations is $O(1)$. Brodal, Lagogiannis, and Tarjan [73] subsequently devised strict Fibonacci heaps, which make these time bounds the actual running times. If the keys are unique and drawn from the set $\{0, 1, \ldots, n-1\}$ of nonnegative integers, van Emde Boas trees [440, 441] support the operations INSERT, DELETE, SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR in $O(\lg \lg n)$ time.

If the data are b-bit integers, and the computer memory consists of addressable b-bit words, Fredman and Willard [157] showed how to implement MINIMUM in $O(1)$ time and INSERT and EXTRACT-MIN in $O(\sqrt{\lg n})$ time. Thorup [436] has

improved the $O(\sqrt{\lg n})$ bound to $O(\lg \lg n)$ time by using randomized hashing, requiring only linear space.

An important special case of priority queues occurs when the sequence of EXTRACT-MIN operations is ***monotone***, that is, the values returned by successive EXTRACT-MIN operations are monotonically increasing over time. This case arises in several important applications, such as Dijkstra's single-source shortest-paths algorithm, which we discuss in Chapter 22, and in discrete-event simulation. For Dijkstra's algorithm it is particularly important that the DECREASE-KEY operation be implemented efficiently. For the monotone case, if the data are integers in the range $1, 2, \ldots, C$, Ahuja, Mehlhorn, Orlin, and Tarjan [8] describe how to implement EXTRACT-MIN and INSERT in $O(\lg C)$ amortized time (Chapter 16 presents amortized analysis) and DECREASE-KEY in $O(1)$ time, using a data structure called a radix heap. The $O(\lg C)$ bound can be improved to $O(\sqrt{\lg C})$ using Fibonacci heaps in conjunction with radix heaps. Cherkassky, Goldberg, and Silverstein [90] further improved the bound to $O(\lg^{1/3+\epsilon} C)$ expected time by combining the multilevel bucketing structure of Denardo and Fox [112] with the heap of Thorup mentioned earlier. Raman [375] further improved these results to obtain a bound of $O\left(\min\left\{\lg^{1/4+\epsilon} C, \lg^{1/3+\epsilon} n\right\}\right)$, for any fixed $\epsilon > 0$.

Many other variants of heaps have been proposed. Brodal [72] surveys some of these developments.

7 Quicksort

The quicksort algorithm has a worst-case running time of $\Theta(n^2)$ on an input array of n numbers. Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on average: its expected running time is $\Theta(n \lg n)$ when all numbers are distinct, and the constant factors hidden in the $\Theta(n \lg n)$ notation are small. Unlike merge sort, it also has the advantage of sorting in place (see page 158), and it works well even in virtual-memory environments.

Our study of quicksort is broken into four sections. Section 7.1 describes the algorithm and an important subroutine used by quicksort for partitioning. Because the behavior of quicksort is complex, we'll start with an intuitive discussion of its performance in Section 7.2 and analyze it precisely at the end of the chapter. Section 7.3 presents a randomized version of quicksort. When all elements are distinct,[1] this randomized algorithm has a good expected running time and no particular input elicits its worst-case behavior. (See Problem 7-2 for the case in which elements may be equal.) Section 7.4 analyzes the randomized algorithm, showing that it runs in $\Theta(n^2)$ time in the worst case and, assuming distinct elements, in expected $O(n \lg n)$ time.

---

[1] You can enforce the assumption that the values in an array A are distinct at the cost of $\Theta(n)$ additional space and only constant overhead in running time by converting each input value $A[i]$ to an ordered pair $(A[i], i)$ with $(A[i], i) < (A[j], j)$ if $A[i] < A[j]$ or if $A[i] = A[j]$ and $i < j$. There are also more practical variants of quicksort that work well when elements are not distinct.

7.1 Description of quicksort

Quicksort, like merge sort, applies the divide-and-conquer method introduced in Section 2.3.1. Here is the three-step divide-and-conquer process for sorting a subarray $A[p:r]$:

Divide by partitioning (rearranging) the array $A[p:r]$ into two (possibly empty) subarrays $A[p:q-1]$ (the ***low side***) and $A[q+1:r]$ (the ***high side***) such that each element in the low side of the partition is less than or equal to the ***pivot*** $A[q]$, which is, in turn, less than or equal to each element in the high side. Compute the index q of the pivot as part of this partitioning procedure.

Conquer by calling quicksort recursively to sort each of the subarrays $A[p:q-1]$ and $A[q+1:r]$.

Combine by doing nothing: because the two subarrays are already sorted, no work is needed to combine them. All elements in $A[p:q-1]$ are sorted and less than or equal to $A[q]$, and all elements in $A[q+1:r]$ are sorted and greater than or equal to the pivot $A[q]$. The entire subarray $A[p:r]$ cannot help but be sorted!

The QUICKSORT procedure implements quicksort. To sort an entire n-element array $A[1:n]$, the initial call is QUICKSORT$(A, 1, n)$.

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q - 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

Partitioning the array

The key to the algorithm is the PARTITION procedure on the next page, which rearranges the subarray $A[p:r]$ in place, returning the index of the dividing point between the two sides of the partition.

Figure 7.1 shows how PARTITION works on an 8-element array. PARTITION always selects the element $x = A[r]$ as the pivot. As the procedure runs, each element falls into exactly one of four regions, some of which may be empty. At the start of each iteration of the **for** loop in lines 3–6, the regions satisfy certain properties, shown in Figure 7.2. We state these properties as a loop invariant:

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                     // index of a new slot in the low side
6          exchange A[i] with A[j]   // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

At the beginning of each iteration of the loop of lines 3–6, for any array index k, the following conditions hold:

1. if $p \le k \le i$, then $A[k] \le x$ (the tan region of Figure 7.2);
2. if $i + 1 \le k \le j - 1$, then $A[k] > x$ (the blue region);
3. if $k = r$, then $A[k] = x$ (the yellow region).

We need to show that this loop invariant is true prior to the first iteration, that each iteration of the loop maintains the invariant, that the loop terminates, and that correctness follows from the invariant when the loop terminates.

Initialization: Prior to the first iteration of the loop, we have $i = p - 1$ and $j = p$. Because no values lie between p and i and no values lie between $i + 1$ and $j - 1$, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

Maintenance: As Figure 7.3 shows, we consider two cases, depending on the outcome of the test in line 4. Figure 7.3(a) shows what happens when $A[j] > x$: the only action in the loop is to increment j. After j has been incremented, the second condition holds for $A[j - 1]$ and all other entries remain unchanged. Figure 7.3(b) shows what happens when $A[j] \le x$: the loop increments i, swaps $A[i]$ and $A[j]$, and then increments j. Because of the swap, we now have that $A[i] \le x$, and condition 1 is satisfied. Similarly, we also have that $A[j - 1] > x$, since the item that was swapped into $A[j - 1]$ is, by the loop invariant, greater than x.

Termination: Since the loop makes exactly $r - p$ iterations, it terminates, whereupon $j = r$. At that point, the unexamined subarray $A[j : r - 1]$ is empty, and every entry in the array belongs to one of the other three sets described by the invariant. Thus, the values in the array have been partitioned into three sets: those less than or equal to x (the low side), those greater than x (the high side), and a singleton set containing x (the pivot).

Figure 7.1 The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Tan array elements all belong to the low side of the partition, with values at most x. Blue elements belong to the high side, with values greater than x. White elements have not yet been put into either side of the partition, and the yellow element is the pivot x. **(a)** The initial array and variable settings. None of the elements have been placed into either side of the partition. **(b)** The value 2 is "swapped with itself" and put into the low side. **(c)–(d)** The values 8 and 7 are placed into to high side. **(e)** The values 1 and 8 are swapped, and the low side grows. **(f)** The values 3 and 7 are swapped, and the low side grows. **(g)–(h)** The high side of the partition grows to include 5 and 6, and the loop terminates. **(i)** Line 7 swaps the pivot element so that it lies between the two sides of the partition, and line 8 returns the pivot's new index.

The final two lines of PARTITION finish up by swapping the pivot with the leftmost element greater than x, thereby moving the pivot into its correct place in the partitioned array, and then returning the pivot's new index. The output of PARTITION now satisfies the specifications given for the divide step. In fact, it satisfies a slightly stronger condition: after line 3 of QUICKSORT, $A[q]$ is strictly less than every element of $A[q+1:r]$.

Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p:r]$. The tan values in $A[p:i]$ are all less than or equal to x, the blue values in $A[i+1:j-1]$ are all greater than x, the white values in $A[j:r-1]$ have unknown relationships to x, and $A[r] = x$.

Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \le x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

Exercise 7.1-3 asks you to show that the running time of PARTITION on a subarray $A[p:r]$ of $n = r - p + 1$ elements is $\Theta(n)$.

Exercises

7.1-1

Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

7.1-2
What value of q does PARTITION return when all elements in the subarray $A[p:r]$ have the same value? Modify PARTITION so that $q = \lfloor (p+r)/2 \rfloor$ when all elements in the subarray $A[p:r]$ have the same value.

7.1-3
Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

7.1-4
Modify QUICKSORT to sort into monotonically decreasing order.

7.2 Performance of quicksort

The running time of quicksort depends on how balanced each partitioning is, which in turn depends on which elements are used as pivots. If the two sides of a partition are about the same size—the partitioning is balanced—then the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort. To allow you to gain some intuition before diving into a formal analysis, this section informally investigates how quicksort performs under the assumptions of balanced versus unbalanced partitioning.

But first, let's briefly look at the maximum amount of memory that quicksort requires. Although quicksort sorts in place according to the definition on page 158, the amount of memory it uses—aside from the array being sorted—is not constant. Since each recursive call requires a constant amount of space on the runtime stack, outside of the array being sorted, quicksort requires space proportional to the maximum depth of the recursion. As we'll see now, that could be as bad as $\Theta(n)$ in the worst case.

Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning produces one subproblem with $n-1$ elements and one with 0 elements. (See Section 7.4.1.) Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs $\Theta(n)$ time. Since the recursive call on an array of size 0 just returns without doing anything, $T(0) = \Theta(1)$, and the recurrence for the running time is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

By summing the costs incurred at each level of the recursion, we obtain an arithmetic series (equation (A.3) on page 1141), which evaluates to $\Theta(n^2)$. Indeed, the substitution method can be used to prove that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$. (See Exercise 7.2-1.)

Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$. The worst-case running time of quicksort is therefore no better than that of insertion sort. Moreover, the $\Theta(n^2)$ running time occurs when the input array is already completely sorted—a situation in which insertion sort runs in $O(n)$ time.

Best-case partitioning

In the most even possible split, PARTITION produces two subproblems, each of size no more than $n/2$, since one is of size $\lfloor (n-1)/2 \rfloor \le n/2$ and one of size $\lceil (n-1)/2 \rceil - 1 \le n/2$. In this case, quicksort runs much faster. An upper bound on the running time can then be described by the recurrence

$$T(n) = 2T(n/2) + \Theta(n) .$$

By case 2 of the master theorem (Theorem 4.1 on page 102), this recurrence has the solution $T(n) = \Theta(n \lg n)$. Thus, if the partitioning is equally balanced at every level of the recursion, an asymptotically faster algorithm results.

Balanced partitioning

As the analyses in Section 7.4 will show, the average-case running time of quicksort is much closer to the best case than to the worst case. By appreciating how the balance of the partitioning affects the recurrence describing the running time, we can gain an understanding of why.

Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the recurrence

$$T(n) = T(9n/10) + T(n/10) + \Theta(n) ,$$

on the running time of quicksort. Figure 7.4 shows the recursion tree for this recurrence, where for simplicity the $\Theta(n)$ driving function has been replaced by n, which won't affect the asymptotic solution of the recurrence (as Exercise 4.7-1 on page 118 justifies). Every level of the tree has cost n, until the recursion bottoms out in a base case at depth $\log_{10} n = \Theta(\lg n)$, and then the levels have cost

Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right.

at most n. The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$. Thus, with a 9-to-1 proportional split at every level of recursion, which intuitively seems highly unbalanced, quicksort runs in $O(n \lg n)$ time—asymptotically the same as if the split were right down the middle. Indeed, even a 99-to-1 split yields an $O(n \lg n)$ running time. In fact, any split of *constant* proportionality yields a recursion tree of depth $\Theta(\lg n)$, where the cost at each level is $O(n)$. The running time is therefore $O(n \lg n)$ whenever the split has constant proportionality. The ratio of the split affects only the constant hidden in the O-notation.

Intuition for the average case

To develop a clear notion of the expected behavior of quicksort, we must assume something about how its inputs are distributed. Because quicksort determines the sorted order using only comparisons between input elements, its behavior depends on the relative ordering of the values in the array elements given as the input, not on the particular values in the array. As in the probabilistic analysis of the hiring problem in Section 5.2, assume that all permutations of the input numbers are equally likely and that the elements are distinct.

When quicksort runs on a random input array, the partitioning is highly unlikely to happen in the same way at every level, as our informal analysis has assumed.

Figure 7.5 **(a)** Two levels of a recursion tree for quicksort. The partitioning at the root costs n and produces a "bad" split: two subarrays of sizes 0 and $n - 1$. The partitioning of the subarray of size $n - 1$ costs $n - 1$ and produces a "good" split: subarrays of size $(n - 1)/2 - 1$ and $(n - 1)/2$. **(b)** A single level of a recursion tree that is well balanced. In both parts, the partitioning cost for the subproblems shown with blue shading is $\Theta(n)$. Yet the subproblems remaining to be solved in (a), shown with tan shading, are no larger than the corresponding subproblems remaining to be solved in (b).

We expect that some of the splits will be reasonably well balanced and that some will be fairly unbalanced. For example, Exercise 7.2-6 asks you to show that about 80% of the time PARTITION produces a split that is at least as balanced as 9 to 1, and about 20% of the time it produces a split that is less balanced than 9 to 1.

In the average case, PARTITION produces a mix of "good" and "bad" splits. In a recursion tree for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree. Suppose for the sake of intuition that the good and bad splits alternate levels in the tree, and that the good splits are best-case splits and the bad splits are worst-case splits. Figure 7.5(a) shows the splits at two consecutive levels in the recursion tree. At the root of the tree, the cost is n for partitioning, and the subarrays produced have sizes $n - 1$ and 0: the worst case. At the next level, the subarray of size $n - 1$ undergoes best-case partitioning into subarrays of size $(n - 1)/2 - 1$ and $(n - 1)/2$. Let's assume that the base-case cost is 1 for the subarray of size 0.

The combination of the bad split followed by the good split produces three subarrays of sizes 0, $(n - 1)/2 - 1$, and $(n - 1)/2$ at a combined partitioning cost of $\Theta(n) + \Theta(n - 1) = \Theta(n)$. This situation is at most a constant factor worse than that in Figure 7.5(b), namely, where a single level of partitioning produces two subarrays of size $(n - 1)/2$, at a cost of $\Theta(n)$. Yet this latter situation is balanced! Intuitively, the $\Theta(n - 1)$ cost of the bad split in Figure 7.5(a) can be absorbed into the $\Theta(n)$ cost of the good split, and the resulting split is good. Thus, the running time of quicksort, when levels alternate between good and bad splits, is like the running time for good splits alone: still $O(n \lg n)$, but with a slightly larger constant hidden by the O-notation. We'll analyze the expected running time of a randomized version of quicksort rigorously in Section 7.4.2.

Exercises

7.2-1
Use the substitution method to prove that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$, as claimed at the beginning of Section 7.2.

7.2-2
What is the running time of QUICKSORT when all elements of array A have the same value?

7.2-3
Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

7.2-4
Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Explain persuasively why the procedure INSERTION-SORT might tend to beat the procedure QUICKSORT on this problem.

7.2-5
Suppose that the splits at every level of quicksort are in the constant proportion α to β, where $\alpha + \beta = 1$ and $0 < \alpha \le \beta < 1$. Show that the minimum depth of a leaf in the recursion tree is approximately $\log_{1/\alpha} n$ and that the maximum depth is approximately $\log_{1/\beta} n$. (Don't worry about integer round-off.)

7.2-6
Consider an array with distinct elements and for which all permutations of the elements are equally likely. Argue that for any constant $0 < \alpha \le 1/2$, the probability is approximately $1 - 2\alpha$ that PARTITION produces a split at least as balanced as $1 - \alpha$ to α.

7.3 A randomized version of quicksort

In exploring the average-case behavior of quicksort, we have assumed that all permutations of the input numbers are equally likely. This assumption does not always hold, however, as, for example, in the situation laid out in the premise for

Exercise 7.2-4. Section 5.3 showed that judicious randomization can sometimes be added to an algorithm to obtain good expected performance over all inputs. For quicksort, randomization yields a fast and practical algorithm. Many software libraries provide a randomized version of quicksort as their algorithm of choice for sorting large data sets.

In Section 5.3, the RANDOMIZED-HIRE-ASSISTANT procedure explicitly permutes its input and then runs the deterministic HIRE-ASSISTANT procedure. We could do the same for quicksort as well, but a different randomization technique yields a simpler analysis. Instead of always using $A[r]$ as the pivot, a randomized version randomly chooses the pivot from the subarray $A[p:r]$, where each element in $A[p:r]$ has an equal probability of being chosen. It then exchanges that element with $A[r]$ before partitioning. Because the pivot is chosen randomly, we expect the split of the input array to be reasonably well balanced on average.

The changes to PARTITION and QUICKSORT are small. The new partitioning procedure, RANDOMIZED-PARTITION, simply swaps before performing the partitioning. The new quicksort procedure, RANDOMIZED-QUICKSORT, calls RANDOMIZED-PARTITION instead of PARTITION. We'll analyze this algorithm in the next section.

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Exercises

7.3-1

Why do we analyze the expected running time of a randomized algorithm and not its worst-case running time?

7.3-2
When RANDOMIZED-QUICKSORT runs, how many calls are made to the random-number generator RANDOM in the worst case? How about in the best case? Give your answer in terms of Θ-notation.

7.4 Analysis of quicksort

Section 7.2 gave some intuition for the worst-case behavior of quicksort and for why we expect the algorithm to run quickly. This section analyzes the behavior of quicksort more rigorously. We begin with a worst-case analysis, which applies to either QUICKSORT or RANDOMIZED-QUICKSORT, and conclude with an analysis of the expected running time of RANDOMIZED-QUICKSORT.

7.4.1 Worst-case analysis

We saw in Section 7.2 that a worst-case split at every level of recursion in quicksort produces a $\Theta(n^2)$ running time, which, intuitively, is the worst-case running time of the algorithm. We now prove this assertion.

We'll use the substitution method (see Section 4.3) to show that the running time of quicksort is $O(n^2)$. Let $T(n)$ be the worst-case time for the procedure QUICKSORT on an input of size n. Because the procedure PARTITION produces two subproblems with total size $n-1$, we obtain the recurrence

$$T(n) = \max\{T(q) + T(n-1-q) : 0 \le q \le n-1\} + \Theta(n) , \tag{7.1}$$

We guess that $T(n) \le cn^2$ for some constant $c > 0$. Substituting this guess into recurrence (7.1) yields

$$\begin{aligned} T(n) &\le \max\left\{cq^2 + c(n-1-q)^2 : 0 \le q \le n-1\right\} + \Theta(n) \\ &= c \cdot \max\left\{q^2 + (n-1-q)^2 : 0 \le q \le n-1\right\} + \Theta(n) . \end{aligned}$$

Let's focus our attention on the maximization. For $q = 0, 1, \ldots, n-1$, we have

$$\begin{aligned} q^2 + (n-1-q)^2 &= q^2 + (n-1)^2 - 2q(n-1) + q^2 \\ &= (n-1)^2 + 2q(q-(n-1)) \\ &\le (n-1)^2 \end{aligned}$$

because $q \le n-1$ implies that $2q(q-(n-1)) \le 0$. Thus every term in the maximization is bounded by $(n-1)^2$.

Continuing with our analysis of $T(n)$, we obtain

$$\begin{aligned} T(n) &\le c(n-1)^2 + \Theta(n) \\ &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$$

by picking the constant c large enough that the $c(2n-1)$ term dominates the $\Theta(n)$ term. Thus $T(n) = O(n^2)$. Section 7.2 showed a specific case where quicksort takes $\Omega(n^2)$ time: when partitioning is maximally unbalanced. Thus, the worst-case running time of quicksort is $\Theta(n^2)$.

7.4.2 Expected running time

We have already seen the intuition behind why the expected running time of RANDOMIZED-QUICKSORT is $O(n \lg n)$: if, in each level of recursion, the split induced by RANDOMIZED-PARTITION puts any constant fraction of the elements on one side of the partition, then the recursion tree has depth $\Theta(\lg n)$ and $O(n)$ work is performed at each level. Even if we add a few new levels with the most unbalanced split possible between these levels, the total time remains $O(n \lg n)$. We can analyze the expected running time of RANDOMIZED-QUICKSORT precisely by first understanding how the partitioning procedure operates and then using this understanding to derive an $O(n \lg n)$ bound on the expected running time. This upper bound on the expected running time, combined with the $\Theta(n \lg n)$ best-case bound we saw in Section 7.2, yields a $\Theta(n \lg n)$ expected running time. We assume throughout that the values of the elements being sorted are distinct.

Running time and comparisons

The QUICKSORT and RANDOMIZED-QUICKSORT procedures differ only in how they select pivot elements. They are the same in all other respects. We can therefore analyze RANDOMIZED-QUICKSORT by considering the QUICKSORT and PARTITION procedures, but with the assumption that pivot elements are selected randomly from the subarray passed to RANDOMIZED-PARTITION. Let's start by relating the asymptotic running time of QUICKSORT to the number of times elements are compared (all in line 4 of PARTITION), understanding that this analysis also applies to RANDOMIZED-QUICKSORT. Note that we are counting the number of times that *array elements* are compared, not comparisons of indices.

Lemma 7.1
The running time of QUICKSORT on an n-element array is $O(n + X)$, where X is the number of element comparisons performed.

Proof The running time of QUICKSORT is dominated by the time spent in the PARTITION procedure. Each time PARTITION is called, it selects a pivot element, which is never included in any future recursive calls to QUICKSORT and PARTITION. Thus, there can be at most n calls to PARTITION over the entire execution of the quicksort algorithm. Each time QUICKSORT calls PARTITION, it also recursively calls itself twice, so there are at most $2n$ calls to the QUICKSORT procedure itself.

One call to PARTITION takes $O(1)$ time plus an amount of time that is proportional to the number of iterations of the **for** loop in lines 3–6. Each iteration of this **for** loop performs one comparison in line 4, comparing the pivot element to another element of the array A. Therefore, the total time spent in the **for** loop across all executions is proportional to X. Since there are at most n calls to PARTITION and the time spent outside the **for** loop is $O(1)$ for each call, the total time spent in PARTITION outside of the **for** loop is $O(n)$. Thus the total time for quicksort is $O(n + X)$. ■

Our goal for analyzing RANDOMIZED-QUICKSORT, therefore, is to compute the expected value $\mathrm{E}[X]$ of the random variable X denoting the total number of comparisons performed in all calls to PARTITION. To do so, we must understand when the quicksort algorithm compares two elements of the array and when it does not. For ease of analysis, let's index the elements of the array A by their position in the sorted output, rather than their position in the input. That is, although the elements in A may start out in any order, we'll refer to them by $z_1, z_2, \ldots, z_n$, where $z_1 < z_2 < \cdots < z_n$, with strict inequality because we assume that all elements are distinct. We denote the set $\{z_i, z_{i+1}, \ldots, z_j\}$ by Z_{ij}.

The next lemma characterizes when two elements are compared.

Lemma 7.2
During the execution of RANDOMIZED-QUICKSORT on an array of n distinct elements $z_1 < z_2 < \cdots < z_n$, an element z_i is compared with an element z_j, where $i < j$, if and only if one of them is chosen as a pivot before any other element in the set Z_{ij}. Moreover, no two elements are ever compared twice.

Proof Let's look at the first time that an element $x \in Z_{ij}$ is chosen as a pivot during the execution of the algorithm. There are three cases to consider. If x is neither z_i nor z_j—that is, $z_i < x < z_j$—then z_i and z_j are not compared at any subsequent time, because they fall into different sides of the partition around x. If $x = z_i$, then PARTITION compares z_i with every other item in Z_{ij}. Similarly, if $x = z_j$, then PARTITION compares z_j with every other item in Z_{ij}. Thus, z_i and z_j are compared if and only if the first element to be chosen as a pivot from Z_{ij} is either z_i or z_j. In the latter two cases, where one of z_i and z_j is chosen

as a pivot, since the pivot is removed from future comparisons, it is never compared again with the other element. ■

As an example of this lemma, consider an input to quicksort of the numbers 1 through 10 in some arbitrary order. Suppose that the first pivot element is 7. Then the first call to PARTITION separates the numbers into two sets: $\{1, 2, 3, 4, 5, 6\}$ and $\{8, 9, 10\}$. In the process, the pivot element 7 is compared with all other elements, but no number from the first set (e.g., 2) is or ever will be compared with any number from the second set (e.g., 9). The values 7 and 9 are compared because 7 is the first item from $Z_{7,9}$ to be chosen as a pivot. In contrast, 2 and 9 are never compared because the first pivot element chosen from $Z_{2,9}$ is 7.

The next lemma gives the probability that two elements are compared.

Lemma 7.3
Consider an execution of the procedure RANDOMIZED-QUICKSORT on an array of n distinct elements $z_1 < z_2 < \cdots < z_n$. Given two arbitrary elements z_i and z_j where $i < j$, the probability that they are compared is $2/(j - i + 1)$.

Proof Let's look at the tree of recursive calls that RANDOMIZED-QUICKSORT makes, and consider the sets of elements provided as input to each call. Initially, the root set contains all the elements of Z_{ij}, since the root set contains every element in A. The elements belonging to Z_{ij} all stay together for each recursive call of RANDOMIZED-QUICKSORT until PARTITION chooses some element $x \in Z_{ij}$ as a pivot. From that point on, the pivot x appears in no subsequent input set. The first time that RANDOMIZED-SELECT chooses a pivot $x \in Z_{ij}$ from a set containing all the elements of Z_{ij}, each element in Z_{ij} is equally likely to be x because the pivot is chosen uniformly at random. Since $|Z_{ij}| = j - i + 1$, the probability is $1/(j - i + 1)$ that any given element in Z_{ij} is the first pivot chosen from Z_{ij}. Thus, by Lemma 7.2, we have

$$\begin{aligned}
\Pr\{z_i \text{ is compared with } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} \\
&\qquad + \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&= \frac{2}{j - i + 1} ,
\end{aligned}$$

where the second line follows from the first because the two events are mutually exclusive. ■

We can now complete the analysis of randomized quicksort.

Theorem 7.4
The expected running time of RANDOMIZED-QUICKSORT on an input of n distinct elements is $O(n \lg n)$.

Proof The analysis uses indicator random variables (see Section 5.2). Let the n distinct elements be $z_1 < z_2 < \cdots < z_n$, and for $1 \le i < j \le n$, define the indicator random variable $X_{ij} = \mathrm{I}\{z_i \text{ is compared with } z_j\}$. From Lemma 7.2, each pair is compared at most once, and so we can express X as follows:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \ .$$

By taking expectations of both sides and using linearity of expectation (equation (C.24) on page 1192) and Lemma 5.1 on page 130, we obtain

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared with } z_j\} && \text{(by Lemma 5.1)} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} && \text{(by Lemma 7.3)} \ .
\end{aligned}$$

We can evaluate this sum using a change of variables ($k = j - i$) and the bound on the harmonic series in equation (A.9) on page 1142:

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) \ .
\end{aligned}$$

This bound and Lemma 7.1 allow us to conclude that the expected running time of RANDOMIZED-QUICKSORT is $O(n \lg n)$ (assuming that the element values are distinct). ■

Exercises

7.4-1
Show that the recurrence

$$T(n) = \max\{T(q) + T(n-q-1) : 0 \le q \le n-1\} + \Theta(n)$$

has a lower bound of $T(n) = \Omega(n^2)$.

7.4-2
Show that quicksort's best-case running time is $\Omega(n \lg n)$.

7.4-3
Show that the expression $q^2 + (n-q-1)^2$ achieves its maximum value over $q = 0, 1, \ldots, n-1$ when $q = 0$ or $q = n-1$.

7.4-4
Show that RANDOMIZED-QUICKSORT's expected running time is $\Omega(n \lg n)$.

7.4-5
Coarsening the recursion, as we did in Problem 2-1 for merge sort, is a common way to improve the running time of quicksort in practice. We modify the base case of the recursion so that if the array has fewer than k elements, the subarray is sorted by insertion sort, rather than by continued recursive calls to quicksort. Argue that the randomized version of this sorting algorithm runs in $O(nk + n \lg(n/k))$ expected time. How should you pick k, both in theory and in practice?

★ ***7.4-6***
Consider modifying the PARTITION procedure by randomly picking three elements from subarray $A[p:r]$ and partitioning about their median (the middle value of the three elements). Approximate the probability of getting worse than an α-to-$(1-\alpha)$ split, as a function of α in the range $0 < \alpha < 1/2$.

Problems

7-1 Hoare partition correctness

The version of PARTITION given in this chapter is not the original partitioning algorithm. Here is the original partitioning algorithm, which is due to C. A. R. Hoare.

```
HOARE-PARTITION(A, p, r)
1  x = A[p]
2  i = p - 1
3  j = r + 1
4  while TRUE
5      repeat
6          j = j - 1
7      until A[j] ≤ x
8      repeat
9          i = i + 1
10     until A[i] ≥ x
11     if i < j
12         exchange A[i] with A[j]
13     else return j
```

a. Demonstrate the operation of HOARE-PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, showing the values of the array and the indices i and j after each iteration of the **while** loop in lines 4–13.

b. Describe how the PARTITION procedure in Section 7.1 differs from HOARE-PARTITION when all elements in $A[p:r]$ are equal. Describe a practical advantage of HOARE-PARTITION over PARTITION for use in quicksort.

The next three questions ask you to give a careful argument that the procedure HOARE-PARTITION is correct. Assuming that the subarray $A[p:r]$ contains at least two elements, prove the following:

c. The indices i and j are such that the procedure never accesses an element of A outside the subarray $A[p:r]$.

d. When HOARE-PARTITION terminates, it returns a value j such that $p \le j < r$.

e. Every element of $A[p:j]$ is less than or equal to every element of $A[j+1:r]$ when HOARE-PARTITION terminates.

The PARTITION procedure in Section 7.1 separates the pivot value (originally in $A[r]$) from the two partitions it forms. The HOARE-PARTITION procedure, on the other hand, always places the pivot value (originally in $A[p]$) into one of the two partitions $A[p : j]$ and $A[j + 1 : r]$. Since $p \leq j < r$, neither partition is empty.

f. Rewrite the QUICKSORT procedure to use HOARE-PARTITION.

7-2 Quicksort with equal element values

The analysis of the expected running time of randomized quicksort in Section 7.4.2 assumes that all element values are distinct. This problem examines what happens when they are not.

a. Suppose that all element values are equal. What is randomized quicksort's running time in this case?

b. The PARTITION procedure returns an index q such that each element of $A[p : q - 1]$ is less than or equal to $A[q]$ and each element of $A[q + 1 : r]$ is greater than $A[q]$. Modify the PARTITION procedure to produce a procedure PARTITION$'(A, p, r)$, which permutes the elements of $A[p : r]$ and returns two indices q and t, where $p \leq q \leq t \leq r$, such that

- all elements of $A[q : t]$ are equal,
- each element of $A[p : q - 1]$ is less than $A[q]$, and
- each element of $A[t + 1 : r]$ is greater than $A[q]$.

Like PARTITION, your PARTITION$'$ procedure should take $\Theta(r - p)$ time.

c. Modify the RANDOMIZED-PARTITION procedure to call PARTITION$'$, and name the new procedure RANDOMIZED-PARTITION$'$. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT$'(A, p, r)$ that calls RANDOMIZED-PARTITION$'$ and recurses only on partitions where elements are not known to be equal to each other.

d. Using QUICKSORT$'$, adjust the analysis in Section 7.4.2 to avoid the assumption that all elements are distinct.

7-3 Alternative quicksort analysis

An alternative analysis of the running time of randomized quicksort focuses on the expected running time of each individual recursive call to RANDOMIZED-QUICKSORT, rather than on the number of comparisons performed. As in the analysis of Section 7.4.2, assume that the values of the elements are distinct.

a. Argue that, given an array of size n, the probability that any particular element is chosen as the pivot is $1/n$. Use this probability to define indicator random variables $X_i = \mathrm{I}\{i\text{th smallest element is chosen as the pivot}\}$. What is $\mathrm{E}[X_i]$?

b. Let $T(n)$ be a random variable denoting the running time of quicksort on an array of size n. Argue that

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q \left(T(q-1) + T(n-q) + \Theta(n)\right)\right] . \tag{7.2}$$

c. Show how to rewrite equation (7.2) as

$$\mathrm{E}[T(n)] = \frac{2}{n}\sum_{q=1}^{n-1} \mathrm{E}[T(q)] + \Theta(n) . \tag{7.3}$$

d. Show that

$$\sum_{q=1}^{n-1} q \lg q \le \frac{n^2}{2}\lg n - \frac{n^2}{8} \tag{7.4}$$

for $n \ge 2$. (*Hint:* Split the summation into two parts, one summation for $q = 1, 2, \ldots, \lceil n/2 \rceil - 1$ and one summation for $q = \lceil n/2 \rceil, \ldots, n-1$.)

e. Using the bound from equation (7.4), show that the recurrence in equation (7.3) has the solution $\mathrm{E}[T(n)] = O(n \lg n)$. (*Hint:* Show, by substitution, that $\mathrm{E}[T(n)] \le an \lg n$ for sufficiently large n and for some positive constant a.)

7-4 Stooge sort

Professors Howard, Fine, and Howard have proposed a deceptively simple sorting algorithm, named stooge sort in their honor, appearing on the following page.

a. Argue that the call STOOGE-SORT$(A, 1, n)$ correctly sorts the array $A[1:n]$.

b. Give a recurrence for the worst-case running time of STOOGE-SORT and a tight asymptotic (Θ-notation) bound on the worst-case running time.

c. Compare the worst-case running time of STOOGE-SORT with that of insertion sort, merge sort, heapsort, and quicksort. Do the professors deserve tenure?

```
STOOGE-SORT(A, p, r)
if A[p] > A[r]
    exchange A[p] with A[r]
if p + 1 < r
    k = ⌊(r − p + 1)/3⌋          // round down
    STOOGE-SORT(A, p, r − k)     // first two-thirds
    STOOGE-SORT(A, p + k, r)     // last two-thirds
    STOOGE-SORT(A, p, r − k)     // first two-thirds again
```

7-5 Stack depth for quicksort

The QUICKSORT procedure of Section 7.1 makes two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the low side of the partition and then it recursively sorts the high side of the partition. The second recursive call in QUICKSORT is not really necessary, because the procedure can instead use an iterative control structure. This transformation technique, called ***tail-recursion elimination***, is provided automatically by good compilers. Applying tail-recursion elimination transforms QUICKSORT into the TRE-QUICKSORT procedure.

```
TRE-QUICKSORT(A, p, r)
while p < r
    // Partition and then sort the low side.
    q = PARTITION(A, p, r)
    TRE-QUICKSORT(A, p, q − 1)
    p = q + 1
```

a. Argue that TRE-QUICKSORT$(A, 1, n)$ correctly sorts the array $A[1:n]$.

Compilers usually execute recursive procedures by using a ***stack*** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. When a procedure is called, its information is ***pushed*** onto the stack, and when it terminates, its information is ***popped***. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The ***stack depth*** is the maximum amount of stack space used at any time during a computation.

b. Describe a scenario in which TRE-QUICKSORT's stack depth is $\Theta(n)$ on an n-element input array.

c. Modify TRE-QUICKSORT so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

7-6 Median-of-3 partition

One way to improve the RANDOMIZED-QUICKSORT procedure is to partition around a pivot that is chosen more carefully than by picking a random element from the subarray. A common approach is the ***median-of-3*** method: choose the pivot as the median (middle element) of a set of 3 elements randomly selected from the subarray. (See Exercise 7.4-6.) For this problem, assume that the n elements in the input subarray $A[p:r]$ are distinct and that $n \geq 3$. Denote the sorted version of $A[p:r]$ by $z_1, z_2, \ldots, z_n$. Using the median-of-3 method to choose the pivot element x, define $p_i = \Pr\{x = z_i\}$.

a. Give an exact formula for p_i as a function of n and i for $i = 2, 3, \ldots, n-1$. (Observe that $p_1 = p_n = 0$.)

b. By what amount does the median-of-3 method increase the likelihood of choosing the pivot to be $x = z_{\lfloor (n+1)/2 \rfloor}$, the median of $A[p:r]$, compared with the ordinary implementation? Assume that $n \to \infty$, and give the limiting ratio of these probabilities.

c. Suppose that we define a "good" split to mean choosing the pivot as $x = z_i$, where $n/3 \leq i \leq 2n/3$. By what amount does the median-of-3 method increase the likelihood of getting a good split compared with the ordinary implementation? (*Hint:* Approximate the sum by an integral.)

d. Argue that in the $\Omega(n \lg n)$ running time of quicksort, the median-of-3 method affects only the constant factor.

7-7 Fuzzy sorting of intervals

Consider a sorting problem in which you do not know the numbers exactly. Instead, for each number, you know an interval on the real line to which it belongs. That is, you are given n closed intervals of the form $[a_i, b_i]$, where $a_i \leq b_i$. The goal is to ***fuzzy-sort*** these intervals: to produce a permutation $\langle i_1, i_2, \ldots, i_n \rangle$ of the intervals such that for $j = 1, 2, \ldots, n$, there exist $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \leq c_2 \leq \cdots \leq c_n$.

a. Design a randomized algorithm for fuzzy-sorting n intervals. Your algorithm should have the general structure of an algorithm that quicksorts the left endpoints (the a_i values), but it should take advantage of overlapping intervals to improve the running time. (As the intervals overlap more and more, the prob-

lem of fuzzy-sorting the intervals becomes progressively easier. Your algorithm should take advantage of such overlapping, to the extent that it exists.)

b. Argue that your algorithm runs in $\Theta(n \lg n)$ expected time in general, but runs in $\Theta(n)$ expected time when all of the intervals overlap (i.e., when there exists a value x such that $x \in [a_i, b_i]$ for all i). Your algorithm should not be checking for this case explicitly, but rather, its performance should naturally improve as the amount of overlap increases.

Chapter notes

Quicksort was invented by Hoare [219], and his version of PARTITION appears in Problem 7-1. Bentley [51, p. 117] attributes the PARTITION procedure given in Section 7.1 to N. Lomuto. The analysis in Section 7.4 based on an analysis due to Motwani and Raghavan [336]. Sedgewick [401] and Bentley [51] provide good references on the details of implementation and how they matter.

McIlroy [323] shows how to engineer a "killer adversary" that produces an array on which virtually any implementation of quicksort takes $\Theta(n^2)$ time.

8 Sorting in Linear Time

We have now seen a handful of algorithms that can sort n numbers in $O(n \lg n)$ time. Whereas merge sort and heapsort achieve this upper bound in the worst case, quicksort achieves it on average. Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \lg n)$ time.

These algorithms share an interesting property: *the sorted order they determine is based only on comparisons between the input elements*. We call such sorting algorithms ***comparison sorts***. All the sorting algorithms introduced thus far are comparison sorts.

In Section 8.1, we'll prove that any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Sections 8.2, 8.3, and 8.4 examine three sorting algorithms—counting sort, radix sort, and bucket sort—that run in linear time on certain types of inputs. Of course, these algorithms use operations other than comparisons to determine the sorted order. Consequently, the $\Omega(n \lg n)$ lower bound does not apply to them.

8.1 Lower bounds for sorting

A comparison sort uses only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$. That is, given two elements a_i and a_j, it performs one of the tests $a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, or $a_i > a_j$ to determine their relative order. It may not inspect the values of the elements or gain order information about them in any other way.

Since we are proving a lower bound, we assume without loss of generality in this section that all the input elements are distinct. After all, a lower bound for distinct elements applies when elements may or may not be distinct. Consequently,

Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node (shown in blue) annotated by $i:j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The highlighted path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$. Going left from the root node, labeled 1:2, indicates that $a_1 \le a_2$. Going right from the node labeled 2:3 indicates that $a_2 > a_3$. Going right from the node labeled 1:3 indicates that $a_1 > a_3$. Therefore, we have the ordering $a_3 \le a_1 \le a_2$, as indicated in the leaf labeled $\langle 3, 1, 2 \rangle$. Because the three input elements have $3! = 6$ possible permutations, the decision tree must have at least 6 leaves.

comparisons of the form $a_i = a_j$ are useless, which means that we can assume that no comparisons for exact equality occur. Moreover, the comparisons $a_i \le a_j$, $a_i \ge a_j$, $a_i > a_j$, and $a_i < a_j$ are all equivalent in that they yield identical information about the relative order of a_i and a_j. We therefore assume that all comparisons have the form $a_i \le a_j$.

The decision-tree model

We can view comparison sorts abstractly in terms of decision trees. A ***decision tree*** is a full binary tree (each node is either a leaf or has both children) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. Figure 8.1 shows the decision tree corresponding to the insertion sort algorithm from Section 2.1 operating on an input sequence of three elements.

A decision tree has each internal node annotated by $i:j$ for some i and j in the range $1 \le i, j \le n$, where n is the number of elements in the input sequence. We also annotate each leaf by a permutation $\langle \pi(1), \pi(2), \pi(n) \rangle$. (See Section C.1 for background on permutations.) Indices in the internal nodes and the leaves always refer to the original positions of the array elements at the start of the sorting algorithm. The execution of the comparison sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf. Each internal node indicates a comparison $a_i \le a_j$. The left subtree then dictates sub-

sequent comparisons once we know that $a_i \le a_j$, and the right subtree dictates subsequent comparisons when $a_i > a_j$. Arriving at a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as at least one of the leaves of the decision tree for a comparison sort to be correct. Furthermore, each of these leaves must be reachable from the root by a downward path corresponding to an actual execution of the comparison sort. (We call such leaves "reachable.") Thus, we consider only decision trees in which each permutation appears as a reachable leaf.

A lower bound for the worst case

The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree. A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm. The following theorem establishes such a lower bound.

Theorem 8.1
Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as one or more leaves, we have $n! \le l$. Since a binary tree of height h has no more than 2^h leaves, we have

$$n! \le l \le 2^h ,$$

which, by taking logarithms, implies

$$\begin{aligned} h &\ge \lg(n!) && \text{(since the lg function is monotonically increasing)} \\ &= \Omega(n \lg n) && \text{(by equation (3.28) on page 67)} . \end{aligned}$$

■

Corollary 8.2
Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof The $O(n \lg n)$ upper bounds on the running times for heapsort and merge sort match the $\Omega(n \lg n)$ worst-case lower bound from Theorem 8.1. ■

Exercises

8.1-1
What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

8.1-2
Obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation. Instead, evaluate the summation $\sum_{k=1}^{n} \lg k$ using techniques from Section A.2.

8.1-3
Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

8.1-4
You are given an n-element input sequence, and you know in advance that it is partly sorted in the following sense. Each element initially in position i such that $i \bmod 4 = 0$ is either already in its correct position, or it is one place away from its correct position. For example, you know that after sorting, the element initially in position 12 belongs in position 11, 12, or 13. You have no advance information about the other elements, in positions i where $i \bmod 4 \neq 0$. Show that an $\Omega(n \lg n)$ lower bound on comparison-based sorting still holds in this case.

8.2 Counting sort

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. It runs in $\Theta(n + k)$ time, so that when $k = O(n)$, counting sort runs in $\Theta(n)$ time.

Counting sort first determines, for each input element x, the number of elements less than or equal to x. It then uses this information to place element x directly into its position in the output array. For example, if 17 elements are less than or equal to x, then x belongs in output position 17. We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want them all to end up in the same position.

The COUNTING-SORT procedure on the facing page takes as input an array $A[1:n]$, the size n of this array, and the limit k on the nonnegative integer values in A. It returns its sorted output in the array $B[1:n]$ and uses an array $C[0:k]$ for temporary working storage.

```
COUNTING-SORT(A, n, k)
1  let B[1 : n] and C[0 : k] be new arrays
2  for i = 0 to k
3      C[i] = 0
4  for j = 1 to n
5      C[A[j]] = C[A[j]] + 1
6  // C[i] now contains the number of elements equal to i.
7  for i = 1 to k
8      C[i] = C[i] + C[i − 1]
9  // C[i] now contains the number of elements less than or equal to i.
10 // Copy A to B, starting from the end of A.
11 for j = n downto 1
12     B[C[A[j]]] = A[j]
13     C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14 return B
```

Figure 8.2 illustrates counting sort. After the **for** loop of lines 2–3 initializes the array C to all zeros, the **for** loop of lines 4–5 makes a pass over the array A to inspect each input element. Each time it finds an input element whose value is i, it increments $C[i]$. Thus, after line 5, $C[i]$ holds the number of input elements equal to i for each integer $i = 0, 1, \ldots, k$. Lines 7–8 determine for each $i = 0, 1, \ldots, k$ how many input elements are less than or equal to i by keeping a running sum of the array C.

Finally, the **for** loop of lines 11–13 makes another pass over A, but in reverse, to place each element $A[j]$ into its correct sorted position in the output array B. If all n elements are distinct, then when line 11 is first entered, for each $A[j]$, the value $C[A[j]]$ is the correct final position of $A[j]$ in the output array, since there are $C[A[j]]$ elements less than or equal to $A[j]$. Because the elements might not be distinct, the loop decrements $C[A[j]]$ each time it places a value $A[j]$ into B. Decrementing $C[A[j]]$ causes the previous element in A with a value equal to $A[j]$, if one exists, to go to the position immediately before $A[j]$ in the output array B.

How much time does counting sort require? The **for** loop of lines 2–3 takes $\Theta(k)$ time, the **for** loop of lines 4–5 takes $\Theta(n)$ time, the **for** loop of lines 7–8 takes $\Theta(k)$ time, and the **for** loop of lines 11–13 takes $\Theta(n)$ time. Thus, the overall time is $\Theta(k + n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.

Counting sort can beat the lower bound of $\Omega(n \lg n)$ proved in Section 8.1 because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1:8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 11–13, respectively. Only the tan elements of array B have been filled in. **(f)** The final sorted output array B.

elements to index into an array. The $\Omega(n \lg n)$ lower bound for sorting does not apply when we depart from the comparison sort model.

An important property of counting sort is that it is ***stable***: elements with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two elements by the rule that whichever element appears first in the input array appears first in the output array. Normally, the property of stability is important only when satellite data are carried around with the element being sorted. Counting sort's stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

Exercises

8.2-1

Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the array $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

8.2-2

Prove that COUNTING-SORT is stable.

8.2-3
Suppose that we were to rewrite the **for** loop header in line 11 of the COUNTING-SORT as

```
11  for j = 1 to n
```

Show that the algorithm still works properly, but that it is not stable. Then rewrite the pseudocode for counting sort so that elements with the same value are written into the output array in order of increasing index and the algorithm is stable.

8.2-4
Prove the following loop invariant for COUNTING-SORT:

> At the start of each iteration of the **for** loop of lines 11–13, the last element in A with value i that has not yet been copied into B belongs in $B[C[i]]$.

8.2-5
Suppose that the array being sorted contains only integers in the range 0 to k and that there are no satellite data to move with those keys. Modify counting sort to use just the arrays A and C, putting the sorted result back into array A instead of into a new array B.

8.2-6
Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a : b]$ in $O(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

8.2-7
Counting sort can also work efficiently if the input values have fractional parts, but the number of digits in the fractional part is small. Suppose that you are given n numbers in the range 0 to k, each with at most d decimal (base 10) digits to the right of the decimal point. Modify counting sort to run in $\Theta(n + 10^d k)$ time.

8.3 Radix sort

Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums. The cards have 80 columns, and in each column a machine can punch a hole in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one

329	720	720	329
457	355	329	355
657	436	436	436
839 →	457 →	839 →	457
436	657	355	657
720	329	457	720
355	839	657	839

Figure 8.3 The operation of radix sort on seven 3-digit numbers. The leftmost column is the input. The remaining columns show the numbers after successive sorts on increasingly significant digit positions. Tan shading indicates the digit position sorted on to produce each list from the previous one.

of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

For decimal digits, each column uses only 10 places. (The other two places are reserved for encoding nonnumeric characters.) A d-digit number occupies a field of d columns. Since the card sorter can look at only one column at a time, the problem of sorting n cards on a d-digit number requires a sorting algorithm.

Intuitively, you might sort numbers on their *most significant* (leftmost) digit, sort each of the resulting bins recursively, and then combine the decks in order. Unfortunately, since the cards in 9 of the 10 bins must be put aside to sort each of the bins, this procedure generates many intermediate piles of cards that you would have to keep track of. (See Exercise 8.3-6.)

Radix sort solves the problem of card sorting—counterintuitively—by sorting on the *least significant* digit first. The algorithm then combines the cards into a single deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the cards in the 2 bin, and so on. Then it sorts the entire deck again on the second-least significant digit and recombines the deck in a like manner. The process continues until the cards have been sorted on all d digits. Remarkably, at that point the cards are fully sorted on the d-digit number. Thus, only d passes through the deck are required to sort. Figure 8.3 shows how radix sort operates on a "deck" of seven 3-digit numbers.

In order for radix sort to work correctly, the digit sorts must be stable. The sort performed by a card sorter is stable, but the operator must be careful not to change the order of the cards as they come out of a bin, even though all the cards in a bin have the same digit in the chosen column.

In a typical computer, which is a sequential random-access machine, we sometimes use radix sort to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates,

compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day (the "least significant" part), next on month, and finally on year.

The code for radix sort is straightforward. The RADIX-SORT procedure assumes that each element in array $A[1:n]$ has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

```
RADIX-SORT(A, n, d)
for i = 1 to d
    use a stable sort to sort array A[1:n] on digit i
```

Although the pseudocode for RADIX-SORT does not specify which stable sort to use, COUNTING-SORT is commonly used. If you use COUNTING-SORT as the stable sort, you can make RADIX-SORT a little more efficient by revising COUNTING-SORT to take a pointer to the output array as a parameter, having RADIX-SORT preallocate this array, and alternating input and output between the two arrays in successive iterations of the **for** loop in RADIX-SORT.

Lemma 8.3
Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n+k))$ time if the stable sort it uses takes $\Theta(n+k)$ time.

Proof The correctness of radix sort follows by induction on the column being sorted (see Exercise 8.3-3). The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit lies in the range 0 to $k-1$ (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes $\Theta(n+k)$ time. There are d passes, and so the total time for radix sort is $\Theta(d(n+k))$. ■

When d is constant and $k = O(n)$, we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits.

Lemma 8.4
Given n b-bit numbers and any positive integer $r \le b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n+2^r))$ time if the stable sort it uses takes $\Theta(n+k)$ time for inputs in the range 0 to k.

Proof For a value $r \le b$, view each key as having $d = \lceil b/r \rceil$ digits of r bits each. Each digit is an integer in the range 0 to $2^r - 1$, so that we can use counting sort with $k = 2^r - 1$. (For example, we can view a 32-bit word as having four 8-bit digits, so that $b = 32$, $r = 8$, $k = 2^r - 1 = 255$, and $d = b/r = 4$.) Each pass of counting sort takes $\Theta(n + k) = \Theta(n + 2^r)$ time and there are d passes, for a total running time of $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. ■

Given n and b, what value of $r \le b$ minimizes the expression $(b/r)(n + 2^r)$? As r decreases, the factor b/r increases, but as r increases so does 2^r. The answer depends on whether $b < \lfloor \lg n \rfloor$. If $b < \lfloor \lg n \rfloor$, then $r \le b$ implies $(n+2^r) = \Theta(n)$. Thus, choosing $r = b$ yields a running time of $(b/b)(n + 2^b) = \Theta(n)$, which is asymptotically optimal. If $b \ge \lfloor \lg n \rfloor$, then choosing $r = \lfloor \lg n \rfloor$ gives the best running time to within a constant factor, which we can see as follows.[1] Choosing $r = \lfloor \lg n \rfloor$ yields a running time of $\Theta(bn/\lg n)$. As r increases above $\lfloor \lg n \rfloor$, the 2^r term in the numerator increases faster than the r term in the denominator, and so increasing r above $\lfloor \lg n \rfloor$ yields a running time of $\Omega(bn/\lg n)$. If instead r were to decrease below $\lfloor \lg n \rfloor$, then the b/r term increases and the $n + 2^r$ term remains at $\Theta(n)$.

Is radix sort preferable to a comparison-based sorting algorithm, such as quicksort? If $b = O(\lg n)$, as is often the case, and $r \approx \lg n$, then radix sort's running time is $\Theta(n)$, which appears to be better than quicksort's expected running time of $\Theta(n \lg n)$. The constant factors hidden in the Θ-notation differ, however. Although radix sort may make fewer passes than quicksort over the n keys, each pass of radix sort may take significantly longer. Which sorting algorithm to prefer depends on the characteristics of the implementations, of the underlying machine (e.g., quicksort often uses hardware caches more effectively than radix sort), and of the input data. Moreover, the version of radix sort that uses counting sort as the intermediate stable sort does not sort in place, which many of the $\Theta(n \lg n)$-time comparison sorts do. Thus, when primary memory storage is at a premium, an in-place algorithm such as quicksort could be the better choice.

Exercises

8.3-1

Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

---

[1] The choice of $r = \lfloor \lg n \rfloor$ assumes that $n > 1$. If $n \le 1$, there is nothing to sort.

8.3-2
Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any comparison sort stable. How much additional time and space does your scheme entail?

8.3-3
Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

8.3-4
Suppose that COUNTING-SORT is used as the stable sort within RADIX-SORT. If RADIX-SORT calls COUNTING-SORT d times, then since each call of COUNTING-SORT makes two passes over the data (lines 4–5 and 11–13), altogether $2d$ passes over the data occur. Describe how to reduce the total number of passes to $d + 1$.

8.3-5
Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

★ ***8.3-6***
In the first card-sorting algorithm in this section, which sorts on the most significant digit first, exactly how many sorting passes are needed to sort d-digit decimal numbers in the worst case? How many piles of cards does an operator need to keep track of in the worst case?

8.4 Bucket sort

Bucket sort assumes that the input is drawn from a uniform distribution and has an average-case running time of $O(n)$. Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval $[0, 1)$. (See Section C.2 for a definition of a uniform distribution.)

Bucket sort divides the interval $[0, 1)$ into n equal-sized subintervals, or ***buckets***, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over $[0, 1)$, we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

The BUCKET-SORT procedure on the next page assumes that the input is an array $A[1:n]$ and that each element $A[i]$ in the array satisfies $0 \le A[i] < 1$. The code requires an auxiliary array $B[0:n-1]$ of linked lists (buckets) and assumes

Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1:10]$. **(b)** The array $B[0:9]$ of sorted lists (buckets) after line 7 of the algorithm, with slashes indicating the end of each bucket. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation of the lists $B[0], B[1], \ldots, B[9]$ in order.

that there is a mechanism for maintaining such lists. (Section 10.2 describes how to implement basic operations on linked lists.) Figure 8.4 shows the operation of bucket sort on an input array of 10 numbers.

```
BUCKET-SORT(A, n)
1  let B[0 : n − 1] be a new array
2  for i = 0 to n − 1
3      make B[i] an empty list
4  for i = 1 to n
5      insert A[i] into list B[⌊n · A[i]⌋]
6  for i = 0 to n − 1
7      sort list B[i] with insertion sort
8  concatenate the lists B[0], B[1], ..., B[n − 1] together in order
9  return the concatenated lists
```

To see that this algorithm works, consider two elements $A[i]$ and $A[j]$. Assume without loss of generality that $A[i] \le A[j]$. Since $\lfloor n \cdot A[i] \rfloor \le \lfloor n \cdot A[j] \rfloor$, either element $A[i]$ goes into the same bucket as $A[j]$ or it goes into a bucket with a lower index. If $A[i]$ and $A[j]$ go into the same bucket, then the **for** loop of lines 6–7 puts them into the proper order. If $A[i]$ and $A[j]$ go into different buckets, then line 8 puts them into the proper order. Therefore, bucket sort works correctly.

To analyze the running time, observe that, together, all lines except line 7 take $O(n)$ time in the worst case. We need to analyze the total time taken by the n calls to insertion sort in line 7.

To analyze the cost of the calls to insertion sort, let n_i be the random variable denoting the number of elements placed in bucket $B[i]$. Since insertion sort runs in quadratic time (see Section 2.2), the running time of bucket sort is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \; . \tag{8.1}$$

We now analyze the average-case running time of bucket sort, by computing the expected value of the running time, where we take the expectation over the input distribution. Taking expectations of both sides and using linearity of expectation (equation (C.24) on page 1192), we have

$$\begin{aligned} \mathrm{E}\left[T(n)\right] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \quad \text{(by linearity of expectation)} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right) \quad \text{(by equation (C.25) on page 1193)} \; . \end{aligned} \tag{8.2}$$

We claim that

$$\mathrm{E}\left[n_i^2\right] = 2 - 1/n \tag{8.3}$$

for $i = 0, 1, \ldots, n-1$. It is no surprise that each bucket i has the same value of $\mathrm{E}\left[n_i^2\right]$, since each value in the input array A is equally likely to fall in any bucket.

To prove equation (8.3), view each random variable n_i as the number of successes in n Bernoulli trials (see Section C.4). Success in a trial occurs when an element goes into bucket $B[i]$, with a probability $p = 1/n$ of success and $q = 1 - 1/n$ of failure. A binomial distribution counts n_i, the number of successes, in the n trials. By equations (C.41) and (C.44) on pages 1199–1200, we have $\mathrm{E}\left[n_i\right] = np = n(1/n) = 1$ and $\mathrm{Var}\left[n_i\right] = npq = 1 - 1/n$. Equation (C.31) on page 1194 gives

$$\begin{aligned} \mathrm{E}\left[n_i^2\right] &= \mathrm{Var}\left[n_i\right] + \mathrm{E}^2\left[n_i\right] \\ &= (1 - 1/n) + 1^2 \\ &= 2 - 1/n \; , \end{aligned}$$

which proves equation (8.3). Using this expected value in equation (8.2), we get that the average-case running time for bucket sort is $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$.

Even if the input is not drawn from a uniform distribution, bucket sort may still run in linear time. As long as the input has the property that the sum of the squares of the bucket sizes is linear in the total number of elements, equation (8.1) tells us that bucket sort runs in linear time.

Exercises

8.4-1

Using Figure 8.4 as a model, illustrate the operation of BUCKET-SORT on the array $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.

8.4-2

Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

8.4-3

Let X be a random variable that is equal to the number of heads in two flips of a fair coin. What is $E[X^2]$? What is $E^2[X]$?

8.4-4

An array A of size $n > 10$ is filled in the following way. For each element $A[i]$, choose two random variables x_i and y_i uniformly and independently from $[0, 1)$. Then set

$$A[i] = \frac{\lfloor 10x_i \rfloor}{10} + \frac{y_i}{n} .$$

Modify bucket sort so that it sorts the array A in $O(n)$ expected time.

★ ***8.4-5***

You are given n points in the unit disk, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \le 1$ for $i = 1, 2, \ldots, n$. Suppose that the points are uniformly distributed, that is, the probability of finding a point in any region of the disk is proportional to the area of that region. Design an algorithm with an average-case running time of $\Theta(n)$ to sort the n points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin. (*Hint:* Design the bucket sizes in BUCKET-SORT to reflect the uniform distribution of the points in the unit disk.)

★ ***8.4-6***

A ***probability distribution function*** $P(x)$ for a random variable X is defined by $P(x) = \Pr\{X \le x\}$. Suppose that you draw a list of n random variables

$X_1, X_2, \ldots, X_n$ from a continuous probability distribution function P that is computable in $O(1)$ time (given y you can find x such that $P(x) = y$ in $O(1)$ time). Give an algorithm that sorts these numbers in linear average-case time.

Problems

8-1 Probabilistic lower bounds on comparison sorting

In this problem, you will prove a probabilistic $\Omega(n \lg n)$ lower bound on the running time of any deterministic or randomized comparison sort on n distinct input elements. You'll begin by examining a deterministic comparison sort A with decision tree T_A. Assume that every permutation of A's inputs is equally likely.

a. Suppose that each leaf of T_A is labeled with the probability that it is reached given a random input. Prove that exactly $n!$ leaves are labeled $1/n!$ and that the rest are labeled 0.

b. Let $D(T)$ denote the external path length of a decision tree T—the sum of the depths of all the leaves of T. Let T be a decision tree with $k > 1$ leaves, and let LT and RT be the left and right subtrees of T. Show that $D(T) = D(LT) + D(RT) + k$.

c. Let $d(k)$ be the minimum value of $D(T)$ over all decision trees T with $k > 1$ leaves. Show that $d(k) = \min\{d(i) + d(k-i) + k : 1 \le i \le k-1\}$. (*Hint:* Consider a decision tree T with k leaves that achieves the minimum. Let i_0 be the number of leaves in LT and $k - i_0$ the number of leaves in RT.)

d. Prove that for a given value of $k > 1$ and i in the range $1 \le i \le k-1$, the function $i \lg i + (k-i)\lg(k-i)$ is minimized at $i = k/2$. Conclude that $d(k) = \Omega(k \lg k)$.

e. Prove that $D(T_A) = \Omega(n!\lg(n!))$, and conclude that the average-case time to sort n elements is $\Omega(n \lg n)$.

Now consider a *randomized* comparison sort B. We can extend the decision-tree model to handle randomization by incorporating two kinds of nodes: ordinary comparison nodes and "randomization" nodes. A randomization node models a random choice of the form RANDOM$(1, r)$ made by algorithm B. The node has r children, each of which is equally likely to be chosen during an execution of the algorithm.

f. Show that for any randomized comparison sort B, there exists a deterministic comparison sort A whose expected number of comparisons is no more than those made by B.

8-2 Sorting in place in linear time

You have an array of n data records to sort, each with a key of 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

a. Give an algorithm that satisfies criteria 1 and 2 above.

b. Give an algorithm that satisfies criteria 1 and 3 above.

c. Give an algorithm that satisfies criteria 2 and 3 above.

d. Can you use any of your sorting algorithms from parts (a)–(c) as the sorting method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts n records with b-bit keys in $O(bn)$ time? Explain how or why not.

e. Suppose that the n records have keys in the range from 1 to k. Show how to modify counting sort so that it sorts the records in place in $O(n + k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable?

8-3 Sorting variable-length items

a. You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over *all* the integers in the array is n. Show how to sort the array in $O(n)$ time.

b. You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n. Show how to sort the strings in $O(n)$ time. (The desired order is the standard alphabetical order: for example, `a` < `ab` < `b`.)

8-4 Water jugs

You are given n red and n blue water jugs, all of different shapes and sizes. All the red jugs hold different amounts of water, as do all the blue jugs, and you cannot tell from the size of a jug how much water it holds. Moreover, for every jug of one color, there is a jug of the other color that holds the same amount of water.

Your task is to group the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair

of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This operation tells you whether the red jug or the blue jug can hold more water, or that they have the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

a. Describe a deterministic algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs.

b. Prove a lower bound of $\Omega(n \lg n)$ for the number of comparisons that an algorithm solving this problem must make.

c. Give a randomized algorithm whose expected number of comparisons is $O(n \lg n)$, and prove that this bound is correct. What is the worst-case number of comparisons for your algorithm?

8-5 Average sorting

Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an n-element array A ***k-sorted*** if, for all $i = 1, 2, \ldots, n - k$, the following holds:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \le \frac{\sum_{j=i+1}^{i+k} A[j]}{k} .$$

a. What does it mean for an array to be 1-sorted?

b. Give a permutation of the numbers $1, 2, \ldots, 10$ that is 2-sorted, but not sorted.

c. Prove that an n-element array is k-sorted if and only if $A[i] \le A[i + k]$ for all $i = 1, 2, \ldots, n - k$.

d. Give an algorithm that k-sorts an n-element array in $O(n \lg(n/k))$ time.

We can also show a lower bound on the time to produce a k-sorted array, when k is a constant.

e. Show how to sort a k-sorted array of length n in $O(n \lg k)$ time. (*Hint:* Use the solution to Exercise 6.5-11.)

f. Show that when k is a constant, k-sorting an n-element array requires $\Omega(n \lg n)$ time. (*Hint:* Use the solution to part (e) along with the lower bound on comparison sorts.)

8-6 Lower bound on merging sorted lists

The problem of merging two sorted lists arises frequently. We have seen a procedure for it as the subroutine MERGE in Section 2.3.1. In this problem, you will prove a lower bound of $2n - 1$ on the worst-case number of comparisons required to merge two sorted lists, each containing n items. First, you will show a lower bound of $2n - o(n)$ comparisons by using a decision tree.

a. Given $2n$ numbers, compute the number of possible ways to divide them into two sorted lists, each with n numbers.

b. Using a decision tree and your answer to part (a), show that any algorithm that correctly merges two sorted lists must perform at least $2n - o(n)$ comparisons.

Now you will show a slightly tighter $2n - 1$ bound.

c. Show that if two elements are consecutive in the sorted order and from different lists, then they must be compared.

d. Use your answer to part (c) to show a lower bound of $2n - 1$ comparisons for merging two sorted lists.

8-7 The 0-1 sorting lemma and columnsort

A ***compare-exchange*** operation on two array elements $A[i]$ and $A[j]$, where $i < j$, has the form

```
COMPARE-EXCHANGE(A, i, j)
if A[i] > A[j]
    exchange A[i] with A[j]
```

After the compare-exchange operation, we know that $A[i] \le A[j]$.

An ***oblivious compare-exchange algorithm*** operates solely by a sequence of prespecified compare-exchange operations. The indices of the positions compared in the sequence must be determined in advance, and although they can depend on the number of elements being sorted, they cannot depend on the values being sorted, nor can they depend on the result of any prior compare-exchange operation. For example, the COMPARE-EXCHANGE-INSERTION-SORT procedure on the facing page shows a variation of insertion sort as an oblivious compare-exchange algorithm. (Unlike the INSERTION-SORT procedure on page 19, the oblivious version runs in $\Theta(n^2)$ time in all cases.)

The ***0-1 sorting lemma*** provides a powerful way to prove that an oblivious compare-exchange algorithm produces a sorted result. It states that if an oblivious compare-exchange algorithm correctly sorts all input sequences consisting of only 0s and 1s, then it correctly sorts all inputs containing arbitrary values.

```
COMPARE-EXCHANGE-INSERTION-SORT(A, n)
for i = 2 to n
    for j = i - 1 downto 1
        COMPARE-EXCHANGE(A, j, j + 1)
```

You will prove the 0-1 sorting lemma by proving its contrapositive: if an oblivious compare-exchange algorithm fails to sort an input containing arbitrary values, then it fails to sort some 0-1 input. Assume that an oblivious compare-exchange algorithm X fails to correctly sort the array $A[1:n]$. Let $A[p]$ be the smallest value in A that algorithm X puts into the wrong location, and let $A[q]$ be the value that algorithm X moves to the location into which $A[p]$ should have gone. Define an array $B[1:n]$ of 0s and 1s as follows:

$$B[i] = \begin{cases} 0 & \text{if } A[i] \le A[p] , \\ 1 & \text{if } A[i] > A[p] . \end{cases}$$

a. Argue that $A[q] > A[p]$, so that $B[p] = 0$ and $B[q] = 1$.

b. To complete the proof of the 0-1 sorting lemma, prove that algorithm X fails to sort array B correctly.

Now you will use the 0-1 sorting lemma to prove that a particular sorting algorithm works correctly. The algorithm, ***columnsort***, works on a rectangular array of n elements. The array has r rows and s columns (so that $n = rs$), subject to three restrictions:

- r must be even,
- s must be a divisor of r, and
- $r \ge 2s^2$.

When columnsort completes, the array is sorted in ***column-major order***: reading down each column in turn, from left to right, the elements monotonically increase.

Columnsort operates in eight steps, regardless of the value of n. The odd steps are all the same: sort each column individually. Each even step is a fixed permutation. Here are the steps:

1. Sort each column.
2. Transpose the array, but reshape it back to r rows and s columns. In other words, turn the leftmost column into the top r/s rows, in order; turn the next column into the next r/s rows, in order; and so on.

10 14 5	4 1 2	4 8 10	1 3 6	1 4 11
8 7 17	8 3 5	12 16 18	2 5 7	3 8 14
12 1 6	10 7 6	1 3 7	4 8 10	6 10 17
16 9 11	12 9 11	9 14 15	9 13 15	2 9 12
4 15 2	16 14 13	2 5 6	11 14 17	5 13 16
18 3 13	18 15 17	11 13 17	12 16 18	7 15 18
(a)	(b)	(c)	(d)	(e)

1 4 11	5 10 16	4 10 16	1 7 13
2 8 12	6 13 17	5 11 17	2 8 14
3 9 14	7 15 18	6 12 18	3 9 15
5 10 16	1 4 11	1 7 13	4 10 16
6 13 17	2 8 12	2 8 14	5 11 17
7 15 18	3 9 14	3 9 15	6 12 18
(f)	(g)	(h)	(i)

Figure 8.5 The steps of columnsort. **(a)** The input array with 6 rows and 3 columns. (This example does not obey the $r \geq 2s^2$ requirement, but it works.) **(b)** After sorting each column in step 1. **(c)** After transposing and reshaping in step 2. **(d)** After sorting each column in step 3. **(e)** After performing step 4, which inverts the permutation from step 2. **(f)** After sorting each column in step 5. **(g)** After shifting by half a column in step 6. **(h)** After sorting each column in step 7. **(i)** After performing step 8, which inverts the permutation from step 6. Steps 6–8 sort the bottom half of each column with the top half of the next column. After step 8, the array is sorted in column-major order.

3. Sort each column.
4. Perform the inverse of the permutation performed in step 2.
5. Sort each column.
6. Shift the top half of each column into the bottom half of the same column, and shift the bottom half of each column into the top half of the next column to the right. Leave the top half of the leftmost column empty. Shift the bottom half of the last column into the top half of a new rightmost column, and leave the bottom half of this new column empty.
7. Sort each column.
8. Perform the inverse of the permutation performed in step 6.

You can think of steps 6–8 as a single step that sorts the bottom half of each column and the top half of the next column. Figure 8.5 shows an example of the steps of columnsort with $r = 6$ and $s = 3$. (Even though this example violates the requirement that $r \geq 2s^2$, it happens to work.)

c. Argue that we can treat columnsort as an oblivious compare-exchange algorithm, even if we do not know what sorting method the odd steps use.

Although it might seem hard to believe that columnsort actually sorts, you will use the 0-1 sorting lemma to prove that it does. The 0-1 sorting lemma applies because we can treat columnsort as an oblivious compare-exchange algorithm. A couple of definitions will help you apply the 0-1 sorting lemma. We say that an area of an array is ***clean*** if we know that it contains either all 0s or all 1s or if it is empty. Otherwise, the area might contain mixed 0s and 1s, and it is ***dirty***. From here on, assume that the input array contains only 0s and 1s, and that we can treat it as an array with r rows and s columns.

d. Prove that after steps 1–3, the array consists of clean rows of 0s at the top, clean rows of 1s at the bottom, and at most s dirty rows between them. (One of the clean rows could be empty.)

e. Prove that after step 4, the array, read in column-major order, starts with a clean area of 0s, ends with a clean area of 1s, and has a dirty area of at most s^2 elements in the middle. (Again, one of the clean areas could be empty.)

f. Prove that steps 5–8 produce a fully sorted 0-1 output. Conclude that columnsort correctly sorts all inputs containing arbitrary values.

g. Now suppose that s does not divide r. Prove that after steps 1–3, the array consists of clean rows of 0s at the top, clean rows of 1s at the bottom, and at most $2s - 1$ dirty rows between them. (Once again, one of the clean areas could be empty.) How large must r be, compared with s, for columnsort to correctly sort when s does not divide r?

h. Suggest a simple change to step 1 that allows us to maintain the requirement that $r \geq 2s^2$ even when s does not divide r, and prove that with your change, columnsort correctly sorts.

Chapter notes

The decision-tree model for studying comparison sorts was introduced by Ford and Johnson [150]. Knuth's comprehensive treatise on sorting [261] covers many variations on the sorting problem, including the information-theoretic lower bound on the complexity of sorting given here. Ben-Or [46] studied lower bounds for sorting using generalizations of the decision-tree model.

Knuth credits H. H. Seward with inventing counting sort in 1954, as well as with the idea of combining counting sort with radix sort. Radix sorting starting with the least significant digit appears to be a folk algorithm widely used by operators of

mechanical card-sorting machines. According to Knuth, the first published reference to the method is a 1929 document by L. J. Comrie describing punched-card equipment. Bucket sorting has been in use since 1956, when the basic idea was proposed by Isaac and Singleton [235].

Munro and Raman [338] give a stable sorting algorithm that performs $O(n^{1+\epsilon})$ comparisons in the worst case, where $0 < \epsilon \leq 1$ is any fixed constant. Although any of the $O(n \lg n)$-time algorithms make fewer comparisons, the algorithm by Munro and Raman moves data only $O(n)$ times and operates in place.

The case of sorting n b-bit integers in $o(n \lg n)$ time has been considered by many researchers. Several positive results have been obtained, each under slightly different assumptions about the model of computation and the restrictions placed on the algorithm. All the results assume that the computer memory is divided into addressable b-bit words. Fredman and Willard [157] introduced the fusion tree data structure and used it to sort n integers in $O(n \lg n / \lg \lg n)$ time. This bound was later improved to $O(n \sqrt{\lg n})$ time by Andersson [17]. These algorithms require the use of multiplication and several precomputed constants. Andersson, Hagerup, Nilsson, and Raman [18] have shown how to sort n integers in $O(n \lg \lg n)$ time without using multiplication, but their method requires storage that can be unbounded in terms of n. Using multiplicative hashing, we can reduce the storage needed to $O(n)$, but then the $O(n \lg \lg n)$ worst-case bound on the running time becomes an expected-time bound. Generalizing the exponential search trees of Andersson [17], Thorup [434] gave an $O(n(\lg \lg n)^2)$-time sorting algorithm that does not use multiplication or randomization, and it uses linear space. Combining these techniques with some new ideas, Han [207] improved the bound for sorting to $O(n \lg \lg n \lg \lg \lg n)$ time. Although these algorithms are important theoretical breakthroughs, they are all fairly complicated and at the present time seem unlikely to compete with existing sorting algorithms in practice.

The columnsort algorithm in Problem 8-7 is by Leighton [286].

9 Medians and Order Statistics

The ith ***order statistic*** of a set of n elements is the ith smallest element. For example, the ***minimum*** of a set of elements is the first order statistic ($i = 1$), and the ***maximum*** is the nth order statistic ($i = n$). A ***median***, informally, is the "halfway point" of the set. When n is odd, the median is unique, occurring at $i = (n+1)/2$. When n is even, there are two medians, the ***lower median*** occurring at $i = n/2$ and the ***upper median*** occurring at $i = n/2 + 1$. Thus, regardless of the parity of n, medians occur at $i = \lfloor (n+1)/2 \rfloor$ and $i = \lceil (n+1)/2 \rceil$. For simplicity in this text, however, we consistently use the phrase "the median" to refer to the lower median.

This chapter addresses the problem of selecting the ith order statistic from a set of n distinct numbers. We assume for convenience that the set contains distinct numbers, although virtually everything that we do extends to the situation in which a set contains repeated values. We formally specify the ***selection problem*** as follows:

Input: A set A of n distinct numbers[1] and an integer i, with $1 \le i \le n$.

Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements of A.

We can solve the selection problem in $O(n \lg n)$ time simply by sorting the numbers using heapsort or merge sort and then outputting the ith element in the sorted array. This chapter presents asymptotically faster algorithms.

Section 9.1 examines the problem of selecting the minimum and maximum of a set of elements. More interesting is the general selection problem, which we investigate in the subsequent two sections. Section 9.2 analyzes a practical randomized algorithm that achieves an $O(n)$ expected running time, assuming dis-

[1] As in the footnote on page 182, you can enforce the assumption that the numbers are distinct by converting each input value $A[i]$ to an ordered pair $(A[i], i)$ with $(A[i], i) < (A[j], j)$ if either $A[i] < A[j]$ or $A[i] = A[j]$ and $i < j$.

tinct elements. Section 9.3 contains an algorithm of more theoretical interest that achieves the $O(n)$ running time in the worst case.

9.1 Minimum and maximum

How many comparisons are necessary to determine the minimum of a set of n elements? To obtain an upper bound of $n - 1$ comparisons, just examine each element of the set in turn and keep track of the smallest element seen so far. The MINIMUM procedure assumes that the set resides in array $A[1:n]$.

```
MINIMUM(A, n)
min = A[1]
for i = 2 to n
    if min > A[i]
        min = A[i]
return min
```

It's no more difficult to find the maximum with $n - 1$ comparisons.

Is this algorithm for minimum the best we can do? Yes, because it turns out that there's a lower bound of $n - 1$ comparisons for the problem of determining the minimum. Think of any algorithm that determines the minimum as a tournament among the elements. Each comparison is a match in the tournament in which the smaller of the two elements wins. Since every element except the winner must lose at least one match, we can conclude that $n - 1$ comparisons are necessary to determine the minimum. Hence the algorithm MINIMUM is optimal with respect to the number of comparisons performed.

Simultaneous minimum and maximum

Some applications need to find both the minimum and the maximum of a set of n elements. For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display screen or other graphical output device. To do so, the program must first determine the minimum and maximum value of each coordinate.

Of course, we can determine both the minimum and the maximum of n elements using $\Theta(n)$ comparisons. We simply find the minimum and maximum independently, using $n - 1$ comparisons for each, for a total of $2n - 2 = \Theta(n)$ comparisons.

Although $2n-2$ comparisons is asymptotically optimal, it is possible to improve the leading constant. We can find both the minimum and the maximum using at most $3\lfloor n/2 \rfloor$ comparisons. The trick is to maintain both the minimum and maximum elements seen thus far. Rather than processing each element of the input by comparing it against the current minimum and maximum, at a cost of 2 comparisons per element, process elements in pairs. Compare pairs of elements from the input first *with each other*, and then compare the smaller with the current minimum and the larger to the current maximum, at a cost of 3 comparisons for every 2 elements.

How you set up initial values for the current minimum and maximum depends on whether n is odd or even. If n is odd, set both the minimum and maximum to the value of the first element, and then process the rest of the elements in pairs. If n is even, perform 1 comparison on the first 2 elements to determine the initial values of the minimum and maximum, and then process the rest of the elements in pairs as in the case for odd n.

Let's count the total number of comparisons. If n is odd, then $3\lfloor n/2 \rfloor$ comparisons occur. If n is even, 1 initial comparison occurs, followed by another $3(n-2)/2$ comparisons, for a total of $3n/2-2$. Thus, in either case, the total number of comparisons is at most $3\lfloor n/2 \rfloor$.

Exercises

9.1-1

Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. (*Hint:* Also find the smallest element.)

9.1-2

Given $n > 2$ distinct numbers, you want to find a number that is neither the minimum nor the maximum. What is the smallest number of comparisons that you need to perform?

9.1-3

A racetrack can run races with five horses at a time to determine their relative speeds. For 25 horses, it takes six races to determine the fastest horse, assuming transitivity (see page 1159). What's the minimum number of races it takes to determine the fastest three horses out of 25?

★ ***9.1-4***

Prove the lower bound of $\lceil 3n/2 \rceil - 2$ comparisons in the worst case to find both the maximum and minimum of n numbers. (*Hint:* Consider how many numbers are potentially either the maximum or minimum, and investigate how a comparison affects these counts.)

9.2 Selection in expected linear time

The general selection problem—finding the ith order statistic for any value of i—appears more difficult than the simple problem of finding a minimum. Yet, surprisingly, the asymptotic running time for both problems is the same: $\Theta(n)$. This section presents a divide-and-conquer algorithm for the selection problem. The algorithm RANDOMIZED-SELECT is modeled after the quicksort algorithm of Chapter 7. Like quicksort it partitions the input array recursively. But unlike quicksort, which recursively processes both sides of the partition, RANDOMIZED-SELECT works on only one side of the partition. This difference shows up in the analysis: whereas quicksort has an expected running time of $\Theta(n \lg n)$, the expected running time of RANDOMIZED-SELECT is $\Theta(n)$, assuming that the elements are distinct.

RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION introduced in Section 7.3. Like RANDOMIZED-QUICKSORT, it is a randomized algorithm, since its behavior is determined in part by the output of a random-number generator. The RANDOMIZED-SELECT procedure returns the ith smallest element of the array $A[p:r]$, where $1 \le i \le r - p + 1$.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]        // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6      return A[q]        // the pivot value is the answer
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

Figure 9.1 illustrates how the RANDOMIZED-SELECT procedure works. Line 1 checks for the base case of the recursion, in which the subarray $A[p:r]$ consists of just one element. In this case, i must equal 1, and line 2 simply returns $A[p]$ as the ith smallest element. Otherwise, the call to RANDOMIZED-PARTITION in line 3 partitions the array $A[p:r]$ into two (possibly empty) subarrays $A[p:q-1]$ and $A[q+1:r]$ such that each element of $A[p:q-1]$ is less than or equal to $A[q]$, which in turn is less than each element of $A[q+1:r]$. (Although our analysis assumes that the elements are distinct, the procedure still yields the correct result even if equal elements are present.) As in quicksort, we'll refer to $A[q]$ as the ***pivot*** element. Line 4 computes the number k of elements in the subarray $A[p:q]$, that is,

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	p	r	i	partitioning	helpful?
$A^{(0)}$	6	19	4	12	14	9	15	7	8	11	3	13	2	5	10	1	15	5		
																			1	no
$A^{(1)}$	6	4	12	10	9	7	8	11	3	13	2	5	14	19	15	1	12	5		
																			2	yes
$A^{(2)}$	3	2	4	10	9	7	8	11	6	13	5	12	14	19	15	4	12	2		
																			3	no
$A^{(3)}$	3	2	4	10	9	7	8	11	6	12	5	13	14	19	15	4	11	2		
																			4	yes
$A^{(4)}$	3	2	4	5	6	7	8	11	9	12	10	13	14	19	15	4	5	2		
																			5	yes
$A^{(5)}$	3	2	4	5	6	7	8	11	9	12	10	13	14	19	15	5	5	1		

Figure 9.1 The action of RANDOMIZED-SELECT as successive partitionings narrow the subarray $A[p:r]$, showing the values of the parameters p, r, and i at each recursive call. The subarray $A[p:r]$ in each recursive step is shown in tan, with the dark tan element selected as the pivot for the next partitioning. Blue elements are outside $A[p:r]$. The answer is the tan element in the bottom array, where $p = r = 5$ and $i = 1$. The array designations $A^{(0)}, A^{(1)}, \ldots, A^{(5)}$, the partitioning numbers, and whether the partitioning is helpful are explained on the following page.

the number of elements in the low side of the partition, plus 1 for the pivot element. Line 5 then checks whether $A[q]$ is the ith smallest element. If it is, then line 6 returns $A[q]$. Otherwise, the algorithm determines in which of the two subarrays $A[p:q-1]$ and $A[q+1:r]$ the ith smallest element lies. If $i < k$, then the desired element lies on the low side of the partition, and line 8 recursively selects it from the subarray. If $i > k$, however, then the desired element lies on the high side of the partition. Since we already know k values that are smaller than the ith smallest element of $A[p:r]$—namely, the elements of $A[p:q]$—the desired element is the $(i-k)$th smallest element of $A[q+1:r]$, which line 9 finds recursively. The code appears to allow recursive calls to subarrays with 0 elements, but Exercise 9.2-1 asks you to show that this situation cannot happen.

The worst-case running time for RANDOMIZED-SELECT is $\Theta(n^2)$, even to find the minimum, because it could be extremely unlucky and always partition around the largest remaining element before identifying the ith smallest when only one element remains. In this worst case, each recursive step removes only the pivot from consideration. Because partitioning n elements takes $\Theta(n)$ time, the recurrence for the worst-case running time is the same as for QUICKSORT:

$T(n) = T(n-1) + \Theta(n)$, with the solution $T(n) = \Theta(n^2)$. We'll see that the algorithm has a linear expected running time, however, and because it is randomized, no particular input elicits the worst-case behavior.

To see the intuition behind the linear expected running time, suppose that each time the algorithm randomly selects a pivot element, the pivot lies somewhere within the second and third quartiles—the "middle half"—of the remaining elements in sorted order. If the ith smallest element is less than the pivot, then all the elements greater than the pivot are ignored in all future recursive calls. These ignored elements include at least the uppermost quartile, and possibly more. Likewise, if the ith smallest element is greater than the pivot, then all the elements less than the pivot—at least the first quartile—are ignored in all future recursive calls. Either way, therefore, at least $1/4$ of the remaining elements are ignored in all future recursive calls, leaving at most $3/4$ of the remaining elements ***in play***: residing in the subarray $A[p:r]$. Since RANDOMIZED-PARTITION takes $\Theta(n)$ time on a subarray of n elements, the recurrence for the worst-case running time is $T(n) = T(3n/4) + \Theta(n)$. By case 3 of the master method (Theorem 4.1 on page 102), this recurrence has solution $T(n) = \Theta(n)$.

Of course, the pivot does not necessarily fall into the middle half every time. Since the pivot is selected at random, the probability that it falls into the middle half is about $1/2$ each time. We can view the process of selecting the pivot as a Bernoulli trial (see Section C.4) with success equating to the pivot residing in the middle half. Thus the expected number of trials needed for success is given by a geometric distribution: just two trials on average (equation (C.36) on page 1197). In other words, we expect that half of the partitionings reduce the number of elements still in play by at least $3/4$ and that half of the partitionings do not help as much. Consequently, the expected number of partitionings at most doubles from the case when the pivot always falls into the middle half. The cost of each extra partitioning is less than the one that preceded it, so that the expected running time is still $\Theta(n)$.

To make the above argument rigorous, we start by defining the random variable $A^{(j)}$ as the set of elements of A that are still in play after j partitionings (that is, within the subarray $A[p:r]$ after j calls of RANDOMIZED-SELECT), so that $A^{(0)}$ consists of all the elements in A. Since each partitioning removes at least one element—the pivot—from being in play, the sequence $|A^{(0)}|, |A^{(1)}|, |A^{(2)}|, \ldots$ strictly decreases. Set $A^{(j-1)}$ is in play before the jth partitioning, and set $A^{(j)}$ remains in play afterward. For convenience, assume that the initial set $A^{(0)}$ is the result of a 0th "dummy" partitioning.

Let's call the jth partitioning ***helpful*** if $|A^{(j)}| \le (3/4)|A^{(j-1)}|$. Figure 9.1 shows the sets $A^{(j)}$ and whether partitionings are helpful for an example array. A helpful partitioning corresponds to a successful Bernoulli trial. The following lemma shows that a partitioning is at least as likely to be helpful as not.

Lemma 9.1
A partitioning is helpful with probability at least $1/2$.

Proof Whether a partitioning is helpful depends on the randomly chosen pivot. We discussed the "middle half" in the informal argument above. Let's more precisely define the middle half of an n-element subarray as all but the smallest $\lceil n/4 \rceil - 1$ and greatest $\lceil n/4 \rceil - 1$ elements (that is, all but the first $\lceil n/4 \rceil - 1$ and last $\lceil n/4 \rceil - 1$ elements if the subarray were sorted). We'll prove that if the pivot falls into the middle half, then the pivot leads to a helpful partitioning, and we'll also prove that the probability of the pivot falling into the middle half is at least $1/2$.

Regardless of where the pivot falls, either all the elements greater than it or all the elements less than it, along with the pivot itself, will no longer be in play after partitioning. If the pivot falls into the middle half, therefore, at least $\lceil n/4 \rceil - 1$ elements less than the pivot or $\lceil n/4 \rceil - 1$ elements greater than the pivot, plus the pivot, will no longer be in play after partitioning. That is, at least $\lceil n/4 \rceil$ elements will no longer be in play. The number of elements remaining in play will be at most $n - \lceil n/4 \rceil$, which equals $\lfloor 3n/4 \rfloor$ by Exercise 3.3-2 on page 70. Since $\lfloor 3n/4 \rfloor \le 3n/4$, the partitioning is helpful.

To determine a lower bound on the probability that a randomly chosen pivot falls into the middle half, we determine an upper bound on the probability that it does not. That probability is

$$\begin{aligned}
\frac{2(\lceil n/4 \rceil - 1)}{n} &\le \frac{2((n/4+1)-1)}{n} \quad \text{(by inequality (3.2) on page 64)} \\
&= \frac{n/2}{n} \\
&= 1/2 \,.
\end{aligned}$$

Thus, the pivot has a probability of at least $1/2$ of falling into the middle half, and so the probability is at least $1/2$ that a partitioning is helpful. ■

We can now bound the expected running time of RANDOMIZED-SELECT.

Theorem 9.2
The procedure RANDOMIZED-SELECT on an input array of n distinct elements has an expected running time of $\Theta(n)$.

Proof Since not every partitioning is necessarily helpful, let's give each partitioning an index starting at 0 and denote by $\langle h_0, h_1, h_2, \ldots, h_m \rangle$ the sequence of partitionings that are helpful, so that the h_kth partitioning is helpful for $k = 0, 1, 2, \ldots, m$. Although the number m of helpful partitionings is a random vari-

Figure 9.2 The sets within each generation in the proof of Theorem 9.2. Vertical lines represent the sets, with the height of each line indicating the size of the set, which equals the number of elements in play. Each generation starts with a set $A^{(h_k)}$, which is the result of a helpful partitioning. These sets are drawn in black and are at most $3/4$ the size of the sets to their immediate left. Sets drawn in orange are not the first within a generation. A generation may contain just one set. The sets in generation k are $A^{(h_k)}, A^{(h_k+1)}, \ldots, A^{(h_{k+1}-1)}$. The sets $A^{(h_k)}$ are defined so that $|A^{(h_k)}| \le (3/4)|A^{(h_{k-1})}|$. If the partitioning gets all the way to generation h_m, set $A^{(h_m)}$ has at most one element in play.

able, we can bound it, since after at most $\lceil \log_{4/3} n \rceil$ helpful partitionings, only one element remains in play. Consider the dummy 0th partitioning as helpful, so that $h_0 = 0$. Denote $|A^{(h_k)}|$ by n_k, where $n_0 = |A^{(0)}|$ is the original problem size. Since the h_kth partitioning is helpful and the sizes of the sets $A^{(j)}$ strictly decrease, we have $n_k = |A^{(h_k)}| \le (3/4)|A^{(h_k-1)}| = (3/4)\, n_{k-1}$ for $k = 1, 2, \ldots, m$. By iterating $n_k \le (3/4)\, n_{k-1}$, we have that $n_k \le (3/4)^k n_0$ for $k = 0, 1, 2, \ldots, m$.

As Figure 9.2 depicts, we break up the sequence of sets $A^{(j)}$ into m ***generations*** consisting of consecutively partitioned sets, starting with the result $A^{(h_k)}$ of a helpful partitioning and ending with the last set $A^{(h_{k+1}-1)}$ before the next helpful partitioning, so that the sets in generation k are $A^{(h_k)}, A^{(h_k+1)}, \ldots, A^{(h_{k+1}-1)}$. Then for each set of elements $A^{(j)}$in the kth generation, we have that $|A^{(j)}| \le |A^{(h_k)}| = n_k \le (3/4)^k n_0$.

Next, we define the random variable

$$X_k = h_{k+1} - h_k$$

for $k = 0, 1, 2, \ldots, m-1$. That is, X_k is the number of sets in the kth generation, so that the sets in the kth generation are $A^{(h_k)}, A^{(h_k+1)}, \ldots, A^{(h_k+X_k-1)}$.

By Lemma 9.1, the probability that a partitioning is helpful is at least $1/2$. The probability is actually even higher, since a partitioning is helpful even if the pivot

does not fall into the middle half but the ith smallest element happens to lie in the smaller side of the partitioning. We'll just use the lower bound of $1/2$, however, and then equation (C.36) gives that $\mathrm{E}[X_k] \le 2$ for $k = 0, 1, 2, \ldots, m-1$.

Let's derive an upper bound on how many comparisons are made altogether during partitioning, since the running time is dominated by the comparisons. Since we are calculating an upper bound, assume that the recursion goes all the way until only one element remains in play. The jth partitioning takes the set $A^{(j-1)}$ of elements in play, and it compares the randomly chosen pivot with all the other $|A^{(j-1)}| - 1$ elements, so that the jth partitioning makes fewer than $|A^{(j-1)}|$ comparisons. The sets in the kth generation have sizes $|A^{(h_k)}|, |A^{(h_k+1)}|, \ldots, |A^{(h_k+X_k-1)}|$. Thus, the total number of comparisons during partitioning is less than

$$\begin{aligned}
\sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(j)}| &\le \sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(h_k)}| \\
&= \sum_{k=0}^{m-1} X_k \, |A^{(h_k)}| \\
&\le \sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0 \, .
\end{aligned}$$

Since $\mathrm{E}[X_k] \le 2$, we have that the expected total number of comparisons during partitioning is less than

$$\begin{aligned}
\mathrm{E}\left[\sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0\right] &= \sum_{k=0}^{m-1} \mathrm{E}\left[X_k \left(\frac{3}{4}\right)^k n_0\right] \quad \text{(by linearity of expectation)} \\
&= n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \mathrm{E}[X_k] \\
&\le 2n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \\
&< 2n_0 \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k \\
&= 8n_0 \qquad \text{(by equation (A.7) on page 1142)} \, .
\end{aligned}$$

Since n_0 is the size of the original array A, we conclude that the expected number of comparisons, and thus the expected running time, for RANDOMIZED-SELECT is $O(n)$. All n elements are examined in the first call of RANDOMIZED-

PARTITION, giving a lower bound of $\Omega(n)$. Hence the expected running time is $\Theta(n)$. ■

Exercises

9.2-1
Show that RANDOMIZED-SELECT never makes a recursive call to a 0-length array.

9.2-2
Write an iterative version of RANDOMIZED-SELECT.

9.2-3
Suppose that RANDOMIZED-SELECT is used to select the minimum element of the array $A = \langle 2, 3, 0, 5, 7, 9, 1, 8, 6, 4 \rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

9.2-4
Argue that the expected running time of RANDOMIZED-SELECT does not depend on the order of the elements in its input array $A[p : r]$. That is, the expected running time is the same for any permutation of the input array $A[p : r]$. (*Hint:* Argue by induction on the length n of the input array.)

9.3 Selection in worst-case linear time

We'll now examine a remarkable and theoretically interesting selection algorithm whose running time is $\Theta(n)$ in the worst case. Although the RANDOMIZED-SELECT algorithm from Section 9.2 achieves linear expected time, we saw that its running time in the worst case was quadratic. The selection algorithm presented in this section achieves linear time in the worst case, but it is not nearly as practical as RANDOMIZED-SELECT. It is mostly of theoretical interest.

Like the expected linear-time RANDOMIZED-SELECT, the worst-case linear-time algorithm SELECT finds the desired element by recursively partitioning the input array. Unlike RANDOMIZED-SELECT, however, SELECT *guarantees* a good split by choosing a provably good pivot when partitioning the array. The cleverness in the algorithm is that it finds the pivot recursively. Thus, there are two invocations of SELECT: one to find a good pivot, and a second to recursively find the desired order statistic.

The partitioning algorithm used by SELECT is like the deterministic partitioning algorithm PARTITION from quicksort (see Section 7.1), but modified to take the element to partition around as an additional input parameter. Like PARTITION, the

PARTITION-AROUND algorithm returns the index of the pivot. Since it's so similar to PARTITION, the pseudocode for PARTITION-AROUND is omitted.

The SELECT procedure takes as input a subarray $A[p:r]$ of $n = r - p + 1$ elements and an integer i in the range $1 \le i \le n$. It returns the ith smallest element of A. The pseudocode is actually more understandable than it might appear at first.

```
SELECT(A, p, r, i)
1  while (r − p + 1) mod 5 ≠ 0
2      for j = p + 1 to r                    // put the minimum into A[p]
3          if A[p] > A[j]
4              exchange A[p] with A[j]
5      // If we want the minimum of A[p : r], we're done.
6      if i == 1
7          return A[p]
8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
9      p = p + 1
10     i = i − 1
11 g = (r − p + 1)/5                         // number of 5-element groups
12 for j = p to p + g − 1                    // sort each group
13     sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14 // All group medians now lie in the middle fifth of A[p : r].
15 // Find the pivot x recursively as the median of the group medians.
16 x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17 q = PARTITION-AROUND(A, p, r, x)   // partition around the pivot
18 // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19 k = q − p + 1
20 if i == k
21     return A[q]                           // the pivot value is the answer
22 elseif i < k
23     return SELECT(A, p, q − 1, i)
24 else return SELECT(A, q + 1, r, i − k)
```

The pseudocode starts by executing the **while** loop in lines 1–10 to reduce the number $r - p + 1$ of elements in the subarray until it is divisible by 5. The **while** loop executes 0 to 4 times, each time rearranging the elements of $A[p:r]$ so that $A[p]$ contains the minimum element. If $i = 1$, which means that we actually want the minimum element, then the procedure simply returns it in line 7. Otherwise, SELECT eliminates the minimum from the subarray $A[p:r]$ and iterates to find the $(i-1)$st element in $A[p+1:r]$. Lines 9–10 do so by incrementing p and decrementing i. If the **while** loop completes all of its iterations without returning a

Figure 9.3 The relationships between elements (shown as circles) immediately after line 17 of the selection algorithm SELECT. There are $g = (r - p + 1)/5$ groups of 5 elements, each of which occupies a column. For example, the leftmost column contains elements $A[p]$, $A[p + g]$, $A[p + 2g]$, $A[p + 3g]$, $A[p + 4g]$, and the next column contains $A[p + 1]$, $A[p + g + 1]$, $A[p + 2g + 1]$, $A[p + 3g + 1]$, $A[p + 4g + 1]$. The medians of the groups are red, and the pivot x is labeled. Arrows go from smaller elements to larger. The elements on the blue background are all known to be less than or equal to x and cannot fall into the high side of the partition around x. The elements on the yellow background are known to be greater than or equal to x and cannot fall into the low side of the partition around x. The pivot x belongs to both the blue and yellow regions and is shown on a green background. The elements on the white background could lie on either side of the partition.

result, the procedure executes the core of the algorithm in lines 11–24, assured that the number $r - p + 1$ of elements in $A[p : r]$ is evenly divisible by 5.

The next part of the algorithm implements the following idea, illustrated in Figure 9.3. Divide the elements in $A[p : r]$ into $g = (r - p + 1)/5$ groups of 5 elements each. The first 5-element group is

$$\langle A[p], A[p + g], A[p + 2g], A[p + 3g], A[p + 4g] \rangle ,$$

the second is

$$\langle A[p + 1], A[p + g + 1], A[p + 2g + 1], A[p + 3g + 1], A[p + 4g + 1] \rangle ,$$

and so forth until the last, which is

$$\langle A[p + g - 1], A[p + 2g - 1], A[p + 3g - 1], A[p + 4g - 1], A[r] \rangle .$$

(Note that $r = p + 5g - 1$.) Line 13 puts each group in order using, for example, insertion sort (Section 2.1), so that for $j = p, p + 1, \ldots, p + g - 1$, we have

$$A[j] \le A[j+g] \le A[j+2g] \le A[j+3g] \le A[j+4g] .$$

Each vertical column in Figure 9.3 depicts a sorted group of 5 elements. The median of each 5-element group is $A[j+2g]$, and thus all the 5-element medians, shown in red, lie in the range $A[p+2g : p+3g-1]$.

Next, line 16 determines the pivot x by recursively calling SELECT to find the median (specifically, the $\lceil g/2 \rceil$th smallest) of the g group medians. Line 17 uses the modified PARTITION-AROUND algorithm to partition the elements of $A[p:r]$ around x, returning the index q of x, so that $A[q] = x$, elements in $A[p:q]$ are all at most x, and elements in $A[q:r]$ are greater than or equal to x.

The remainder of the code mirrors that of RANDOMIZED-SELECT. If the pivot x is the ith largest, the procedure returns it. Otherwise, the procedure recursively calls itself on either $A[p:q-1]$ or $A[q+1:r]$, depending on the value of i.

Let's analyze the running time of SELECT and see how the judicious choice of the pivot x plays into a guarantee on its worst-case running time.

Theorem 9.3
The running time of SELECT on an input of n elements is $\Theta(n)$.

Proof Define $T(n)$ as the worst-case time to run SELECT on any input subarray $A[p:r]$ of size at most n, that is, for which $r - p + 1 \le n$. By this definition, $T(n)$ is monotonically increasing.

We first determine an upper bound on the time spent outside the recursive calls in lines 16, 23, and 24. The **while** loop in lines 1–10 executes 0 to 4 times, which is $O(1)$ times. Since the dominant time within the loop is the computation of the minimum in lines 2–4, which takes $\Theta(n)$ time, lines 1–10 execute in $O(1) \cdot \Theta(n) = O(n)$ time. The sorting of the 5-element groups in lines 12–13 takes $\Theta(n)$ time because each 5-element group takes $\Theta(1)$ time to sort (even using an asymptotically inefficient sorting algorithm such as insertion sort), and there are g elements to sort, where $n/5 - 1 < g \le n/5$. Finally, the time to partition in line 17 is $\Theta(n)$, as Exercise 7.1-3 on page 187 asks you to show. Because the remaining bookkeeping only costs $\Theta(1)$ time, the total amount of time spent outside of the recursive calls is $O(n) + \Theta(n) + \Theta(n) + \Theta(1) = \Theta(n)$.

Now let's determine the running time for the recursive calls. The recursive call to find the pivot in line 16 takes $T(g) \le T(n/5)$ time, since $g \le n/5$ and $T(n)$ monotonically increases. Of the two recursive calls in lines 23 and 24, at most one is executed. But we'll see that no matter which of these two recursive calls to SELECT actually executes, the number of elements in the recursive call turns out to be at most $7n/10$, and hence the worst-case cost for lines 23 and 24 is at most $T(7n/10)$. Let's now show that the machinations with group medians and the choice of the pivot x as the median of the group medians guarantees this property.

Figure 9.3 helps to visualize what's going on. There are $g \le n/5$ groups of 5 elements, with each group shown as a column sorted from bottom to top. The arrows show the ordering of elements within the columns. The columns are ordered from left to right with groups to the left of x's group having a group median less than x and those to the right of x's group having a group median greater than x. Although the relative order within each group matters, the relative order among groups to the left of x's column doesn't really matter, and neither does the relative order among groups to the right of x's column. The important thing is that the groups to the left have group medians less than x (shown by the horizontal arrows entering x), and that the groups to the right have group medians greater than x (shown by the horizontal arrows leaving x). Thus, the yellow region contains elements that we know are greater than or equal to x, and the blue region contains elements that we know are less than or equal to x.

These two regions each contain at least $3g/2$ elements. The number of group medians in the yellow region is $\lfloor g/2 \rfloor + 1$, and for each group median, two additional elements are greater than it, making a total of $3(\lfloor g/2 \rfloor + 1) \ge 3g/2$ elements. Similarly, the number of group medians in the blue region is $\lceil g/2 \rceil$, and for each group median, two additional elements are less than it, making a total of $3\lceil g/2 \rceil \ge 3g/2$.

The elements in the yellow region cannot fall into the low side of the partition around x, and those in the blue region cannot fall into the high side. The elements in neither region—those lying on a white background—could fall into either side of the partition. But since the low side of the partition excludes the elements in the yellow region, and there are a total of $5g$ elements, we know that the low side of the partition can contain at most $5g - 3g/2 = 7g/2 \le 7n/10$ elements. Likewise, the high side of the partition excludes the elements in the blue region, and a similar calculation shows that it also contains at most $7n/10$ elements.

All of which leads to the following recurrence for the worst-case running time of SELECT:

$$T(n) \le T(n/5) + T(7n/10) + \Theta(n) . \tag{9.1}$$

We can show that $T(n) = O(n)$ by substitution.[2] More specifically, we'll prove that $T(n) \le cn$ for some suitably large constant $c > 0$ and all $n > 0$. Substituting this inductive hypothesis into the right-hand side of recurrence (9.1) and assuming that $n \ge 5$ yields

[2] We could also use the Akra-Bazzi method from Section 4.7, which involves calculus, to solve this recurrence. Indeed, a similar recurrence (4.24) on page 117 was used to illustrate that method.

$$\begin{aligned} T(n) &\le c(n/5) + c(7n/10) + \Theta(n) \\ &\le 9cn/10 + \Theta(n) \\ &= cn - cn/10 + \Theta(n) \\ &\le cn \end{aligned}$$

if c is chosen large enough that $c/10$ dominates the upper-bound constant hidden by the $\Theta(n)$. In addition to this constraint, we can pick c large enough that $T(n) \le cn$ for all $n \le 4$, which is the base case of the recursion within SELECT. The running time of SELECT is therefore $O(n)$ in the worst case, and because line 13 alone takes $\Theta(n)$ time, the total time is $\Theta(n)$. ■

As in a comparison sort (see Section 8.1), SELECT and RANDOMIZED-SELECT determine information about the relative order of elements only by comparing elements. Recall from Chapter 8 that sorting requires $\Omega(n \lg n)$ time in the comparison model, even on average (see Problem 8-1). The linear-time sorting algorithms in Chapter 8 make assumptions about the type of the input. In contrast, the linear-time selection algorithms in this chapter do not require any assumptions about the input's type, only that the elements are distinct and can be pairwise compared according to a linear order. The algorithms in this chapter are not subject to the $\Omega(n \lg n)$ lower bound, because they manage to solve the selection problem without sorting all the elements. Thus, solving the selection problem by sorting and indexing, as presented in the introduction to this chapter, is asymptotically inefficient in the comparison model.

Exercises

9.3-1

In the algorithm SELECT, the input elements are divided into groups of 5. Show that the algorithm works in linear time if the input elements are divided into groups of 7 instead of 5.

9.3-2

Suppose that the preprocessing in lines 1–10 of SELECT is replaced by a base case for $n \ge n_0$, where n_0 is a suitable constant; that g is chosen as $\lfloor r - p + 1)/5 \rfloor$; and that the elements in $A[5g : n]$ belong to no group. Show that although the recurrence for the running time becomes messier, it still solves to $\Theta(n)$.

9.3-3

Show how to use SELECT as a subroutine to make quicksort run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

Figure 9.4 Professor Olay needs to determine the position of the east-west oil pipeline that minimizes the total length of the north-south spurs.

★ ***9.3-4***

Suppose that an algorithm uses only comparisons to find the ith smallest element in a set of n elements. Show that it can also find the $i - 1$ smaller elements and the $n - i$ larger elements without performing any additional comparisons.

9.3-5

Show how to determine the median of a 5-element set using only 6 comparisons.

9.3-6

You have a "black-box" worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

9.3-7

Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. The company wants to connect a spur pipeline from each well directly to the main pipeline along a shortest route (either north or south), as shown in Figure 9.4. Given the x- and y-coordinates of the wells, how should the professor pick an optimal location of the main pipeline to minimize the total length of the spurs? Show how to determine an optimal location in linear time.

9.3-8

The kth ***quantiles*** of an n-element set are the $k - 1$ order statistics that divide the sorted set into k equal-sized sets (to within 1). Give an $O(n \lg k)$-time algorithm to list the kth quantiles of a set.

9.3-9

Describe an $O(n)$-time algorithm that, given a set S of n distinct numbers and a positive integer $k \le n$, determines the k numbers in S that are closest to the median of S.

9.3-10

Let $X[1:n]$ and $Y[1:n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays X and Y. Assume that all $2n$ numbers are distinct.

Problems

9-1 Largest i numbers in sorted order

You are given a set of n numbers, and you wish to find the i largest in sorted order using a comparison-based algorithm. Describe the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of n and i.

a. Sort the numbers, and list the i largest.

b. Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.

c. Use an order-statistic algorithm to find the ith largest number, partition around that number, and sort the i largest numbers.

9-2 Variant of randomized selection

Professor Mendel has proposed simplifying RANDOMIZED-SELECT by eliminating the check for whether i and k are equal. The simplified procedure is SIMPLER-RANDOMIZED-SELECT.

```
SIMPLER-RANDOMIZED-SELECT(A, p, r, i)
if p == r
    return A[p]        // 1 ≤ i ≤ r − p + 1 means that i = 1
q = RANDOMIZED-PARTITION(A, p, r)
k = q − p + 1
if i ≤ k
    return SIMPLER-RANDOMIZED-SELECT(A, p, q, i)
else return SIMPLER-RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

a. Argue that in the worst case, SIMPLER-RANDOMIZED-SELECT never terminates.

b. Prove that the expected running time of SIMPLER-RANDOMIZED-SELECT is still $O(n)$.

9-3 Weighted median

Consider n elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$. The ***weighted (lower) median*** is an element x_k satisfying

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

and

$$\sum_{x_i > x_k} w_i \le \frac{1}{2} .$$

For example, consider the following elements x_i and weights w_i:

i	1	2	3	4	5	6	7
x_i	3	8	2	5	4	1	6
w_i	0.12	0.35	0.025	0.08	0.15	0.075	0.2

For these elements, the median is $x_5 = 4$, but the weighted median is $x_7 = 6$. To see why the weighted median is x_7, observe that the elements less than x_7 are x_1, x_3, x_4, x_5, and x_6, and the sum $w_1 + w_3 + w_4 + w_5 + w_6 = 0.45$, which is less than $1/2$. Furthermore, only element x_2 is greater than x_7, and $w_2 = 0.35$, which is no greater than $1/2$.

a. Argue that the median of $x_1, x_2, \ldots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, 2, \ldots, n$.

b. Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time using sorting.

c. Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as SELECT from Section 9.3.

The ***post-office location problem*** is defined as follows. The input is n points $p_1, p_2, \ldots, p_n$ with associated weights $w_1, w_2, \ldots, w_n$. A solution is a point p (not necessarily one of the input points) that minimizes the sum $\sum_{i=1}^{n} w_i \, d(p, p_i)$, where $d(a, b)$ is the distance between points a and b.

d. Argue that the weighted median is a best solution for the one-dimensional post-office location problem, in which points are simply real numbers and the distance between points a and b is $d(a, b) = |a - b|$.

e. Find the best solution for the two-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the ***Manhattan distance*** given by $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

9-4 Small order statistics

Let's denote by $S(n)$ the worst-case number of comparisons used by SELECT to select the ith order statistic from n numbers. Although $S(n) = \Theta(n)$, the constant hidden by the Θ-notation is rather large. When i is small relative to n, there is an algorithm that uses SELECT as a subroutine but makes fewer comparisons in the worst case.

a. Describe an algorithm that uses $U_i(n)$ comparisons to find the ith smallest of n elements, where

$$U_i(n) = \begin{cases} S(n) & \text{if } i \ge n/2 , \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + S(2i) & \text{otherwise} . \end{cases}$$

(*Hint:* Begin with $\lfloor n/2 \rfloor$ disjoint pairwise comparisons, and recurse on the set containing the smaller element from each pair.)

b. Show that, if $i < n/2$, then $U_i(n) = n + O(S(2i) \lg(n/i))$.

c. Show that if i is a constant less than $n/2$, then $U_i(n) = n + O(\lg n)$.

d. Show that if $i = n/k$ for $k \ge 2$, then $U_i(n) = n + O(S(2n/k) \lg k)$.

9-5 Alternative analysis of randomized selection

In this problem, you will use indicator random variables to analyze the procedure RANDOMIZED-SELECT in a manner akin to our analysis of RANDOMIZED-QUICKSORT in Section 7.4.2.

As in the quicksort analysis, we assume that all elements are distinct, and we rename the elements of the input array A as $z_1, z_2, \ldots, z_n$, where z_i is the ith smallest element. Thus the call RANDOMIZED-SELECT$(A, 1, n, i)$ returns z_i.

For $1 \le j < k \le n$, let

$$X_{ijk} = \mathrm{I}\{z_j \text{ is compared with } z_k \text{ sometime during the execution of the algorithm to find } z_i\} .$$

a. Give an exact expression for $\mathrm{E}[X_{ijk}]$. (*Hint:* Your expression may have different values, depending on the values of i, j, and k.)

b. Let X_i denote the total number of comparisons between elements of array A when finding z_i. Show that

$$\mathrm{E}[X_i] \le 2\left(\sum_{j=1}^{i}\sum_{k=i}^{n}\frac{1}{k-j+1} + \sum_{k=i+1}^{n}\frac{k-i-1}{k-i+1} + \sum_{j=1}^{i-2}\frac{i-j-1}{i-j+1}\right) .$$

c. Show that $\mathrm{E}[X_i] \le 4n$.

d. Conclude that, assuming all elements of array A are distinct, RANDOMIZED-SELECT runs in $O(n)$ expected time.

9-6 Select with groups of 3

Exercise 9.3-1 asks you to show that the SELECT algorithm still runs in linear time if the elements are divided into groups of 7. This problem asks about dividing into groups of 3.

a. Show that SELECT runs in linear time if you divide the elements into groups whose size is any odd constant greater than 3.

b. Show that SELECT runs in $O(n \lg n)$ time if you divide the elements into groups of size 3.

Because the bound in part (b) is just an upper bound, we do not know whether the groups-of-3 strategy actually runs in $O(n)$ time. But by repeating the groups-of-3 idea on the middle group of medians, we can pick a pivot that guarantees $O(n)$ time. The SELECT3 algorithm on the next page determines the ith smallest of an input array of $n > 1$ distinct elements.

c. Describe in English how the SELECT3 algorithm works. Include in your description one or more suitable diagrams.

d. Show that SELECT3 runs in $O(n)$ time in the worst case.

Chapter notes

The worst-case linear-time median-finding algorithm was devised by Blum, Floyd, Pratt, Rivest, and Tarjan [62]. The fast randomized version is due to Hoare [218]. Floyd and Rivest [147] have developed an improved randomized version that partitions around an element recursively selected from a small sample of the elements.

```
SELECT3(A, p, r, i)
1  while (r − p + 1) mod 9 ≠ 0
2      for j = p + 1 to r                    // put the minimum into A[p]
3          if A[p] > A[j]
4              exchange A[p] with A[j]
5      // If we want the minimum of A[p : r], we're done.
6      if i == 1
7          return A[p]
8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
9      p = p + 1
10     i = i − 1
11 g = (r − p + 1)/3                         // number of 3-element groups
12 for j = p to p + g − 1                    // run through the groups
13     sort ⟨A[j], A[j + g], A[j + 2g]⟩ in place
14 // All group medians now lie in the middle third of A[p : r].
15 g′ = g/3                                  // number of 3-element subgroups
16 for j = p + g to p + g + g′ − 1           // sort the subgroups
17     sort ⟨A[j], A[j + g′]A[j + 2g′]⟩ in place
18 // All subgroup medians now lie in the middle ninth of A[p : r].
19 // Find the pivot x recursively as the median of the subgroup medians.
20 x = SELECT3(A, p + 4g′, p + 5g′ − 1, ⌈g′/2⌉)
21 q = PARTITION-AROUND(A, p, r, x)   // partition around the pivot
22 // The rest is just like lines 19–24 of SELECT.
23 k = q − p + 1
24 if i == k
25     return A[q]                           // the pivot value is the answer
26 elseif i < k
27     return SELECT3(A, p, q − 1, i)
28 else return SELECT3(A, q + 1, r, i − k)
```

It is still unknown exactly how many comparisons are needed to determine the median. Bent and John [48] gave a lower bound of $2n$ comparisons for median finding, and Schönhage, Paterson, and Pippenger [397] gave an upper bound of $3n$. Dor and Zwick have improved on both of these bounds. Their upper bound [123] is slightly less than $2.95n$, and their lower bound [124] is $(2 + \epsilon)n$, for a small positive constant ϵ, thereby improving slightly on related work by Dor et al. [122]. Paterson [354] describes some of these results along with other related work.

Problem 9-6 was inspired by a paper by Chen and Dumitrescu [84].

Part III Data Structures

Introduction

Sets are as fundamental to computer science as they are to mathematics. Whereas mathematical sets are unchanging, the sets manipulated by algorithms can grow, shrink, or otherwise change over time. We call such sets ***dynamic***. The next four chapters present some basic techniques for representing finite dynamic sets and manipulating them on a computer.

Algorithms may require several types of operations to be performed on sets. For example, many algorithms need only the ability to insert elements into, delete elements from, and test membership in a set. We call a dynamic set that supports these operations a ***dictionary***. Other algorithms require more complicated operations. For example, min-priority queues, which Chapter 6 introduced in the context of the heap data structure, support the operations of inserting an element into and extracting the smallest element from a set. The best way to implement a dynamic set depends upon the operations that you need to support.

Elements of a dynamic set

In a typical implementation of a dynamic set, each element is represented by an object whose attributes can be examined and manipulated given a pointer to the object. Some kinds of dynamic sets assume that one of the object's attributes is an identifying ***key***. If the keys are all different, we can think of the dynamic set as being a set of key values. The object may contain ***satellite data***, which are carried around in other object attributes but are otherwise unused by the set implementation. It may also have attributes that are manipulated by the set operations. These attributes may contain data or pointers to other objects in the set.

Some dynamic sets presuppose that the keys are drawn from a totally ordered set, such as the real numbers, or the set of all words under the usual alphabetic

ordering. A total ordering allows us to define the minimum element of the set, for example, or to speak of the next element larger than a given element in a set.

Operations on dynamic sets

Operations on a dynamic set can be grouped into two categories: ***queries***, which simply return information about the set, and ***modifying operations***, which change the set. Here is a list of typical operations. Any specific application will usually require only a few of these to be implemented.

SEARCH(S, k)
: A query that, given a set S and a key value k, returns a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S.

INSERT(S, x)
: A modifying operation that adds the element pointed to by x to the set S. We usually assume that any attributes in element x needed by the set implementation have already been initialized.

DELETE(S, x)
: A modifying operation that, given a pointer x to an element in the set S, removes x from S. (Note that this operation takes a pointer to an element x, not a key value.)

MINIMUM(S) and MAXIMUM(S)
: Queries on a totally ordered set S that return a pointer to the element of S with the smallest (for MINIMUM) or largest (for MAXIMUM) key.

SUCCESSOR(S, x)
: A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

PREDECESSOR(S, x)
: A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or NIL if x is the minimum element.

In some situations, we can extend the queries SUCCESSOR and PREDECESSOR so that they apply to sets with nondistinct keys. For a set on n keys, the normal presumption is that a call to MINIMUM followed by $n - 1$ calls to SUCCESSOR enumerates the elements in the set in sorted order.

We usually measure the time taken to execute a set operation in terms of the size of the set. For example, Chapter 13 describes a data structure that can support any of the operations listed above on a set of size n in $O(\lg n)$ time.

Of course, you can always choose to implement a dynamic set with an array. The advantage of doing so is that the algorithms for the dynamic-set operations are simple. The downside, however, is that many of these operations have a worst-case running time of $\Theta(n)$. If the array is not sorted, INSERT and DELETE can take $\Theta(1)$ time, but the remaining operations take $\Theta(n)$ time. If instead the array is maintained in sorted order, then MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR take $\Theta(1)$ time; SEARCH takes $O(\lg n)$ time if implemented with binary search; but INSERT and DELETE take $\Theta(n)$ time in the worst case. The data structures studied in this part improve on the array implementation for many of the dynamic-set operations.

Overview of Part III

Chapters 10–13 describe several data structures that we can use to implement dynamic sets. We'll use many of these data structures later to construct efficient algorithms for a variety of problems. We already saw another important data structure —the heap—in Chapter 6.

Chapter 10 presents the essentials of working with simple data structures such as arrays, matrices, stacks, queues, linked lists, and rooted trees. If you have taken an introductory programming course, then much of this material should be familiar to you.

Chapter 11 introduces hash tables, a widely used data structure supporting the dictionary operations INSERT, DELETE, and SEARCH. In the worst case, hash tables require $\Theta(n)$ time to perform a SEARCH operation, but the expected time for hash-table operations is $O(1)$. We rely on probability to analyze hash-table operations, but you can understand how the operations work even without probability.

Binary search trees, which are covered in Chapter 12, support all the dynamic-set operations listed above. In the worst case, each operation takes $\Theta(n)$ time on a tree with n elements. Binary search trees serve as the basis for many other data structures.

Chapter 13 introduces red-black trees, which are a variant of binary search trees. Unlike ordinary binary search trees, red-black trees are guaranteed to perform well: operations take $O(\lg n)$ time in the worst case. A red-black tree is a balanced search tree. Chapter 18 in Part V presents another kind of balanced search tree, called a B-tree. Although the mechanics of red-black trees are somewhat intricate, you can glean most of their properties from the chapter without studying the mechanics in detail. Nevertheless, you probably will find walking through the code to be instructive.

10 Elementary Data Structures

In this chapter, we examine the representation of dynamic sets by simple data structures that use pointers. Although you can construct many complex data structures using pointers, we present only the rudimentary ones: arrays, matrices, stacks, queues, linked lists, and rooted trees.

10.1 Simple array-based data structures: arrays, matrices, stacks, queues

10.1.1 Arrays

We assume that, as in most programming languages, an array is stored as a contiguous sequence of bytes in memory. If the first element of an array has index s (for example, in an array with 1-origin indexing, $s = 1$), the array starts at memory address a, and each array element occupies b bytes, then the ith element occupies bytes $a + b(i - s)$ through $a + b(i - s + 1) - 1$. Since most of the arrays in this book are indexed starting at 1, and a few starting at 0, we can simplify these formulas a little. When $s = 1$, the ith element occupies bytes $a + b(i - 1)$ through $a + bi - 1$, and when $s = 0$, the ith element occupies bytes $a + bi$ through $a + b(i + 1) - 1$. Assuming that the computer can access all memory locations in the same amount of time (as in the RAM model described in Section 2.2), it takes constant time to access any array element, regardless of the index.

Most programming languages require each element of a particular array to be the same size. If the elements of a given array might occupy different numbers of bytes, then the above formulas fail to apply, since the element size b is not a constant. In such cases, the array elements are usually objects of varying sizes, and what actually appears in each array element is a pointer to the object. The number of bytes occupied by a pointer is typically the same, no matter what the pointer references, so that to access an object in an array, the above formulas give the address of the pointer to the object and then the pointer must be followed to access the object itself.

Figure 10.1 Four ways to store the 2×3 matrix M from equation (10.1). **(a)** In row-major order, in a single array. **(b)** In column-major order, in a single array. **(c)** In row-major order, with one array per row (tan) and a single array (blue) of pointers to the row arrays. **(d)** In column-major order, with one array per column (tan) and a single array (blue) of pointers to the column arrays.

10.1.2 Matrices

We typically represent a matrix or two-dimensional array by one or more one-dimensional arrays. The two most common ways to store a matrix are row-major and column-major order. Let's consider an $m \times n$ matrix—a matrix with m rows and n columns. In ***row-major order***, the matrix is stored row by row, and in ***column-major order***, the matrix is stored column by column. For example, consider the 2×3 matrix

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}. \tag{10.1}$$

Row-major order stores the two rows 1 2 3 and 4 5 6, whereas column-major order stores the three columns 1 4; 2 5; and 3 6.

Parts (a) and (b) of Figure 10.1 show how to store this matrix using a single one-dimensional array. It's stored in row-major order in part (a) and in column-major order in part (b). If the rows, columns, and the single array all are indexed starting at s, then $M[i, j]$—the element in row i and column j—is at array index $s + (n(i - s)) + (j - s)$ with row-major order and $s + (m(j - s)) + (i - s)$ with column-major order. When $s = 1$, the single-array indices are $n(i - 1) + j$ with row-major order and $i + m(j - 1)$ with column-major order. When $s = 0$, the single-array indices are simpler: $ni + j$ with row-major order and $i + mj$ with column-major order. For the example matrix M with 1-origin indexing, element $M[2, 1]$ is stored at index $3(2-1)+1 = 4$ in the single array using row-major order and at index $2 + 2(1 - 1) = 2$ using column-major order.

Parts (c) and (d) of Figure 10.1 show multiple-array strategies for storing the example matrix. In part (c), each row is stored in its own array of length n, shown in tan. Another array, with m elements, shown in blue, points to the m row arrays. If we call the blue array A, then $A[i]$ points to the array storing the entries for row i of M, and array element $A[i][j]$ stores matrix element $M[i, j]$. Part (d) shows the column-major version of the multiple-array representation, with n arrays, each of

length m, representing the n columns. Matrix element $M[i, j]$ is stored in array element $A[j][i]$.

Single-array representations are typically more efficient on modern machines than multiple-array representations. But multiple-array representations can sometimes be more flexible, for example, allowing for "ragged arrays," in which the rows in the row-major version may have different lengths, or symmetrically for the column-major version, where columns may have different lengths.

Occasionally, other schemes are used to store matrices. In the ***block representation***, the matrix is divided into blocks, and each block is stored contiguously. For example, a 4×4 matrix that is divided into 2×2 blocks, such as

$$\left(\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right)$$

might be stored in a single array in the order $\langle 1, 2, 5, 6, 3, 4, 7, 8, 9, 10, 13, 14, 11, 12, 15, 16\rangle$.

10.1.3 Stacks and queues

Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a ***stack***, the element deleted from the set is the one most recently inserted: the stack implements a ***last-in, first-out***, or ***LIFO***, policy. Similarly, in a ***queue***, the element deleted is always the one that has been in the set for the longest time: the queue implements a ***first-in, first-out***, or ***FIFO***, policy. There are several efficient ways to implement stacks and queues on a computer. Here, you will see how to use an array with attributes to store them.

Stacks

The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP. These names are allusions to physical stacks, such as the spring-loaded stacks of plates used in cafeterias. The order in which plates are popped from the stack is the reverse of the order in which they were pushed onto the stack, since only the top plate is accessible.

Figure 10.2 shows how to implement a stack of at most n elements with an array $S[1:n]$. The stack has attributes $S.top$, indexing the most recently inserted element, and $S.size$, equaling the size n of the array. The stack consists of elements $S[1:S.top]$, where $S[1]$ is the element at the bottom of the stack and $S[S.top]$ is the element at the top.

Figure 10.2 An array implementation of a stack S. Stack elements appear only in the tan positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack. The top is element 17.

When $S.top = 0$, the stack contains no elements and is ***empty***. We can test whether the stack is empty with the query operation STACK-EMPTY. Upon an attempt to pop an empty stack, the stack ***underflows***, which is normally an error. If $S.top$ exceeds $S.size$, the stack ***overflows***.

The procedures STACK-EMPTY, PUSH, and POP implement each of the stack operations with just a few lines of code. Figure 10.2 shows the effects of the modifying operations PUSH and POP. Each of the three stack operations takes $O(1)$ time.

```
STACK-EMPTY(S)
if S.top == 0
    return TRUE
else return FALSE

PUSH(S, x)
if S.top == S.size
    error "overflow"
else S.top = S.top + 1
    S[S.top] = x

POP(S)
if STACK-EMPTY(S)
    error "underflow"
else S.top = S.top - 1
    return S[S.top + 1]
```

Figure 10.3 A queue implemented using an array $Q[1:12]$. Queue elements appear only in the tan positions. **(a)** The queue has 5 elements, in locations $Q[7:11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

Queues

We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE. Like the stack operation POP, DEQUEUE takes no element argument. The FIFO property of a queue causes it to operate like a line of customers waiting for service. The queue has a ***head*** and a ***tail***. When an element is enqueued, it takes its place at the tail of the queue, just as a newly arriving customer takes a place at the end of the line. The element dequeued is always the one at the head of the queue, like the customer at the head of the line, who has waited the longest.

Figure 10.3 shows one way to implement a queue of at most $n - 1$ elements using an array $Q[1:n]$, with the attribute $Q.size$ equaling the size n of the array. The queue has an attribute $Q.head$ that indexes, or points to, its head. The attribute $Q.tail$ indexes the next location at which a newly arriving element will be inserted into the queue. The elements in the queue reside in locations $Q.head$, $Q.head + 1$, $\ldots$, $Q.tail - 1$, where we "wrap around" in the sense that location 1 immediately follows location n in a circular order. When $Q.head = Q.tail$, the queue is empty. Initially, we have $Q.head = Q.tail = 1$. An attempt to dequeue an element from an empty queue causes the queue to underflow. When $Q.head = Q.tail + 1$ or both

$Q.head = 1$ and $Q.tail = Q.size$, the queue is full, and an attempt to enqueue an element causes the queue to overflow.

In the procedures ENQUEUE and DEQUEUE, we have omitted the error checking for underflow and overflow. (Exercise 10.1-5 asks you to supply these checks.) Figure 10.3 shows the effects of the ENQUEUE and DEQUEUE operations. Each operation takes $O(1)$ time.

```
ENQUEUE(Q, x)
Q[Q.tail] = x
if Q.tail == Q.size
    Q.tail = 1
else Q.tail = Q.tail + 1

DEQUEUE(Q)
x = Q[Q.head]
if Q.head == Q.size
    Q.head = 1
else Q.head = Q.head + 1
return x
```

Exercises

10.1-1

Consider an $m \times n$ matrix in row-major order, where both m and n are powers of 2 and rows and columns are indexed from 0. We can represent a row index i in binary by the $\lg m$ bits $\langle i_{\lg m-1}, i_{\lg m-2}, \ldots, i_0 \rangle$ and a column index j in binary by the $\lg n$ bits $\langle j_{\lg n-1}, j_{\lg n-2}, \ldots, j_0 \rangle$. Suppose that this matrix is a 2×2 block matrix, where each block has $m/2$ rows and $n/2$ columns, and it is to be represented by a single array with 0-origin indexing. Show how to construct the binary representation of the $(\lg m + \lg n)$-bit index into the single array from the binary representations of i and j.

10.1-2

Using Figure 10.2 as a model, illustrate the result of each operation in the sequence PUSH$(S, 4)$, PUSH$(S, 1)$, PUSH$(S, 3)$, POP(S), PUSH$(S, 8)$, and POP(S) on an initially empty stack S stored in array $S[1:6]$

10.1-3
Explain how to implement two stacks in one array $A[1:n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in $O(1)$ time.

10.1-4
Using Figure 10.3 as a model, illustrate the result of each operation in the sequence ENQUEUE$(Q, 4)$, ENQUEUE$(Q, 1)$, ENQUEUE$(Q, 3)$, DEQUEUE(Q), ENQUEUE$(Q, 8)$, and DEQUEUE(Q) on an initially empty queue Q stored in array $Q[1:6]$.

10.1-5
Rewrite ENQUEUE and DEQUEUE to detect underflow and overflow of a queue.

10.1-6
Whereas a stack allows insertion and deletion of elements at only one end, and a queue allows insertion at one end and deletion at the other end, a ***deque*** (double-ended queue, pronounced like "deck") allows insertion and deletion at both ends. Write four $O(1)$-time procedures to insert elements into and delete elements from both ends of a deque implemented by an array.

10.1-7
Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

10.1-8
Show how to implement a stack using two queues. Analyze the running time of the stack operations.

10.2 Linked lists

A ***linked list*** is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object. Since the elements of linked lists often contain keys that can be searched for, linked lists are sometimes called ***search lists***. Linked lists provide a simple, flexible representation for dynamic sets, supporting (though not necessarily efficiently) all the operations listed on page 250.

As shown in Figure 10.4, each element of a ***doubly linked list*** L is an object with an attribute *key* and two pointer attributes: *next* and *prev*. The object may

Figure 10.4 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. **(b)** Following the execution of LIST-PREPEND(L, x), where $x.key = 25$, the linked list has an object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of calling LIST-INSERT(x, y), where $x.key = 36$ and y points to the object with key 9. **(d)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

also contain other satellite data. Given an element x in the list, $x.next$ points to its successor in the linked list, and $x.prev$ points to its predecessor. If $x.prev = \text{NIL}$, the element x has no predecessor and is therefore the first element, or ***head***, of the list. If $x.next = \text{NIL}$, the element x has no successor and is therefore the last element, or ***tail***, of the list. An attribute $L.head$ points to the first element of the list. If $L.head = \text{NIL}$, the list is empty.

A list may have one of several forms. It may be either singly linked or doubly linked, it may be sorted or not, and it may be circular or not. If a list is ***singly linked***, each element has a *next* pointer but not a *prev* pointer. If a list is ***sorted***, the linear order of the list corresponds to the linear order of keys stored in elements of the list. The minimum element is then the head of the list, and the maximum element is the tail. If the list is ***unsorted***, the elements can appear in any order. In a ***circular list***, the *prev* pointer of the head of the list points to the tail, and the *next* pointer of the tail of the list points to the head. You can think of a circular list as a ring of elements. In the remainder of this section, we assume that the lists we are working with are unsorted and doubly linked.

Searching a linked list

The procedure LIST-SEARCH(L, k) finds the first element with key k in list L by a simple linear search, returning a pointer to this element. If no object with key k appears in the list, then the procedure returns NIL. For the linked list in Figure 10.4(a), the call LIST-SEARCH$(L, 4)$ returns a pointer to the third element, and the call LIST-SEARCH$(L, 7)$ returns NIL. To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

```
LIST-SEARCH(L, k)
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

Inserting into a linked list

Given an element x whose *key* attribute has already been set, the LIST-PREPEND procedure adds x to the front of the linked list, as shown in Figure 10.4(b). (Recall that our attribute notation can cascade, so that $L.head.prev$ denotes the *prev* attribute of the object that $L.head$ points to.) The running time for LIST-PREPEND on a list of n elements is $O(1)$.

```
LIST-PREPEND(L, x)
x.next = L.head
x.prev = NIL
if L.head ≠ NIL
    L.head.prev = x
L.head = x
```

You can insert anywhere within a linked list. As Figure 10.4(c) shows, if you have a pointer y to an object in the list, the LIST-INSERT procedure on the facing page "splices" a new element x into the list, immediately following y, in $O(1)$ time. Since LIST-INSERT never references the list object L, it is not supplied as a parameter.

```
LIST-INSERT(x, y)
x.next = y.next
x.prev = y
if y.next ≠ NIL
    y.next.prev = x
y.next = x
```

Deleting from a linked list

The procedure LIST-DELETE removes an element x from a linked list L. It must be given a pointer to x, and it then "'splices" x out of the list by updating pointers. To delete an element with a given key, first call LIST-SEARCH to retrieve a pointer to the element. Figure 10.4(d) shows how an element is deleted from a linked list. LIST-DELETE runs in $O(1)$ time, but to delete an element with a given key, the call to LIST-SEARCH makes the worst-case running time be $\Theta(n)$.

```
LIST-DELETE(L, x)
if x.prev ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    x.next.prev = x.prev
```

Insertion and deletion are faster operations on doubly linked lists than on arrays. If you want to insert a new first element into an array or delete the first element in an array, maintaining the relative order of all the existing elements, then each of the existing elements needs to be moved by one position. In the worst case, therefore, insertion and deletion take $\Theta(n)$ time in an array, compared with $O(1)$ time for a doubly linked list. (Exercise 10.2-1 asks you to show that deleting an element from a singly linked list takes $\Theta(n)$ time in the worst case.) If, however, you want to find the kth element in the linear order, it takes just $O(1)$ time in an array regardless of k, but in a linked list, you'd have to traverse k elements, taking $\Theta(k)$ time.

Sentinels

The code for LIST-DELETE is simpler if you ignore the boundary conditions at the head and tail of the list:

Figure 10.5 A circular, doubly linked list with a sentinel. The sentinel $L.nil$, in blue, appears between the head and tail. The attribute $L.head$ is no longer needed, since the head of the list is $L.nil.next$. **(a)** An empty list. **(b)** The linked list from Figure 10.4(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT$'(x, L.nil)$, where $x.key = 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4. **(e)** The list after executing LIST-INSERT$'(x, y)$, where $x.key = 36$ and y points to the object with key 9.

```
LIST-DELETE'(x)
x.prev.next = x.next
x.next.prev = x.prev
```

A ***sentinel*** is a dummy object that allows us to simplify boundary conditions. In a linked list L, the sentinel is an object $L.nil$ that represents NIL but has all the attributes of the other objects in the list. References to NIL are replaced by references to the sentinel $L.nil$. As shown in Figure 10.5, this change turns a regular doubly linked list into a ***circular, doubly linked list with a sentinel***, in which the sentinel $L.nil$ lies between the head and tail. The attribute $L.nil.next$ points to the head of the list, and $L.nil.prev$ points to the tail. Similarly, both the *next* attribute of the tail and the *prev* attribute of the head point to $L.nil$. Since $L.nil.next$ points to the head, the attribute $L.head$ is eliminated altogether, with references to it replaced by references to $L.nil.next$. Figure 10.5(a) shows that an empty list consists of just the sentinel, and both $L.nil.next$ and $L.nil.prev$ point to $L.nil$.

To delete an element from the list, just use the two-line procedure LIST-DELETE$'$ from before. Just as LIST-INSERT never references the list object L, neither does

LIST-DELETE′. You should never delete the sentinel $L.nil$ unless you are deleting the entire list!

The LIST-INSERT′ procedure inserts an element x into the list following object y. No separate procedure for prepending is necessary: to insert at the head of the list, let y be $L.nil$; and to insert at the tail, let y be $L.nil.prev$. Figure 10.5 shows the effects of LIST-INSERT′ and LIST-DELETE′ on a sample list.

```
LIST-INSERT′(x, y)
x.next = y.next
x.prev = y
y.next.prev = x
y.next = x
```

Searching a circular, doubly linked list with a sentinel has the same asymptotic running time as without a sentinel, but it is possible to decrease the constant factor. The test in line 2 of LIST-SEARCH makes two comparisons: one to check whether the search has run off the end of the list and, if not, one to check whether the key resides in the current element x. Suppose that you *know* that the key is somewhere in the list. Then you do not need to check whether the search runs off the end of the list, thereby eliminating one comparison in each iteration of the **while** loop.

The sentinel provides a place to put the key before starting the search. The search starts at the head $L.nil.next$ of list L, and it stops if it finds the key somewhere in the list. Now the search is guaranteed to find the key, either in the sentinel or before reaching the sentinel. If the key is found before reaching the sentinel, then it really is in the element where the search stops. If, however, the search goes through all the elements in the list and finds the key only in the sentinel, then the key is not really in the list, and the search returns NIL. The procedure LIST-SEARCH′ embodies this idea. (If your sentinel requires its *key* attribute to be NIL, then you might want to assign $L.nil.key$ = NIL before line 5.)

```
LIST-SEARCH′(L, k)
L.nil.key = k          // store the key in the sentinel to guarantee it is in list
x = L.nil.next         // start at the head of the list
while x.key ≠ k
    x = x.next
if x == L.nil          // found k in the sentinel
    return NIL         // k was not really in the list
else return x          // found k in element x
```

Sentinels often simplify code and, as in searching a linked list, they might speed up code by a small constant factor, but they don't typically improve the asymptotic running time. Use them judiciously. When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory. In this book, we use sentinels only when they significantly simplify the code.

Exercises

10.2-1
Explain why the dynamic-set operation INSERT on a singly linked list can be implemented in $O(1)$ time, but the worst-case time for DELETE is $\Theta(n)$.

10.2-2
Implement a stack using a singly linked list. The operations PUSH and POP should still take $O(1)$ time. Do you need to add any attributes to the list?

10.2-3
Implement a queue using a singly linked list. The operations ENQUEUE and DEQUEUE should still take $O(1)$ time. Do you need to add any attributes to the list?

10.2-4
The dynamic-set operation UNION takes two disjoint sets S_1 and S_2 as input, and it returns a set $S = S_1 \cup S_2$ consisting of all the elements of S_1 and S_2. The sets S_1 and S_2 are usually destroyed by the operation. Show how to support UNION in $O(1)$ time using a suitable list data structure.

10.2-5
Give a $\Theta(n)$-time nonrecursive procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

★ ***10.2-6***
Explain how to implement doubly linked lists using only one pointer value $x.np$ per item instead of the usual two (*next* and *prev*). Assume that all pointer values can be interpreted as k-bit integers, and define $x.np = x.next$ XOR $x.prev$, the k-bit "exclusive-or" of $x.next$ and $x.prev$. The value NIL is represented by 0. Be sure to describe what information you need to access the head of the list. Show how to implement the SEARCH, INSERT, and DELETE operations on such a list. Also show how to reverse such a list in $O(1)$ time.

10.3 Representing rooted trees

Linked lists work well for representing linear relationships, but not all relationships are linear. In this section, we look specifically at the problem of representing rooted trees by linked data structures. We first look at binary trees, and then we present a method for rooted trees in which nodes can have an arbitrary number of children.

We represent each node of a tree by an object. As with linked lists, we assume that each node contains a *key* attribute. The remaining attributes of interest are pointers to other nodes, and they vary according to the type of tree.

Binary trees

Figure 10.6 shows how to use the attributes p, *left*, and *right* to store pointers to the parent, left child, and right child of each node in a binary tree T. If $x.p = \text{NIL}$, then x is the root. If node x has no left child, then $x.left = \text{NIL}$, and similarly for the right child. The root of the entire tree T is pointed to by the attribute $T.root$. If $T.root = \text{NIL}$, then the tree is empty.

Rooted trees with unbounded branching

It's simple to extend the scheme for representing a binary tree to any class of trees in which the number of children of each node is at most some constant k: replace the *left* and *right* attributes by $child_1, child_2, \ldots, child_k$. This scheme no longer works when the number of children of a node is unbounded, however, since we do not know how many attributes to allocate in advance. Moreover, if k, the number of children, is bounded by a large constant but most nodes have a small number of children, we may waste a lot of memory.

Fortunately, there is a clever scheme to represent trees with arbitrary numbers of children. It has the advantage of using only $O(n)$ space for any n-node rooted tree. The ***left-child, right-sibling representation*** appears in Figure 10.7. As before, each node contains a parent pointer p, and $T.root$ points to the root of tree T. Instead of having a pointer to each of its children, however, each node x has only two pointers:

1. $x.left\text{-}child$ points to the leftmost child of node x, and
2. $x.right\text{-}sibling$ points to the sibling of x immediately to its right.

If node x has no children, then $x.left\text{-}child = \text{NIL}$, and if node x is the rightmost child of its parent, then $x.right\text{-}sibling = \text{NIL}$.

Figure 10.6 The representation of a binary tree T. Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The *key* attributes are not shown.

Figure 10.7 The left-child, right-sibling representation of a tree T. Each node x has attributes $x.p$ (top), $x.left\text{-}child$ (lower left), and $x.right\text{-}sibling$ (lower right). The *key* attributes are not shown.

Other tree representations

We sometimes represent rooted trees in other ways. In Chapter 6, for example, we represented a heap, which is based on a complete binary tree, by a single array along with an attribute giving the index of the last node in the heap. The trees that appear in Chapter 19 are traversed only toward the root, and so only the parent pointers are present: there are no pointers to children. Many other schemes are possible. Which scheme is best depends on the application.

Exercises

10.3-1
Draw the binary tree rooted at index 6 that is represented by the following attributes:

index	*key*	*left*	*right*
1	17	8	9
2	14	NIL	NIL
3	12	NIL	NIL
4	20	10	NIL
5	33	2	NIL
6	15	1	4
7	28	NIL	NIL
8	22	NIL	NIL
9	13	3	7
10	25	NIL	5

10.3-2
Write an $O(n)$-time recursive procedure that, given an n-node binary tree, prints out the key of each node in the tree.

10.3-3
Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

10.3-4
Write an $O(n)$-time procedure that prints out all the keys of an arbitrary rooted tree with n nodes, where the tree is stored using the left-child, right-sibling representation.

★ ***10.3-5***
Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node. Use no more than constant extra space outside

of the tree itself and do not modify the tree, even temporarily, during the procedure.

★ ***10.3-6***
The left-child, right-sibling representation of an arbitrary rooted tree uses three pointers in each node: *left-child*, *right-sibling*, and *parent*. From any node, its parent can be accessed in constant time and all its children can be accessed in time linear in the number of children. Show how to use only two pointers and one boolean value in each node x so that x's parent or all of x's children can be accessed in time linear in the number of x's children.

Problems

10-1 Comparisons among lists
For each of the four types of lists in the following table, what is the asymptotic worst-case running time for each dynamic-set operation listed?

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
SEARCH				
INSERT				
DELETE				
SUCCESSOR				
PREDECESSOR				
MINIMUM				
MAXIMUM				

10-2 Mergeable heaps using linked lists
A ***mergeable heap*** supports the following operations: MAKE-HEAP (which creates an empty mergeable heap), INSERT, MINIMUM, EXTRACT-MIN, and UNION.[1]

[1] Because we have defined a mergeable heap to support MINIMUM and EXTRACT-MIN, we can also refer to it as a ***mergeable min-heap***. Alternatively, if it supports MAXIMUM and EXTRACT-MAX, it is a ***mergeable max-heap***.

Show how to implement mergeable heaps using linked lists in each of the following cases. Try to make each operation as efficient as possible. Analyze the running time of each operation in terms of the size of the dynamic set(s) being operated on.

a. Lists are sorted.

b. Lists are unsorted.

c. Lists are unsorted, and dynamic sets to be merged are disjoint.

10-3 Searching a sorted compact list

We can represent a singly linked list with two arrays, *key* and *next*. Given the index i of an element, its value is stored in $key[i]$, and the index of its successor is given by $next[i]$, where $next[i] = \text{NIL}$ for the last element. We also need the index *head* of the first element in the list. An n-element list stored in this way is ***compact*** if it is stored only in positions 1 through n of the *key* and *next* arrays.

Let's assume that all keys are distinct and that the compact list is also sorted, that is, $key[i] < key[next[i]]$ for all $i = 1, 2, \ldots, n$ such that $next[i] \neq \text{NIL}$. Under these assumptions, you will show that the randomized algorithm COMPACT-LIST-SEARCH searches the list for key k in $O(\sqrt{n})$ expected time.

```
COMPACT-LIST-SEARCH(key, next, head, n, k)
 1  i = head
 2  while i ≠ NIL and key[i] < k
 3      j = RANDOM(1, n)
 4      if key[i] < key[j] and key[j] ≤ k
 5          i = j
 6          if key[i] == k
 7              return i
 8      i = next[i]
 9  if i == NIL or key[i] > k
10      return NIL
11  else return i
```

If you ignore lines 3–7 of the procedure, you can see that it's an ordinary algorithm for searching a sorted linked list, in which index i points to each position of the list in turn. The search terminates once the index i "falls off" the end of the list or once $key[i] \geq k$. In the latter case, if $key[i] = k$, the procedure has found a key with the value k. If, however, $key[i] > k$, then the search will never find a key with the value k, so that terminating the search was the correct action.

Lines 3–7 attempt to skip ahead to a randomly chosen position j. Such a skip helps if $key[j]$ is larger than $key[i]$ and no larger than k. In such a case, j marks a position in the list that i would reach during an ordinary list search. Because the list is compact, we know that any choice of j between 1 and n indexes some element in the list.

Instead of analyzing the performance of COMPACT-LIST-SEARCH directly, you will analyze a related algorithm, COMPACT-LIST-SEARCH′, which executes two separate loops. This algorithm takes an additional parameter t, which specifies an upper bound on the number of iterations of the first loop.

```
COMPACT-LIST-SEARCH′(key, next, head, n, k, t)
1  i = head
2  for q = 1 to t
3      j = RANDOM(1, n)
4      if key[i] < key[j] and key[j] ≤ k
5          i = j
6          if key[i] == k
7              return i
8  while i ≠ NIL and key[i] < k
9      i = next[i]
10 if i == NIL or key[i] > k
11     return NIL
12 else return i
```

To compare the execution of the two algorithms, assume that the sequence of calls of RANDOM$(1, n)$ yields the same sequence of integers for both algorithms.

a. Argue that for any value of t, COMPACT-LIST-SEARCH$(key, next, head, n, k)$ and COMPACT-LIST-SEARCH′$(key, next, head, n, k, t)$ return the same result and that the number of iterations of the **while** loop of lines 2–8 in COMPACT-LIST-SEARCH is at most the total number of iterations of both the **for** and **while** loops in COMPACT-LIST-SEARCH′.

In the call COMPACT-LIST-SEARCH′$(key, next, head, n, k, t)$, let X_t be the random variable that describes the distance in the linked list (that is, through the chain of *next* pointers) from position i to the desired key k after t iterations of the **for** loop of lines 2–7 have occurred.

b. Argue that COMPACT-LIST-SEARCH′$(key, next, head, n, k, t)$ has an expected running time of $O(t + \mathrm{E}[X_t])$.

c. Show that $\mathrm{E}[X_t] = \sum_{r=1}^{n} (1 - r/n)^t$. (*Hint:* Use equation (C.28) on page 1193.)

d. Show that $\sum_{r=0}^{n-1} r^t \leq n^{t+1}/(t+1)$. (*Hint:* Use inequality (A.18) on page 1150.)

e. Prove that $\mathrm{E}[X_t] \leq n/(t+1)$.

f. Show that COMPACT-LIST-SEARCH$'(key, next, head, n, k, t)$ has an expected running time of $O(t + n/t)$.

g. Conclude that COMPACT-LIST-SEARCH runs in $O(\sqrt{n})$ expected time.

h. Why do we assume that all keys are distinct in COMPACT-LIST-SEARCH? Argue that random skips do not necessarily help asymptotically when the list contains repeated key values.

Chapter notes

Aho, Hopcroft, and Ullman [6] and Knuth [259] are excellent references for elementary data structures. Many other texts cover both basic data structures and their implementation in a particular programming language. Examples of these types of textbooks include Goodrich and Tamassia [196], Main [311], Shaffer [406], and Weiss [452, 453, 454]. The book by Gonnet and Baeza-Yates [193] provides experimental data on the performance of many data-structure operations.

The origin of stacks and queues as data structures in computer science is unclear, since corresponding notions already existed in mathematics and paper-based business practices before the introduction of digital computers. Knuth [259] cites A. M. Turing for the development of stacks for subroutine linkage in 1947.

Pointer-based data structures also seem to be a folk invention. According to Knuth, pointers were apparently used in early computers with drum memories. The A-1 language developed by G. M. Hopper in 1951 represented algebraic formulas as binary trees. Knuth credits the IPL-II language, developed in 1956 by A. Newell, J. C. Shaw, and H. A. Simon, for recognizing the importance and promoting the use of pointers. Their IPL-III language, developed in 1957, included explicit stack operations.

11 Hash Tables

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. For example, a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifiers in the language. A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list—$\Theta(n)$ time in the worst case—in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$. Indeed, the built-in dictionaries of Python are implemented with hash tables.

A hash table generalizes the simpler notion of an ordinary array. Directly addressing into an ordinary array takes advantage of the $O(1)$ access time for any array element. Section 11.1 discusses direct addressing in more detail. To use direct addressing, you must be able to allocate an array that contains a position for every possible key.

When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, we *compute* the array index from the key. Section 11.2 presents the main ideas, focusing on "chaining" as a way to handle "collisions," in which more than one key maps to the same array index. Section 11.3 describes how to compute array indices from keys using hash functions. We present and analyze several variations on the basic theme. Section 11.4 looks at "open addressing," which is another way to deal with collisions. The bottom line is that hashing is an extremely effective and practical technique: the basic dictionary operations require only $O(1)$ time on the average. Section 11.5 discusses the hierarchical memory systems of modern computer systems have and illustrates how to design hash tables that work well in such systems.

11.1 Direct-address tables

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a distinct key drawn from the universe $U = \{0, 1, \ldots, m-1\}$, where m is not too large.

To represent the dynamic set, you can use an array, or ***direct-address table***, denoted by $T[0:m-1]$, in which each position, or ***slot***, corresponds to a key in the universe U. Figure 11.1 illustrates this approach. Slot k points to an element in the set with key k. If the set contains no element with key k, then $T[k] = \text{NIL}$.

The dictionary operations DIRECT-ADDRESS-SEARCH, DIRECT-ADDRESS-INSERT, and DIRECT-ADDRESS-DELETE on the following page are trivial to implement. Each takes only $O(1)$ time.

For some applications, the direct-address table itself can hold the elements in the dynamic set. That is, rather than storing an element's key and satellite data in an object external to the direct-address table, with a pointer from a slot in the table to the object, save space by storing the object directly in the slot. To indicate an empty slot, use a special key. Then again, why store the key of the object at all? The index of the object *is* its key! Of course, then you'd need some way to tell whether slots are empty.

Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index into the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, in blue, contain NIL.

```
DIRECT-ADDRESS-SEARCH(T, k)
return T[k]

DIRECT-ADDRESS-INSERT(T, x)
T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
T[x.key] = NIL
```

Exercises

11.1-1
A dynamic set S is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?

11.1-2
A ***bit vector*** is simply an array of bits (each either 0 or 1). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements drawn from the set $\{0, 1, \ldots, m - 1\}$ and with no satellite data. Dictionary operations should run in $O(1)$ time.

11.1-3
Suggest how to implement a direct-address table in which the keys of stored elements do not need to be distinct and the elements can have satellite data. All three dictionary operations (INSERT, DELETE, and SEARCH) should run in $O(1)$ time. (Don't forget that DELETE takes as an argument a pointer to an object to be deleted, not a key.)

★ ***11.1-4***
Suppose that you want to implement a dictionary by using direct addressing on a *huge* array. That is, if the array size is m and the dictionary contains at most n elements at any one time, then $m \gg n$. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations SEARCH, INSERT, and DELETE should take $O(1)$ time each; and initializing the data structure should take $O(1)$ time. (*Hint:* Use an additional array, treated somewhat like a stack whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

11.2 Hash tables

The downside of direct addressing is apparent: if the universe U is large or infinite, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys *actually stored* may be so small relative to U that most of the space allocated for T would be wasted.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct-address table. Specifically, the storage requirement reduces to $\Theta(|K|)$ while maintaining the benefit that searching for an element in the hash table still requires only $O(1)$ time. The catch is that this bound is for the *average-case time*,[1] whereas for direct addressing it holds for the *worst-case time*.

With direct addressing, an element with key k is stored in slot k, but with hashing, we use a ***hash function*** h to compute the slot number from the key k, so that the element goes into slot $h(k)$. The hash function h maps the universe U of keys into the slots of a ***hash table*** $T[0:m-1]$:

$$h : U \to \{0, 1, \ldots, m-1\} ,$$

where the size m of the hash table is typically much less than $|U|$. We say that an element with key k ***hashes*** to slot $h(k)$, and we also say that $h(k)$ is the ***hash value*** of key k. Figure 11.2 illustrates the basic idea. The hash function reduces the range of array indices and hence the size of the array. Instead of a size of $|U|$, the array can have size m. An example of a simple, but not particularly good, hash function is $h(k) = k \bmod m$.

There is one hitch, namely that two keys may hash to the same slot. We call this situation a ***collision***. Fortunately, there are effective techniques for resolving the conflict created by collisions.

Of course, the ideal solution is to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function h. One idea is to make h appear to be "random," thus avoiding collisions or at least minimizing their number. The very term "to hash," evoking images of random mixing and chopping, captures the spirit of this approach. (Of course, a hash function h must be deterministic in that a given input k must always produce the same output $h(k)$.) Because $|U| > m$, however, there must be at least two keys that have the same hash value,

[1] The definition of "average-case" requires care—are we assuming an input distribution over the keys, or are we randomizing the choice of hash function itself? We'll consider both approaches, but with an emphasis on the use of a randomly chosen hash function.

Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

and avoiding collisions altogether is impossible. Thus, although a well-designed, "random"-looking hash function can reduce the number of collisions, we still need a method for resolving the collisions that do occur.

The remainder of this section first presents a definition of "independent uniform hashing," which captures the simplest notion of what it means for a hash function to be "random." It then presents and analyzes the simplest collision resolution technique, called chaining. Section 11.4 introduces an alternative method for resolving collisions, called open addressing.

Independent uniform hashing

An "ideal" hashing function h would have, for each possible input k in the domain U, an output $h(k)$ that is an element randomly and independently chosen uniformly from the range $\{0, 1, \ldots, m-1\}$. Once a value $h(k)$ is randomly chosen, each subsequent call to h with the same input k yields the same output $h(k)$.

We call such an ideal hash function an ***independent uniform hash function***. Such a function is also often called a ***random oracle*** [43]. When hash tables are implemented with an independent uniform hash function, we say we are using ***independent uniform hashing***.

Independent uniform hashing is an ideal theoretical abstraction, but it is not something that can reasonably be implemented in practice. Nonetheless, we'll analyze the efficiency of hashing under the assumption of independent uniform hashing and then present ways of achieving useful practical approximations to this ideal.

Figure 11.3 Collision resolution by chaining. Each nonempty hash-table slot $T[j]$ points to a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$. The list can be either singly or doubly linked. We show it as doubly linked because deletion may be faster that way when the deletion procedure knows which list element (not just which key) is to be deleted.

Collision resolution by chaining

At a high level, you can think of hashing with chaining as a nonrecursive form of divide-and-conquer: the input set of n elements is divided randomly into m subsets, each of approximate size n/m. A hash function determines which subset an element belongs to. Each subset is managed independently as a list.

Figure 11.3 shows the idea behind ***chaining***: each nonempty slot points to a linked list, and all the elements that hash to the same slot go into that slot's linked list. Slot j contains a pointer to the head of the list of all stored elements with hash value j. If there are no such elements, then slot j contains NIL.

When collisions are resolved by chaining, the dictionary operations are straightforward to implement. They appear on the next page and use the linked-list procedures from Section 10.2. The worst-case running time for insertion is $O(1)$. The insertion procedure is fast in part because it assumes that the element x being inserted is not already present in the table. To enforce this assumption, you can search (at additional cost) for an element whose key is $x.key$ before inserting. For searching, the worst-case running time is proportional to the length of the list. (We'll analyze this operation more closely below.) Deletion takes $O(1)$ time if the lists are doubly linked, as in Figure 11.3. (Since CHAINED-HASH-DELETE takes as input an element x and not its key k, no search is needed. If the hash table supports deletion, then its linked lists should be doubly linked in order to delete an item quickly. If the lists were only singly linked, then by Exercise 10.2-1, deletion

```
CHAINED-HASH-INSERT(T, x)
1  LIST-PREPEND(T[h(x.key)], x)

CHAINED-HASH-SEARCH(T, k)
1  return LIST-SEARCH(T[h(k)], k)

CHAINED-HASH-DELETE(T, x)
1  LIST-DELETE(T[h(x.key)], x)
```

could take time proportional to the length of the list. With singly linked lists, both deletion and searching would have the same asymptotic running times.)

Analysis of hashing with chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given a hash table T with m slots that stores n elements, we define the ***load factor*** α for T as n/m, that is, the average number of elements stored in a chain. Our analysis will be in terms of α, which can be less than, equal to, or greater than 1.

The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n. The worst-case time for searching is thus $\Theta(n)$ plus the time to compute the hash function—no better than using one linked list for all the elements. We clearly don't use hash tables for their worst-case performance.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average (meaning with respect to the distribution of keys to be hashed and with respect to the choice of hash function, if this choice is randomized). Section 11.3 discusses these issues, but for now we assume that any given element is equally likely to hash into any of the m slots. That is, the hash function is ***uniform***. We further assume that where a given element hashes to is *independent* of where any other elements hash to. In other words, we assume that we are using ***independent uniform hashing***.

Because hashes of distinct keys are assumed to be independent, independent uniform hashing is ***universal***: the chance that any two distinct keys k_1 and k_2 collide is at most $1/m$. Universality is important in our analysis and also in the specification of universal families of hash functions, which we'll see in Section 11.3.2.

For $j = 0, 1, \ldots, m-1$, denote the length of the list $T[j]$ by n_j, so that

$$n = n_0 + n_1 + \cdots + n_{m-1} , \tag{11.1}$$

and the expected value of n_j is $\mathrm{E}[n_j] = \alpha = n/m$.

We assume that $O(1)$ time suffices to compute the hash value $h(k)$, so that the time required to search for an element with key k depends linearly on the length $n_{h(k)}$ of the list $T[h(k)]$. Setting aside the $O(1)$ time required to compute the hash function and to access slot $h(k)$, we'll consider the expected number of elements examined by the search algorithm, that is, the number of elements in the list $T[h(k)]$ that the algorithm checks to see whether any have a key equal to k. We consider two cases. In the first, the search is unsuccessful: no element in the table has key k. In the second, the search successfully finds an element with key k.

Theorem 11.1
In a hash table in which collisions are resolved by chaining, an unsuccessful search takes $\Theta(1 + \alpha)$ time on average, under the assumption of independent uniform hashing.

Proof Under the assumption of independent uniform hashing, any key k not already stored in the table is equally likely to hash to any of the m slots. The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$, which has expected length $\mathrm{E}[n_{h(k)}] = \alpha$. Thus, the expected number of elements examined in an unsuccessful search is α, and the total time required (including the time for computing $h(k)$) is $\Theta(1 + \alpha)$. ■

The situation for a successful search is slightly different. An unsuccessful search is equally likely to go to any slot of the hash table. A successful search, however, cannot go to an empty slot, since it is for an element that is present in one of the linked lists. We assume that the element searched for is equally likely to be any one of the elements in the table, so the longer the list, the more likely that the search is for one of its elements. Even so, the expected search time still turns out to be $\Theta(1 + \alpha)$.

Theorem 11.2
In a hash table in which collisions are resolved by chaining, a successful search takes $\Theta(1 + \alpha)$ time on average, under the assumption of independent uniform hashing.

Proof We assume that the element being searched for is equally likely to be any of the n elements stored in the table. The number of elements examined during a successful search for an element x is 1 more than the number of elements that appear before x in x's list. Because new elements are placed at the front of the list,

elements before x in the list were all inserted after x was inserted. Let x_i denote the ith element inserted into the table, for $i = 1, 2, \ldots, n$, and let $k_i = x_i.key$.

Our analysis uses indicator random variables extensively. For each slot q in the table and for each pair of distinct keys k_i and k_j, we define the indicator random variable

$$X_{ijq} = \mathrm{I}\{\text{the search is for } x_i, h(k_i) = q, \text{ and } h(k_j) = q\} .$$

That is, $X_{ijq} = 1$ when keys k_i and k_j collide at slot q and the search is for element x_i. Because $\Pr\{\text{the search is for } x_i\} = 1/n$, $\Pr\{h(k_i) = q\} = 1/m$, $\Pr\{h(k_j) = q\} = 1/m$, and these events are all independent, we have that $\Pr\{X_{ijq} = 1\} = 1/nm^2$. Lemma 5.1 on page 130 gives $\mathrm{E}[X_{ijq}] = 1/nm^2$.

Next, we define, for each element x_j, the indicator random variable

$$\begin{aligned} Y_j &= \mathrm{I}\{x_j \text{ appears in a list prior to the element being searched for}\} \\ &= \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq} , \end{aligned}$$

since at most one of the X_{ijq} equals 1, namely when the element x_i being searched for belongs to the same list as x_j (pointed to by slot q), and $i < j$ (so that x_i appears after x_j in the list).

Our final random variable is Z, which counts how many elements appear in the list prior to the element being searched for:

$$Z = \sum_{j=1}^{n} Y_j .$$

Because we must count the element being searched for as well as all those preceding it in its list, we wish to compute $\mathrm{E}[Z + 1]$. Using linearity of expectation (equation (C.24) on page 1192), we have

$$\begin{aligned} \mathrm{E}[Z + 1] &= \mathrm{E}\left[1 + \sum_{j=1}^{n} Y_j\right] \\ &= 1 + \mathrm{E}\left[\sum_{j=1}^{n} \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq}\right] \\ &= 1 + \mathrm{E}\left[\sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} X_{ijq}\right] \\ &= 1 + \sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} \mathrm{E}[X_{ijq}] \qquad \text{(by linearity of expectation)} \end{aligned}$$

$$
\begin{aligned}
&= 1 + \sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} \frac{1}{nm^2} \\
&= 1 + m \cdot \frac{n(n-1)}{2} \cdot \frac{1}{nm^2} \qquad \text{(by equation (A.2) on page 1141)} \\
&= 1 + \frac{n-1}{2m} \\
&= 1 + \frac{n}{2m} - \frac{1}{2m} \\
&= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} .
\end{aligned}
$$

Thus, the total time required for a successful search (including the time for computing the hash function) is $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$. ■

What does this analysis mean? If the number of elements in the table is at most proportional to the number of hash-table slots, we have $n = O(m)$ and, consequently, $\alpha = n/m = O(m)/m = O(1)$. Thus, searching takes constant time on average. Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked (assuming that the list element to be deleted is known, and not just its key), we can support all dictionary operations in $O(1)$ time on average.

The analysis in the preceding two theorems depends only on two essential properties of independent uniform hashing: uniformity (each key is equally likely to hash to any one of the m slots), and independence (so any two distinct keys collide with probability $1/m$).

Exercises

11.2-1

You use a hash function h to hash n distinct keys into an array T of length m. Assuming independent uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{k_1, k_2\} : k_1 \neq k_2 \text{ and } h(k_1) = h(k_2)\}$?

11.2-2

Consider a hash table with 9 slots and the hash function $h(k) = k \bmod 9$. Demonstrate what happens upon inserting the keys $5, 28, 19, 15, 20, 33, 12, 17, 10$ with collisions resolved by chaining.

11.2-3
Professor Marley hypothesizes that he can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

11.2-4
Suggest how to allocate and deallocate storage for elements within the hash table itself by creating a "free list": a linked list of all the unused slots. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in $O(1)$ expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?

11.2-5
You need to store a set of n keys in a hash table of size m. Show that if the keys are drawn from a universe U with $|U| > (n-1)m$, then U has a subset of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$.

11.2-6
You have stored n keys in a hash table of size m, with collisions resolved by chaining, and you know the length of each chain, including the length L of the longest chain. Describe a procedure that selects a key uniformly at random from among the keys in the hash table and returns it in expected time $O(L \cdot (1 + 1/\alpha))$.

11.3 Hash functions

For hashing to work well, it needs a good hash function. Along with being efficiently computable, what properties does a good hash function have? How do you design good hash functions?

This section first attempts to answer these questions based on two ad hoc approaches for creating hash functions: hashing by division and hashing by multiplication. Although these methods work well for some sets of input keys, they are limited because they try to provide a single fixed hash function that works well on any data—an approach called ***static hashing***.

We then see that provably good average-case performance for *any* data can be obtained by designing a suitable *family* of hash functions and choosing a hash function at random from this family at runtime, independent of the data to be hashed. The approach we examine is called random hashing. A particular kind of random

hashing, universal hashing, works well. As we saw with quicksort in Chapter 7, randomization is a powerful algorithmic design tool.

What makes a good hash function?

A good hash function satisfies (approximately) the assumption of independent uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other keys have hashed to. What does "equally likely" mean here? If the hash function is fixed, any probabilities would have to be based on the probability distribution of the input keys.

Unfortunately, you typically have no way to check this condition, unless you happen to know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently.

Occasionally you might know the distribution. For example, if you know that the keys are random real numbers k independently and uniformly distributed in the range $0 \le k < 1$, then the hash function

$$h(k) = \lfloor km \rfloor$$

satisfies the condition of independent uniform hashing.

A good static hashing approach derives the hash value in a way that you expect to be independent of any patterns that might exist in the data. For example, the "division method" (discussed in Section 11.3.1) computes the hash value as the remainder when the key is divided by a specified prime number. This method may give good results, if you (somehow) choose a prime number that is unrelated to any patterns in the distribution of keys.

Random hashing, described in Section 11.3.2, picks the hash function to be used at random from a suitable family of hashing functions. This approach removes any need to know anything about the probability distribution of the input keys, as the randomization necessary for good average-case behavior then comes from the (known) random process used to pick the hash function from the family of hash functions, rather than from the (unknown) process used to create the input keys. We recommend that you use random hashing.

Keys are integers, vectors, or strings

In practice, a hash function is designed to handle keys that are one of the following two types:

- A short nonnegative integer that fits in a w-bit machine word. Typical values for w would be 32 or 64.

- A short vector of nonnegative integers, each of bounded size. For example, each element might be an 8-bit byte, in which case the vector is often called a (byte) string. The vector might be of variable length.

To begin, we assume that keys are short nonnegative integers. Handling vector keys is more complicated and discussed in Sections 11.3.5 and 11.5.2.

11.3.1 Static hashing

Static hashing uses a single, fixed hash function. The only randomization available is through the (usually unknown) distribution of input keys. This section discusses two standard approaches for static hashing: the division method and the multiplication method. Although static hashing is no longer recommended, the multiplication method also provides a good foundation for "nonstatic" hashing—better known as random hashing—where the hash function is chosen at random from a suitable family of hash functions.

The division method

The ***division method*** for creating hash functions maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$$h(k) = k \bmod m \ .$$

For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$. Since it requires only a single division operation, hashing by division is quite fast.

The division method may work well when m is a prime not too close to an exact power of 2. There is no guarantee that this method provides good average-case performance, however, and it may complicate applications since it constrains the size of the hash tables to be prime.

The multiplication method

The general ***multiplication method*** for creating hash functions operates in two steps. First, multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, multiply this value by m and take the floor of the result. That is, the hash function is

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor \ ,$$

where "$kA \bmod 1$" means the fractional part of kA, that is, $kA - \lfloor kA \rfloor$. The general multiplication method has the advantage that the value of m is not critical and you can choose it independently of how you choose the multiplicative constant A.

Figure 11.4 The multiply-shift method to compute a hash function. The w-bit representation of the key k is multiplied by the w-bit value $a = A \cdot 2^w$. The ℓ highest-order bits of the lower w-bit half of the product form the desired hash value $h_a(k)$.

The multiply-shift method

In practice, the multiplication method is best in the special case where the number m of hash-table slots is an exact power of 2, so that $m = 2^\ell$ for some integer ℓ, where $\ell \le w$ and w is the number of bits in a machine word. If you choose a fixed w-bit positive integer $a = A\,2^w$, where $0 < A < 1$ as in the multiplication method so that a is in the range $0 < a < 2^w$, you can implement the function on most computers as follows. We assume that a key k fits into a single w-bit word.

Referring to Figure 11.4, first multiply k by the w-bit integer a. The result is a $2w$-bit value $r_1 2^w + r_0$, where r_1 is the high-order w-bit word of the product and r_0 is the low-order w-bit word of the product. The desired ℓ-bit hash value consists of the ℓ most significant bits of r_0. (Since r_1 is ignored, the hash function can be implemented on a computer that produces only a w-bit product given two w-bit inputs, that is, where the multiplication operation computes modulo 2^w.)

In other words, you define the hash function $h = h_a$, where

$$h_a(k) = (ka \bmod 2^w) \ggg (w - \ell) \tag{11.2}$$

for a fixed nonzero w-bit value a. Since the product ka of two w-bit words occupies $2w$ bits, taking this product modulo 2^w zeroes out the high-order w bits (r_1), leaving only the low-order w bits (r_0). The $\ggg$ operator performs a logical right shift by $w - \ell$ bits, shifting zeros into the vacated positions on the left, so that the ℓ most significant bits of r_0 move into the ℓ rightmost positions. (It's the same as dividing by $2^{w-\ell}$ and taking the floor of the result.) The resulting value equals the ℓ most significant bits of r_0. The hash function h_a can be implemented with three machine instructions: multiplication, subtraction, and logical right shift.

As an example, suppose that $k = 123456$, $\ell = 14$, $m = 2^{14} = 16384$, and $w = 32$. Suppose further that we choose $a = 2654435769$ (following a suggestion

of Knuth [261]). Then $ka = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, and so $r_1 = 76300$ and $r_0 = 17612864$. The 14 most significant bits of r_0 yield the value $h_a(k) = 67$.

Even though the multiply-shift method is fast, it doesn't provide any guarantee of good average-case performance. The universal hashing approach presented in the next section provides such a guarantee. A simple randomized variant of the multiply-shift method works well on the average, when the program begins by picking a as a randomly chosen odd integer.

11.3.2 Random hashing

Suppose that a malicious adversary chooses the keys to be hashed by some fixed hash function. Then the adversary can choose n keys that all hash to the same slot, yielding an average retrieval time of $\Theta(n)$. Any static hash function is vulnerable to such terrible worst-case behavior. The only effective way to improve the situation is to choose the hash function *randomly* in a way that is *independent* of the keys that are actually going to be stored. This approach is called ***random hashing***. A special case of this approach, called ***universal hashing***, can yield provably good performance on average when collisions are handled by chaining, no matter which keys the adversary chooses.

To use random hashing, at the beginning of program execution you select the hash function at random from a suitable family of functions. As in the case of quicksort, randomization guarantees that no single input always evokes worst-case behavior. Because you randomly select the hash function, the algorithm can behave differently on each execution, even for the same set of keys to be hashed, guaranteeing good average-case performance.

Let $\mathcal{H}$ be a finite family of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$. Such a family is said to be ***universal*** if for each pair of distinct keys $k_1, k_2 \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k_1) = h(k_2)$ is at most $|\mathcal{H}|/m$. In other words, with a hash function randomly chosen from $\mathcal{H}$, the chance of a collision between distinct keys k_1 and k_2 is no more than the chance $1/m$ of a collision if $h(k_1)$ and $h(k_2)$ were randomly and independently chosen from the set $\{0, 1, \ldots, m-1\}$.

Independent uniform hashing is the same as picking a hash function uniformly at random from a family of m^n hash functions, each member of that family mapping the n keys to the m hash values in a different way.

Every independent uniform random family of hash function is universal, but the converse need not be true: consider the case where $U = \{0, 1, \ldots, m-1\}$ and the only hash function in the family is the identity function. The probability that two distinct keys collide is zero, even though each key is hashes to a fixed value.

The following corollary to Theorem 11.2 on page 279 says that universal hashing provides the desired payoff: it becomes impossible for an adversary to pick a sequence of operations that forces the worst-case running time.

Corollary 11.3
Using universal hashing and collision resolution by chaining in an initially empty table with m slots, it takes $\Theta(s)$ expected time to handle any sequence of s INSERT, SEARCH, and DELETE operations containing $n = O(m)$ INSERT operations.

Proof The INSERT and DELETE operations take constant time. Since the number n of insertions is $O(m)$, we have that $\alpha = O(1)$. Furthermore, the expected time for each SEARCH operation is $O(1)$, which can be seen by examining the proof of Theorem 11.2. That analysis depends only on collision probabilities, which are $1/m$ for any pair k_1, k_2 of keys by the choice of an independent uniform hash function in that theorem. Using a universal family of hash functions here instead of using independent uniform hashing changes the probability of collision from $1/m$ to at most $1/m$. By linearity of expectation, therefore, the expected time for the entire sequence of s operations is $O(s)$. Since each operation takes $\Omega(1)$ time, the $\Theta(s)$ bound follows. ■

11.3.3 Achievable properties of random hashing

There is a rich literature on the properties a family $\mathcal{H}$ of hash functions can have, and how they relate to the efficiency of hashing. We summarize a few of the most interesting ones here.

Let $\mathcal{H}$ be a family of hash functions, each with domain U and range $\{0, 1, \ldots, m-1\}$, and let h be any hash function that is picked uniformly at random from $\mathcal{H}$. The probabilities mentioned are probabilities over the picks of h.

- The family $\mathcal{H}$ is ***uniform*** if for any key k in U and any slot q in the range $\{0, 1, \ldots, m-1\}$, the probability that $h(k) = q$ is $1/m$.
- The family $\mathcal{H}$ is ***universal*** if for any distinct keys k_1 and k_2 in U, the probability that $h(k_1) = h(k_2)$ is at most $1/m$.
- The family $\mathcal{H}$ of hash functions is ***ϵ-universal*** if for any distinct keys k_1 and k_2 in U, the probability that $h(k_1) = h(k_2)$ is at most ϵ. Therefore, a universal family of hash functions is also $1/m$-universal.[2]

[2] In the literature, a (c/m)-universal hash function is sometimes called c-universal or c-approximately universal. We'll stick with the notation (c/m)-universal.

- The family $\mathcal{H}$ is ***d*-independent** if for any distinct keys $k_1, k_2, \ldots, k_d$ in U and any slots $q_1, q_2, \ldots, q_d$, not necessarily distinct, in $\{0, 1, \ldots, m-1\}$ the probability that $h(k_i) = q_i$ for $i = 1, 2, \ldots, d$ is $1/m^d$.

Universal hash-function families are of particular interest, as they are the simplest type supporting provably efficient hash-table operations for any input data set. Many other interesting and desirable properties, such as those noted above, are also possible and allow for efficient specialized hash-table operations.

11.3.4 Designing a universal family of hash functions

This section present two ways to design a universal (or ϵ-universal) family of hash functions: one based on number theory and another based on a randomized variant of the multiply-shift method presented in Section 11.3.1. The first method is a bit easier to prove universal, but the second method is newer and faster in practice.

A universal family of hash functions based on number theory

We can design a universal family of hash functions using a little number theory. You may wish to refer to Chapter 31 if you are unfamiliar with basic concepts in number theory.

Begin by choosing a prime number p large enough so that every possible key k lies in the range 0 to $p-1$, inclusive. We assume here that p has a "reasonable" length. (See Section 11.3.5 for a discussion of methods for handling long input keys, such as variable-length strings.) Let $\mathbb{Z}_p$ denote the set $\{0, 1, \ldots, p-1\}$, and let $\mathbb{Z}_p^*$ denote the set $\{1, 2, \ldots, p-1\}$. Since p is prime, we can solve equations modulo p with the methods given in Chapter 31. Because the size of the universe of keys is greater than the number of slots in the hash table (otherwise, just use direct addressing), we have $p > m$.

Given any $a \in \mathbb{Z}_p^*$ and any $b \in \mathbb{Z}_p$, define the hash function h_{ab} as a linear transformation followed by reductions modulo p and then modulo m:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m \; . \tag{11.3}$$

For example, with $p = 17$ and $m = 6$, we have

$$\begin{aligned} h_{3,4}(8) &= ((3 \cdot 8 + 4) \bmod 17) \bmod 6 \\ &= (28 \bmod 17) \bmod 6 \\ &= 11 \bmod 6 \\ &= 5 \; . \end{aligned}$$

Given p and m, the family of all such hash functions is

$$\mathcal{H}_{pm} = \left\{ h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p \right\} \; . \tag{11.4}$$

Each hash function h_{ab} maps $\mathbb{Z}_p$ to $\mathbb{Z}_m$. This family of hash functions has the nice property that the size m of the output range (which is the size of the hash table) is arbitrary—it need not be prime. Since you can choose from among $p-1$ values for a and p values for b, the family $\mathcal{H}_{pm}$ contains $p(p-1)$ hash functions.

Theorem 11.4
The family $\mathcal{H}_{pm}$ of hash functions defined by equations (11.3) and (11.4) is universal.

Proof Consider two distinct keys k_1 and k_2 from $\mathbb{Z}_p$, so that $k_1 \neq k_2$. For a given hash function h_{ab}, let

$$\begin{aligned} r_1 &= (ak_1 + b) \bmod p \ , \\ r_2 &= (ak_2 + b) \bmod p \ . \end{aligned}$$

We first note that $r_1 \neq r_2$. Why? Since we have $r_1 - r_2 = a(k_1 - k_2) \pmod p$, it follows that $r_1 \neq r_2$ because p is prime and both a and $(k_1 - k_2)$ are nonzero modulo p. By Theorem 31.6 on page 908, their product must also be nonzero modulo p. Therefore, when computing any $h_{ab} \in \mathcal{H}_{pm}$, distinct inputs k_1 and k_2 map to distinct values r_1 and r_2 modulo p, and there are no collisions yet at the "mod p level." Moreover, each of the possible $p(p-1)$ choices for the pair (a, b) with $a \neq 0$ yields a *different* resulting pair (r_1, r_2) with $r_1 \neq r_2$, since we can solve for a and b given r_1 and r_2:

$$\begin{aligned} a &= \big((r_1 - r_2)((k_1 - k_2)^{-1} \bmod p)\big) \bmod p \ , \\ b &= (r_1 - ak_1) \bmod p \ , \end{aligned}$$

where $((k_1 - k_2)^{-1} \bmod p)$ denotes the unique multiplicative inverse, modulo p, of $k_1 - k_2$. For each of the p possible values of r_1, there are only $p-1$ possible values of r_2 that do not equal r_1, making only $p(p-1)$ possible pairs (r_1, r_2) with $r_1 \neq r_2$. Therefore, there is a one-to-one correspondence between pairs (a, b) with $a \neq 0$ and pairs (r_1, r_2) with $r_1 \neq r_2$. Thus, for any given pair of distinct inputs k_1 and k_2, if we pick (a, b) uniformly at random from $\mathbb{Z}_p^* \times \mathbb{Z}_p$, the resulting pair (r_1, r_2) is equally likely to be any pair of distinct values modulo p.

Therefore, the probability that distinct keys k_1 and k_2 collide is equal to the probability that $r_1 = r_2 \pmod m$ when r_1 and r_2 are randomly chosen as distinct values modulo p. For a given value of r_1, of the $p-1$ possible remaining values for r_2, the number of values r_2 such that $r_2 \neq r_1$ and $r_2 = r_1 \pmod m$ is at most

$$\begin{aligned} \left\lceil \frac{p}{m} \right\rceil - 1 &\le \frac{p+m-1}{m} - 1 \quad \text{(by inequality (3.7) on page 64)} \\ &= \frac{p-1}{m} \ . \end{aligned}$$

The probability that r_2 collides with r_1 when reduced modulo m is at most $((p-1)/m)/(p-1) = 1/m$, since r_2 is equally likely to be any of the $p-1$ values in Z_p that are different from r_1, but at most $(p-1)/m$ of those values are equivalent to r_1 modulo m.

Therefore, for any pair of distinct values $k_1, k_2 \in \mathbb{Z}_p$,

$$\Pr\{h_{ab}(k_1) = h_{ab}(k_2)\} \le 1/m ,$$

so that $\mathcal{H}_{pm}$ is indeed universal. ■

A $2/m$-universal family of hash functions based on the multiply-shift method

We recommend that in practice you use the following hash-function family based on the multiply-shift method. It is exceptionally efficient and (although we omit the proof) provably $2/m$-universal. Define $\mathcal{H}$ to be the family of multiply-shift hash functions with odd constants a:

$$\mathcal{H} = \{h_a : a \text{ is odd}, 1 \le a < m, \text{and } h_a \text{ is defined by equation (11.2)}\} . \qquad (11.5)$$

Theorem 11.5
The family of hash functions $\mathcal{H}$ given by equation (11.5) is $2/m$-universal. ■

That is, the probability that any two distinct keys collide is at most $2/m$. In many practical situations, the speed of computing the hash function more than compensates for the higher upper bound on the probability that two distinct keys collide when compared with a universal hash function.

11.3.5 Hashing long inputs such as vectors or strings

Sometimes hash function inputs are so long that they cannot be easily encoded modulo a reasonably sized prime number p or encoded within a single word of, say, 64 bits. As an example, consider the class of vectors, such as vectors of 8-bit bytes (which is how strings in many programming languages are stored). A vector might have an arbitrary nonnegative length, in which case the length of the input to the hash function may vary from input to input.

Number-theoretic approaches

One way to design good hash functions for variable-length inputs is to extend the ideas used in Section 11.3.4 to design universal hash functions. Exercise 11.3-6 explores one such approach.

Cryptographic hashing

Another way to design a good hash function for variable-length inputs is to use a hash function designed for cryptographic applications. ***Cryptographic hash functions*** are complex pseudorandom functions, designed for applications requiring properties beyond those needed here, but are robust, widely implemented, and usable as hash functions for hash tables.

A cryptographic hash function takes as input an arbitrary byte string and returns a fixed-length output. For example, the NIST standard deterministic cryptographic hash function SHA-256 [346] produces a 256-bit (32-byte) output for any input.

Some chip manufacturers include instructions in their CPU architectures to provide fast implementations of some cryptographic functions. Of particular interest are instructions that efficiently implement rounds of the Advanced Encryption Standard (AES), the "AES-NI" instructions. These instructions execute in a few tens of nanoseconds, which is generally fast enough for use with hash tables. A message authentication code such as CBC-MAC based on AES and the use of the AES-NI instructions could be a useful and efficient hash function. We don't pursue the potential use of specialized instruction sets further here.

Cryptographic hash functions are useful because they provide a way of implementing an approximate version of a random oracle. As noted earlier, a random oracle is equivalent to an independent uniform hash function family. From a theoretical point of view, a random oracle is an unachievable ideal: a deterministic function that provides a randomly selected output for each input. Because it is deterministic, it provides the same output if queried again for the same input. From a practical point of view, constructions of hash function families based on cryptographic hash functions are sensible substitutes for random oracles.

There are many ways to use a cryptographic hash function as a hash function. For example, we could define

$$h(k) = \text{SHA-256}(k) \bmod m \ .$$

To define a family of such hash functions one may prepend a "salt" string a to the input before hashing it, as in

$$h_a(k) = \text{SHA-256}(a \parallel k) \bmod m \ ,$$

where $a \parallel k$ denotes the string formed by concatenating the strings a and k. The literature on message authentication codes (MACs) provides additional approaches.

Cryptographic approaches to hash-function design are becoming more practical as computers arrange their memories in hierarchies of differing capacities and speeds. Section 11.5 discusses one hash-function design based on the RC6 encryption method.

Exercises

11.3-1
You wish to search a linked list of length n, where each element contains a key k along with a hash value $h(k)$. Each key is a long character string. How might you take advantage of the hash values when searching the list for an element with a given key?

11.3-2
You hash a string of r characters into m slots by treating it as a radix-128 number and then using the division method. You can represent the number m as a 32-bit computer word, but the string of r characters, treated as a radix-128 number, takes many words. How can you apply the division method to compute the hash value of the character string without using more than a constant number of words of storage outside the string itself?

11.3-3
Consider a version of the division method in which $h(k) = k \bmod m$, where $m = 2^p - 1$ and k is a character string interpreted in radix 2^p. Show that if string x can be converted to string y by permuting its characters, then x and y hash to the same value. Give an example of an application in which this property would be undesirable in a hash function.

11.3-4
Consider a hash table of size $m = 1000$ and a corresponding hash function $h(k) = \lfloor m\,(kA \bmod 1) \rfloor$ for $A = (\sqrt{5} - 1)/2$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

★ ***11.3-5***
Show that any ϵ-universal family $\mathcal{H}$ of hash functions from a finite set U to a finite set Q has $\epsilon \geq 1/|Q| - 1/|U|$.

★ ***11.3-6***
Let U be the set of d-tuples of values drawn from $\mathbb{Z}_p$, and let $Q = \mathbb{Z}_p$, where p is prime. Define the hash function $h_b : U \to Q$ for $b \in \mathbb{Z}_p$ on an input d-tuple $\langle a_0, a_1, \ldots, a_{d-1} \rangle$ from U as

$$h_b(\langle a_0, a_1, \ldots, a_{d-1} \rangle) = \left(\sum_{j=0}^{d-1} a_j b^j \right) \bmod p \ ,$$

and let $\mathcal{H} = \{h_b : b \in \mathbb{Z}_p\}$. Argue that $\mathcal{H}$ is ϵ-universal for $\epsilon = (d-1)/p$. (*Hint:* See Exercise 31.4-4.)

11.4 Open addressing

This section describes open addressing, a method for collision resolution that, unlike chaining, does not make use of storage outside of the hash table itself. In ***open addressing***, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. No lists or elements are stored outside the table, unlike in chaining. Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made. One consequence is that the load factor α can never exceed 1.

Collisions are handled as follows: when a new element is to be inserted into the table, it is placed in its "first-choice" location if possible. If that location is already occupied, the new element is placed in its "second-choice" location. The process continues until an empty slot is found in which to place the new element. Different elements have different preference orders for the locations.

To search for an element, systematically examine the preferred table slots for that element, in order of decreasing preference, until either you find the desired element or you find an empty slot and thus verify that the element is not in the table.

Of course, you could use chaining and store the linked lists inside the hash table, in the otherwise unused hash-table slots (see Exercise 11.2-4), but the advantage of open addressing is that it avoids pointers altogether. Instead of following pointers, you compute the sequence of slots to be examined. The memory freed by not storing pointers provides the hash table with a larger number of slots in the same amount of memory, potentially yielding fewer collisions and faster retrieval.

To perform insertion using open addressing, successively examine, or ***probe***, the hash table until you find an empty slot in which to put the key. Instead of being fixed in the order $0, 1, \ldots, m-1$ (which implies a $\Theta(n)$ search time), the sequence of positions probed depends upon the key being inserted. To determine which slots to probe, the hash function includes the probe number (starting from 0) as a second input. Thus, the hash function becomes

$$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\} \ .$$

Open addressing requires that for every key k, the ***probe sequence*** $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ be a permutation of $\langle 0, 1, \ldots, m-1\rangle$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up. The HASH-INSERT procedure on the following page assumes that the elements in the hash table T are keys with no satellite information: the key k is identical to the element containing key k. Each slot contains either a key or NIL (if the slot is empty). The HASH-INSERT procedure takes as input a hash table T and a key k

that is assumed to be not already present in the hash table. It either returns the slot number where it stores key k or flags an error because the hash table is already full.

```
HASH-INSERT(T, k)
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        T[q] = k
        return q
    else i = i + 1
until i == m
error "hash table overflow"

HASH-SEARCH(T, k)
i = 0
repeat
    q = h(k, i)
    if T[q] == k
        return q
    i = i + 1
until T[q] == NIL or i == m
return NIL
```

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. Therefore, the search can terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence. The procedure HASH-SEARCH takes as input a hash table T and a key k, returning q if it finds that slot q contains key k, or NIL if key k is not present in table T.

Deletion from an open-address hash table is tricky. When you delete a key from slot q, it would be a mistake to mark that slot as empty by simply storing NIL in it. If you did, you might be unable to retrieve any key k for which slot q was probed and found occupied when k was inserted. One way to solve this problem is by marking the slot, storing in it the special value DELETED instead of NIL. The HASH-INSERT procedure then has to treat such a slot as empty so that it can insert a new key there. The HASH-SEARCH procedure passes over DELETED values while searching, since slots containing DELETED were filled when the key being searched for was inserted. Using the special value DELETED, however, means that search times no longer depend on the load factor α, and for this reason chaining is

frequently selected as a collision resolution technique when keys must be deleted. There is a simple special case of open addressing, linear probing, that avoids the need to mark slots with DELETED. Section 11.5.1 shows how to delete from a hash table when using linear probing.

In our analysis, we assume ***independent uniform permutation hashing*** (also confusingly known as ***uniform hashing*** in the literature): the probe sequence of each key is equally likely to be any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$. Independent uniform permutation hashing generalizes the notion of independent uniform hashing defined earlier to a hash function that produces not just a single slot number, but a whole probe sequence. True independent uniform permutation hashing is difficult to implement, however, and in practice suitable approximations (such as double hashing, defined below) are used.

We'll examine both double hashing and its special case, linear probing. These techniques guarantee that $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$ is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$ for each key k. (Recall that the second parameter to the hash function h is the probe number.) Neither double hashing nor linear probing meets the assumption of independent uniform permutation hashing, however. Double hashing cannot generate more than m^2 different probe sequences (instead of the $m!$ that independent uniform permutation hashing requires). Nonetheless, double hashing has a large number of possible probe sequences and, as you might expect, seems to give good results. Linear probing is even more restricted, capable of generating only m different probe sequences.

Double hashing

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. ***Double hashing*** uses a hash function of the form

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m ,$$

where both h_1 and h_2 are ***auxiliary hash functions***. The initial probe goes to position $T[h_1(k)]$, and successive probe positions are offset from previous positions by the amount $h_2(k)$, modulo m. Thus, the probe sequence here depends in two ways upon the key k, since the initial probe position $h_1(k)$, the step size $h_2(k)$, or both, may vary. Figure 11.5 gives an example of insertion by double hashing.

In order for the entire hash table to be searched, the value $h_2(k)$ must be relatively prime to the hash-table size m. (See Exercise 11.4-5.) A convenient way to ensure this condition is to let m be an exact power of 2 and to design h_2 so that it always produces an odd number. Another way is to let m be prime and to design h_2 so that it always returns a positive integer less than m. For example, you

Figure 11.5 Insertion by double hashing. The hash table has size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 = 1 \pmod{13}$ and $14 = 3 \pmod{11}$, the key 14 goes into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

could choose m prime and let

$$h_1(k) = k \bmod m ,$$
$$h_2(k) = 1 + (k \bmod m') ,$$

where m' is chosen to be slightly less than m (say, $m - 1$). For example, if $k = 123456$, $m = 701$, and $m' = 700$, then $h_1(k) = 80$ and $h_2(k) = 257$, so that the first probe goes to position 80, and successive probes examine every 257th slot (modulo m) until the key has been found or every slot has been examined.

Although values of m other than primes or exact powers of 2 can in principle be used with double hashing, in practice it becomes more difficult to efficiently generate $h_2(k)$ (other than choosing $h_2(k) = 1$, which gives linear probing) in a way that ensures that it is relatively prime to m, in part because the relative density $\phi(m)/m$ of such numbers for general m may be small (see equation (31.25) on page 921).

When m is prime or an exact power of 2, double hashing produces $\Theta(m^2)$ probe sequences, since each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence. As a result, for such values of m, double hashing appears to perform close to the "ideal" scheme of independent uniform permutation hashing.

Linear probing

Linear probing, a special case of double hashing, is the simplest open-addressing approach to resolving collisions. As with double hashing, an auxiliary hash function h_1 determines the first probe position $h_1(k)$ for inserting an element. If slot $T[h_1(k)]$ is already occupied, probe the next position $T[h_1(k)+1]$. Keep going as necessary, on up to slot $T[m-1]$, and then wrap around to slots $T[0]$, $T[1]$, and so on, but never going past slot $T[h_1(k)-1]$. To view linear probing as a special case of double hashing, just set the double-hashing step function h_2 to be fixed at 1: $h_2(k) = 1$ for all k. That is, the hash function is

$$h(k,i) = (h_1(k) + i) \bmod m \tag{11.6}$$

for $i = 0, 1, \ldots, m-1$. The value of $h_1(k)$ determines the entire probe sequence, and so assuming that $h_1(k)$ can take on any value in $\{0, 1, \ldots, m-1\}$, linear probing allows only m distinct probe sequences.

We'll revisit linear probing in Section 11.5.1.

Analysis of open-address hashing

As in our analysis of chaining in Section 11.2, we analyze open addressing in terms of the load factor $\alpha = n/m$ of the hash table. With open addressing, at most one element occupies each slot, and thus $n \le m$, which implies $\alpha \le 1$. The analysis below requires α to be strictly less than 1, and so we assume that at least one slot is empty. Because deleting from an open-address hash table does not really free up a slot, we assume as well that no deletions occur.

For the hash function, we assume independent uniform permutation hashing. In this idealized scheme, the probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ used to insert or search for each key k is equally likely to be any permutation of $\langle 0, 1, \ldots, m-1\rangle$. Of course, any given key has a unique fixed probe sequence associated with it. What we mean here is that, considering the probability distribution on the space of keys and the operation of the hash function on the keys, each possible probe sequence is equally likely.

We now analyze the expected number of probes for hashing with open addressing under the assumption of independent uniform permutation hashing, beginning with the expected number of probes made in an unsuccessful search (assuming, as stated above, that $\alpha < 1$).

The bound proven, of $1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$, has an intuitive interpretation. The first probe always occurs. With probability approximately α, the first probe finds an occupied slot, so that a second probe happens. With probability approximately α^2, the first two slots are occupied so that a third probe ensues, and so on.

Theorem 11.6
Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming independent uniform permutation hashing and no deletions.

Proof In an unsuccessful search, every probe but the last accesses an occupied slot that does not contain the desired key, and the last slot probed is empty. Let the random variable X denote the number of probes made in an unsuccessful search, and define the event A_i, for $i = 1, 2, \ldots$, as the event that an ith probe occurs and it is to an occupied slot. Then the event $\{X \geq i\}$ is the intersection of events $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$. We bound $\Pr\{X \geq i\}$ by bounding $\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$. By Exercise C.2-5 on page 1190,

$$\begin{aligned}\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} &= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ &\qquad \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\} \ .\end{aligned}$$

Since there are n elements and m slots, $\Pr\{A_1\} = n/m$. For $j > 1$, the probability that there is a jth probe and it is to an occupied slot, given that the first $j-1$ probes were to occupied slots, is $(n-j+1)/(m-j+1)$. This probability follows because the jth probe would be finding one of the remaining $(n-(j-1))$ elements in one of the $(m-(j-1))$ unexamined slots, and by the assumption of independent uniform permutation hashing, the probability is the ratio of these quantities. Since $n < m$ implies that $(n-j)/(m-j) \leq n/m$ for all j in the range $0 \leq j < m$, it follows that for all i in the range $1 \leq i \leq m$, we have

$$\begin{aligned}\Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1} \ .\end{aligned}$$

The product in the first line has $i-1$ factors. When $i = 1$, the product is 1, the identity for multiplication, and we get $\Pr\{X \geq 1\} = 1$, which makes sense, since there must always be at least 1 probe. If each of the first n probes is to an occupied slot, then all occupied slots have been probed. Then, the $(n+1)$st probe must be to an empty slot, which gives $\Pr\{X \geq i\} = 0$ for $i > n+1$. Now, we use equation (C.28) on page 1193 to bound the expected number of probes:

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ &= \sum_{i=1}^{n+1} \Pr\{X \geq i\} + \sum_{i > n+1} \Pr\{X \geq i\}\end{aligned}$$

$$
\begin{aligned}
&\le \sum_{i=1}^{\infty} \alpha^{i-1} + 0 \\
&= \sum_{i=0}^{\infty} \alpha^{i} \\
&= \frac{1}{1-\alpha} \qquad \text{(by equation (A.7) on page 1142 because } 0 \le \alpha < 1) . \qquad \blacksquare
\end{aligned}
$$

If α is a constant, Theorem 11.6 predicts that an unsuccessful search runs in $O(1)$ time. For example, if the hash table is half full, the average number of probes in an unsuccessful search is at most $1/(1-.5) = 2$. If it is 90% full, the average number of probes is at most $1/(1-.9) = 10$.

Theorem 11.6 yields almost immediately how well the HASH-INSERT procedure performs.

Corollary 11.7
Inserting an element into an open-address hash table with load factor α, where $\alpha < 1$, requires at most $1/(1-\alpha)$ probes on average, assuming independent uniform permutation hashing and no deletions.

Proof An element is inserted only if there is room in the table, and thus $\alpha < 1$. Inserting a key requires an unsuccessful search followed by placing the key into the first empty slot found. Thus, the expected number of probes is at most $1/(1-\alpha)$. ■

It takes a little more work to compute the expected number of probes for a successful search.

Theorem 11.8
Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha} ,$$

assuming independent uniform permutation hashing with no deletions and assuming that each key in the table is equally likely to be searched for.

Proof A search for a key k reproduces the same probe sequence as when the element with key k was inserted. If k was the $(i+1)$st key inserted into the hash table, then the load factor at the time it was inserted was i/m, and so by Corollary 11.7, the expected number of probes made in a search for k is at most $1/(1-i/m) = m/(m-i)$. Averaging over all n keys in the hash table gives us

the expected number of probes in a successful search:

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\le \frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}\,dx \qquad \text{(by inequality (A.19) on page 1150)} \\
&= \frac{1}{\alpha}(\ln m - \ln(m-n)) \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}\,. \qquad \blacksquare
\end{aligned}$$

If the hash table is half full, the expected number of probes in a successful search is less than 1.387. If the hash table is 90% full, the expected number of probes is less than 2.559. If $\alpha = 1$, then in an unsuccessful search, all m slots must be probed. Exercise 11.4-4 asks you to analyze a successful search when $\alpha = 1$.

Exercises

11.4-1

Consider inserting the keys $10, 22, 31, 4, 15, 28, 17, 88, 59$ into a hash table of length $m = 11$ using open addressing. Illustrate the result of inserting these keys using linear probing with $h(k, i) = (k + i) \bmod m$ and using double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \bmod (m - 1))$.

11.4-2

Write pseudocode for HASH-DELETE that fills the deleted key's slot with the special value DELETED, and modify HASH-SEARCH and HASH-INSERT as needed to handle DELETED.

11.4-3

Consider an open-address hash table with independent uniform permutation hashing and no deletions. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is $3/4$ and when it is $7/8$.

11.4-4
Show that the expected number of probes required for a successful search when $\alpha = 1$ (that is, when $n = m$), is H_m, the mth harmonic number.

★ ***11.4-5***
Show that, with double hashing, if m and $h_2(k)$ have greatest common divisor $d \geq 1$ for some key k, then an unsuccessful search for key k examines $(1/d)$th of the hash table before returning to slot $h_1(k)$. Thus, when $d = 1$, so that m and $h_2(k)$ are relatively prime, the search may examine the entire hash table. (*Hint:* See Chapter 31.)

★ ***11.4-6***
Consider an open-address hash table with a load factor α. Approximate the nonzero value α for which the expected number of probes in an unsuccessful search equals twice the expected number of probes in a successful search. Use the upper bounds given by Theorems 11.6 and 11.8 for these expected numbers of probes.

11.5 Practical considerations

Efficient hash table algorithms are not only of theoretical interest, but also of immense practical importance. Constant factors can matter. For this reason, this section discusses two aspects of modern CPUs that are not included in the standard RAM model presented in Section 2.2:

Memory hierarchies: The memory of modern CPUs has a number of levels, from the fast registers, through one or more levels of ***cache memory***, to the main-memory level. Each successive level stores more data than the previous level, but access is slower. As a consequence, a complex computation (such as a complicated hash function) that works entirely within the fast registers can take less time than a single read operation from main memory. Furthermore, cache memory is organized in ***cache blocks*** of (say) 64 bytes each, which are always fetched together from main memory. There is a substantial benefit for ensuring that memory usage is local: reusing the same cache block is much more efficient than fetching a different cache block from main memory.

The standard RAM model measures efficiency of a hash-table operation by counting the number of hash-table slots probed. In practice, this metric is only a crude approximation to the truth, since once a cache block is in the cache, successive probes to that cache block are much faster than probes that must access main memory.

Advanced instruction sets: Modern CPUs may have sophisticated instruction sets that implement advanced primitives useful for encryption or other forms of cryptography. These instructions may be useful in the design of exceptionally efficient hash functions.

Section 11.5.1 discusses linear probing, which becomes the collision-resolution method of choice in the presence of a memory hierarchy. Section 11.5.2 suggests how to construct "advanced" hash functions based on cryptographic primitives, suitable for use on computers with hierarchical memory models.

11.5.1 Linear probing

Linear probing is often disparaged because of its poor performance in the standard RAM model. But linear probing excels for hierarchical memory models, because successive probes are usually to the same cache block of memory.

Deletion with linear probing

Another reason why linear probing is often not used in practice is that deletion seems complicated or impossible without using the special DELETED value. Yet we'll now see that deletion from a hash table based on linear probing is not all that difficult, even without the DELETED marker. The deletion procedure works for linear probing, but not for open-address probing in general, because with linear probing keys all follow the same simple cyclic probing sequence (albeit with different starting points).

The deletion procedure relies on an "inverse" function to the linear-probing hash function $h(k,i) = (h_1(k) + i) \bmod m$, which maps a key k and a probe number i to a slot number in the hash table. The inverse function g maps a key k and a slot number q, where $0 \le q < m$, to the probe number that reaches slot q:

$$g(k,q) = (q - h_1(k)) \bmod m .$$

If $h(k,i) = q$, then $g(k,q) = i$, and so $h(k, g(k,q)) = q$.

The procedure LINEAR-PROBING-HASH-DELETE on the facing page deletes the key stored in position q from hash table T. Figure 11.6 shows how it works. The procedure first deletes the key in position q by setting $T[q]$ to NIL in line 2. It then searches for a slot q' (if any) that contains a key that should be moved to the slot q just vacated by key k. Line 9 asks the critical question: does the key k' in slot q' need to be moved to the vacated slot q in order to preserve the accessibility of k'? If $g(k',q) < g(k',q')$, then during the insertion of k' into the table, slot q was examined but found to be already occupied. But now slot q, where a search will look for k', is empty. In this case, key k' moves to slot q in line 10, and the

Figure 11.6 Deletion in a hash table that uses linear probing. The hash table has size 10 with $h_1(k) = k \bmod 10$. **(a)** The hash table after inserting keys in the order 74, 43, 93, 18, 82, 38, 92. **(b)** The hash table after deleting the key 43 from slot 3. Key 93 moves up to slot 3 to keep it accessible, and then key 92 moves up to slot 5 just vacated by key 93. No other keys need to be moved.

search continues, to see whether any later key also needs to be moved to the slot q' that was just freed up when k' moved.

```
LINEAR-PROBING-HASH-DELETE(T, q)
while TRUE
    T[q] = NIL                          // make slot q empty
    q' = q                              // starting point for search
    repeat
        q' = (q' + 1) mod m             // next slot number with linear probing
        k' = T[q']                      // next key to try to move
        if k' == NIL
            return                      // return when an empty slot is found
    until g(k', q) < g(k', q')          // was empty slot q probed before q'?
    T[q] = k'                           // move k' into slot q
    q = q'                              // free up slot q'
```

Analysis of linear probing

Linear probing is popular to implement, but it exhibits a phenomenon known as ***primary clustering***. Long runs of occupied slots build up, increasing the average

search time. Clusters arise because an empty slot preceded by i full slots gets filled next with probability $(i + 1)/m$. Long runs of occupied slots tend to get longer, and the average search time increases.

In the standard RAM model, primary clustering is a problem, and general double hashing usually performs better than linear probing. By contrast, in a hierarchical memory model, primary clustering is a beneficial property, as elements are often stored together in the same cache block. Searching proceeds through one cache block before advancing to search the next cache block. With linear probing, the running time for a key k of HASH-INSERT, HASH-SEARCH, or LINEAR-PROBING-HASH-DELETE is at most proportional to the distance from $h_1(k)$ to the next empty slot.

The following theorem is due to Pagh et al. [351]. A more recent proof is given by Thorup [438]. We omit the proof here. The need for 5-independence is by no means obvious; see the cited proofs.

Theorem 11.9
If h_1 is 5-independent and $\alpha \leq 2/3$, then it takes expected constant time to search for, insert, or delete a key in a hash table using linear probing. ■

(Indeed, the expected operation time is $O(1/\epsilon^2)$ for $\alpha = 1 - \epsilon$.)

★ 11.5.2 Hash functions for hierarchical memory models

This section illustrates an approach for designing efficient hash tables in a modern computer system having a memory hierarchy.

Because of the memory hierarchy, linear probing is a good choice for resolving collisions, as probe sequences are sequential and tend to stay within cache blocks. But linear probing is most efficient when the hash function is complex (for example, 5-independent as in Theorem 11.9). Fortunately, having a memory hierarchy means that complex hash functions can be implemented efficiently.

As noted in Section 11.3.5, one approach is to use a cryptographic hash function such as SHA-256. Such functions are complex and sufficiently random for hash table applications. On machines with specialized instructions, cryptographic functions can be quite efficient.

Instead, we present here a simple hash function based only on addition, multiplication, and swapping the halves of a word. This function can be implemented entirely within the fast registers, and on a machine with a memory hierarchy, its latency is small compared with the time taken to access a random slot of the hash table. It is related to the RC6 encryption algorithm and can for practical purposes be considered a "random oracle."

The wee hash function

Let w denote the word size of the machine (e.g., $w = 64$), assumed to be even, and let a and b be w-bit unsigned (nonnegative) integers such that a is odd. Let $\text{swap}(x)$ denote the w-bit result of swapping the two $w/2$-bit halves of w-bit input x. That is,

$$\text{swap}(x) = (x \ggg (w/2)) + (x \lll (w/2))$$

where "$\ggg$" is "logical right shift" (as in equation (11.2)) and "$\lll$ is "left shift." Define

$$f_a(k) = \text{swap}((2k^2 + ak) \bmod 2^w) .$$

Thus, to compute $f_a(k)$, evaluate the quadratic function $2k^2 + ak$ modulo 2^w and then swap the left and right halves of the result.

Let r denote a desired number of "rounds" for the computation of the hash function. We'll use $r = 4$, but the hash function is well defined for any nonnegative r. Denote by $f_a^{(r)}(k)$ the result of iterating f_a a total of r times (that is, r rounds) starting with input value k. For any odd a and any $r \geq 0$, the function $f_a^{(r)}$, although complicated, is one-to-one (see Exercise 11.5-1). A cryptographer would view $f_a^{(r)}$ as a simple block cipher operating on w-bit input blocks, with r rounds and key a.

We first define the wee hash function h for short inputs, where by "short" we means "whose length t is at most w-bits," so that the input fits within one computer word. We would like inputs of different lengths to be hashed differently. The ***wee hash function*** $h_{a,b,t,r}(k)$ for parameters a, b, and r on t-bit input k is defined as

$$h_{a,b,t,r}(k) = \left(f_{a+2t}^{(r)}(k + b)\right) \bmod m . \tag{11.7}$$

That is, the hash value for t-bit input k is obtained by applying $f_{a+2t}^{(r)}$ to $k + b$, then taking the final result modulo m. Adding the value b provides hash-dependent randomization of the input, in a way that ensures that for variable-length inputs the 0-length input does not have a fixed hash value. Adding the value $2t$ to a ensures that the hash function acts differently for inputs of different lengths. (We use $2t$ rather than t to ensure that the key $a + 2t$ is odd if a is odd.) We call this hash function "wee" because it uses a tiny amount of memory—more precisely, it can be implemented efficiently using only the computer's fast registers. (This hash function does not have a name in the literature; it is a variant we developed for this textbook.)

Speed of the wee hash function

It is surprising how much efficiency can be bought with locality. Experiments (unpublished, by the authors) suggest that evaluating the wee hash function takes less

time than probing a *single* randomly chosen slot in a hash table. These experiments were run on a laptop (2019 MacBook Pro) with $w = 64$ and $a = 123$. For large hash tables, evaluating the wee hash function was 2 to 10 times faster than performing a single probe of the hash table.

The wee hash function for variable-length inputs

Sometimes inputs are long—more than one w-bit word in length—or have variable length, as discussed in Section 11.3.5. We can extend the wee hash function, defined above for inputs that are at most single w-bit word in length, to handle long or variable-length inputs. Here is one method for doing so.

Suppose that an input k has length t (measured in bits). Break k into a sequence $\langle k_1, k_2, \ldots, k_u \rangle$ of w-bit words, where $u = \lceil t/w \rceil$, k_1 contains the least-significant w bits of k, and k_u contains the most significant bits. If t is not a multiple of w, then k_u contains fewer than w bits, in which case, pad out the unused high-order bits of k_u with 0-bits. Define the function chop to return a sequence of the w-bit words in k:

$$\text{chop}(k) = \langle k_1, k_2, \ldots, k_u \rangle \ .$$

The most important property of the chop operation is that it is one-to-one, given t: for any two t-bit keys k and k', if $k \neq k'$ then $\text{chop}(k) \neq \text{chop}(k')$, and k can be derived from $\text{chop}(k)$ and t. The chop operation also has the useful property that a single-word input key yields a single-word output sequence: $\text{chop}(k) = \langle k \rangle$.

With the chop function in hand, we specify the wee hash function $h_{a,b,t,r}(k)$ for an input k of length t bits as follows:

$$h_{a,b,t,r}(k) = \textsc{Wee}(k, a, b, t, r, m) \ ,$$

where the procedure WEE defined on the facing page iterates through the elements of the w-bit words returned by $\text{chop}(k)$, applying f_a^r to the sum of the current word k_i and the previously computed hash value so far, finally returning the result obtained modulo m. This definition for variable-length and long (multiple-word) inputs is a consistent extension of the definition in equation (11.7) for short (single-word) inputs. For practical use, we recommend that a be a randomly chosen odd w-bit word, b be a randomly chosen w-bit word, and that $r = 4$.

Note that the wee hash function is really a hash function family, with individual hash functions determined by parameters a, b, t, r, and m. The (approximate) 5-independence of the wee hash function family for variable-length inputs can be argued based on the assumption that the 1-word wee hash function is a random oracle and on the security of the cipher-block-chaining message authentication code (CBC-MAC), as studied by Bellare et al. [42]. The case here is actually simpler than that studied in the literature, since if two messages have different lengths t and t', then their "keys" are different: $a + 2t \neq a + 2t'$. We omit the details.

```
WEE(k, a, b, t, r, m)
u = ⌈t/w⌉
⟨k_1, k_2, ..., k_u⟩ = chop(k)
q = b
for i = 1 to u
    q = f^(r)_{a+2t}(k_i + q)
return q mod m
```

This definition of a cryptographically inspired hash-function family is meant to be realistic, yet only illustrative, and many variations and improvements are possible. See the chapter notes for suggestions.

In summary, we see that when the memory system is hierarchical, it becomes advantageous to use linear probing (a special case of double hashing), since successive probes tend to stay in the same cache block. Furthermore, hash functions that can be implemented using only the computer's fast registers are exceptionally efficient, so they can be quite complex and even cryptographically inspired, providing the high degree of independence needed for linear probing to work most efficiently.

Exercises

★ ***11.5-1***

Complete the argument that for any odd positive integer a and any integer $r \geq 0$, the function $f_a^{(r)}$ is one-to-one. Use a proof by contradiction and make use of the fact that the function f_a works modulo 2^w.

★ ***11.5-2***

Argue that a random oracle is 5-independent.

★ ***11.5-3***

Consider what happens to the value $f_a^{(r)}(k)$ as we flip a single bit k_i of the input value k, for various values of r. Let $k = \sum_{i=0}^{w-1} k_i 2^i$ and $g_a(k) = \sum_{j=0}^{w-1} b_j 2^j$ define the bit values k_i in the input (with k_0 the least-significant bit) and the bit values b_j in $g_a(k) = (2k^2 + ak) \bmod 2^w$ (where $g_a(k)$ is the value that, when its halves are swapped, becomes $f_a(k)$). Suppose that flipping a single bit k_i of the input k may cause any bit b_j of $g_a(k)$ to flip, for $j \geq i$. What is the least value of r for which flipping the value of any single bit k_i may cause *any* bit of the output $f_a^{(r)}(k)$ to flip? Explain.

Problems

11-1 Longest-probe bound for hashing

Suppose you are using an open-addressed hash table of size m to store $n \le m/2$ items.

a. Assuming independent uniform permutation hashing, show that for $i = 1, 2, \ldots, n$, the probability is at most 2^{-p} that the ith insertion requires strictly more than p probes.

b. Show that for $i = 1, 2, \ldots, n$, the probability is $O(1/n^2)$ that the ith insertion requires more than $2 \lg n$ probes.

Let the random variable X_i denote the number of probes required by the ith insertion. You have shown in part (b) that $\Pr\{X_i > 2 \lg n\} = O(1/n^2)$. Let the random variable $X = \max\{X_i : 1 \le i \le n\}$ denote the maximum number of probes required by any of the n insertions.

c. Show that $\Pr\{X > 2 \lg n\} = O(1/n)$.

d. Show that the expected length $\mathrm{E}[X]$ of the longest probe sequence is $O(\lg n)$.

11-2 Searching a static set

You are asked to implement a searchable set of n elements in which the keys are numbers. The set is static (no INSERT or DELETE operations), and the only operation required is SEARCH. You are given an arbitrary amount of time to preprocess the n elements so that SEARCH operations run quickly.

a. Show how to implement SEARCH in $O(\lg n)$ worst-case time using no extra storage beyond what is needed to store the elements of the set themselves.

b. Consider implementing the set by open-address hashing on m slots, and assume independent uniform permutation hashing. What is the minimum amount of extra storage $m - n$ required to make the average performance of an unsuccessful SEARCH operation be at least as good as the bound in part (a)? Your answer should be an asymptotic bound on $m - n$ in terms of n.

11-3 Slot-size bound for chaining

Given a hash table with n slots, with collisions resolved by chaining, suppose that n keys are inserted into the table. Each key is equally likely to be hashed to each slot. Let M be the maximum number of keys in any slot after all the keys have

been inserted. Your mission is to prove an $O(\lg n / \lg\lg n)$ upper bound on $\mathrm{E}[M]$, the expected value of M.

a. Argue that the probability Q_k that exactly k keys hash to a particular slot is given by

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k} .$$

b. Let P_k be the probability that $M = k$, that is, the probability that the slot containing the most keys contains k keys. Show that $P_k \le n Q_k$.

c. Show that $Q_k < e^k / k^k$. *Hint:* Use Stirling's approximation, equation (3.25) on page 67.

d. Show that there exists a constant $c > 1$ such that $Q_{k_0} < 1/n^3$ for $k_0 = c \lg n / \lg\lg n$. Conclude that $P_k < 1/n^2$ for $k \ge k_0 = c \lg n / \lg\lg n$.

e. Argue that

$$\mathrm{E}[M] \le \Pr\left\{M > \frac{c \lg n}{\lg\lg n}\right\} \cdot n + \Pr\left\{M \le \frac{c \lg n}{\lg\lg n}\right\} \cdot \frac{c \lg n}{\lg\lg n} .$$

Conclude that $\mathrm{E}[M] = O(\lg n / \lg\lg n)$.

11-4 Hashing and authentication

Let $\mathcal{H}$ be a family of hash functions in which each hash function $h \in \mathcal{H}$ maps the universe U of keys to $\{0, 1, \ldots, m-1\}$.

a. Show that if the family $\mathcal{H}$ of hash functions is 2-independent, then it is universal.

b. Suppose that the universe U is the set of n-tuples of values drawn from $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, where p is prime. Consider an element $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle \in U$. For any n-tuple $a = \langle a_0, a_1, \ldots, a_{n-1} \rangle \in U$, define the hash function h_a by

$$h_a(x) = \left(\sum_{j=0}^{n-1} a_j x_j\right) \bmod p .$$

Let $\mathcal{H} = \{h_a : a \in U\}$. Show that $\mathcal{H}$ is universal, but not 2-independent. (*Hint:* Find a key for which all hash functions in $\mathcal{H}$ produce the same value.)

c. Suppose that we modify $\mathcal{H}$ slightly from part (b): for any $a \in U$ and for any $b \in \mathbb{Z}_p$, define

$$h'_{ab}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p$$

and $\mathcal{H}' = \{h'_{ab} : a \in U \text{ and } b \in \mathbb{Z}_p\}$. Argue that $\mathcal{H}'$ is 2-independent. (*Hint:* Consider fixed n-tuples $x \in U$ and $y \in U$, with $x_i \neq y_i$ for some i. What happens to $h'_{ab}(x)$ and $h'_{ab}(y)$ as a_i and b range over $\mathbb{Z}_p$?)

d. Alice and Bob secretly agree on a hash function h from a 2-independent family $\mathcal{H}$ of hash functions. Each $h \in \mathcal{H}$ maps from a universe of keys U to $\mathbb{Z}_p$, where p is prime. Later, Alice sends a message m to Bob over the internet, where $m \in U$. She authenticates this message to Bob by also sending an authentication tag $t = h(m)$, and Bob checks that the pair (m, t) he receives indeed satisfies $t = h(m)$. Suppose that an adversary intercepts (m, t) en route and tries to fool Bob by replacing the pair (m, t) with a different pair (m', t'). Argue that the probability that the adversary succeeds in fooling Bob into accepting (m', t') is at most $1/p$, no matter how much computing power the adversary has, even if the adversary knows the family $\mathcal{H}$ of hash functions used.

Chapter notes

The books by Knuth [261] and Gonnet and Baeza-Yates [193] are excellent references for the analysis of hashing algorithms. Knuth credits H. P. Luhn (1953) for inventing hash tables, along with the chaining method for resolving collisions. At about the same time, G. M. Amdahl originated the idea of open addressing. The notion of a random oracle was introduced by Bellare et al. [43]. Carter and Wegman [80] introduced the notion of universal families of hash functions in 1979.

Dietzfelbinger et al. [113] invented the multiply-shift hash function and gave a proof of Theorem 11.5. Thorup [437] provides extensions and additional analysis. Thorup [438] gives a simple proof that linear probing with 5-independent hashing takes constant expected time per operation. Thorup also describes the method for deletion in a hash table using linear probing.

Fredman, Komlós, and Szemerédi [154] developed a perfect hashing scheme for static sets—"perfect" because all collisions are avoided. An extension of their method to dynamic sets, handling insertions and deletions in amortized expected time $O(1)$, has been given by Dietzfelbinger et al. [114].

The wee hash function is based on the RC6 encryption algorithm [379]. Leiserson et al. [292] propose an "RC6MIX" function that is essentially the same as the

wee hash function. They give experimental evidence that it has good randomness, and they also give a "DOTMIX" function for dealing with variable-length inputs. Bellare et al. [42] provide an analysis of the security of the cipher-block-chaining message authentication code. This analysis implies that the wee hash function has the desired pseudorandomness properties.

12 Binary Search Trees

The search tree data structure supports each of the dynamic-set operations listed on page 250: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, you can use a search tree both as a dictionary and as a priority queue.

Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(\lg n)$ worst-case time. If the tree is a linear chain of n nodes, however, the same operations take $\Theta(n)$ worst-case time. In Chapter 13, we'll see a variation of binary search trees, red-black trees, whose operations guarantee a height of $O(\lg n)$. We won't prove it here, but if you build a binary search tree on a random set of n keys, its expected height is $O(\lg n)$ even if you don't try to limit its height.

After presenting the basic properties of binary search trees, the following sections show how to walk a binary search tree to print its values in sorted order, how to search for a value in a binary search tree, how to find the minimum or maximum element, how to find the predecessor or successor of an element, and how to insert into or delete from a binary search tree. The basic mathematical properties of trees appear in Appendix B.

12.1 What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree, as shown in Figure 12.1. You can represent such a tree with a linked data structure, as in Section 10.3. In addition to a *key* and satellite data, each node object contains attributes *left*, *right*, and *p* that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate attribute contains the value NIL. The tree itself has an attribute *root*

Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. The top figure shows how to view the tree conceptually, and the bottom figure shows the *left*, *right*, and p attributes in each node, in the style of Figure 10.6 on page 266. **(b)** A less efficient binary search tree, with height 4, that contains the same keys.

that points to the root node, or NIL if the tree is empty. The root node $T.root$ is the only node in a tree T whose parent is NIL.

The keys in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree property***:

> Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \le x.key$. If y is a node in the right subtree of x, then $y.key \ge x.key$.

Thus, in Figure 12.1(a), the key of the root is 6, the keys 2, 5, and 5 in its left subtree are no larger than 6, and the keys 7 and 8 in its right subtree are no smaller than 6. The same property holds for every node in the tree. For example, looking at the root's left child as the root of a subtree, this subtree root has the key 5, the key 2 in its left subtree is no larger than 5, and the key 5 in its right subtree is no smaller than 5.

Because of the binary-search-tree property, you can print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an ***inorder tree walk***, given by the procedure INORDER-TREE-WALK. This algorithm is so named because it prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree. (Similarly, a ***preorder tree walk*** prints the root before the values in either subtree, and a ***postorder tree walk*** prints the root after the values in its subtrees.) To print all the elements in a binary search tree T, call INORDER-TREE-WALK$(T.root)$. For example, the inorder tree walk prints the keys in each of the two binary search trees from Figure 12.1 in the order $2, 5, 5, 6, 7, 8$. The correctness of the algorithm follows by induction directly from the binary-search-tree property.

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

It takes $\Theta(n)$ time to walk an n-node binary search tree, since after the initial call, the procedure calls itself recursively exactly twice for each node in the tree—once for its left child and once for its right child. The following theorem gives a formal proof that it takes linear time to perform an inorder tree walk.

Theorem 12.1
If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

Proof Let $T(n)$ denote the time taken by INORDER-TREE-WALK when it is called on the root of an n-node subtree. Since INORDER-TREE-WALK visits all n nodes of the subtree, we have $T(n) = \Omega(n)$. It remains to show that $T(n) = O(n)$.

Since INORDER-TREE-WALK takes a small, constant amount of time on an empty subtree (for the test $x \neq$ NIL), we have $T(0) = c$ for some constant $c > 0$.

For $n > 0$, suppose that INORDER-TREE-WALK is called on a node x whose left subtree has k nodes and whose right subtree has $n - k - 1$ nodes. The time to perform INORDER-TREE-WALK(x) is bounded by $T(n) \leq T(k) + T(n-k-1) + d$ for some constant $d > 0$ that reflects an upper bound on the time to execute the body of INORDER-TREE-WALK(x), exclusive of the time spent in recursive calls.

We use the substitution method to show that $T(n) = O(n)$ by proving that $T(n) \leq (c+d)n + c$. For $n = 0$, we have $(c+d) \cdot 0 + c = c = T(0)$. For $n > 0$, we have

$$\begin{aligned} T(n) &\leq T(k) + T(n-k-1) + d \\ &\leq ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\ &= (c+d)n + c - (c+d) + c + d \\ &= (c+d)n + c \ , \end{aligned}$$

which completes the proof. ■

Exercises

12.1-1

For the set $\{1, 4, 5, 10, 16, 17, 21\}$ of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.

12.1-2

What is the difference between the binary-search-tree property and the min-heap property on page 163? Can the min-heap property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time? Show how, or explain why not.

12.1-3

Give a nonrecursive algorithm that performs an inorder tree walk. (*Hint:* An easy solution uses a stack as an auxiliary data structure. A more complicated, but elegant, solution uses no stack but assumes that you can test two pointers for equality.)

12.1-4

Give recursive algorithms that perform preorder and postorder tree walks in $\Theta(n)$ time on a tree of n nodes.

12.1-5

Argue that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \lg n)$ time in the worst case.

12.2 Querying a binary search tree

Binary search trees can support the queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR, as well as SEARCH. This section examines these operations and shows how to support each one in $O(h)$ time on any binary search tree of height h.

Searching

To search for a node with a given key in a binary search tree, call the TREE-SEARCH procedure. Given a pointer x to the root of a subtree and a key k, TREE-SEARCH(x, k) returns a pointer to a node with key k if one exists in the subtree; otherwise, it returns NIL. To search for key k in the entire binary search tree T, call TREE-SEARCH$(T.root, k)$.

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)

ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

The TREE-SEARCH procedure begins its search at the root and traces a simple path downward in the tree, as shown in Figure 12.2(a). For each node x it encounters, it compares the key k with $x.key$. If the two keys are equal, the search terminates. If k is smaller than $x.key$, the search continues in the left subtree of x, since the binary-search-tree property implies that k cannot reside in the right subtree. Symmetrically, if k is larger than $x.key$, the search continues in the right subtree. The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of TREE-SEARCH is $O(h)$, where h is the height of the tree.

Figure 12.2 Queries on a binary search tree. Nodes and paths followed in each query are colored blue. **(a)** A search for the key 13 in the tree follows the path $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$ from the root. **(b)** The minimum key in the tree is 2, which is found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. **(c)** The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. **(d)** The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

Since the TREE-SEARCH procedure recurses on either the left subtree or the right subtree, but not both, we can rewrite the algorithm to "unroll" the recursion into a **while** loop. On most computers, the ITERATIVE-TREE-SEARCH procedure on the facing page is more efficient.

Minimum and maximum

To find an element in a binary search tree whose key is a minimum, just follow *left* child pointers from the root until you encounter a NIL, as shown in Figure 12.2(b).

The TREE-MINIMUM procedure returns a pointer to the minimum element in the subtree rooted at a given node x, which we assume to be non-NIL.

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x

TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

The binary-search-tree property guarantees that TREE-MINIMUM is correct. If node x has no left subtree, then since every key in the right subtree of x is at least as large as $x.key$, the minimum key in the subtree rooted at x is $x.key$. If node x has a left subtree, then since no key in the right subtree is smaller than $x.key$ and every key in the left subtree is not larger than $x.key$, the minimum key in the subtree rooted at x resides in the subtree rooted at $x.left$.

The pseudocode for TREE-MAXIMUM is symmetric. Both TREE-MINIMUM and TREE-MAXIMUM run in $O(h)$ time on a tree of height h since, as in TREE-SEARCH, the sequence of nodes encountered forms a simple path downward from the root.

Successor and predecessor

Given a node in a binary search tree, how can you find its successor in the sorted order determined by an inorder tree walk? If all keys are distinct, the successor of a node x is the node with the smallest key greater than $x.key$. Regardless of whether the keys are distinct, we define the ***successor*** of a node as the next node visited in an inorder tree walk. The structure of a binary search tree allows you to determine the successor of a node without comparing keys. The TREE-SUCCESSOR procedure on the facing page returns the successor of a node x in a binary search tree if it exists, or NIL if x is the last node that would be visited during an inorder walk.

The code for TREE-SUCCESSOR has two cases. If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x's right subtree, which line 2 finds by calling TREE-MINIMUM($x.right$). For example, the successor of the node with key 15 in Figure 12.2(c) is the node with key 17.

On the other hand, as Exercise 12.2-6 asks you to show, if the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)   // leftmost node in right subtree
3  else // find the lowest ancestor of x whose left child is an ancestor of x
4      y = x.p
5      while y ≠ NIL and x == y.right
6          x = y
7          y = y.p
8      return y
```

left child is also an ancestor of x. In Figure 12.2(d), the successor of the node with key 13 is the node with key 15. To find y, go up the tree from x until you encounter either the root or a node that is the left child of its parent. Lines 4–8 of TREE-SUCCESSOR handle this case.

The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$, since it either follows a simple path up the tree or follows a simple path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in $O(h)$ time.

In summary, we have proved the following theorem.

Theorem 12.2

The dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can be implemented so that each one runs in $O(h)$ time on a binary search tree of height h. ■

Exercises

12.2-1

You are searching for the number 363 in a binary search tree containing numbers between 1 and 1000. Which of the following sequences *cannot* be the sequence of nodes examined?

a. 2, 252, 401, 398, 330, 344, 397, 363.

b. 924, 220, 911, 244, 898, 258, 362, 363.

c. 925, 202, 911, 240, 912, 245, 363.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

e. 935, 278, 347, 621, 299, 392, 358, 363.

12.2-2
Write recursive versions of TREE-MINIMUM and TREE-MAXIMUM.

12.2-3
Write the TREE-PREDECESSOR procedure.

12.2-4
Professor Kilmer claims to have discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up at a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Kilmer claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \le b \le c$. Give a smallest possible counterexample to the professor's claim.

12.2-5
Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

12.2-6
Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)

12.2-7
An alternative method of performing an inorder tree walk of an n-node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making $n - 1$ calls to TREE-SUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

12.2-8
Prove that no matter what node you start at in a height-h binary search tree, k successive calls to TREE-SUCCESSOR take $O(k + h)$ time.

12.2-9
Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that $y.key$ is either the smallest key in T larger than $x.key$ or the largest key in T smaller than $x.key$.

12.3 Insertion and deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold. We'll see that modifying the tree to insert a new element is relatively straightforward, but deleting a node from a binary search tree is more complicated.

Insertion

The TREE-INSERT procedure inserts a new node into a binary search tree. The procedure takes a binary search tree T and a node z for which $z.key$ has already been filled in, $z.left = \text{NIL}$, and $z.right = \text{NIL}$. It modifies T and some of the attributes of z so as to insert z into an appropriate position in the tree.

```
TREE-INSERT(T, z)
1  x = T.root              // node being compared with z
2  y = NIL                 // y will be parent of z
3  while x ≠ NIL           // descend until reaching a leaf
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y                 // found the location—insert z with parent y
9  if y == NIL
10     T.root = z          // tree T was empty
11 elseif z.key < y.key
12     y.left = z
13 else y.right = z
```

Figure 12.3 shows how TREE-INSERT works. Just like the procedures TREE-SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT begins at the root of the tree and the pointer x traces a simple path downward looking for a NIL to replace with the input node z. The procedure maintains the ***trailing pointer*** y as the parent of x. After initialization, the **while** loop in lines 3–7 causes these two pointers to move down the tree, going left or right depending on the comparison of $z.key$ with $x.key$, until x becomes NIL. This NIL occupies the position where node z will go. More precisely, this NIL is a *left* or *right* attribute of the node that will become z's parent, or it is $T.root$ if tree T is currently empty. The procedure needs the

Figure 12.3 Inserting a node with key 13 into a binary search tree. The simple path from the root down to the position where the node is inserted is shown in blue. The new node and the link to its parent are highlighted in orange.

trailing pointer y, because by the time it finds the NIL where z belongs, the search has proceeded one step beyond the node that needs to be changed. Lines 8–13 set the pointers that cause z to be inserted.

Like the other primitive operations on search trees, the procedure TREE-INSERT runs in $O(h)$ time on a tree of height h.

Deletion

The overall strategy for deleting a node z from a binary search tree T has three basic cases and, as we'll see, one of the cases is a bit tricky.

- If z has no children, then simply remove it by modifying its parent to replace z with NIL as its child.
- If z has just one child, then elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.
- If z has two children, find z's successor y—which must belong to z's right subtree—and move y to take z's position in the tree. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree. Because y is z's successor, it cannot have a left child, and y's original right child moves into y's original position, with the rest of y's original right subtree following automatically. This case is the tricky one because, as we'll see, it matters whether y is z's right child.

The procedure for deleting a given node z from a binary search tree T takes as arguments pointers to T and z. It organizes its cases a bit differently from the three cases outlined previously by considering the four cases shown in Figure 12.4.

- If z has no left child, then as in part (a) of the figure, replace z by its right child, which may or may not be NIL. When z's right child is NIL, this case deals with

Figure 12.4 Deleting a node z, in blue, from a binary search tree. Node z may be the root, a left child of node q, or a right child of q. The node that will replace node z in its position in the tree is colored orange. **(a)** Node z has no left child. Replace z by its right child r, which may or may not be NIL. **(b)** Node z has a left child l but no right child. Replace z by l. **(c)** Node z has two children. Its left child is node l, its right child is its successor y (which has no left child), and y's right child is node x. Replace z by y, updating y's left child to become l, but leaving x as y's right child. **(d)** Node z has two children (left child l and right child r), and its successor $y \neq r$ lies within the subtree rooted at r. First replace y by its own right child x, and set y to be r's parent. Then set y to be q's child and the parent of l.

the situation in which z has no children. When z's right child is non-NIL, this case handles the situation in which z has just one child, which is its right child.

- Otherwise, if z has just one child, then that child is a left child. As in part (b) of the figure, replace z by its left child.
- Otherwise, z has both a left and a right child. Find z's successor y, which lies in z's right subtree and has no left child (see Exercise 12.2-5). Splice node y out of its current location and replace z by y in the tree. How to do so depends on whether y is z's right child:
  - If y is z's right child, then as in part (c) of the figure, replace z by y, leaving y's right child alone.
  - Otherwise, y lies within z's right subtree but is not z's right child. In this case, as in part (d) of the figure, first replace y by its own right child, and then replace z by y.

As part of the process of deleting a node, subtrees need to move around within the binary search tree. The subroutine TRANSPLANT replaces one subtree as a child of its parent with another subtree. When TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent becomes node v's parent, and u's parent ends up having v as its appropriate child. TRANSPLANT allows v to be NIL instead of a pointer to a node.

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

Here is how TRANSPLANT works. Lines 1–2 handle the case in which u is the root of T. Otherwise, u is either a left child or a right child of its parent. Lines 3–4 take care of updating $u.p.left$ if u is a left child, and line 5 updates $u.p.right$ if u is a right child. Because v may be NIL, lines 6–7 update $v.p$ only if v is non-NIL. The procedure TRANSPLANT does not attempt to update $v.left$ and $v.right$. Doing so, or not doing so, is the responsibility of TRANSPLANT's caller.

The procedure TREE-DELETE on the facing page uses TRANSPLANT to delete node z from binary search tree T. It executes the four cases as follows. Lines 1–2 handle the case in which node z has no left child (Figure 12.4(a)), and lines 3–4

handle the case in which z has a left child but no right child (Figure 12.4(b)). Lines 5–12 deal with the remaining two cases, in which z has two children. Line 5 finds node y, which is the successor of z. Because z has a nonempty right subtree, its successor must be the node in that subtree with the smallest key; hence the call to TREE-MINIMUM($z.right$). As we noted before, y has no left child. The procedure needs to splice y out of its current location and replace z by y in the tree. If y is z's right child (Figure 12.4(c)), then lines 10–12 replace z as a child of its parent by y and replace y's left child by z's left child. Node y retains its right child (x in Figure 12.4(c)), and so no change to $y.right$ needs to occur. If y is not z's right child (Figure 12.4(d)), then two nodes have to move. Lines 7–9 replace y as a child of its parent by y's right child (x in Figure 12.4(c)) and make z's right child (r in the figure) become y's right child instead. Finally, lines 10–12 replace z as a child of its parent by y and replace y's left child by z's left child.

```
TREE-DELETE(T, z)
1  if z.left == NIL
2      TRANSPLANT(T, z, z.right)          // replace z by its right child
3  elseif z.right == NIL
4      TRANSPLANT(T, z, z.left)           // replace z by its left child
5  else y = TREE-MINIMUM(z.right)         // y is z's successor
6      if y ≠ z.right                     // is y farther down the tree?
7          TRANSPLANT(T, y, y.right)      // replace y by its right child
8          y.right = z.right              // z's right child becomes
9          y.right.p = y                  //       y's right child
10     TRANSPLANT(T, z, y)                // replace z by its successor y
11     y.left = z.left                    // and give z's left child to y,
12     y.left.p = y                       //       which had no left child
```

Each line of TREE-DELETE, including the calls to TRANSPLANT, takes constant time, except for the call to TREE-MINIMUM in line 5. Thus, TREE-DELETE runs in $O(h)$ time on a tree of height h.

In summary, we have proved the following theorem.

Theorem 12.3

The dynamic-set operations INSERT and DELETE can be implemented so that each one runs in $O(h)$ time on a binary search tree of height h. ■

Exercises

12.3-1
Give a recursive version of the TREE-INSERT procedure.

12.3-2
Suppose that you construct a binary search tree by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is 1 plus the number of nodes examined when the value was first inserted into the tree.

12.3-3
You can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

12.3-4
When TREE-DELETE calls TRANSPLANT, under what circumstances can the parameter v of TRANSPLANT be NIL?

12.3-5
Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.

12.3-6
Suppose that instead of each node x keeping the attribute $x.p$, pointing to x's parent, it keeps $x.succ$, pointing to x's successor. Give pseudocode for TREE-SEARCH, TREE-INSERT, and TREE-DELETE on a binary search tree T using this representation. These procedures should operate in $O(h)$ time, where h is the height of the tree T. You may assume that all keys in the binary search tree are distinct. (*Hint:* You might wish to implement a subroutine that returns the parent of a node.)

12.3-7
When node z in TREE-DELETE has two children, you can choose node y to be its predecessor rather than its successor. What other changes to TREE-DELETE are necessary if you do so? Some have argued that a fair strategy, giving equal priority to predecessor and successor, yields better empirical performance. How might TREE-DELETE be minimally changed to implement such a fair strategy?

Problems

12-1 Binary search trees with equal keys

Equal keys pose a problem for the implementation of binary search trees.

a. What is the asymptotic performance of TREE-INSERT when used to insert n items with identical keys into an initially empty binary search tree?

Consider changing TREE-INSERT to test whether $z.key = x.key$ before line 5 and to test whether $z.key = y.key$ before line 11. If equality holds, implement one of the following strategies. For each strategy, find the asymptotic performance of inserting n items with identical keys into an initially empty binary search tree. (The strategies are described for line 5, which compares the keys of z and x. Substitute y for x to arrive at the strategies for line 11.)

b. Keep a boolean flag $x.b$ at node x, and set x to either $x.left$ or $x.right$ based on the value of $x.b$, which alternates between FALSE and TRUE each time TREE-INSERT visits x while inserting a node with the same key as x.

c. Keep a list of nodes with equal keys at x, and insert z into the list.

d. Randomly set x to either $x.left$ or $x.right$. (Give the worst-case performance and informally derive the expected running time.)

12-2 Radix trees

Given two strings $a = a_0 a_1 \ldots a_p$ and $b = b_0 b_1 \ldots b_q$, where each a_i and each b_j belongs to some ordered set of characters, we say that string a is ***lexicographically less than*** string b if either

1. there exists an integer j, where $0 \le j \le \min\{p, q\}$, such that $a_i = b_i$ for all $i = 0, 1, \ldots, j-1$ and $a_j < b_j$, or
2. $p < q$ and $a_i = b_i$ for all $i = 0, 1, \ldots, p$.

For example, if a and b are bit strings, then 10100<10110 by rule 1 (letting $j = 3$) and 10100<101000 by rule 2. This ordering is similar to that used in English-language dictionaries.

The ***radix tree*** data structure shown in Figure 12.5 (also known as a ***trie***) stores the bit strings 1011, 10, 011, 100, and 0. When searching for a key $a = a_0 a_1 \ldots a_p$, go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. Let S be a set of distinct bit strings whose lengths sum to n. Show how to use a radix tree to sort S lexicographically in $\Theta(n)$ time. For the example in Figure 12.5, the output of the sort should be the sequence 0, 011, 10, 100, 1011.

Figure 12.5 A radix tree storing the bit strings 1011, 10, 011, 100, and 0. To determine each node's key, traverse the simple path from the root to that node. There is no need, therefore, to store the keys in the nodes. The keys appear here for illustrative purposes only. Keys corresponding to blue nodes are not in the tree. Such nodes are present only to establish a path to other nodes.

12-3 Average node depth in a randomly built binary search tree

A ***randomly built binary search tree*** on n keys is a binary search tree created by starting with an empty tree and inserting the keys in random order, where each of the $n!$ permutations of the keys is equally likely. In this problem, you will prove that the average depth of a node in a randomly built binary search tree with n nodes is $O(\lg n)$. The technique reveals a surprising similarity between the building of a binary search tree and the execution of RANDOMIZED-QUICKSORT from Section 7.3.

Denote the depth of any node x in tree T by $d(x, T)$. Then the ***total path length*** $P(T)$ of a tree T is the sum, over all nodes x in T, of $d(x, T)$.

a. Argue that the average depth of a node in T is

$$\frac{1}{n} \sum_{x \in T} d(x, T) = \frac{1}{n} P(T) .$$

Thus, you need to show that the expected value of $P(T)$ is $O(n \lg n)$.

b. Let T_L and T_R denote the left and right subtrees of tree T, respectively. Argue that if T has n nodes, then

$$P(T) = P(T_L) + P(T_R) + n - 1 .$$

c. Let $P(n)$ denote the average total path length of a randomly built binary search tree with n nodes. Show that

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1) .$$

d. Show how to rewrite $P(n)$ as

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n) .$$

e. Recalling the alternative analysis of the randomized version of quicksort given in Problem 7-3, conclude that $P(n) = O(n \lg n)$.

Each recursive invocation of randomized quicksort chooses a random pivot element to partition the set of elements being sorted. Each node of a binary search tree partitions the set of elements that fall into the subtree rooted at that node.

f. Describe an implementation of quicksort in which the comparisons to sort a set of elements are exactly the same as the comparisons to insert the elements into a binary search tree. (The order in which comparisons are made may differ, but the same comparisons must occur.)

12-4 Number of different binary trees

Let b_n denote the number of different binary trees with n nodes. In this problem, you will find a formula for b_n, as well as an asymptotic estimate.

a. Show that $b_0 = 1$ and that, for $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} .$$

b. Referring to Problem 4-5 on page 121 for the definition of a generating function, let $B(x)$ be the generating function

$$B(x) = \sum_{n=0}^{\infty} b_n x^n .$$

Show that $B(x) = xB(x)^2 + 1$, and hence one way to express $B(x)$ in closed form is

$$B(x) = \frac{1}{2x}\left(1 - \sqrt{1-4x}\right) .$$

The ***Taylor expansion*** of $f(x)$ around the point $x = a$ is given by

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k ,$$

where $f^{(k)}(x)$ is the kth derivative of f evaluated at x.

c. Show that

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(the nth ***Catalan number***) by using the Taylor expansion of $\sqrt{1-4x}$ around $x = 0$. (If you wish, instead of using the Taylor expansion, you may use the generalization of the binomial theorem, equation (C.4) on page 1181, to noninteger exponents n, where for any real number n and for any integer k, you can interpret $\binom{n}{k}$ to be $n(n-1)\cdots(n-k+1)/k!$ if $k \geq 0$, and 0 otherwise.)

d. Show that

$$b_n = \frac{4^n}{\sqrt{\pi}n^{3/2}}(1 + O(1/n)).$$

Chapter notes

Knuth [261] contains a good discussion of simple binary search trees as well as many variations. Binary search trees seem to have been independently discovered by a number of people in the late 1950s. Radix trees are often called "tries," which comes from the middle letters in the word *retrieval*. Knuth [261] also discusses them.

Many texts, including the first two editions of this book, describe a somewhat simpler method of deleting a node from a binary search tree when both of its children are present. Instead of replacing node z by its successor y, delete node y but copy its key and satellite data into node z. The downside of this approach is that the node actually deleted might not be the node passed to the delete procedure. If other components of a program maintain pointers to nodes in the tree, they could mistakenly end up with "stale" pointers to nodes that have been deleted. Although the deletion method presented in this edition of this book is a bit more complicated, it guarantees that a call to delete node z deletes node z and only node z.

Section 14.5 will show how to construct an optimal binary search tree when you know the search frequencies before constructing the tree. That is, given the frequencies of searching for each key and the frequencies of searching for values that fall between keys in the tree, a set of searches in the constructed binary search tree examines the minimum number of nodes.

13 Red-Black Trees

Chapter 12 showed that a binary search tree of height h can support any of the basic dynamic-set operations—such as SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, and DELETE—in $O(h)$ time. Thus, the set operations are fast if the height of the search tree is small. If its height is large, however, the set operations may run no faster than with a linked list. Red-black trees are one of many search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take $O(\lg n)$ time in the worst case.

13.1 Properties of red-black trees

A ***red-black tree*** is a binary search tree with one extra bit of storage per node: its ***color***, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately ***balanced***. Indeed, as we're about to see, the height of a red-black tree with n keys is at most $2\lg(n+1)$, which is $O(\lg n)$.

Each node of the tree now contains the attributes *color*, *key*, *left*, *right*, and p. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. Think of these NILs as pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as internal nodes of the tree.

A red-black tree is a binary search tree that satisfies the following ***red-black properties***:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.

4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Figure 13.1(a) shows an example of a red-black tree.

As a matter of convenience in dealing with boundary conditions in red-black tree code, we use a single sentinel to represent NIL (see page 262). For a red-black tree T, the sentinel $T.nil$ is an object with the same attributes as an ordinary node in the tree. Its *color* attribute is BLACK, and its other attributes—p, *left*, *right*, and *key*—can take on arbitrary values. As Figure 13.1(b) shows, all pointers to NIL are replaced by pointers to the sentinel $T.nil$.

Why use the sentinel? The sentinel makes it possible to treat a NIL child of a node x as an ordinary node whose parent is x. An alternative design would use a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined. That approach needlessly wastes space, however. Instead, just the one sentinel $T.nil$ represents all the NILs—all leaves and the root's parent. The values of the attributes p, *left*, *right*, and *key* of the sentinel are immaterial. The red-black tree procedures can place whatever values in the sentinel that yield simpler code.

We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values. The remainder of this chapter omits the leaves in drawings of red-black trees, as shown in Figure 13.1(c).

We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the ***black-height*** of the node, denoted $\mathrm{bh}(x)$. By property 5, the notion of black-height is well defined, since all descending simple paths from the node have the same number of black nodes. The black-height of a red-black tree is the black-height of its root.

The following lemma shows why red-black trees make good search trees.

Lemma 13.1
A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

Proof We start by showing that the subtree rooted at any node x contains at least $2^{\mathrm{bh}(x)} - 1$ internal nodes. We prove this claim by induction on the height of x. If the height of x is 0, then x must be a leaf ($T.nil$), and the subtree rooted at x indeed contains at least $2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 0$ internal nodes. For the inductive step, consider a node x that has positive height and is an internal node. Then node x has two children, either or both of which may be a leaf. If a child is black, then it contributes 1 to x's black-height but not to its own. If a child is red, then it contributes to neither x's black-height nor its own. Therefore, each child has a black-height of either $\mathrm{bh}(x) - 1$ (if it's black) or $\mathrm{bh}(x)$ (if it's red). Since the height of a child of x is less than the height of x itself, we can apply the inductive

Figure 13.1 A red-black tree. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height, where NILs have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *T.nil*, which is always black, and with black-heights omitted. The root's parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root's parent omitted entirely. The remainder of this chapter uses this drawing style.

hypothesis to conclude that each child has at least $2^{\mathrm{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{\mathrm{bh}(x)-1} - 1) + (2^{\mathrm{bh}(x)-1} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$ internal nodes, which proves the claim.

To complete the proof of the lemma, let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least $h/2$, and thus,

$$n \geq 2^{h/2} - 1 .$$

Moving the 1 to the left-hand side and taking logarithms on both sides yields $\lg(n+1) \geq h/2$, or $h \leq 2\lg(n+1)$. ■

As an immediate consequence of this lemma, each of the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR runs in $O(\lg n)$ time on a red-black tree, since each can run in $O(h)$ time on a binary search tree of height h (as shown in Chapter 12) and any red-black tree on n nodes is a binary search tree with height $O(\lg n)$. (Of course, references to NIL in the algorithms of Chapter 12 have to be replaced by $T.nil$.) Although the procedures TREE-INSERT and TREE-DELETE from Chapter 12 run in $O(\lg n)$ time when given a red-black tree as input, you cannot just use them to implement the dynamic-set operations INSERT and DELETE. They do not necessarily maintain the red-black properties, so you might not end up with a legal red-black tree. The remainder of this chapter shows how to insert into and delete from a red-black tree in $O(\lg n)$ time.

Exercises

13.1-1

In the style of Figure 13.1(a), draw the complete binary search tree of height 3 on the keys $\{1, 2, \ldots, 15\}$. Add the NIL leaves and color the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

13.1-2

Draw the red-black tree that results after TREE-INSERT is called on the tree in Figure 13.1 with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?

13.1-3

Define a ***relaxed red-black tree*** as a binary search tree that satisfies red-black properties 1, 3, 4, and 5, but whose root may be either red or black. Consider a relaxed red-black tree T whose root is red. If the root of T is changed to black but no other changes occur, is the resulting tree a red-black tree?

13.1-4
Suppose that every black node in a red-black tree "absorbs" all of its red children, so that the children of any red node become children of the black parent. (Ignore what happens to the keys.) What are the possible degrees of a black node after all its red children are absorbed? What can you say about the depths of the leaves of the resulting tree?

13.1-5
Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

13.1-6
What is the largest possible number of internal nodes in a red-black tree with black-height k? What is the smallest possible number?

13.1-7
Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

13.1-8
Argue that in a red-black tree, a red node cannot have exactly one non-NIL child.

13.2 Rotations

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take $O(\lg n)$ time. Because they modify the tree, the result may violate the red-black properties enumerated in Section 13.1. To restore these properties, colors and pointers within nodes need to change.

The pointer structure changes through ***rotation***, which is a local operation in a search tree that preserves the binary-search-tree property. Figure 13.2 shows the two kinds of rotations: left rotations and right rotations. Let's look at a left rotation on a node x, which transforms the structure on the right side of the figure to the structure on the left. Node x has a right child y, which must not be $T.nil$. The left rotation changes the subtree originally rooted at x by "twisting" the link between x and y to the left. The new root of the subtree is node y, with x as y's left child and y's original left child (the subtree represented by β in the figure) as x's right child.

The pseudocode for LEFT-ROTATE appearing on the following page assumes that $x.right \neq T.nil$ and that the root's parent is $T.nil$. Figure 13.3 shows an

Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE(T, y) transforms the configuration on the left into the configuration on the right. The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $x.key$, which precedes the keys in β, which precede $y.key$, which precedes the keys in γ.

example of how LEFT-ROTATE modifies a binary search tree. The code for RIGHT-ROTATE is symmetric. Both LEFT-ROTATE and RIGHT-ROTATE run in $O(1)$ time. Only pointers are changed by a rotation, and all other attributes in a node remain the same.

```
LEFT-ROTATE(T, x)
y = x.right
x.right = y.left          // turn y's left subtree into x's right subtree
if y.left ≠ T.nil         // if y's left subtree is not empty ...
    y.left.p = x          // ... then x becomes the parent of the subtree's root
y.p = x.p                 // x's parent becomes y's parent
if x.p == T.nil           // if x was the root ...
    T.root = y            // ... then y becomes the root
elseif x == x.p.left      // otherwise, if x was a left child ...
    x.p.left = y          // ... then y becomes a left child
else x.p.right = y        // otherwise, x was a right child, and now y is
y.left = x                // make x become y's left child
x.p = y
```

Exercises

13.2-1

Write pseudocode for RIGHT-ROTATE.

Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

13.2-2

Argue that in every n-node binary search tree, there are exactly $n - 1$ possible rotations.

13.2-3

Let a, b, and c be arbitrary nodes in subtrees α, β, and γ, respectively, in the right tree of Figure 13.2. How do the depths of a, b, and c change when a left rotation is performed on node x in the figure?

13.2-4

Show that any arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using $O(n)$ rotations. (*Hint:* First show that at most $n - 1$ right rotations suffice to transform the tree into a right-going chain.)

★ *13.2-5*

We say that a binary search tree T_1 can be ***right-converted*** to binary search tree T_2 if it is possible to obtain T_2 from T_1 via a series of calls to RIGHT-ROTATE. Give an example of two trees T_1 and T_2 such that T_1 cannot be right-converted to T_2. Then, show that if a tree T_1 can be right-converted to T_2, it can be right-converted using $O(n^2)$ calls to RIGHT-ROTATE.

13.3 Insertion

In order to insert a node into a red-black tree with n internal nodes in $O(\lg n)$ time and maintain the red-black properties, we'll need to slightly modify the TREE-INSERT procedure on page 321. The procedure RB-INSERT starts by inserting node z into the tree T as if it were an ordinary binary search tree, and then it colors z red. (Exercise 13.3-1 asks you to explain why to make node z red rather than black.) To guarantee that the red-black properties are preserved, an auxiliary procedure RB-INSERT-FIXUP on the facing page recolors nodes and performs rotations. The call RB-INSERT(T, z) inserts node z, whose *key* is assumed to have already been filled in, into the red-black tree T.

```
RB-INSERT(T, z)
1  x = T.root                  // node being compared with z
2  y = T.nil                   // y will be parent of z
3  while x ≠ T.nil             // descend until reaching the sentinel
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y                     // found the location—insert z with parent y
9  if y == T.nil
10     T.root = z              // tree T was empty
11 elseif z.key < y.key
12     y.left = z
13 else y.right = z
14 z.left = T.nil              // both of z's children are the sentinel
15 z.right = T.nil
16 z.color = RED               // the new node starts out red
17 RB-INSERT-FIXUP(T, z)       // correct any violations of red-black properties
```

The procedures TREE-INSERT and RB-INSERT differ in four ways. First, all instances of NIL in TREE-INSERT are replaced by $T.nil$. Second, lines 14–15 of RB-INSERT set $z.left$ and $z.right$ to $T.nil$, in order to maintain the proper tree structure. (TREE-INSERT assumed that z's children were already NIL.) Third, line 16 colors z red. Fourth, because coloring z red may cause a violation of one of the red-black properties, line 17 of RB-INSERT calls RB-INSERT-FIXUP(T, z) in order to restore the red-black properties.

```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left                // is z's parent a left child?
3          y = z.p.p.right                 // y is z's uncle
4          if y.color == RED               // are z's parent and uncle both red?
5              z.p.color = BLACK
6              y.color = BLACK             } case 1
7              z.p.p.color = RED
8              z = z.p.p
9          else
10             if z == z.p.right
11                 z = z.p                 } case 2
12                 LEFT-ROTATE(T, z)
13             z.p.color = BLACK
14             z.p.p.color = RED           } case 3
15             RIGHT-ROTATE(T, z.p.p)
16     else // same as lines 3–15, but with "right" and "left" exchanged
17         y = z.p.p.left
18         if y.color == RED
19             z.p.color = BLACK
20             y.color = BLACK
21             z.p.p.color = RED
22             z = z.p.p
23         else
24             if z == z.p.left
25                 z = z.p
26                 RIGHT-ROTATE(T, z)
27             z.p.color = BLACK
28             z.p.p.color = RED
29             LEFT-ROTATE(T, z.p.p)
30 T.root.color = BLACK
```

To understand how RB-INSERT-FIXUP works, let's examine the code in three major steps. First, we'll determine which violations of the red-black properties might arise in RB-INSERT upon inserting node z and coloring it red. Second, we'll consider the overall goal of the **while** loop in lines 1–29. Finally, we'll explore each of the three cases within the **while** loop's body (case 2 falls through into case 3, so these two cases are not mutually exclusive) and see how they accomplish the goal.

In describing the structure of a red-black tree, we'll often need to refer to the sibling of a node's parent. We use the term ***uncle*** for such a node.[1] Figure 13.4 shows how RB-INSERT-FIXUP operates on a sample red-black tree, with cases depending in part on the colors of a node, its parent, and its uncle.

What violations of the red-black properties might occur upon the call to RB-INSERT-FIXUP? Property 1 certainly continues to hold (every node is either red or black), as does property 3 (every leaf is black), since both children of the newly inserted red node are the sentinel $T.nil$. Property 5, which says that the number of black nodes is the same on every simple path from a given node, is satisfied as well, because node z replaces the (black) sentinel, and node z is red with sentinel children. Thus, the only properties that might be violated are property 2, which requires the root to be black, and property 4, which says that a red node cannot have a red child. Both possible violations may arise because z is colored red. Property 2 is violated if z is the root, and property 4 is violated if z's parent is red. Figure 13.4(a) shows a violation of property 4 after the node z has been inserted.

The **while** loop of lines 1–29 has two symmetric possibilities: lines 3–15 deal with the situation in which node z's parent $z.p$ is a left child of z's grandparent $z.p.p$, and lines 17–29 apply when z's parent is a right child. Our proof will focus only on lines 3–15, relying on the symmetry in lines 17–29.

We'll show that the **while** loop maintains the following three-part invariant at the start of each iteration of the loop:

a. Node z is red.

b. If $z.p$ is the root, then $z.p$ is black.

c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 2 or property 4, but not both. If the tree violates property 2, it is because z is the root and is red. If the tree violates property 4, it is because both z and $z.p$ are red.

Part (c), which deals with violations of red-black properties, is more central to showing that RB-INSERT-FIXUP restores the red-black properties than parts (a) and (b), which we'll use along the way to understand situations in the code. Because we'll be focusing on node z and nodes near it in the tree, it helps to know from part (a) that z is red. Part (b) will help show that z's grandparent $z.p.p$ exists when it's referenced in lines 2, 3, 7, 8, 14, and 15 (recall that we're focusing only on lines 3–15).

---

[1] Although we try to avoid gendered language in this book, the English language lacks a gender-neutral word for a parent's sibling.

Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies. Node z's grandparent $z.p.p$ must be black, and its blackness transfers down one level to z's parent and uncle. Once the pointer z moves up two levels in the tree, the tree shown in **(b)** results. Once again, z and its parent are both red, but this time z's uncle y is black. Since z is the right child of $z.p$, case 2 applies. Performing a left rotation results in the tree in **(c)**. Now z is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in **(d)**, which is a legal red-black tree.

Recall that to use a loop invariant, we need to show that the invariant is true upon entering the first iteration of the loop, that each iteration maintains it, that the loop terminates, and that the loop invariant gives us a useful property at loop termination. We'll see that each iteration of the loop has two possible outcomes: either the pointer z moves up the tree, or some rotations occur and then the loop terminates.

Initialization: Before RB-INSERT is called, the red-black tree has no violations. RB-INSERT adds a red node z and calls RB-INSERT-FIXUP. We'll show that each part of the invariant holds at the time RB-INSERT-FIXUP is called:

a. When RB-INSERT-FIXUP is called, z is the red node that was added.

b. If $z.p$ is the root, then $z.p$ started out black and did not change before the call of RB-INSERT-FIXUP.

c. We have already seen that properties 1, 3, and 5 hold when RB-INSERT-FIXUP is called.

   If the tree violates property 2 (the root must be black), then the red root must be the newly added node z, which is the only internal node in the tree. Because the parent and both children of z are the sentinel, which is black, the tree does not also violate property 4 (both children of a red node are black). Thus this violation of property 2 is the only violation of red-black properties in the entire tree.

   If the tree violates property 4, then, because the children of node z are black sentinels and the tree had no other violations prior to z being added, the violation must be because both z and $z.p$ are red. Moreover, the tree violates no other red-black properties.

Maintenance: There are six cases within the **while** loop, but we'll examine only the three cases in lines 3–15, when node z's parent $z.p$ is a left child of z's grandparent $z.p.p$. The proof for lines 17–29 is symmetric. The node $z.p.p$ exists, since by part (b) of the loop invariant, if $z.p$ is the root, then $z.p$ is black. Since RB-INSERT-FIXUP enters a loop iteration only if $z.p$ is red, we know that $z.p$ cannot be the root. Hence, $z.p.p$ exists.

Case 1 differs from cases 2 and 3 by the color of z's uncle y. Line 3 makes y point to z's uncle $z.p.p.right$, and line 4 tests y's color. If y is red, then case 1 executes. Otherwise, control passes to cases 2 and 3. In all three cases, z's grandparent $z.p.p$ is black, since its parent $z.p$ is red, and property 4 is violated only between z and $z.p$.

Figure 13.5 Case 1 of the procedure RB-INSERT-FIXUP. Both z and its parent $z.p$ are red, violating property 4. In case 1, z's uncle y is red. The same action occurs regardless of whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root—possibly the sentinel—and each has the same black-height. The code for case 1 moves the blackness of z's grandparent down to z's parent and uncle, preserving property 5: all downward simple paths from a node to a leaf have the same number of blacks. The **while** loop continues with node z's grandparent $z.p.p$ as the new z. If the action of case 1 causes a new violation of property 4 to occur, it must be only between the new z, which is red, and its parent, if it is red as well.

Case 1: z's uncle y is red

Figure 13.5 shows the situation for case 1 (lines 5–8), which occurs when both $z.p$ and y are red. Because z's grandparent $z.p.p$ is black, its blackness can transfer down one level to both $z.p$ and y, thereby fixing the problem of z and $z.p$ both being red. Having had its blackness transferred down one level, z's grandparent becomes red, thereby maintaining property 5. The **while** loop repeats with $z.p.p$ as the new node z, so that the pointer z moves up two levels in the tree.

Now, we show that case 1 maintains the loop invariant at the start of the next iteration. We use z to denote node z in the current iteration, and $z' = z.p.p$ to denote the node that will be called node z at the test in line 1 upon the next iteration.

a. Because this iteration colors $z.p.p$ red, node z' is red at the start of the next iteration.

b. The node $z'.p$ is $z.p.p.p$ in this iteration, and the color of this node does not change. If this node is the root, it was black prior to this iteration, and it remains black at the start of the next iteration.

Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $z.p$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise case 1 would apply), and each has the same black-height. Case 2 transforms into case 3 by a left rotation, which preserves property 5: all downward simple paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The **while** loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.

c. We have already argued that case 1 maintains property 5, and it does not introduce a violation of properties 1 or 3.

   If node z' is the root at the start of the next iteration, then case 1 corrected the lone violation of property 4 in this iteration. Since z' is red and it is the root, property 2 becomes the only one that is violated, and this violation is due to z'.

   If node z' is not the root at the start of the next iteration, then case 1 has not created a violation of property 2. Case 1 corrected the lone violation of property 4 that existed at the start of this iteration. It then made z' red and left $z'.p$ alone. If $z'.p$ was black, there is no violation of property 4. If $z'.p$ was red, coloring z' red created one violation of property 4, between z' and $z'.p$.

Case 2: z's uncle y is black and z is a right child
Case 3: z's uncle y is black and z is a left child

In cases 2 and 3, the color of z's uncle y is black. We distinguish the two cases, which assume that z's parent $z.p$ is red and a left child, according to whether z is a right or left child of $z.p$. Lines 11–12 constitute case 2, which is shown in Figure 13.6 together with case 3. In case 2, node z is a right child of its parent. A left rotation immediately transforms the situation into case 3 (lines 13–15), in which node z is a left child. Because both z and $z.p$ are red, the rotation affects neither the black-heights of nodes nor property 5. Whether case 3 executes directly or through case 2, z's uncle y is black, since otherwise case 1 would have run. Additionally, the node $z.p.p$ exists, since we have argued that this

node existed at the time that lines 2 and 3 were executed, and after moving z up one level in line 11 and then down one level in line 12, the identity of $z.p.p$ remains unchanged. Case 3 performs some color changes and a right rotation, which preserve property 5. At this point, there are no longer two red nodes in a row. The **while** loop terminates upon the next test in line 1, since $z.p$ is now black.

We now show that cases 2 and 3 maintain the loop invariant. (As we have just argued, $z.p$ will be black upon the next test in line 1, and the loop body will not execute again.)

a. Case 2 makes z point to $z.p$, which is red. No further change to z or its color occurs in cases 2 and 3.

b. Case 3 makes $z.p$ black, so that if $z.p$ is the root at the start of the next iteration, it is black.

c. As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.

   Since node z is not the root in cases 2 and 3, we know that there is no violation of property 2. Cases 2 and 3 do not introduce a violation of property 2, since the only node that is made red becomes a child of a black node by the rotation in case 3.

   Cases 2 and 3 correct the lone violation of property 4, and they do not introduce another violation.

Termination: To see that the loop terminates, observe that if only case 1 occurs, then the node pointer z moves toward the root in each iteration, so that eventually $z.p$ is black. (If z is the root, then $z.p$ is the sentinel $T.nil$, which is black.) If either case 2 or case 3 occurs, then we've seen that the loop terminates. Since the loop terminates because $z.p$ is black, the tree does not violate property 4 at loop termination. By the loop invariant, the only property that might fail to hold is property 2. Line 30 restores this property by coloring the root black, so that when RB-INSERT-FIXUP terminates, all the red-black properties hold.

Thus, we have shown that RB-INSERT-FIXUP correctly restores the red-black properties.

Analysis

What is the running time of RB-INSERT? Since the height of a red-black tree on n nodes is $O(\lg n)$, lines 1–16 of RB-INSERT take $O(\lg n)$ time. In RB-INSERT-FIXUP, the **while** loop repeats only if case 1 occurs, and then the pointer z moves two levels up the tree. The total number of times the **while** loop can be executed is therefore $O(\lg n)$. Thus, RB-INSERT takes a total of $O(\lg n)$ time. Moreover, it

never performs more than two rotations, since the **while** loop terminates if case 2 or case 3 is executed.

Exercises

13.3-1
Line 16 of RB-INSERT sets the color of the newly inserted node z to red. If instead z's color were set to black, then property 4 of a red-black tree would not be violated. Why not set z's color to black?

13.3-2
Show the red-black trees that result after successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty red-black tree.

13.3-3
Suppose that the black-height of each of the subtrees $\alpha, \beta, \gamma, \delta, \varepsilon$ in Figures 13.5 and 13.6 is k. Label each node in each figure with its black-height to verify that the indicated transformation preserves property 5.

13.3-4
Professor Teach is concerned that RB-INSERT-FIXUP might set $T.nil.color$ to RED, in which case the test in line 1 would not cause the loop to terminate when z is the root. Show that the professor's concern is unfounded by arguing that RB-INSERT-FIXUP never sets $T.nil.color$ to RED.

13.3-5
Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

13.3-6
Suggest how to implement RB-INSERT efficiently if the representation for red-black trees includes no storage for parent pointers.

13.4 Deletion

Like the other basic operations on an n-node red-black tree, deletion of a node takes $O(\lg n)$ time. Deleting a node from a red-black tree is more complicated than inserting a node.

The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure on page 325. First, we need to customize the TRANSPLANT

subroutine on page 324 that TREE-DELETE calls so that it applies to a red-black tree. Like TRANSPLANT, the new procedure RB-TRANSPLANT replaces the subtree rooted at node u by the subtree rooted at node v. The RB-TRANSPLANT procedure differs from TRANSPLANT in two ways. First, line 1 references the sentinel $T.nil$ instead of NIL. Second, the assignment to $v.p$ in line 6 occurs unconditionally: the procedure can assign to $v.p$ even if v points to the sentinel. We'll take advantage of the ability to assign to $v.p$ when $v = T.nil$.

```
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```

The procedure RB-DELETE on the next page is like the TREE-DELETE procedure, but with additional lines of pseudocode. The additional lines deal with nodes x and y that may be involved in violations of the red-black properties. When the node z being deleted has at most one child, then y will be z. When z has two children, then, as in TREE-DELETE, y will be z's successor, which has no left child and moves into z's position in the tree. Additionally, y takes on z's color. In either case, node y has at most one child: node x, which takes y's place in the tree. (Node x will be the sentinel $T.nil$ if y has no children.) Since node y will be either removed from the tree or moved within the tree, the procedure needs to keep track of y's original color. If the red-black properties might be violated after deleting node z, RB-DELETE calls the auxiliary procedure RB-DELETE-FIXUP, which changes colors and performs rotations to restore the red-black properties.

Although RB-DELETE contains almost twice as many lines of pseudocode as TREE-DELETE, the two procedures have the same basic structure. You can find each line of TREE-DELETE within RB-DELETE (with the changes of replacing NIL by $T.nil$ and replacing calls to TRANSPLANT by calls to RB-TRANSPLANT), executed under the same conditions.

In detail, here are the other differences between the two procedures:

- Lines 1 and 9 set node y as described above: line 1 when node z has at most one child and line 9 when z has two children.
- Because node y's color might change, the variable *y-original-color* stores y's color before any changes occur. Lines 2 and 10 set this variable immediately after assignments to y. When node z has two children, then nodes y and z are

```
RB-DELETE(T, z)
1  y = z
2  y-original-color = y.color
3  if z.left == T.nil
4      x = z.right
5      RB-TRANSPLANT(T, z, z.right)        // replace z by its right child
6  elseif z.right == T.nil
7      x = z.left
8      RB-TRANSPLANT(T, z, z.left)         // replace z by its left child
9  else y = TREE-MINIMUM(z.right)          // y is z's successor
10     y-original-color = y.color
11     x = y.right
12     if y ≠ z.right                      // is y farther down the tree?
13         RB-TRANSPLANT(T, y, y.right)    // replace y by its right child
14         y.right = z.right               // z's right child becomes
15         y.right.p = y                   //       y's right child
16     else x.p = y                        // in case x is T.nil
17     RB-TRANSPLANT(T, z, y)              // replace z by its successor y
18     y.left = z.left                     // and give z's left child to y,
19     y.left.p = y                        //       which had no left child
20     y.color = z.color
21 if y-original-color == BLACK        // if any red-black violations occurred,
22     RB-DELETE-FIXUP(T, x)           //       correct them
```

distinct. In this case, line 17 moves y into z's original position in the tree (that is, z's location in the tree at the time RB-DELETE was called), and line 20 gives y the same color as z. When node y was originally black, removing or moving it could cause violations of the red-black properties, which are corrected by the call of RB-DELETE-FIXUP in line 22.

- As discussed, the procedure keeps track of the node x that moves into node y's original position at the time of call. The assignments in lines 4, 7, and 11 set x to point to either y's only child or, if y has no children, the sentinel $T.nil$.
- Since node x moves into node y's original position, the attribute $x.p$ must be set correctly. If node z has two children and y is z's right child, then y just moves into z's position, with x remaining a child of y. Line 12 checks for this case. Although you might think that setting $x.p$ to y in line 16 is unnecessary since x is a child of y, the call of RB-DELETE-FIXUP relies on $x.p$ being y even if x is $T.nil$. Thus, when z has two children and y is z's right child, executing

line 16 is necessary if y's right child is $T.nil$, and otherwise it does not change anything.

Otherwise, node z is either the same as node y or it is a proper ancestor of y's original parent. In these cases, the calls of RB-TRANSPLANT in lines 5, 8, and 13 set $x.p$ correctly in line 6 of RB-TRANSPLANT. (In these calls of RB-TRANSPLANT, the third parameter passed is the same as x.)

- Finally, if node y was black, one or more violations of the red-black properties might arise. The call of RB-DELETE-FIXUP in line 22 restores the red-black properties. If y was red, the red-black properties still hold when y is removed or moved, for the following reasons:

  1. No black-heights in the tree have changed. (See Exercise 13.4-1.)
  2. No red nodes have been made adjacent. If z has at most one child, then y and z are the same node. That node is removed, with a child taking its place. If the removed node was red, then neither its parent nor its children can also be red, so moving a child to take its place cannot cause two red nodes to become adjacent. If, on the other hand, z has two children, then y takes z's place in the tree, along with z's color, so there cannot be two adjacent red nodes at y's new position in the tree. In addition, if y was not z's right child, then y's original right child x replaces y in the tree. Since y is red, x must be black, and so replacing y by x cannot cause two red nodes to become adjacent.
  3. Because y could not have been the root if it was red, the root remains black.

If node y was black, three problems may arise, which the call of RB-DELETE-FIXUP will remedy. First, if y was the root and a red child of y became the new root, property 2 is violated. Second, if both x and its new parent are red, then a violation of property 4 occurs. Third, moving y within the tree causes any simple path that previously contained y to have one less black node. Thus, property 5 is now violated by any ancestor of y in the tree. We can correct the violation of property 5 by saying that when the black node y is removed or moved, its blackness transfers to the node x that moves into y's original position, giving x an "extra" black. That is, if we add 1 to the count of black nodes on any simple path that contains x, then under this interpretation, property 5 holds. But now another problem emerges: node x is neither red nor black, thereby violating property 1. Instead, node x is either "doubly black" or "red-and-black," and it contributes either 2 or 1, respectively, to the count of black nodes on simple paths containing x. The *color* attribute of x will still be either RED (if x is red-and-black) or BLACK (if x is doubly black). In other words, the extra black on a node is reflected in x's pointing to the node rather than in the *color* attribute.

The procedure RB-DELETE-FIXUP on the next page restores properties 1, 2, and 4. Exercises 13.4-2 and 13.4-3 ask you to show that the procedure restores properties 2 and 4, and so in the remainder of this section, we focus on property 1. The goal of the **while** loop in lines 1–43 is to move the extra black up the tree until

1. x points to a red-and-black node, in which case line 44 colors x (singly) black;
2. x points to the root, in which case the extra black simply vanishes; or
3. having performed suitable rotations and recolorings, the loop exits.

Like RB-INSERT-FIXUP, the RB-DELETE-FIXUP procedure handles two symmetric situations: lines 3–22 for when node x is a left child, and lines 24–43 for when x is a right child. Our proof focuses on the four cases shown in lines 3–22.

Within the **while** loop, x always points to a nonroot doubly black node. Line 2 determines whether x is a left child or a right child of its parent $x.p$ so that either lines 3–22 or 24–43 will execute in a given iteration. The sibling of x is always denoted by a pointer w. Since node x is doubly black, node w cannot be $T.nil$, because otherwise, the number of blacks on the simple path from $x.p$ to the (singly black) leaf w would be smaller than the number on the simple path from $x.p$ to x.

Recall that the RB-DELETE procedure always assigns to $x.p$ before calling RB-DELETE-FIXUP (either within the call of RB-TRANSPLANT in line 13 or the assignment in line 16), even when node x is the sentinel $T.nil$. That is because RB-DELETE-FIXUP references x's parent $x.p$ in several places, and this attribute must point to the node that became x's parent in RB-DELETE—even if x is $T.nil$.

Figure 13.7 demonstrates the four cases in the code when node x is a left child. (As in RB-INSERT-FIXUP, the cases in RB-DELETE-FIXUP are not mutually exclusive.) Before examining each case in detail, let's look more generally at how we can verify that the transformation in each of the cases preserves property 5. The key idea is that in each case, the transformation applied preserves the number of black nodes (including x's extra black) from (and including) the root of the subtree shown to the roots of each of the subtrees $\alpha, \beta, \ldots, \zeta$. Thus, if property 5 holds prior to the transformation, it continues to hold afterward. For example, in Figure 13.7(a), which illustrates case 1, the number of black nodes from the root to the root of either subtree α or β is 3, both before and after the transformation. (Again, remember that node x adds an extra black.) Similarly, the number of black nodes from the root to the root of any of γ, δ, ε, and ζ is 2, both before and after the transformation.[2] In Figure 13.7(b), the counting must involve the value c of the *color* attribute of the root of the subtree shown, which can be either RED or BLACK.

---

[2] If property 5 holds, we can assume that paths from the roots of γ, δ, ε, and ζ down to leaves contain one more black than do paths from the roots of α and β down to leaves.

```
RB-DELETE-FIXUP(T, x)
1  while x ≠ T.root and x.color == BLACK
2      if x == x.p.left                  // is x a left child?
3          w = x.p.right                 // w is x's sibling
4          if w.color == RED
5              w.color = BLACK
6              x.p.color = RED
7              LEFT-ROTATE(T, x.p)                  } case 1
8              w = x.p.right
9          if w.left.color == BLACK and w.right.color == BLACK
10             w.color = RED                        } case 2
11             x = x.p
12         else
13             if w.right.color == BLACK
14                 w.left.color = BLACK
15                 w.color = RED
16                 RIGHT-ROTATE(T, w)               } case 3
17                 w = x.p.right
18             w.color = x.p.color
19             x.p.color = BLACK
20             w.right.color = BLACK                } case 4
21             LEFT-ROTATE(T, x.p)
22             x = T.root
23     else // same as lines 3–22, but with "right" and "left" exchanged
24         w = x.p.left
25         if w.color == RED
26             w.color = BLACK
27             x.p.color = RED
28             RIGHT-ROTATE(T, x.p)
29             w = x.p.left
30         if w.right.color == BLACK and w.left.color == BLACK
31             w.color = RED
32             x = x.p
33         else
34             if w.left.color == BLACK
35                 w.right.color = BLACK
36                 w.color = RED
37                 LEFT-ROTATE(T, w)
38                 w = x.p.left
39             w.color = x.p.color
40             x.p.color = BLACK
41             w.left.color = BLACK
42             RIGHT-ROTATE(T, x.p)
43             x = T.root
44 x.color = BLACK
```

Figure 13.7 The cases in lines 3–22 of the procedure RB-DELETE-FIXUP. Brown nodes have *color* attributes represented by c and c', which may be either RED or BLACK. The letters $\alpha, \beta, \ldots, \zeta$ represent arbitrary subtrees. Each case transforms the configuration on the left into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by x has an extra black and is either doubly black or red-and-black. Only case 2 causes the loop to repeat. **(a)** Case 1 is transformed into case 2, 3, or 4 by exchanging the colors of nodes B and D and performing a left rotation. **(b)** In case 2, the extra black represented by the pointer x moves up the tree by coloring node D red and setting x to point to node B. If case 2 is entered through case 1, the **while** loop terminates because the new node x is red-and-black, and therefore the value c of its *color* attribute is RED. **(c)** Case 3 is transformed to case 4 by exchanging the colors of nodes C and D and performing a right rotation. **(d)** Case 4 removes the extra black represented by x by changing some colors and performing a left rotation (without violating the red-black properties), and then the loop terminates.

If we define count(RED) $= 0$ and count(BLACK) $= 1$, then the number of black nodes from the root to α is $2 + \text{count}(c)$, both before and after the transformation. In this case, after the transformation, the new node x has *color* attribute c, but this node is really either red-and-black (if $c =$ RED) or doubly black (if $c =$ BLACK). You can verify the other cases similarly (see Exercise 13.4-6).

Case 1: x's sibling w is red

Case 1 (lines 5–8 and Figure 13.7(a)) occurs when node w, the sibling of node x, is red. Because w is red, it must have black children. This case switches the colors of w and $x.p$ and then performs a left-rotation on $x.p$ without violating any of the red-black properties. The new sibling of x, which is one of w's children prior to the rotation, is now black, and thus case 1 converts into one of cases 2, 3, or 4.

Cases 2, 3, and 4 occur when node w is black and are distinguished by the colors of w's children.

Case 2: x's sibling w is black, and both of w's children are black

In case 2 (lines 10–11 and Figure 13.7(b)), both of w's children are black. Since w is also black, this case removes one black from both x and w, leaving x with only one black and leaving w red. To compensate for x and w each losing one black, x's parent $x.p$ can take on an extra black. Line 11 does so by moving x up one level, so that the **while** loop repeats with $x.p$ as the new node x. If case 2 enters through case 1, the new node x is red-and-black, since the original $x.p$ was red. Hence, the value c of the *color* attribute of the new node x is RED, and the loop terminates when it tests the loop condition. Line 44 then colors the new node x (singly) black.

Case 3: x's sibling w is black, w's left child is red, and w's right child is black

Case 3 (lines 14–17 and Figure 13.7(c)) occurs when w is black, its left child is red, and its right child is black. This case switches the colors of w and its left child $w.left$ and then performs a right rotation on w without violating any of the red-black properties. The new sibling w of x is now a black node with a red right child, and thus case 3 falls through into case 4.

Case 4: x's sibling w is black, and w's right child is red

Case 4 (lines 18–22 and Figure 13.7(d)) occurs when node x's sibling w is black and w's right child is red. Some color changes and a left rotation on $x.p$ allow the extra black on x to vanish, making it singly black, without violating any of the red-black properties. Line 22 sets x to be the root, and the **while** loop terminates when it next tests the loop condition.

Analysis

What is the running time of RB-DELETE? Since the height of a red-black tree of n nodes is $O(\lg n)$, the total cost of the procedure without the call to RB-DELETE-FIXUP takes $O(\lg n)$ time. Within RB-DELETE-FIXUP, each of cases 1, 3, and 4 lead to termination after performing a constant number of color changes and at most three rotations. Case 2 is the only case in which the **while** loop can be repeated, and then the pointer x moves up the tree at most $O(\lg n)$ times, performing no rotations. Thus, the procedure RB-DELETE-FIXUP takes $O(\lg n)$ time and performs at most three rotations, and the overall time for RB-DELETE is therefore also $O(\lg n)$.

Exercises

13.4-1
Show that if node y in RB-DELETE is red, then no black-heights change.

13.4-2
Argue that after RB-DELETE-FIXUP executes, the root of the tree must be black.

13.4-3
Argue that if in RB-DELETE both x and $x.p$ are red, then property 4 is restored by the call to RB-DELETE-FIXUP(T, x).

13.4-4
In Exercise 13.3-2 on page 346, you found the red-black tree that results from successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order $8, 12, 19, 31, 38, 41$.

13.4-5
Which lines of the code for RB-DELETE-FIXUP might examine or modify the sentinel $T.nil$?

13.4-6
In each of the cases of Figure 13.7, give the count of black nodes from the root of the subtree shown to the roots of each of the subtrees $\alpha, \beta, \ldots, \zeta$, and verify that each count remains the same after the transformation. When a node has a *color* attribute c or c', use the notation count(c) or count(c') symbolically in your count.

13.4-7
Professors Skelton and Baron worry that at the start of case 1 of RB-DELETE-FIXUP, the node $x.p$ might not be black. If $x.p$ is not black, then lines 5–6 are

wrong. Show that $x.p$ must be black at the start of case 1, so that the professors need not be concerned.

13.4-8
A node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree always the same as the initial red-black tree? Justify your answer.

★ ***13.4-9***
Consider the operation RB-ENUMERATE(T, r, a, b), which outputs all the keys k such that $a \le k \le b$ in a subtree rooted at node r in an n-node red-black tree T. Describe how to implement RB-ENUMERATE in $\Theta(m + \lg n)$ time, where m is the number of keys that are output. Assume that the keys in T are unique and that the values a and b appear as keys in T. How does your solution change if a and b might not appear in T?

Problems

13-1 Persistent dynamic sets
During the course of an algorithm, you sometimes find that you need to maintain past versions of a dynamic set as it is updated. We call such a set ***persistent***. One way to implement a persistent set is to copy the entire set whenever it is modified, but this approach can slow down a program and also consume a lot of space. Sometimes, you can do much better.

Consider a persistent set S with the operations INSERT, DELETE, and SEARCH, which you implement using binary search trees as shown in Figure 13.8(a). Maintain a separate root for every version of the set. In order to insert the key 5 into the set, create a new node with key 5. This node becomes the left child of a new node with key 7, since you cannot modify the existing node with key 7. Similarly, the new node with key 7 becomes the left child of a new node with key 8 whose right child is the existing node with key 10. The new node with key 8 becomes, in turn, the right child of a new root r' with key 4 whose left child is the existing node with key 3. Thus, you copy only part of the tree and share some of the nodes with the original tree, as shown in Figure 13.8(b).

Assume that each tree node has the attributes *key*, *left*, and *right* but no parent. (See also Exercise 13.3-6 on page 346.)

a. For a persistent binary search tree (not a red-black tree, just a binary search tree), identify the nodes that need to change to insert or delete a node.

Figure 13.8 **(a)** A binary search tree with keys $2, 3, 4, 7, 8, 10$. **(b)** The persistent binary search tree that results from the insertion of key 5. The most recent version of the set consists of the nodes reachable from the root r', and the previous version consists of the nodes reachable from r. Blue nodes are added when key 5 is inserted.

b. Write a procedure PERSISTENT-TREE-INSERT(T, z) that, given a persistent binary search tree T and a node z to insert, returns a new persistent tree T' that is the result of inserting z into T. Assume that you have a procedure COPY-NODE(x) that makes a copy of node x, including all of its attributes.

c. If the height of the persistent binary search tree T is h, what are the time and space requirements of your implementation of PERSISTENT-TREE-INSERT? (The space requirement is proportional to the number of nodes that are copied.)

d. Suppose that you include the parent attribute in each node. In this case, the PERSISTENT-TREE-INSERT procedure needs to perform additional copying. Prove that PERSISTENT-TREE-INSERT then requires $\Omega(n)$ time and space, where n is the number of nodes in the tree.

e. Show how to use red-black trees to guarantee that the worst-case running time and space are $O(\lg n)$ per insertion or deletion. You may assume that all keys are distinct.

13-2 Join operation on red-black trees

The ***join*** operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $x_1.key \le x.key \le x_2.key$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.

a. Suppose that you store the black-height of a red-black tree T as the new attribute $T.bh$. Argue that RB-INSERT and RB-DELETE can maintain the bh

attribute without requiring extra storage in the nodes of the tree and without increasing the asymptotic running times. Show how to determine the black-height of each node visited while descending through T, using $O(1)$ time per node visited.

Let T_1 and T_2 be red-black trees and x be a key value such that for any nodes x_1 in T_1 and x_2 in T_2, we have $x_1.key \le x.key \le x_2.key$. You will show how to implement the operation RB-JOIN(T_1, x, T_2), which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2.

b. Assume that $T_1.bh \ge T_2.bh$. Describe an $O(\lg n)$-time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose black-height is $T_2.bh$.

c. Let T_y be the subtree rooted at y. Describe how $T_y \cup \{x\} \cup T_2$ can replace T_y in $O(1)$ time without destroying the binary-search-tree property.

d. What color should you make x so that red-black properties 1, 3, and 5 are maintained? Describe how to enforce properties 2 and 4 in $O(\lg n)$ time.

e. Argue that no generality is lost by making the assumption in part (b). Describe the symmetric situation that arises when $T_1.bh \le T_2.bh$.

f. Argue that the running time of RB-JOIN is $O(\lg n)$.

13-3 AVL trees

An ***AVL tree*** is a binary search tree that is ***height balanced***: for each node x, the heights of the left and right subtrees of x differ by at most 1. To implement an AVL tree, maintain an extra attribute h in each node such that $x.h$ is the height of node x. As for any other binary search tree T, assume that $T.root$ points to the root node.

a. Prove that an AVL tree with n nodes has height $O(\lg n)$. (*Hint:* Prove that an AVL tree of height h has at least F_h nodes, where F_h is the hth Fibonacci number.)

b. To insert into an AVL tree, first place a node into the appropriate place in binary search tree order. Afterward, the tree might no longer be height balanced. Specifically, the heights of the left and right children of some node might differ by 2. Describe a procedure BALANCE(x), which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ

by at most 2, so that $|x.right.h - x.left.h| \leq 2$, and alters the subtree rooted at x to be height balanced. The procedure should return a pointer to the node that is the root of the subtree after alterations occur. (*Hint:* Use rotations.)

c. Using part (b), describe a recursive procedure AVL-INSERT(T, z) that takes an AVL tree T and a newly created node z (whose key has already been filled in), and adds z into T, maintaining the property that T is an AVL tree. As in TREE-INSERT from Section 12.3, assume that $z.key$ has already been filled in and that $z.left = \text{NIL}$ and $z.right = \text{NIL}$. Assume as well that $z.h = 0$.

d. Show that AVL-INSERT, run on an n-node AVL tree, takes $O(\lg n)$ time and performs $O(\lg n)$ rotations.

Chapter notes

The idea of balancing a search tree is due to Adel'son-Vel'skiĭ and Landis [2], who introduced a class of balanced search trees called "AVL trees" in 1962, described in Problem 13-3. Another class of search trees, called "2-3 trees," was introduced by J. E. Hopcroft (unpublished) in 1970. A 2-3 tree maintains balance by manipulating the degrees of nodes in the tree, where each node has either two or three children. Chapter 18 covers a generalization of 2-3 trees introduced by Bayer and McCreight [39], called "B-trees."

Red-black trees were invented by Bayer [38] under the name "symmetric binary B-trees." Guibas and Sedgewick [202] studied their properties at length and introduced the red/black color convention. Andersson [16] gives a simpler-to-code variant of red-black trees. Weiss [451] calls this variant AA-trees. An AA-tree is similar to a red-black tree except that left children can never be red.

Sedgewick and Wayne [402] present red-black trees as a modified version of 2-3 trees in which a node with three children is split into two nodes with two children each. One of these nodes becomes the left child of the other, and only left children can be red. They call this structure a "left-leaning red-black binary search tree." Although the code for left-leaning red-black binary search trees is more concise than the red-black tree pseudocode in this chapter, operations on left-leaning red-black binary search trees do not limit the number of rotations per operation to a constant. This distinction will matter in Chapter 17.

Treaps, a hybrid of binary search trees and heaps, were proposed by Seidel and Aragon [404]. They are the default implementation of a dictionary in LEDA [324], which is a well-implemented collection of data structures and algorithms.

There are many other variations on balanced binary trees, including weight-balanced trees [344], k-neighbor trees [318], and scapegoat trees [174]. Perhaps

the most intriguing are the "splay trees" introduced by Sleator and Tarjan [418], which are "self-adjusting." (See Tarjan [429] for a good description of splay trees.) Splay trees maintain balance without any explicit balance condition such as color. Instead, "splay operations" (which involve rotations) are performed within the tree every time an access is made. The amortized cost (see Chapter 16) of each operation on an n-node tree is $O(\lg n)$. Splay trees have been conjectured to perform within a constant factor of the best offline rotation-based tree. The best known competitive ratio (see Chapter 27) for a rotation-based tree is the Tango Tree of Demaine et al. [109].

Skip lists [369] provide an alternative to balanced binary trees. A skip list is a linked list that is augmented with a number of additional pointers. Each dictionary operation runs in $O(\lg n)$ expected time on a skip list of n items.

Part IV Advanced Design and Analysis Techniques

Introduction

This part covers three important techniques used in designing and analyzing efficient algorithms: dynamic programming (Chapter 14), greedy algorithms (Chapter 15), and amortized analysis (Chapter 16). Earlier parts have presented other widely applicable techniques, such as divide-and-conquer, randomization, and how to solve recurrences. The techniques in this part are somewhat more sophisticated, but you will be able to use them solve many computational problems. The themes introduced in this part will recur later in this book.

Dynamic programming typically applies to optimization problems in which you make a set of choices in order to arrive at an optimal solution, each choice generates subproblems of the same form as the original problem, and the same subproblems arise repeatedly. The key strategy is to store the solution to each such subproblem rather than recompute it. Chapter 14 shows how this simple idea can sometimes transform exponential-time algorithms into polynomial-time algorithms.

Like dynamic-programming algorithms, greedy algorithms typically apply to optimization problems in which you make a set of choices in order to arrive at an optimal solution. The idea of a greedy algorithm is to make each choice in a locally optimal manner, resulting in a faster algorithm than you get with dynamic programming. Chapter 15 will help you determine when the greedy approach works.

The technique of amortized analysis applies to certain algorithms that perform a sequence of similar operations. Instead of bounding the cost of the sequence of operations by bounding the actual cost of each operation separately, an amortized analysis provides a worst-case bound on the actual cost of the entire sequence. One advantage of this approach is that although some operations might be expensive, many others might be cheap. You can use amortized analysis when designing algorithms, since the design of an algorithm and the analysis of its running time are often closely intertwined. Chapter 16 introduces three ways to perform an amortized analysis of an algorithm.

14 Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. ("Programming" in this context refers to a tabular method, not to writing computer code.) As we saw in Chapters 2 and 4, divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

Dynamic programming typically applies to ***optimization problems***. Such problems can have many possible solutions. Each solution has a value, and you want to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

To develop a dynamic-programming algorithm, follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1–3 form the basis of a dynamic-programming solution to a problem. If you need only the value of an optimal solution, and not the solution itself, then you can omit step 4. When you do perform step 4, it often pays to maintain additional information during step 3 so that you can easily construct an optimal solution.

The sections that follow use the dynamic-programming method to solve some optimization problems. Section 14.1 examines the problem of cutting a rod into

rods of smaller length in a way that maximizes their total value. Section 14.2 shows how to multiply a chain of matrices while performing the fewest total scalar multiplications. Given these examples of dynamic programming, Section 14.3 discusses two key characteristics that a problem must have for dynamic programming to be a viable solution technique. Section 14.4 then shows how to find the longest common subsequence of two sequences via dynamic programming. Finally, Section 14.5 uses dynamic programming to construct binary search trees that are optimal, given a known distribution of keys to be looked up.

14.1 Rod cutting

Our first example uses dynamic programming to solve a simple problem in deciding where to cut steel rods. Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free. The management of Serling Enterprises wants to know the best way to cut up the rods.

Serling Enterprises has a table giving, for $i = 1, 2, \ldots$, the price p_i in dollars that they charge for a rod of length i inches. The length of each rod in inches is always an integer. Figure 14.1 gives a sample price table.

The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. If the price p_n for a rod of length n is large enough, an optimal solution might require no cutting at all.

Consider the case when $n = 4$. Figure 14.2 shows all the ways to cut up a rod of 4 inches in length, including the way with no cuts at all. Cutting a 4-inch rod into two 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.

Serling Enterprises can cut up a rod of length n in 2^{n-1} different ways, since they have an independent option of cutting, or not cutting, at distance i inches from the left end, for $i = 1, 2, \ldots, n-1$.[1] We denote a decomposition into pieces using ordinary additive notation, so that $7 = 2 + 2 + 3$ indicates that a rod of length 7 is cut into three pieces—two of length 2 and one of length 3. If an optimal solution cuts the rod into k pieces, for some $1 \le k \le n$, then an optimal decomposition

$$n = i_1 + i_2 + \cdots + i_k$$

[1] If pieces are required to be cut in order of monotonically increasing size, there are fewer ways to consider. For $n = 4$, only 5 such ways are possible: parts (a), (b), (c), (e), and (h) in Figure 14.2. The number of ways is called the ***partition function***, which is approximately equal to $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$. This quantity is less than 2^{n-1}, but still much greater than any polynomial in n. We won't pursue this line of inquiry further, however.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Figure 14.1 A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.

Figure 14.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 14.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

of the rod into pieces of lengths $i_1, i_2, \ldots, i_k$ provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k} .$$

For the sample problem in Figure 14.1, you can determine the optimal revenue figures r_i, for $i = 1, 2, \ldots, 10$, by inspection, with the corresponding optimal decompositions

$$\begin{aligned}
r_1 &= 1 \quad \text{from solution } 1 = 1 \quad \text{(no cuts)} ,\\
r_2 &= 5 \quad \text{from solution } 2 = 2 \quad \text{(no cuts)} ,\\
r_3 &= 8 \quad \text{from solution } 3 = 3 \quad \text{(no cuts)} ,\\
r_4 &= 10 \quad \text{from solution } 4 = 2 + 2 ,\\
r_5 &= 13 \quad \text{from solution } 5 = 2 + 3 ,\\
r_6 &= 17 \quad \text{from solution } 6 = 6 \quad \text{(no cuts)} ,\\
r_7 &= 18 \quad \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3 ,\\
r_8 &= 22 \quad \text{from solution } 8 = 2 + 6 ,\\
r_9 &= 25 \quad \text{from solution } 9 = 3 + 6 ,\\
r_{10} &= 30 \quad \text{from solution } 10 = 10 \quad \text{(no cuts)} .
\end{aligned}$$

More generally, we can express the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\} \ . \tag{14.1}$$

The first argument, p_n, corresponds to making no cuts at all and selling the rod of length n as is. The other $n-1$ arguments to max correspond to the maximum revenue obtained by making an initial cut of the rod into two pieces of size i and $n-i$, for each $i = 1, 2, \ldots, n-1$, and then optimally cutting up those pieces further, obtaining revenues r_i and r_{n-i} from those two pieces. Since you don't know ahead of time which value of i optimizes revenue, you have to consider all possible values for i and pick the one that maximizes revenue. You also have the option of picking no i at all if the greatest revenue comes from selling the rod uncut.

To solve the original problem of size n, you solve smaller problems of the same type. Once you make the first cut, the two resulting pieces form independent instances of the rod-cutting problem. The overall optimal solution incorporates optimal solutions to the two resulting subproblems, maximizing revenue from each of those two pieces. We say that the rod-cutting problem exhibits ***optimal substructure***: optimal solutions to a problem incorporate optimal solutions to related subproblems, which you may solve independently.

In a related, but slightly simpler, way to arrange a recursive structure for the rod-cutting problem, let's view a decomposition as consisting of a first piece of length i cut off the left-hand end, and then a right-hand remainder of length $n-i$. Only the remainder, and not the first piece, may be further divided. Think of every decomposition of a length-n rod in this way: as a first piece followed by some decomposition of the remainder. Then we can express the solution with no cuts at all by saying that the first piece has size $i = n$ and revenue p_n and that the remainder has size 0 with corresponding revenue $r_0 = 0$. We thus obtain the following simpler version of equation (14.1):

$$r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\} \ . \tag{14.2}$$

In this formulation, an optimal solution embodies the solution to only *one* related subproblem—the remainder—rather than two.

Recursive top-down implementation

The CUT-ROD procedure on the following page implements the computation implicit in equation (14.2) in a straightforward, top-down, recursive manner. It takes as input an array $p[1:n]$ of prices and an integer n, and it returns the maximum revenue possible for a rod of length n. For length $n = 0$, no revenue is possible, and so CUT-ROD returns 0 in line 2. Line 3 initializes the maximum revenue q to $-\infty$, so that the **for** loop in lines 4–5 correctly computes

$q = \max\{p_i + \text{CUT-ROD}(p, n-i) : 1 \le i \le n\}$. Line 6 then returns this value. A simple induction on n proves that this answer is equal to the desired answer r_n, using equation (14.2).

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max{q, p[i] + CUT-ROD(p, n - i)}
6  return q
```

If you code up CUT-ROD in your favorite programming language and run it on your computer, you'll find that once the input size becomes moderately large, your program takes a long time to run. For $n = 40$, your program may take several minutes and possibly more than an hour. For large values of n, you'll also discover that each time you increase n by 1, your program's running time approximately doubles.

Why is CUT-ROD so inefficient? The problem is that CUT-ROD calls itself recursively over and over again with the same parameter values, which means that it solves the same subproblems repeatedly. Figure 14.3 shows a recursion tree demonstrating what happens for $n = 4$: CUT-ROD(p, n) calls CUT-ROD$(p, n-i)$ for $i = 1, 2, \ldots, n$. Equivalently, CUT-ROD(p, n) calls CUT-ROD(p, j) for each $j = 0, 1, \ldots, n-1$. When this process unfolds recursively, the amount of work done, as a function of n, grows explosively.

To analyze the running time of CUT-ROD, let $T(n)$ denote the total number of calls made to CUT-ROD(p, n) for a particular value of n. This expression equals the number of nodes in a subtree whose root is labeled n in the recursion tree. The count includes the initial call at its root. Thus, $T(0) = 1$ and

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) . \tag{14.3}$$

The initial 1 is for the call at the root, and the term $T(j)$ counts the number of calls (including recursive calls) due to the call CUT-ROD$(p, n-i)$, where $j = n - i$. As Exercise 14.1-1 asks you to show,

$$T(n) = 2^n , \tag{14.4}$$

and so the running time of CUT-ROD is exponential in n.

In retrospect, this exponential running time is not so surprising. CUT-ROD explicitly considers all possible ways of cutting up a rod of length n. How many ways

Figure 14.3 The recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for $n = 4$. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and leaving a remaining subproblem of size t. A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n. In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

are there? A rod of length n has $n - 1$ potential locations to cut. Each possible way to cut up the rod makes a cut at some subset of these $n - 1$ locations, including the empty set, which makes for no cuts. Viewing each cut location as a distinct member of a set of $n - 1$ elements, you can see that there are 2^{n-1} subsets. Each leaf in the recursion tree of Figure 14.3 corresponds to one possible way to cut up the rod. Hence, the recursion tree has 2^{n-1} leaves. The labels on the simple path from the root to a leaf give the sizes of each remaining right-hand piece before making each cut. That is, the labels give the corresponding cut points, measured from the right-hand end of the rod.

Using dynamic programming for optimal rod cutting

Now, let's see how to use dynamic programming to convert CUT-ROD into an efficient algorithm.

The dynamic-programming method works as follows. Instead of solving the same subproblems repeatedly, as in the naive recursion solution, arrange for each subproblem to be solved *only once*. There's actually an obvious way to do so: the first time you solve a subproblem, *save its solution*. If you need to refer to this subproblem's solution again later, just look it up, rather than recomputing it.

Saving subproblem solutions comes with a cost: the additional memory needed to store solutions. Dynamic programming thus serves as an example of a ***time-memory trade-off***. The savings may be dramatic. For example, we're about to use dynamic programming to go from the exponential-time algorithm for rod cutting

down to a $\Theta(n^2)$-time algorithm. A dynamic-programming approach runs in polynomial time when the number of *distinct* subproblems involved is polynomial in the input size and you can solve each such subproblem in polynomial time.

There are usually two equivalent ways to implement a dynamic-programming approach. Solutions to the rod-cutting problem illustrate both of them.

The first approach is ***top-down*** with ***memoization***.[2] In this approach, you write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table). The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level. If not, the procedure computes the value in the usual manner but also saves it. We say that the recursive procedure has been ***memoized***: it "remembers" what results it has computed previously.

The second approach is the ***bottom-up method***. This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems. Solve the subproblems in size order, smallest first, storing the solution to each subproblem when it is first solved. In this way, when solving a particular subproblem, there are already saved solutions for all of the smaller subproblems its solution depends upon. You need to solve each subproblem only once, and when you first see it, you have already solved all of its prerequisite subproblems.

These two approaches yield algorithms with the same asymptotic running time, except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems. The bottom-up approach often has much better constant factors, since it has lower overhead for procedure calls.

The procedures MEMOIZED-CUT-ROD and MEMOIZED-CUT-ROD-AUX on the facing page demonstrate how to memoize the top-down CUT-ROD procedure. The main procedure MEMOIZED-CUT-ROD initializes a new auxiliary array $r[0:n]$ with the value $-\infty$ which, since known revenue values are always nonnegative, is a convenient choice for denoting "unknown." MEMOIZED-CUT-ROD then calls its helper procedure, MEMOIZED-CUT-ROD-AUX, which is just the memoized version of the exponential-time procedure, CUT-ROD. It first checks in line 1 to see whether the desired value is already known and, if it is, then line 2 returns it. Otherwise, lines 3–7 compute the desired value q in the usual manner, line 8 saves it in $r[n]$, and line 9 returns it.

The bottom-up version, BOTTOM-UP-CUT-ROD on the next page, is even simpler. Using the bottom-up dynamic-programming approach, BOTTOM-UP-CUT-ROD takes advantage of the natural ordering of the subproblems: a subproblem of

---

[2] The technical term "memoization" is not a misspelling of "memorization." The word "memoization" comes from "memo," since the technique consists of recording a value to be looked up later.

```
MEMOIZED-CUT-ROD(p, n)
let r[0:n] be a new array          // will remember solution values in r
for i = 0 to n
    r[i] = -∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0                        // already have a solution for length n?
    return r[n]
if n == 0
    q = 0
else q = -∞
    for i = 1 to n                 // i is the position of the first cut
        q = max {q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r)}
r[n] = q                           // remember the solution value for length n
return q

BOTTOM-UP-CUT-ROD(p, n)
1 let r[0:n] be a new array        // will remember solution values in r
2 r[0] = 0
3 for j = 1 to n                   // for increasing rod length j
4     q = -∞
5     for i = 1 to j               // i is the position of the first cut
6         q = max {q, p[i] + r[j - i]}
7     r[j] = q                     // remember the solution value for length j
8 return r[n]
```

size i is "smaller" than a subproblem of size j if $i < j$. Thus, the procedure solves subproblems of sizes $j = 0, 1, \ldots, n$, in that order.

Line 1 of BOTTOM-UP-CUT-ROD creates a new array $r[0:n]$ in which to save the results of the subproblems, and line 2 initializes $r[0]$ to 0, since a rod of length 0 earns no revenue. Lines 3–6 solve each subproblem of size j, for $j = 1, 2, \ldots, n$, in order of increasing size. The approach used to solve a problem of a particular size j is the same as that used by CUT-ROD, except that line 6 now directly references array entry $r[j-i]$ instead of making a recursive call to solve the subproblem of size $j-i$. Line 7 saves in $r[j]$ the solution to the subproblem of size j. Finally, line 8 returns $r[n]$, which equals the optimal value r_n.

The bottom-up and top-down versions have the same asymptotic running time. The running time of BOTTOM-UP-CUT-ROD is $\Theta(n^2)$, due to its doubly nested

Figure 14.4 The subproblem graph for the rod-cutting problem with $n = 4$. The vertex labels give the sizes of the corresponding subproblems. A directed edge (x, y) indicates that solving subproblem x requires a solution to subproblem y. This graph is a reduced version of the recursion tree of Figure 14.3, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

loop structure. The number of iterations of its inner **for** loop, in lines 5–6, forms an arithmetic series. The running time of its top-down counterpart, MEMOIZED-CUT-ROD, is also $\Theta(n^2)$, although this running time may be a little harder to see. Because a recursive call to solve a previously solved subproblem returns immediately, MEMOIZED-CUT-ROD solves each subproblem just once. It solves subproblems for sizes $0, 1, \ldots, n$. To solve a subproblem of size n, the **for** loop of lines 6–7 iterates n times. Thus, the total number of iterations of this **for** loop, over all recursive calls of MEMOIZED-CUT-ROD, forms an arithmetic series, giving a total of $\Theta(n^2)$ iterations, just like the inner **for** loop of BOTTOM-UP-CUT-ROD. (We actually are using a form of aggregate analysis here. We'll see aggregate analysis in detail in Section 16.1.)

Subproblem graphs

When you think about a dynamic-programming problem, you need to understand the set of subproblems involved and how subproblems depend on one another.

The ***subproblem graph*** for the problem embodies exactly this information. Figure 14.4 shows the subproblem graph for the rod-cutting problem with $n = 4$. It is a directed graph, containing one vertex for each distinct subproblem. The subproblem graph has a directed edge from the vertex for subproblem x to the vertex for subproblem y if determining an optimal solution for subproblem x involves directly considering an optimal solution for subproblem y. For example, the subproblem graph contains an edge from x to y if a top-down recursive procedure for solving x directly calls itself to solve y. You can think of the subproblem graph as

a "reduced" or "collapsed" version of the recursion tree for the top-down recursive method, with all nodes for the same subproblem coalesced into a single vertex and all edges directed from parent to child.

The bottom-up method for dynamic programming considers the vertices of the subproblem graph in such an order that you solve the subproblems y adjacent to a given subproblem x before you solve subproblem x. (As Section B.4 notes, the adjacency relation in a directed graph is not necessarily symmetric.) Using terminology that we'll see in Section 20.4, in a bottom-up dynamic-programming algorithm, you consider the vertices of the subproblem graph in an order that is a "reverse topological sort," or a "topological sort of the transpose" of the subproblem graph. In other words, no subproblem is considered until all of the subproblems it depends upon have been solved. Similarly, using notions that we'll visit in Section 20.3, you can view the top-down method (with memoization) for dynamic programming as a "depth-first search" of the subproblem graph.

The size of the subproblem graph $G = (V, E)$ can help you determine the running time of the dynamic-programming algorithm. Since you solve each subproblem just once, the running time is the sum of the times needed to solve each subproblem. Typically, the time to compute the solution to a subproblem is proportional to the degree (number of outgoing edges) of the corresponding vertex in the subproblem graph, and the number of subproblems is equal to the number of vertices in the subproblem graph. In this common case, the running time of dynamic programming is linear in the number of vertices and edges.

Reconstructing a solution

The procedures MEMOIZED-CUT-ROD and BOTTOM-UP-CUT-ROD return the *value* of an optimal solution to the rod-cutting problem, but they do not return the solution *itself*: a list of piece sizes.

Let's see how to extend the dynamic-programming approach to record not only the optimal *value* computed for each subproblem, but also a *choice* that led to the optimal value. With this information, you can readily print an optimal solution. The procedure EXTENDED-BOTTOM-UP-CUT-ROD on the next page computes, for each rod size j, not only the maximum revenue r_j, but also s_j, the optimal size of the first piece to cut off. It's similar to BOTTOM-UP-CUT-ROD, except that it creates the array s in line 1, and it updates $s[j]$ in line 8 to hold the optimal size i of the first piece to cut off when solving a subproblem of size j.

The procedure PRINT-CUT-ROD-SOLUTION on the following page takes as input an array $p[1:n]$ of prices and a rod size n. It calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the array $s[1:n]$ of optimal first-piece sizes. Then it prints out the complete list of piece sizes in an optimal decomposition of a

rod of length n. For the sample price chart appearing in Figure 14.1, the call EXTENDED-BOTTOM-UP-CUT-ROD$(p, 10)$ returns the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$		1	2	3	2	2	6	1	2	3	10

A call to PRINT-CUT-ROD-SOLUTION$(p, 10)$ prints just 10, but a call with $n = 7$ prints the cuts 1 and 6, which correspond to the first optimal decomposition for r_7 given earlier.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0:n] and s[1:n] be new arrays
r[0] = 0
for j = 1 to n                  // for increasing rod length j
    q = -∞
    for i = 1 to j              // i is the position of the first cut
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i            // best cut location so far for length j
    r[j] = q                    // remember the solution value for length j
return r and s

PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]          // cut location for length n
    n = n - s[n]        // length of the remainder of the rod
```

Exercises

14.1-1

Show that equation (14.4) follows from equation (14.3) and the initial condition $T(0) = 1$.

14.1-2

Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the ***density*** of a rod of length i to be p_i / i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \le i \le n$, having maximum

density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

14.1-3
Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

14.1-4
Modify CUT-ROD and MEMOIZED-CUT-ROD-AUX so that their **for** loops go up to only $\lfloor n/2 \rfloor$, rather than up to n. What other changes to the procedures do you need to make? How are their running times affected?

14.1-5
Modify MEMOIZED-CUT-ROD to return not only the value but the actual solution.

14.1-6
The Fibonacci numbers are defined by recurrence (3.31) on page 69. Give an $O(n)$-time dynamic-programming algorithm to compute the nth Fibonacci number. Draw the subproblem graph. How many vertices and edges does the graph contain?

14.2 Matrix-chain multiplication

Our next example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication. Given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, where the matrices aren't necessarily square, the goal is to compute the product

$$A_1 A_2 \cdots A_n \; . \tag{14.5}$$

using the standard algorithm[3] for multiplying rectangular matrices, which we'll see in a moment, while minimizing the number of scalar multiplications.

You can evaluate the expression (14.5) using the algorithm for multiplying pairs of rectangular matrices as a subroutine once you have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. A product of

[3] None of the three methods from Sections 4.1 and Section 4.2 can be used directly, because they apply only to square matrices.

matrices is ***fully parenthesized*** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, then you can fully parenthesize the product $A_1 A_2 A_3 A_4$ in five distinct ways:

$$
\begin{aligned}
&(A_1(A_2(A_3 A_4))) \,, \\
&(A_1((A_2 A_3) A_4)) \,, \\
&((A_1 A_2)(A_3 A_4)) \,, \\
&((A_1(A_2 A_3)) A_4) \,, \\
&(((A_1 A_2) A_3) A_4) \,.
\end{aligned}
$$

How you parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two rectangular matrices. The standard algorithm is given by the procedure RECTANGULAR-MATRIX-MULTIPLY, which generalizes the square-matrix multiplication procedure MATRIX-MULTIPLY on page 81. The RECTANGULAR-MATRIX-MULTIPLY procedure computes $C = C + A \cdot B$ for three matrices $A = (a_{ij})$, $B = (b_{ij})$, and $C = (c_{ij})$, where A is $p \times q$, B is $q \times r$, and C is $p \times r$.

```
RECTANGULAR-MATRIX-MULTIPLY(A, B, C, p, q, r)
1  for i = 1 to p
2      for j = 1 to r
3          for k = 1 to q
4              c_ij = c_ij + a_ik · b_kj
```

The running time of RECTANGULAR-MATRIX-MULTIPLY is dominated by the number of scalar multiplications in line 4, which is pqr. Therefore, we'll consider the cost of multiplying matrices to be the number of scalar multiplications. (The number of scalar multiplications dominates even if we consider initializing $C = 0$ to perform just $C = A \cdot B$.)

To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices. Suppose that the dimensions of the matrices are 10×100, 100×5, and 5×50, respectively. Multiplying according to the parenthesization $((A_1 A_2) A_3)$ performs $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute the 10×5 matrix product $A_1 A_2$, plus another $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by A_3, for a total of 7500 scalar multiplications. Multiplying according to the alternative parenthesization $(A_1(A_2 A_3))$ performs $100 \cdot 5 \cdot 50 = 25{,}000$ scalar multiplications to compute the 100×50 matrix product $A_2 A_3$, plus another $10 \cdot 100 \cdot 50 = 50{,}000$ scalar multiplications to multiply A_1 by this matrix, for a

total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

We state the ***matrix-chain multiplication problem*** as follows: given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications. The input is the sequence of dimensions $\langle p_0, p_1, p_2, \ldots, p_n \rangle$.

The matrix-chain multiplication problem does not entail actually multiplying matrices. The goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications (such as performing only 7500 scalar multiplications instead of 75,000).

Counting the number of parenthesizations

Before solving the matrix-chain multiplication problem by dynamic programming, let us convince ourselves that exhaustively checking all possible parenthesizations is not an efficient algorithm. Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$. When $n = 1$, the sequence consists of just one matrix, and therefore there is only one way to fully parenthesize the matrix product. When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the kth and $(k+1)$st matrices for any $k = 1, 2, \ldots, n-1$. Thus, we obtain the recurrence

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 . \end{cases} \tag{14.6}$$

Problem 12-4 on page 329 asked you to show that the solution to a similar recurrence is the sequence of ***Catalan numbers***, which grows as $\Omega(4^n / n^{3/2})$. A simpler exercise (see Exercise 14.2-3) is to show that the solution to the recurrence (14.6) is $\Omega(2^n)$. The number of solutions is thus exponential in n, and the brute-force method of exhaustive search makes for a poor strategy when determining how to optimally parenthesize a matrix chain.

Applying dynamic programming

Let's use the dynamic-programming method to determine how to optimally parenthesize a matrix chain, by following the four-step sequence that we stated at the beginning of this chapter:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

We'll go through these steps in order, demonstrating how to apply each step to the problem.

Step 1: The structure of an optimal parenthesization

In the first step of the dynamic-programming method, you find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems. To perform this step for the matrix-chain multiplication problem, it's convenient to first introduce some notation. Let $A_{i:j}$, where $i \le j$, denote the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$. If the problem is nontrivial, that is, $i < j$, then to parenthesize the product $A_i A_{i+1} \cdots A_j$, the product must split between A_k and A_{k+1} for some integer k in the range $i \le k < j$. That is, for some value of k, first compute the matrices $A_{i:k}$ and $A_{k+1:j}$, and then multiply them together to produce the final product $A_{i:j}$. The cost of parenthesizing this way is the cost of computing the matrix $A_{i:k}$, plus the cost of computing $A_{k+1:j}$, plus the cost of multiplying them together.

The optimal substructure of this problem is as follows. Suppose that to optimally parenthesize $A_i A_{i+1} \cdots A_j$, you split the product between A_k and A_{k+1}. Then the way you parenthesize the "prefix" subchain $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$. Why? If there were a less costly way to parenthesize $A_i A_{i+1} \cdots A_k$, then you could substitute that parenthesization in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ to produce another way to parenthesize $A_i A_{i+1} \cdots A_j$ whose cost is lower than the optimum: a contradiction. A similar observation holds for how to parenthesize the subchain $A_{k+1} A_{k+2} \cdots A_j$ in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$: it must be an optimal parenthesization of $A_{k+1} A_{k+2} \cdots A_j$.

Now let's use the optimal substructure to show how to construct an optimal solution to the problem from optimal solutions to subproblems. Any solution to a nontrivial instance of the matrix-chain multiplication problem requires splitting the product, and any optimal solution contains within it optimal solutions to subproblem instances. Thus, to build an optimal solution to an instance of the matrix-chain multiplication problem, split the problem into two subproblems (optimally parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$), find optimal solutions to the two subproblem instances, and then combine these optimal subproblem solutions. To ensure that you've examined the optimal split, you must consider all possible splits.

Step 2: A recursive solution

The next step is to define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the matrix-chain multiplication problem, a subproblem is to determine the minimum cost of parenthesizing $A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$. Given the input dimensions $\langle p_0, p_1, p_2, \ldots, p_n \rangle$, an index pair i, j specifies a subproblem. Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i:j}$. For the full problem, the lowest-cost way to compute $A_{1:n}$ is thus $m[1, n]$.

We can define $m[i, j]$ recursively as follows. If $i = j$, the problem is trivial: the chain consists of just one matrix $A_{i:i} = A_i$, so that no scalar multiplications are necessary to compute the product. Thus, $m[i, i] = 0$ for $i = 1, 2, \ldots, n$. To compute $m[i, j]$ when $i < j$, we take advantage of the structure of an optimal solution from step 1. Suppose that an optimal parenthesization splits the product $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}, where $i \le k < j$. Then, $m[i, j]$ equals the minimum cost $m[i, k]$ for computing the subproduct $A_{i:k}$, plus the minimum cost $m[k+1, j]$ for computing the subproduct, $A_{k+1:j}$, plus the cost of multiplying these two matrices together. Because each matrix A_i is $p_{i-1} \times p_i$, computing the matrix product $A_{i:k} A_{k+1:j}$ takes $p_{i-1} p_k p_j$ scalar multiplications. Thus, we obtain

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j \ .$$

This recursive equation assumes that you know the value of k. But you don't, at least not yet. You have to try all possible values of k. How many are there? Just $j - i$, namely $k = i, i + 1, \ldots, j - 1$. Since the optimal parenthesization must use one of these values for k, you need only check them all to find the best. Thus, the recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min \{ m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j : i \le k < j \} & \text{if } i < j \ . \end{cases} \tag{14.7}$$

The $m[i, j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information you need to construct an optimal solution. To help you do so, let's define $s[i, j]$ to be a value of k at which you split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization. That is, $s[i, j]$ equals a value k such that $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

Step 3: Computing the optimal costs

At this point, you could write a recursive algorithm based on recurrence (14.7) to compute the minimum cost $m[1, n]$ for multiplying $A_1 A_2 \cdots A_n$. But as we saw

for the rod-cutting problem, and as we shall see in Section 14.3, this recursive algorithm takes exponential time. That's no better than the brute-force method of checking each way of parenthesizing the product.

Fortunately, there aren't all that many distinct subproblems: just one subproblem for each choice of i and j satisfying $1 \le i \le j \le n$, or $\binom{n}{2} + n = \Theta(n^2)$ in all.[4] A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree. This property of overlapping subproblems is the second hallmark of when dynamic programming applies (the first hallmark being optimal substructure).

Instead of computing the solution to recurrence (14.7) recursively, let's compute the optimal cost by using a tabular, bottom-up approach, as in the procedure MATRIX-CHAIN-ORDER. (The corresponding top-down approach using memoization appears in Section 14.3.) The input is a sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$ of matrix dimensions, along with n, so that for $i = 1, 2, \ldots, n$, matrix A_i has dimensions $p_{i-1} \times p_i$. The procedure uses an auxiliary table $m[1:n, 1:n]$ to store the $m[i, j]$ costs and another auxiliary table $s[1:n-1, 2:n]$ that records which index k achieved the optimal cost in computing $m[i, j]$. The table s will help in constructing an optimal solution.

```
MATRIX-CHAIN-ORDER(p, n)
let m[1:n, 1:n] and s[1:n − 1, 2:n] be new tables
for i = 1 to n                          // chain length 1
    m[i, i] = 0
for l = 2 to n                          // l is the chain length
    for i = 1 to n − l + 1              // chain begins at A_i
        j = i + l − 1                   // chain ends at A_j
        m[i, j] = ∞
        for k = i to j − 1              // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q             // remember this cost
                s[i, j] = k             // remember this index
return m and s
```

In what order should the algorithm fill in the table entries? To answer this question, let's see which entries of the table need to be accessed when computing the

[4] The $\binom{n}{2}$ term counts all pairs in which $i < j$. Because i and j may be equal, we need to add in the n term.

cost $m[i, j]$. Equation (14.7) tells us that to compute the cost of matrix product $A_{i:j}$, first the costs of the products $A_{i:k}$ and $A_{k+1:j}$ need to have been computed for all $k = i, i+1, \ldots, j-1$. The chain $A_i A_{i+1} \cdots A_j$ consists of $j - i + 1$ matrices, and the chains $A_i A_{i+1} \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_j$ consist of $k - i + 1$ and $j - k$ matrices, respectively. Since $k < j$, a chain of $k - i + 1$ matrices consists of fewer than $j - i + 1$ matrices. Likewise, since $k \geq i$, a chain of $j - k$ matrices consists of fewer than $j - i + 1$ matrices. Thus, the algorithm should fill in the table m from shorter matrix chains to longer matrix chains. That is, for the subproblem of optimally parenthesizing the chain $A_i A_{i+1} \cdots A_j$, it makes sense to consider the subproblem size as the length $j - i + 1$ of the chain.

Now, let's see how the MATRIX-CHAIN-ORDER procedure fills in the $m[i, j]$ entries in order of increasing chain length. Lines 2–3 initialize $m[i, i] = 0$ for $i = 1, 2, \ldots, n$, since any matrix chain with just one matrix requires no scalar multiplications. In the **for** loop of lines 4–12, the loop variable l denotes the length of matrix chains whose minimum costs are being computed. Each iteration of this loop uses recurrence (14.7) to compute $m[i, i+l-1]$ for $i = 1, 2, \ldots, n-l+1$. In the first iteration, $l = 2$, and so the loop computes $m[i, i+1]$ for $i = 1, 2, \ldots, n-1$: the minimum costs for chains of length $l = 2$. The second time through the loop, it computes $m[i, i+2]$ for $i = 1, 2, \ldots, n-2$: the minimum costs for chains of length $l = 3$. And so on, ending with a single matrix chain of length $l = n$ and computing $m[1, n]$. When lines 7–12 compute an $m[i, j]$ cost, this cost depends only on table entries $m[i, k]$ and $m[k+1, j]$, which have already been computed.

Figure 14.5 illustrates the m and s tables, as filled in by the MATRIX-CHAIN-ORDER procedure on a chain of $n = 6$ matrices. Since $m[i, j]$ is defined only for $i \leq j$, only the portion of the table m on or above the main diagonal is used. The figure shows the table rotated to make the main diagonal run horizontally. The matrix chain is listed along the bottom. Using this layout, the minimum cost $m[i, j]$ for multiplying a subchain $A_i A_{i+1} \cdots A_j$ of matrices appears at the intersection of lines running northeast from A_i and northwest from A_j. Reading across, each diagonal in the table contains the entries for matrix chains of the same length. MATRIX-CHAIN-ORDER computes the rows from bottom to top and from left to right within each row. It computes each entry $m[i, j]$ using the products $p_{i-1} p_k p_j$ for $k = i, i+1, \ldots, j-1$ and all entries southwest and southeast from $m[i, j]$.

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm. The loops are nested three deep, and each loop index (l, i, and k) takes on at most $n-1$ values. Exercise 14.2-5 asks you to show that the running time of this algorithm is in fact also $\Omega(n^3)$. The algorithm requires $\Theta(n^2)$ space to store the m and s tables. Thus, MATRIX-CHAIN-ORDER is much more efficient than the exponential-time method of enumerating all possible parenthesizations and checking each one.

Figure 14.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the entries that are not tan, the pairs that have the same color are taken together in line 9 when computing

$$\begin{aligned} m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 &= 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 &= 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 &= 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11{,}375 \end{cases} \\ &= 7125\,. \end{aligned}$$

Step 4: Constructing an optimal solution

Although MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices. The table $s[1 : n-1, 2 : n]$ provides the information needed to do so. Each entry $s[i, j]$ records a value of k such that an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}. The final matrix multiplication in computing $A_{1:n}$ optimally is $A_{1:s[1,n]} A_{s[1,n]+1:n}$. The s table contains the information needed to determine the earlier matrix multiplications as well, using recursion: $s[1, s[1, n]]$ determines the last matrix multiplication when computing $A_{1:s[1,n]}$ and $s[s[1, n] + 1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1:n}$. The recursive procedure PRINT-OPTIMAL-PARENS on the facing page prints an optimal parenthesization of the matrix chain product $A_i A_{i+1} \cdots A_j$, given the s table computed by MATRIX-CHAIN-ORDER and the in-

dices i and j. The initial call PRINT-OPTIMAL-PARENS$(s, 1, n)$ prints an optimal parenthesization of the full matrix chain product $A_1 A_2 \cdots A_n$. In the example of Figure 14.5, the call PRINT-OPTIMAL-PARENS$(s, 1, 6)$ prints the optimal parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

Exercises

14.2-1

Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

14.2-2

Give a recursive algorithm MATRIX-CHAIN-MULTIPLY(A, s, i, j) that actually performs the optimal matrix-chain multiplication, given the sequence of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j. (The initial call is MATRIX-CHAIN-MULTIPLY$(A, s, 1, n)$.) Assume that the call RECTANGULAR-MATRIX-MULTIPLY(A, B) returns the product of matrices A and B.

14.2-3

Use the substitution method to show that the solution to the recurrence (14.6) is $\Omega(2^n)$.

14.2-4

Describe the subproblem graph for matrix-chain multiplication with an input chain of length n. How many vertices does it have? How many edges does it have, and which edges are they?

14.2-5

Let $R(i, j)$ be the number of times that table entry $m[i, j]$ is referenced while computing other table entries in a call of MATRIX-CHAIN-ORDER. Show that the total number of references for the entire table is

$$\sum_{i=1}^{n} \sum_{j=i}^{n} R(i, j) = \frac{n^3 - n}{3} .$$

(*Hint:* You may find equation (A.4) on page 1141 useful.)

14.2-6
Show that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses.

14.3 Elements of dynamic programming

Although you have just seen two complete examples of the dynamic-programming method, you might still be wondering just when the method applies. From an engineering perspective, when should you look for a dynamic-programming solution to a problem? In this section, we'll examine the two key ingredients that an optimization problem must have in order for dynamic programming to apply: optimal substructure and overlapping subproblems. We'll also revisit and discuss more fully how memoization might help you take advantage of the overlapping-subproblems property in a top-down recursive approach.

Optimal substructure

The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution. Recall that a problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems. When a problem exhibits optimal substructure, that gives you a good clue that dynamic programming might apply. (As Chapter 15 discusses, it also might mean that a greedy strategy applies, however.) Dynamic programming builds an optimal solution to the problem from optimal solutions to subproblems. Consequently, you must take care to ensure that the range of subproblems you consider includes those used in an optimal solution.

Optimal substructure was key to solving both of the previous problems in this chapter. In Section 14.1, we observed that the optimal way of cutting up a rod of length n (if Serling Enterprises makes any cuts at all) involves optimally cutting up the two pieces resulting from the first cut. In Section 14.2, we noted that an optimal parenthesization of the matrix chain product $A_i A_{i+1} \cdots A_j$ that splits the product between A_k and A_{k+1} contains within it optimal solutions to the problems of parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$.

You will find yourself following a common pattern in discovering optimal substructure:

1. You show that a solution to the problem consists of making a choice, such as choosing an initial cut in a rod or choosing an index at which to split the matrix chain. Making this choice leaves one or more subproblems to be solved.
2. You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.
3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
4. You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a "cut-and-paste" technique. You do so by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction. In particular, by "cutting out" the nonoptimal solution to each subproblem and "pasting in" the optimal one, you show that you can get a better solution to the original problem, thus contradicting your supposition that you already had an optimal solution. If an optimal solution gives rise to more than one subproblem, they are typically so similar that you can modify the cut-and-paste argument for one to apply to the others with little effort.

To characterize the space of subproblems, a good rule of thumb says to try to keep the space as simple as possible and then expand it as necessary. For example, the space of subproblems for the rod-cutting problem contained the problems of optimally cutting up a rod of length i for each size i. This subproblem space worked well, and it was not necessary to try a more general space of subproblems.

Conversely, suppose that you tried to constrain the subproblem space for matrix-chain multiplication to matrix products of the form $A_1 A_2 \cdots A_j$. As before, an optimal parenthesization must split this product between A_k and A_{k+1} for some $1 \le k < j$. Unless you can guarantee that k always equals $j-1$, you will find that you have subproblems of the form $A_1 A_2 \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$. Moreover, the latter subproblem does not have the form $A_1 A_2 \cdots A_j$. To solve this problem by dynamic programming, you need to allow the subproblems to vary at "both ends." That is, both i and j need to vary in the subproblem of parenthesizing the product $A_i A_{i+1} \cdots A_j$.

Optimal substructure varies across problem domains in two ways:

1. how many subproblems an optimal solution to the original problem uses, and
2. how many choices you have in determining which subproblem(s) to use in an optimal solution.

In the rod-cutting problem, an optimal solution for cutting up a rod of size n uses just one subproblem (of size $n-i$), but we have to consider n choices for i in order to determine which one yields an optimal solution. Matrix-chain multiplication for the subchain $A_i A_{i+1} \cdots A_j$ serves an example with two subproblems and $j-i$ choices. For a given matrix A_k where the product splits, two subproblems arise—parenthesizing $A_i A_{i+1} \cdots A_k$ and parenthesizing $A_{k+1} A_{k+2} \cdots A_j$—and we have to solve *both* of them optimally. Once we determine the optimal solutions to subproblems, we choose from among $j-i$ candidates for the index k.

Informally, the running time of a dynamic-programming algorithm depends on the product of two factors: the number of subproblems overall and how many choices you look at for each subproblem. In rod cutting, we had $\Theta(n)$ subproblems overall, and at most n choices to examine for each, yielding an $O(n^2)$ running time. Matrix-chain multiplication had $\Theta(n^2)$ subproblems overall, and each had at most $n-1$ choices, giving an $O(n^3)$ running time (actually, a $\Theta(n^3)$ running time, by Exercise 14.2-5).

Usually, the subproblem graph gives an alternative way to perform the same analysis. Each vertex corresponds to a subproblem, and the choices for a subproblem are the edges incident from that subproblem. Recall that in rod cutting, the subproblem graph has n vertices and at most n edges per vertex, yielding an $O(n^2)$ running time. For matrix-chain multiplication, if you were to draw the subproblem graph, it would have $\Theta(n^2)$ vertices and each vertex would have degree at most $n-1$, giving a total of $O(n^3)$ vertices and edges.

Dynamic programming often uses optimal substructure in a bottom-up fashion. That is, you first find optimal solutions to subproblems and, having solved the subproblems, you find an optimal solution to the problem. Finding an optimal solution to the problem entails making a choice among subproblems as to which you will use in solving the problem. The cost of the problem solution is usually the subproblem costs plus a cost that is directly attributable to the choice itself. In rod cutting, for example, first we solved the subproblems of determining optimal ways to cut up rods of length i for $i = 0, 1, \ldots, n-1$, and then we determined which of these subproblems yielded an optimal solution for a rod of length n, using equation (14.2). The cost attributable to the choice itself is the term p_i in equation (14.2). In matrix-chain multiplication, we determined optimal parenthesizations of subchains of $A_i A_{i+1} \cdots A_j$, and then we chose the matrix A_k at which to split the product. The cost attributable to the choice itself is the term $p_{i-1} p_k p_j$.

Chapter 15 explores "greedy algorithms," which have many similarities to dynamic programming. In particular, problems to which greedy algorithms apply have optimal substructure. One major difference between greedy algorithms and dynamic programming is that instead of first finding optimal solutions to subproblems and then making an informed choice, greedy algorithms first make a "greedy" choice—the choice that looks best at the time—and then solve a resulting subprob-

lem, without bothering to solve all possible related smaller subproblems. Surprisingly, in some cases this strategy works!

Subtleties

You should be careful not to assume that optimal substructure applies when it does not. Consider the following two problems whose input consists of a directed graph $G = (V, E)$ and vertices $u, v \in V$.

Unweighted shortest path:[5] Find a path from u to v consisting of the fewest edges. Such a path must be simple, since removing a cycle from a path produces a path with fewer edges.

Unweighted longest simple path: Find a simple path from u to v consisting of the most edges. (Without the requirement that the path must be simple, the problem is undefined, since repeatedly traversing a cycle creates paths with an arbitrarily large number of edges.)

The unweighted shortest-path problem exhibits optimal substructure. Here's how. Suppose that $u \neq v$, so that the problem is nontrivial. Then, any path p from u to v must contain an intermediate vertex, say w. (Note that w may be u or v.) Then, we can decompose the path $u \overset{p}{\leadsto} v$ into subpaths $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$. The number of edges in p equals the number of edges in p_1 plus the number of edges in p_2. We claim that if p is an optimal (i.e., shortest) path from u to v, then p_1 must be a shortest path from u to w. Why? As suggested earlier, use a "cut-and-paste" argument: if there were another path, say p'_1, from u to w with fewer edges than p_1, then we could cut out p_1 and paste in p'_1 to produce a path $u \overset{p'_1}{\leadsto} w \overset{p_2}{\leadsto} v$ with fewer edges than p, thus contradicting p's optimality. Likewise, p_2 must be a shortest path from w to v. Thus, to find a shortest path from u to v, consider all intermediate vertices w, find a shortest path from u to w and a shortest path from w to v, and choose an intermediate vertex w that yields the overall shortest path. Section 23.2 uses a variant of this observation of optimal substructure to find a shortest path between every pair of vertices on a weighted, directed graph.

You might be tempted to assume that the problem of finding an unweighted longest simple path exhibits optimal substructure as well. After all, if we decompose a longest simple path $u \overset{p}{\leadsto} v$ into subpaths $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$, then mustn't p_1 be a longest simple path from u to w, and mustn't p_2 be a longest simple path from w to v? The answer is no! Figure 14.6 supplies an example. Consider the

[5] We use the term "unweighted" to distinguish this problem from that of finding shortest paths with weighted edges, which we shall see in Chapters 22 and 23. You can use the breadth-first search technique of Chapter 20 to solve the unweighted problem.

Figure 14.6 A directed graph showing that the problem of finding a longest simple path in an unweighted directed graph does not have optimal substructure. The path $q \to r \to t$ is a longest simple path from q to t, but the subpath $q \to r$ is not a longest simple path from q to r, nor is the subpath $r \to t$ a longest simple path from r to t.

path $q \to r \to t$, which is a longest simple path from q to t. Is $q \to r$ a longest simple path from q to r? No, for the path $q \to s \to t \to r$ is a simple path that is longer. Is $r \to t$ a longest simple path from r to t? No again, for the path $r \to q \to s \to t$ is a simple path that is longer.

This example shows that for longest simple paths, not only does the problem lack optimal substructure, but you cannot necessarily assemble a "legal" solution to the problem from solutions to subproblems. If you combine the longest simple paths $q \to s \to t \to r$ and $r \to q \to s \to t$, you get the path $q \to s \to t \to r \to q \to s \to t$, which is not simple. Indeed, the problem of finding an unweighted longest simple path does not appear to have any sort of optimal substructure. No efficient dynamic-programming algorithm for this problem has ever been found. In fact, this problem is NP-complete, which—as we shall see in Chapter 34—means that we are unlikely to find a way to solve it in polynomial time.

Why is the substructure of a longest simple path so different from that of a shortest path? Although a solution to a problem for both longest and shortest paths uses two subproblems, the subproblems in finding the longest simple path are not ***independent***, whereas for shortest paths they are. What do we mean by subproblems being independent? We mean that the solution to one subproblem does not affect the solution to another subproblem of the same problem. For the example of Figure 14.6, we have the problem of finding a longest simple path from q to t with two subproblems: finding longest simple paths from q to r and from r to t. For the first of these subproblems, we chose the path $q \to s \to t \to r$, which used the vertices s and t. These vertices cannot appear in a solution to the second subproblem, since the combination of the two solutions to subproblems yields a path that is not simple. If vertex t cannot be in the solution to the second problem, then there is no way to solve it, since t is required to be on the path that forms the solution, and it is not the vertex where the subproblem solutions are "spliced" together (that vertex being r). Because vertices s and t appear in one subproblem solution, they cannot appear in the other subproblem solution. One of them must be in the solution to the other subproblem, however, and an optimal solution requires both.

Thus, we say that these subproblems are not independent. Looked at another way, using resources in solving one subproblem (those resources being vertices) renders them unavailable for the other subproblem.

Why, then, are the subproblems independent for finding a shortest path? The answer is that by nature, the subproblems do not share resources. We claim that if a vertex w is on a shortest path p from u to v, then we can splice together *any* shortest path $u \overset{p_1}{\rightsquigarrow} w$ and *any* shortest path $w \overset{p_2}{\rightsquigarrow} v$ to produce a shortest path from u to v. We are assured that, other than w, no vertex can appear in both paths p_1 and p_2. Why? Suppose that some vertex $x \neq w$ appears in both p_1 and p_2, so that we can decompose p_1 as $u \overset{p_{ux}}{\rightsquigarrow} x \rightsquigarrow w$ and p_2 as $w \rightsquigarrow x \overset{p_{xv}}{\rightsquigarrow} v$. By the optimal substructure of this problem, path p has as many edges as p_1 and p_2 together. Let's say that p has e edges. Now let us construct a path $p' = u \overset{p_{ux}}{\rightsquigarrow} x \overset{p_{xv}}{\rightsquigarrow} v$ from u to v. Because we have excised the paths from x to w and from w to x, each of which contains at least one edge, path p' contains at most $e - 2$ edges, which contradicts the assumption that p is a shortest path. Thus, we are assured that the subproblems for the shortest-path problem are independent.

The two problems examined in Sections 14.1 and 14.2 have independent subproblems. In matrix-chain multiplication, the subproblems are multiplying subchains $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$. These subchains are disjoint, so that no matrix could possibly be included in both of them. In rod cutting, to determine the best way to cut up a rod of length n, we looked at the best ways of cutting up rods of length i for $i = 0, 1, \ldots, n-1$. Because an optimal solution to the length-n problem includes just one of these subproblem solutions (after cutting off the first piece), independence of subproblems is not an issue.

Overlapping subproblems

The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be "small" in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems. Typically, the total number of distinct subproblems is a polynomial in the input size. When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has ***overlapping subproblems***.[6] In contrast, a problem for which a divide-and-

[6] It may seem strange that dynamic programming relies on subproblems being both independent and overlapping. Although these requirements may sound contradictory, they describe two different notions, rather than two points on the same axis. Two subproblems of the same problem are independent if they do not share resources. Two subproblems are overlapping if they are really the same subproblem that occurs as a subproblem of different problems.

Figure 14.7 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node contains the parameters i and j. The computations performed in a subtree shaded blue are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN.

conquer approach is suitable usually generates brand-new problems at each step of the recursion. Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

In Section 14.1, we briefly examined how a recursive solution to rod cutting makes exponentially many calls to find solutions of smaller subproblems. The dynamic-programming solution reduces the running time from the exponential time of the recursive algorithm down to quadratic time.

To illustrate the overlapping-subproblems property in greater detail, let's revisit the matrix-chain multiplication problem. Referring back to Figure 14.5, observe that MATRIX-CHAIN-ORDER repeatedly looks up the solution to subproblems in lower rows when solving subproblems in higher rows. For example, it references entry $m[3, 4]$ four times: during the computations of $m[2, 4]$, $m[1, 4]$, $m[3, 5]$, and $m[3, 6]$. If the algorithm were to recompute $m[3, 4]$ each time, rather than just looking it up, the running time would increase dramatically. To see how, consider the inefficient recursive procedure RECURSIVE-MATRIX-CHAIN on the facing page, which determines $m[i, j]$, the minimum number of scalar multiplications needed to compute the matrix-chain product $A_{i:j} = A_i A_{i+1} \cdots A_j$. The procedure is based directly on the recurrence (14.7). Figure 14.7 shows the recursion tree produced by the call RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node is labeled by the values of the parameters i and j. Observe that some pairs of values occur many times.

In fact, the time to compute $m[1, n]$ by this recursive procedure is at least exponential in n. To see why, let $T(n)$ denote the time taken by RECURSIVE-MATRIX-

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
             + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
             + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```

CHAIN to compute an optimal parenthesization of a chain of n matrices. Because the execution of lines 1–2 and of lines 6–7 each take at least unit time, as does the multiplication in line 5, inspection of the procedure yields the recurrence

$$T(n) \ge \begin{cases} 1 & \text{if } n = 1 , \\ 1 + \displaystyle\sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) & \text{if } n > 1 . \end{cases}$$

Noting that for $i = 1, 2, \ldots, n-1$, each term $T(i)$ appears once as $T(k)$ and once as $T(n-k)$, and collecting the $n-1$ 1s in the summation together with the 1 out front, we can rewrite the recurrence as

$$T(n) \ge 2 \sum_{i=1}^{n-1} T(i) + n . \tag{14.8}$$

Let's prove that $T(n) = \Omega(2^n)$ using the substitution method. Specifically, we'll show that $T(n) \ge 2^{n-1}$ for all $n \ge 1$. For the base case $n = 1$, the summation is empty, and we get $T(1) \ge 1 = 2^0$. Inductively, for $n \ge 2$ we have

$$\begin{aligned} T(n) &\ge 2 \sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2 \sum_{j=0}^{n-2} 2^j + n && \text{(letting } j = i - 1) \\ &= 2(2^{n-1} - 1) + n && \text{(by equation (A.6) on page 1142)} \\ &= 2^n - 2 + n \\ &\ge 2^{n-1} , \end{aligned}$$

which completes the proof. Thus, the total amount of work performed by the call RECURSIVE-MATRIX-CHAIN$(p, 1, n)$ is at least exponential in n.

Compare this top-down, recursive algorithm (without memoization) with the bottom-up dynamic-programming algorithm. The latter is more efficient because it takes advantage of the overlapping-subproblems property. Matrix-chain multiplication has only $\Theta(n^2)$ distinct subproblems, and the dynamic-programming algorithm solves each exactly once. The recursive algorithm, on the other hand, must solve each subproblem every time it reappears in the recursion tree. Whenever a recursion tree for the natural recursive solution to a problem contains the same subproblem repeatedly, and the total number of distinct subproblems is small, dynamic programming can improve efficiency, sometimes dramatically.

Reconstructing an optimal solution

As a practical matter, you'll often want to store in a separate table which choice you made in each subproblem so that you do not have to reconstruct this information from the table of costs.

For matrix-chain multiplication, the table $s[i, j]$ saves a significant amount of work when we need to reconstruct an optimal solution. Suppose that the MATRIX-CHAIN-ORDER procedure on page 378 did not maintain the $s[i, j]$ table, so that it filled in only the table $m[i, j]$ containing optimal subproblem costs. The procedure chooses from among $j - i$ possibilities when determining which subproblems to use in an optimal solution to parenthesizing $A_i A_{i+1} \cdots A_j$, and $j - i$ is not a constant. Therefore, it would take $\Theta(j - i) = \omega(1)$ time to reconstruct which subproblems it chose for a solution to a given problem. Because MATRIX-CHAIN-ORDER stores in $s[i, j]$ the index of the matrix at which it split the product $A_i A_{i+1} \cdots A_j$, the PRINT-OPTIMAL-PARENS procedure on page 381 can look up each choice in $O(1)$ time.

Memoization

As we saw for the rod-cutting problem, there is an alternative approach to dynamic programming that often offers the efficiency of the bottom-up dynamic-programming approach while maintaining a top-down strategy. The idea is to ***memoize*** the natural, but inefficient, recursive algorithm. As in the bottom-up approach, you maintain a table with subproblem solutions, but the control structure for filling in the table is more like the recursive algorithm.

A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem. Each table entry initially contains a special value to indicate that the entry has yet to be filled in. When the subproblem is first encountered as the recursive algorithm unfolds, its solution is computed and then stored in the table.

Each subsequent encounter of this subproblem simply looks up the value stored in the table and returns it.[7]

The procedure MEMOIZED-MATRIX-CHAIN is a memoized version of the procedure RECURSIVE-MATRIX-CHAIN on page 389. Note where it resembles the memoized top-down method on page 369 for the rod-cutting problem.

```
MEMOIZED-MATRIX-CHAIN(p, n)
1  let m[1:n, 1:n] be a new table
2  for i = 1 to n
3      for j = i to n
4          m[i, j] = ∞
5  return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN(m, p, i, k)
               + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

The MEMOIZED-MATRIX-CHAIN procedure, like the bottom-up MATRIX-CHAIN-ORDER procedure on page 378, maintains a table $m[1:n, 1:n]$ of computed values of $m[i, j]$, the minimum number of scalar multiplications needed to compute the matrix $A_{i:j}$. Each table entry initially contains the value ∞ to indicate that the entry has yet to be filled in. Upon calling LOOKUP-CHAIN(m, p, i, j), if line 1 finds that $m[i, j] < \infty$, then the procedure simply returns the previously computed cost $m[i, j]$ in line 2. Otherwise, the cost is computed as in RECURSIVE-MATRIX-CHAIN, stored in $m[i, j]$, and returned. Thus, LOOKUP-CHAIN(m, p, i, j) always returns the value of $m[i, j]$, but it computes it only upon the first call of LOOKUP-CHAIN with these specific values of i and j.

[7] This approach presupposes that you know the set of all possible subproblem parameters and that you have established the relationship between table positions and subproblems. Another, more general, approach is to memoize by using hashing with the subproblem parameters as keys.

Figure 14.7 illustrates how MEMOIZED-MATRIX-CHAIN saves time compared with RECURSIVE-MATRIX-CHAIN. Subtrees shaded blue represent values that are looked up rather than recomputed.

Like the bottom-up procedure MATRIX-CHAIN-ORDER, the memoized procedure MEMOIZED-MATRIX-CHAIN runs in $O(n^3)$ time. To begin with, line 4 of MEMOIZED-MATRIX-CHAIN executes $\Theta(n^2)$ times, which dominates the running time outside of the call to LOOKUP-CHAIN in line 5. We can categorize the calls of LOOKUP-CHAIN into two types:

1. calls in which $m[i, j] = \infty$, so that lines 3–9 execute, and
2. calls in which $m[i, j] < \infty$, so that LOOKUP-CHAIN simply returns in line 2.

There are $\Theta(n^2)$ calls of the first type, one per table entry. All calls of the second type are made as recursive calls by calls of the first type. Whenever a given call of LOOKUP-CHAIN makes recursive calls, it makes $O(n)$ of them. Therefore, there are $O(n^3)$ calls of the second type in all. Each call of the second type takes $O(1)$ time, and each call of the first type takes $O(n)$ time plus the time spent in its recursive calls. The total time, therefore, is $O(n^3)$. Memoization thus turns an $\Omega(2^n)$-time algorithm into an $O(n^3)$-time algorithm.

We have seen how to solve the matrix-chain multiplication problem by either a top-down, memoized dynamic-programming algorithm or a bottom-up dynamic-programming algorithm in $O(n^3)$ time. Both the bottom-up and memoized methods take advantage of the overlapping-subproblems property. There are only $\Theta(n^2)$ distinct subproblems in total, and either of these methods computes the solution to each subproblem only once. Without memoization, the natural recursive algorithm runs in exponential time, since solved subproblems are repeatedly solved.

In general practice, if all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms the corresponding top-down memoized algorithm by a constant factor, because the bottom-up algorithm has no overhead for recursion and less overhead for maintaining the table. Moreover, for some problems you can exploit the regular pattern of table accesses in the dynamic-programming algorithm to reduce time or space requirements even further. On the other hand, in certain situations, some of the subproblems in the subproblem space might not need to be solved at all. In that case, the memoized solution has the advantage of solving only those subproblems that are definitely required.

Exercises

14.3-1

Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesiz-

ing the product and computing the number of multiplications for each, or running RECURSIVE-MATRIX-CHAIN? Justify your answer.

14.3-2
Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as MERGE-SORT.

14.3-3
Consider the antithetical variant of the matrix-chain multiplication problem where the goal is to parenthesize the sequence of matrices so as to maximize, rather than minimize, the number of scalar multiplications. Does this problem exhibit optimal substructure?

14.3-4
As stated, in dynamic programming, you first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that she does not always need to solve all the subproblems in order to find an optimal solution. She suggests that she can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} \cdots A_j$ (by selecting k to minimize the quantity $p_{i-1} p_k p_j$) before solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

14.3-5
Suppose that the rod-cutting problem of Section 14.1 also had a limit l_i on the number of pieces of length i allowed to be produced, for $i = 1, 2, \ldots, n$. Show that the optimal-substructure property described in Section 14.1 no longer holds.

14.4 Longest common subsequence

Biological applications often need to compare the DNA of two (or more) different organisms. A strand of DNA consists of a string of molecules called ***bases***, where the possible bases are adenine, cytosine, guanine, and thymine. Representing each of these bases by its initial letter, we can express a strand of DNA as a string over the 4-element set $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. (See Section C.1 for the definition of a string.) For example, the DNA of one organism may be $S_1 = \mathtt{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$, and the DNA of another organism may be $S_2 = \mathtt{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$. One reason to com-

pare two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are. We can, and do, define similarity in many different ways. For example, we can say that two DNA strands are similar if one is a substring of the other. (Chapter 32 explores algorithms to solve this problem.) In our example, neither S_1 nor S_2 is a substring of the other. Alternatively, we could say that two strands are similar if the number of changes needed to turn one into the other is small. (Problem 14-5 looks at this notion.) Yet another way to measure the similarity of strands S_1 and S_2 is by finding a third strand S_3 in which the bases in S_3 appear in each of S_1 and S_2. These bases must appear in the same order, but not necessarily consecutively. The longer the strand S_3 we can find, the more similar S_1 and S_2 are. In our example, the longest strand S_3 is `GTCGTCGGAAGCCGGCCGAA`.

We formalize this last notion of similarity as the longest-common-subsequence problem. A subsequence of a given sequence is just the given sequence with 0 or more elements left out. Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a ***subsequence*** of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$. For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$.

Given two sequences X and Y, we say that a sequence Z is a ***common subsequence*** of X and Y if Z is a subsequence of both X and Y. For example, if $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of both X and Y. The sequence $\langle B, C, A \rangle$ is not a *longest* common subsequence (***LCS***) of X and Y, however, since it has length 3 and the sequence $\langle B, C, B, A \rangle$, which is also common to both sequences X and Y, has length 4. The sequence $\langle B, C, B, A \rangle$ is an LCS of X and Y, as is the sequence $\langle B, D, A, B \rangle$, since X and Y have no common subsequence of length 5 or greater.

In the ***longest-common-subsequence problem***, the input is two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, and the goal is to find a maximum-length common subsequence of X and Y. This section shows how to efficiently solve the LCS problem using dynamic programming.

Step 1: Characterizing a longest common subsequence

You can solve the LCS problem with a brute-force approach: enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence you find. Each subsequence of X corresponds to a subset of the indices $\{1, 2, \ldots, m\}$ of X. Because X has 2^m subsequences, this approach requires exponential time, making it impractical for long sequences.

The LCS problem has an optimal-substructure property, however, as the following theorem shows. As we'll see, the natural classes of subproblems correspond to pairs of "prefixes" of the two input sequences. To be precise, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, we define the ith ***prefix*** of X, for $i = 0, 1, \ldots, m$, as $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. For example, if $X = \langle A, B, C, B, D, A, B \rangle$, then $X_4 = \langle A, B, C, B \rangle$ and X_0 is the empty sequence.

Theorem 14.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is an LCS of X and Y_{n-1}.

Proof (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, contradicting the supposition that Z is a *longest* common subsequence of X and Y. Thus, we must have $z_k = x_m = y_n$. Now, the prefix Z_{k-1} is a length-$(k - 1)$ common subsequence of X_{m-1} and Y_{n-1}. We wish to show that it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k - 1$. Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction.

(2) If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.

(3) The proof is symmetric to (2). ■

The way that Theorem 14.1 characterizes longest common subsequences says that an LCS of two sequences contains within it an LCS of prefixes of the two sequences. Thus, the LCS problem has an optimal-substructure property. A recursive solution also has the overlapping-subproblems property, as we'll see in a moment.

Step 2: A recursive solution

Theorem 14.1 implies that you should examine either one or two subproblems when finding an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. If $x_m = y_n$, you need to find an LCS of X_{m-1} and Y_{n-1}. Appending $x_m = y_n$ to this LCS yields an LCS of X and Y. If $x_m \neq y_n$, then you have to solve two subproblems: finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}.

Whichever of these two LCSs is longer is an LCS of X and Y. Because these cases exhaust all possibilities, one of the optimal subproblem solutions must appear within an LCS of X and Y.

The LCS problem has the overlapping-subproblems property. Here's how. To find an LCS of X and Y, you might need to find the LCSs of X and Y_{n-1} and of X_{m-1} and Y. But each of these subproblems has the subsubproblem of finding an LCS of X_{m-1} and Y_{n-1}. Many other subproblems share subsubproblems.

As in the matrix-chain multiplication problem, solving the LCS problem recursively involves establishing a recurrence for the value of an optimal solution. Let's define $c[i, j]$ to be the length of an LCS of the sequences X_i and Y_j. If either $i = 0$ or $j = 0$, one of the sequences has length 0, and so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases} \tag{14.9}$$

In this recursive formulation, a condition in the problem restricts which subproblems to consider. When $x_i = y_j$, you can and should consider the subproblem of finding an LCS of X_{i-1} and Y_{j-1}. Otherwise, you instead consider the two subproblems of finding an LCS of X_i and Y_{j-1} and of X_{i-1} and Y_j. In the previous dynamic-programming algorithms we have examined—for rod cutting and matrix-chain multiplication—we didn't rule out any subproblems due to conditions in the problem. Finding an LCS is not the only dynamic-programming algorithm that rules out subproblems based on conditions in the problem. For example, the edit-distance problem (see Problem 14-5) has this characteristic.

Step 3: Computing the length of an LCS

Based on equation (14.9), you could write an exponential-time recursive algorithm to compute the length of an LCS of two sequences. Since the LCS problem has only $\Theta(mn)$ distinct subproblems (computing $c[i, j]$ for $0 \le i \le m$ and $0 \le j \le n$), dynamic programming can compute the solutions bottom up.

The procedure LCS-LENGTH on the next page takes two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ as inputs, along with their lengths. It stores the $c[i, j]$ values in a table $c[0:m, 0:n]$, and it computes the entries in ***row-major*** order. That is, the procedure fills in the first row of c from left to right, then the second row, and so on. The procedure also maintains the table $b[1:m, 1:n]$ to help in constructing an optimal solution. Intuitively, $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$. The procedure returns the b and c tables, where $c[m, n]$ contains the length of an LCS of X and Y. Figure 14.8 shows the tables produced by LCS-LENGTH on the

sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The running time of the procedure is $\Theta(mn)$, since each table entry takes $\Theta(1)$ time to compute.

```
LCS-LENGTH(X, Y, m, n)
let b[1 : m, 1 : n] and c[0 : m, 0 : n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m          // compute table entries in row-major order
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return              // the LCS has length 0
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i           // same as y_j
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

Step 4: Constructing an LCS

With the b table returned by LCS-LENGTH, you can quickly construct an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Begin at $b[m, n]$ and trace through the table by following the arrows. Each "↖" encountered in an entry $b[i, j]$ implies that $x_i = y_j$ is an element of the LCS that LCS-LENGTH found. This method gives you the elements of this LCS in reverse order. The recursive procedure PRINT-LCS prints out an LCS of X and Y in the proper, forward order.

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

Figure 14.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner, as shown by the sequence shaded blue. Each "↖" on the shaded-blue sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

The initial call is PRINT-LCS(b, X, m, n). For the b table in Figure 14.8, this procedure prints $BCBA$. The procedure takes $O(m + n)$ time, since it decrements at least one of i and j in each recursive call.

Improving the code

Once you have developed an algorithm, you will often find that you can improve on the time or space it uses. Some changes can simplify the code and improve constant factors but otherwise yield no asymptotic improvement in performance. Others can yield substantial asymptotic savings in time and space.

In the LCS algorithm, for example, you can eliminate the b table altogether. Each $c[i, j]$ entry depends on only three other c table entries: $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$. Given the value of $c[i, j]$, you can determine in $O(1)$ time which of these three values was used to compute $c[i, j]$, without inspecting table b. Thus, you can reconstruct an LCS in $O(m+n)$ time using a procedure similar to PRINT-LCS. (Exercise 14.4-2 asks you to give the pseudocode.) Although this method saves $\Theta(mn)$ space, the auxiliary space requirement for computing

an LCS does not asymptotically decrease, since the c table takes $\Theta(mn)$ space anyway.

You can, however, reduce the asymptotic space requirements for LCS-LENGTH, since it needs only two rows of table c at a time: the row being computed and the previous row. (In fact, as Exercise 14.4-4 asks you to show, you can use only slightly more than the space for one row of c to compute the length of an LCS.) This improvement works if you need only the length of an LCS. If you need to reconstruct the elements of an LCS, the smaller table does not keep enough information to retrace the algorithm's steps in $O(m + n)$ time.

Exercises

14.4-1
Determine an LCS of $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ and $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.

14.4-2
Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ in $O(m + n)$ time, without using the b table.

14.4-3
Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time.

14.4-4
Show how to compute the length of an LCS using only $2 \cdot \min\{m, n\}$ entries in the c table plus $O(1)$ additional space. Then show how to do the same thing, but using $\min\{m, n\}$ entries plus $O(1)$ additional space.

14.4-5
Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

★ ***14.4-6***
Give an $O(n \lg n)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. (*Hint:* The last element of a candidate subsequence of length i is at least as large as the last element of a candidate subsequence of length $i - 1$. Maintain candidate subsequences by linking them through the input sequence.)

14.5 Optimal binary search trees

Suppose that you are designing a program to translate text from English to Latvian. For each occurrence of each English word in the text, you need to look up its Latvian equivalent. You can perform these lookup operations by building a binary search tree with n English words as keys and their Latvian equivalents as satellite data. Because you will search the tree for each individual word in the text, you want the total time spent searching to be as low as possible. You can ensure an $O(\lg n)$ search time per occurrence by using a red-black tree or any other balanced binary search tree. Words appear with different frequencies, however, and a frequently used word such as *the* can end up appearing far from the root while a rarely used word such as *naumachia* appears near the root. Such an organization would slow down the translation, since the number of nodes visited when searching for a key in a binary search tree equals 1 plus the depth of the node containing the key. You want words that occur frequently in the text to be placed nearer the root.[8] Moreover, some words in the text might have no Latvian translation,[9] and such words would not appear in the binary search tree at all. How can you organize a binary search tree so as to minimize the number of nodes visited in all searches, given that you know how often each word occurs?

What you need is an ***optimal binary search tree***. Formally, given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys such that $k_1 < k_2 < \cdots < k_n$, build a binary search tree containing them. For each key k_i, you are given the probability p_i that any given search is for key k_i. Since some searches may be for values not in K, you also have $n + 1$ "dummy" keys $d_0, d_1, d_2, \ldots, d_n$ representing those values. In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n, and for $i = 1, 2, \ldots, n - 1$, the dummy key d_i represents all values between k_i and k_{i+1}. For each dummy key d_i, you have the probability q_i that a search corresponds to d_i. Figure 14.9 shows two binary search trees for a set of $n = 5$ keys. Each key k_i is an internal node, and each dummy key d_i is a leaf. Since every search is either successful (finding some key k_i) or unsuccessful (finding some dummy key d_i), we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 . \tag{14.10}$$

[8] If the subject of the text is ancient Rome, you might want *naumachia* to appear near the root.

[9] Yes, *naumachia* has a Latvian counterpart: *nomačija*.

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

(a)

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
Total			2.75

(b)

Figure 14.9 Two binary search trees for a set of $n = 5$ keys with the following probabilities:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(a) A binary search tree with expected search cost 2.80. **(b)** A binary search tree with expected search cost 2.75. This tree is optimal.

Knowing the probabilities of searches for each key and each dummy key allows us to determine the expected cost of a search in a given binary search tree T. Let us assume that the actual cost of a search equals the number of nodes examined, which is the depth of the node found by the search in T, plus 1. Then the expected cost of a search in T is

$$\begin{aligned} \mathrm{E}\,[\text{search cost in } T] &= \sum_{i=1}^{n} (\mathrm{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\mathrm{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \mathrm{depth}_T(d_i) \cdot q_i \,, \end{aligned} \tag{14.11}$$

where depth_T denotes a node's depth in the tree T. The last equation follows from equation (14.10). Figure 14.9 shows how to calculate the expected search cost node by node.

For a given set of probabilities, your goal is to construct a binary search tree whose expected search cost is smallest. We call such a tree an ***optimal binary search tree***. Figure 14.9(a) shows one binary search tree, with expected cost 2.80, for the probabilities given in the figure caption. Part (b) of the figure displays an optimal binary search tree, with expected cost 2.75. This example demonstrates that an optimal binary search tree is not necessarily a tree whose overall height is smallest. Nor does an optimal binary search tree always have the key with the greatest probability at the root. Here, key k_5 has the greatest search probability of any key, yet the root of the optimal binary search tree shown is k_2. (The lowest expected cost of any binary search tree with k_5 at the root is 2.85.)

As with matrix-chain multiplication, exhaustive checking of all possibilities fails to yield an efficient algorithm. You can label the nodes of any n-node binary tree with the keys $k_1, k_2, \ldots, k_n$ to construct a binary search tree, and then add in the dummy keys as leaves. In Problem 12-4 on page 329, we saw that the number of binary trees with n nodes is $\Omega(4^n/n^{3/2})$. Thus you would need to examine an exponential number of binary search trees to perform an exhaustive search. We'll see how to solve this problem more efficiently with dynamic programming.

Step 1: The structure of an optimal binary search tree

To characterize the optimal substructure of optimal binary search trees, we start with an observation about subtrees. Consider any subtree of a binary search tree. It must contain keys in a contiguous range $k_i, \ldots, k_j$, for some $1 \le i \le j \le n$. In addition, a subtree that contains keys $k_i, \ldots, k_j$ must also have as its leaves the dummy keys $d_{i-1}, \ldots, d_j$.

Now we can state the optimal substructure: if an optimal binary search tree T has a subtree T' containing keys $k_i, \ldots, k_j$, then this subtree T' must be optimal as well for the subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$. The usual cut-and-paste argument applies. If there were a subtree T'' whose expected cost is lower than that of T', then cutting T' out of T and pasting in T'' would result in a binary search tree of lower expected cost than T, thus contradicting the optimality of T.

With the optimal substructure in hand, here is how to construct an optimal solution to the problem from optimal solutions to subproblems. Given keys $k_i, \ldots, k_j$, one of these keys, say k_r ($i \le r \le j$), is the root of an optimal subtree containing these keys. The left subtree of the root k_r contains the keys $k_i, \ldots, k_{r-1}$ (and dummy keys $d_{i-1}, \ldots, d_{r-1}$), and the right subtree contains the keys $k_{r+1}, \ldots, k_j$ (and dummy keys $d_r, \ldots, d_j$). As long as you examine all candidate roots k_r,

where $i \le r \le j$, and you determine all optimal binary search trees containing $k_i, \ldots, k_{r-1}$ and those containing $k_{r+1}, \ldots, k_j$, you are guaranteed to find an optimal binary search tree.

There is one technical detail worth understanding about "empty" subtrees. Suppose that in a subtree with keys $k_i, \ldots, k_j$, you select k_i as the root. By the above argument, k_i's left subtree contains the keys $k_i, \ldots, k_{i-1}$: no keys at all. Bear in mind, however, that subtrees also contain dummy keys. We adopt the convention that a subtree containing keys $k_i, \ldots, k_{i-1}$ has no actual keys but does contain the single dummy key d_{i-1}. Symmetrically, if you select k_j as the root, then k_j's right subtree contains the keys $k_{j+1}, \ldots, k_j$. This right subtree contains no actual keys, but it does contain the dummy key d_j.

Step 2: A recursive solution

To define the value of an optimal solution recursively, the subproblem domain is finding an optimal binary search tree containing the keys $k_i, \ldots, k_j$, where $i \ge 1$, $j \le n$, and $j \ge i - 1$. (When $j = i - 1$, there is just the dummy key d_{i-1}, but no actual keys.) Let $e[i, j]$ denote the expected cost of searching an optimal binary search tree containing the keys $k_i, \ldots, k_j$. Your goal is to compute $e[1, n]$, the expected cost of searching an optimal binary search tree for all the actual and dummy keys.

The easy case occurs when $j = i - 1$. Then the subproblem consists of just the dummy key d_{i-1}. The expected search cost is $e[i, i-1] = q_{i-1}$.

When $j \ge i$, you need to select a root k_r from among $k_i, \ldots, k_j$ and then make an optimal binary search tree with keys $k_i, \ldots, k_{r-1}$ as its left subtree and an optimal binary search tree with keys $k_{r+1}, \ldots, k_j$ as its right subtree. What happens to the expected search cost of a subtree when it becomes a subtree of a node? The depth of each node in the subtree increases by 1. By equation (14.11), the expected search cost of this subtree increases by the sum of all the probabilities in the subtree. For a subtree with keys $k_i, \ldots, k_j$, denote this sum of probabilities as

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \; . \tag{14.12}$$

Thus, if k_r is the root of an optimal subtree containing keys $k_i, \ldots, k_j$, we have

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j)) \; .$$

Noting that

$$w(i, j) = w(i, r-1) + p_r + w(r+1, j) \; ,$$

we rewrite $e[i, j]$ as

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j) \ . \tag{14.13}$$

The recursive equation (14.13) assumes that you know which node k_r to use as the root. Of course, you choose the root that gives the lowest expected search cost, giving the final recursive formulation:

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \ , \\ \min\{e[i, r-1] + e[r+1, j] + w(i, j) : i \le r \le j\} & \text{if } i \le j \ . \end{cases} \tag{14.14}$$

The $e[i, j]$ values give the expected search costs in optimal binary search trees. To help keep track of the structure of optimal binary search trees, define $root[i, j]$, for $1 \le i \le j \le n$, to be the index r for which k_r is the root of an optimal binary search tree containing keys $k_i, \ldots, k_j$. Although we'll see how to compute the values of $root[i, j]$, the construction of an optimal binary search tree from these values is left as Exercise 14.5-1.

Step 3: Computing the expected search cost of an optimal binary search tree

At this point, you may have noticed some similarities between our characterizations of optimal binary search trees and matrix-chain multiplication. For both problem domains, the subproblems consist of contiguous index subranges. A direct, recursive implementation of equation (14.14) would be just as inefficient as a direct, recursive matrix-chain multiplication algorithm. Instead, you can store the $e[i, j]$ values in a table $e[1 : n+1, 0 : n]$. The first index needs to run to $n+1$ rather than n because in order to have a subtree containing only the dummy key d_n, you need to compute and store $e[n+1, n]$. The second index needs to start from 0 because in order to have a subtree containing only the dummy key d_0, you need to compute and store $e[1, 0]$. Only the entries $e[i, j]$ for which $j \ge i-1$ are filled in. The table $root[i, j]$ records the root of the subtree containing keys $k_i, \ldots, k_j$ and uses only the entries for which $1 \le i \le j \le n$.

One other table makes the dynamic-programming algorithm a little faster. Instead of computing the value of $w(i, j)$ from scratch every time you compute $e[i, j]$, which would take $\Theta(j - i)$ additions, store these values in a table $w[1 : n+1, 0 : n]$. For the base case, compute $w[i, i-1] = q_{i-1}$ for $1 \le i \le n+1$. For $j \ge i$, compute

$$w[i, j] = w[i, j-1] + p_j + q_j \ . \tag{14.15}$$

Thus, you can compute the $\Theta(n^2)$ values of $w[i, j]$ in $\Theta(1)$ time each.

The OPTIMAL-BST procedure on the next page takes as inputs the probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$ and the size n, and it returns the tables e and $root$. From the description above and the similarity to the MATRIX-CHAIN-ORDER procedure

in Section 14.2, you should find the operation of this procedure to be fairly straightforward. The **for** loop of lines 2–4 initializes the values of $e[i, i-1]$ and $w[i, i-1]$. Then the **for** loop of lines 5–14 uses the recurrences (14.14) and (14.15) to compute $e[i, j]$ and $w[i, j]$ for all $1 \le i \le j \le n$. In the first iteration, when $l = 1$, the loop computes $e[i, i]$ and $w[i, i]$ for $i = 1, 2, \ldots, n$. The second iteration, with $l = 2$, computes $e[i, i+1]$ and $w[i, i+1]$ for $i = 1, 2, \ldots, n-1$, and so on. The innermost **for** loop, in lines 10–14, tries each candidate index r to determine which key k_r to use as the root of an optimal binary search tree containing keys $k_i, \ldots, k_j$. This **for** loop saves the current value of the index r in $root[i, j]$ whenever it finds a better key to use as the root.

```
OPTIMAL-BST(p, q, n)
1  let e[1 : n + 1, 0 : n], w[1 : n + 1, 0 : n],
           and root[1 : n, 1 : n] be new tables
2  for i = 1 to n + 1                  // base cases
3      e[i, i − 1] = q_{i−1}           // equation (14.14)
4      w[i, i − 1] = q_{i−1}
5  for l = 1 to n
6      for i = 1 to n − l + 1
7          j = i + l − 1
8          e[i, j] = ∞
9          w[i, j] = w[i, j − 1] + p_j + q_j          // equation (14.15)
10         for r = i to j                             // try all possible roots r
11             t = e[i, r − 1] + e[r + 1, j] + w[i, j]  // equation (14.14)
12             if t < e[i, j]                         // new minimum?
13                 e[i, j] = t
14                 root[i, j] = r
15 return e and root
```

Figure 14.10 shows the tables $e[i, j]$, $w[i, j]$, and $root[i, j]$ computed by the procedure OPTIMAL-BST on the key distribution shown in Figure 14.9. As in the matrix-chain multiplication example of Figure 14.5, the tables are rotated to make the diagonals run horizontally. OPTIMAL-BST computes the rows from bottom to top and from left to right within each row.

The OPTIMAL-BST procedure takes $\Theta(n^3)$ time, just like MATRIX-CHAIN-ORDER. Its running time is $O(n^3)$, since its **for** loops are nested three deep and each loop index takes on at most n values. The loop indices in OPTIMAL-BST do not have exactly the same bounds as those in MATRIX-CHAIN-ORDER, but they are within at most 1 in all directions. Thus, like MATRIX-CHAIN-ORDER, the OPTIMAL-BST procedure takes $\Omega(n^3)$ time.

Figure 14.10 The tables $e[i, j]$, $w[i, j]$, and $root[i, j]$ computed by OPTIMAL-BST on the key distribution shown in Figure 14.9. The tables are rotated so that the diagonals run horizontally.

Exercises

14.5-1

Write pseudocode for the procedure CONSTRUCT-OPTIMAL-BST$(root, n)$ which, given the table $root[1:n, 1:n]$, outputs the structure of an optimal binary search tree. For the example in Figure 14.10, your procedure should print out the structure

k_2 is the root
k_1 is the left child of k_2
d_0 is the left child of k_1
d_1 is the right child of k_1
k_5 is the right child of k_2
k_4 is the left child of k_5
k_3 is the left child of k_4
d_2 is the left child of k_3
d_3 is the right child of k_3
d_4 is the right child of k_4
d_5 is the right child of k_5

corresponding to the optimal binary search tree shown in Figure 14.9(b).

14.5-2

Determine the cost and structure of an optimal binary search tree for a set of $n = 7$ keys with the following probabilities:

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

14.5-3

Suppose that instead of maintaining the table $w[i, j]$, you computed the value of $w(i, j)$ directly from equation (14.12) in line 9 of OPTIMAL-BST and used this computed value in line 11. How would this change affect the asymptotic running time of OPTIMAL-BST?

★ ***14.5-4***

Knuth [264] has shown that there are always roots of optimal subtrees such that $root[i, j-1] \le root[i, j] \le root[i+1, j]$ for all $1 \le i < j \le n$. Use this fact to modify the OPTIMAL-BST procedure to run in $\Theta(n^2)$ time.

Problems

14-1 Longest simple path in a directed acyclic graph

You are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights and two distinguished vertices s and t. The ***weight*** of a path is the sum of the weights of the edges in the path. Describe a dynamic-programming approach for finding a longest weighted simple path from s to t. What is the running time of your algorithm?

14-2 Longest palindrome subsequence

A ***palindrome*** is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

14-3 Bitonic euclidean traveling-salesperson problem

In the ***euclidean traveling-salesperson problem***, you are given a set of n points in the plane, and your goal is to find the shortest closed tour that connects all n points.

(a) (b)

Figure 14.11 Seven points in the plane, shown on a unit grid. **(a)** The shortest closed tour, with length approximately 24.89. This tour is not bitonic. **(b)** The shortest bitonic tour for the same set of points. Its length is approximately 25.58.

Figure 14.11(a) shows the solution to a 7-point problem. The general problem is NP-hard, and its solution is therefore believed to require more than polynomial time (see Chapter 34).

J. L. Bentley has suggested simplifying the problem by considering only ***bitonic tours***, that is, tours that start at the leftmost point, go strictly rightward to the rightmost point, and then go strictly leftward back to the starting point. Figure 14.11(b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.

Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate and that all operations on real numbers take unit time. (*Hint:* Scan left to right, maintaining optimal possibilities for the two parts of the tour.)

14-4 Printing neatly

Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width). The input text is a sequence of n words of lengths $l_1, l_2, \ldots, l_n$, measured in characters, which are to be printed neatly on a number of lines that hold a maximum of M characters each. No word exceeds the line length, so that $l_i \le M$ for $i = 1, 2, \ldots, n$. The criterion of "neatness" is as follows. If a given line contains words i through j, where $i \le j$, and exactly one space appears between words, then the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. The goal is to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly. Analyze the running time and space requirements of your algorithm.

14-5 Edit distance

In order to transform a source string of text $x[1:m]$ to a target string $y[1:n]$, you can perform various transformation operations. The goal is, given x and y, to produce a series of transformations that changes x to y. An array z—assumed to be large enough to hold all the characters it needs—holds the intermediate results. Initially, z is empty, and at termination, you should have $z[j] = y[j]$ for $j = 1, 2, \ldots, n$. The procedure for solving this problem maintains current indices i into x and j into z, and the operations are allowed to alter z and these indices. Initially, $i = j = 1$. Every character in x must be examined during the transformation, which means that at the end of the sequence of transformation operations, $i = m + 1$.

You may choose from among six transformation operations, each of which has a constant cost that depends on the operation:

Copy a character from x to z by setting $z[j] = x[i]$ and then incrementing both i and j. This operation examines $x[i]$ and has cost Q_C.

Replace a character from x by another character c, by setting $z[j] = c$, and then incrementing both i and j. This operation examines $x[i]$ and has cost Q_R.

Delete a character from x by incrementing i but leaving j alone. This operation examines $x[i]$ and has cost Q_D.

Insert the character c into z by setting $z[j] = c$ and then incrementing j, but leaving i alone. This operation examines no characters of x and has cost Q_I.

Twiddle (i.e., exchange) the next two characters by copying them from x to z but in the opposite order: setting $z[j] = x[i+1]$ and $z[j+1] = x[i]$, and then setting $i = i + 2$ and $j = j + 2$. This operation examines $x[i]$ and $x[i+1]$ and has cost Q_T.

Kill the remainder of x by setting $i = m + 1$. This operation examines all characters in x that have not yet been examined. This operation, if performed, must be the final operation. It has cost Q_K.

Figure 14.12 gives one way to transform the source string `algorithm` to the target string `altruistic`. Several other sequences of transformation operations can transform `algorithm` to `altruistic`.

Assume that $Q_C < Q_D + Q_I$ and $Q_R < Q_D + Q_I$, since otherwise, the copy and replace operations would not be used. The cost of a given sequence of transformation operations is the sum of the costs of the individual operations in the sequence. For the sequence above, the cost of transforming `algorithm` to `altruistic` is $3Q_C + Q_R + Q_D + 4Q_I + Q_T + Q_K$.

a. Given two sequences $x[1:m]$ and $y[1:n]$ and the costs of the transformation operations, the ***edit distance*** from x to y is the cost of the least expensive op-

Operation	x	z
initial strings	algorithm	_
copy	algorithm	a_
copy	algorithm	al_
replace by t	algorithm	alt_
delete	algorithm	alt_
copy	algorithm	altr_
insert u	algorithm	altru_
insert i	algorithm	altrui_
insert s	algorithm	altruis_
twiddle	algorithm	altruisti_
insert c	algorithm	altruistic_
kill	algorithm_	altruistic_

Figure 14.12 A sequence of operations that transforms the source `algorithm` to the target string `altruistic`. The underlined characters are $x[i]$ and $z[j]$ after the operation.

eration sequence that transforms x to y. Describe a dynamic-programming algorithm that finds the edit distance from $x[1:m]$ to $y[1:n]$ and prints an optimal operation sequence. Analyze the running time and space requirements of your algorithm.

The edit-distance problem generalizes the problem of aligning two DNA sequences (see, for example, Setubal and Meidanis [405, Section 3.2]). There are several methods for measuring the similarity of two DNA sequences by aligning them. One such method to align two sequences x and y consists of inserting spaces at arbitrary locations in the two sequences (including at either end) so that the resulting sequences x' and y' have the same length but do not have a space in the same position (i.e., for no position j are both $x'[j]$ and $y'[j]$ a space). Then we assign a "score" to each position. Position j receives a score as follows:

- $+1$ if $x'[j] = y'[j]$ and neither is a space,
- -1 if $x'[j] \neq y'[j]$ and neither is a space,
- -2 if either $x'[j]$ or $y'[j]$ is a space.

The score for the alignment is the sum of the scores of the individual positions. For example, given the sequences $x =$ `GATCGGCAT` and $y =$ `CAATGTGAATC`, one alignment is

```
G ATCG GCAT
CAAT GTGAATC
-*++*+*+-++*
```

A + under a position indicates a score of $+1$ for that position, a – indicates a score of -1, and a $*$ indicates a score of -2, so that this alignment has a total score of $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$.

b. Explain how to cast the problem of finding an optimal alignment as an edit-distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill.

14-6 Planning a company party

Professor Blutarsky is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure, that is, the supervisor relation forms a tree rooted at the president. The human resources department has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

Professor Blutarsky is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation described in Section 10.3. Each node of the tree holds, in addition to the pointers, the name of an employee and that employee's conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

14-7 Viterbi algorithm

Dynamic programming on a directed graph can play a part in speech recognition. A directed graph $G = (V, E)$ with labeled edges forms a formal model of a person speaking a restricted language. Each edge $(u, v) \in E$ is labeled with a sound $\sigma(u, v)$ from a finite set Σ of sounds. Each directed path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds produced by the model, with the label of a path being the concatenation of the labels of the edges on that path.

a. Describe an efficient algorithm that, given an edge-labeled directed graph G with distinguished vertex v_0 and a sequence $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ of sounds from Σ, returns a path in G that begins at v_0 and has s as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of your algorithm. (*Hint:* You may find concepts from Chapter 20 useful.)

Now suppose that every edge $(u, v) \in E$ has an associated nonnegative probability $p(u, v)$ of being traversed, so that the corresponding sound is produced. The sum of the probabilities of the edges leaving any vertex equals 1. The probability of a path is defined to be the product of the probabilities of its edges. Think of

the probability of a path beginning at vertex v_0 as the probability that a "random walk" beginning at v_0 follows the specified path, where the edge leaving a vertex u is taken randomly, according to the probabilities of the available edges leaving u.

b. Extend your answer to part (a) so that if a path is returned, it is a *most probable path* starting at vertex v_0 and having label s. Analyze the running time of your algorithm.

14-8 Image compression by seam carving

Suppose that you are given a color picture consisting of an $m \times n$ array $A[1:m, 1:n]$ of pixels, where each pixel specifies a triple of red, green, and blue (RGB) intensities. You want to compress this picture slightly, by removing one pixel from each of the m rows, so that the whole picture becomes one pixel narrower. To avoid incongruous visual effects, however, the pixels removed in two adjacent rows must lie in either the same column or adjacent columns. In this way, the pixels removed form a "seam" from the top row to the bottom row, where successive pixels in the seam are adjacent vertically or diagonally.

a. Show that the number of such possible seams grows at least exponentially in m, assuming that $n > 1$.

b. Suppose now that along with each pixel $A[i, j]$, you are given a real-valued disruption measure $d[i, j]$, indicating how disruptive it would be to remove pixel $A[i, j]$. Intuitively, the lower a pixel's disruption measure, the more similar the pixel is to its neighbors. Define the disruption measure of a seam as the sum of the disruption measures of its pixels.

Give an algorithm to find a seam with the lowest disruption measure. How efficient is your algorithm?

14-9 Breaking a string

A certain string-processing programming language allows you to break a string into two pieces. Because this operation copies the string, it costs n time units to break a string of n characters into two pieces. Suppose that you want to break a string into many pieces. The order in which the breaks occur can affect the total amount of time used. For example, suppose that you want to break a 20-character string after characters 2, 8, and 10 (numbering the characters in ascending order from the left-hand end, starting from 1). If you program the breaks to occur in left-to-right order, then the first break costs 20 time units, the second break costs 18 time units (breaking the string from characters 3 to 20 at character 8), and the third break costs 12 time units, totaling 50 time units. If you program the breaks to occur in right-to-left order, however, then the first break costs 20 time units, the

second break costs 10 time units, and the third break costs 8 time units, totaling 38 time units. In yet another order, you could break first at 8 (costing 20), then break the left piece at 2 (costing another 8), and finally the right piece at 10 (costing 12), for a total cost of 40.

Design an algorithm that, given the numbers of characters after which to break, determines a least-cost way to sequence those breaks. More formally, given an array $L[1:m]$ containing the break points for a string of n characters, compute the lowest cost for a sequence of breaks, along with a sequence of breaks that achieves this cost.

14-10 Planning an investment strategy

Your knowledge of algorithms helps you obtain an exciting job with a hot startup, along with a $10,000 signing bonus. You decide to invest this money with the goal of maximizing your return at the end of 10 years. You decide to use your investment manager, G. I. Luvcache, to manage your signing bonus. The company that Luvcache works with requires you to observe the following rules. It offers n different investments, numbered 1 through n. In each year j, investment i provides a return rate of r_{ij}. In other words, if you invest d dollars in investment i in year j, then at the end of year j, you have $d r_{ij}$ dollars. The return rates are guaranteed, that is, you are given all the return rates for the next 10 years for each investment. You make investment decisions only once per year. At the end of each year, you can leave the money made in the previous year in the same investments, or you can shift money to other investments, by either shifting money between existing investments or moving money to a new investment. If you do not move your money between two consecutive years, you pay a fee of f_1 dollars, whereas if you switch your money, you pay a fee of f_2 dollars, where $f_2 > f_1$. You pay the fee once per year at the end of the year, and it is the same amount, f_2, whether you move money in and out of only one investment, or in and out of many investments.

a. The problem, as stated, allows you to invest your money in multiple investments in each year. Prove that there exists an optimal investment strategy that, in each year, puts all the money into a single investment. (Recall that an optimal investment strategy maximizes the amount of money after 10 years and is not concerned with any other objectives, such as minimizing risk.)

b. Prove that the problem of planning your optimal investment strategy exhibits optimal substructure.

c. Design an algorithm that plans your optimal investment strategy. What is the running time of your algorithm?

d. Suppose that Luvcache's company imposes the additional restriction that, at any point, you can have no more than \$15,000 in any one investment. Show that the problem of maximizing your income at the end of 10 years no longer exhibits optimal substructure.

14-11 Inventory planning

The Rinky Dink Company makes machines that resurface ice rinks. The demand for such products varies from month to month, and so the company needs to develop a strategy to plan its manufacturing given the fluctuating, but predictable, demand. The company wishes to design a plan for the next n months. For each month i, the company knows the demand d_i, that is, the number of machines that it will sell. Let $D = \sum_{i=1}^{n} d_i$ be the total demand over the next n months. The company keeps a full-time staff who provide labor to manufacture up to m machines per month. If the company needs to make more than m machines in a given month, it can hire additional, part-time labor, at a cost that works out to c dollars per machine. Furthermore, if the company is holding any unsold machines at the end of a month, it must pay inventory costs. The company can hold up to D machines, with the cost for holding j machines given as a function $h(j)$ for $j = 1, 2, \ldots, D$ that monotonically increases with j.

Give an algorithm that calculates a plan for the company that minimizes its costs while fulfilling all the demand. The running time should be polynomial in n and D.

14-12 Signing free-agent baseball players

Suppose that you are the general manager for a major-league baseball team. During the off-season, you need to sign some free-agent players for your team. The team owner has given you a budget of $\$X$ to spend on free agents. You are allowed to spend less than $\$X$, but the owner will fire you if you spend any more than $\$X$.

You are considering N different positions, and for each position, P free-agent players who play that position are available.[10] Because you do not want to overload your roster with too many players at any position, for each position you may sign at most one free agent who plays that position. (If you do not sign any players at a particular position, then you plan to stick with the players you already have at that position.)

---

[10] Although there are nine positions on a baseball team, N is not necessarily equal to 9 because some general managers have particular ways of thinking about positions. For example, a general manager might consider right-handed pitchers and left-handed pitchers to be separate "positions," as well as starting pitchers, long relief pitchers (relief pitchers who can pitch several innings), and short relief pitchers (relief pitchers who normally pitch at most only one inning).

To determine how valuable a player is going to be, you decide to use a sabermetric statistic[11] known as "WAR," or "wins above replacement." A player with a higher WAR is more valuable than a player with a lower WAR. It is not necessarily more expensive to sign a player with a higher WAR than a player with a lower WAR, because factors other than a player's value determine how much it costs to sign them.

For each available free-agent player p, you have three pieces of information:

- the player's position,
- $p.cost$, the amount of money it costs to sign the player, and
- $p.war$, the player's WAR.

Devise an algorithm that maximizes the total WAR of the players you sign while spending no more than $\$X$. You may assume that each player signs for a multiple of \$100,000. Your algorithm should output the total WAR of the players you sign, the total amount of money you spend, and a list of which players you sign. Analyze the running time and space requirement of your algorithm.

Chapter notes

Bellman [44] began the systematic study of dynamic programming in 1955, publishing a book about it in 1957. The word "programming," both here and in linear programming, refers to using a tabular solution method. Although optimization techniques incorporating elements of dynamic programming were known earlier, Bellman provided the area with a solid mathematical basis.

Galil and Park [172] classify dynamic-programming algorithms according to the size of the table and the number of other table entries each entry depends on. They call a dynamic-programming algorithm tD/eD if its table size is $O(n^t)$ and each entry depends on $O(n^e)$ other entries. For example, the matrix-chain multiplication algorithm in Section 14.2 is $2D/1D$, and the longest-common-subsequence algorithm in Section 14.4 is $2D/0D$.

The MATRIX-CHAIN-ORDER algorithm on page 378 is by Muraoka and Kuck [339]. Hu and Shing [230, 231] give an $O(n \lg n)$-time algorithm for the matrix-chain multiplication problem.

The $O(mn)$-time algorithm for the longest-common-subsequence problem appears to be a folk algorithm. Knuth [95] posed the question of whether subquadratic

[11] ***Sabermetrics*** is the application of statistical analysis to baseball records. It provides several ways to compare the relative values of individual players.

algorithms for the LCS problem exist. Masek and Paterson [316] answered this question in the affirmative by giving an algorithm that runs in $O(mn/\lg n)$ time, where $n \leq m$ and the sequences are drawn from a set of bounded size. For the special case in which no element appears more than once in an input sequence, Szymanski [425] shows how to solve the problem in $O((n+m)\lg(n+m))$ time. Many of these results extend to the problem of computing string edit distances (Problem 14-5).

An early paper on variable-length binary encodings by Gilbert and Moore [181], which had applications to constructing optimal binary search trees for the case in which all probabilities p_i are 0, contains an $O(n^3)$-time algorithm. Aho, Hopcroft, and Ullman [5] present the algorithm from Section 14.5. Splay trees [418], which modify the tree in response to the search queries, come within a constant factor of the optimal bounds without being initialized with the frequencies. Exercise 14.5-4 is due to Knuth [264]. Hu and Tucker [232] devised an algorithm for the case in which all probabilities p_i are 0 that uses $O(n^2)$ time and $O(n)$ space. Subsequently, Knuth [261] reduced the time to $O(n \lg n)$.

Problem 14-8 is due to Avidan and Shamir [30], who have posted on the web a wonderful video illustrating this image-compression technique.

15 Greedy Algorithms

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill, and simpler, more efficient algorithms will do. A ***greedy algorithm*** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice leads to a globally optimal solution. This chapter explores optimization problems for which greedy algorithms provide optimal solutions. Before reading this chapter, you should read about dynamic programming in Chapter 14, particularly Section 14.3.

Greedy algorithms do not always yield optimal solutions, but for many problems they do. We first examine, in Section 15.1, a simple but nontrivial problem, the activity-selection problem, for which a greedy algorithm efficiently computes an optimal solution. We'll arrive at the greedy algorithm by first considering a dynamic-programming approach and then showing that an optimal solution can result from always making greedy choices. Section 15.2 reviews the basic elements of the greedy approach, giving a direct approach for proving greedy algorithms correct. Section 15.3 presents an important application of greedy techniques: designing data-compression (Huffman) codes. Finally, Section 15.4 shows that in order to decide which blocks to replace when a miss occurs in a cache, the "furthest-in-future" strategy is optimal if the sequence of block accesses is known in advance.

The greedy method is quite powerful and works well for a wide range of problems. Later chapters will present many algorithms that you can view as applications of the greedy method, including minimum-spanning-tree algorithms (Chapter 21), Dijkstra's algorithm for shortest paths from a single source (Section 22.3), and a greedy set-covering heuristic (Section 35.3). Minimum-spanning-tree algorithms furnish a classic example of the greedy method. Although you can read this chapter and Chapter 21 independently of each other, you might find it useful to read them together.

15.1 An activity-selection problem

Our first example is the problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities. Imagine that you are in charge of scheduling a conference room. You are presented with a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed ***activities*** that wish to reserve the conference room, and the room can serve only one activity at a time. Each activity a_i has a ***start time*** s_i and a ***finish time*** f_i, where $0 \le s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are ***compatible*** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$. (Assume that if your staff needs time to change over the room from one activity to the next, the changeover time is built into the intervals.) In the ***activity-selection problem***, your goal is to select a maximum-size subset of mutually compatible activities. Assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n \,. \tag{15.1}$$

(We'll see later the advantage that this assumption provides.) For example, consider the set of activities in Figure 15.1. The subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities, and another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

We'll see how to solve this problem, proceeding in several steps. First we'll explore a dynamic-programming solution, in which you consider several choices when determining which subproblems to use in an optimal solution. We'll then observe that you need to consider only one choice—the greedy choice—and that when you make the greedy choice, only one subproblem remains. Based on these observations, we'll develop a recursive greedy algorithm to solve the activity-selection problem. Finally, we'll complete the process of developing a greedy solution by converting the recursive algorithm to an iterative one. Although the steps we go through in this section are slightly more involved than is typical when developing a greedy algorithm, they illustrate the relationship between greedy algorithms and dynamic programming.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Figure 15.1 A set $\{a_1, a_2, \ldots, a_{11}\}$ of activities. Activity a_i has start time s_i and finish time f_i.

The optimal substructure of the activity-selection problem

Let's verify that the activity-selection problem exhibits optimal substructure. Denote by S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts. Suppose that you want to find a maximum set of mutually compatible activities in S_{ij}, and suppose further that such a maximum set is A_{ij}, which includes some activity a_k. By including a_k in an optimal solution, you are left with two subproblems: finding mutually compatible activities in the set S_{ik} (activities that start after activity a_i finishes and that finish before activity a_k starts) and finding mutually compatible activities in the set S_{kj} (activities that start after activity a_k finishes and that finish before activity a_j starts). Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$, so that A_{ik} contains the activities in A_{ij} that finish before a_k starts and A_{kj} contains the activities in A_{ij} that start after a_k finishes. Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the maximum-size set A_{ij} of mutually compatible activities in S_{ij} consists of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.

The usual cut-and-paste argument shows that an optimal solution A_{ij} must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}. If you could find a set A'_{kj} of mutually compatible activities in S_{kj} where $|A'_{kj}| > |A_{kj}|$, then you could use A'_{kj}, rather than A_{kj}, in a solution to the subproblem for S_{ij}. You would have constructed a set of $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ mutually compatible activities, which contradicts the assumption that A_{ij} is an optimal solution. A symmetric argument applies to the activities in S_{ik}.

This way of characterizing optimal substructure suggests that you can solve the activity-selection problem by dynamic programming. Let's denote the size of an optimal solution for the set S_{ij} by $c[i, j]$. Then, the dynamic-programming approach gives the recurrence

$$c[i, j] = c[i, k] + c[k, j] + 1 .$$

Of course, if you do not know that an optimal solution for the set S_{ij} includes activity a_k, you must examine all activities in S_{ij} to find which one to choose, so that

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max \{c[i, k] + c[k, j] + 1 : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset . \end{cases} \tag{15.2}$$

You can then develop a recursive algorithm and memoize it, or you can work bottom-up and fill in table entries as you go along. But you would be overlooking another important characteristic of the activity-selection problem that you can use to great advantage.

Making the greedy choice

What if you could choose an activity to add to an optimal solution without having to first solve all the subproblems? That could save you from having to consider all the choices inherent in recurrence (15.2). In fact, for the activity-selection problem, you need to consider only one choice: the greedy choice.

What is the greedy choice for the activity-selection problem? Intuition suggests that you should choose an activity that leaves the resource available for as many other activities as possible. Of the activities you end up choosing, one of them must be the first one to finish. Intuition says, therefore, choose the activity in S with the earliest finish time, since that leaves the resource available for as many of the activities that follow it as possible. (If more than one activity in S has the earliest finish time, then choose any such activity.) In other words, since the activities are sorted in monotonically increasing order by finish time, the greedy choice is activity a_1. Choosing the first activity to finish is not the only way to think of making a greedy choice for this problem. Exercise 15.1-3 asks you to explore other possibilities.

Once you make the greedy choice, you have only one remaining subproblem to solve: finding activities that start after a_1 finishes. Why don't you have to consider activities that finish before a_1 starts? Because $s_1 < f_1$, and because f_1 is the earliest finish time of any activity, no activity can have a finish time less than or equal to s_1. Thus, all activities that are compatible with activity a_1 must start after a_1 finishes.

Furthermore, we have already established that the activity-selection problem exhibits optimal substructure. Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes. If you make the greedy choice of activity a_1, then S_1 remains as the only subproblem to solve.[1] Optimal substructure says that if a_1 belongs to an optimal solution, then an optimal solution to the original problem consists of activity a_1 and all the activities in an optimal solution to the subproblem S_1.

One big question remains: Is this intuition correct? Is the greedy choice—in which you choose the first activity to finish—always part of some optimal solution? The following theorem shows that it is.

---

[1] We sometimes refer to the sets S_k as subproblems rather than as just sets of activities. The context will make it clear whether we are referring to S_k as a set of activities or as a subproblem whose input is that set.

Theorem 15.1
Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done, since we have shown that a_m belongs to some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$ be A_k but substituting a_m for a_j. The activities in A'_k are compatible, which follows because the activities in A_k are compatible, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m. ■

Although you might be able to solve the activity-selection problem with dynamic programming, Theorem 15.1 says that you don't need to. Instead, you can repeatedly choose the activity that finishes first, keep only the activities compatible with this activity, and repeat until no activities remain. Moreover, because you always choose the activity with the earliest finish time, the finish times of the activities that you choose must strictly increase. You can consider each activity just once overall, in monotonically increasing order of finish times.

An algorithm to solve the activity-selection problem does not need to work bottom-up, like a table-based dynamic-programming algorithm. Instead, it can work top-down, choosing an activity to put into the optimal solution that it constructs and then solving the subproblem of choosing activities from those that are compatible with those already chosen. Greedy algorithms typically have this top-down design: make a choice and then solve a subproblem, rather than the bottom-up technique of solving subproblems before making a choice.

A recursive greedy algorithm

Now that you know you can bypass the dynamic-programming approach and instead use a top-down, greedy algorithm, let's see a straightforward, recursive procedure to solve the activity-selection problem. The procedure RECURSIVE-ACTIVITY-SELECTOR on the following page takes the start and finish times of the activities, represented as arrays s and f,[2] the index k that defines the subproblem S_k it is to solve, and the size n of the original problem. It returns a maximum-

[2] Because the pseudocode takes s and f as arrays, it indexes into them with square brackets rather than with subscripts.

size set of mutually compatible activities in S_k. The procedure assumes that the n input activities are already ordered by monotonically increasing finish time, according to equation (15.1). If not, you can first sort them into this order in $O(n \lg n)$ time, breaking ties arbitrarily. In order to start, add the fictitious activity a_0 with $f_0 = 0$, so that subproblem S_0 is the entire set of activities S. The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Figure 15.2 shows how the algorithm operates on the activities in Figure 15.1. In a given recursive call RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n), the **while** loop of lines 2–3 looks for the first activity in S_k to finish. The loop examines $a_{k+1}, a_{k+2}, \ldots, a_n$, until it finds the first activity a_m that is compatible with a_k, which means that $s_m \geq f_k$. If the loop terminates because it finds such an activity, line 5 returns the union of $\{a_m\}$ and the maximum-size subset of S_m returned by the recursive call RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n). Alternatively, the loop may terminate because $m > n$, in which case the procedure has examined all activities in S_k without finding one that is compatible with a_k. In this case, $S_k = \emptyset$, and so line 6 returns $\emptyset$.

Assuming that the activities have already been sorted by finish times, the running time of the call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$ is $\Theta(n)$. To see why, observe that over all recursive calls, each activity is examined exactly once in the **while** loop test of line 2. In particular, activity a_i is examined in the last call made in which $k < i$.

An iterative greedy algorithm

The recursive procedure can be converted to an iterative one because the procedure RECURSIVE-ACTIVITY-SELECTOR is almost "tail recursive" (see Problem 7-5): it ends with a recursive call to itself followed by a union operation. It is usually a straightforward task to transform a tail-recursive procedure to an iterative form. In fact, some compilers for certain programming languages perform this task automatically.

Figure 15.2 The operation of RECURSIVE-ACTIVITY-SELECTOR on the 11 activities from Figure 15.1. Activities considered in each recursive call appear between horizontal lines. The fictitious activity a_0 finishes at time 0, and the initial call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, 11)$, selects activity a_1. In each recursive call, the activities that have already been selected are blue, and the activity shown in tan is being considered. If the starting time of an activity occurs before the finish time of the most recently added activity (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is selected. The last recursive call, RECURSIVE-ACTIVITY-SELECTOR$(s, f, 11, 11)$, returns $\emptyset$. The resulting set of selected activities is $\{a_1, a_4, a_8, a_{11}\}$.

The procedure GREEDY-ACTIVITY-SELECTOR is an iterative version of the procedure RECURSIVE-ACTIVITY-SELECTOR. It, too, assumes that the input activities are ordered by monotonically increasing finish time. It collects selected activities into a set A and returns this set when it is done.

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)
1  A = {a_1}
2  k = 1
3  for m = 2 to n
4      if s[m] ≥ f[k]          // is a_m in S_k?
5          A = A ∪ {a_m}       // yes, so choose it
6          k = m               // and continue from there
7  return A
```

The procedure works as follows. The variable k indexes the most recent addition to A, corresponding to the activity a_k in the recursive version. Since the procedure considers the activities in order of monotonically increasing finish time, f_k is always the maximum finish time of any activity in A. That is,

$$f_k = \max\{f_i : a_i \in A\} \ . \tag{15.3}$$

Lines 1–2 select activity a_1, initialize A to contain just this activity, and initialize k to index this activity. The **for** loop of lines 3–6 finds the earliest activity in S_k to finish. The loop considers each activity a_m in turn and adds a_m to A if it is compatible with all previously selected activities. Such an activity is the earliest in S_k to finish. To see whether activity a_m is compatible with every activity currently in A, it suffices by equation (15.3) to check (in line 4) that its start time s_m is not earlier than the finish time f_k of the activity most recently added to A. If activity a_m is compatible, then lines 5–6 add activity a_m to A and set k to m. The set A returned by the call GREEDY-ACTIVITY-SELECTOR(s, f) is precisely the set returned by the initial call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Like the recursive version, GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times.

Exercises

15.1-1

Give a dynamic-programming algorithm for the activity-selection problem, based on recurrence (15.2). Have your algorithm compute the sizes $c[i, j]$ as defined above and also produce the maximum-size subset of mutually compatible activities.

Assume that the inputs have been sorted as in equation (15.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

15.1-2

Suppose that instead of always selecting the first activity to finish, you instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

15.1-3

Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

15.1-4

You are given a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. You wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

(This problem is also known as the ***interval-graph coloring problem***. It is modeled by an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices have the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

15.1-5

Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, the goal is to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

15.2 Elements of the greedy strategy

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm makes the choice that seems best at the moment. This heuristic strategy does not always produce an optimal solution, but as in the activity-selection problem, sometimes it does. This section discusses some of the general properties of greedy methods.

The process that we followed in Section 15.1 to develop a greedy algorithm was a bit more involved than is typical. It consisted of the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution. (For the activity-selection problem, we formulated recurrence (15.2), but bypassed developing a recursive algorithm based solely on this recurrence.)
3. Show that if you make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

These steps highlighted in great detail the dynamic-programming underpinnings of a greedy algorithm. For example, the first cut at the activity-selection problem defined the subproblems S_{ij}, where both i and j varied. We then found that if you always make the greedy choice, you can restrict the subproblems to be of the form S_k.

An alternative approach is to fashion optimal substructure with a greedy choice in mind, so that the choice leaves just one subproblem to solve. In the activity-selection problem, start by dropping the second subscript and defining subproblems of the form S_k. Then prove that a greedy choice (the first activity a_m to finish in S_k), combined with an optimal solution to the remaining set S_m of compatible activities, yields an optimal solution to S_k. More generally, you can design greedy algorithms according to the following sequence of steps:

1. Cast the optimization problem as one in which you make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if you combine an

optimal solution to the subproblem with the greedy choice you have made, you arrive at an optimal solution to the original problem.

Later sections of this chapter will use this more direct process. Nevertheless, beneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution.

How can you tell whether a greedy algorithm will solve a particular optimization problem? No way works all the time, but the greedy-choice property and optimal substructure are the two key ingredients. If you can demonstrate that the problem has these properties, then you are well on the way to developing a greedy algorithm for it.

Greedy-choice property

The first key ingredient is the ***greedy-choice property***: you can assemble a globally optimal solution by making locally optimal (greedy) choices. In other words, when you are considering which choice to make, you make the choice that looks best in the current problem, without considering results from subproblems.

Here is where greedy algorithms differ from dynamic programming. In dynamic programming, you make a choice at each step, but the choice usually depends on the solutions to subproblems. Consequently, you typically solve dynamic-programming problems in a bottom-up manner, progressing from smaller subproblems to larger subproblems. (Alternatively, you can solve them top down, but memoizing. Of course, even though the code works top down, you still must solve the subproblems before making a choice.) In a greedy algorithm, you make whatever choice seems best at the moment and then solve the subproblem that remains. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, unlike dynamic programming, which solves the subproblems before making the first choice, a greedy algorithm makes its first choice before solving any subproblems. A dynamic-programming algorithm proceeds bottom up, whereas a greedy strategy usually progresses top down, making one greedy choice after another, reducing each given problem instance to a smaller one.

Of course, you need to prove that a greedy choice at each step yields a globally optimal solution. Typically, as in the case of Theorem 15.1, the proof examines a globally optimal solution to some subproblem. It then shows how to modify the solution to substitute the greedy choice for some other choice, resulting in one similar, but smaller, subproblem.

You can usually make the greedy choice more efficiently than when you have to consider a wider set of choices. For example, in the activity-selection problem, assuming that the activities were already sorted in monotonically increasing order by finish times, each activity needed to be examined just once. By preprocessing

the input or by using an appropriate data structure (often a priority queue), you often can make greedy choices quickly, thus yielding an efficient algorithm.

Optimal substructure

As we saw in Chapter 14, a problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing whether dynamic programming applies, and it's also essential for greedy algorithms. As an example of optimal substructure, recall how Section 15.1 demonstrated that if an optimal solution to subproblem S_{ij} includes an activity a_k, then it must also contain optimal solutions to the subproblems S_{ik} and S_{kj}. Given this optimal substructure, we argued that if you know which activity to use as a_k, you can construct an optimal solution to S_{ij} by selecting a_k along with all activities in optimal solutions to the subproblems S_{ik} and S_{kj}. This observation of optimal substructure gave rise to the recurrence (15.2) that describes the value of an optimal solution.

You will usually use a more direct approach regarding optimal substructure when applying it to greedy algorithms. As mentioned above, you have the luxury of assuming that you arrived at a subproblem by having made the greedy choice in the original problem. All you really need to do is argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem. This scheme implicitly uses induction on the subproblems to prove that making the greedy choice at every step produces an optimal solution.

Greedy versus dynamic programming

Because both the greedy and dynamic-programming strategies exploit optimal substructure, you might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices or, conversely, you might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required. To illustrate the subtle differences between the two techniques, let's investigate two variants of a classical optimization problem.

The ***0-1 knapsack problem*** is the following. A thief robbing a store wants to take the most valuable load that can be carried in a knapsack capable of carrying at most W pounds of loot. The thief can choose to take any subset of n items in the store. The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. Which items should the thief take? (We call this the 0-1 knapsack problem because for each item, the thief must either take it or leave it behind. The thief cannot take a fractional amount of an item or take an item more than once.)

In the ***fractional knapsack problem***, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as being like a gold ingot and an item in the fractional knapsack problem as more like gold dust.

Both knapsack problems exhibit the optimal-substructure property. For the 0-1 problem, if the most valuable load weighing at most W pounds includes item j, then the remaining load must be the most valuable load weighing at most $W - w_j$ pounds that the thief can take from the $n - 1$ original items excluding item j. For the comparable fractional problem, if if the most valuable load weighing at most W pounds includes weight w of item j, then the remaining load must be the most valuable load weighing at most $W - w$ pounds that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j.

Although the problems are similar, a greedy strategy works to solve the fractional knapsack problem, but not the 0-1 problem. To solve the fractional problem, first compute the value per pound v_i / w_i for each item. Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and the thief can still carry more, then the thief takes as much as possible of the item with the next greatest value per pound, and so forth, until reaching the weight limit W. Thus, by sorting the items by value per pound, the greedy algorithm runs in $O(n \lg n)$ time. You are asked to prove that the fractional knapsack problem has the greedy-choice property in Exercise 15.2-1.

To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illustrated in Figure 15.3(a). This example has three items and a knapsack that can hold 50 pounds. Item 1 weighs 10 pounds and is worth \$60. Item 2 weighs 20 pounds and is worth \$100. Item 3 weighs 30 pounds and is worth \$120. Thus, the value per pound of item 1 is \$6 per pound, which is greater than the value per pound of either item 2 (\$5 per pound) or item 3 (\$4 per pound). The greedy strategy, therefore, would take item 1 first. As you can see from the case analysis in Figure 15.3(b), however, the optimal solution takes items 2 and 3, leaving item 1 behind. The two possible solutions that take item 1 are both suboptimal.

For the comparable fractional problem, however, the greedy strategy, which takes item 1 first, does yield an optimal solution, as shown in Figure 15.3(c). Taking item 1 doesn't work in the 0-1 problem, because the thief is unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the load. In the 0-1 problem, when you consider whether to include an item in the knapsack, you must compare the solution to the subproblem that includes the item with the solution to the subproblem that excludes the item before you can make the choice. The problem formulated in this way gives rise to many overlapping sub-

Figure 15.3 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

problems—a hallmark of dynamic programming, and indeed, as Exercise 15.2-2 asks you to show, you can use dynamic programming to solve the 0-1 problem.

Exercises

15.2-1

Prove that the fractional knapsack problem has the greedy-choice property.

15.2-2

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(n\,W)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in the knapsack.

15.2-3

Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

15.2-4

Professor Gekko has always dreamed of inline skating across North Dakota. The professor plans to cross the state on highway U.S. 2, which runs from Grand Forks, on the eastern border with Minnesota, to Williston, near the western border with Montana. The professor can carry two liters of water and can skate m miles before

running out of water. (Because North Dakota is relatively flat, the professor does not have to worry about drinking water at a greater rate on uphill sections than on flat or downhill sections.) The professor will start in Grand Forks with two full liters of water. The professor has an official North Dakota state map, which shows all the places along U.S. 2 to refill water and the distances between these locations.

The professor's goal is to minimize the number of water stops along the route across the state. Give an efficient method by which the professor can determine which water stops to make. Prove that your strategy yields an optimal solution, and give its running time.

15.2-5

Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

★ ***15.2-6***

Show how to solve the fractional knapsack problem in $O(n)$ time.

15.2-7

You are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the ith element of set A, and let b_i be the ith element of set B. You then receive a payoff of $\prod_{i=1}^{n} a_i^{b_i}$. Give an algorithm that maximizes your payoff. Prove that your algorithm maximizes the payoff, and state its running time, omitting the time for reordering the sets.

15.3 Huffman codes

Huffman codes compress data well: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. The data arrive as a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (its frequency) to build up an optimal way of representing each character as a binary string.

Suppose that you have a 100,000-character data file that you wish to store compactly and you know that the 6 distinct characters in the file occur with the frequencies given by Figure 15.4. The character `a` occurs 45,000 times, the character `b` occurs 13,000 times, and so on.

You have many options for how to represent such a file of information. Here, we consider the problem of designing a ***binary character code*** (or ***code*** for short)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 15.4 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. With each character represented by a 3-bit codeword, encoding the file requires 300,000 bits. With the variable-length code shown, the encoding requires only 224,000 bits.

in which each character is represented by a unique binary string, which we call a ***codeword***. If you use a ***fixed-length code***, you need $\lceil \lg n \rceil$ bits to represent $n \geq 2$ characters. For 6 characters, therefore, you need 3 bits: a = 000, b = 001, c = 010, d = 011, e = 100, and f = 101. This method requires 300,000 bits to encode the entire file. Can you do better?

A ***variable-length code*** can do considerably better than a fixed-length code. The idea is simple: give frequent characters short codewords and infrequent characters long codewords. Figure 15.4 shows such a code. Here, the 1-bit string 0 represents a, and the 4-bit string 1100 represents f. This code requires

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file, as we shall see.

Prefix-free codes

We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called ***prefix-free codes***. Although we won't prove it here, a prefix-free code can always achieve the optimal data compression among any character code, and so we suffer no loss of generality by restricting our attention to prefix-free codes.

Encoding is always simple for any binary character code: just concatenate the codewords representing each character of the file. For example, with the variable-length prefix-free code of Figure 15.4, the 4-character file face has the encoding $1100 \cdot 0 \cdot 100 \cdot 1101 = 110001001101$, where "$\cdot$" denotes concatenation.

Prefix-free codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. You can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. In our example, the string 100011001101 parses uniquely as $100 \cdot 0 \cdot 1100 \cdot 1101$, which decodes to cafe.

Figure 15.5 Trees corresponding to the coding schemes in Figure 15.4. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. All frequencies are in thousands. **(a)** The tree corresponding to the fixed-length code $\mathtt{a} = 000$, $\mathtt{b} = 001$, $\mathtt{c} = 010$, $\mathtt{d} = 011$, $\mathtt{e} = 100$, $\mathtt{f} = 101$. **(b)** The tree corresponding to the optimal prefix-free code $\mathtt{a} = 0$, $\mathtt{b} = 101$, $\mathtt{c} = 100$, $\mathtt{d} = 111$, $\mathtt{e} = 1101$, $\mathtt{f} = 1100$.

The decoding process needs a convenient representation for the prefix-free code so that you can easily pick off the initial codeword. A binary tree whose leaves are the given characters provides one such representation. Interpret the binary codeword for a character as the simple path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child." Figure 15.5 shows the trees for the two codes of our example. Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.

An optimal code for a file is always represented by a *full* binary tree, in which every nonleaf node has two children (see Exercise 15.3-2). The fixed-length code in our example is not optimal since its tree, shown in Figure 15.5(a), is not a full binary tree: it contains codewords beginning with 10, but none beginning with 11. Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix-free code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C| - 1$ internal nodes (see Exercise B.5-3 on page 1175).

Given a tree T corresponding to a prefix-free code, we can compute the number of bits required to encode a file. For each character c in the alphabet C, let the attribute $c.freq$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c) \,, \tag{15.4}$$

which we define as the ***cost*** of the tree T.

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix-free code, called a ***Huffman code*** in his honor. In line with our observations in Section 15.2, its proof of correctness relies on the greedy-choice property and optimal substructure. Rather than demonstrating that these properties hold and then developing pseudocode, we present the pseudocode first. Doing so will help clarify how the algorithm makes greedy choices.

The procedure HUFFMAN assumes that C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency. The algorithm builds the tree T corresponding to an optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations to create the final tree. The algorithm uses a min-priority queue Q, keyed on the *freq* attribute, to identify the two least-frequent objects to merge together. The result of merging two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    x = EXTRACT-MIN(Q)
    y = EXTRACT-MIN(Q)
    z.left = x
    z.right = y
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

For our example, Huffman's algorithm proceeds as shown in Figure 15.6. Since the alphabet contains 6 letters, the initial queue size is $n = 6$, and 5 merge steps build the tree. The final tree represents the optimal prefix-free code. The codeword for a letter is the sequence of edge labels on the simple path from the root to the letter.

Figure 15.6 The steps of Huffman's algorithm for the frequencies given in Figure 15.4. Each part shows the contents of the queue sorted into increasing order by frequency. Each step merges the two trees with the lowest frequencies. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of their children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. **(a)** The initial set of $n = 6$ nodes, one for each letter. **(b)–(e)** Intermediate stages. **(f)** The final tree.

The HUFFMAN procedure works as follows. Line 2 initializes the min-priority queue Q with the characters in C. The **for** loop in lines 3–10 repeatedly extracts the two nodes x and y of lowest frequency from the queue and replaces them in the queue with a new node z representing their merger. The frequency of z is computed as the sum of the frequencies of x and y in line 9. The node z has x as its left child and y as its right child. (This order is arbitrary. Switching the left and right child of any node yields a different code of the same cost.) After $n - 1$ mergers, line 11 returns the one node left in the queue, which is the root of the code tree.

The algorithm produces the same result without the variables x and y, assigning the values returned by the EXTRACT-MIN calls directly to $z.left$ and $z.right$ in lines 7 and 8, and changing line 9 to $z.freq = z.left.freq + z.right.freq$. We'll use the node names x and y in the proof of correctness, however, so we leave them in.

The running time of Huffman's algorithm depends on how the min-priority queue Q is implemented. Let's assume that it's implemented as a binary min-heap (see Chapter 6). For a set C of n characters, the BUILD-MIN-HEAP procedure discussed in Section 6.3 can initialize Q in line 2 in $O(n)$ time. The **for** loop in lines 3–10 executes exactly $n - 1$ times, and since each heap operation runs in $O(\lg n)$ time, the loop contributes $O(n \lg n)$ to the running time. Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$.

Correctness of Huffman's algorithm

To prove that the greedy algorithm HUFFMAN is correct, we'll show that the problem of determining an optimal prefix-free code exhibits the greedy-choice and optimal-substructure properties. The next lemma shows that the greedy-choice property holds.

Lemma 15.2 (Optimal prefix-free codes have the greedy-choice property)
Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix-free code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof The idea of the proof is to take the tree T representing an arbitrary optimal prefix-free code and modify it to make a tree representing another optimal prefix-free code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. In such a tree, the codewords for x and y have the same length and differ only in the last bit.

Let a and b be any two characters that are sibling leaves of maximum depth in T. Without loss of generality, assume that $a.freq \le b.freq$ and $x.freq \le y.freq$. Since $x.freq$ and $y.freq$ are the two lowest leaf frequencies, in order, and $a.freq$ and $b.freq$ are two arbitrary frequencies, in order, we have $x.freq \le a.freq$ and $y.freq \le b.freq$.

In the remainder of the proof, it is possible that we could have $x.freq = a.freq$ or $y.freq = b.freq$, but $x.freq = b.freq$ implies that $a.freq = b.freq = x.freq = y.freq$ (see Exercise 15.3-1), and the lemma would be trivially true. Therefore, assume that $x.freq \ne b.freq$, which means that $x \ne b$.

As Figure 15.7 shows, imagine exchanging the positions in T of a and x to produce a tree T', and then exchanging the positions in T' of b and y to produce a

Figure 15.7 An illustration of the key step in the proof of Lemma 15.2. In the optimal tree T, leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the lowest frequencies. They appear in arbitrary positions in T. Assuming that $x \neq b$, swapping leaves a and x produces tree T', and then swapping leaves b and y produces tree T''. Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.

tree T'' in which x and y are sibling leaves of maximum depth. (Note that if $x = b$ but $y \neq a$, then tree T'' does not have x and y as sibling leaves of maximum depth. Because we assume that $x \neq b$, this situation cannot occur.) By equation (15.4), the difference in cost between T and T' is

$$\begin{aligned}
&B(T) - B(T') \\
&\quad = \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&\quad = (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\quad \geq 0 ,
\end{aligned}$$

because both $a.freq - x.freq$ and $d_T(a) - d_T(x)$ are nonnegative. More specifically, $a.freq - x.freq$ is nonnegative because x is a minimum-frequency leaf, and $d_T(a) - d_T(x)$ is nonnegative because a is a leaf of maximum depth in T. Similarly, exchanging y and b does not increase the cost, and so $B(T') - B(T'')$ is nonnegative. Therefore, $B(T'') \leq B(T') \leq B(T)$, and since T is optimal, we have $B(T) \leq B(T'')$, which implies $B(T'') = B(T)$. Thus, T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth, from which the lemma follows. ■

Lemma 15.2 implies that the process of building up an optimal tree by mergers can, without loss of generality, begin with the greedy choice of merging together those two characters of lowest frequency. Why is this a greedy choice? We can view the cost of a single merger as being the sum of the frequencies of the two items being merged. Exercise 15.3-4 shows that the total cost of the tree constructed equals the sum of the costs of its mergers. Of all possible mergers at each step, HUFFMAN chooses the one that incurs the least cost.

The next lemma shows that the problem of constructing optimal prefix-free codes has the optimal-substructure property.

Lemma 15.3 (Optimal prefix-free codes have the optimal-substructure property)
Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = (C - \{x, y\}) \cup \{z\}$. Define *freq* for all characters in C' with the same values as in C, along with $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix-free code for alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix-free code for the alphabet C.

Proof We first show how to express the cost $B(T)$ of tree T in terms of the cost $B(T')$ of tree T', by considering the component costs in equation (15.4). For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and hence $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) , \end{aligned}$$

from which we conclude that

$$B(T) = B(T') + x.freq + y.freq$$

or, equivalently,

$$B(T') = B(T) - x.freq - y.freq .$$

We now prove the lemma by contradiction. Suppose that T does not represent an optimal prefix-free code for C. Then there exists an optimal tree T'' such that $B(T'') < B(T)$. Without loss of generality (by Lemma 15.2), T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $z.freq = x.freq + y.freq$. Then

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) \ - x.freq - y.freq \\ &= B(T') , \end{aligned}$$

yielding a contradiction to the assumption that T' represents an optimal prefix-free code for C'. Thus, T must represent an optimal prefix-free code for the alphabet C. ■

Theorem 15.4
Procedure HUFFMAN produces an optimal prefix-free code.

Proof Immediate from Lemmas 15.2 and 15.3. ■

Exercises

15.3-1
Explain why, in the proof of Lemma 15.2, if $x.freq = b.freq$, then we must have $a.freq = b.freq = x.freq = y.freq$.

15.3-2
Prove that a non-full binary tree cannot correspond to an optimal prefix-free code.

15.3-3
What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

`a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21`

Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

15.3-4
Prove that the total cost $B(T)$ of a full binary tree T for a code equals the sum, over all internal nodes, of the combined frequencies of the two children of the node.

15.3-5
Given an optimal prefix-free code on a set C of n characters, you wish to transmit the code itself using as few bits as possible. Show how to represent any optimal prefix-free code on C using only $2n - 1 + n \lceil \lg n \rceil$ bits. (*Hint:* Use $2n - 1$ bits to specify the structure of the tree, as discovered by a walk of the tree.)

15.3-6
Generalize Huffman's algorithm to ternary codewords (i.e., codewords using the symbols 0, 1, and 2), and prove that it yields optimal ternary codes.

15.3-7
A data file contains a sequence of 8-bit characters such that all 256 characters are about equally common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

15.3-8
Show that no lossless (invertible) compression scheme can guarantee that for every input file, the corresponding output file is shorter. (*Hint:* Compare the number of possible files with the number of possible encoded files.)

15.4 Offline caching

Computer systems can decrease the time to access data by storing a subset of the main memory in the ***cache***: a small but faster memory. A cache organizes data into ***cache blocks*** typically comprising 32, 64, or 128 bytes. You can also think of main memory as a cache for disk-resident data in a virtual-memory system. Here, the blocks are called ***pages***, and 4096 bytes is a typical size.

As a computer program executes, it makes a sequence of memory requests. Say that there are n memory requests, to data in blocks $b_1, b_2, \ldots, b_n$, in that order. The blocks in the access sequence might not be distinct, and indeed, any given block is usually accessed multiple times. For example, a program that accesses four distinct blocks p, q, r, s might make a sequence of requests to blocks $s, q, s, q, q, s, p, p, r, s, s, q, p, r, q$. The cache can hold up to some fixed number k of cache blocks. It starts out empty before the first request. Each request causes at most one block to enter the cache and at most one block to be evicted from the cache. Upon a request for block b_i, any one of three scenarios may occur:

1. Block b_i is already in the cache, due to a previous request for the same block. The cache remains unchanged. This situation is known as a ***cache hit***.
2. Block b_i is not in the cache at that time, but the cache contains fewer than k blocks. In this case, block b_i is placed into the cache, so that the cache contains one more block than it did before the request.
3. Block b_i is not in the cache at that time and the cache is full: it contains k blocks. Block b_i is placed into the cache, but before that happens, some other block in the cache must be evicted from the cache in order to make room.

The latter two situations, in which the requested block is not already in the cache, are called ***cache misses***. The goal is to minimize the number of cache misses or, equivalently, to maximize the number of cache hits, over the entire sequence of n requests. A cache miss that occurs while the cache holds fewer than k blocks—that is, as the cache is first being filled up—is known as a ***compulsory miss***, since no prior decision could have kept the requested block in the cache. When a cache miss occurs and the cache is full, ideally the choice of which block to evict should allow for the smallest possible number of cache misses over the entire sequence of future requests.

Typically, caching is an online problem. That is, the computer has to decide which blocks to keep in the cache without knowing the future requests. Here, however, let's consider the offline version of this problem, in which the computer knows in advance the entire sequence of n requests and the cache size k, with a goal of minimizing the total number of cache misses.

To solve this offline problem, you can use a greedy strategy called ***furthest-in-future***, which chooses to evict the block in the cache whose next access in the request sequence comes furthest in the future. Intuitively, this strategy makes sense: if you're not going to need something for a while, why keep it around? We'll show that the furthest-in-future strategy is indeed optimal by showing that the offline caching problem exhibits optimal substructure and that furthest-in-future has the greedy-choice property.

Now, you might be thinking that since the computer usually doesn't know the sequence of requests in advance, there is no point in studying the offline problem. Actually, there is. In some situations, you do know the sequence of requests in advance. For example, if you view the main memory as the cache and the full set of data as residing on disk (or a solid-state drive), there are algorithms that plan out the entire set of reads and writes in advance. Furthermore, we can use the number of cache misses produced by an optimal algorithm as a baseline for comparing how well online algorithms perform. We'll do just that in Section 27.3.

Offline caching can even model real-world problems. For example, consider a scenario where you know in advance a fixed schedule of n events at known locations. Events may occur at a location multiple times, not necessarily consecutively. You are managing a group of k agents, you need to ensure that you have one agent at each location when an event occurs, and you want to minimize the number of times that agents have to move. Here, the agents are like the blocks, the events are like the requests, and moving an agent is akin to a cache miss.

Optimal substructure of offline caching

To show that the offline problem exhibits optimal substructure, let's define the subproblem (C, i) as processing requests for blocks $b_i, b_{i+1}, \ldots, b_n$ with cache configuration C at the time that the request for block b_i occurs, that is, C is a subset of the set of blocks such that $|C| \le k$. A solution to subproblem (C, i) is a sequence of decisions that specifies which block to evict (if any) upon each request for blocks $b_i, b_{i+1}, \ldots, b_n$. An optimal solution to subproblem (C, i) minimizes the number of cache misses.

Consider an optimal solution S to subproblem (C, i), and let C' be the contents of the cache after processing the request for block b_i in solution S. Let S' be the subsolution of S for the resulting subproblem $(C', i + 1)$. If the request for b_i results in a cache hit, then the cache remains unchanged, so that $C' = C$. If the request for block b_i results in a cache miss, then the contents of the cache change, so that $C' \ne C$. We claim that in either case, S' is an optimal solution to subproblem $(C', i + 1)$. Why? If S' is not an optimal solution to subproblem $(C', i + 1)$, then there exists another solution S'' to subproblem $(C', i + 1)$ that makes fewer cache misses than S'. Combining S'' with the decision of S at the request for

block b_i yields another solution that makes fewer cache misses than S, which contradicts the assumption that S is an optimal solution to subproblem (C, i).

To quantify a recursive solution, we need a little more notation. Let $R_{C,i}$ be the set of all cache configurations that can immediately follow configuration C after processing a request for block b_i. If the request results in a cache hit, then the cache remains unchanged, so that $R_{C,i} = \{C\}$. If the request for b_i results in a cache miss, then there are two possibilities. If the cache is not full ($|C| < k$), then the cache is filling up and the only choice is to insert b_i into the cache, so that $R_{C,i} = \{C \cup \{b_i\}\}$. If the cache is full ($|C| = k$) upon a cache miss, then $R_{C,i}$ contains k potential configurations: one for each candidate block in C that could be evicted and replaced by block b_i. In this case, $R_{C,i} = \{(C - \{x\}) \cup \{b_i\} : x \in C\}$. For example, if $C = \{p, q, r\}$, $k = 3$, and block s is requested, then $R_{C,i} = \{\{p, q, s\}, \{p, r, s\}, \{q, r, s\}\}$.

Let $miss(C, i)$ denote the minimum number of cache misses in a solution for subproblem (C, i). Here is a recurrence for $miss(C, i)$:

$$miss(C, i) = \begin{cases} 0 & \text{if } i = n \text{ and } b_n \in C \ , \\ 1 & \text{if } i = n \text{ and } b_n \notin C \ , \\ miss(C, i + 1) & \text{if } i < n \text{ and } b_i \in C \ , \\ 1 + \min\{miss(C', i + 1) : C' \in R_{C,i}\} & \text{if } i < n \text{ and } b_i \notin C \ . \end{cases}$$

Greedy-choice property

To prove that the furthest-in-future strategy yields an optimal solution, we need to show that optimal offline caching exhibits the greedy-choice property. Combined with the optimal-substructure property, the greedy-choice property will prove that furthest-in-future produces the minimum possible number of cache misses.

Theorem 15.5 (Optimal offline caching has the greedy-choice property)
Consider a subproblem (C, i) when the cache C contains k blocks, so that it is full, and a cache miss occurs. When block b_i is requested, let $z = b_m$ be the block in C whose next access is furthest in the future. (If some block in the cache will never again be referenced, then consider any such block to be block z, and add a dummy request for block $z = b_m = b_{n+1}$.) Then evicting block z upon a request for block b_i is included in some optimal solution for the subproblem (C, i).

Proof Let S be an optimal solution to subproblem (C, i). If S evicts block z upon the request for block b_i, then we are done, since we have shown that some optimal solution includes evicting z.

So now suppose that optimal solution S evicts some other block x when block b_i is requested. We'll construct another solution S' to subproblem (C, i) which, upon

the request for b_i, evicts block z instead of x and induces no more cache misses than S does, so that S' is also optimal. Because different solutions may yield different cache configurations, denote by $C_{S,j}$ the configuration of the cache under solution S just before the request for some block b_j, and likewise for solution S' and $C_{S',j}$. We'll show how to construct S' with the following properties:

1. For $j = i+1, \ldots, m$, let $D_j = C_{S,j} \cap C_{S',j}$. Then, $|D_j| \geq k-1$, so that the cache configurations $C_{S,j}$ and $C_{S',j}$ differ by at most one block. If they differ, then $C_{S,j} = D_j \cup \{z\}$ and $C_{S',j} = D_j \cup \{y\}$ for some block $y \neq z$.
2. For each request of blocks $b_i, \ldots, b_{m-1}$, if solution S has a cache hit, then solution S' also has a cache hit.
3. For all $j > m$, the cache configurations $C_{S,j}$ and $C_{S',j}$ are identical.
4. Over the sequence of requests for blocks $b_i, \ldots, b_m$, the number of cache misses produced by solution S' is at most the number of cache misses produced by solution S.

We'll prove inductively that these properties hold for each request.

1. We proceed by induction on j, for $j = i+1, \ldots, m$. For the base case, the initial caches $C_{S,i}$ and $C_{S',i}$ are identical. Upon the request for block b_i, solution S evicts x and solution S' evicts z. Thus, cache configurations $C_{S,i+1}$ and $C_{S',i+1}$ differ by just one block, $C_{S,i+1} = D_{i+1} \cup \{z\}$, $C_{S',i+1} = D_{i+1} \cup \{x\}$, and $x \neq z$.

   The inductive step defines how solution S' behaves upon a request for block b_j for $i+1 \leq j \leq m-1$. The inductive hypothesis is that property 1 holds when b_j is requested. Because $z = b_m$ is the block in $C_{S,i}$ whose next reference is furthest in the future, we know that $b_j \neq z$. We consider several scenarios:

   - If $C_{S,j} = C_{S',j}$ (so that $|D_j| = k$), then solution S' makes the same decision upon the request for b_j as S makes, so that $C_{S,j+1} = C_{S',j+1}$.
   - If $|D_j| = k-1$ and $b_j \in D_j$, then both caches already contain block b_j, and both solutions S and S' have cache hits. Therefore, $C_{S,j+1} = C_{S,j}$ and $C_{S',j+1} = C_{S',j}$.
   - If $|D_j| = k-1$ and $b_j \notin D_j$, then because $C_{S,j} = D_j \cup \{z\}$ and $b_j \neq z$, solution S has a cache miss. It evicts either block z or some block $w \in D_j$.
     - If solution S evicts block z, then $C_{S,j+1} = D_j \cup \{b_j\}$. There are two cases, depending on whether $b_j = y$:
       - If $b_j = y$, then solution S' has a cache hit, so that $C_{S',j+1} = C_{S',j} = D_j \cup \{b_j\}$. Thus, $C_{S,j+1} = C_{S',j+1}$.
       - If $b_j \neq y$, then solution S' has a cache miss. It evicts block y, so that $C_{S',j+1} = D_j \cup \{b_j\}$, and again $C_{S,j+1} = C_{S',j+1}$.

- If solution S evicts some block $w \in D_j$, then $C_{S,j+1} = (D_j - \{w\}) \cup \{b_j, z\}$. Once again, there are two cases, depending on whether $b_j = y$:
  - If $b_j = y$, then solution S' has a cache hit, so that $C_{S',j+1} = C_{S',j} = D_j \cup \{b_j\}$. Since $w \in D_j$ and w was not evicted by solution S', we have $w \in C_{S',j+1}$. Therefore, $w \notin D_{j+1}$ and $b_j \in D_{j+1}$, so that $D_{j+1} = (D_j - \{w\}) \cup \{b_j\}$. Thus, $C_{S,j+1} = D_{j+1} \cup \{z\}$, $C_{S',j+1} = D_{j+1} \cup \{w\}$, and because $w \neq z$, property 1 holds when block b_{j+1} is requested. (In other words, block w replaces block y in property 1.)
  - If $b_j \neq y$, then solution S' has a cache miss. It evicts block w, so that $C_{S',j+1} = (D_j - \{w\}) \cup \{b_j, y\}$. Therefore, we have that $D_{j+1} = (D_j - \{w\}) \cup \{b_j\}$ and so $C_{S,j+1} = D_{j+1} \cup \{z\}$ and $C_{S',j+1} = D_{j+1} \cup \{y\}$.

2. In the above discussion about maintaining property 1, solution S may have a cache hit in only the first two cases, and solution S' has a cache hit in these cases if and only if S does.

3. If $C_{S,m} = C_{S',m}$, then solution S' makes the same decision upon the request for block $z = b_m$ as S makes, so that $C_{S,m+1} = C_{S',m+1}$. If $C_{S,m} \neq C_{S',m}$, then by property 1, $C_{S,m} = D_m \cup \{z\}$ and $C_{S',m} = D_m \cup \{y\}$, where $y \neq z$. In this case, solution S has a cache hit, so that $C_{S,m+1} = C_{S,m} = D_m \cup \{z\}$. Solution S' evicts block y and brings in block z, so that $C_{S',m+1} = D_m \cup \{z\} = C_{S,m+1}$. Thus, regardless of whether or not $C_{S,m} = C_{S',m}$, we have $C_{S,m+1} = C_{S',m+1}$, and starting with the request for block b_{m+1}, solution S' simply makes the same decisions as S.

4. By property 2, upon the requests for blocks $b_i, \ldots, b_{m-1}$, whenever solution S has a cache hit, so does S'. Only the request for block $b_m = z$ remains to be considered. If S has a cache miss upon the request for b_m, then regardless of whether S' has a cache hit or a cache miss, we are done: S' has at most the same number of cache misses as S.

   So now suppose that S has a cache hit and S' has a cache miss upon the request for b_m. We'll show that there exists a request for at least one of blocks $b_{i+1}, \ldots, b_{m-1}$ in which the request results in a cache miss for S and a cache hit for S', thereby compensating for what happens upon the request for block b_m. The proof is by contradiction. Assume that no request for blocks $b_{i+1}, \ldots, b_{m-1}$ results in a cache miss for S and a cache hit for S'.

   We start by observing that once the caches $C_{S,j}$ and $C_{S'j}$ are equal for some $j > i$, they remain equal thereafter. Observe also that if $b_m \in C_{S,m}$ and $b_m \notin C_{S',m}$, then $C_{S,m} \neq C_{S',m}$. Therefore, solution S cannot have evicted block z upon the requests for blocks $b_i, \ldots, b_{m-1}$, for if it had, then these two

cache configurations would be equal. The remaining possibility is that upon each of these requests, we had $C_{S,j} = D_j \cup \{z\}$, $C_{S',j} = D_j \cup \{y\}$ for some block $y \neq z$, and solution S evicted some block $w \in D_j$. Moreover, since none of these requests resulted in a cache miss for S and a cache hit for S', the case of $b_j = y$ never occurred. That is, for every request of blocks $b_{i+1}, \ldots, b_{m-1}$, the requested block b_j was never the block $y \in C_{S',j} - C_{S,j}$. In these cases, after processing the request, we had $C_{S',j+1} = D_{j+1} \cup \{y\}$: the difference between the two caches did not change. Now, let's go back to the request for block b_i, where afterward, we had $C_{S',i+1} = D_{i+1} \cup \{x\}$. Because every succeeding request until requesting block b_m did not change the difference between the caches, we had $C_{S',j} = D_j \cup \{x\}$ for $j = i+1, \ldots, m$.

By definition, block $z = b_m$ is requested after block x. That means at least one of blocks $b_{i+1}, \ldots, b_{m-1}$ is block x. But for $j = i+1, \ldots, m$, we have $x \in C_{S',j}$ and $x \notin C_{S,j}$, so that at least one of these requests had a cache hit for S' and a cache miss for S, a contradiction. We conclude that if solution S has a cache hit and solution S' has a cache miss upon the request for block b_m, then some earlier request had the opposite result, and so solution S' produces no more cache misses than solution S. Since S is assumed to be optimal, S' is optimal as well. ■

Along with the optimal-substructure property, Theorem 15.5 tells us that the furthest-in-future strategy yields the minimum number of cache misses.

Exercises

15.4-1

Write pseudocode for a cache manager that uses the furthest-in-future strategy. It should take as input a set C of blocks in the cache, the number of blocks k that the cache can hold, a sequence $b_1, b_2, \ldots, b_n$ of requested blocks, and the index i into the sequence for the block b_i being requested. For each request, it should print out whether a cache hit or cache miss occurs, and for each cache miss, it should also print out which block, if any, is evicted.

15.4-2

Real cache managers do not know the future requests, and so they often use the past to decide which block to evict. The ***least-recently-used***, or ***LRU***, strategy evicts the block that, of all blocks currently in the cache, was the least recently requested. (You can think of LRU as "furthest-in-past.") Give an example of a request sequence in which the LRU strategy is not optimal, by showing that it induces more cache misses than the furthest-in-future strategy does on the same request sequence.

15.4-3

Professor Croesus suggests that in the proof of Theorem 15.5, the last clause in property 1 can change to $C_{S',j} = D_j \cup \{x\}$ or, equivalently, require the block y given in property 1 to always be the block x evicted by solution S upon the request for block b_i. Show where the proof breaks down with this requirement.

15.4-4

This section has assumed that at most one block is placed into the cache whenever a block is requested. You can imagine, however, a strategy in which multiple blocks may enter the cache upon a single request. Show that for every solution that allows multiple blocks to enter the cache upon each request, there is another solution that brings in only one block upon each request and is at least as good.

Problems

15-1 Coin changing

Consider the problem of making change for n cents using the smallest number of coins. Assume that each coin's value is an integer.

a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

b. Suppose that the available coins are in denominations that are powers of c: the denominations are $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

d. Give an $O(nk)$-time algorithm that makes change for any set of k different coin denominations using the smallest number of coins, assuming that one of the coins is a penny.

15-2 Scheduling to minimize average completion time

You are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete. Let C_i be the ***completion time*** of task a_i, that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $(1/n)\sum_{i=1}^{n} C_i$. For example, suppose that there are two tasks a_1 and a_2 with $p_1 = 3$ and $p_2 = 5$, and consider the schedule

in which a_2 runs first, followed by a_1. Then we have $C_2 = 5$, $C_1 = 8$, and the average completion time is $(5+8)/2 = 6.5$. If task a_1 runs first, however, then we have $C_1 = 3$, $C_2 = 8$, and the average completion time is $(3+8)/2 = 5.5$.

a. Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run nonpreemptively, that is, once task a_i starts, it must run continuously for p_i units of time until it is done. Prove that your algorithm minimizes the average completion time, and analyze the running time of your algorithm.

b. Suppose now that the tasks are not all available at once. That is, each task cannot start until its ***release time*** b_i. Suppose also that tasks may be ***preempted***, so that a task can be suspended and restarted at a later time. For example, a task a_i with processing time $p_i = 6$ and release time $b_i = 1$ might start running at time 1 and be preempted at time 4. It might then resume at time 10 but be preempted at time 11, and it might finally resume at time 13 and complete at time 15. Task a_i has run for a total of 6 time units, but its running time has been divided into three pieces. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and analyze the running time of your algorithm.

Chapter notes

Much more material on greedy algorithms can be found in Lawler [276] and Papadimitriou and Steiglitz [353]. The greedy algorithm first appeared in the combinatorial optimization literature in a 1971 article by Edmonds [131].

The proof of correctness of the greedy algorithm for the activity-selection problem is based on that of Gavril [179].

Huffman codes were invented in 1952 [233]. Lelewer and Hirschberg [294] surveys data-compression techniques known as of 1987.

The furthest-in-future strategy was proposed by Belady [41], who suggested it for virtual-memory systems. Alternative proofs that furthest-in-future is optimal appear in articles by Lee et al. [284] and Van Roy [443].

16 Amortized Analysis

Imagine that you join Buff's Gym. Buff charges a membership fee of \$60 per month, plus \$3 for every time you use the gym. Because you are disciplined, you visit Buff's Gym every day during the month of November. On top of the \$60 monthly charge for November, you pay another $3 \times \$30 = \$90$ that month. Although you can think of your fees as a flat fee of \$60 and another \$90 in daily fees, you can think about it in another way. All together, you pay \$150 over 30 days, or an average of \$5 per day. When you look at your fees in this way, you are ***amortizing*** the monthly fee over the 30 days of the month, spreading it out at \$2 per day.

You can do the same thing when you analyze running times. In an ***amortized analysis***, you average the time required to perform a sequence of data-structure operations over all the operations performed. With amortized analysis, you show that if you average over a sequence of operations, then the average cost of an operation is small, even though a single operation within the sequence might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved. An amortized analysis guarantees the *average performance of each operation in the worst case*.

The first three sections of this chapter cover the three most common techniques used in amortized analysis. Section 16.1 starts with aggregate analysis, in which you determine an upper bound $T(n)$ on the total cost of a sequence of n operations. The average cost per operation is then $T(n)/n$. You take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.

Section 16.2 covers the accounting method, in which you determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method overcharges some operations early in the sequence, storing the overcharge as "pre-

paid credit" on specific objects in the data structure. Later in the sequence, the credit pays for operations that are charged less than they actually cost.

Section 16.3 discusses the potential method, which is like the accounting method in that you determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method maintains the credit as the "potential energy" of the data structure as a whole instead of associating the credit with individual objects within the data structure.

We'll use use two examples in this chapter to examine each of these three methods. One is a stack with the additional operation MULTIPOP, which pops several objects at once. The other is a binary counter that counts up from 0 by means of the single operation INCREMENT.

While reading this chapter, bear in mind that the charges assigned during an amortized analysis are for analysis purposes only. They need not—and should not—appear in the code. If, for example, you assign a credit to an object x when using the accounting method, you have no need to assign an appropriate amount to some attribute, such as $x.credit$, in the code.

When you perform an amortized analysis, you often gain insight into a particular data structure, and this insight can help you optimize the design. For example, Section 16.4 will use the potential method to analyze a dynamically expanding and contracting table.

16.1 Aggregate analysis

In ***aggregate analysis***, you show that for all n, a sequence of n operations takes $T(n)$ *worst-case* time in total. In the worst case, the average cost, or ***amortized cost***, per operation is therefore $T(n)/n$. This amortized cost applies to each operation, even when there are several types of operations in the sequence. The other two methods we shall study in this chapter, the accounting method and the potential method, may assign different amortized costs to different types of operations.

Stack operations

As the first example of aggregate analysis, let's analyze stacks that have been augmented with a new operation. Section 10.1.3 presented the two fundamental stack operations, each of which takes $O(1)$ time:

PUSH(S, x) pushes object x onto stack S.

POP(S) pops the top of stack S and returns the popped object. Calling POP on an empty stack generates an error.

Figure 16.1 The action of MULTIPOP on a stack S, shown initially in **(a)**. The top 4 objects are popped by MULTIPOP$(S, 4)$, whose result is shown in **(b)**. The next operation is MULTIPOP$(S, 7)$, which empties the stack—shown in **(c)**—since fewer than 7 objects remained.

Since each of these operations runs in $O(1)$ time, let us consider the cost of each to be 1. The total cost of a sequence of n PUSH and POP operations is therefore n, and the actual running time for n operations is therefore $\Theta(n)$.

Now let's add the stack operation MULTIPOP(S, k), which removes the k top objects of stack S, popping the entire stack if the stack contains fewer than k objects. Of course, the procedure assumes that k is positive, and otherwise, the MULTIPOP operation leaves the stack unchanged. In the pseudocode for MULTIPOP, the operation STACK-EMPTY returns TRUE if there are no objects currently on the stack, and FALSE otherwise. Figure 16.1 shows an example of MULTIPOP.

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2      POP(S)
3      k = k - 1
```

What is the running time of MULTIPOP(S, k) on a stack of s objects? The actual running time is linear in the number of POP operations actually executed, and thus we can analyze MULTIPOP in terms of the abstract costs of 1 each for PUSH and POP. The number of iterations of the **while** loop is the number $\min\{s, k\}$ of objects popped off the stack. Each iteration of the loop makes one call to POP in line 2. Thus, the total cost of MULTIPOP is $\min\{s, k\}$, and the actual running time is a linear function of this cost.

Now let's analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack. The worst-case cost of a MULTIPOP operation in the sequence is $O(n)$, since the stack size is at most n. The worst-case time of any stack operation is therefore $O(n)$, and hence a sequence of n operations costs $O(n^2)$, since the sequence contains at most n MULTIPOP operations costing $O(n)$ each.

Although this analysis is correct, the $O(n^2)$ result, which came from considering the worst-case cost of each operation individually, is not tight.

Yes, a single MULTIPOP might be expensive, but an aggregate analysis shows that any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack has an upper bound on its cost of $O(n)$. Why? An object cannot be popped from the stack unless it was first pushed. Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most n. For any value of n, any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time. Averaging over the n operations gives an average cost per operation of $O(n)/n = O(1)$. Aggregate analysis assigns the amortized cost of each operation to be the average cost. In this example, therefore, all three stack operations have an amortized cost of $O(1)$.

To recap: although the average cost, and hence the running time, of a stack operation is $O(1)$, the analysis did not rely on probabilistic reasoning. Instead, the analysis yielded a *worst-case* bound of $O(n)$ on a sequence of n operations. Dividing this total cost by n yielded that the average cost per operation—that is, the amortized cost—is $O(1)$.

Incrementing a binary counter

As another example of aggregate analysis, consider the problem of implementing a k-bit binary counter that counts upward from 0. An array $A[0:k-1]$ of bits represents the counter. A binary number x that is stored in the counter has its lowest-order bit in $A[0]$ and its highest-order bit in $A[k-1]$, so that $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Initially, $x = 0$, and thus $A[i] = 0$ for $i = 0, 1, \ldots, k-1$. To add 1 (modulo 2^k) to the value in the counter, call the INCREMENT procedure.

```
INCREMENT(A, k)
1  i = 0
2  while i < k and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < k
6      A[i] = 1
```

Figure 16.2 shows what happens to a binary counter when INCREMENT is called 16 times, starting with the initial value 0 and ending with the value 16. Each iteration of the **while** loop in lines 2–4 adds a 1 into position i. If $A[i] = 1$, then adding 1 flips the bit to 0 in position i and yields a carry of 1, to be added into

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 16.2 An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded in blue. The running cost for flipping bits is shown at the right. The total cost is always less than twice the total number of INCREMENT operations.

position $i + 1$ during the next iteration of the loop. Otherwise, the loop ends, and then, if $i < k$, $A[i]$ must be 0, so that line 6 adds a 1 into position i, flipping the 0 to a 1. If the loop ends with $i = k$, then the call of INCREMENT flipped all k bits from 1 to 0. The cost of each INCREMENT operation is linear in the number of bits flipped.

As with the stack example, a cursory analysis yields a bound that is correct but not tight. A single execution of INCREMENT takes $\Theta(k)$ time in the worst case, in which all the bits in array A are 1. Thus, a sequence of n INCREMENT operations on an initially zero counter takes $O(nk)$ time in the worst case.

Although a single call of INCREMENT might flip all k bits, not all bits flip upon each call. (Note the similarity to MULTIPOP, where a single call might pop many objects, but not every call pops many objects.) As Figure 16.2 shows, $A[0]$ does flip each time INCREMENT is called. The next bit up, $A[1]$, flips only every other time: a sequence of n INCREMENT operations on an initially zero counter causes $A[1]$ to flip $\lfloor n/2 \rfloor$ times. Similarly, bit $A[2]$ flips only every fourth time, or $\lfloor n/4 \rfloor$ times in a sequence of n INCREMENT operations. In general, for $i = 0, 1, \ldots, k-1$, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of n INCREMENT operations on an initially zero counter. For $i \geq k$, bit $A[i]$ does not exist, and so it cannot flip. The total number

of flips in the sequence is thus

$$\begin{aligned}\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor &< n \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= 2n ,\end{aligned}$$

by equation (A.7) on page 1142. Thus, a sequence of n INCREMENT operations on an initially zero counter takes $O(n)$ time in the worst case. The average cost of each operation, and therefore the amortized cost per operation, is $O(n)/n = O(1)$.

Exercises

16.1-1
If the set of stack operations includes a MULTIPUSH operation, which pushes k items onto the stack, does the $O(1)$ bound on the amortized cost of stack operations continue to hold?

16.1-2
Show that if a DECREMENT operation is included in the k-bit counter example, n operations can cost as much as $\Theta(nk)$ time.

16.1-3
Use aggregate analysis to determine the amortized cost per operation for a sequence of n operations on a data structure in which the ith operation costs i if i is an exact power of 2, and 1 otherwise.

16.2 The accounting method

In the ***accounting method*** of amortized analysis, you assign differing charges to different operations, with some operations charged more or less than they actually cost. The amount that you charge an operation is its ***amortized cost***. When an operation's amortized cost exceeds its actual cost, you assign the difference to specific objects in the data structure as ***credit***. Credit can help pay for later operations whose amortized cost is less than their actual cost. Thus, you can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up. Different operations may have different amortized costs. This method differs from aggregate analysis, in which all operations have the same amortized cost.

You must choose the amortized costs of operations carefully. If you want to use amortized costs to show that in the worst case the average cost per operation is

small, you must ensure that the total amortized cost of a sequence of operations provides an upper bound on the total actual cost of the sequence. Moreover, as in aggregate analysis, the upper bound must apply to all sequences of operations. Let's denote the actual cost of the ith operation by c_i and the amortized cost of the ith operation by $\widehat{c}_i$. Then you need to have

$$\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i \tag{16.1}$$

for all sequences of n operations. The total credit stored in the data structure is the difference between the total amortized cost and the total actual cost, or $\sum_{i=1}^{n} \widehat{c}_i - \sum_{i=1}^{n} c_i$. By inequality (16.1), the total credit associated with the data structure must be nonnegative at all times. If you ever allowed the total credit to become negative (the result of undercharging early operations with the promise of repaying the account later on), then the total amortized costs incurred at that time would be below the total actual costs incurred. In that case, for the sequence of operations up to that time, the total amortized cost would not be an upper bound on the total actual cost. Thus, you must take care that the total credit in the data structure never becomes negative.

Stack operations

To illustrate the accounting method of amortized analysis, we return to the stack example. Recall that the actual costs of the operations were

PUSH	1 ,
POP	1 ,
MULTIPOP	$\min\{s, k\}$,

where k is the argument supplied to MULTIPOP and s is the stack size when it is called. Let us assign the following amortized costs:

PUSH	2 ,
POP	0 ,
MULTIPOP	0 .

The amortized cost of MULTIPOP is a constant (0), whereas the actual cost is variable, and thus all three amortized costs are constant. In general, the amortized costs of the operations under consideration may differ from each other, and they may even differ asymptotically.

Now let's see how to pay for any sequence of stack operations by charging the amortized costs. Let \$1 represent each unit of cost. At first, the stack is empty. Recall the analogy of Section 10.1.3 between the stack data structure and a stack of plates in a cafeteria. Upon pushing a plate onto the stack, use \$1 to pay the

actual cost of the push, leaving a credit of \$1 (out of the \$2 charged). Place that \$1 of credit on top of the plate. At any point in time, every plate on the stack has \$1 of credit on it.

The \$1 stored on the plate serves to prepay the cost of popping the plate from the stack. A POP operation incurs no charge: pay the actual cost of popping a plate by taking the \$1 of credit off the plate. Thus, by charging the PUSH operation a little bit more, we can view the POP operation as free.

Moreover, the MULTIPOP operation also incurs no charge, since it's just repeated POP operations, each of which is free. If a MULTIPOP operation pops k plates, then the actual cost is paid by the k dollars stored on the k plates. Because each plate on the stack has \$1 of credit on it, and the stack always has a nonnegative number of plates, the amount of credit is always nonnegative. Thus, for *any* sequence of n PUSH, POP, and MULTIPOP operations, the total amortized cost is an upper bound on the total actual cost. Since the total amortized cost is $O(n)$, so is the total actual cost.

Incrementing a binary counter

As another illustration of the accounting method, let's analyze the INCREMENT operation on a binary counter that starts at 0. Recall that the running time of this operation is proportional to the number of bits flipped, which serves as the cost for this example. Again, we'll use \$1 to represent each unit of cost (the flipping of a bit in this example).

For the amortized analysis, the amortized cost to set a 0-bit to 1 is \$2. When a bit is set to 1, \$1 of the \$2 pays to actually set the bit. The second \$1 resides on the bit as credit to be used later if and when the bit is reset to 0. At any point in time, every 1-bit in the counter has \$1 of credit on it, and thus resetting a bit to 0 can be viewed as costing nothing, and the \$1 on the bit prepays for the reset.

Here is how to determine the amortized cost of INCREMENT. The cost of resetting the bits to 0 within the **while** loop is paid for by the dollars on the bits that are reset. The INCREMENT procedure sets at most one bit to 1, in line 6, and therefore the amortized cost of an INCREMENT operation is at most \$2. The number of 1-bits in the counter never becomes negative, and thus the amount of credit stays nonnegative at all times. Thus, for n INCREMENT operations, the total amortized cost is $O(n)$, which bounds the total actual cost.

Exercises

16.2-1

You perform a sequence of PUSH and POP operations on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made automat-

ically, for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

16.2-2
Redo Exercise 16.1-3 using an accounting method of analysis.

16.2-3
You wish not only to increment a counter but also to reset it to 0 (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes $O(n)$ time on an initially zero counter. (*Hint:* Keep a pointer to the high-order 1.)

16.3 The potential method

Instead of representing prepaid work as credit stored with specific objects in the data structure, the ***potential method*** of amortized analysis represents the prepaid work as "potential energy," or just "potential," which can be released to pay for future operations. The potential applies to the data structure as a whole rather than to specific objects within the data structure.

The potential method works as follows. Starting with an initial data structure D_0, a sequence of n operations occurs. For each $i = 1, 2, \ldots, n$, let c_i be the actual cost of the ith operation and D_i be the data structure that results after applying the ith operation to data structure D_{i-1}. A ***potential function*** Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the ***potential*** associated with D_i. The ***amortized cost*** $\widehat{c}_i$ of the ith operation with respect to potential function Φ is defined by

$$\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \tag{16.2}$$

The amortized cost of each operation is therefore its actual cost plus the change in potential due to the operation. By equation (16.2), the total amortized cost of the n operations is

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) . \end{aligned} \tag{16.3}$$

The second equation follows from equation (A.12) on page 1143 because the $\Phi(D_i)$ terms telescope.

If you can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \widehat{c}_i$ gives an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$. In practice, you don't always know how many operations might be performed. Therefore, if you require that $\Phi(D_i) \geq \Phi(D_0)$ for all i, then you guarantee, as in the accounting method, that you've paid in advance. It's usually simplest to just define $\Phi(D_0)$ to be 0 and then show that $\Phi(D_i) \geq 0$ for all i. (See Exercise 16.3-1 for an easy way to handle cases in which $\Phi(D_0) \neq 0$.)

Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the ith operation is positive, then the amortized cost $\widehat{c}_i$ represents an overcharge to the ith operation, and the potential of the data structure increases. If the potential difference is negative, then the amortized cost represents an undercharge to the ith operation, and the decrease in the potential pays for the actual cost of the operation.

The amortized costs defined by equations (16.2) and (16.3) depend on the choice of the potential function Φ. Different potential functions may yield different amortized costs, yet still be upper bounds on the actual costs. You will often find tradeoffs that you can make in choosing a potential function. The best potential function to use depends on the desired time bounds.

Stack operations

To illustrate the potential method, we return once again to the example of the stack operations PUSH, POP, and MULTIPOP. We define the potential function Φ on a stack to be the number of objects in the stack. The potential of the empty initial stack D_0 is $\Phi(D_0) = 0$. Since the number of objects in the stack is never negative, the stack D_i that results after the ith operation has nonnegative potential, and thus

$$\begin{aligned} \Phi(D_i) &\geq 0 \\ &= \Phi(D_0) \, . \end{aligned}$$

The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

Now let's compute the amortized costs of the various stack operations. If the ith operation on a stack containing s objects is a PUSH operation, then the potential difference is

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1 \, . \end{aligned}$$

By equation (16.2), the amortized cost of this PUSH operation is

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2 .\end{aligned}$$

Suppose that the ith operation on the stack of s objects is MULTIPOP(S, k), which causes $k' = \min\{s, k\}$ objects to be popped off the stack. The actual cost of the operation is k', and the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = -k' .$$

Thus, the amortized cost of the MULTIPOP operation is

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 .\end{aligned}$$

Similarly, the amortized cost of an ordinary POP operation is 0.

The amortized cost of each of the three operations is $O(1)$, and thus the total amortized cost of a sequence of n operations is $O(n)$. Since $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore $O(n)$.

Incrementing a binary counter

As another example of the potential method, we revisit incrementing a k-bit binary counter. This time, the potential of the counter after the ith INCREMENT operation is defined to be the number of 1-bits in the counter after the ith operation, which we'll denote by b_i.

Here is how to compute the amortized cost of an INCREMENT operation. Suppose that the ith INCREMENT operation resets t_i bits to 0. The actual cost c_i of the operation is therefore at most $t_i + 1$, since in addition to resetting t_i bits, it sets at most one bit to 1. If $b_i = 0$, then the ith operation had reset all k bits to 0, and so $b_{i-1} = t_i = k$. If $b_i > 0$, then $b_i = b_{i-1} - t_i + 1$. In either case, $b_i \leq b_{i-1} - t_i + 1$, and the potential difference is

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i .\end{aligned}$$

The amortized cost is therefore

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2 .\end{aligned}$$

If the counter starts at 0, then $\Phi(D_0) = 0$. Since $\Phi(D_i) \geq 0$ for all i, the total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost, and so the worst-case cost of n INCREMENT operations is $O(n)$.

The potential method provides a simple and clever way to analyze the counter even when it does not start at 0. The counter starts with b_0 1-bits, and after n INCREMENT operations it has b_n 1-bits, where $0 \leq b_0, b_n \leq k$. Rewrite equation (16.3) as

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \widehat{c}_i - \Phi(D_n) + \Phi(D_0) .$$

Since $\Phi(D_0) = b_0$, $\Phi(D_n) = b_n$, and $\widehat{c}_i \leq 2$ for all $1 \leq i \leq n$, the total actual cost of n INCREMENT operations is

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0 . \end{aligned}$$

In particular, $b_0 \leq k$ means that as long as $k = O(n)$, the total actual cost is $O(n)$. In other words, if at least $n = \Omega(k)$ INCREMENT operations occur, the total actual cost is $O(n)$, no matter what initial value the counter contains.

Exercises

16.3-1

Suppose you have a potential function Φ such that $\Phi(D_i) \geq \Phi(D_0)$ for all i, but $\Phi(D_0) \neq 0$. Show that there exists a potential function Φ' such that $\Phi'(D_0) = 0$, $\Phi'(D_i) \geq 0$ for all $i \geq 1$, and the amortized costs using Φ' are the same as the amortized costs using Φ.

16.3-2

Redo Exercise 16.1-3 using a potential method of analysis.

16.3-3

Consider an ordinary binary min-heap data structure supporting the instructions INSERT and EXTRACT-MIN that, when there are n items in the heap, implements each operation in $O(\lg n)$ worst-case time. Give a potential function Φ such that the amortized cost of INSERT is $O(\lg n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that your potential function yields these amortized time bounds. Note that in the analysis, n is the number of items currently in the heap, and you do not know a bound on the maximum number of items that can ever be stored in the heap.

16.3-4
What is the total cost of executing n of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with s_0 objects and finishes with s_n objects?

16.3-5
Show how to implement a queue with two ordinary stacks (Exercise 10.1-7) so that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$.

16.3-6
Design a data structure to support the following two operations for a dynamic multiset S of integers, which allows duplicate values:

INSERT(S, x) inserts x into S.

DELETE-LARGER-HALF(S) deletes the largest $\lceil |S|/2 \rceil$ elements from S.

Explain how to implement this data structure so that any sequence of m INSERT and DELETE-LARGER-HALF operations runs in $O(m)$ time. Your implementation should also include a way to output the elements of S in $O(|S|)$ time.

16.4 Dynamic tables

When you design an application that uses a table, you do not always know in advance how many items the table will hold. You might allocate space for the table, only to find out later that it is not enough. The program must then reallocate the table with a larger size and copy all items stored in the original table over into the new, larger table. Similarly, if many items have been deleted from the table, it might be worthwhile to reallocate the table with a smaller size. This section studies this problem of dynamically expanding and contracting a table. Amortized analyses will show that the amortized cost of insertion and deletion is only $O(1)$, even though the actual cost of an operation is large when it triggers an expansion or a contraction. Moreover, you'll see how to guarantee that the unused space in a dynamic table never exceeds a constant fraction of the total space.

Let's assume that the dynamic table supports the operations TABLE-INSERT and TABLE-DELETE. TABLE-INSERT inserts into the table an item that occupies a single ***slot***, that is, a space for one item. Likewise, TABLE-DELETE removes an item from the table, thereby freeing a slot. The details of the data-structuring method used to organize the table are unimportant: it could be a stack (Section 10.1.3), a heap (Chapter 6), a hash table (Chapter 11), or something else.

It is convenient to use a concept introduced in Section 11.2, where we analyzed hashing. The ***load factor*** $\alpha(T)$ of a nonempty table T is defined as the number of items stored in the table divided by the size (number of slots) of the table. An empty table (one with no slots) has size 0, and its load factor is defined to be 1. If the load factor of a dynamic table is bounded below by a constant, the unused space in the table is never more than a constant fraction of the total amount of space.

We start by analyzing a dynamic table that allows only insertion and then move on to the more general case that supports both insertion and deletion.

16.4.1 Table expansion

Let's assume that storage for a table is allocated as an array of slots. A table fills up when all slots have been used or, equivalently, when its load factor is 1.[1] In some software environments, upon an attempt to insert an item into a full table, the only alternative is to abort with an error. The scenario in this section assumes, however, that the software environment, like many modern ones, provides a memory-management system that can allocate and free blocks of storage on request. Thus, upon inserting an item into a full table, the system can ***expand*** the table by allocating a new table with more slots than the old table had. Because the table must always reside in contiguous memory, the system must allocate a new array for the larger table and then copy items from the old table into the new table.

A common heuristic allocates a new table with twice as many slots as the old one. If the only table operations are insertions, then the load factor of the table is always at least $1/2$, and thus the amount of wasted space never exceeds half the total space in the table.

The TABLE-INSERT procedure on the following page assumes that T is an object representing the table. The attribute $T.table$ contains a pointer to the block of storage representing the table, $T.num$ contains the number of items in the table, and $T.size$ gives the total number of slots in the table. Initially, the table is empty: $T.num = T.size = 0$.

There are two types of insertion here: the TABLE-INSERT procedure itself and the ***elementary insertion*** into a table in lines 6 and 10. We can analyze the running time of TABLE-INSERT in terms of the number of elementary insertions by assigning a cost of 1 to each elementary insertion. In most computing environments, the overhead for allocating an initial table in line 2 is constant and the overhead for allocating and freeing storage in lines 5 and 7 is dominated by the cost of transfer-

[1] In some situations, such as an open-address hash table, it's better to consider a table to be full if its load factor equals some constant strictly less than 1. (See Exercise 16.4-2.)

```
TABLE-INSERT(T, x)
1  if T.size == 0
2      allocate T.table with 1 slot
3      T.size = 1
4  if T.num == T.size
5      allocate new-table with 2 · T.size slots
6      insert all items in T.table into new-table
7      free T.table
8      T.table = new-table
9      T.size = 2 · T.size
10 insert x into T.table
11 T.num = T.num + 1
```

ring items in line 6. Thus, the actual running time of TABLE-INSERT is linear in the number of elementary insertions. An ***expansion*** occurs when lines 5–9 execute.

Now, we'll use all three amortized analysis techniques to analyze a sequence of n TABLE-INSERT operations on an initially empty table. First, we need to determine the actual cost c_i of the ith operation. If the current table has room for the new item (or if this is the first operation), then $c_i = 1$, since the only elementary insertion performed is the one in line 10. If the current table is full, however, and an expansion occurs, then $c_i = i$: the cost is 1 for the elementary insertion in line 10 plus $i - 1$ for the items copied from the old table to the new table in line 6. For n operations, the worst-case cost of an operation is $O(n)$, which leads to an upper bound of $O(n^2)$ on the total running time for n operations.

This bound is not tight, because the table rarely expands in the course of n TABLE-INSERT operations. Specifically, the ith operation causes an expansion only when $i - 1$ is an exact power of 2. The amortized cost of an operation is in fact $O(1)$, as an aggregate analysis shows. The cost of the ith operation is

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is an exact power of 2 ,} \\ 1 & \text{otherwise .} \end{cases}$$

The total cost of n TABLE-INSERT operations is therefore

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \qquad \text{(by equation (A.6) on page 1142)} \\ &= 3n \ , \end{aligned}$$

Figure 16.3 Analysis of table expansion by the accounting method. Each call of TABLE-INSERT charges \$3 as follows: \$1 to pay for the elementary insertion, \$1 on the item inserted as prepayment for it to be reinserted later, and \$1 on an item that was already in the table, also as prepayment for reinsertion. **(a)** The table immediately after an expansion, with 8 slots, 4 items (tan slots), and no stored credit. **(b)–(e)** After each of 4 calls to TABLE-INSERT, the table has one more item, with \$1 stored on the new item and \$1 stored on one of the 4 items that were present immediately after the expansion. Slots with these new items are blue. **(f)** Upon the next call to TABLE-INSERT, the table is full, and so it expands again. Each item had \$1 to pay for it to be reinserted. Now the table looks as it did in part (a), with no stored credit but 16 slots and 8 items.

because at most n operations cost 1 each and the costs of the remaining operations form a geometric series. Since the total cost of n TABLE-INSERT operations is bounded by $3n$, the amortized cost of a single operation is at most 3.

The accounting method can provide some intuition for why the amortized cost of a TABLE-INSERT operation should be 3. You can think of each item paying for three elementary insertions: inserting itself into the current table, moving itself the next time that the table expands, and moving some other item that was already in the table the next time that the table expands. For example, suppose that the size of the table is m immediately after an expansion, as shown in Figure 16.3 for $m = 8$. Then the table holds $m/2$ items, and it contains no credit. Each call of TABLE-INSERT charges \$3. The elementary insertion that occurs immediately costs \$1. Another \$1 resides on the item inserted as credit. The third \$1 resides as credit on one of the $m/2$ items already in the table. The table will not fill again until another $m/2 - 1$ items have been inserted, and thus, by the time the table contains m items and is full, each item has \$1 on it to pay for it to be reinserted it during the expansion.

Now, let's see how to use the potential method. We'll use it again in Section 16.4.2 to design a TABLE-DELETE operation that has an $O(1)$ amortized cost

as well. Just as the accounting method had no stored credit immediately after an expansion—that is, when $T.num = T.size/2$—let's define the potential to be 0 when $T.num = T.size/2$. As elementary insertions occur, the potential needs to increase enough to pay for all the reinsertions that will happen when the table next expands. The table fills after another $T.size/2$ calls of TABLE-INSERT, when $T.num = T.size$. The next call of TABLE-INSERT after these $T.size/2$ calls triggers an expansion with a cost of $T.size$ to reinsert all the items. Therefore, over the course of $T.size/2$ calls of TABLE-INSERT, the potential must increase from 0 to $T.size$. To achieve this increase, let's design the potential so that each call of TABLE-INSERT increases it by

$$\frac{T.size}{T.size/2} = 2 ,$$

until the table expands. You can see that the potential function

$$\Phi(T) = 2(T.num - T.size/2) \tag{16.4}$$

equals 0 immediately after the table expands, when $T.num = T.size/2$, and it increases by 2 upon each insertion until the table fills. Once the table fills, that is, when $T.num = T.size$, the potential $\Phi(T)$ equals $T.size$. The initial value of the potential is 0, and since the table is always at least half full, $T.num \geq T.size/2$, which implies that $\Phi(T)$ is always nonnegative. Thus, the sum of the amortized costs of n TABLE-INSERT operations gives an upper bound on the sum of the actual costs.

To analyze the amortized costs of table operations, it is convenient to think in terms of the change in potential due to each operation. Letting Φ_i denote the potential after the ith operation, we can rewrite equation (16.2) as

$$\begin{aligned} \widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= c_i + \Delta\Phi_i , \end{aligned}$$

where $\Delta\Phi_i$ is the change in potential due to the ith operation. First, consider the case when the ith insertion does not cause the table to expand. In this case, $\Delta\Phi_i$ is 2. Since the actual cost c_i is 1, the amortized cost is

$$\begin{aligned} \widehat{c}_i &= c_i + \Delta\Phi_i \\ &= 1 + 2 \\ &= 3 . \end{aligned}$$

Now, consider the change in potential when the table does expand during the ith insertion because it was full immediately before the insertion. Let num_i denote the number of items stored in the table after the ith operation and $size_i$ denote the total size of the table after the ith operation, so that $size_{i-1} = num_{i-1} = i - 1$

Figure 16.4 The effect of a sequence of n TABLE-INSERT operations on the number num_i of items in the table (the brown line), the number $size_i$ of slots in the table (the blue line), and the potential $\Phi_i = 2(num_i - size_i/2)$ (the red line), each being measured after the ith operation. Immediately before an expansion, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table. Afterward, the potential drops to 0, but it immediately increases by 2 upon insertion of the item that caused the expansion.

and therefore $\Phi_{i-1} = 2(size_{i-1} - size_{i-1}/2) = size_{i-1} = i - 1$. Immediately after the expansion, the potential goes down to 0, and then the new item is inserted, causing the potential to increase to $\Phi_i = 2$. Thus, when the ith insertion triggers an expansion, $\Delta\Phi_i = 2 - (i - 1) = 3 - i$. When the table expands in the ith TABLE-INSERT operation, the actual cost c_i equals i (to reinsert $i - 1$ items and insert the ith item), giving an amortized cost of

$$\begin{aligned}\hat{c}_i &= c_i + \Delta\Phi_i \\ &= i + (3 - i) \\ &= 3 .\end{aligned}$$

Figure 16.4 plots the values of num_i, $size_i$, and Φ_i against i. Notice how the potential builds to pay for expanding the table.

16.4.2 Table expansion and contraction

To implement a TABLE-DELETE operation, it is simple enough to remove the specified item from the table. In order to limit the amount of wasted space, however, you might want to ***contract*** the table when the load factor becomes too small. Ta-

ble contraction is analogous to table expansion: when the number of items in the table drops too low, allocate a new, smaller table and then copy the items from the old table into the new one. You can then free the storage for the old table by returning it to the memory-management system. In order to not waste space, yet keep the amortized costs low, the insertion and deletion procedures should preserve two properties:

- the load factor of the dynamic table is bounded below by a positive constant, as well as above by 1, and
- the amortized cost of a table operation is bounded above by a constant.

The actual cost of each operation equals the number of elementary insertions or deletions.

You might think that if you double the table size upon inserting an item into a full table, then you should halve the size when deleting an item that would cause the table to become less than half full. This strategy does indeed guarantee that the load factor of the table never drops below $1/2$. Unfortunately, it can also cause the amortized cost of an operation to be quite large. Consider the following scenario. Perform n operations on a table T of size $n/2$, where n is an exact power of 2. The first $n/2$ operations are insertions, which by our previous analysis cost a total of $\Theta(n)$. At the end of this sequence of insertions, $T.num = T.size = n/2$. For the second $n/2$ operations, perform the following sequence:

insert, delete, delete, insert, insert, delete, delete, insert, insert, ...

The first insertion causes the table to expand to size n. The two deletions that follow cause the table to contract back to size $n/2$. Two further insertions cause another expansion, and so forth. The cost of each expansion and contraction is $\Theta(n)$, and there are $\Theta(n)$ of them. Thus, the total cost of the n operations is $\Theta(n^2)$, making the amortized cost of an operation $\Theta(n)$.

The problem with this strategy is that after the table expands, not enough deletions occur to pay for a contraction. Likewise, after the table contracts, not enough insertions take place to pay for an expansion.

How can we solve this problem? Allow the load factor of the table to drop below $1/2$. Specifically, continue to double the table size upon inserting an item into a full table, but halve the table size when deleting an item causes the table to become less than $1/4$ full, rather than $1/2$ full as before. The load factor of the table is therefore bounded below by the constant $1/4$, and the load factor is $1/2$ immediately after a contraction.

An expansion or contraction should exhaust all the built-up potential, so that immediately after expansion or contraction, when the load factor is $1/2$, the table's potential is 0. Figure 16.5 shows the idea. As the load factor deviates from $1/2$, the

Figure 16.5 How to think about the potential function Φ for table insertion and deletion. When the load factor α is $1/2$, the potential is 0. In order to accumulate sufficient potential to pay for reinserting all $T.size$ items when the table fills, the potential needs to increase by 2 upon each insertion when $\alpha \geq 1/2$. Correspondingly, the potential decreases by 2 upon each deletion that leaves $\alpha \geq 1/2$. In order to accrue enough potential to cover the cost of reinserting all $T.size/4$ items when the table contracts, the potential needs to increase by 1 upon each deletion when $\alpha < 1/2$, and correspondingly the potential decreases by 1 upon each insertion that leaves $\alpha<1/2$. The red area represents load factors less than $1/4$, which are not allowed.

potential increases so that by the time an expansion or contraction occurs, the table has garnered sufficient potential to pay for copying all the items into the newly allocated table. Thus, the potential function should grow to $T.num$ by the time that the load factor has either increased to 1 or decreased to $1/4$. Immediately after either expanding or contracting the table, the load factor goes back to $1/2$ and the table's potential reduces back to 0.

We omit the code for TABLE-DELETE, since it is analogous to TABLE-INSERT. We assume that if a contraction occurs during TABLE-DELETE, it occurs after the item is deleted from the table. The analysis assumes that whenever the number of items in the table drops to 0, the table occupies no storage. That is, if $T.num = 0$, then $T.size = 0$.

How do we design a potential function that gives constant amortized time for both insertion and deletion? When the load factor is at least $1/2$, the same potential function, $\Phi(T) = 2(T.num - T.size/2)$, that we used for insertion still works. When the table is at least half full, each insertion increases the potential by 2 if the table does not expand, and each deletion reduces the potential by 2 if it does not cause the load factor to drop below $1/2$.

What about when the load factor is less than $1/2$, that is, when $1/4 \leq \alpha(T) < 1/2$? As before, when $\alpha(T) = 1/2$, so that $T.num = T.size/2$, the potential $\Phi(T)$ should be 0. To get the load factor from $1/2$ down to $1/4$, $T.size/4$ deletions need

to occur, at which time $T.num = T.size/4$. To pay for all the reinsertions, the potential must increase from 0 to $T.size/4$ over these $T.size/4$ deletions. Therefore, for each call of TABLE-DELETE until the table contracts, the potential should increase by

$$\frac{T.size/4}{T.size/4} = 1 .$$

Likewise, when $\alpha < 1/2$, each call of TABLE-INSERT should decrease the potential by 1. When $1/4 \le \alpha(T) < 1/2$, the potential function

$$\Phi(T) = T.size/2 - T.num$$

produces this desired behavior.

Putting the two cases together, we get the potential function

$$\Phi(T) = \begin{cases} 2(T.num - T.size/2) & \text{if } \alpha(T) \ge 1/2 , \\ T.size/2 - T.num & \text{if } \alpha(T) < 1/2 . \end{cases} \tag{16.5}$$

The potential of an empty table is 0 and the potential is never negative. Thus, the total amortized cost of a sequence of operations with respect to Φ provides an upper bound on the actual cost of the sequence. Figure 16.6 illustrates how the potential function behaves over a sequence of insertions and deletions.

Now, let's determine the amortized costs of each operation. As before, let num_i denote the number of items stored in the table after the ith operation, $size_i$ denote the total size of the table after the ith operation, $\alpha_i = num_i/size_i$ denote the load factor after the ith operation, Φ_i denote the potential after the ith operation, and $\Delta\Phi_i$ denote the change in potential due to the ith operation. Initially, $num_0 = 0$, $size_0 = 0$, and $\Phi_0 = 0$.

The cases in which the table does not expand or contract and the load factor does not cross $\alpha = 1/2$ are straightforward. As we have seen, if $\alpha_{i-1} \ge 1/2$ and the ith operation is an insertion that does not cause the table to expand, then $\Delta\Phi_i = 2$. Likewise, if the ith operation is a deletion and $\alpha_i \ge 1/2$, then $\Delta\Phi_i = -2$. Furthermore, if $\alpha_{i-1} < 1/2$ and the ith operation is a deletion that does not trigger a contraction, then $\Delta\Phi_i = 1$, and if the ith operation is an insertion and $\alpha_i < 1/2$, then $\Delta\Phi_i = -1$. In other words, if no expansion or contraction occurs and the load factor does not cross $\alpha = 1/2$, then

- if the load factor stays at or above $1/2$, then the potential increases by 2 for an insertion and decreases by 2 for a deletion, and
- if the load factor stays below $1/2$, then the potential increases by 1 for a deletion and decreases by 1 for an insertion.

In each of these cases, the actual cost c_i of the ith operation is just 1, and so

Figure 16.6 The effect of a sequence of n TABLE-INSERT and TABLE-DELETE operations on the number num_i of items in the table (the brown line), the number $size_i$ of slots in the table (the blue line), and the potential (the red line)

$$\Phi_i = \begin{cases} 2(num_i - size_i/2) & \text{if } \alpha_i \geq 1/2 , \\ size_i/2 - num_i & \text{if } \alpha_i < 1/2 , \end{cases}$$

where $\alpha_i = num_i/size_i$, each measured after the ith operation. Immediately before an expansion or contraction, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table.

- if the ith operation is an insertion, its amortized cost $\widehat{c}_i$ is $c_i + \Delta\Phi_i$, which is $1 + 2 = 3$ if the load factor stays at or above $1/2$, and $1 + (-1) = 0$ if the load factor stays below $1/2$, and
- if the ith operation is a deletion, its amortized cost $\widehat{c}_i$ is $c_i + \Delta\Phi_i$, which is $1 + (-2) = -1$ if the load factor stays at or above $1/2$, and $1 + 1 = 2$ if the load factor stays below $1/2$.

Four cases remain: an insertion that takes the load factor from below $1/2$ to $1/2$, a deletion that takes the load factor from $1/2$ to below $1/2$, a deletion that causes the table to contract, and an insertion that causes the table to expand. We analyzed that last case at the end of Section 16.4.1 to show that its amortized cost is 3.

When the ith operation is a deletion that causes the table to contract, we have $num_{i-1} = size_{i-1}/4$ before the contraction, then the item is deleted, and finally $num_i = size_i/2 - 1$ after the contraction. Thus, by equation (16.5) we have

$$\begin{aligned}\Phi_{i-1} &= size_{i-1}/2 - num_{i-1} \\ &= size_{i-1}/2 - size_{i-1}/4 \\ &= size_{i-1}/4 \ ,\end{aligned}$$

which also equals the actual cost c_i of deleting one item and copying $size_{i-1}/4 - 1$ items into the new, smaller table. Since $num_i = size_i/2 - 1$ after the operation has completed, $\alpha_i < 1/2$, and so

$$\begin{aligned}\Phi_i &= size_i/2 - num_i \\ &= 1 \ ,\end{aligned}$$

giving $\Delta\Phi_i = 1 - size_{i-1}/4$. Therefore, when the ith operation is a deletion that triggers a contraction, its amortized cost is

$$\begin{aligned}\widehat{c}_i &= c_i + \Delta\Phi_i \\ &= size_{i-1}/4 + (1 - size_{i-1}/4) \\ &= 1 \ .\end{aligned}$$

Finally, we handle the cases where the load factor fits one case of equation (16.5) before the operation and the other case afterward. We start with deletion, where we have $num_{i-1} = size_{i-1}/2$, so that $\alpha_{i-1} = 1/2$, beforehand, and $num_i = size_i/2 - 1$, so that $\alpha_i < 1/2$ afterward. Because $\alpha_{i-1} = 1/2$, we have $\Phi_{i-1} = 0$, and because $\alpha_i < 1/2$, we have $\Phi_i = size_i/2 - num_i = 1$. Thus we get that $\Delta\Phi_i = 1 - 0 = 1$. Since the ith operation is a deletion that does not cause a contraction, the actual cost c_i equals 1, and the amortized cost $\widehat{c}_i$ is $c_i + \Delta\Phi_i = 1 + 1 = 2$.

Conversely, if the ith operation is an insertion that takes the load factor from below $1/2$ to equaling $1/2$, the change in potential $\Delta\Phi_i$ equals -1. Again, the actual cost c_i is 1, and now the amortized cost $\widehat{c}_i$ is $c_i + \Delta\Phi_i = 1 + (-1) = 0$.

In summary, since the amortized cost of each operation is bounded above by a constant, the actual time for any sequence of n operations on a dynamic table is $O(n)$.

Exercises

16.4-1

Using the potential method, analyze the amortized cost of the first table insertion.

16.4-2

You wish to implement a dynamic, open-address hash table. Why might you consider the table to be full when its load factor reaches some value α that is strictly less than 1? Describe briefly how to make insertion into a dynamic, open-address hash table run in such a way that the expected value of the amortized cost per

insertion is $O(1)$. Why is the expected value of the actual cost per insertion not necessarily $O(1)$ for all insertions?

16.4-3
Discuss how to use the accounting method to analyze both the insertion and deletion operations, assuming that the table doubles in size when its load factor exceeds 1 and the table halves in size when its load factor goes below $1/4$.

16.4-4
Suppose that instead of contracting a table by halving its size when its load factor drops below $1/4$, you contract the table by multiplying its size by $2/3$ when its load factor drops below $1/3$. Using the potential function

$$\Phi(T) = |2(T.num - T.size/2)| \ ,$$

show that the amortized cost of a TABLE-DELETE that uses this strategy is bounded above by a constant.

Problems

16-1 Binary reflected Gray code
A ***binary Gray code*** represents a sequence of nonnegative integers in binary such that to go from one integer to the next, exactly one bit flips every time. The ***binary reflected Gray code*** represents a sequence of the integers 0 to $2^k - 1$ for some positive integer k according to the following recursive method:

- For $k = 1$, the binary reflected Gray code is $\langle 0, 1 \rangle$.
- For $k \geq 2$, first form the binary reflected Gray code for $k - 1$, giving the 2^{k-1} integers 0 to $2^{k-1} - 1$. Then form the reflection of this sequence, which is just the sequence in reverse. (That is, the jth integer in the sequence becomes the $(2^{k-1} - j - 1)$st integer in the reflection). Next, add 2^{k-1} to each of the 2^{k-1} integers in the reflected sequence. Finally, concatenate the two sequences.

For example, for $k = 2$, first form the binary reflected Gray code $\langle 0, 1 \rangle$ for $k = 1$. Its reflection is the sequence $\langle 1, 0 \rangle$. Adding $2^{k-1} = 2$ to each integer in the reflection gives the sequence $\langle 3, 2 \rangle$. Concatenating the two sequences gives $\langle 0, 1, 3, 2 \rangle$ or, in binary, $\langle 00, 01, 11, 10 \rangle$, so that each integer differs from its predecessor by exactly one bit. For $k = 3$, the reflection of the binary reflected Gray code for $k = 2$ is $\langle 2, 3, 1, 0 \rangle$ and adding $2^{k-1} = 4$ gives $\langle 6, 7, 5, 4 \rangle$. Concatenating produces the sequence $\langle 0, 1, 3, 2, 6, 7, 5, 4 \rangle$, which in binary is $\langle 000, 001, 011, 010, 110, 111, 101, 100 \rangle$. In the binary reflected Gray code, only one bit flips even when wrapping around from the last integer to the first.

a. Index the integers in a binary reflected Gray code from 0 to $2^k - 1$, and consider the ith integer in the binary reflected Gray code. To go from the $(i-1)$st integer to the ith integer in the binary reflected Gray code, exactly one bit flips. Show how to determine which bit flips, given the index i.

b. Assuming that given a bit number j, you can flip bit j of an integer in constant time, show how to compute the entire binary reflected Gray code sequence of 2^k numbers in $\Theta(2^k)$ time.

16-2 Making binary search dynamic

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. You can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that you wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \lg(n+1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. Maintain k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k-1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i \, 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

a. Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

b. Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times, assuming that the only operations are INSERT and SEARCH.

c. Describe how to implement DELETE. Analyze its worst-case and amortized running times, assuming that there can be DELETE, INSERT, and SEARCH operations.

16-3 Amortized weight-balanced trees

Consider an ordinary binary search tree augmented by adding to each node x the attribute $x.size$, which gives the number of keys stored in the subtree rooted at x. Let α be a constant in the range $1/2 \le \alpha < 1$. We say that a given node x is ***α-balanced*** if $x.left.size \le \alpha \cdot x.size$ and $x.right.size \le \alpha \cdot x.size$. The tree as a whole is ***α-balanced*** if every node in the tree is α-balanced. The following amortized approach to maintaining weight-balanced trees was suggested by G. Varghese.

a. A $1/2$-balanced tree is, in a sense, as balanced as it can be. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes $1/2$-balanced. Your algorithm should run in $\Theta(x.size)$ time, and it can use $O(x.size)$ auxiliary storage.

b. Show that performing a search in an n-node α-balanced binary search tree takes $O(\lg n)$ worst-case time.

For the remainder of this problem, assume that the constant α is strictly greater than $1/2$. Suppose that you implement INSERT and DELETE as usual for an n-node binary search tree, except that after every such operation, if any node in the tree is no longer α-balanced, then you "rebuild" the subtree rooted at the highest such node in the tree so that it becomes $1/2$-balanced.

We'll analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T, define

$$\Delta(x) = |x.left.size - x.right.size| \ .$$

Define the potential of T as

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x),$$

where c is a sufficiently large constant that depends on α.

c. Argue that any binary search tree has nonnegative potential and also that a $1/2$-balanced tree has potential 0.

d. Suppose that m units of potential can pay for rebuilding an m-node subtree. How large must c be in terms of α in order for it to take $O(1)$ amortized time to rebuild a subtree that is not α-balanced?

e. Show that inserting a node into or deleting a node from an n-node α-balanced tree costs $O(\lg n)$ amortized time.

16-4 The cost of restructuring red-black trees

There are four basic operations on red-black trees that perform ***structural modifications***: node insertions, node deletions, rotations, and color changes. We have seen that RB-INSERT and RB-DELETE use only $O(1)$ rotations, node insertions, and node deletions to maintain the red-black properties, but they may make many more color changes.

a. Describe a legal red-black tree with n nodes such that calling RB-INSERT to add the $(n + 1)$st node causes $\Omega(\lg n)$ color changes. Then describe a legal

red-black tree with n nodes for which calling RB-DELETE on a particular node causes $\Omega(\lg n)$ color changes.

Although the worst-case number of color changes per operation can be logarithmic, you will prove that any sequence of m RB-INSERT and RB-DELETE operations on an initially empty red-black tree causes $O(m)$ structural modifications in the worst case.

b. Some of the cases handled by the main loop of the code of both RB-INSERT-FIXUP and RB-DELETE-FIXUP are ***terminating***: once encountered, they cause the loop to terminate after a constant number of additional operations. For each of the cases of RB-INSERT-FIXUP and RB-DELETE-FIXUP, specify which are terminating and which are not. (*Hint:* Look at Figures 13.5, 13.6, and 13.7 in Sections 13.3 and 13.4.)

You will first analyze the structural modifications when only insertions are performed. Let T be a red-black tree, and define $\Phi(T)$ to be the number of red nodes in T. Assume that one unit of potential can pay for the structural modifications performed by any of the three cases of RB-INSERT-FIXUP.

c. Let T' be the result of applying Case 1 of RB-INSERT-FIXUP to T. Argue that $\Phi(T') = \Phi(T) - 1$.

d. We can break the operation of the RB-INSERT procedure into three parts. List the structural modifications and potential changes resulting from lines 1–16 of RB-INSERT, from nonterminating cases of RB-INSERT-FIXUP, and from terminating cases of RB-INSERT-FIXUP.

e. Using part (d), argue that the amortized number of structural modifications performed by any call of RB-INSERT is $O(1)$.

Next you will prove that there are $O(m)$ structural modifications when both insertions and deletions occur. Define, for each node x,

$$w(x) = \begin{cases} 0 & \text{if } x \text{ is red ,} \\ 1 & \text{if } x \text{ is black and has no red children ,} \\ 0 & \text{if } x \text{ is black and has one red child ,} \\ 2 & \text{if } x \text{ is black and has two red children .} \end{cases}$$

Now redefine the potential of a red-black tree T as

$$\Phi(T) = \sum_{x \in T} w(x) ,$$

and let T' be the tree that results from applying any nonterminating case of RB-INSERT-FIXUP or RB-DELETE-FIXUP to T.

f. Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-INSERT-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-INSERT-FIXUP is $O(1)$.

g. Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-DELETE-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-DELETE-FIXUP is $O(1)$.

h. Complete the proof that in the worst case, any sequence of m RB-INSERT and RB-DELETE operations performs $O(m)$ structural modifications.

Chapter notes

Aho, Hopcroft, and Ullman [5] used aggregate analysis to determine the running time of operations on a disjoint-set forest. We'll analyze this data structure using the potential method in Chapter 19. Tarjan [430] surveys the accounting and potential methods of amortized analysis and presents several applications. He attributes the accounting method to several authors, including M. R. Brown, R. E. Tarjan, S. Huddleston, and K. Mehlhorn. He attributes the potential method to D. D. Sleator. The term "amortized" is due to D. D. Sleator and R. E. Tarjan.

Potential functions are also useful for proving lower bounds for certain types of problems. For each configuration of the problem, define a potential function that maps the configuration to a real number. Then determine the potential Φ_{init} of the initial configuration, the potential Φ_{final} of the final configuration, and the maximum change in potential $\Delta\Phi_{\max}$ due to any step. The number of steps must therefore be at least $|\Phi_{\text{final}} - \Phi_{\text{init}}| \,/\, |\Delta\Phi_{\max}|$. Examples of potential functions to prove lower bounds in I/O complexity appear in works by Cormen, Sundquist, and Wisniewski [105], Floyd [146], and Aggarwal and Vitter [3]. Krumme, Cybenko, and Venkataraman [271] applied potential functions to prove lower bounds on ***gossiping***: communicating a unique item from each vertex in a graph to every other vertex.

Part V Advanced Data Structures

Introduction

This part returns to studying data structures that support operations on dynamic sets, but at a more advanced level than Part III. One of the chapters, for example, makes extensive use of the amortized analysis techniques from Chapter 16.

Chapter 17 shows how to augment red-black trees—adding additional information in each node—to support dynamic-set operations in addition to those covered in Chapters 12 and 13. The first example augments red-black trees to dynamically maintain order statistics for a set of keys. Another example augments them in a different way to maintain intervals of real numbers. Chapter 17 includes a theorem giving sufficient conditions for when a red-black tree can be augmented while maintaining the $O(\lg n)$ running times for insertion and deletion.

Chapter 18 presents B-trees, which are balanced search trees specifically designed to be stored on disks. Since disks operate much more slowly than random-access memory, B-tree performance depends not only on how much computing time the dynamic-set operations consume but also on how many disk accesses they perform. For each B-tree operation, the number of disk accesses increases with the height of the B-tree, but B-tree operations keep the height low.

Chapter 19 examines data structures for disjoint sets. Starting with a universe of n elements, each initially in its own singleton set, the operation UNION unites two sets. At all times, the n elements are partitioned into disjoint sets, even as calls to the UNION operation change the members of a set dynamically. The query FIND-SET identifies the unique set that contains a given element at the moment. Representing each set as a simple rooted tree yields surprisingly fast operations: a sequence of m operations runs in $O(m\,\alpha(n))$ time, where $\alpha(n)$ is an incredibly slowly growing function—$\alpha(n)$ is at most 4 in any conceivable application. The amortized analysis that proves this time bound is as complex as the data structure is simple.

The topics covered in this part are by no means the only examples of "advanced" data structures. Other advanced data structures include the following:

- ***Fibonacci heaps*** [156] implement mergeable heaps (see Problem 10-2 on page 268) with the operations INSERT, MINIMUM, and UNION taking only $O(1)$ actual and amortized time, and the operations EXTRACT-MIN and DELETE taking $O(\lg n)$ amortized time. The most significant advantage of these data structures, however, is that DECREASE-KEY takes only $O(1)$ amortized time. ***Strict Fibonacci heaps*** [73], developed later, made all of these time bounds actual. Because the DECREASE-KEY operation takes constant amortized time, (strict) Fibonacci heaps constitute key components of some of the asymptotically fastest algorithms to date for graph problems.
- ***Dynamic trees*** [415, 429] maintain a forest of disjoint rooted trees. Each edge in each tree has a real-valued cost. Dynamic trees support queries to find parents, roots, edge costs, and the minimum edge cost on a simple path from a node up to a root. Trees may be manipulated by cutting edges, updating all edge costs on a simple path from a node up to a root, linking a root into another tree, and making a node the root of the tree it appears in. One implementation of dynamic trees gives an $O(\lg n)$ amortized time bound for each operation, while a more complicated implementation yields $O(\lg n)$ worst-case time bounds. Dynamic trees are used in some of the asymptotically fastest network-flow algorithms.
- ***Splay trees*** [418, 429] are a form of binary search tree on which the standard search-tree operations run in $O(\lg n)$ amortized time. One application of splay trees simplifies dynamic trees.
- ***Persistent*** data structures allow queries, and sometimes updates as well, on past versions of a data structure. For example, linked data structures can be made persistent with only a small time and space cost [126]. Problem 13-1 gives a simple example of a persistent dynamic set.
- Several data structures allow a faster implementation of dictionary operations (INSERT, DELETE, and SEARCH) for a restricted universe of keys. By taking advantage of these restrictions, they are able to achieve better worst-case asymptotic running times than comparison-based data structures. If the keys are unique integers drawn from the set $\{0, 1, 2, \ldots, u - 1\}$, where u is an exact power of 2, then a recursive data structure known as a ***van Emde Boas tree*** [440, 441] supports each of the operations SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(\lg \lg u)$ time. ***Fusion trees*** [157] were the first data structure to allow faster dictionary operations when the universe is restricted to integers, implementing these operations in $O(\lg n / \lg \lg n)$ time. Several subsequent data structures, including ***exponential search trees*** [17], have also given improved bounds on some or all of

the dictionary operations and are mentioned in the chapter notes throughout this book.

- ***Dynamic graph data structures*** support various queries while allowing the structure of a graph to change through operations that insert or delete vertices or edges. Examples of the queries that they support include vertex connectivity [214], edge connectivity, minimum spanning trees [213], biconnectivity, and transitive closure [212].

Chapter notes throughout this book mention additional data structures.

17 Augmenting Data Structures

Some solutions require no more than a "textbook" data structure—such as a doubly linked list, a hash table, or a binary search tree—but many others require a dash of creativity. Rarely will you need to create an entirely new type of data structure, though. More often, you can augment a textbook data structure by storing additional information in it. You can then program new operations for the data structure to support your application. Augmenting a data structure is not always straightforward, however, since the added information must be updated and maintained by the ordinary operations on the data structure.

This chapter discusses two data structures based on red-black trees that are augmented with additional information. Section 17.1 describes a data structure that supports general order-statistic operations on a dynamic set: quickly finding the ith smallest number or the rank of a given element. Section 17.2 abstracts the process of augmenting a data structure and provides a theorem that you can use when augmenting red-black trees. Section 17.3 uses this theorem to help design a data structure for maintaining a dynamic set of intervals, such as time intervals. You can use this data structure to quickly find an interval that overlaps a given query interval.

17.1 Dynamic order statistics

Chapter 9 introduced the notion of an order statistic. Specifically, the ith order statistic of a set of n elements, where $i \in \{1, 2, \ldots, n\}$, is simply the element in the set with the ith smallest key. In Chapter 9, you saw how to determine any order statistic in $O(n)$ time from an unordered set. This section shows how to modify red-black trees so that you can determine any order statistic for a dynamic set in $O(\lg n)$ time and also compute the ***rank*** of an element—its position in the linear order of the set—in $O(\lg n)$ time.

Figure 17.1 An order-statistic tree, which is an augmented red-black tree. In addition to its usual attributes, each node x has an attribute $x.size$, which is the number of nodes, other than the sentinel, in the subtree rooted at x.

Figure 17.1 shows a data structure that can support fast order-statistic operations. An ***order-statistic tree*** T is simply a red-black tree with additional information stored in each node. Each node x contains the usual red-black tree attributes $x.key$, $x.color$, $x.p$, $x.left$, and $x.right$, along with a new attribute, $x.size$. This attribute contains the number of internal nodes in the subtree rooted at x (including x itself, but not including any sentinels), that is, the size of the subtree. If we define the sentinel's size to be 0—that is, we set $T.nil.size$ to be 0—then we have the identity

$$x.size = x.left.size + x.right.size + 1 .$$

Keys need not be distinct in an order-statistic tree. For example, the tree in Figure 17.1 has two keys with value 14 and two keys with value 21. When equal keys are present, the above notion of rank is not well defined. We remove this ambiguity for an order-statistic tree by defining the rank of an element as the position at which it would be printed in an inorder walk of the tree. In Figure 17.1, for example, the key 14 stored in a black node has rank 5, and the key 14 stored in a red node has rank 6.

Retrieving the element with a given rank

Before we show how to maintain the size information during insertion and deletion, let's see how to implement two order-statistic queries that use this additional information. We begin with an operation that retrieves the element with a given rank. The procedure OS-SELECT(x, i) on the following page returns a pointer to the node containing the ith smallest key in the subtree rooted at x. To find the node with the ith smallest key in an order-statistic tree T, call OS-SELECT$(T.root, i)$.

Here is how OS-SELECT works. Line 1 computes r, the rank of node x within the subtree rooted at x. The value of $x.left.size$ is the number of nodes that come

```
OS-SELECT(x, i)
1  r = x.left.size + 1    // rank of x within the subtree rooted at x
2  if i == r
3      return x
4  elseif i < r
5      return OS-SELECT(x.left, i)
6  else return OS-SELECT(x.right, i − r)
```

before x in an inorder tree walk of the subtree rooted at x. Thus, $x.left.size + 1$ is the rank of x within the subtree rooted at x. If $i = r$, then node x is the ith smallest element, and so line 3 returns x. If $i < r$, then the ith smallest element resides in x's left subtree, and therefore, line 5 recurses on $x.left$. If $i > r$, then the ith smallest element resides in x's right subtree. Since the subtree rooted at x contains r elements that come before x's right subtree in an inorder tree walk, the ith smallest element in the subtree rooted at x is the $(i - r)$th smallest element in the subtree rooted at $x.right$. Line 6 determines this element recursively.

As an example of how OS-SELECT operates, consider a search for the 17th smallest element in the order-statistic tree of Figure 17.1. The search starts with x as the root, whose key is 26, and with $i = 17$. Since the size of 26's left subtree is 12, its rank is 13. Thus, the node with rank 17 is the $17 - 13 = 4$th smallest element in 26's right subtree. In the recursive call, x is the node with key 41, and $i = 4$. Since the size of 41's left subtree is 5, its rank within its subtree is 6. Therefore, the node with rank 4 is the 4th smallest element in 41's left subtree. In the recursive call, x is the node with key 30, and its rank within its subtree is 2. The procedure recurses once again to find the $4 - 2 = 2$nd smallest element in the subtree rooted at the node with key 38. Its left subtree has size 1, which means it is the second smallest element. Thus, the procedure returns a pointer to the node with key 38.

Because each recursive call goes down one level in the order-statistic tree, the total time for OS-SELECT is at worst proportional to the height of the tree. Since the tree is a red-black tree, its height is $O(\lg n)$, where n is the number of nodes. Thus, the running time of OS-SELECT is $O(\lg n)$ for a dynamic set of n elements.

Determining the rank of an element

Given a pointer to a node x in an order-statistic tree T, the procedure OS-RANK on the facing page returns the position of x in the linear order determined by an inorder tree walk of T.

```
OS-RANK(T, x)
1  r = x.left.size + 1          // rank of x within the subtree rooted at x
2  y = x                        // root of subtree being examined
3  while y ≠ T.root
4      if y == y.p.right                  // if root of a right subtree ...
5          r = r + y.p.left.size + 1      // ... add in parent and its left subtree
6      y = y.p                            // move y toward the root
7  return r
```

The OS-RANK procedure works as follows. You can think of node x's rank as the number of nodes preceding x in an inorder tree walk, plus 1 for x itself. OS-RANK maintains the following loop invariant:

> At the start of each iteration of the **while** loop of lines 3–6, r is the rank of $x.key$ in the subtree rooted at node y.

We use this loop invariant to show that OS-RANK works correctly as follows:

Initialization: Prior to the first iteration, line 1 sets r to be the rank of $x.key$ within the subtree rooted at x. Setting $y = x$ in line 2 makes the invariant true the first time the test in line 3 executes.

Maintenance: At the end of each iteration of the **while** loop, line 6 sets $y = y.p$. Thus, we must show that if r is the rank of $x.key$ in the subtree rooted at y at the start of the loop body, then r is the rank of $x.key$ in the subtree rooted at $y.p$ at the end of the loop body. In each iteration of the **while** loop, consider the subtree rooted at $y.p$. The value of r already includes the number of nodes in the subtree rooted at node y that precede x in an inorder walk, and so the procedure must add the nodes in the subtree rooted at y's sibling that precede x in an inorder walk, plus 1 for $y.p$ if it, too, precedes x. If y is a left child, then neither $y.p$ nor any node in $y.p$'s right subtree precedes x, and so OS-RANK leaves r alone. Otherwise, y is a right child and all the nodes in $y.p$'s left subtree precede x, as does $y.p$ itself. In this case, line 5 adds $y.p.left.size + 1$ to the current value of r.

Termination: Because each iteration of the loop moves y toward the root and the loop terminates when $y = T.root$, the loop eventually terminates. Moreover, the subtree rooted at y is the entire tree. Thus, the value of r is the rank of $x.key$ in the entire tree.

As an example, when OS-RANK runs on the order-statistic tree of Figure 17.1 to find the rank of the node with key 38, the following sequence of values of $y.key$ and r occurs at the top of the **while** loop:

iteration	$y.key$	r
1	38	2
2	30	4
3	41	4
4	26	17

The procedure returns the rank 17.

Since each iteration of the **while** loop takes $O(1)$ time, and y goes up one level in the tree with each iteration, the running time of OS-RANK is at worst proportional to the height of the tree: $O(\lg n)$ on an n-node order-statistic tree.

Maintaining subtree sizes

Given the *size* attribute in each node, OS-SELECT and OS-RANK can quickly compute order-statistic information. But if the basic modifying operations on red-black trees cannot efficiently maintain the *size* attribute, our work will have been for naught. Let's see how to maintain subtree sizes for both insertion and deletion without affecting the asymptotic running time of either operation.

Recall from Section 13.3 that insertion into a red-black tree consists of two phases. The first phase goes down the tree from the root, inserting the new node as a child of an existing node. The second phase goes up the tree, changing colors and performing rotations to maintain the red-black properties.

To maintain the subtree sizes in the first phase, simply increment $x.size$ for each node x on the simple path traversed from the root down toward the leaves. The new node added gets a *size* of 1. Since there are $O(\lg n)$ nodes on the traversed path, the additional cost of maintaining the *size* attributes is $O(\lg n)$.

In the second phase, the only structural changes to the underlying red-black tree are caused by rotations, of which there are at most two. Moreover, a rotation is a local operation: only two nodes have their *size* attributes invalidated. The link around which the rotation is performed is incident on these two nodes. Referring to the code for LEFT-ROTATE(T, x) on page 336, add the following lines:

```
y.size = x.size
x.size = x.left.size + x.right.size + 1
```

Figure 17.2 illustrates how the attributes are updated. The change to RIGHT-ROTATE is symmetric.

Since inserting into a red-black tree requires at most two rotations, updating the *size* attributes in the second phase costs only $O(1)$ additional time. Thus, the total time for insertion into an n-node order-statistic tree is $O(\lg n)$, which is asymptotically the same as for an ordinary red-black tree.

Figure 17.2 Updating subtree sizes during rotations. The updates are local, requiring only the *size* information stored in x, y, and the roots of the subtrees shown as triangles.

Deletion from a red-black tree also consists of two phases: the first operates on the underlying search tree, and the second causes at most three rotations and otherwise performs no structural changes. (See Section 13.4.) The first phase removes one node z from the tree and could move at most two other nodes within the tree (nodes y and x in Figure 12.4 on page 323). To update the subtree sizes, simply traverse a simple path from the lowest node that moves (starting from its original position within the tree) up to the root, decrementing the *size* attribute of each node on the path. Since this path has length $O(\lg n)$ in an n-node red-black tree, the additional time spent maintaining *size* attributes in the first phase is $O(\lg n)$. For the $O(1)$ rotations in the second phase of deletion, handle them in the same manner as for insertion. Thus, both insertion and deletion, including maintaining the *size* attributes, take $O(\lg n)$ time for an n-node order-statistic tree.

Exercises

17.1-1
Show how OS-SELECT$(T.root, 10)$ operates on the red-black tree T shown in Figure 17.1.

17.1-2
Show how OS-RANK(T, x) operates on the red-black tree T shown in Figure 17.1 and the node x with $x.key = 35$.

17.1-3
Write a nonrecursive version of OS-SELECT.

17.1-4
Write a procedure OS-KEY-RANK(T, k) that takes an order-statistic tree T and a key k and returns the rank of k in the dynamic set represented by T. Assume that the keys of T are distinct.

17.1-5
Given an element x in an n-node order-statistic tree and a natural number i, show how to determine the ith successor of x in the linear order of the tree in $O(\lg n)$ time.

17.1-6
The procedures OS-SELECT and OS-RANK use the *size* attribute of a node only to compute a rank. Suppose that you store in each node its rank in the subtree of which it is the root instead of the *size* attribute. Show how to maintain this information during insertion and deletion. (Remember that these two operations can cause rotations.)

17.1-7
Show how to use an order-statistic tree to count the number of inversions (see Problem 2-4 on page 47) in an array of n distinct elements in $O(n \lg n)$ time.

★ ***17.1-8***
Consider n chords on a circle, each defined by its endpoints. Describe an $O(n \lg n)$-time algorithm to determine the number of pairs of chords that intersect inside the circle. (For example, if the n chords are all diameters that meet at the center, then the answer is $\binom{n}{2}$.) Assume that no two chords share an endpoint.

17.2 How to augment a data structure

The process of augmenting a basic data structure to support additional functionality occurs quite frequently in algorithm design. We'll use it again in the next section to design a data structure that supports operations on intervals. This section examines the steps involved in such augmentation. It includes a useful theorem that allows you to augment red-black trees easily in many cases.

You can break the process of augmenting a data structure into four steps:

1. Choose an underlying data structure.
2. Determine additional information to maintain in the underlying data structure.
3. Verify that you can maintain the additional information for the basic modifying operations on the underlying data structure.
4. Develop new operations.

As with any prescriptive design method, you'll rarely be able to follow the steps precisely in the order given. Most design work contains an element of trial and error, and progress on all steps usually proceeds in parallel. There is no point,

for example, in determining additional information and developing new operations (steps 2 and 4) if you cannot maintain the additional information efficiently. Nevertheless, this four-step method provides a good focus for your efforts in augmenting a data structure, and it is also a good framework for documenting an augmented data structure.

We followed these four steps in Section 17.1 to design order-statistic trees. For step 1, we chose red-black trees as the underlying data structure. Red-black trees seemed like a good starting point because they efficiently support other dynamic-set operations on a total order, such as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR.

In Step 2, we added the *size* attribute, so that each node x stores the size of the subtree rooted at x. Generally, the additional information makes operations more efficient. For example, it is possible to implement OS-SELECT and OS-RANK using just the keys stored in the tree, but then they would not run in $O(\lg n)$ time. Sometimes, the additional information is pointer information rather than data, as in Exercise 17.2-1.

For step 3, we ensured that insertion and deletion can maintain the *size* attributes while still running in $O(\lg n)$ time. Ideally, you would like to update only a few elements of the data structure in order to maintain the additional information. For example, if each node simply stores its rank in the tree, the OS-SELECT and OS-RANK procedures run quickly, but inserting a new minimum element might cause a change to this information in every node of the tree. Because we chose to store subtree sizes instead, inserting a new element causes information to change in only $O(\lg n)$ nodes.

In Step 4, we developed the operations OS-SELECT and OS-RANK. After all, the need for new operations is why anyone bothers to augment a data structure in the first place. Occasionally, rather than developing new operations, you can use the additional information to expedite existing ones, as in Exercise 17.2-1.

Augmenting red-black trees

When red-black trees underlie an augmented data structure, we can prove that insertion and deletion can always efficiently maintain certain kinds of additional information, thereby simplifying step 3. The proof of the following theorem is similar to the argument from Section 17.1 that we can maintain the *size* attribute for order-statistic trees.

Theorem 17.1 (Augmenting a red-black tree)
Let f be an attribute that augments a red-black tree T of n nodes, and suppose that the value of f for each node x depends only the information in nodes x, $x.left$, and $x.right$ (possibly including $x.left.f$ and $x.right.f$), and that the value of $x.f$ can

be computed from this information in $O(1)$ time. Then, the insertion and deletion operations can maintain the values of f in all nodes of T without asymptotically affecting the $O(\lg n)$ running times of these operations.

Proof The main idea of the proof is that a change to an f attribute in a node x propagates only to ancestors of x in the tree. That is, changing $x.f$ may require $x.p.f$ to be updated, but nothing else; updating $x.p.f$ may require $x.p.p.f$ to be updated, but nothing else; and so on up the tree. After updating $T.root.f$, no other node depends on the new value, and so the process terminates. Since the height of a red-black tree is $O(\lg n)$, changing an f attribute in a node costs $O(\lg n)$ time in updating all nodes that depend on the change.

As we saw in Section 13.3, insertion of a node x into red-black tree T consists of two phases. If the tree T is empty, then the first phase simply makes x be the root of T. If T is not empty, then the first phase inserts x as a child of an existing node. Because we assume that the value of $x.f$ depends only on information in the other attributes of x itself and the information in x's children, and because x's children are both the sentinel $T.nil$, it takes only $O(1)$ time to compute the value of $x.f$. Having computed $x.f$, the change propagates up the tree. Thus, the total time for the first phase of insertion is $O(\lg n)$. During the second phase, the only structural changes to the tree come from rotations. Since only two nodes change in a rotation, but a change to an attribute might need to propagate up to the root, the total time for updating the f attributes is $O(\lg n)$ per rotation. Since the number of rotations during insertion is at most two, the total time for insertion is $O(\lg n)$.

Like insertion, deletion has two phases, as Section 13.4 discusses. In the first phase, changes to the tree occur when a node is deleted, and at most two other nodes could move within the tree. Propagating the updates to f caused by these changes costs at most $O(\lg n)$, since the changes modify the tree locally along a simple path from the lowest changed node to the root. Fixing up the red-black tree during the second phase requires at most three rotations, and each rotation requires at most $O(\lg n)$ time to propagate the updates to f. Thus, like insertion, the total time for deletion is $O(\lg n)$. ■

In many cases, such as maintaining the *size* attributes in order-statistic trees, the cost of updating after a rotation is $O(1)$, rather than the $O(\lg n)$ derived in the proof of Theorem 17.1. Exercise 17.2-3 gives an example.

On the other hand, when an update after a rotation requires a traversal all the way up to the root, it is important that insertion into and deletion from a red-black tree require a constant number of rotations. The chapter notes for Chapter 13 list other schemes for balancing search trees that do not bound the number of rotations per insertion or deletion by a constant. If each operation might require $\Theta(\lg n)$ rota-

tions and each rotation traverses a path up to the root, then a single operation could require $\Theta(\lg^2 n)$ time, rather than the $O(\lg n)$ time bound given by Theorem 17.1.

Exercises

17.2-1
Show, by adding pointers to the nodes, how to support each of the dynamic-set queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(1)$ worst-case time on an augmented order-statistic tree. The asymptotic performance of other operations on order-statistic trees should not be affected.

17.2-2
Can you maintain the black-heights of nodes in a red-black tree as attributes in the nodes of the tree without affecting the asymptotic performance of any of the red-black tree operations? Show how, or argue why not. How about maintaining the depths of nodes?

17.2-3
Let $\otimes$ be an associative binary operator, and let a be an attribute maintained in each node of a red-black tree. Suppose that you want to include in each node x an additional attribute f such that $x.f = x_1.a \otimes x_2.a \otimes \cdots \otimes x_m.a$, where $x_1, x_2, \ldots, x_m$ is the inorder listing of nodes in the subtree rooted at x. Show how to update the f attributes in $O(1)$ time after a rotation. Modify your argument slightly to apply it to the *size* attributes in order-statistic trees.

17.3 Interval trees

This section shows how to augment red-black trees to support operations on dynamic sets of intervals. In this section, we'll assume that intervals are closed. Extending the results to open and half-open intervals is conceptually straightforward. (See page 1157 for definitions of closed, open, and half-open intervals.)

Intervals are convenient for representing events that each occupy a continuous period of time. For example, you could query a database of time intervals to find out which events occurred during a given interval. The data structure in this section provides an efficient means for maintaining such an interval database.

A simple way to represent an interval $[t_1, t_2]$ is as an object i with attributes $i.low = t_1$ (the ***low endpoint***) and $i.high = t_2$ (the ***high endpoint***). We say that intervals i and i' ***overlap*** if $i \cap i' \neq \emptyset$, that is, if $i.low \leq i'.high$ and $i'.low \leq i.high$.

Figure 17.3 The interval trichotomy for two closed intervals i and i'. **(a)** If i and i' overlap, there are four situations, and in each, $i.low \le i'.high$ and $i'.low \le i.high$. **(b)** The intervals do not overlap, and $i.high < i'.low$. **(c)** The intervals do not overlap, and $i'.high < i.low$.

As Figure 17.3 shows, any two intervals i and i' satisfy the ***interval trichotomy***, that is, exactly one of the following three properties holds:

a. i and i' overlap,

b. i is to the left of i' (i.e., $i.high < i'.low$),

c. i is to the right of i' (i.e., $i'.high < i.low$).

An ***interval tree*** is a red-black tree that maintains a dynamic set of elements, with each element x containing an interval $x.int$. Interval trees support the following operations:

INTERVAL-INSERT(T, x) adds the element x, whose *int* attribute is assumed to contain an interval, to the interval tree T.

INTERVAL-DELETE(T, x) removes the element x from the interval tree T.

INTERVAL-SEARCH(T, i) returns a pointer to an element x in the interval tree T such that $x.int$ overlaps interval i, or a pointer to the sentinel $T.nil$ if no such element belongs to the set.

Figure 17.4 shows how an interval tree represents a set of intervals. The four-step method from Section 17.2 will guide our design of an interval tree and the operations that run on it.

Step 1: Underlying data structure

A red-black tree serves as the underlying data structure. Each node x contains an interval $x.int$. The key of x is the low endpoint, $x.int.low$, of the interval. Thus, an inorder tree walk of the data structure lists the intervals in sorted order by low endpoint.

Figure 17.4 An interval tree. **(a)** A set of 10 intervals, shown sorted bottom to top by left endpoint. **(b)** The interval tree that represents them. Each node x contains an interval, shown above the dashed line, and the maximum value of any interval endpoint in the subtree rooted at x, shown below the dashed line. An inorder tree walk of the tree lists the nodes in sorted order by left endpoint.

Step 2: Additional information

In addition to the intervals themselves, each node x contains a value $x.max$, which is the maximum value of any interval endpoint stored in the subtree rooted at x.

Step 3: Maintaining the information

We must verify that insertion and deletion take $O(\lg n)$ time on an interval tree of n nodes. It is simple enough to determine $x.max$ in $O(1)$ time, given interval $x.int$ and the max values of node x's children:

$$x.max = \max\{x.int.high, x.left.max, x.right.max\} \ .$$

Thus, by Theorem 17.1, insertion and deletion run in $O(\lg n)$ time. In fact, you can use either Exercise 17.2-3 or 17.3-1 to show how to update all the *max* attributes that change after a rotation in just $O(1)$ time.

Step 4: Developing new operations

The only new operation is INTERVAL-SEARCH(T, i), which finds a node in tree T whose interval overlaps interval i. If there is no interval in the tree that overlaps i, the procedure returns a pointer to the sentinel $T.nil$.

```
INTERVAL-SEARCH(T, i)
x = T.root
while x ≠ T.nil and i does not overlap x.int
    if x.left ≠ T.nil and x.left.max ≥ i.low
        x = x.left    // overlap in left subtree or no overlap in right subtree
    else x = x.right  // no overlap in left subtree
return x
```

The search for an interval that overlaps i starts at the root of the tree and proceeds downward. It terminates when either it finds an overlapping interval or it reaches the sentinel $T.nil$. Since each iteration of the basic loop takes $O(1)$ time, and since the height of an n-node red-black tree is $O(\lg n)$, the INTERVAL-SEARCH procedure takes $O(\lg n)$ time.

Before we see why INTERVAL-SEARCH is correct, let's examine how it works on the interval tree in Figure 17.4. Let's look for an interval that overlaps the interval $i = [22, 25]$. Begin with x as the root, which contains $[16, 21]$ and does not overlap i. Since $x.left.max = 23$ is greater than $i.low = 22$, the loop continues with x as the left child of the root—the node containing $[8, 9]$, which also does not overlap i. This time, $x.left.max = 10$ is less than $i.low = 22$, and so the loop continues with the right child of x as the new x. Because the interval $[15, 23]$ stored in this node overlaps i, the procedure returns this node.

Now let's try an unsuccessful search, for an interval that overlaps $i = [11, 14]$ in the interval tree of Figure 17.4. Again, begin with x as the root. Since the root's interval $[16, 21]$ does not overlap i, and since $x.left.max = 23$ is greater than $i.low = 11$, go left to the node containing $[8, 9]$. Interval $[8, 9]$ does not overlap i, and $x.left.max = 10$ is less than $i.low = 11$, and so the search goes right. (No interval in the left subtree overlaps i.) Interval $[15, 23]$ does not overlap i, and its left child is $T.nil$, so again the search goes right, the loop terminates, and INTERVAL-SEARCH returns the sentinel $T.nil$.

To see why INTERVAL-SEARCH is correct, we must understand why it suffices to examine a single path from the root. The basic idea is that at any node x, if $x.int$ does not overlap i, the search always proceeds in a safe direction: the search will definitely find an overlapping interval if the tree contains one. The following theorem states this property more precisely.

Theorem 17.2
Any execution of INTERVAL-SEARCH(T, i) either returns a node whose interval overlaps i, or it returns $T.nil$ and the tree T contains no node whose interval overlaps i.

Proof The **while** loop of lines 2–5 terminates when either $x = T.nil$ or i overlaps $x.int$. In the latter case, it is certainly correct to return x. Therefore, we focus on the former case, in which the **while** loop terminates because $x = T.nil$, which is the node that INTERVAL-SEARCH returns.

We'll prove that if the procedure returns $T.nil$, then it did not miss any intervals in T that overlap i. The idea is to show that whether the search goes left in line 4 or right in line 5, it always heads toward a node containing an interval overlapping i, if any such interval exists. In particular, we'll prove that

1. If the search goes left in line 4, then the left subtree of node x contains an interval that overlaps i or the right subtree of x contains no interval that overlaps i. Therefore, even if x's left subtree contains no interval that overlaps i but the search goes left, it does not make a mistake, because x's right subtree does not contain an interval overlapping i, either.
2. If the search goes right in line 5, then the left subtree of x contains no interval that overlaps i. Thus, if the search goes right, it does not make a mistake.

For both cases, we rely on the interval trichotomy. Let's start with the case where the search goes right, whose proof is simpler. By the tests in line 3, we know that $x.left = T.nil$ or $x.left.max < i.low$. If $x.left = T.nil$, then x's left subtree contains no interval that overlaps i, since it contains no intervals at all. Now suppose that $x.left \neq T.nil$, so that we must have $x.left.max < i.low$. Consider any interval i' in x's left subtree. Because $x.left.max$ is the maximum endpoint in x's left subtree, we have $i'.high \leq x.left.max$. Thus, as Figure 17.5(a) shows,

$$\begin{aligned} i'.high &\leq x.left.max \\ &< i.low . \end{aligned}$$

By the interval trichotomy, therefore, intervals i and i' do not overlap, and so x's left subtree contains no interval that overlaps i.

Now we examine the case in which the search goes left. If the left subtree of node x contains an interval that overlaps i, we're done, so let's assume that no node

Figure 17.5 Intervals in the proof of Theorem 17.2. The value of $x.left.max$ is shown in each case as a dashed line. **(a)** The search goes right. No interval i' in x's left subtree can overlap i. **(b)** The search goes left. The left subtree of x contains an interval that overlaps i (situation not shown), or x's left subtree contains an interval i' such that $i'.high = x.left.max$. Since i does not overlap i', neither does it overlap any interval i'' in x's right subtree, since $i'.low \le i''.low$.

in x's left subtree overlaps i. We need to show that in this case, no node in x's right subtree overlaps i, so that going left will not miss any overlaps in x's right subtree. By the tests in line 3, the left subtree of x is not empty and $x.left.max \ge i.low$. By the definition of the *max* attribute, x's left subtree contains some interval i' such that

$$\begin{aligned} i'.high &= x.left.max \\ &\ge i.low , \end{aligned}$$

as illustrated in Figure 17.5(b). Since i' is in x's left subtree, it does not overlap i, and since $i'.high \ge i.low$, the interval trichotomy tells us that $i.high < i'.low$. Now we bring in the property that interval trees are keyed on the low endpoints of intervals. Because i' is in x's left subtree, we have $i'.low \le x.int.low$. Now consider any interval i'' in x's right subtree, so that $x.int.low \le i''.low$. Putting inequalities together, we get

$$\begin{aligned} i.high &< i'.low \\ &\le x.int.low \\ &\le i''.low . \end{aligned}$$

Because $i.high < i''.low$, the interval trichotomy tells us that i and i'' do not overlap. Since we chose i'' as any interval in x's right subtree, no node in x's right subtree overlaps i. ■

Thus, the INTERVAL-SEARCH procedure works correctly.

Exercises

17.3-1
Write pseudocode for LEFT-ROTATE that operates on nodes in an interval tree and updates all the *max* attributes that change in $O(1)$ time.

17.3-2
Describe an efficient algorithm that, given an interval i, returns an interval overlapping i that has the minimum low endpoint, or $T.nil$ if no such interval exists.

17.3-3
Given an interval tree T and an interval i, describe how to list all intervals in T that overlap i in $O(\min\{n, k \lg n\})$ time, where k is the number of intervals in the output list. (*Hint:* One simple method makes several queries, modifying the tree between queries. A slightly more complicated method does not modify the tree.)

17.3-4
Suggest modifications to the interval-tree procedures to support the new operation INTERVAL-SEARCH-EXACTLY(T, i), where T is an interval tree and i is an interval. The operation should return a pointer to a node x in T such that $x.int.low = i.low$ and $x.int.high = i.high$, or $T.nil$ if T contains no such node. All operations, including INTERVAL-SEARCH-EXACTLY, should run in $O(\lg n)$ time on an n-node interval tree.

17.3-5
Show how to maintain a dynamic set Q of numbers that supports the operation MIN-GAP, which gives the absolute value of the difference of the two closest numbers in Q. For example, if we have $Q = \{1, 5, 9, 15, 18, 22\}$, then MIN-GAP$(Q)$ returns 3, since 15 and 18 are the two closest numbers in Q. Make the operations INSERT, DELETE, SEARCH, and MIN-GAP as efficient as possible, and analyze their running times.

★ ***17.3-6***
VLSI databases commonly represent an integrated circuit as a list of rectangles. Assume that each rectangle is rectilinearly oriented (sides parallel to the x- and y-axes), so that each rectangle is represented by four values: its minimum and maximum x- and y-coordinates. Give an $O(n \lg n)$-time algorithm to decide whether a set of n rectangles so represented contains two rectangles that overlap. Your algorithm need not report all intersecting pairs, but it must report that an overlap exists if one rectangle entirely covers another, even if the boundary lines do not intersect. (*Hint:* Move a "sweep" line across the set of rectangles.)

Problems

17-1 Point of maximum overlap

You wish to keep track of a ***point of maximum overlap*** in a set of intervals—a point with the largest number of intervals in the set that overlap it.

a. Show that there is always a point of maximum overlap that is an endpoint of one of the intervals.

b. Design a data structure that efficiently supports the operations INTERVAL-INSERT, INTERVAL-DELETE, and FIND-POM, which returns a point of maximum overlap. (*Hint:* Keep a red-black tree of all the endpoints. Associate a value of $+1$ with each left endpoint, and associate a value of -1 with each right endpoint. Augment each node of the tree with some extra information to maintain the point of maximum overlap.)

17-2 Josephus permutation

We define the ***Josephus problem*** as follows. A group of n people form a circle, and we are given a positive integer $m \le n$. Beginning with a designated first person, proceed around the circle, removing every mth person. After each person is removed, counting continues around the circle that remains. This process continues until nobody remains in the circle. The order in which the people are removed from the circle defines the ***(n, m)-Josephus permutation*** of the integers $1, 2, \ldots, n$. For example, the $(7, 3)$-Josephus permutation is $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.

a. Suppose that m is a constant. Describe an $O(n)$-time algorithm that, given an integer n, outputs the (n, m)-Josephus permutation.

b. Suppose that m is not necessarily a constant. Describe an $O(n \lg n)$-time algorithm that, given integers n and m, outputs the (n, m)-Josephus permutation.

Chapter notes

In their book, Preparata and Shamos [364] describe several of the interval trees that appear in the literature, citing work by H. Edelsbrunner (1980) and E. M. McCreight (1981). The book details an interval tree that, given a static database of n intervals, allows us to enumerate all k intervals that overlap a given query interval in $O(k + \lg n)$ time.

18 B-Trees

B-trees are balanced search trees designed to work well on disk drives or other direct-access secondary storage devices. B-trees are similar to red-black trees (Chapter 13), but they are better at minimizing the number of operations that access disks. (We often say just "disk" instead of "disk drive.") Many database systems use B-trees, or variants of B-trees, to store information.

B-trees differ from red-black trees in that B-tree nodes may have many children, from a few to thousands. That is, the "branching factor" of a B-tree can be quite large, although it usually depends on characteristics of the disk drive used. B-trees are similar to red-black trees in that every n-node B-tree has height $O(\lg n)$, so that B-trees can implement many dynamic-set operations in $O(\lg n)$ time. But a B-tree has a larger branching factor than a red-black tree, so the base of the logarithm that expresses its height is larger, and hence its height can be considerably lower.

B-trees generalize binary search trees in a natural manner. Figure 18.1 shows a simple B-tree. If an internal B-tree node x contains $x.n$ keys, then x has $x.n + 1$ children. The keys in node x serve as dividing points separating the range of keys handled by x into $x.n + 1$ subranges, each handled by one child of x. A search for a key in a B-tree makes an $(x.n + 1)$-way decision based on comparisons with the $x.n$ keys stored at node x. An internal node contains pointers to its children, but a leaf node does not.

Section 18.1 gives a precise definition of B-trees and proves that the height of a B-tree grows only logarithmically with the number of nodes it contains. Section 18.2 describes how to search for a key and insert a key into a B-tree, and Section 18.3 discusses deletion. Before proceeding, however, we need to ask why we evaluate data structures designed to work on a disk drive differently from data structures designed to work in main random-access memory.

Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $x.n$ keys has $x.n + 1$ children. All leaves are at the same depth in the tree. The blue nodes are examined in a search for the letter R.

Data structures on secondary storage

Computer systems take advantage of various technologies that provide memory capacity. The ***main memory*** of a computer system normally consists of silicon memory chips. This technology is typically more than an order of magnitude more expensive per bit stored than magnetic storage technology, such as tapes or disk drives. Most computer systems also have ***secondary storage*** based on solid-state drives (SSDs) or magnetic disk drives. The amount of such secondary storage often exceeds the amount of primary memory by one to two orders of magnitude. SSDs have faster access times than magnetic disk drives, which are mechanical devices. In recent years, SSD capacities have increased while their prices have decreased. Magnetic disk drives typically have much higher capacities than SSDs, and they remain a more cost-effective means for storing massive amounts of information. Disk drives that store several terabytes[1] can be found for under $100.

Figure 18.2 shows a typical disk drive. The drive consists of one or more ***platters***, which rotate at a constant speed around a common ***spindle***. A magnetizable material covers the surface of each platter. The drive reads and writes each platter by a ***head*** at the end of an ***arm***. The arms can move their heads toward or away from the spindle. The surface that passes underneath a given head when it is stationary is called a ***track***.

Although disk drives are cheaper and have higher capacity than main memory, they are much, much slower because they have moving mechanical parts. The mechanical motion has two components: platter rotation and arm movement. As of this writing, commodity disk drives rotate at speeds of 5400–15,000 revolutions per minute (RPM). Typical speeds are 15,000 RPM in server-grade drives, 7200 RPM

[1] When specifying disk capacities, one terabyte is one trillion bytes, rather than 2^{40} bytes.

Figure 18.2 A typical magnetic disk drive. It consists of one or more platters covered with a magnetizable material (two platters are shown here) that rotate around a spindle. Each platter is read and written with a head, shown in red, at the end of an arm. Arms rotate around a common pivot axis. A track, drawn in blue, is the surface that passes beneath the read/write head when the head is stationary.

in drives for desktops, and 5400 RPM in drives for laptops. Although 7200 RPM may seem fast, one rotation takes 8.33 milliseconds, which is over 5 orders of magnitude longer than the 50 nanosecond access times (more or less) commonly found for main memory. In other words, if a computer waits a full rotation for a particular item to come under the read/write head, it could access main memory more than 100,000 times during that span. The average wait is only half a rotation, but still, the difference in access times for main memory compared with disk drives is enormous. Moving the arms also takes some time. As of this writing, average access times for commodity disk drives are around 4 milliseconds.

In order to amortize the time spent waiting for mechanical movements, also known as ***latency***, disk drives access not just one item but several at a time. Information is divided into a number of equal-sized ***blocks*** of bits that appear consecutively within tracks, and each disk read or write is of one or more entire blocks.[2] Typical disk drives have block sizes running from 512 to 4096 bytes. Once the read/write head is positioned correctly and the platter has rotated to the beginning of the desired block, reading or writing a magnetic disk drive is entirely electronic (aside from the rotation of the platter), and the disk drive can quickly read or write large amounts of data.

---

[2] SSDs also exhibit greater latency than main memory and access data in blocks.

Often, accessing a block of information and reading it from a disk drive takes longer than processing all the information read. For this reason, in this chapter we'll look separately at the two principal components of the running time:

- the number of disk accesses, and
- the CPU (computing) time.

We measure the number of disk accesses in terms of the number of blocks of information that need to be read from or written to the disk drive. Although disk-access time is not constant—it depends on the distance between the current track and the desired track and also on the initial rotational position of the platters—the number of blocks read or written provides a good first-order approximation of the total time spent accessing the disk drive.

In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once. The B-tree algorithms copy selected blocks from disk into main memory as needed and write back onto disk the blocks that have changed. B-tree algorithms keep only a constant number of blocks in main memory at any time, and thus the size of main memory does not limit the size of B-trees that can be handled.

B-tree procedures need to be able to read information from disk into main memory and write information from main memory to disk. Consider some object x. If x is currently in the computer's main memory, then the code can refer to the attributes of x as usual: $x.key$, for example. If x resides on disk, however, then the procedure must perform the operation DISK-READ(x) to read the block containing object x into main memory before it can refer to x's attributes. (Assume that if x is already in main memory, then DISK-READ(x) requires no disk accesses: it is a "no-op.") Similarly, procedures call DISK-WRITE(x) to save any changes that have been made to the attributes of object x by writing to disk the block containing x. Thus, the typical pattern for working with an object is as follows:

```
x = a pointer to some object
DISK-READ(x)
operations that access and/or modify the attributes of x
DISK-WRITE(x)            // omitted if no attributes of x were changed
other operations that access but do not modify attributes of x
```

The system can keep only a limited number of blocks in main memory at any one time. Our B-tree algorithms assume that the system automatically flushes from main memory blocks that are no longer in use.

Since in most systems the running time of a B-tree algorithm depends primarily on the number of DISK-READ and DISK-WRITE operations it performs, we

Figure 18.3 A B-tree of height 2 containing over one billion keys. Shown inside each node x is $x.n$, the number of keys in x. Each internal node and leaf contains 1000 keys. This B-tree has 1001 nodes at depth 1 and over one million leaves at depth 2.

typically want each of these operations to read or write as much information as possible. Thus, a B-tree node is usually as large as a whole disk block, and this size limits the number of children a B-tree node can have.

Large B-trees stored on disk drives often have branching factors between 50 and 2000, depending on the size of a key relative to the size of a block. A large branching factor dramatically reduces both the height of the tree and the number of disk accesses required to find any key. Figure 18.3 shows a B-tree with a branching factor of 1001 and height 2 that can store over one billion keys. Nevertheless, if the root node is kept permanently in main memory, at most two disk accesses suffice to find any key in this tree.

18.1 Definition of B-trees

To keep things simple, let's assume, as we have for binary search trees and red-black trees, that any satellite information associated with a key resides in the same node as the key. In practice, you might actually store with each key just a pointer to another disk block containing the satellite information for that key. The pseudocode in this chapter implicitly assumes that the satellite information associated with a key, or the pointer to such satellite information, travels with the key whenever the key is moved from node to node. A common variant on a B-tree, known as a ***B^+-tree***, stores all the satellite information in the leaves and stores only keys and child pointers in the internal nodes, thus maximizing the branching factor of the internal nodes.

A ***B-tree*** T is a rooted tree with root $T.root$ having the following properties:

1. Every node x has the following attributes:

   a. $x.n$, the number of keys currently stored in node x,

   b. the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in monotonically increasing order, so that $x.key_1 \le x.key_2 \le \cdots \le x.key_{x.n}$,

   c. $x.leaf$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.

2. Each internal node x also contains $x.n + 1$ pointers $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

   $$k_1 \le x.key_1 \le k_2 \le x.key_2 \le \cdots \le x.key_{x.n} \le k_{x.n+1} .$$

4. All leaves have the same depth, which is the tree's height h.

5. Nodes have lower and upper bounds on the number of keys they can contain, expressed in terms of a fixed integer $t \ge 2$ called the ***minimum degree*** of the B-tree:

   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

   b. Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t - 1$ keys.[3]

The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and it is a ***2-3-4 tree***. In practice, however, much larger values of t yield B-trees with smaller height.

The height of a B-tree

The number of disk accesses required for most operations on a B-tree is proportional to the height of the B-tree. The following theorem bounds the worst-case height of a B-tree.

---

[3] Another common variant on a B-tree, known as a ***B*-tree***, requires each internal node to be at least 2/3 full, rather than at least half full, as a B-tree requires.

Figure 18.4 A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is $x.n$.

Theorem 18.1
If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2} .$$

Proof By definition, the root of a nonempty B-tree T contains at least one key, and all other nodes contain at least $t - 1$ keys. Let h be the height of T. Then T contains at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on, until at depth h, it has at least $2t^{h-1}$ nodes. Figure 18.4 illustrates such a tree for $h = 3$. The number n of keys therefore satisfies the inequality

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \quad \text{(by equation (A.6) on page 1142)} \\ &= 2t^h - 1 , \end{aligned}$$

so that $t^h \leq (n+1)/2$. Taking base-t logarithms of both sides proves the theorem. ■

You can see the power of B-trees as compared with red-black trees. Although the height of the tree grows as $O(\log n)$ in both cases (recall that t is a constant), for B-trees the base of the logarithm can be many times larger. Thus, B-trees save

a factor of about $\lg t$ over red-black trees in the number of nodes examined for most tree operations. Because examining an arbitrary node in a tree usually entails accessing the disk, B-trees avoid a substantial number of disk accesses.

Exercises

18.1-1
Why isn't a minimum degree of $t = 1$ allowed?

18.1-2
For what values of t is the tree of Figure 18.1 a legal B-tree?

18.1-3
Show all legal B-trees of minimum degree 2 that store the keys $1, 2, 3, 4, 5$.

18.1-4
As a function of the minimum degree t, what is the maximum number of keys that can be stored in a B-tree of height h?

18.1-5
Describe the data structure that results if each black node in a red-black tree absorbs its red children, incorporating their children with its own.

18.2 Basic operations on B-trees

This section presents the details of the operations B-TREE-SEARCH, B-TREE-CREATE, and B-TREE-INSERT. These procedures observe two conventions:

- The root of the B-tree is always in main memory, so that no procedure ever needs to perform a DISK-READ on the root. If any changes to the root node occur, however, then DISK-WRITE must be called on the root.
- Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.

The procedures are all "one-pass" algorithms that proceed downward from the root of the tree, without having to back up.

Searching a B-tree

Searching a B-tree is much like searching a binary search tree, except that instead of making a binary, or "two-way," branching decision at each node, the search

makes a multiway branching decision according to the number of the node's children. More precisely, at each internal node x, the search makes an $(x.n + 1)$-way branching decision.

The procedure B-TREE-SEARCH generalizes the TREE-SEARCH procedure defined for binary search trees on page 316. It takes as input a pointer to the root node x of a subtree and a key k to be searched for in that subtree. The top-level call is thus of the form B-TREE-SEARCH$(T.root, k)$. If k is in the B-tree, then B-TREE-SEARCH returns the ordered pair (y, i) consisting of a node y and an index i such that $y.key_i = k$. Otherwise, the procedure returns NIL.

```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3      i = i + 1
4  if i ≤ x.n and k == x.key_i
5      return (x, i)
6  elseif x.leaf
7      return NIL
8  else DISK-READ(x.c_i)
9      return B-TREE-SEARCH(x.c_i, k)
```

Using a linear-search procedure, lines 1–3 of B-TREE-SEARCH find the smallest index i such that $k \le x.key_i$, or else they set i to $x.n + 1$. Lines 4–5 check to see whether the search has discovered the key, returning if it has. Otherwise, if x is a leaf, then line 7 terminates the search unsuccessfully, and if x is an internal node, lines 8–9 recurse to search the appropriate subtree of x, after performing the necessary DISK-READ on that child. Figure 18.1 illustrates the operation of B-TREE-SEARCH. The blue nodes are those examined during a search for the key R.

As in the TREE-SEARCH procedure for binary search trees, the nodes encountered during the recursion form a simple path downward from the root of the tree. The B-TREE-SEARCH procedure therefore accesses $O(h) = O(\log_t n)$ disk blocks, where h is the height of the B-tree and n is the number of keys in the B-tree. Since $x.n < 2t$, the **while** loop of lines 2–3 takes $O(t)$ time within each node, and the total CPU time is $O(th) = O(t \log_t n)$.

Creating an empty B-tree

To build a B-tree T, first use the B-TREE-CREATE procedure on the next page to create an empty root node and then call the B-TREE-INSERT procedure on

page 508 to add new keys. Both of these procedures use an auxiliary procedure ALLOCATE-NODE, whose pseudocode we omit and which allocates one disk block to be used as a new node in $O(1)$ time. A node created by ALLOCATE-NODE requires no DISK-READ, since there is as yet no useful information stored on the disk for that node. B-TREE-CREATE requires $O(1)$ disk operations and $O(1)$ CPU time.

```
B-TREE-CREATE(T)
x = ALLOCATE-NODE()
x.leaf = TRUE
x.n = 0
DISK-WRITE(x)
T.root = x
```

Inserting a key into a B-tree

Inserting a key into a B-tree is significantly more complicated than inserting a key into a binary search tree. As with binary search trees, you search for the leaf position at which to insert the new key. With a B-tree, however, you cannot simply create a new leaf node and insert it, as the resulting tree would fail to be a valid B-tree. Instead, you insert the new key into an existing leaf node. Since you cannot insert a key into a leaf node that is full, you need an operation that ***splits*** a full node y (having $2t - 1$ keys) around its ***median key*** $y.key_t$ into two nodes having only $t - 1$ keys each. The median key moves up into y's parent to identify the dividing point between the two new trees. But if y's parent is also full, you must split it before you can insert the new key, and thus you could end up splitting full nodes all the way up the tree.

To avoid having to go back up the tree, just split every full node you encounter as you go down the tree. In this way, whenever you need to split a full node, you are assured that its parent is not full. Inserting a key into a B-tree then requires only a single pass down the tree from the root to a leaf.

Splitting a node in a B-tree

The procedure B-TREE-SPLIT-CHILD on the facing page takes as input a *nonfull* internal node x (assumed to reside in main memory) and an index i such that $x.c_i$ (also assumed to reside in main memory) is a *full* child of x. The procedure splits this child in two and adjusts x so that it has an additional child. To split a full root, you first need to make the root a child of a new empty root node, so that you can

use B-TREE-SPLIT-CHILD. The tree thus grows in height by 1: splitting is the only means by which the tree grows taller.

```
B-TREE-SPLIT-CHILD(x, i)
1  y = x.c_i                          // full node to split
2  z = ALLOCATE-NODE()                // z will take half of y
3  z.leaf = y.leaf
4  z.n = t - 1
5  for j = 1 to t - 1                 // z gets y's greatest keys ...
6      z.key_j = y.key_{j+t}
7  if not y.leaf
8      for j = 1 to t                 // ... and its corresponding children
9          z.c_j = y.c_{j+t}
10 y.n = t - 1                        // y keeps t - 1 keys
11 for j = x.n + 1 downto i + 1       // shift x's children to the right ...
12     x.c_{j+1} = x.c_j
13 x.c_{i+1} = z                      // ... to make room for z as a child
14 for j = x.n downto i               // shift the corresponding keys in x
15     x.key_{j+1} = x.key_j
16 x.key_i = y.key_t                  // insert y's median key
17 x.n = x.n + 1                      // x has gained a child
18 DISK-WRITE(y)
19 DISK-WRITE(z)
20 DISK-WRITE(x)
```

Figure 18.5 illustrates how a node splits. B-TREE-SPLIT-CHILD splits the full node $y = x.c_i$ about its median key (S in the figure), which moves up into y's parent node x. Those keys in y that are greater than the median key move into a new node z, which becomes a new child of x.

B-TREE-SPLIT-CHILD works by straightforward cutting and pasting. Node x is the parent of the node y being split, which is x's ith child (set in line 1). Node y originally has $2t$ children and $2t - 1$ keys, but splitting reduces y to t children and $t - 1$ keys. The t largest children and $t - 1$ keys of node y move over to node z, which becomes a new child of x, positioned just after y in x's table of children. The median key of y moves up to become the key in node x that separates the pointers to nodes y and z.

Lines 2–9 create node z and give it the largest $t - 1$ keys and, if y and z are internal nodes, the corresponding t children of y. Line 10 adjusts the key count for y. Then, lines 11–17 shift keys and child pointers in x to the right in order to make room for x's new child, insert z as a new child of x, move the median key

Figure 18.5 Splitting a node with $t = 4$. Node $y = x.c_i$ splits into two nodes, y and z, and the median key S of y moves up into y's parent.

from y up to x in order to separate y from z, and adjust x's key count. Lines 18–20 write out all modified disk blocks. The CPU time used by B-TREE-SPLIT-CHILD is $\Theta(t)$, due to the **for** loops in lines 5–6 and 8–9. (The **for** loops in lines 11–12 and 14–15 also run for $O(t)$ iterations.) The procedure performs $O(1)$ disk operations.

Inserting a key into a B-tree in a single pass down the tree

Inserting a key k into a B-tree T of height h requires just a single pass down the tree and $O(h)$ disk accesses. The CPU time required is $O(th) = O(t \log_t n)$. The B-TREE-INSERT procedure uses B-TREE-SPLIT-CHILD to guarantee that the recursion never descends to a full node. If the root is full, B-TREE-INSERT splits it by calling the procedure B-TREE-SPLIT-ROOT on the facing page.

```
B-TREE-INSERT(T, k)
1  r = T.root
2  if r.n == 2t − 1
3      s = B-TREE-SPLIT-ROOT(T)
4      B-TREE-INSERT-NONFULL(s, k)
5  else B-TREE-INSERT-NONFULL(r, k)
```

B-TREE-INSERT works as follows. If the root is full, then line 3 calls B-TREE-SPLIT-ROOT in line 3 to split it. A new node s (with two children) becomes the root and is returned by B-TREE-SPLIT-ROOT. Splitting the root, illustrated in Figure 18.6, is the only way to increase the height of a B-tree. Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom. Regardless of whether the root split, B-TREE-INSERT finishes by calling B-TREE-INSERT-NONFULL to insert key k into the tree rooted at the nonfull root node,

Figure 18.6 Splitting the root with $t = 4$. Root node r splits in two, and a new root node s is created. The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split. A B-tree's height increases only when the root splits.

which is either the new root (the call in line 4) or the original root (the call in line 5).

```
B-TREE-SPLIT-ROOT(T)
1  s = ALLOCATE-NODE()
2  s.leaf = FALSE
3  s.n = 0
4  s.c1 = T.root
5  T.root = s
6  B-TREE-SPLIT-CHILD(s, 1)
7  return s
```

The auxiliary procedure B-TREE-INSERT-NONFULL on page 511 inserts key k into node x, which is assumed to be nonfull when the procedure is called. B-TREE-INSERT-NONFULL recurses as necessary down the tree, at all times guaranteeing that the node to which it recurses is not full by calling B-TREE-SPLIT-CHILD as necessary. The operation of B-TREE-INSERT and the recursive operation of B-TREE-INSERT-NONFULL guarantee that this assumption is true.

Figure 18.7 illustrates the various cases of how B-TREE-INSERT-NONFULL inserts a key into a B-tree. Lines 3–8 handle the case in which x is a leaf node by inserting key k into x, shifting to the right all keys in x that are greater than k. If x is not a leaf node, then k should go into the appropriate leaf node in the subtree rooted at internal node x. Lines 9–11 determine the child $x.c_i$ to which the recursion descends. Line 13 detects whether the recursion would descend to a full child, in which case line 14 calls B-TREE-SPLIT-CHILD to split that child into two non-

Figure 18.7 Inserting keys into a B-tree. The minimum degree t for this B-tree is 3, so that a node can hold at most 5 keys. Blue nodes are modified by the insertion process. **(a)** The initial tree for this example. **(b)** The result of inserting B into the initial tree. This case is a simple insertion into a leaf node. **(c)** The result of inserting Q into the previous tree. The node $RSTUV$ splits into two nodes containing RS and UV, the key T moves up to the root, and Q is inserted in the leftmost of the two halves (the RS node). **(d)** The result of inserting L into the previous tree. The root splits right away, since it is full, and the B-tree grows in height by one. Then L is inserted into the leaf containing JK. **(e)** The result of inserting F into the previous tree. The node $ABCDE$ splits before F is inserted into the rightmost of the two halves (the DE node).

```
B-TREE-INSERT-NONFULL(x, k)
1  i = x.n
2  if x.leaf                                  // inserting into a leaf?
3      while i ≥ 1 and k < x.key_i            // shift keys in x to make room for k
4          x.key_{i+1} = x.key_i
5          i = i − 1
6      x.key_{i+1} = k                        // insert key k in x
7      x.n = x.n + 1                          // now x has 1 more key
8      DISK-WRITE(x)
9  else while i ≥ 1 and k < x.key_i           // find the child where k belongs
10         i = i − 1
11     i = i + 1
12     DISK-READ(x.c_i)
13     if x.c_i.n == 2t − 1                   // split the child if it's full
14         B-TREE-SPLIT-CHILD(x, i)
15         if k > x.key_i                     // does k go into x.c_i or x.c_{i+1}?
16             i = i + 1
17     B-TREE-INSERT-NONFULL(x.c_i, k)
```

full children, and lines 15–16 determine which of the two children is the correct one to descend to. (Note that DISK-READ$(x.c_i)$ is not needed after line 16 increments i, since the recursion descends in this case to a child that was just created by B-TREE-SPLIT-CHILD.) The net effect of lines 13–16 is thus to guarantee that the procedure never recurses to a full node. Line 17 then recurses to insert k into the appropriate subtree.

For a B-tree of height h, B-TREE-INSERT performs $O(h)$ disk accesses, since only $O(1)$ DISK-READ and DISK-WRITE operations occur at each level of the tree. The total CPU time used is $O(t)$ in each level of the tree, or $O(th) = O(t \log_t n)$ overall. Since B-TREE-INSERT-NONFULL is tail-recursive, you can instead implement it with a **while** loop, thereby demonstrating that the number of blocks that need to be in main memory at any time is $O(1)$.

Exercises

18.2-1

Show the results of inserting the keys

$$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$$

in order into an empty B-tree with minimum degree 2. Draw only the configurations of the tree just before some node must split, and also draw the final configuration.

18.2-2
Explain under what circumstances, if any, redundant DISK-READ or DISK-WRITE operations occur during the course of executing a call to B-TREE-INSERT. (A redundant DISK-READ is a DISK-READ for a block that is already in memory. A redundant DISK-WRITE writes to disk a block of information that is identical to what is already stored there.)

18.2-3
Professor Bunyan asserts that the B-TREE-INSERT procedure always results in a B-tree with the minimum possible height. Show that the professor is mistaken by proving that with $t = 2$ and the set of keys $\{1, 2, \ldots, 15\}$, there is no insertion sequence that results in a B-tree with the minimum possible height.

★ ***18.2-4***
If you insert the keys $\{1, 2, \ldots, n\}$ into an empty B-tree with minimum degree 2, how many nodes does the final B-tree have?

18.2-5
Since leaf nodes require no pointers to children, they could conceivably use a different (larger) t value than internal nodes for the same disk block size. Show how to modify the procedures for creating and inserting into a B-tree to handle this variation.

18.2-6
Suppose that you implement B-TREE-SEARCH to use binary search rather than linear search within each node. Show that this change makes the required CPU time $O(\lg n)$, independent of how t might be chosen as a function of n.

18.2-7
Suppose that disk hardware allows you to choose the size of a disk block arbitrarily, but that the time it takes to read the disk block is $a + bt$, where a and b are specified constants and t is the minimum degree for a B-tree using blocks of the selected size. Describe how to choose t so as to minimize (approximately) the B-tree search time. Suggest an optimal value of t for the case in which $a = 5$ milliseconds and $b = 10$ microseconds.

18.3 Deleting a key from a B-tree

Deletion from a B-tree is analogous to insertion but a little more complicated, because you can delete a key from any node—not just a leaf—and when you delete a key from an internal node, you must rearrange the node's children. As in insertion, you must guard against deletion producing a tree whose structure violates the B-tree properties. Just as a node should not get too big due to insertion, a node must not get too small during deletion (except that the root is allowed to have fewer than the minimum number $t - 1$ of keys). And just as a simple insertion algorithm might have to back up if a node on the path to where the key is to be inserted is full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The procedure B-TREE-DELETE deletes the key k from the subtree rooted at x. Unlike the procedures TREE-DELETE on page 325 and RB-DELETE on page 348, which are given the node to delete—presumably as the result of a prior search—B-TREE-DELETE combines the search for key k with the deletion process. Why do we combine search and deletion in B-TREE-DELETE? Just as B-TREE-INSERT prevents any node from becoming overfull (having more than $2t - 1$ keys) while making a single pass down the tree, B-TREE-DELETE prevents any node from becoming underfull (having fewer than $t - 1$ keys) while also making a single pass down the tree, searching for and ultimately deleting the key.

To prevent any node from becoming underfull, the design of B-TREE-DELETE guarantees that whenever it calls itself recursively on a node x, the number of keys in x is at least the minimum degree t at the time of the call. (Although the root may have fewer than t keys and a recursive call may be made *from* the root, no recursive call is made *on* the root.) This condition requires one more key than the minimum required by the usual B-tree conditions, and so a key might have to be moved from x into one of its child nodes (still leaving x with at least the minimum $t - 1$ keys) before a recursive call is made on that child, thus allowing deletion to occur in one downward pass without having to traverse back up the tree.

We describe how the procedure B-TREE-DELETE(T, k) deletes a key k from a B-tree T instead of presenting detailed pseudocode. We examine three cases, illustrated in Figure 18.8. The cases are for when the search arrives at a leaf, at an internal node containing key k, and at an internal node not containing key k. As mentioned above, in all three cases node x has at least t keys (with the possible exception of when x is the root). Cases 2 and 3—when x is an internal node—guarantee this property as the recursion descends through the B-tree.

Figure 18.8 Deleting keys from a B-tree. The minimum degree for this B-tree is $t = 3$, so that, other than the root, every node must have at least 2 keys. Blue nodes are those that are modified by the deletion process. **(a)** The B-tree of Figure 18.7(e). **(b)** Deletion of F, which is case 1: simple deletion from a leaf when all nodes visited during the search (other than the root) have at least $t = 3$ keys. **(c)** Deletion of M, which is case 2a: the predecessor L of M moves up to take M's position.

Case 1: *The search arrives at a leaf node x.* If x contains key k, then delete k from x. If x does not contain key k, then k was not in the B-tree and nothing else needs to be done.

Case 2: *The search arrives at an internal node x that contains key k.* Let $k = x.key_i$. One of the following three cases applies, depending on the number of keys in $x.c_i$ (the child of x that precedes k) and $x.c_{i+1}$ (the child of x that follows k).

Case 2a: *$x.c_i$ has at least t keys.* Find the predecessor k' of k in the subtree rooted at $x.c_i$. Recursively delete k' from $x.c_i$, and replace k by k' in x. (Key k' can be found and deleted in a single downward pass.)

Case 2b: *$x.c_i$ has $t - 1$ keys and $x.c_{i+1}$ has at least t keys.* This case is symmetric to case 2a. Find the successor k' of k in the subtree rooted at $x.c_{i+1}$.

Figure 18.8, continued **(d)** Deletion of G, which is case 2c: push G down to make node $DEGJK$ and then delete G from this leaf (case 1). **(e)** Deletion of D, which is case 3b: since the recursion cannot descend to node CL because it has only 2 keys, push P down and merge it with CL and TX to form $CLPTX$. Then delete D from a leaf (case 1). **(e′)** After (e), delete the empty root. The tree shrinks in height by 1. **(f)** Deletion of B, which is case 3a: C moves to fill B's position and E moves to fill C's position.

Recursively delete k' from $x.c_{i+1}$, and replace k by k' in x. (Again, finding and deleting k' can be done in a single downward pass.)

Case 2c: *Both $x.c_i$ and $x.c_{i+1}$ have $t-1$ keys.* Merge k and all of $x.c_{i+1}$ into $x.c_i$, so that x loses both k and the pointer to $x.c_{i+1}$, and $x.c_i$ now contains $2t-1$ keys. Then free $x.c_{i+1}$ and recursively delete k from $x.c_i$.

Case 3: *The search arrives at an internal node x that does not contain key k.* Continue searching down the tree while ensuring that each node visited has at least t keys. To do so, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ has only $t-1$ keys, execute

case 3a or 3b as necessary to guarantee descending to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

Case 3a: *$x.c_i$ has only $t - 1$ keys but has an immediate sibling with at least t keys.* Give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.

Case 3b: *$x.c_i$ and each of $x.c_i$'s immediate siblings have $t - 1$ keys.* (It is possible for $x.c_i$ to have either one or two siblings.) Merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

In cases 2c and 3b, if node x is the root, it could end up having no keys. When this situation occurs, then x is deleted, and x's only child $x.c_1$ becomes the new root of the tree. This action decreases the height of the tree by one and preserves the property that the root of the tree contains at least one key (unless the tree is empty).

Since most of the keys in a B-tree are in the leaves, deletion operations often end up deleting keys from leaves. The B-TREE-DELETE procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node x, however, the procedure might make a downward pass through the tree to find the key's predecessor or successor and then return to node x to replace the key with its predecessor or successor (cases 2a and 2b). Returning to node x does not require a traversal through all the levels between x and the node containing the predecessor or successor, however, since the procedure can just keep a pointer to x and the key position within x and put the predecessor or successor key directly there.

Although this procedure seems complicated, it involves only $O(h)$ disk operations for a B-tree of height h, since only $O(1)$ calls to DISK-READ and DISK-WRITE are made between recursive invocations of the procedure. The CPU time required is $O(th) = O(t \log_t n)$.

Exercises

18.3-1

Show the results of deleting C, P, and V, in order, from the tree of Figure 18.8(f).

18.3-2

Write pseudocode for B-TREE-DELETE.

Problems

18-1 Stacks on secondary storage

Consider implementing a stack in a computer that has a relatively small amount of fast primary memory and a relatively large amount of slower disk storage. The operations PUSH and POP work on single-word values. The stack can grow to be much larger than can fit in memory, and thus most of it must be stored on disk.

A simple, but inefficient, stack implementation keeps the entire stack on disk. Maintain in memory a stack pointer, which is the disk address of the top element on the stack. Indexing block numbers and word offsets within blocks from 0, if the pointer has value p, the top element is the $(p \bmod m)$th word on block $\lfloor p/m \rfloor$ of the disk, where m is the number of words per block.

To implement the PUSH operation, increment the stack pointer, read the appropriate block into memory from disk, copy the element to be pushed to the appropriate word on the block, and write the block back to disk. A POP operation is similar. Read in the appropriate block from disk, save the top of the stack, decrement the stack pointer, and return the saved value. You need not write back the block, since it was not modified, and the word in the block that contained the popped value is ignored.

As in the analyses of B-tree operations, two costs matter: the total number of disk accesses and the total CPU time. A disk access also incurs a cost in CPU time. In particular, any disk access to a block of m words incurs charges of one disk access and $\Theta(m)$ CPU time.

a. Asymptotically, what is the worst-case number of disk accesses for n stack operations using this simple implementation? What is the CPU time for n stack operations? Express your answer in terms of m and n for this and subsequent parts.

Now consider a stack implementation in which you keep one block of the stack in memory. (You also maintain a small amount of memory to record which block is currently in memory.) You can perform a stack operation only if the relevant disk block resides in memory. If necessary, you can write the block currently in memory to the disk and read the new block from the disk into memory. If the relevant disk block is already in memory, then no disk accesses are required.

b. What is the worst-case number of disk accesses required for n PUSH operations? What is the CPU time?

c. What is the worst-case number of disk accesses required for n stack operations? What is the CPU time?

Suppose that you now implement the stack by keeping two blocks in memory (in addition to a small number of words for bookkeeping).

d. Describe how to manage the stack blocks so that the amortized number of disk accesses for any stack operation is $O(1/m)$ and the amortized CPU time for any stack operation is $O(1)$.

18-2 Joining and splitting 2-3-4 trees

The ***join*** operation takes two dynamic sets S' and S'' and an element x such that $x'.key < x.key < x''.key$ for any $x' \in S'$ and $x'' \in S''$. It returns a set $S = S' \cup \{x\} \cup S''$. The ***split*** operation is like an "inverse" join: given a dynamic set S and an element $x \in S$, it creates a set S' that consists of all elements in $S - \{x\}$ whose keys are less than $x.key$ and another set S'' that consists of all elements in $S - \{x\}$ whose keys are greater than $x.key$. This problem investigates how to implement these operations on 2-3-4 trees (B-trees with $t = 2$). Assume for convenience that elements consist only of keys and that all key values are distinct.

a. Show how to maintain, for every node x of a 2-3-4 tree, the height of the subtree rooted at x as an attribute $x.height$. Make sure that your implementation does not affect the asymptotic running times of searching, insertion, and deletion.

b. Show how to implement the join operation. Given two 2-3-4 trees T' and T'' and a key k, the join operation should run in $O(1 + |h' - h''|)$ time, where h' and h'' are the heights of T' and T'', respectively.

c. Consider the simple path p from the root of a 2-3-4 tree T to a given key k, the set S' of keys in T that are less than k, and the set S'' of keys in T that are greater than k. Show that p breaks S' into a set of trees $\{T'_0, T'_1, \ldots, T'_m\}$ and a set of keys $\{k'_1, k'_2, \ldots, k'_m\}$ such that $y < k'_i < z$ for $i = 1, 2, \ldots, m$ and any keys $y \in T'_{i-1}$ and $z \in T'_i$. What is the relationship between the heights of T'_{i-1} and T'_i? Describe how p breaks S'' into sets of trees and keys.

d. Show how to implement the split operation on T. Use the join operation to assemble the keys in S' into a single 2-3-4 tree T' and the keys in S'' into a single 2-3-4 tree T''. The running time of the split operation should be $O(\lg n)$, where n is the number of keys in T. (*Hint:* The costs for joining should telescope.)

Chapter notes

Knuth [261], Aho, Hopcroft, and Ullman [5], and Sedgewick and Wayne [402] give further discussions of balanced-tree schemes and B-trees. Comer [99] provides a comprehensive survey of B-trees. Guibas and Sedgewick [202] discuss the relationships among various kinds of balanced-tree schemes, including red-black trees and 2-3-4 trees.

In 1970, J. E. Hopcroft invented 2-3 trees, a precursor to B-trees and 2-3-4 trees, in which every internal node has either two or three children. Bayer and McCreight [39] introduced B-trees in 1972 with no explanation of their choice of name.

Bender, Demaine, and Farach-Colton [47] studied how to make B-trees perform well in the presence of memory-hierarchy effects. Their ***cache-oblivious*** algorithms work efficiently without explicitly knowing the data transfer sizes within the memory hierarchy.

19 Data Structures for Disjoint Sets

Some applications involve grouping n distinct elements into a collection of disjoint sets—sets with no elements in common. These applications often need to perform two operations in particular: finding the unique set that contains a given element and uniting two sets. This chapter explores methods for maintaining a data structure that supports these operations.

Section 19.1 describes the operations supported by a disjoint-set data structure and presents a simple application. Section 19.2 looks at a simple linked-list implementation for disjoint sets. Section 19.3 presents a more efficient representation using rooted trees. The running time using the tree representation is theoretically superlinear, but for all practical purposes it is linear. Section 19.4 defines and discusses a very quickly growing function and its very slowly growing inverse, which appears in the running time of operations on the tree-based implementation, and then, by a complex amortized analysis, proves an upper bound on the running time that is just barely superlinear.

19.1 Disjoint-set operations

A ***disjoint-set data structure*** maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. To identify each set, choose a ***representative***, which is some member of the set. In some applications, it doesn't matter which member is used as the representative; it matters only that if you ask for the representative of a dynamic set twice without modifying the set between the requests, you get the same answer both times. Other applications may require a prespecified rule for choosing the representative, such as choosing the smallest member in the set (for a set whose elements can be ordered).

As in the other dynamic-set implementations we have studied, each element of a set is represented by an object. Letting x denote an object, we'll see how to support the following operations:

MAKE-SET(x), where x does not already belong to some other set, creates a new set whose only member (and thus representative) is x.

UNION(x, y) unites two disjoint, dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. The representative of the resulting set is any member of $S_x \cup S_y$, although many implementations of UNION specifically choose the representative of either S_x or S_y as the new representative. Since the sets in the collection must at all times be disjoint, the UNION operation destroys sets S_x and S_y, removing them from the collection $\mathcal{S}$. In practice, implementations often absorb the elements of one of the sets into the other set.

FIND-SET(x) returns a pointer to the representative of the unique set containing x.

Throughout this chapter, we'll analyze the running times of disjoint-set data structures in terms of two parameters: n, the number of MAKE-SET operations, and m, the total number of MAKE-SET, UNION, and FIND-SET operations. Because the total number of operations m includes the n MAKE-SET operations, $m \geq n$. The first n operations are always MAKE-SET operations, so that after the first n operations, the collection consists of n singleton sets. Since the sets are disjoint at all times, each UNION operation reduces the number of sets by 1. After $n - 1$ UNION operations, therefore, only one set remains, and so at most $n - 1$ UNION operations can occur.

An application of disjoint-set data structures

One of the many applications of disjoint-set data structures arises in determining the connected components of an undirected graph (see Section B.4). Figure 19.1(a), for example, shows a graph with four connected components.

The procedure CONNECTED-COMPONENTS on the following page uses the disjoint-set operations to compute the connected components of a graph. Once the CONNECTED-COMPONENTS procedure has preprocessed the graph, the procedure SAME-COMPONENT answers queries about whether two vertices belong to the same connected component. In pseudocode, we denote the set of vertices of a graph G by $G.V$ and the set of edges by $G.E$.

The procedure CONNECTED-COMPONENTS initially places each vertex v in its own set. Then, for each edge (u, v), it unites the sets containing u and v. By Exercise 19.1-2, after all the edges are processed, two vertices belong to the same connected component if and only if the objects corresponding to the vertices belong

(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, f)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(h, i)	$\{a, c\}$	$\{b, d\}$			$\{e, f\}$		$\{g\}$	$\{h, i\}$		$\{j\}$
(a, b)	$\{a, b, c, d\}$				$\{e, f\}$		$\{g\}$	$\{h, i\}$		$\{j\}$
(f, g)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$
(b, c)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$

(b)

Figure 19.1 **(a)** A graph with four connected components: $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$, and $\{j\}$. **(b)** The collection of disjoint sets after processing each edge.

```
CONNECTED-COMPONENTS(G)
for each vertex v ∈ G.V
    MAKE-SET(v)
for each edge (u, v) ∈ G.E
    if FIND-SET(u) ≠ FIND-SET(v)
        UNION(u, v)

SAME-COMPONENT(u, v)
if FIND-SET(u) == FIND-SET(v)
    return TRUE
else return FALSE
```

to the same set. Thus CONNECTED-COMPONENTS computes sets in such a way that the procedure SAME-COMPONENT can determine whether two vertices are in the same connected component. Figure 19.1(b) illustrates how CONNECTED-COMPONENTS computes the disjoint sets.

In an actual implementation of this connected-components algorithm, the representations of the graph and the disjoint-set data structure would need to reference each other. That is, an object representing a vertex would contain a pointer to the corresponding disjoint-set object, and vice versa. Since these programming details depend on the implementation language, we do not address them further here.

When the edges of the graph are static—not changing over time—depth-first search can compute the connected components faster (see Exercise 20.3-12 on

page 572). Sometimes, however, the edges are added dynamically, with the connected components updated as each edge is added. In this case, the implementation given here can be more efficient than running a new depth-first search for each new edge.

Exercises

19.1-1
The CONNECTED-COMPONENTS procedure is run on the undirected graph $G = (V, E)$, where $V = \{a, b, c, d, e, f, g, h, i, j, k\}$, and the edges of E are processed in the order $(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e)$. List the vertices in each connected component after each iteration of lines 3–5.

19.1-2
Show that after all edges are processed by CONNECTED-COMPONENTS, two vertices belong to the same connected component if and only if they belong to the same set.

19.1-3
During the execution of CONNECTED-COMPONENTS on an undirected graph $G = (V, E)$ with k connected components, how many times is FIND-SET called? How many times is UNION called? Express your answers in terms of $|V|$, $|E|$, and k.

19.2 Linked-list representation of disjoint sets

Figure 19.2(a) shows a simple way to implement a disjoint-set data structure: each set is represented by its own linked list. The object for each set has attributes *head*, pointing to the first object in the list, and *tail*, pointing to the last object. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Within each linked list, the objects may appear in any order. The representative is the set member in the first object in the list.

With this linked-list representation, both MAKE-SET and FIND-SET require only $O(1)$ time. To carry out MAKE-SET(x), create a new linked list whose only object is x. For FIND-SET(x), just follow the pointer from x back to its set object and then return the member in the object that *head* points to. For example, in Figure 19.2(a), the call FIND-SET(g) returns f.

Figure 19.2 **(a)** Linked-list representations of two sets. Set S_1 contains members d, f, and g, with representative f, and set S_2 contains members b, c, e, and h, with representative c. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers *head* and *tail* to the first and last objects, respectively. **(b)** The result of UNION(g, e), which appends the linked list containing e to the linked list containing g. The representative of the resulting set is f. The set object for e's list, S_2, is destroyed.

A simple implementation of union

The simplest implementation of the UNION operation using the linked-list set representation takes significantly more time than MAKE-SET or FIND-SET. As Figure 19.2(b) shows, the operation UNION(x, y) appends y's list onto the end of x's list. The representative of x's list becomes the representative of the resulting set. To quickly find where to append y's list, use the *tail* pointer for x's list. Because all members of y's list join x's list, the UNION operation destroys the set object for y's list. The UNION operation is where this implementation pays the price for FIND-SET taking constant time: UNION must also update the pointer to the set object for each object originally on y's list, which takes time linear in the length of y's list. In Figure 19.2, for example, the operation UNION(g, e) causes pointers to be updated in the objects for b, c, e, and h.

In fact, we can construct a sequence of m operations on n objects that requires $\Theta(n^2)$ time. Starting with objects $x_1, x_2, \ldots, x_n$, execute the sequence of n MAKE-SET operations followed by $n - 1$ UNION operations shown in Figure 19.3, so that $m = 2n - 1$. The n MAKE-SET operations take $\Theta(n)$ time. Because the ith UNION operation updates i objects, the total number of objects updated by all $n - 1$ UNION operations forms an arithmetic series:

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
$\vdots$	$\vdots$
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
$\vdots$	$\vdots$
UNION(x_n, x_{n-1})	$n-1$

Figure 19.3 A sequence of $2n-1$ operations on n objects that takes $\Theta(n^2)$ time, or $\Theta(n)$ time per operation on average, using the linked-list set representation and the simple implementation of UNION.

$$\sum_{i=1}^{n-1} i = \Theta(n^2) .$$

The total number of operations is $2n-1$, and so each operation on average requires $\Theta(n)$ time. That is, the amortized time of an operation is $\Theta(n)$.

A weighted-union heuristic

In the worst case, the above implementation of UNION requires an average of $\Theta(n)$ time per call, because it might be appending a longer list onto a shorter list, and the procedure must update the pointer to the set object for each member of the longer list. Suppose instead that each list also includes the length of the list (which can be maintained straightforwardly with constant overhead) and that the UNION procedure always appends the shorter list onto the longer, breaking ties arbitrarily. With this simple ***weighted-union heuristic***, a single UNION operation can still take $\Omega(n)$ time if both sets have $\Omega(n)$ members. As the following theorem shows, however, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes $O(m + n \lg n)$ time.

Theorem 19.1
Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes $O(m + n \lg n)$ time.

Proof Because each UNION operation unites two disjoint sets, at most $n-1$ UNION operations occur over all. We now bound the total time taken by these

UNION operations. We start by determining, for each object, an upper bound on the number of times the object's pointer back to its set object is updated. Consider a particular object x. Each time x's pointer is updated, x must have started in the smaller set. The first time x's pointer is updated, therefore, the resulting set must have at least 2 members. Similarly, the next time x's pointer is updated, the resulting set must have had at least 4 members. Continuing on, for any $k \le n$, after x's pointer has been updated $\lceil \lg k \rceil$ times, the resulting set must have at least k members. Since the largest set has at most n members, each object's pointer is updated at most $\lceil \lg n \rceil$ times over all the UNION operations. Thus the total time spent updating object pointers over all UNION operations is $O(n \lg n)$. We must also account for updating the *tail* pointers and the list lengths, which take only $\Theta(1)$ time per UNION operation. The total time spent in all UNION operations is thus $O(n \lg n)$.

The time for the entire sequence of m operations follows. Each MAKE-SET and FIND-SET operation takes $O(1)$ time, and there are $O(m)$ of them. The total time for the entire sequence is thus $O(m + n \lg n)$. ■

Exercises

19.2-1

Write pseudocode for MAKE-SET, FIND-SET, and UNION using the linked-list representation and the weighted-union heuristic. Make sure to specify the attributes that you assume for set objects and list objects.

19.2-2

Show the data structure that results and the answers returned by the FIND-SET operations in the following program. Use the linked-list representation with the weighted-union heuristic. Assume that if the sets containing x_i and x_j have the same size, then the operation UNION(x_i, x_j) appends x_j's list onto x_i's list.

```
for i = 1 to 16
    MAKE-SET(x_i)
for i = 1 to 15 by 2
    UNION(x_i, x_{i+1})
for i = 1 to 13 by 4
    UNION(x_i, x_{i+2})
UNION(x_1, x_5)
UNION(x_11, x_13)
UNION(x_1, x_10)
FIND-SET(x_2)
FIND-SET(x_9)
```

19.2-3
Adapt the aggregate proof of Theorem 19.1 to obtain amortized time bounds of $O(1)$ for MAKE-SET and FIND-SET and $O(\lg n)$ for UNION using the linked-list representation and the weighted-union heuristic.

19.2-4
Give a tight asymptotic bound on the running time of the sequence of operations in Figure 19.3 assuming the linked-list representation and the weighted-union heuristic.

19.2-5
Professor Gompers suspects that it might be possible to keep just one pointer in each set object, rather than two (*head* and *tail*), while keeping the number of pointers in each list element at two. Show that the professor's suspicion is well founded by describing how to represent each set by a linked list such that each operation has the same running time as the operations described in this section. Describe also how the operations work. Your scheme should allow for the weighted-union heuristic, with the same effect as described in this section. (*Hint:* Use the tail of a linked list as its set's representative.)

19.2-6
Suggest a simple change to the UNION procedure for the linked-list representation that removes the need to keep the *tail* pointer to the last object in each list. Regardless of whether the weighted-union heuristic is used, your change should not change the asymptotic running time of the UNION procedure. (*Hint:* Rather than appending one list to another, splice them together.)

19.3 Disjoint-set forests

A faster implementation of disjoint sets represents sets by rooted trees, with each node containing one member and each tree representing one set. In a ***disjoint-set forest***, illustrated in Figure 19.4(a), each member points only to its parent. The root of each tree contains the representative and is its own parent. As we'll see, although the straightforward algorithms that use this representation are no faster than ones that use the linked-list representation, two heuristics—"union by rank" and "path compression"—yield an asymptotically optimal disjoint-set data structure.

The three disjoint-set operations have simple implementations. A MAKE-SET operation simply creates a tree with just one node. A FIND-SET operation follows parent pointers until it reaches the root of the tree. The nodes visited on this sim-

Figure 19.4 A disjoint-set forest. **(a)** Trees representing the two sets of Figure 19.2. The tree on the left represents the set $\{b, c, e, h\}$, with c as the representative, and the tree on the right represents the set $\{d, f, g\}$, with f as the representative. **(b)** The result of UNION(e, g).

ple path toward the root constitute the ***find path***. A UNION operation, shown in Figure 19.4(b), simply causes the root of one tree to point to the root of the other.

Heuristics to improve the running time

So far, disjoint-set forests have not improved on the linked-list implementation. A sequence of $n - 1$ UNION operations could create a tree that is just a linear chain of n nodes. By using two heuristics, however, we can achieve a running time that is almost linear in the total number m of operations.

The first heuristic, ***union by rank***, is similar to the weighted-union heuristic we used with the linked-list representation. The common-sense approach is to make the root of the tree with fewer nodes point to the root of the tree with more nodes. Rather than explicitly keeping track of the size of the subtree rooted at each node, however, we'll adopt an approach that eases the analysis. For each node, maintain a ***rank***, which is an upper bound on the height of the node. Union by rank makes the root with smaller rank point to the root with larger rank during a UNION operation.

The second heuristic, ***path compression***, is also quite simple and highly effective. As shown in Figure 19.5, FIND-SET operations use it to make each node on the find path point directly to the root. Path compression does not change any ranks.

Pseudocode for disjoint-set forests

The union-by-rank heuristic requires its implementation to keep track of ranks. With each node x, maintain the integer value $x.rank$, which is an upper bound on the height of x (the number of edges in the longest simple path from a descendant leaf to x). When MAKE-SET creates a singleton set, the single node in the

Figure 19.5 Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted. **(a)** A tree representing a set prior to executing FIND-SET(a). Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. **(b)** The same set after executing FIND-SET(a). Each node on the find path now points directly to the root.

corresponding tree has an initial rank of 0. Each FIND-SET operation leaves all ranks unchanged. The UNION operation has two cases, depending on whether the roots of the trees have equal rank. If the roots have unequal ranks, make the root with higher rank the parent of the root with lower rank, but don't change the ranks themselves. If the roots have equal ranks, arbitrarily choose one of the roots as the parent and increment its rank.

Let's put this method into pseudocode, appearing on the next page. The parent of node x is denoted by $x.p$. The LINK procedure, a subroutine called by UNION, takes pointers to two roots as inputs. The FIND-SET procedure with path compression, implemented recursively, turns out to be quite simple.

The FIND-SET procedure is a ***two-pass method***: as it recurses, it makes one pass up the find path to find the root, and as the recursion unwinds, it makes a second pass back down the find path to update each node to point directly to the root. Each call of FIND-SET(x) returns $x.p$ in line 3. If x is the root, then FIND-SET skips line 2 and just returns $x.p$, which is x. In this case the recursion bottoms out. Otherwise, line 2 executes, and the recursive call with parameter $x.p$ returns

```
MAKE-SET(x)
x.p = x
x.rank = 0

UNION(x, y)
LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
if x.rank > y.rank
    y.p = x
else x.p = y
    if x.rank == y.rank
        y.rank = y.rank + 1

FIND-SET(x)
if x ≠ x.p                    // not the root?
    x.p = FIND-SET(x.p)       // the root becomes the parent
return x.p                    // return the root
```

a pointer to the root. Line 2 updates node x to point directly to the root, and line 3 returns this pointer.

Effect of the heuristics on the running time

Separately, either union by rank or path compression improves the running time of the operations on disjoint-set forests, and combining the two heuristics yields an even greater improvement. Alone, union by rank yields a running time of $O(m \lg n)$ for a sequence of m operations, n of which are MAKE-SET (see Exercise 19.4-4), and this bound is tight (see Exercise 19.3-3). Although we won't prove it here, for a sequence of n MAKE-SET operations (and hence at most $n - 1$ UNION operations) and f FIND-SET operations, the worst-case running time using only the path-compression heuristic is $\Theta(n + f \cdot (1 + \log_{2+f/n} n))$.

Combining union by rank and path compression gives a worst-case running time of $O(m\alpha(n))$, where $\alpha(n)$ is a *very* slowly growing function, defined in Section 19.4. In any conceivable application of a disjoint-set data structure, $\alpha(n) \le 4$, and thus, its running time is as good as linear in m for all practical purposes. Mathematically speaking, however, it is superlinear. Section 19.4 proves this $O(m\alpha(n))$ upper bound.

Exercises

19.3-1
Redo Exercise 19.2-2 using a disjoint-set forest with union by rank and path compression. Show the resulting forest with each node including its x_i and rank.

19.3-2
Write a nonrecursive version of FIND-SET with path compression.

19.3-3
Give a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, that takes $\Omega(m \lg n)$ time when using only union by rank and not path compression.

19.3-4
Consider the operation PRINT-SET(x), which is given a node x and prints all the members of x's set, in any order. Show how to add just a single attribute to each node in a disjoint-set forest so that PRINT-SET(x) takes time linear in the number of members of x's set and the asymptotic running times of the other operations are unchanged. Assume that you can print each member of the set in $O(1)$ time.

★ ***19.3-5***
Show that any sequence of m MAKE-SET, FIND-SET, and LINK operations, where all the LINK operations appear before any of the FIND-SET operations, takes only $O(m)$ time when using both path compression and union by rank. You may assume that the arguments to LINK are roots within the disjoint-set forest. What happens in the same situation when using only path compression and not union by rank?

★ 19.4 Analysis of union by rank with path compression

As noted in Section 19.3, the combined union-by-rank and path-compression heuristic runs in $O(m\,\alpha(n))$ time for m disjoint-set operations on n elements. In this section, we'll explore the function α to see just how slowly it grows. Then we'll analyze the running time using the potential method of amortized analysis.

A very quickly growing function and its very slowly growing inverse

For integers $j, k \geq 0$, we define the function $A_k(j)$ as

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0\,, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1\,, \end{cases} \tag{19.1}$$

where the expression $A_{k-1}^{(j+1)}(j)$ uses the functional-iteration notation defined in equation (3.30) on page 68. Specifically, equation (3.30) gives $A_{k-1}^{(0)}(j) = j$ and $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$ for $i \geq 1$. We call the parameter k the ***level*** of the function A.

The function $A_k(j)$ strictly increases with both j and k. To see just how quickly this function grows, we first obtain closed-form expressions for $A_1(j)$ and $A_2(j)$.

Lemma 19.2
For any integer $j \geq 1$, we have $A_1(j) = 2j + 1$.

Proof We first use induction on i to show that $A_0^{(i)}(j) = j + i$. For the base case, $A_0^{(0)}(j) = j = j + 0$. For the inductive step, assume that $A_0^{(i-1)}(j) = j + (i-1)$. Then $A_0^{(i)}(j) = A_0(A_0^{(i-1)}(j)) = (j + (i-1)) + 1 = j + i$. Finally, we note that $A_1(j) = A_0^{(j+1)}(j) = j + (j+1) = 2j + 1$. ■

Lemma 19.3
For any integer $j \geq 1$, we have $A_2(j) = 2^{j+1}(j+1) - 1$.

Proof We first use induction on i to show that $A_1^{(i)}(j) = 2^i(j+1) - 1$. For the base case, we have $A_1^{(0)}(j) = j = 2^0(j+1) - 1$. For the inductive step, assume that $A_1^{(i-1)}(j) = 2^{i-1}(j+1) - 1$. Then $A_1^{(i)}(j) = A_1(A_1^{(i-1)}(j)) = A_1(2^{i-1}(j+1) - 1) = 2 \cdot (2^{i-1}(j+1) - 1) + 1 = 2^i(j+1) - 2 + 1 = 2^i(j+1) - 1$. Finally, we note that $A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j+1) - 1$. ■

Now we can see how quickly $A_k(j)$ grows by simply examining $A_k(1)$ for levels $k = 0, 1, 2, 3, 4$. From the definition of $A_0(j)$ and the above lemmas, we have $A_0(1) = 1 + 1 = 2$, $A_1(1) = 2 \cdot 1 + 1 = 3$, and $A_2(1) = 2^{1+1} \cdot (1+1) - 1 = 7$. We also have

$$\begin{aligned} A_3(1) &= A_2^{(2)}(1) \\ &= A_2(A_2(1)) \\ &= A_2(7) \\ &= 2^8 \cdot 8 - 1 \\ &= 2^{11} - 1 \\ &= 2047 \end{aligned}$$

and

$$\begin{aligned}
A_4(1) &= A_3^{(2)}(1) \\
&= A_3(A_3(1)) \\
&= A_3(2047) \\
&= A_2^{(2048)}(2047) \\
&\gg A_2(2047) \\
&= 2^{2048} \cdot 2048 - 1 \\
&= 2^{2059} - 1 \\
&> 2^{2056} \\
&= (2^4)^{514} \\
&= 16^{514} \\
&\gg 10^{80} ,
\end{aligned}$$

which is the estimated number of atoms in the observable universe. (The symbol "$\gg$" denotes the "much-greater-than" relation.)

We define the inverse of the function $A_k(n)$, for integer $n \geq 0$, by

$$\alpha(n) = \min \{k : A_k(1) \geq n\} . \tag{19.2}$$

In words, $\alpha(n)$ is the lowest level k for which $A_k(1)$ is at least n. From the above values of $A_k(1)$, we see that

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 , \\ 1 & \text{for } n = 3 , \\ 2 & \text{for } 4 \leq n \leq 7 , \\ 3 & \text{for } 8 \leq n \leq 2047 , \\ 4 & \text{for } 2048 \leq n \leq A_4(1) . \end{cases}$$

It is only for values of n so large that the term "astronomical" understates them (greater than $A_4(1)$, a huge number) that $\alpha(n) > 4$, and so $\alpha(n) \leq 4$ for all practical purposes.

Properties of ranks

In the remainder of this section, we prove an $O(m\,\alpha(n))$ bound on the running time of the disjoint-set operations with union by rank and path compression. In order to prove this bound, we first prove some simple properties of ranks.

Lemma 19.4

For all nodes x, we have $x.rank \leq x.p.rank$, with strict inequality if $x \neq x.p$ (x is not a root). The value of $x.rank$ is initially 0, increases through time until $x \neq x.p$,

and from then on, $x.rank$ does not change. The value of $x.p.rank$ monotonically increases over time.

Proof The proof is a straightforward induction on the number of operations, using the implementations of MAKE-SET, UNION, and FIND-SET that appear on page 530, and is left as Exercise 19.4-1. ■

Corollary 19.5
On the simple path from any node going up toward a root, node ranks strictly increase. ■

Lemma 19.6
Every node has rank at most $n - 1$.

Proof Each node's rank starts at 0, and it increases only upon LINK operations. Because there are at most $n - 1$ UNION operations, there are also at most $n - 1$ LINK operations. Because each LINK operation either leaves all ranks alone or increases some node's rank by 1, all ranks are at most $n - 1$. ■

Lemma 19.6 provides a weak bound on ranks. In fact, every node has rank at most $\lfloor \lg n \rfloor$ (see Exercise 19.4-2). The looser bound of Lemma 19.6 suffices for our purposes, however.

Proving the time bound

In order to prove the $O(m\ \alpha(n))$ time bound, we'll use the potential method of amortized analysis from Section 16.3. In performing the amortized analysis, it will be convenient to assume that we invoke the LINK operation rather than the UNION operation. That is, since the parameters of the LINK procedure are pointers to two roots, we act as though we perform the appropriate FIND-SET operations separately. The following lemma shows that even if we count the extra FIND-SET operations induced by UNION calls, the asymptotic running time remains unchanged.

Lemma 19.7
Suppose that we convert a sequence S' of m' MAKE-SET, UNION, and FIND-SET operations into a sequence S of m MAKE-SET, LINK, and FIND-SET operations by turning each UNION into two FIND-SET operations followed by one LINK. Then, if sequence S runs in $O(m\,\alpha(n))$ time, sequence S' runs in $O(m'\,\alpha(n))$ time.

Proof Since each UNION operation in sequence S' is converted into three operations in S, we have $m' \le m \le 3m'$, so that $m = \Theta(m')$, Thus, an $O(m\ \alpha(n))$

time bound for the converted sequence S implies an $O(m' \alpha(n))$ time bound for the original sequence S'. ■

From now on, we assume that the initial sequence of m' MAKE-SET, UNION, and FIND-SET operations has been converted to a sequence of m MAKE-SET, LINK, and FIND-SET operations. We now prove an $O(m \alpha(n))$ time bound for the converted sequence and appeal to Lemma 19.7 to prove the $O(m' \alpha(n))$ running time of the original sequence of m' operations.

Potential function

The potential function we use assigns a potential $\phi_q(x)$ to each node x in the disjoint-set forest after q operations. For the potential Φ_q of the entire forest after q operations, sum the individual node potentials: $\Phi_q = \sum_x \phi_q(x)$. Because the forest is empty before the first operation, the sum is taken over an empty set, and so $\Phi_0 = 0$. No potential Φ_q is ever negative.

The value of $\phi_q(x)$ depends on whether x is a tree root after the qth operation. If it is, or if $x.rank = 0$, then $\phi_q(x) = \alpha(n) \cdot x.rank$.

Now suppose that after the qth operation, x is not a root and that $x.rank \geq 1$. We need to define two auxiliary functions on x before we can define $\phi_q(x)$. First we define

$$\text{level}(x) = \max \{k : x.p.rank \geq A_k(x.rank)\} \ . \tag{19.3}$$

That is, level(x) is the greatest level k for which A_k, applied to x's rank, is no greater than x's parent's rank.

We claim that

$$0 \leq \text{level}(x) < \alpha(n) \ , \tag{19.4}$$

which we see as follows. We have

$$\begin{aligned} x.p.rank &\geq x.rank + 1 && \text{(by Lemma 19.4 because } x \text{ is not a root)} \\ &= A_0(x.rank) && \text{(by the definition (19.1) of } A_0(j)) \ , \end{aligned}$$

which implies that level$(x) \geq 0$, and

$$\begin{aligned} A_{\alpha(n)}(x.rank) &\geq A_{\alpha(n)}(1) && \text{(because } A_k(j) \text{ is strictly increasing)} \\ &\geq n && \text{(by the definition (19.2) of } \alpha(n)) \\ &> x.\ p.rank && \text{(by Lemma 19.6)} \ , \end{aligned}$$

which implies that level$(x) < \alpha(n)$.

For a given nonroot node x, the value of level(x) monotonically increases over time. Why? Because x is not a root, its rank does not change. The rank of $x.p$

monotonically increases over time, since if $x.p$ is not a root then its rank does not change, and if $x.p$ is a root then its rank can never decrease. Thus, the difference between $x.rank$ and $x.p.rank$ monotonically increases over time. Therefore, the value of k needed for $A_k(x.rank)$ to overtake $x.p.rank$ monotonically increases over time as well.

The second auxiliary function applies when $x.rank \geq 1$:

$$\text{iter}(x) = \max\left\{i : x.p.rank \geq A^{(i)}_{\text{level}(x)}(x.rank)\right\} . \tag{19.5}$$

That is, $\text{iter}(x)$ is the largest number of times we can iteratively apply $A_{\text{level}(x)}$, applied initially to x's rank, before exceeding x's parent's rank.

We claim that when $x.rank \geq 1$, we have

$$1 \leq \text{iter}(x) \leq x.rank , \tag{19.6}$$

which we see as follows. We have

$$\begin{aligned} x.p.rank &\geq A_{\text{level}(x)}(x.rank) && \text{(by the definition (19.3) of level}(x)) \\ &= A^{(1)}_{\text{level}(x)}(x.rank) && \text{(by the definition (3.30) of functional iteration)} , \end{aligned}$$

which implies that $\text{iter}(x) \geq 1$. We also have

$$\begin{aligned} A^{(x.rank+1)}_{\text{level}(x)}(x.rank) &= A_{\text{level}(x)+1}(x.rank) && \text{(by the definition (19.1) of } A_k(j)) \\ &> x.p.rank && \text{(by the definition (19.3) of level}(x)) , \end{aligned}$$

which implies that $\text{iter}(x) \leq x.rank$. Note that because $x.p.rank$ monotonically increases over time, in order for $\text{iter}(x)$ to decrease, $\text{level}(x)$ must increase. As long as $\text{level}(x)$ remains unchanged, $\text{iter}(x)$ must either increase or remain unchanged.

With these auxiliary functions in place, we are ready to define the potential of node x after q operations:

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot x.rank & \text{if } x \text{ is a root or } x.rank = 0 , \\ (\alpha(n) - \text{level}(x)) \cdot x.rank - \text{iter}(x) & \\ & \text{if } x \text{ is not a root and } x.rank \geq 1 . \end{cases} \tag{19.7}$$

We next investigate some useful properties of node potentials.

Lemma 19.8

For every node x, and for all operation counts q, we have

$$0 \leq \phi_q(x) \leq \alpha(n) \cdot x.rank .$$

Proof If x is a root or $x.rank = 0$, then $\phi_q(x) = \alpha(n) \cdot x.rank$ by definition. Now suppose that x is not a root and that $x.rank \geq 1$. We can obtain a lower bound on $\phi_q(x)$ by maximizing $\text{level}(x)$ and $\text{iter}(x)$. The bounds (19.4) and (19.6) give $\alpha(n) - \text{level}(x) \geq 1$ and $\text{iter}(x) \leq x.rank$. Thus, we have

$$\begin{aligned}\phi_q(x) &= (\alpha(n) - \text{level}(x)) \cdot x.rank - \text{iter}(x) \\ &\geq x.rank - x.rank \\ &= 0 .\end{aligned}$$

Similarly, minimizing level(x) and iter(x) provides an upper bound on $\phi_q(x)$. By the bound (19.4), level$(x) \geq 0$, and by the bound (19.6), iter$(x) \geq 1$. Thus, we have

$$\begin{aligned}\phi_q(x) &\leq (\alpha(n) - 0) \cdot x.rank - 1 \\ &= \alpha(n) \cdot x.rank - 1 \\ &< \alpha(n) \cdot x.rank .\end{aligned}$$

■

Corollary 19.9
If node x is not a root and $x.rank > 0$, then $\phi_q(x) < \alpha(n) \cdot x.rank$. ■

Potential changes and amortized costs of operations

We are now ready to examine how the disjoint-set operations affect node potentials. Once we understand how each operation can change the potential, we can determine the amortized costs.

Lemma 19.10
Let x be a node that is not a root, and suppose that the qth operation is either a LINK or a FIND-SET. Then after the qth operation, $\phi_q(x) \leq \phi_{q-1}(x)$. Moreover, if $x.rank \geq 1$ and either level(x) or iter(x) changes due to the qth operation, then $\phi_q(x) \leq \phi_{q-1}(x) - 1$. That is, x's potential cannot increase, and if it has positive rank and either level(x) or iter(x) changes, then x's potential drops by at least 1.

Proof Because x is not a root, the qth operation does not change $x.rank$, and because n does not change after the initial n MAKE-SET operations, $\alpha(n)$ remains unchanged as well. Hence, these components of the formula for x's potential remain the same after the qth operation. If $x.rank = 0$, then $\phi_q(x) = \phi_{q-1}(x) = 0$.

Now assume that $x.rank \geq 1$. Recall that level(x) monotonically increases over time. If the qth operation leaves level(x) unchanged, then iter(x) either increases or remains unchanged. If both level(x) and iter(x) are unchanged, then $\phi_q(x) = \phi_{q-1}(x)$. If level$(x)$ is unchanged and iter(x) increases, then it increases by at least 1, and so $\phi_q(x) \leq \phi_{q-1}(x) - 1$.

Finally, if the qth operation increases level(x), it increases by at least 1, so that the value of the term $(\alpha(n) - \text{level}(x)) \cdot x.rank$ drops by at least $x.rank$. Because level(x) increased, the value of iter(x) might drop, but according to the bound (19.6), the drop is by at most $x.rank - 1$. Thus, the increase in poten-

tial due to the change in $\text{iter}(x)$ is less than the decrease in potential due to the change in $\text{level}(x)$, yielding $\phi_q(x) \le \phi_{q-1}(x) - 1$. ■

Our final three lemmas show that the amortized cost of each MAKE-SET, LINK, and FIND-SET operation is $O(\alpha(n))$. Recall from equation (16.2) on page 456 that the amortized cost of each operation is its actual cost plus the change in potential due to the operation.

Lemma 19.11
The amortized cost of each MAKE-SET operation is $O(1)$.

Proof Suppose that the qth operation is MAKE-SET(x). This operation creates node x with rank 0, so that $\phi_q(x) = 0$. No other ranks or potentials change, and so $\Phi_q = \Phi_{q-1}$. Noting that the actual cost of the MAKE-SET operation is $O(1)$ completes the proof. ■

Lemma 19.12
The amortized cost of each LINK operation is $O(\alpha(n))$.

Proof Suppose that the qth operation is LINK(x, y). The actual cost of the LINK operation is $O(1)$. Without loss of generality, suppose that the LINK makes y the parent of x.

To determine the change in potential due to the LINK, note that the only nodes whose potentials may change are x, y, and the children of y just prior to the operation. We'll show that the only node whose potential can increase due to the LINK is y, and that its increase is at most $\alpha(n)$:

- By Lemma 19.10, any node that is y's child just before the LINK cannot have its potential increase due to the LINK.
- From the definition (19.7) of $\phi_q(x)$, note that, since x was a root just before the qth operation, $\phi_{q-1}(x) = \alpha(n) \cdot x.rank$ at that time. If $x.rank = 0$, then $\phi_q(x) = \phi_{q-1}(x) = 0$. Otherwise,

  $$\begin{aligned}\phi_q(x) &< \alpha(n) \cdot x.rank \quad \text{(by Corollary 19.9)}\\ &= \phi_{q-1}(x)\ ,\end{aligned}$$

  and so x's potential decreases.
- Because y is a root prior to the LINK, $\phi_{q-1}(y) = \alpha(n) \cdot y.rank$. After the LINK operation, y remains a root, so that y's potential still equals $\alpha(n)$ times its rank after the operation. The LINK operation either leaves y's rank alone or increases y's rank by 1. Therefore, either $\phi_q(y) = \phi_{q-1}(y)$ or $\phi_q(y) = \phi_{q-1}(y) + \alpha(n)$.

The increase in potential due to the LINK operation, therefore, is at most $\alpha(n)$. The amortized cost of the LINK operation is $O(1) + \alpha(n) = O(\alpha(n))$. ■

Lemma 19.13
The amortized cost of each FIND-SET operation is $O(\alpha(n))$.

Proof Suppose that the qth operation is a FIND-SET and that the find path contains s nodes. The actual cost of the FIND-SET operation is $O(s)$. We will show that no node's potential increases due to the FIND-SET and that at least $\max\{0, s - (\alpha(n) + 2)\}$ nodes on the find path have their potential decrease by at least 1.

We first show that no node's potential increases. Lemma 19.10 takes care of all nodes other than the root. If x is the root, then its potential is $\alpha(n) \cdot x.rank$, which does not change due to the FIND-SET operation.

Now we show that at least $\max\{0, s - (\alpha(n) + 2)\}$ nodes have their potential decrease by at least 1. Let x be a node on the find path such that $x.rank > 0$ and x is followed somewhere on the find path by another node y that is not a root, where $\text{level}(y) = \text{level}(x)$ just before the FIND-SET operation. (Node y need not *immediately* follow x on the find path.) All but at most $\alpha(n) + 2$ nodes on the find path satisfy these constraints on x. Those that do not satisfy them are the first node on the find path (if it has rank 0), the last node on the path (i.e., the root), and the last node w on the path for which $\text{level}(w) = k$, for each $k = 0, 1, 2, \ldots, \alpha(n) - 1$.

Consider such a node x. It has positive rank and is followed somewhere on the find path by nonroot node y such that $\text{level}(y) = \text{level}(x)$ before the path compression occurs. We claim that the path compression decreases x's potential by at least 1. To prove this claim, let $k = \text{level}(x) = \text{level}(y)$ and $i = \text{iter}(x)$ before the path compression occurs. Just prior to the path compression caused by the FIND-SET, we have

$$\begin{aligned}
x.p.rank &\ge A_k^{(i)}(x.rank) && \text{(by the definition (19.5) of iter}(x))\ , \\
y.p.rank &\ge A_k(y.rank) && \text{(by the definition (19.3) of level}(y))\ , \\
y.rank &\ge x.p.rank && \text{(by Corollary 19.5 and because} \\
& && \quad y \text{ follows } x \text{ on the find path)}\ .
\end{aligned}$$

Putting these inequalities together gives

$$\begin{aligned}
y.p.rank &\ge A_k(y.rank) \\
&\ge A_k(x.p.rank) && \text{(because } A_k(j) \text{ is strictly increasing)} \\
&\ge A_k(A_k^{(i)}(x.rank)) \\
&= A_k^{(i+1)}(x.rank) && \text{(by the definition (3.30) of functional iteration)}\ .
\end{aligned}$$

Because path compression makes x and y have the same parent, after path compression we have $x.p.rank = y.p.rank$. The parent of y might change due to the path compression, but if it does, the rank of y's new parent compared with the rank of y's parent before path compression is either the same or greater. Since $x.rank$ does not change, $x.p.rank = y.p.rank \geq A_k^{(i+1)}(x.rank)$ after path compression. By the definition (19.5) of the iter function, the value of $\text{iter}(x)$ increases from i to at least $i + 1$. By Lemma 19.10, $\phi_q(x) \leq \phi_{q-1}(x) - 1$, so that x's potential decreases by at least 1.

The amortized cost of the FIND-SET operation is the actual cost plus the change in potential. The actual cost is $O(s)$, and we have shown that the total potential decreases by at least $\max\{0, s - (\alpha(n) + 2)\}$. The amortized cost, therefore, is at most $O(s) - (s - (\alpha(n) + 2)) = O(s) - s + O(\alpha(n)) = O(\alpha(n))$, since we can scale up the units of potential to dominate the constant hidden in $O(s)$. (See Exercise 19.4-6.) ■

Putting the preceding lemmas together yields the following theorem.

Theorem 19.14
A sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, can be performed on a disjoint-set forest with union by rank and path compression in $O(m\,\alpha(n))$ time.

Proof Immediate from Lemmas 19.7, 19.11, 19.12, and 19.13. ■

Exercises

19.4-1
Prove Lemma 19.4.

19.4-2
Prove that every node has rank at most $\lfloor \lg n \rfloor$.

19.4-3
In light of Exercise 19.4-2, how many bits are necessary to store $x.rank$ for each node x?

19.4-4
Using Exercise 19.4-2, give a simple proof that operations on a disjoint-set forest with union by rank but without path compression run in $O(m \lg n)$ time.

19.4-5

Professor Dante reasons that because node ranks increase strictly along a simple path to the root, node levels must monotonically increase along the path. In other words, if $x.rank > 0$ and $x.p$ is not a root, then $\text{level}(x) \le \text{level}(x.p)$. Is the professor correct?

19.4-6

The proof of Lemma 19.13 ends with scaling the units of potential to dominate the constant hidden in the $O(s)$ term. To be more precise in the proof, you need to change the definition (19.7) of the potential function to multiply each of the two cases by a constant, say c, that dominates the constant in the $O(s)$ term. How must the rest of the analysis change to accommodate this updated potential function?

★ ***19.4-7***

Consider the function $\alpha'(n) = \min\{k : A_k(1) \ge \lg(n+1)\}$. Show that $\alpha'(n) \le 3$ for all practical values of n and, using Exercise 19.4-2, show how to modify the potential-function argument to prove that performing a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, on a disjoint-set forest with union by rank and path compression takes $O(m\alpha'(n))$ time.

Problems

19-1 Offline minimum

In the ***offline minimum problem***, you maintain a dynamic set T of elements from the domain $\{1, 2, \ldots, n\}$ under the operations INSERT and EXTRACT-MIN. The input is a sequence S of n INSERT and m EXTRACT-MIN calls, where each key in $\{1, 2, \ldots, n\}$ is inserted exactly once. Your goal is to determine which key is returned by each EXTRACT-MIN call. Specifically, you must fill in an array $extracted[1:m]$, where for $i = 1, 2, \ldots, m$, $extracted[i]$ is the key returned by the ith EXTRACT-MIN call. The problem is "offline" in the sense that you are allowed to process the entire sequence S before determining any of the returned keys.

a. Consider the following instance of the offline minimum problem, in which each operation INSERT(i) is represented by the value of i and each EXTRACT-MIN is represented by the letter E:

$$4, 8, \text{E}, 3, \text{E}, 9, 2, 6, \text{E}, \text{E}, \text{E}, 1, 7, \text{E}, 5 .$$

Fill in the correct values in the *extracted* array.

To develop an algorithm for this problem, break the sequence S into homogeneous subsequences. That is, represent S by

$$\mathrm{I}_1, \mathrm{E}, \mathrm{I}_2, \mathrm{E}, \mathrm{I}_3, \ldots, \mathrm{I}_m, \mathrm{E}, \mathrm{I}_{m+1} ,$$

where each E represents a single EXTRACT-MIN call and each I_j represents a (possibly empty) sequence of INSERT calls. For each subsequence I_j, initially place the keys inserted by these operations into a set K_j, which is empty if I_j is empty. Then execute the OFFLINE-MINIMUM procedure.

```
OFFLINE-MINIMUM(m, n)
for i = 1 to n
    determine j such that i ∈ K_j
    if j ≠ m + 1
        extracted[j] = i
        let l be the smallest value greater than j for which set K_l exists
        K_l = K_j ∪ K_l, destroying K_j
return extracted
```

b. Argue that the array *extracted* returned by OFFLINE-MINIMUM is correct.

c. Describe how to implement OFFLINE-MINIMUM efficiently with a disjoint-set data structure. Give as tight a bound as you can on the worst-case running time of your implementation.

19-2 Depth determination

In the ***depth-determination problem***, you maintain a forest $\mathcal{F} = \{T_i\}$ of rooted trees under three operations:

MAKE-TREE(v) creates a tree whose only node is v.

FIND-DEPTH(v) returns the depth of node v within its tree.

GRAFT(r, v) makes node r, which is assumed to be the root of a tree, become the child of node v, which is assumed to be in a different tree from r but may or may not itself be a root.

a. Suppose that you use a tree representation similar to a disjoint-set forest: $v.p$ is the parent of node v, except that $v.p = v$ if v is a root. Suppose further that you implement GRAFT(r, v) by setting $r.p = v$ and FIND-DEPTH(v) by following the find path from v up to the root, returning a count of all nodes other than v encountered. Show that the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH, and GRAFT operations is $\Theta(m^2)$.

By using the union-by-rank and path-compression heuristics, you can reduce the worst-case running time. Use the disjoint-set forest $\mathcal{S} = \{S_i\}$, where each set S_i (which is itself a tree) corresponds to a tree T_i in the forest $\mathcal{F}$. The tree structure within a set S_i, however, does not necessarily correspond to that of T_i. In fact, the implementation of S_i does not record the exact parent-child relationships but nevertheless allows you to determine any node's depth in T_i.

The key idea is to maintain in each node v a "pseudodistance" $v.d$, which is defined so that the sum of the pseudodistances along the simple path from v to the root of its set S_i equals the depth of v in T_i. That is, if the simple path from v to its root in S_i is $v_0, v_1, \ldots, v_k$, where $v_0 = v$ and v_k is S_i's root, then the depth of v in T_i is $\sum_{j=0}^{k} v_j.d$.

b. Give an implementation of MAKE-TREE.

c. Show how to modify FIND-SET to implement FIND-DEPTH. Your implementation should perform path compression, and its running time should be linear in the length of the find path. Make sure that your implementation updates pseudodistances correctly.

d. Show how to implement GRAFT(r, v), which combines the sets containing r and v, by modifying the UNION and LINK procedures. Make sure that your implementation updates pseudodistances correctly. Note that the root of a set S_i is not necessarily the root of the corresponding tree T_i.

e. Give a tight bound on the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH, and GRAFT operations, n of which are MAKE-TREE operations.

19-3 Tarjan's offline lowest-common-ancestors algorithm

The ***lowest common ancestor*** of two nodes u and v in a rooted tree T is the node w that is an ancestor of both u and v and that has the greatest depth in T. In the ***offline lowest-common-ancestors problem***, you are given a rooted tree T and an arbitrary set $P = \{\{u, v\}\}$ of unordered pairs of nodes in T, and you wish to determine the lowest common ancestor of each pair in P.

To solve the offline lowest-common-ancestors problem, the LCA procedure on the following page performs a tree walk of T with the initial call LCA$(T.root)$. Assume that each node is colored WHITE prior to the walk.

a. Argue that line 10 executes exactly once for each pair $\{u, v\} \in P$.

b. Argue that at the time of the call LCA(u), the number of sets in the disjoint-set data structure equals the depth of u in T.

```
LCA(u)
MAKE-SET(u)
FIND-SET(u).ancestor = u
for each child v of u in T
    LCA(v)
    UNION(u, v)
    FIND-SET(u).ancestor = u
u.color = BLACK
for each node v such that {u, v} ∈ P
    if v.color == BLACK
        print "The lowest common ancestor of"
            u "and" v "is" FIND-SET(v).ancestor
```

c. Prove that LCA correctly prints the lowest common ancestor of u and v for each pair $\{u, v\} \in P$.

d. Analyze the running time of LCA, assuming that you use the implementation of the disjoint-set data structure in Section 19.3.

Chapter notes

Many of the important results for disjoint-set data structures are due at least in part to R. E. Tarjan. Using aggregate analysis, Tarjan [427, 429] gave the first tight upper bound in terms of the very slowly growing inverse $\widehat{\alpha}(m, n)$ of Ackermann's function. (The function $A_k(j)$ given in Section 19.4 is similar to Ackermann's function, and the function $\alpha(n)$ is similar to $\widehat{\alpha}(m, n)$. Both $\alpha(n)$ and $\widehat{\alpha}(m, n)$ are at most 4 for all conceivable values of m and n.) An upper bound of $O(m \lg^* n)$ was proven earlier by Hopcroft and Ullman [5, 227]. The treatment in Section 19.4 is adapted from a later analysis by Tarjan [431], which is based on an analysis by Kozen [270]. Harfst and Reingold [209] give a potential-based version of Tarjan's earlier bound.

Tarjan and van Leeuwen [432] discuss variants on the path-compression heuristic, including "one-pass methods," which sometimes offer better constant factors in their performance than do two-pass methods. As with Tarjan's earlier analyses of the basic path-compression heuristic, the analyses by Tarjan and van Leeuwen are aggregate. Harfst and Reingold [209] later showed how to make a small change to the potential function to adapt their path-compression analysis to these one-pass variants. Goel et al. [182] prove that linking disjoint-set trees randomly yields the

same asymptotic running time as union by rank. Gabow and Tarjan [166] show that in certain applications, the disjoint-set operations can be made to run in $O(m)$ time.

Tarjan [428] showed that a lower bound of $\Omega(m\,\hat{\alpha}(m,n))$ time is required for operations on any disjoint-set data structure satisfying certain technical conditions. This lower bound was later generalized by Fredman and Saks [155], who showed that in the worst case, $\Omega(m\,\hat{\alpha}(m,n))$ $(\lg n)$-bit words of memory must be accessed.

Part VI Graph Algorithms

Introduction

Graph problems pervade computer science, and algorithms for working with them are fundamental to the field. Hundreds of interesting computational problems are couched in terms of graphs. This part touches on a few of the more significant ones.

Chapter 20 shows how to represent a graph in a computer and then discusses algorithms based on searching a graph using either breadth-first search or depth-first search. The chapter gives two applications of depth-first search: topologically sorting a directed acyclic graph and decomposing a directed graph into its strongly connected components.

Chapter 21 describes how to compute a minimum-weight spanning tree of a graph: the least-weight way of connecting all of the vertices together when each edge has an associated weight. The algorithms for computing minimum spanning trees serve as good examples of greedy algorithms (see Chapter 15).

Chapters 22 and 23 consider how to compute shortest paths between vertices when each edge has an associated length or "weight." Chapter 22 shows how to find shortest paths from a given source vertex to all other vertices, and Chapter 23 examines methods to compute shortest paths between every pair of vertices.

Chapter 24 shows how to compute a maximum flow of material in a flow network, which is a directed graph having a specified source vertex of material, a specified sink vertex, and specified capacities for the amount of material that can traverse each directed edge. This general problem arises in many forms, and a good algorithm for computing maximum flows can help solve a variety of related problems efficiently.

Finally, Chapter 25 explores matchings in bipartite graphs: methods for pairing up vertices that are partitioned into two sets by selecting edges that go between the sets. Bipartite-matching problems model several situations that arise in the real world. The chapter examines how to find a matching of maximum cardinality; the

"stable-marriage problem," which has the highly practical application of matching medical residents to hospitals; and assignment problems, which maximize the total weight of a bipartite matching.

When we characterize the running time of a graph algorithm on a given graph $G = (V, E)$, we usually measure the size of the input in terms of the number of vertices $|V|$ and the number of edges $|E|$ of the graph. That is, we denote the size of the input with two parameters, not just one. We adopt a common notational convention for these parameters. Inside asymptotic notation (such as O-notation or Θ-notation), and *only* inside such notation, the symbol V denotes $|V|$ and the symbol E denotes $|E|$. For example, we might say, "the algorithm runs in $O(VE)$ time," meaning that the algorithm runs in $O(|V|\,|E|)$ time. This convention makes the running-time formulas easier to read, without risk of ambiguity.

Another convention we adopt appears in pseudocode. We denote the vertex set of a graph G by $G.V$ and its edge set by $G.E$. That is, the pseudocode views vertex and edge sets as attributes of a graph.

20 Elementary Graph Algorithms

This chapter presents methods for representing a graph and for searching a graph. Searching a graph means systematically following the edges of the graph so as to visit the vertices of the graph. A graph-searching algorithm can discover much about the structure of a graph. Many algorithms begin by searching their input graph to obtain this structural information. Several other graph algorithms elaborate on basic graph searching. Techniques for searching a graph lie at the heart of the field of graph algorithms.

Section 20.1 discusses the two most common computational representations of graphs: as adjacency lists and as adjacency matrices. Section 20.2 presents a simple graph-searching algorithm called breadth-first search and shows how to create a breadth-first tree. Section 20.3 presents depth-first search and proves some standard results about the order in which depth-first search visits vertices. Section 20.4 provides our first real application of depth-first search: topologically sorting a directed acyclic graph. A second application of depth-first search, finding the strongly connected components of a directed graph, is the topic of Section 20.5.

20.1 Representations of graphs

You can choose between two standard ways to represent a graph $G = (V, E)$: as a collection of adjacency lists or as an adjacency matrix. Either way applies to both directed and undirected graphs. Because the adjacency-list representation provides a compact way to represent ***sparse*** graphs—those for which $|E|$ is much less than $|V|^2$—it is usually the method of choice. Most of the graph algorithms presented in this book assume that an input graph is represented in adjacency-list form. You might prefer an adjacency-matrix representation, however, when the graph is ***dense***—$|E|$ is close to $|V|^2$—or when you need to be able to tell quickly whether there is an edge connecting two given vertices. For example, two of the

Figure 20.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Figure 20.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

all-pairs shortest-paths algorithms presented in Chapter 23 assume that their input graphs are represented by adjacency matrices.

The ***adjacency-list representation*** of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to u in G. (Alternatively, it can contain pointers to these vertices.) Since the adjacency lists represent the edges of a graph, our pseudocode treats the array Adj as an attribute of the graph, just like the edge set E. In pseudocode, therefore, you will see notation such as $G.Adj[u]$. Figure 20.1(b) is an adjacency-list representation of the undirected graph in Figure 20.1(a). Similarly, Figure 20.2(b) is an adjacency-list representation of the directed graph in Figure 20.2(a).

If G is a directed graph, the sum of the lengths of all the adjacency lists is $|E|$, since an edge of the form (u, v) is represented by having v appear in $Adj[u]$. If G is

an undirected graph, the sum of the lengths of all the adjacency lists is $2|E|$, since if (u, v) is an undirected edge, then u appears in v's adjacency list and vice versa. For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is $\Theta(V + E)$. Finding each edge in the graph also takes $\Theta(V + E)$ time, rather than just $\Theta(E)$, since each of the $|V|$ adjacency lists must be examined. Of course, if $|E| = \Omega(V)$ —such as in a connected, undirected graph or a strongly connected, directed graph—we can say that finding each edge takes $\Theta(E)$ time.

Adjacency lists can also represent ***weighted graphs***, that is, graphs for which each edge has an associated ***weight*** given by a ***weight function*** $w : E \to \mathbb{R}$. For example, let $G = (V, E)$ be a weighted graph with weight function w. Then you can simply store the weight $w(u, v)$ of the edge $(u, v) \in E$ with vertex v in u's adjacency list. The adjacency-list representation is quite robust in that you can modify it to support many other graph variants.

A potential disadvantage of the adjacency-list representation is that it provides no quicker way to determine whether a given edge (u, v) is present in the graph than to search for v in the adjacency list $Adj[u]$. An adjacency-matrix representation of the graph remedies this disadvantage, but at the cost of using asymptotically more memory. (See Exercise 20.1-8 for suggestions of variations on adjacency lists that permit faster edge lookup.)

The ***adjacency-matrix representation*** of a graph $G = (V, E)$ assumes that the vertices are numbered $1, 2, \ldots, |V|$ in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

Figures 20.1(c) and 20.2(c) are the adjacency matrices of the undirected and directed graphs in Figures 20.1(a) and 20.2(a), respectively. The adjacency matrix of a graph requires $\Theta(V^2)$ memory, independent of the number of edges in the graph. Because finding each edge in the graph requires examining the entire adjacency matrix, doing so takes $\Theta(V^2)$ time.

Observe the symmetry along the main diagonal of the adjacency matrix in Figure 20.1(c). Since in an undirected graph, (u, v) and (v, u) represent the same edge, the adjacency matrix A of an undirected graph is its own transpose: $A = A^{\mathrm{T}}$. In some applications, it pays to store only the entries on and above the diagonal of the adjacency matrix, thereby cutting the memory needed to store the graph almost in half.

Like the adjacency-list representation of a graph, an adjacency matrix can represent a weighted graph. For example, if $G = (V, E)$ is a weighted graph with edge-weight function w, you can store the weight $w(u, v)$ of the edge $(u, v) \in E$

as the entry in row u and column v of the adjacency matrix. If an edge does not exist, you can store a NIL value as its corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or ∞.

Although the adjacency-list representation is asymptotically at least as space-efficient as the adjacency-matrix representation, adjacency matrices are simpler, and so you might prefer them when graphs are reasonably small. Moreover, adjacency matrices carry a further advantage for unweighted graphs: they require only one bit per entry.

Representing attributes

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. We indicate these attributes using our usual notation, such as $v.d$ for an attribute d of a vertex v. When we indicate edges as pairs of vertices, we use the same style of notation. For example, if edges have an attribute f, then we denote this attribute for edge (u, v) by $(u, v).f$. For the purpose of presenting and understanding algorithms, our attribute notation suffices.

Implementing vertex and edge attributes in real programs can be another story entirely. There is no one best way to store and access vertex and edge attributes. For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how the rest of your program uses the graph. If you represent a graph using adjacency lists, one design choice is to represent vertex attributes in additional arrays, such as an array $d[1 : |V|]$ that parallels the *Adj* array. If the vertices adjacent to u belong to $Adj[u]$, then the attribute $u.d$ can actually be stored in the array entry $d[u]$. Many other ways of implementing attributes are possible. For example, in an object-oriented programming language, vertex attributes might be represented as instance variables within a subclass of a `Vertex` class.

Exercises

20.1-1

Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

20.1-2

Give an adjacency-list representation for a complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that the edges are undirected and that the vertices are numbered from 1 to 7 as in a binary heap.

20.1-3

The ***transpose*** of a directed graph $G = (V, E)$ is the graph $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, where $E^{\mathrm{T}} = \{(v, u) \in V \times V : (u, v) \in E\}$. That is, G^{T} is G with all its edges reversed. Describe efficient algorithms for computing G^{T} from G, for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.

20.1-4

Given an adjacency-list representation of a multigraph $G = (V, E)$, describe an $O(V + E)$-time algorithm to compute the adjacency-list representation of the "equivalent" undirected graph $G' = (V, E')$, where E' consists of the edges in E with all multiple edges between two vertices replaced by a single edge and with all self-loops removed.

20.1-5

The ***square*** of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. Describe efficient algorithms for computing G^2 from G for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.

20.1-6

Most graph algorithms that take an adjacency-matrix representation as input require $\Omega(V^2)$ time, but there are some exceptions. Show how to determine whether a directed graph G contains a ***universal sink***—a vertex with in-degree $|V| - 1$ and out-degree 0—in $O(V)$ time, given an adjacency matrix for G.

20.1-7

The ***incidence matrix*** of a directed graph $G = (V, E)$ with no self-loops is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

$$b_{ij} = \begin{cases} -1 & \text{if edge } j \text{ leaves vertex } i \ , \\ 1 & \text{if edge } j \text{ enters vertex } i \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

Describe what the entries of the matrix product BB^{T} represent, where B^{T} is the transpose of B.

20.1-8

Suppose that instead of a linked list, each array entry $Adj[u]$ is a hash table containing the vertices v for which $(u, v) \in E$, with collisions resolved by chaining. Under the assumption of uniform independent hashing, if all edge lookups are equally likely, what is the expected time to determine whether an edge is in the graph?

What disadvantages does this scheme have? Suggest an alternate data structure for each edge list that solves these problems. Does your alternative have disadvantages compared with the hash table?

20.2 Breadth-first search

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Prim's minimum-spanning-tree algorithm (Section 21.2) and Dijkstra's single-source shortest-paths algorithm (Section 22.3) use ideas similar to those in breadth-first search.

Given a graph $G = (V, E)$ and a distinguished ***source*** vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance from s to each reachable vertex, where the distance to a vertex v equals the smallest number of edges needed to go from s to v. Breadth-first search also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex v reachable from s, the simple path in the breadth-first tree from s to v corresponds to a shortest path from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. You can think of it as discovering vertices in waves emanating from the source vertex. That is, starting from s, the algorithm first discovers all neighbors of s, which have distance 1. Then it discovers all vertices with distance 2, then all vertices with distance 3, and so on, until it has discovered every vertex reachable from s.

In order to keep track of the waves of vertices, breadth-first search could maintain separate arrays or lists of the vertices at each distance from the source vertex. Instead, it uses a single first-in, first-out queue (see Section 10.1.3) containing some vertices at a distance k, possibly followed by some vertices at distance $k + 1$. The queue, therefore, contains portions of two consecutive waves at any time.

To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white, and vertices not reachable from the source vertex s stay white the entire time. A vertex that is reachable from s is ***discovered*** the first time it is encountered during the search, at which time it becomes gray, indicating that is now on the frontier of the search: the boundary between discovered and undiscovered vertices. The queue contains all the gray vertices. Eventually, all the edges of a gray vertex will be explored, so that all of its neighbors will be

discovered. Once all of a vertex's edges have been explored, the vertex is behind the frontier of the search, and it goes from gray to black.[1]

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever the search discovers a white vertex v in the course of scanning the adjacency list of a gray vertex u, the vertex v and the edge (u, v) are added to the tree. We say that u is the ***predecessor*** or ***parent*** of v in the breadth-first tree. Since every vertex reachable from s is discovered at most once, each vertex reachable from s has exactly one parent. (There is one exception: because s is the root of the breadth-first tree, it has no parent.) Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on the simple path in the tree from the root s to vertex v, then u is an ancestor of v and v is a descendant of u.

The breadth-first-search procedure BFS on the following page assumes that the graph $G = (V, E)$ is represented using adjacency lists. It denotes the queue by Q, and it attaches three additional attributes to each vertex v in the graph:

- $v.color$ is the color of v: WHITE, GRAY, or BLACK.
- $v.d$ holds the distance from the source vertex s to v, as computed by the algorithm.
- $v.\pi$ is v's predecessor in the breadth-first tree. If v has no predecessor because it is the source vertex or is undiscovered, then $v.\pi = \text{NIL}$.

Figure 20.3 illustrates the progress of BFS on an undirected graph.

The procedure BFS works as follows. With the exception of the source vertex s, lines 1–4 paint every vertex white, set $u.d = \infty$ for each vertex u, and set the parent of every vertex to be NIL. Because the source vertex s is always the first vertex discovered, lines 5–7 paint s gray, set $s.d$ to 0, and set the predecessor of s to NIL. Lines 8–9 create the queue Q, initially containing just the source vertex.

The **while** loop of lines 10–18 iterates as long as there remain gray vertices, which are on the frontier: discovered vertices that have not yet had their adjacency lists fully examined. This **while** loop maintains the following invariant:

> At the test in line 10, the queue Q consists of the set of gray vertices.

Although we won't use this loop invariant to prove correctness, it is easy to see that it holds prior to the first iteration and that each iteration of the loop maintains the invariant. Prior to the first iteration, the only gray vertex, and the only vertex

---

[1] We distinguish between gray and black vertices to help us understand how breadth-first search operates. In fact, as Exercise 20.2-3 shows, we get the same result even if we do not distinguish between gray and black vertices.

```
BFS(G, s)
1  for each vertex u ∈ G.V − {s}
2      u.color = WHITE
3      u.d = ∞
4      u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for each vertex v in G.Adj[u]   // search the neighbors of u
13         if v.color == WHITE         // is v being discovered now?
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)           // v is now on the frontier
18     u.color = BLACK                 // u is now behind the frontier
```

in Q, is the source vertex s. Line 11 determines the gray vertex u at the head of the queue Q and removes it from Q. The **for** loop of lines 12–17 considers each vertex v in the adjacency list of u. If v is white, then it has not yet been discovered, and the procedure discovers it by executing lines 14–17. These lines paint vertex v gray, set v's distance $v.d$ to $u.d + 1$, record u as v's parent $v.\pi$, and place v at the tail of the queue Q. Once the procedure has examined all the vertices on u's adjacency list, it blackens u in line 18, indicating that u is now behind the frontier. The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted black (in line 18).

The results of breadth-first search may depend upon the order in which the neighbors of a given vertex are visited in line 12: the breadth-first tree may vary, but the distances d computed by the algorithm do not. (See Exercise 20.2-5.)

A simple change allows the BFS procedure to terminate in many cases before the queue Q becomes empty. Because each vertex is discovered at most once and receives a finite d value only when it is discovered, the algorithm can terminate once every vertex has a finite d value. If BFS keeps count of how many vertices have been discovered, it can terminate once either the queue Q is empty or all $|V|$ vertices are discovered.

Figure 20.3 The operation of BFS on an undirected graph. Each part shows the graph and the queue Q at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear within each vertex and below vertices in the queue. The tan region surrounds the frontier of the search, consisting of the vertices in the queue. The light blue region surrounds the vertices behind the frontier, which have been dequeued. Each part highlights in orange the vertex dequeued and the breadth-first tree edges added, if any, in the previous iteration. Blue edges belong to the breadth-first tree constructed so far.

Analysis

Before proving the various properties of breadth-first search, let's take on the easier job of analyzing its running time on an input graph $G = (V, E)$. We use aggregate analysis, as we saw in Section 16.1. After initialization, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(V)$. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all $|V|$ adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(V + E)$. The overhead for initialization is $O(V)$, and thus the total running time of the BFS procedure is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

Shortest paths

Now, let's see why breadth-first search finds the shortest distance from a given source vertex s to each vertex in a graph. Define the ***shortest-path distance*** $\delta(s, v)$ from s to v as the minimum number of edges in any path from vertex s to vertex v. If there is no path from s to v, then $\delta(s, v) = \infty$. We call a path of length $\delta(s, v)$ from s to v a ***shortest path***[2] from s to v. Before showing that breadth-first search correctly computes shortest-path distances, we investigate an important property of shortest-path distances.

Lemma 20.1
Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

$$\delta(s, v) \le \delta(s, u) + 1 .$$

Proof If u is reachable from s, then so is v. In this case, the shortest path from s to v cannot be longer than the shortest path from s to u followed by the edge (u, v), and thus the inequality holds. If u is not reachable from s, then $\delta(s, u) = \infty$, and again, the inequality holds. ■

Our goal is to show that the BFS procedure properly computes $v.d = \delta(s, v)$ for each vertex $v \in V$. We first show that $v.d$ bounds $\delta(s, v)$ from above.

---

[2] Chapters 22 and 23 generalize shortest paths to weighted graphs, in which every edge has a real-valued weight and the weight of a path is the sum of the weights of its constituent edges. The graphs considered in the present chapter are unweighted or, equivalently, all edges have unit weight.

Lemma 20.2
Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies $v.d \geq \delta(s, v)$ at all times, including at termination.

Proof The lemma is true intuitively, because any finite value assigned to $v.d$ equals the number of edges on some path from s to v. The formal proof is by induction on the number of ENQUEUE operations. The inductive hypothesis is that $v.d \geq \delta(s, v)$ for all $v \in V$.

The base case of the induction is the situation immediately after enqueuing s in line 9 of BFS. The inductive hypothesis holds here, because $s.d = 0 = \delta(s, s)$ and $v.d = \infty \geq \delta(s, v)$ for all $v \in V - \{s\}$.

For the inductive step, consider a white vertex v that is discovered during the search from a vertex u. The inductive hypothesis implies that $u.d \geq \delta(s, u)$. The assignment performed by line 15 and Lemma 20.1 give

$$\begin{aligned} v.d &= u.d + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v) \ . \end{aligned}$$

Vertex v is then enqueued, and it is never enqueued again because it is also grayed and lines 14–17 execute only for white vertices. Thus, the value of $v.d$ never changes again, and the inductive hypothesis is maintained. ■

To prove that $v.d = \delta(s, v)$, we first show more precisely how the queue Q operates during the course of BFS. The next lemma shows that at all times, the d values of vertices in the queue either are all the same or form a sequence $\langle k, k, \ldots, k, k+1, k+1, \ldots, k+1 \rangle$ for some integer $k \geq 0$.

Lemma 20.3
Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, \ldots, r-1$.

Proof The proof is by induction on the number of queue operations. Initially, when the queue contains only s, the lemma trivially holds.

For the inductive step, we must prove that the lemma holds after both dequeuing and enqueuing a vertex. First, we examine dequeuing. When the head v_1 of the queue is dequeued, v_2 becomes the new head. (If the queue becomes empty, then the lemma holds vacuously.) By the inductive hypothesis, $v_1.d \leq v_2.d$. But then we have $v_r.d \leq v_1.d + 1 \leq v_2.d + 1$, and the remaining inequalities are unaffected. Thus, the lemma follows with v_2 as the new head.

Now, we examine enqueuing. When line 17 of BFS enqueues a vertex v onto a queue containing vertices $\langle v_1, v_2, \ldots, v_r \rangle$, the enqueued vertex becomes v_{r+1}. If the queue was empty before v was enqueued, then after enqueuing v, we have $r = 1$ and the lemma trivially holds. Now suppose that the queue was nonempty when v was enqueued. At that time, the procedure has most recently removed vertex u, whose adjacency list is currently being scanned, from the queue Q. Just before u was removed, we had $u = v_1$ and the inductive hypothesis held, so that $u.d \le v_2.d$ and $v_r.d \le u.d + 1$. After u is removed from the queue, the vertex that had been v_2 becomes the new head v_1 of the queue, so that now $u.d \le v_1.d$. Thus, $v_{r+1}.d = v.d = u.d + 1 \le v_1.d + 1$. Since $v_r.d \le u.d + 1$, we have $v_r.d \le u.d + 1 = v.d = v_{r+1}.d$, and the remaining inequalities are unaffected. Thus, the lemma follows when v is enqueued. ■

The following corollary shows that the d values at the time that vertices are enqueued monotonically increase over time.

Corollary 20.4
Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $v_i.d \le v_j.d$ at the time that v_j is enqueued.

Proof Immediate from Lemma 20.3 and the property that each vertex receives a finite d value at most once during the course of BFS. ■

We can now prove that breadth-first search correctly finds shortest-path distances.

Theorem 20.5 (Correctness of breadth-first search)
Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $v.d = \delta(s, v)$ for all $v \in V$. Moreover, for any vertex $v \ne s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.

Proof Assume for the purpose of contradiction that some vertex receives a d value not equal to its shortest-path distance. Of all such vertices, let v be a vertex that has the minimum $\delta(s, v)$. By Lemma 20.2, we have $v.d \ge \delta(s, v)$, and thus $v.d > \delta(s, v)$. We cannot have $v = s$, because $s.d = 0$ and $\delta(s, s) = 0$. Vertex v must be reachable from s, for otherwise we would have $\delta(s, v) = \infty \ge v.d$. Let u be the vertex immediately preceding v on some shortest path from s to v (since $v \ne s$, vertex u must exist), so that $\delta(s, v) = \delta(s, u) + 1$. Because $\delta(s, u) < \delta(s, v)$,

and because of how we chose v, we have $u.d = \delta(s, u)$. Putting these properties together gives

$$v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1 . \tag{20.1}$$

Now consider the time when BFS chooses to dequeue vertex u from Q in line 11. At this time, vertex v is either white, gray, or black. We shall show that each of these cases leads to a contradiction of inequality (20.1). If v is white, then line 15 sets $v.d = u.d + 1$, contradicting inequality (20.1). If v is black, then it was already removed from the queue and, by Corollary 20.4, we have $v.d \le u.d$, again contradicting inequality (20.1). If v is gray, then it was painted gray upon dequeuing some vertex w, which was removed from Q earlier than u and for which $v.d = w.d + 1$. By Corollary 20.4, however, $w.d \le u.d$, and so $v.d = w.d + 1 \le u.d + 1$, once again contradicting inequality (20.1).

Thus we conclude that $v.d = \delta(s, v)$ for all $v \in V$. All vertices v reachable from s must be discovered, for otherwise they would have $\infty = v.d > \delta(s, v)$. To conclude the proof of the theorem, observe from lines 15–16 that if $v.\pi = u$, then $v.d = u.d + 1$. Thus, to form a shortest path from s to v, take a shortest path from s to $v.\pi$ and then traverse the edge $(v.\pi, v)$. ■

Breadth-first trees

The blue edges in Figure 20.3 show the breadth-first tree built by the BFS procedure as it searches the graph. The tree corresponds to the π attributes. More formally, for a graph $G = (V, E)$ with source s, we define the ***predecessor subgraph*** of G as $G_\pi = (V_\pi, E_\pi)$, where

$$V_\pi = \{v \in V : v.\pi \ne \text{NIL}\} \cup \{s\} \tag{20.2}$$

and

$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\} . \tag{20.3}$$

The predecessor subgraph G_π is a ***breadth-first tree*** if V_π consists of the vertices reachable from s and, for all $v \in V_\pi$, the subgraph G_π contains a unique simple path from s to v that is also a shortest path from s to v in G. A breadth-first tree is in fact a tree, since it is connected and $|E_\pi| = |V_\pi| - 1$ (see Theorem B.2 on page 1169). We call the edges in E_π ***tree edges***.

The following lemma shows that the predecessor subgraph produced by the BFS procedure is a breadth-first tree.

Lemma 20.6
When applied to a directed or undirected graph $G = (V, E)$, procedure BFS constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.

Proof Line 16 of BFS sets $v.\pi = u$ if and only if $(u, v) \in E$ and $\delta(s, v) < \infty$—that is, if v is reachable from s—and thus V_π consists of the vertices in V reachable from s. Since the predecessor subgraph G_π forms a tree, by Theorem B.2, it contains a unique simple path from s to each vertex in V_π. Applying Theorem 20.5 inductively yields that every such path is a shortest path in G. ■

The PRINT-PATH procedure prints out the vertices on a shortest path from s to v, assuming that BFS has already computed a breadth-first tree. This procedure runs in time linear in the number of vertices in the path printed, since each recursive call is for a path one vertex shorter.

```
PRINT-PATH(G, s, v)
if v == s
    print s
elseif v.π == NIL
    print "no path from" s "to" v "exists"
else PRINT-PATH(G, s, v.π)
    print v
```

Exercises

20.2-1
Show the d and π values that result from running breadth-first search on the directed graph of Figure 20.2(a), using vertex 3 as the source.

20.2-2
Show the d and π values that result from running breadth-first search on the undirected graph of Figure 20.3, using vertex u as the source. Assume that neighbors of a vertex are visited in alphabetical order.

20.2-3
Show that using a single bit to store each vertex color suffices by arguing that the BFS procedure produces the same result if line 18 is removed. Then show how to obviate the need for vertex colors altogether.

20.2-4
What is the running time of BFS if we represent its input graph by an adjacency matrix and modify the algorithm to handle this form of input?

20.2-5
Argue that in a breadth-first search, the value $u.d$ assigned to a vertex u is independent of the order in which the vertices appear in each adjacency list. Using Figure 20.3 as an example, show that the breadth-first tree computed by BFS can depend on the ordering within adjacency lists.

20.2-6
Give an example of a directed graph $G = (V, E)$, a source vertex $s \in V$, and a set of tree edges $E_\pi \subseteq E$ such that for each vertex $v \in V$, the unique simple path in the graph (V, E_π) from s to v is a shortest path in G, yet the set of edges E_π cannot be produced by running BFS on G, no matter how the vertices are ordered in each adjacency list.

20.2-7
There are two types of professional wrestlers: "faces" (short for "babyfaces," i.e., "good guys") and "heels" ("bad guys"). Between any pair of professional wrestlers, there may or may not be a rivalry. You are given the names of n professional wrestlers and a list of r pairs of wrestlers for which there are rivalries. Give an $O(n + r)$-time algorithm that determines whether it is possible to designate some of the wrestlers as faces and the remainder as heels such that each rivalry is between a face and a heel. If it is possible to perform such a designation, your algorithm should produce it.

★ ***20.2-8***
The ***diameter*** of a tree $T = (V, E)$ is defined as $\max\{\delta(u, v) : u, v \in V\}$, that is, the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

20.3 Depth-first search

As its name implies, depth-first search searches "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until all vertices that are reachable from the original source vertex have been discovered. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, repeating the search

from that source. The algorithm repeats this entire process until it has discovered every vertex.[3]

As in breadth-first search, whenever depth-first search discovers a vertex v during a scan of the adjacency list of an already discovered vertex u, it records this event by setting v's predecessor attribute $v.\pi$ to u. Unlike breadth-first search, whose predecessor subgraph forms a tree, depth-first search produces a predecessor subgraph that might contain several trees, because the search may repeat from multiple sources. Therefore, we define the ***predecessor subgraph*** of a depth-first search slightly differently from that of a breadth-first search: it always includes all vertices, and it accounts for multiple sources. Specifically, for a depth-first search the predecessor subgraph is $G_\pi = (V, E_\pi)$, where

$$E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\} .$$

The predecessor subgraph of a depth-first search forms a ***depth-first forest*** comprising several ***depth-first trees***. The edges in E_π are ***tree edges***.

Like breadth-first search, depth-first search colors vertices during the search to indicate their state. Each vertex is initially white, is grayed when it is ***discovered*** in the search, and is blackened when it is ***finished***, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Besides creating a depth-first forest, depth-first search also ***timestamps*** each vertex. Each vertex v has two timestamps: the first timestamp $v.d$ records when v is first discovered (and grayed), and the second timestamp $v.f$ records when the search finishes examining v's adjacency list (and blackens v). These timestamps provide important information about the structure of the graph and are generally helpful in reasoning about the behavior of depth-first search.

The procedure DFS on the facing page records when it discovers vertex u in the attribute $u.d$ and when it finishes vertex u in the attribute $u.f$. These timestamps are integers between 1 and $2|V|$, since there is one discovery event and one finishing event for each of the $|V|$ vertices. For every vertex u,

$$u.d < u.f . \tag{20.4}$$

Vertex u is WHITE before time $u.d$, GRAY between time $u.d$ and time $u.f$, and BLACK thereafter. In the DFS procedure, the input graph G may be undirected or

[3] It may seem arbitrary that breadth-first search is limited to only one source whereas depth-first search may search from multiple sources. Although conceptually, breadth-first search could proceed from multiple sources and depth-first search could be limited to one source, our approach reflects how the results of these searches are typically used. Breadth-first search usually serves to find shortest-path distances and the associated predecessor subgraph from a given source. Depth-first search is often a subroutine in another algorithm, as we'll see later in this chapter.

directed. The variable *time* is a global variable used for timestamping. Figure 20.4 illustrates the progress of DFS on the graph shown in Figure 20.2 (but with vertices labeled by letters rather than numbers).

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1                  // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each vertex v in G.Adj[u]    // explore each edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  time = time + 1
9  u.f = time
10 u.color = BLACK                  // blacken u; it is finished
```

The DFS procedure works as follows. Lines 1–3 paint all vertices white and initialize their π attributes to NIL. Line 4 resets the global time counter. Lines 5–7 check each vertex in V in turn and, when a white vertex is found, visit it by calling DFS-VISIT. Upon every call of DFS-VISIT(G, u) in line 7, vertex u becomes the root of a new tree in the depth-first forest. When DFS returns, every vertex u has been assigned a ***discovery time*** $u.d$ and a ***finish time*** $u.f$.

In each call DFS-VISIT(G, u), vertex u is initially white. Lines 1–3 increment the global variable *time*, record the new value of *time* as the discovery time $u.d$, and paint u gray. Lines 4–7 examine each vertex v adjacent to u and recursively visit v if it is white. As line 4 considers each vertex $v \in Adj[u]$, the depth-first search ***explores*** edge (u, v). Finally, after every edge leaving u has been explored, lines 8–10 increment *time*, record the finish time in $u.f$, and paint u black.

The results of depth-first search may depend upon the order in which line 5 of DFS examines the vertices and upon the order in which line 4 of DFS-VISIT visits the neighbors of a vertex. These different visitation orders tend not to cause

Figure 20.4 The progress of the depth-first-search algorithm DFS on a directed graph. Edges are classified as they are explored: tree edges are labeled T, back edges B, forward edges F, and cross edges C. Timestamps within vertices indicate discovery time/finish times. Tree edges are highlighted in blue. Orange highlights indicate vertices whose discovery or finish times change and edges that are explored in each step.

problems in practice, because many applications of depth-first search can use the result from any depth-first search.

What is the running time of DFS? The loops on lines 1–3 and lines 5–7 of DFS take $\Theta(V)$ time, exclusive of the time to execute the calls to DFS-VISIT. As we did for breadth-first search, we use aggregate analysis. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray. During an execution of DFS-VISIT(G, v), the loop in lines 4–7 executes $|Adj[v]|$ times. Since $\sum_{v \in V} |Adj[v]| = \Theta(E)$ and DFS-VISIT is called once per vertex, the

total cost of executing lines 4–7 of DFS-VISIT is $\Theta(V + E)$. The running time of DFS is therefore $\Theta(V + E)$.

Properties of depth-first search

Depth-first search yields valuable information about the structure of a graph. Perhaps the most basic property of depth-first search is that the predecessor subgraph G_π does indeed form a forest of trees, since the structure of the depth-first trees exactly mirrors the structure of recursive calls of DFS-VISIT. That is, $u = v.\pi$ if and only if DFS-VISIT(G, v) was called during a search of u's adjacency list. Additionally, vertex v is a descendant of vertex u in the depth-first forest if and only if v is discovered during the time in which u is gray.

Another important property of depth-first search is that discovery and finish times have ***parenthesis structure***. If the DFS-VISIT procedure were to print a left parenthesis "(u" when it discovers vertex u and to print a right parenthesis "u)" when it finishes u, then the printed expression would be well formed in the sense that the parentheses are properly nested. For example, the depth-first search of Figure 20.5(a) corresponds to the parenthesization shown in Figure 20.5(b). The following theorem provides another way to characterize the parenthesis structure.

Theorem 20.7 (Parenthesis theorem)
In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth-first tree, or
- the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth-first tree.

Proof We begin with the case in which $u.d < v.d$. We consider two subcases, according to whether $v.d < u.f$. The first subcase occurs when $v.d < u.f$, so that v was discovered while u was still gray, which implies that v is a descendant of u. Moreover, since v was discovered after u, all of its outgoing edges are explored, and v is finished, before the search returns to and finishes u. In this case, therefore, the interval $[v.d, v.f]$ is entirely contained within the interval $[u.d, u.f]$. In the other subcase, $u.f < v.d$, and by inequality (20.4), $u.d < u.f < v.d < v.f$, and thus the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint. Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.

Figure 20.5 Properties of depth-first search. **(a)** The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 20.4. **(b)** Intervals for the discovery time and finish time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finish times of the corresponding vertex. Only tree edges are shown. If two intervals overlap, then one is nested within the other, and the vertex corresponding to the smaller interval is a descendant of the vertex corresponding to the larger. **(c)** The graph of part (a) redrawn with all tree and forward edges going down within a depth-first tree and all back edges going up from a descendant to an ancestor.

The case in which $v.d < u.d$ is similar, with the roles of u and v reversed in the above argument. ■

Corollary 20.8 (Nesting of descendants' intervals)
Vertex v is a proper descendant of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if $u.d < v.d < v.f < u.f$.

Proof Immediate from Theorem 20.7. ■

The next theorem gives another important characterization of when one vertex is a descendant of another in the depth-first forest.

Theorem 20.9 (White-path theorem)
In a depth-first forest of a (directed or undirected) graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.

Proof $\Rightarrow$: If $v = u$, then the path from u to v contains just vertex u, which is still white when $u.d$ receives a value. Now, suppose that v is a proper descendant of u in the depth-first forest. By Corollary 20.8, $u.d < v.d$, and so v is white at time $u.d$. Since v can be any descendant of u, all vertices on the unique simple path from u to v in the depth-first forest are white at time $u.d$.

$\Leftarrow$: Suppose that there is a path of white vertices from u to v at time $u.d$, but v does not become a descendant of u in the depth-first tree. Without loss of generality, assume that every vertex other than v along the path becomes a descendant of u. (Otherwise, let v be the closest vertex to u along the path that doesn't become a descendant of u.) Let w be the predecessor of v in the path, so that w is a descendant of u (w and u may in fact be the same vertex). By Corollary 20.8, $w.f \le u.f$. Because v must be discovered after u is discovered, but before w is finished, $u.d < v.d < w.f \le u.f$. Theorem 20.7 then implies that the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$. By Corollary 20.8, v must after all be a descendant of u. ■

Classification of edges

You can obtain important information about a graph by classifying its edges during a depth-first search. For example, Section 20.4 will show that a directed graph is acyclic if and only if a depth-first search yields no "back" edges (Lemma 20.11).

The depth-first forest G_π produced by a depth-first search on graph G can contain four types of edges:

1. ***Tree edges*** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. ***Back edges*** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
3. ***Forward edges*** are those nontree edges (u, v) connecting a vertex u to a proper descendant v in a depth-first tree.
4. ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

In Figures 20.4 and 20.5, edge labels indicate edge types. Figure 20.5(c) also shows how to redraw the graph of Figure 20.5(a) so that all tree and forward edges head downward in a depth-first tree and all back edges go up. You can redraw any graph in this fashion.

The DFS algorithm has enough information to classify some edges as it encounters them. The key idea is that when an edge (u, v) is first explored, the color of vertex v says something about the edge:

1. WHITE indicates a tree edge,
2. GRAY indicates a back edge, and
3. BLACK indicates a forward or cross edge.

The first case is immediate from the specification of the algorithm. For the second case, observe that the gray vertices always form a linear chain of descendants corresponding to the stack of active DFS-VISIT invocations. The number of gray vertices is 1 more than the depth in the depth-first forest of the vertex most recently discovered. Depth-first search always explores from the deepest gray vertex, so that an edge that reaches another gray vertex has reached an ancestor. The third case handles the remaining possibility. Exercise 20.3-5 asks you to show that such an edge (u, v) is a forward edge if $u.d < v.d$ and a cross edge if $u.d > v.d$.

According to the following theorem, forward and cross edges never occur in a depth-first search of an undirected graph.

Theorem 20.10
In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

Proof Let (u, v) be an arbitrary edge of G, and suppose without loss of generality that $u.d < v.d$. Then, while u is gray, the search must discover and finish v before it finishes u, since v is on u's adjacency list. If the first time that the search explores edge (u, v), it is in the direction from u to v, then v is undiscovered (white) until that time, for otherwise the search would have explored this edge already in the direction from v to u. Thus, (u, v) becomes a tree edge. If the search explores (u, v) first in the direction from v to u, then (u, v) is a back edge, since there must be a path of tree edges from u to v. ■

Since (u, v) and (v, u) are really the same edge in an undirected graph, the proof of Theorem 20.10 says how to classify the edge. When searching from a vertex, which must be gray, if the adjacent vertex is white, then the edge is a tree edge. Otherwise, the edge is a back edge.

The next two sections apply the above theorems about depth-first search.

Figure 20.6 A directed graph for use in Exercises 20.3-2 and 20.5-2.

Exercises

20.3-1

Make a 3-by-3 chart with row and column labels WHITE, GRAY, and BLACK. In each cell (i, j), indicate whether, at any point during a depth-first search of a directed graph, there can be an edge from a vertex of color i to a vertex of color j. For each possible edge, indicate what edge types it can be. Make a second such chart for depth-first search of an undirected graph.

20.3-2

Show how depth-first search works on the graph of Figure 20.6. Assume that the **for** loop of lines 5–7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finish times for each vertex, and show the classification of each edge.

20.3-3

Show the parenthesis structure of the depth-first search of Figure 20.4.

20.3-4

Show that using a single bit to store each vertex color suffices by arguing that the DFS procedure produces the same result if line 10 of DFS-VISIT is removed.

20.3-5

Show that in a directed graph, edge (u, v) is

a. a tree edge or forward edge if and only if $u.d < v.d < v.f < u.f$,

b. a back edge if and only if $v.d \le u.d < u.f \le v.f$, and

c. a cross edge if and only if $v.d < v.f < u.d < u.f$.

20.3-6
Rewrite the procedure DFS, using a stack to eliminate recursion.

20.3-7
Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, and if $u.d < v.d$ in a depth-first search of G, then v is a descendant of u in the depth-first forest produced.

20.3-8
Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, then any depth-first search must result in $v.d \leq u.f$.

20.3-9
Modify the pseudocode for depth-first search so that it prints out every edge in the directed graph G, together with its type. Show what modifications, if any, you need to make if G is undirected.

20.3-10
Explain how a vertex u of a directed graph can end up in a depth-first tree containing only u, even though u has both incoming and outgoing edges in G.

20.3-11
Let $G = (V, E)$ be a connected, undirected graph. Give an $O(V + E)$-time algorithm to compute a path in G that traverses each edge in E exactly once in each direction. Describe how you can find your way out of a maze if you are given a large supply of pennies.

20.3-12
Show how to use a depth-first search of an undirected graph G to identify the connected components of G, so that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify depth-first search so that it assigns to each vertex v an integer label $v.cc$ between 1 and k, where k is the number of connected components of G, such that $u.cc = v.cc$ if and only if u and v belong to the same connected component.

★ ***20.3-13***
A directed graph $G = (V, E)$ is ***singly connected*** if $u \leadsto v$ implies that G contains at most one simple path from u to v for all vertices $u, v \in V$. Give an efficient algorithm to determine whether a directed graph is singly connected.

20.4 Topological sort

This section shows how to use depth-first search to perform a topological sort of a directed acyclic graph, or a "dag" as it is sometimes called. A ***topological sort*** of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. Topological sorting is defined only on directed graphs that are acyclic; no linear ordering is possible when a directed graph contains a cycle. Think of a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. Topological sorting is thus different from the usual kind of "sorting" studied in Part II.

Many applications use directed acyclic graphs to indicate precedences among events. Figure 20.7 gives an example that arises when Professor Bumstead gets dressed in the morning. The professor must don certain garments before others (e.g., socks before shoes). Other items may be put on in any order (e.g., socks and pants). A directed edge (u, v) in the dag of Figure 20.7(a) indicates that garment u must be donned before garment v. A topological sort of this dag therefore gives a possible order for getting dressed. Figure 20.7(b) shows the topologically sorted dag as an ordering of vertices along a horizontal line such that all directed edges go from left to right.

The procedure TOPOLOGICAL-SORT topologically sorts a dag. Figure 20.7(b) shows how the topologically sorted vertices appear in reverse order of their finish times.

```
TOPOLOGICAL-SORT(G)
call DFS(G) to compute finish times v.f for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices
```

The TOPOLOGICAL-SORT procedure runs in $\Theta(V + E)$ time, since depth-first search takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

To prove the correctness of this remarkably simple and efficient algorithm, we start with the following key lemma characterizing directed acyclic graphs.

Lemma 20.11
A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

Figure 20.7 **(a)** Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finish times from a depth-first search are shown next to each vertex. **(b)** The same graph shown topologically sorted, with its vertices arranged from left to right in order of decreasing finish time. All directed edges go from left to right.

Proof $\Rightarrow$: Suppose that a depth-first search produces a back edge (u, v). Then vertex v is an ancestor of vertex u in the depth-first forest. Thus, G contains a path from v to u, and the back edge (u, v) completes a cycle.

$\Leftarrow$: Suppose that G contains a cycle c. We show that a depth-first search of G yields a back edge. Let v be the first vertex to be discovered in c, and let (u, v) be the preceding edge in c. At time $v.d$, the vertices of c form a path of white vertices from v to u. By the white-path theorem, vertex u becomes a descendant of v in the depth-first forest. Therefore, (u, v) is a back edge. ■

Theorem 20.12
TOPOLOGICAL-SORT produces a topological sort of the directed acyclic graph provided as its input.

Proof Suppose that DFS is run on a given dag $G = (V, E)$ to determine finish times for its vertices. It suffices to show that for any pair of distinct vertices $u, v \in V$, if G contains an edge from u to v, then $v.f < u.f$. Consider any edge (u, v) explored by DFS(G). When this edge is explored, v cannot be gray, since then v would be an ancestor of u and (u, v) would be a back edge, contradicting Lemma 20.11. Therefore, v must be either white or black. If v is

Figure 20.8 A dag for topological sorting.

white, it becomes a descendant of u, and so $v.f < u.f$. If v is black, it has already been finished, so that $v.f$ has already been set. Because the search is still exploring from u, it has yet to assign a timestamp to $u.f$, so that the timestamp eventually assigned to $u.f$ is greater than $v.f$. Thus, $v.f < u.f$ for any edge (u, v) in the dag, proving the theorem. ■

Exercises

20.4-1

Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag of Figure 20.8. Assume that the **for** loop of lines 5–7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically.

20.4-2

Give a linear-time algorithm that, given a directed acyclic graph $G = (V, E)$ and two vertices $a, b \in V$, returns the number of simple paths from a to b in G. For example, the directed acyclic graph of Figure 20.8 contains exactly four simple paths from vertex p to vertex v: $\langle p, o, v\rangle$, $\langle p, o, r, y, v\rangle$, $\langle p, o, s, r, y, v\rangle$, and $\langle p, s, r, y, v\rangle$. Your algorithm needs only to count the simple paths, not list them.

20.4-3

Give an algorithm that determines whether an undirected graph $G = (V, E)$ contains a simple cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

20.4-4

Prove or disprove: If a directed graph G contains cycles, then the vertex ordering produced by TOPOLOGICAL-SORT(G) minimizes the number of "bad" edges that are inconsistent with the ordering produced.

20.4-5

Another way to topologically sort a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(V + E)$. What happens to this algorithm if G has cycles?

20.5 Strongly connected components

We now consider a classic application of depth-first search: decomposing a directed graph into its strongly connected components. This section shows how to do so using two depth-first searches. Many algorithms that work with directed graphs begin with such a decomposition. After decomposing the graph into strongly connected components, such algorithms run separately on each one and then combine the solutions according to the structure of connections among components.

Recall from Appendix B that a strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u, v \in C$, both $u \leadsto v$ and $v \leadsto u$, that is, vertices u and v are reachable from each other. Figure 20.9 shows an example.

The algorithm for finding the strongly connected components of a directed graph $G = (V, E)$ uses the transpose of G, which we defined in Exercise 20.1-3 to be the graph $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, where $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$. That is, E^{T} consists of the edges of G with their directions reversed. Given an adjacency-list representation of G, the time to create G^{T} is $\Theta(V + E)$. The graphs G and G^{T} have exactly the same strongly connected components: u and v are reachable from each other in G if and only if they are reachable from each other in G^{T}. Figure 20.9(b) shows the transpose of the graph in Figure 20.9(a), with the strongly connected components shaded blue in both parts.

The linear-time (i.e., $\Theta(V + E)$-time) procedure STRONGLY-CONNECTED-COMPONENTS on the next page computes the strongly connected components of a directed graph $G = (V, E)$ using two depth-first searches, one on G and one on G^{T}.

The idea behind this algorithm comes from a key property of the ***component graph*** $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$, defined as follows. Suppose that G has strongly connected components $C_1, C_2, \ldots, C_k$. The vertex set V^{SCC} is $\{v_1, v_2, \ldots, v_k\}$, and it contains one vertex v_i for each strongly connected component C_i of G. There is an edge $(v_i, v_j) \in E^{\mathrm{SCC}}$ if G contains a directed edge (x, y) for some $x \in C_i$ and some $y \in C_j$. Looked at another way, if we contract all edges whose incident vertices are within the same strongly connected component of G so that

Figure 20.9 **(a)** A directed graph G. Each region shaded light blue is a strongly connected component of G. Each vertex is labeled with its discovery and finish times in a depth-first search, and tree edges are dark blue. **(b)** The graph G^{T}, the transpose of G, with the depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS shown and tree edges shaded dark blue. Each strongly connected component corresponds to one depth-first tree. Orange vertices b, c, g, and h are the roots of the depth-first trees produced by the depth-first search of G^{T}. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finish times u.f for each vertex u
2  create G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

only a single vertex remains, the resulting graph is G^{SCC}. Figure 20.9(c) shows the component graph of the graph in Figure 20.9(a).

The following lemma gives the key property that the component graph is acyclic. We'll see that the algorithm uses this property to visit the vertices of the component graph in topologically sorted order, by considering vertices in the second depth-first search in decreasing order of the finish times that were computed in the first depth-first search.

Lemma 20.13
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$, let $u', v' \in C'$, and suppose that G contains a path $u \rightsquigarrow u'$. Then G cannot also contain a path $v' \rightsquigarrow v$.

Proof If G contains a path $v' \rightsquigarrow v$, then it contains paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$. Thus, u and v' are reachable from each other, thereby contradicting the assumption that C and C' are distinct strongly connected components. ■

Because the STRONGLY-CONNECTED-COMPONENTS procedure performs two depth-first searches, there are two distinct sets of discovery and finish times. In this section, discovery and finish times always refer to those computed by the *first* call of DFS, in line 1.

The notation for discovery and finish times extends to sets of vertices. For a subset U of vertices, $d(U)$ and $f(U)$ are the earliest discovery time and latest finish time, respectively, of any vertex in U: $d(U) = \min\{u.d : u \in U\}$ and $f(U) = \max\{u.f : u \in U\}$.

The following lemma and its corollary give a key property relating strongly connected components and finish times in the first depth-first search.

Lemma 20.14
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C'$ and $v \in C$. Then $f(C') > f(C)$.

Proof We consider two cases, depending on which strongly connected component, C or C', had the first discovered vertex during the first depth-first search.

If $d(C') < d(C)$, let x be the first vertex discovered in C'. At time $x.d$, all vertices in C and C' are white. At that time, G contains a path from x to each vertex in C' consisting only of white vertices. Because $(u, v) \in E$, for any vertex $w \in C$, there is also a path in G at time $x.d$ from x to w consisting only of white vertices: $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$. By the white-path theorem, all vertices in C and C' become descendants of x in the depth-first tree. By Corollary 20.8, x has the latest finish time of any of its descendants, and so $x.f = f(C') > f(C)$.

Otherwise, $d(C') > d(C)$. Let y be the first vertex discovered in C, so that $y.d = d(C)$. At time $y.d$, all vertices in C are white and G contains a path from y to each vertex in C consisting only of white vertices. By the white-path theorem, all vertices in C become descendants of y in the depth-first tree, and by Corollary 20.8, $y.f = f(C)$. Because $d(C') > d(C) = y.d$, all vertices in C' are white at time $y.d$. Since there is an edge (u, v) from C' to C, Lemma 20.13 implies that there cannot be a path from C to C'. Hence, no vertex in C' is reachable

from y. At time $y.f$, therefore, all vertices in C' are still white. Thus, for any vertex $w \in C'$, we have $w.f > y.f$, which implies that $f(C') > f(C)$. ■

Corollary 20.15
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, and suppose that $f(C) > f(C')$. Then E^{T} contains no edge (v, u) such that $u \in C'$ and $v \in C$.

Proof The contrapositive of Lemma 20.14 says that if $f(C') < f(C)$, then there is no edge $(u, v) \in E$ such that $u \in C'$ and $v \in C$. Because the strongly connected components of G and G^{T} are the same, if there is no such edge $(u, v) \in E$, then there is no edge $(v, u) \in E^{\mathrm{T}}$ such that $u \in C'$ and $v \in C$. ■

Corollary 20.15 provides the key to understanding why the strongly connected components algorithm works. Let's examine what happens during the second depth-first search, which is on G^{T}. The search starts from the vertex x whose finish time from the first depth-first search is maximum. This vertex belongs to some strongly connected component C, and since $x.f$ is maximum, $f(C)$ is maximum over all strongly connected components. When the search starts from x, it visits all vertices in C. By Corollary 20.15, G^{T} contains no edges from C to any other strongly connected component, and so the search from x never visits vertices in any other component. Thus, the tree rooted at x contains exactly the vertices of C. Having completed visiting all vertices in C, the second depth-first search selects as a new root a vertex from some other strongly connected component C' whose finish time $f(C')$ is maximum over all components other than C. Again, the search visits all vertices in C'. But by Corollary 20.15, if any edges in G^{T} go from C' to any other component, they must go to C, which the second depth-first search has already visited. In general, when the depth-first search of G^{T} in line 3 visits any strongly connected component, any edges out of that component must be to components that the search has already visited. Each depth-first tree, therefore, corresponds to exactly one strongly connected component. The following theorem formalizes this argument.

Theorem 20.16
The STRONGLY-CONNECTED-COMPONENTS procedure correctly computes the strongly connected components of the directed graph G provided as its input.

Proof We argue by induction on the number of depth-first trees found in the depth-first search of G^{T} in line 3 that the vertices of each tree form a strongly connected component. The inductive hypothesis is that the first k trees produced

in line 3 are strongly connected components. The basis for the induction, when $k = 0$, is trivial.

In the inductive step, we assume that each of the first k depth-first trees produced in line 3 is a strongly connected component, and we consider the $(k + 1)$st tree produced. Let the root of this tree be vertex u, and let u be in strongly connected component C. Because of how the depth-first search chooses roots in line 3, $u.f = f(C) > f(C')$ for any strongly connected component C' other than C that has yet to be visited. By the inductive hypothesis, at the time that the search visits u, all other vertices of C are white. By the white-path theorem, therefore, all other vertices of C are descendants of u in its depth-first tree. Moreover, by the inductive hypothesis and by Corollary 20.15, any edges in G^{T} that leave C must be to strongly connected components that have already been visited. Thus, no vertex in any strongly connected component other than C is a descendant of u during the depth-first search of G^{T}. The vertices of the depth-first tree in G^{T} that is rooted at u form exactly one strongly connected component, which completes the inductive step and the proof. ■

Here is another way to look at how the second depth-first search operates. Consider the component graph $(G^{\mathrm{T}})^{\mathrm{SCC}}$ of G^{T}. If you map each strongly connected component visited in the second depth-first search to a vertex of $(G^{\mathrm{T}})^{\mathrm{SCC}}$, the second depth-first search visits vertices of $(G^{\mathrm{T}})^{\mathrm{SCC}}$ in the reverse of a topologically sorted order. If you reverse the edges of $(G^{\mathrm{T}})^{\mathrm{SCC}}$, you get the graph $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}}$. Because $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$ (see Exercise 20.5-4), the second depth-first search visits the vertices of G^{SCC} in topologically sorted order.

Exercises

20.5-1

How can the number of strongly connected components of a graph change if a new edge is added?

20.5-2

Show how the procedure STRONGLY-CONNECTED-COMPONENTS works on the graph of Figure 20.6. Specifically, show the finish times computed in line 1 and the forest produced in line 3. Assume that the loop of lines 5–7 of DFS considers vertices in alphabetical order and that the adjacency lists are in alphabetical order.

20.5-3

Professor Bacon rewrites the algorithm for strongly connected components to use the original (instead of the transpose) graph in the second depth-first search and

scan the vertices in order of *increasing* finish times. Does this modified algorithm always produce correct results?

20.5-4
Prove that for any directed graph G, the transpose of the component graph of G^{T} is the same as the component graph of G. That is, $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$.

20.5-5
Give an $O(V + E)$-time algorithm to compute the component graph of a directed graph $G = (V, E)$. Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.

20.5-6
Give an $O(V + E)$-time algorithm that, given a directed graph $G = (V, E)$, constructs another graph $G' = (V, E')$ such that G and G' have the same strongly connected components, G' has the same component graph as G, and $|E'|$ is as small as possible.

20.5-7
A directed graph $G = (V, E)$ is ***semiconnected*** if, for all pairs of vertices $u, v \in V$, we have $u \leadsto v$ or $v \leadsto u$. Give an efficient algorithm to determine whether G is semiconnected. Prove that your algorithm is correct, and analyze its running time.

20.5-8
Let $G = (V, E)$ be a directed graph, and let $l : V \to \mathbb{R}$ be a function that assigns a real-valued label l to each vertex. For vertices $s, t \in V$, define

$$\Delta l(s,t) = \begin{cases} l(t) - l(s) & \text{if there is a path from } s \text{ to } t \text{ in } G\ , \\ -\infty & \text{otherwise}\ . \end{cases}$$

Give an $O(V + E)$-time algorithm to find vertices s and t such that $\Delta l(s,t)$ is maximum over all pairs of vertices. (*Hint:* Use Exercise 20.5-5.)

Problems

20-1 Classifying edges by breadth-first search
A depth-first forest classifies the edges of a graph into tree, back, forward, and cross edges. A breadth-first tree can also be used to classify the edges reachable from the source of the search into the same four categories.

Figure 20.10 The articulation points, bridges, and biconnected components of a connected, undirected graph for use in Problem 20-2. The articulation points are the orange vertices, the bridges are the dark blue edges, and the biconnected components are the edges in the light blue regions, with a *bcc* numbering shown.

a. Prove that in a breadth-first search of an undirected graph, the following properties hold:

1. There are no back edges and no forward edges.
2. If (u, v) is a tree edge, then $v.d = u.d + 1$.
3. If (u, v) is a cross edge, then $v.d = u.d$ or $v.d = u.d + 1$.

b. Prove that in a breadth-first search of a directed graph, the following properties hold:

1. There are no forward edges.
2. If (u, v) is a tree edge, then $v.d = u.d + 1$.
3. If (u, v) is a cross edge, then $v.d \le u.d + 1$.
4. If (u, v) is a back edge, then $0 \le v.d \le u.d$.

20-2 Articulation points, bridges, and biconnected components

Let $G = (V, E)$ be a connected, undirected graph. An ***articulation point*** of G is a vertex whose removal disconnects G. A ***bridge*** of G is an edge whose removal disconnects G. A ***biconnected component*** of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle. Figure 20.10 illustrates these definitions. You can determine articulation points, bridges, and biconnected components using depth-first search. Let $G_\pi = (V, E_\pi)$ be a depth-first tree of G.

a. Prove that the root of G_π is an articulation point of G if and only if it has at least two children in G_π.

b. Let v be a nonroot vertex of G_π. Prove that v is an articulation point of G if and only if v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.

c. Let

$$v.low = \min \begin{cases} v.d\ , \\ w.d : (u, w) \text{ is a back edge for some descendant } u \text{ of } v\ . \end{cases}$$

Show how to compute $v.low$ for all vertices $v \in V$ in $O(E)$ time.

d. Show how to compute all articulation points in $O(E)$ time.

e. Prove that an edge of G is a bridge if and only if it does not lie on any simple cycle of G.

f. Show how to compute all the bridges of G in $O(E)$ time.

g. Prove that the biconnected components of G partition the nonbridge edges of G.

h. Give an $O(E)$-time algorithm to label each edge e of G with a positive integer $e.bcc$ such that $e.bcc = e'.bcc$ if and only if e and e' belong to the same biconnected component.

20-3 Euler tour

An ***Euler tour*** of a strongly connected, directed graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.

a. Show that G has an Euler tour if and only if in-degree(v) = out-degree(v) for each vertex $v \in V$.

b. Describe an $O(E)$-time algorithm to find an Euler tour of G if one exists. (*Hint:* Merge edge-disjoint cycles.)

20-4 Reachability

Let $G = (V, E)$ be a directed graph in which each vertex $u \in V$ is labeled with a unique integer $L(u)$ from the set $\{1, 2, \ldots, |V|\}$. For each vertex $u \in V$, let $R(u) = \{v \in V : u \rightsquigarrow v\}$ be the set of vertices that are reachable from u. Define $\min(u)$ to be the vertex in $R(u)$ whose label is minimum, that is, $\min(u)$ is the vertex v such that $L(v) = \min\{L(w) : w \in R(u)\}$. Give an $O(V + E)$-time algorithm that computes $\min(u)$ for all vertices $u \in V$.

20-5 Inserting and querying vertices in planar graphs

A ***planar*** graph is an undirected graph that can be drawn in the plane with no edges crossing. Euler proved that every planar graph has $|E| < 3|V|$.

Consider the following two operations on a planar graph G:

- INSERT$(G, v, neighbors)$ inserts a new vertex v into G, where *neighbors* is an array (possibly empty) of vertices that have already been inserted into G and will become all the neighbors of v in G when v is inserted.
- NEWEST-NEIGHBOR(G, v) returns the neighbor of vertex v that was most recently inserted into G, or NIL if v has no neighbors.

Design a data structure that supports these two operations such that NEWEST-NEIGHBOR takes $O(1)$ worst-case time and INSERT takes $O(1)$ amortized time. Note that the length of the array *neighbors* given to INSERT may vary. (*Hint:* Use a potential function for the amortized analysis.)

Chapter notes

Even [137] and Tarjan [429] are excellent references for graph algorithms.

Breadth-first search was discovered by Moore [334] in the context of finding paths through mazes. Lee [280] independently discovered the same algorithm in the context of routing wires on circuit boards.

Hopcroft and Tarjan [226] advocated the use of the adjacency-list representation over the adjacency-matrix representation for sparse graphs and were the first to recognize the algorithmic importance of depth-first search. Depth-first search has been widely used since the late 1950s, especially in artificial intelligence programs.

Tarjan [426] gave a linear-time algorithm for finding strongly connected components. The algorithm for strongly connected components in Section 20.5 is adapted from Aho, Hopcroft, and Ullman [6], who credit it to S. R. Kosaraju (unpublished) and Sharir [408]. Dijkstra [117, Chapter 25] also developed an algorithm for strongly connected components that is based on contracting cycles. Subsequently, Gabow [163] rediscovered this algorithm. Knuth [259] was the first to give a linear-time algorithm for topological sorting.

21 Minimum Spanning Trees

Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of n pins, the designer can use an arrangement of $n - 1$ wires, each connecting two pins. Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

To model this wiring problem, use a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of pins, and for each edge $(u, v) \in E$, a weight $w(u, v)$ specifies the cost (amount of wire needed) to connect u and v. The goal is to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

is minimized. Since T is acyclic and connects all of the vertices, it must form a tree, which we call a ***spanning tree*** since it "spans" the graph G. We call the problem of determining the tree T the ***minimum-spanning-tree problem***.[1] Figure 21.1 shows an example of a connected graph and a minimum spanning tree.

This chapter studies two ways to solve the minimum-spanning-tree problem. Kruskal's algorithm and Prim's algorithm both run in $O(E \lg V)$ time. Prim's algorithm achieves this bound by using a binary heap as a priority queue. By using Fibonacci heaps instead (see page 478), Prim's algorithm runs in $O(E + V \lg V)$ time. This bound is better than $O(E \lg V)$ whenever $|E|$ grows asymptotically faster than $|V|$.

---

[1] The phrase "minimum spanning tree" is a shortened form of the phrase "minimum-weight spanning tree." There is no point in minimizing the number of edges in T, since all spanning trees have exactly $|V| - 1$ edges by Theorem B.2 on page 1169.

Figure 21.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the blue edges form a minimum spanning tree. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

The two algorithms are greedy algorithms, as described in Chapter 15. Each step of a greedy algorithm must make one of several possible choices. The greedy strategy advocates making the choice that is the best at the moment. Such a strategy does not generally guarantee that it always finds globally optimal solutions to problems. For the minimum-spanning-tree problem, however, we can prove that certain greedy strategies do yield a spanning tree with minimum weight. Although you can read this chapter independently of Chapter 15, the greedy methods presented here are a classic application of the theoretical notions introduced there.

Section 21.1 introduces a "generic" minimum-spanning-tree method that grows a spanning tree by adding one edge at a time. Section 21.2 gives two algorithms that implement the generic method. The first algorithm, due to Kruskal, is similar to the connected-components algorithm from Section 19.1. The second, due to Prim, resembles Dijkstra's shortest-paths algorithm (Section 22.3).

Because a tree is a type of graph, in order to be precise we must define a tree in terms of not just its edges, but its vertices as well. Because this chapter focuses on trees in terms of their edges, we'll implicitly understand that the vertices of a tree T are those that some edge of T is incident on.

21.1 Growing a minimum spanning tree

The input to the minumum-spanning-tree problem is a connected, undirected graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$. The goal is to find a minimum spanning tree for G. The two algorithms considered in this chapter use a greedy approach to the problem, although they differ in how they apply this approach.

This greedy strategy is captured by the procedure GENERIC-MST on the facing page, which grows the minimum spanning tree one edge at a time. The generic method manages a set A of edges, maintaining the following loop invariant:

Prior to each iteration, A is a subset of some minimum spanning tree.

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

Each step determines an edge (u, v) that the procedure can add to A without violating this invariant, in the sense that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree. We call such an edge a ***safe edge*** for A, since it can be added safely to A while maintaining the invariant.

This generic algorithm uses the loop invariant as follows:

Initialization: After line 1, the set A trivially satisfies the loop invariant.

Maintenance: The loop in lines 2–4 maintains the invariant by adding only safe edges.

Termination: All edges added to A belong to a minimum spanning tree, and the loop must terminate by the time it has considered all edges. Therefore, the set A returned in line 5 must be a minimum spanning tree.

The tricky part is, of course, finding a safe edge in line 3. One must exist, since when line 3 is executed, the invariant dictates that there is a spanning tree T such that $A \subseteq T$. Within the **while** loop body, A must be a proper subset of T, and therefore there must be an edge $(u, v) \in T$ such that $(u, v) \notin A$ and (u, v) is safe for A.

The remainder of this section provides a rule (Theorem 21.1) for recognizing safe edges. The next section describes two algorithms that use this rule to find safe edges efficiently.

We first need some definitions. A ***cut*** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V. Figure 21.2 illustrates this notion. We say that an edge $(u, v) \in E$ ***crosses*** the cut $(S, V - S)$ if one of its endpoints belongs to S and the other belongs to $V - S$. A cut ***respects*** a set A of edges if no edge in A crosses the cut. An edge is a ***light edge*** crossing a cut if its weight is the minimum of any edge crossing the cut. There can be more than one light edge crossing a cut in the case of ties. More generally, we say that an edge is a ***light edge*** satisfying a given property if its weight is the minimum of any edge satisfying the property.

The following theorem gives the rule for recognizing safe edges.

Theorem 21.1

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum

Figure 21.2 A cut $(S, V - S)$ of the graph from Figure 21.1. Orange vertices belong to the set S, and tan vertices belong to $V - S$. The edges crossing the cut are those connecting tan vertices with orange vertices. The edge (d, c) is the unique light edge crossing the cut. Blue edges form a subset A of the edges. The cut $(S, V - S)$ respects A, since no edge of A crosses the cut.

spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.

Proof Let T be a minimum spanning tree that includes A, and assume that T does not contain the light edge (u, v), since if it does, we are done. We'll construct another minimum spanning tree T' that includes $A \cup \{(u, v)\}$ by using a cut-and-paste technique, thereby showing that (u, v) is a safe edge for A.

The edge (u, v) forms a cycle with the edges on the simple path p from u to v in T, as Figure 21.3 illustrates. Since u and v are on opposite sides of the cut $(S, V - S)$, at least one edge in T lies on the simple path p and also crosses the cut. Let (x, y) be any such edge. The edge (x, y) is not in A, because the cut respects A. Since (x, y) is on the unique simple path from u to v in T, removing (x, y) breaks T into two components. Adding (u, v) reconnects them to form a new spanning tree $T' = (T - \{(x, y)\}) \cup \{(u, v)\}$.

We next show that T' is a minimum spanning tree. Since (u, v) is a light edge crossing $(S, V - S)$ and (x, y) also crosses this cut, $w(u, v) \le w(x, y)$. Therefore,

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\le w(T) . \end{aligned}$$

But T is a minimum spanning tree, so that $w(T) \le w(T')$, and thus, T' must be a minimum spanning tree as well.

It remains to show that (u, v) is actually a safe edge for A. We have $A \subseteq T'$, since $A \subseteq T$ and $(x, y) \notin A$, and thus, $A \cup \{(u, v)\} \subseteq T'$. Consequently, since T' is a minimum spanning tree, (u, v) is safe for A. ■

Theorem 21.1 provides insight into how the GENERIC-MST method works on a connected graph $G = (V, E)$. As the method proceeds, the set A is always acyclic, since it is a subset of a minimum spanning tree and a tree may not contain a cycle.

Figure 21.3 The proof of Theorem 21.1. Orange vertices belong to S, and tan vertices belong to $V - S$. Only edges in the minimum spanning tree T are shown, along with edge (u, v), which does not lie in T. The edges in A are blue, and (u, v) is a light edge crossing the cut $(S, V - S)$. The edge (x, y) is an edge on the unique simple path p from u to v in T. To form a minimum spanning tree T' that contains (u, v), remove the edge (x, y) from T and add the edge (u, v).

At any point in the execution, the graph $G_A = (V, A)$ is a forest, and each of the connected components of G_A is a tree. (Some of the trees may contain just one vertex, as is the case, for example, when the method begins: A is empty and the forest contains $|V|$ trees, one for each vertex.) Moreover, any safe edge (u, v) for A connects distinct components of G_A, since $A \cup \{(u, v)\}$ must be acyclic.

The **while** loop in lines 2–4 of GENERIC-MST executes $|V| - 1$ times because it finds one of the $|V| - 1$ edges of a minimum spanning tree in each iteration. Initially, when $A = \emptyset$, there are $|V|$ trees in G_A, and each iteration reduces that number by 1. When the forest contains only a single tree, the method terminates.

The two algorithms in Section 21.2 use the following corollary to Theorem 21.1.

Corollary 21.2
Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u, v) is a light edge connecting C to some other component in G_A, then (u, v) is safe for A.

Proof The cut $(V_C, V - V_C)$ respects A, and (u, v) is a light edge for this cut. Therefore, (u, v) is safe for A. ■

Exercises

21.1-1
Let (u, v) be a minimum-weight edge in a connected graph G. Show that (u, v) belongs to some minimum spanning tree of G.

21.1-2
Professor Sabatier conjectures the following converse of Theorem 21.1. Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a safe edge for A crossing $(S, V - S)$. Then, (u, v) is a light edge for the cut. Show that the professor's conjecture is incorrect by giving a counterexample.

21.1-3
Show that if an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

21.1-4
Give a simple example of a connected graph such that the set of edges $\{(u, v) :$ there exists a cut $(S, V - S)$ such that (u, v) is a light edge crossing $(S, V - S)\}$ does not form a minimum spanning tree.

21.1-5
Let e be a maximum-weight edge on some cycle of connected graph $G = (V, E)$. Prove that there is a minimum spanning tree of $G' = (V, E - \{e\})$ that is also a minimum spanning tree of G. That is, there is a minimum spanning tree of G that does not include e.

21.1-6
Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

21.1-7
Argue that if all edge weights of a graph are positive, then any subset of edges that connects all vertices and has minimum total weight must be a tree. Give an example to show that the same conclusion does not follow if we allow some weights to be nonpositive.

21.1-8
Let T be a minimum spanning tree of a graph G, and let L be the sorted list of the edge weights of T. Show that for any other minimum spanning tree T' of G, the list L is also the sorted list of edge weights of T'.

21.1-9
Let T be a minimum spanning tree of a graph $G = (V, E)$, and let V' be a subset of V. Let T' be the subgraph of T induced by V', and let G' be the subgraph of G induced by V'. Show that if T' is connected, then T' is a minimum spanning tree of G'.

21.1-10
Given a graph G and a minimum spanning tree T, suppose that the weight of one of the edges in T decreases. Show that T is still a minimum spanning tree for G. More formally, let T be a minimum spanning tree for G with edge weights given by weight function w. Choose one edge $(x, y) \in T$ and a positive number k, and define the weight function w' by

$$w'(u, v) = \begin{cases} w(u, v) & \text{if } (u, v) \neq (x, y) , \\ w(x, y) - k & \text{if } (u, v) = (x, y) . \end{cases}$$

Show that T is a minimum spanning tree for G with edge weights given by w'.

★ ***21.1-11***
Given a graph G and a minimum spanning tree T, suppose that the weight of one of the edges *not* in T decreases. Give an algorithm for finding the minimum spanning tree in the modified graph.

21.2 The algorithms of Kruskal and Prim

The two minimum-spanning-tree algorithms described in this section elaborate on the generic method. They each use a specific rule to determine a safe edge in line 3 of GENERIC-MST. In Kruskal's algorithm, the set A is a forest whose vertices are all those of the given graph. The safe edge added to A is always a lowest-weight edge in the graph that connects two distinct components. In Prim's algorithm, the set A forms a single tree. The safe edge added to A is always a lowest-weight edge connecting the tree to a vertex not in the tree. Both algorithms assume that the input graph is connected and represented by adjacency lists.

Figure 21.4 The execution of Kruskal's algorithm on the graph from Figure 21.1. Blue edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. A red arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Kruskal's algorithm

Kruskal's algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) with the lowest weight. Let C_1 and C_2 denote the two trees that are connected by (u, v). Since (u, v) must be a light edge connecting C_1 to some other tree, Corollary 21.2 implies

Figure 21.4, continued Further steps in the execution of Kruskal's algorithm.

that (u, v) is a safe edge for C_1. Kruskal's algorithm qualifies as a greedy algorithm because at each step it adds to the forest an edge with the lowest possible weight.

Like the algorithm to compute connected components from Section 19.1, the procedure MST-KRUSKAL on the following page uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in one tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u. Thus, to determine whether two vertices u and v belong to the same tree, just test whether FIND-SET(u) equals FIND-SET(v). To combine trees, Kruskal's algorithm calls the UNION procedure.

Figure 21.4 shows how Kruskal's algorithm works. Lines 1–3 initialize the set A to the empty set and create $|V|$ trees, one containing each vertex. The **for** loop in lines 6–9 examines edges in order of weight, from lowest to highest. The loop checks, for each edge (u, v), whether the endpoints u and v belong to the same tree. If they do, then the edge (u, v) cannot be added to the forest without creating a cycle, and the edge is ignored. Otherwise, the two vertices belong to different

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  create a single list of the edges in G.E
5  sort the list of edges into monotonically increasing order by weight w
6  for each edge (u, v) taken from the sorted list in order
7      if FIND-SET(u) ≠ FIND-SET(v)
8          A = A ∪ {(u, v)}
9          UNION(u, v)
10 return A
```

trees. In this case, line 8 adds the edge (u, v) to A, and line 9 merges the vertices in the two trees.

The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on the specific implementation of the disjoint-set data structure. Let's assume that it uses the disjoint-set-forest implementation of Section 19.3 with the union-by-rank and path-compression heuristics, since that is the asymptotically fastest implementation known. Initializing the set A in line 1 takes $O(1)$ time, creating a single list of edges in line 4 takes $O(V + E)$ time (which is $O(E)$ because G is connected), and the time to sort the edges in line 5 is $O(E \lg E)$. (We'll account for the cost of the $|V|$ MAKE-SET operations in the **for** loop of lines 2–3 in a moment.) The **for** loop of lines 6–9 performs $O(E)$ FIND-SET and UNION operations on the disjoint-set forest. Along with the $|V|$ MAKE-SET operations, these disjoint-set operations take a total of $O((V + E)\,\alpha(V))$ time, where α is the very slowly growing function defined in Section 19.4. Because we assume that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\,\alpha(V))$ time. Moreover, since $\alpha(|V|) = O(\lg V) = O(\lg E)$, the total running time of Kruskal's algorithm is $O(E \lg E)$. Observing that $|E| < |V|^2$, we have $\lg |E| = O(\lg V)$, and so we can restate the running time of Kruskal's algorithm as $O(E \lg V)$.

Prim's algorithm

Like Kruskal's algorithm, Prim's algorithm is a special case of the generic minimum-spanning-tree method from Section 21.1. Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph, which we'll see in Section 22.3. Prim's algorithm has the property that the edges in the set A always form a single tree. As Figure 21.5 shows, the tree starts from an arbitrary root vertex r and grows until it spans all the vertices in V. Each step adds to the tree A

Figure 21.5 The execution of Prim's algorithm on the graph from Figure 21.1. The root vertex is a. Blue vertices and edges belong to the tree being grown, and tan vertices have yet to be added to the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. The edge and vertex added to the tree are highlighted in orange. In the second step (part (c)), for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

a light edge that connects A to an isolated vertex—one on which no edge of A is incident. By Corollary 21.2, this rule adds only edges that are safe for A. Therefore, when the algorithm terminates, the edges in A form a minimum spanning tree. This strategy qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree's weight.

In the procedure MST-PRIM below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm. In order to efficiently select a new edge to add into tree A, the algorithm maintains a min-priority queue Q of all vertices that are *not* in the tree, based on a *key* attribute. For each vertex v, the attribute $v.key$ is the minimum weight of any edge connecting v to a vertex in the tree, where by convention, $v.key = \infty$ if there is no such edge. The attribute $v.\pi$ names the parent of v in the tree. The algorithm implicitly maintains the set A from GENERIC-MST as

$$A = \{(v, v.\pi) : v \in V - \{r\} - Q\} ,$$

where we interpret the vertices in Q as forming a set. When the algorithm terminates, the min-priority queue Q is empty, and thus the minimum spanning tree A for G is

$$A = \{(v, v.\pi) : v \in V - \{r\}\} .$$

```
MST-PRIM(G, w, r)
1  for each vertex u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = ∅
6  for each vertex u ∈ G.V
7      INSERT(Q, u)
8  while Q ≠ ∅
9      u = EXTRACT-MIN(Q)          // add u to the tree
10     for each vertex v in G.Adj[u]   // update keys of u's non-tree neighbors
11         if v ∈ Q and w(u, v) < v.key
12             v.π = u
13             v.key = w(u, v)
14             DECREASE-KEY(Q, v, w(u, v))
```

Figure 21.5 shows how Prim's algorithm works. Lines 1–7 set the key of each vertex to ∞ (except for the root r, whose key is set to 0 to make it the first vertex processed), set the parent of each vertex to NIL, and insert each vertex into the min-priority queue Q. The algorithm maintains the following three-part loop invariant:

Prior to each iteration of the **while** loop of lines 8–14,

1. $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
3. For all vertices $v \in Q$, if $v.\pi \neq$ NIL, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ connecting v to some vertex already placed into the minimum spanning tree.

Line 9 identifies a vertex $u \in Q$ incident on a light edge that crosses the cut $(V - Q, Q)$ (with the exception of the first iteration, in which $u = r$ due to lines 4–7). Removing u from the set Q adds it to the set $V - Q$ of vertices in the tree, thus adding the edge $(u, u.\pi)$ to A. The **for** loop of lines 10–14 updates the *key* and π attributes of every vertex v adjacent to u but not in the tree, thereby maintaining the third part of the loop invariant. Whenever line 13 updates $v.key$, line 14 calls DECREASE-KEY to inform the min-priority queue that v's key has changed.

The running time of Prim's algorithm depends on the specific implementation of the min-priority queue Q. You can implement Q with a binary min-heap (see Chapter 6), including a way to map between vertices and their corresponding heap elements. The BUILD-MIN-HEAP procedure can perform lines 5–7 in $O(V)$ time. In fact, there is no need to call BUILD-MIN-HEAP. You can just put the key of r at the root of the min-heap, and because all other keys are ∞, they can go anywhere else in the min-heap. The body of the **while** loop executes $|V|$ times, and since each EXTRACT-MIN operation takes $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$. The **for** loop in lines 10–14 executes $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$. Within the **for** loop, the test for membership in Q in line 11 can take constant time if you keep a bit for each vertex that indicates whether it belongs to Q and update the bit when the vertex is removed from Q. Each call to DECREASE-KEY in line 14 takes $O(\lg V)$ time. Thus, the total time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asymptotically the same as for our implementation of Kruskal's algorithm.

You can further improve the asymptotic running time of Prim's algorithm by implementing the min-priority queue with a Fibonacci heap (see page 478). If a Fibonacci heap holds $|V|$ elements, an EXTRACT-MIN operation takes $O(\lg V)$ amortized time and each INSERT and DECREASE-KEY operation takes only $O(1)$ amortized time. Therefore, by using a Fibonacci heap to implement the min-priority queue Q, the running time of Prim's algorithm improves to $O(E + V \lg V)$.

Exercises

21.2-1
Kruskal's algorithm can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted. Show that for each minimum spanning tree T of G, there is a way to sort the edges of G in Kruskal's algorithm so that the algorithm returns T.

21.2-2
Give a simple implementation of Prim's algorithm that runs in $O(V^2)$ time when the graph $G = (V, E)$ is represented as an adjacency matrix.

21.2-3
For a sparse graph $G = (V, E)$, where $|E| = \Theta(V)$, is the implementation of Prim's algorithm with a Fibonacci heap asymptotically faster than the binary-heap implementation? What about for a dense graph, where $|E| = \Theta(V^2)$? How must the sizes $|E|$ and $|V|$ be related for the Fibonacci-heap implementation to be asymptotically faster than the binary-heap implementation?

21.2-4
Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

21.2-5
Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

21.2-6
Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1, and let E_2 be the set of edges that are incident only on vertices in V_2. Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2), and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.

Either argue that the algorithm correctly computes a minimum spanning tree of G, or provide an example for which the algorithm fails.

★ ***21.2-7***
Suppose that the edge weights in a graph are uniformly distributed over the half-open interval $[0, 1)$. Which algorithm, Kruskal's or Prim's, can you make run faster?

★ ***21.2-8***
Suppose that a graph G has a minimum spanning tree already computed. How quickly can you update the minimum spanning tree upon adding a new vertex and incident edges to G?

Problems

21-1 Second-best minimum spanning tree
Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.

We define a second-best minimum spanning tree as follows. Let $\mathcal{T}$ be the set of all spanning trees of G, and let T be a minimum spanning tree of G. Then a ***second-best minimum spanning tree*** is a spanning tree T' such that $w(T') = \min\{w(T'') : T'' \in \mathcal{T} - \{T\}\}$.

a. Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.

b. Let T be the minimum spanning tree of G. Prove that G contains some edge $(u, v) \in T$ and some edge $(x, y) \notin T$ such that $(T - \{(u, v)\}) \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.

c. Now let T be any spanning tree of G and, for any two vertices $u, v \in V$, let $max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(V^2)$-time algorithm that, given T, computes $max[u, v]$ for all $u, v \in V$.

d. Give an efficient algorithm to compute the second-best minimum spanning tree of G.

21-2 Minimum spanning tree in sparse graphs
For a very sparse connected graph $G = (V, E)$, it is possible to further improve upon the $O(E + V \lg V)$ running time of Prim's algorithm with a Fibonacci heap by preprocessing G to decrease the number of vertices before running Prim's algorithm. In particular, for each vertex u, choose the minimum-weight edge (u, v)

incident on u, and put (u, v) into the minimum spanning tree under construction. Then, contract all chosen edges (see Section B.4). Rather than contracting these edges one at a time, first identify sets of vertices that are united into the same new vertex. Then create the graph that would have resulted from contracting these edges one at a time, but do so by "renaming" edges according to the sets into which their endpoints were placed. Several edges from the original graph might be renamed the same as each other. In such a case, only one edge results, and its weight is the minimum of the weights of the corresponding original edges.

Initially, set the minimum spanning tree T being constructed to be empty, and for each edge $(u, v) \in E$, initialize the two attributes $(u, v).orig = (u, v)$ and $(u, v).c = w(u, v)$. Use the *orig* attribute to reference the edge from the initial graph that is associated with an edge in the contracted graph. The c attribute holds the weight of an edge, and as edges are contracted, it is updated according to the above scheme for choosing edge weights. The procedure MST-REDUCE on the facing page takes inputs G and T, and it returns a contracted graph G' with updated attributes $orig'$ and c'. The procedure also accumulates edges of G into the minimum spanning tree T.

a. Let T be the set of edges returned by MST-REDUCE, and let A be the minimum spanning tree of the graph G' formed by the call MST-PRIM(G', c', r), where c' is the weight attribute on the edges of $G'.E$ and r is any vertex in $G'.V$. Prove that $T \cup \{(x, y).orig' : (x, y) \in A\}$ is a minimum spanning tree of G.

b. Argue that $|G'.V| \le |V|/2$.

c. Show how to implement MST-REDUCE so that it runs in $O(E)$ time. (*Hint:* Use simple data structures.)

d. Suppose that you run k phases of MST-REDUCE, using the output G' produced by one phase as the input G to the next phase and accumulating edges in T. Argue that the overall running time of the k phases is $O(kE)$.

e. Suppose that after running k phases of MST-REDUCE, as in part (d), you run Prim's algorithm by calling MST-PRIM(G', c', r), where G', with weight attribute c', is returned by the last phase and r is any vertex in $G'.V$. Show how to pick k so that the overall running time is $O(E \lg\lg V)$. Argue that your choice of k minimizes the overall asymptotic running time.

f. For what values of $|E|$ (in terms of $|V|$) does Prim's algorithm with preprocessing asymptotically beat Prim's algorithm without preprocessing?

```
MST-REDUCE(G, T)
for each vertex v ∈ G.V
    v.mark = FALSE
    MAKE-SET(v)
for each vertex u ∈ G.V
    if u.mark == FALSE
        choose v ∈ G.Adj[u] such that (u, v).c is minimized
        UNION(u, v)
        T = T ∪ {(u, v).orig}
        u.mark = TRUE
        v.mark = TRUE
G'.V = {FIND-SET(v) : v ∈ G.V}
G'.E = ∅
for each edge (x, y) ∈ G.E
    u = FIND-SET(x)
    v = FIND-SET(y)
    if u ≠ v
        if (u, v) ∉ G'.E
            G'.E = G'.E ∪ {(u, v)}
            (u, v).orig' = (x, y).orig
            (u, v).c' = (x, y).c
        elseif (x, y).c < (u, v).c'
            (u, v).orig' = (x, y).orig
            (u, v).c' = (x, y).c
construct adjacency lists G'.Adj for G'
return G' and T
```

21-3 Alternative minimum-spanning-tree algorithms

Consider the three algorithms MAYBE-MST-A, MAYBE-MST-B, and MAYBE-MST-C on the next page. Each one takes a connected graph and a weight function as input and returns a set of edges T. For each algorithm, either prove that T is a minimum spanning tree or prove that T is not necessarily a minimum spanning tree. Also describe the most efficient implementation of each algorithm, regardless of whether it computes a minimum spanning tree.

21-4 Bottleneck spanning tree

A ***bottleneck spanning tree*** T of an undirected graph G is a spanning tree of G whose largest edge weight is minimum over all spanning trees of G. The value of the bottleneck spanning tree is the weight of the maximum-weight edge in T.

```
MAYBE-MST-A(G, w)
sort the edges into monotonically decreasing order of edge weights w
T = E
for each edge e, taken in monotonically decreasing order by weight
    if T − {e} is a connected graph
        T = T − {e}
return T
```

```
MAYBE-MST-B(G, w)
T = ∅
for each edge e, taken in arbitrary order
    if T ∪ {e} has no cycles
        T = T ∪ {e}
return T
```

```
MAYBE-MST-C(G, w)
T = ∅
for each edge e, taken in arbitrary order
    T = T ∪ {e}
    if T has a cycle c
        let e′ be a maximum-weight edge on c
        T = T − {e′}
return T
```

a. Argue that a minimum spanning tree is a bottleneck spanning tree.

Part (a) shows that finding a bottleneck spanning tree is no harder than finding a minimum spanning tree. In the remaining parts, you will show how to find a bottleneck spanning tree in linear time.

b. Give a linear-time algorithm that, given a graph G and an integer b, determines whether the value of the bottleneck spanning tree is at most b.

c. Use your algorithm for part (b) as a subroutine in a linear-time algorithm for the bottleneck-spanning-tree problem. (*Hint:* You might want to use a subroutine that contracts sets of edges, as in the MST-REDUCE procedure described in Problem 21-2.)

Chapter notes

Tarjan [429] surveys the minimum-spanning-tree problem and provides excellent advanced material. Graham and Hell [198] compiled a history of the minimum-spanning-tree problem.

Tarjan attributes the first minimum-spanning-tree algorithm to a 1926 paper by O. Borůvka. Borůvka's algorithm consists of running $O(\lg V)$ iterations of the procedure MST-REDUCE described in Problem 21-2. Kruskal's algorithm was reported by Kruskal [272] in 1956. The algorithm commonly known as Prim's algorithm was indeed invented by Prim [367], but it was also invented earlier by V. Jarník in 1930.

When $|E| = \Omega(V \lg V)$, Prim's algorithm, implemented with a Fibonacci heap, runs in $O(E)$ time. For sparser graphs, using a combination of the ideas from Prim's algorithm, Kruskal's algorithm, and Borůvka's algorithm, together with advanced data structures, Fredman and Tarjan [156] give an algorithm that runs in $O(E \lg^* V)$ time. Gabow, Galil, Spencer, and Tarjan [165] improved this algorithm to run in $O(E \lg \lg^* V)$ time. Chazelle [83] gives an algorithm that runs in $O(E\, \hat{\alpha}(E, V))$ time, where $\hat{\alpha}(E, V)$ is the functional inverse of Ackermann's function. (See the chapter notes for Chapter 19 for a brief discussion of Ackermann's function and its inverse.) Unlike previous minimum-spanning-tree algorithms, Chazelle's algorithm does not follow the greedy method. Pettie and Ramachandran [356] give an algorithm based on precomputed "MST decision trees" that also runs in $O(E\, \hat{\alpha}(E, V))$ time.

A related problem is ***spanning-tree verification***: given a graph $G = (V, E)$ and a tree $T \subseteq E$, determine whether T is a minimum spanning tree of G. King [254] gives a linear-time algorithm to verify a spanning tree, building on earlier work of Komlós [269] and Dixon, Rauch, and Tarjan [120].

The above algorithms are all deterministic and fall into the comparison-based model described in Chapter 8. Karger, Klein, and Tarjan [243] give a randomized minimum-spanning-tree algorithm that runs in $O(V + E)$ expected time. This algorithm uses recursion in a manner similar to the linear-time selection algorithm in Section 9.3: a recursive call on an auxiliary problem identifies a subset of the edges E' that cannot be in any minimum spanning tree. Another recursive call on $E - E'$ then finds the minimum spanning tree. The algorithm also uses ideas from Borůvka's algorithm and King's algorithm for spanning-tree verification.

Fredman and Willard [158] showed how to find a minimum spanning tree in $O(V + E)$ time using a deterministic algorithm that is not comparison based. Their algorithm assumes that the data are b-bit integers and that the computer memory consists of addressable b-bit words.

22 Single-Source Shortest Paths

Suppose that you need to drive from Oceanside, New York, to Oceanside, California, by the shortest possible route. Your GPS contains information about the entire road network of the United States, including the road distance between each pair of adjacent intersections. How can your GPS determine this shortest route?

One possible way is to enumerate all the routes from Oceanside, New York, to Oceanside, California, add up the distances on each route, and select the shortest. But even disallowing routes that contain cycles, your GPS would need to examine an enormous number of possibilities, most of which are simply not worth considering. For example, a route that passes through Miami, Florida, is a poor choice, because Miami is several hundred miles out of the way.

This chapter and Chapter 23 show how to solve such problems efficiently. The input to a ***shortest-paths problem*** is a weighted, directed graph $G = (V, E)$, with a weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights. The ***weight*** $w(p)$ of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

We define the ***shortest-path weight*** $\delta(u, v)$ from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v , \\ \infty & \text{otherwise} . \end{cases}$$

A ***shortest path*** from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$.

In the example of going from Oceanside, New York, to Oceanside, California, your GPS models the road network as a graph: vertices represent intersections, edges represent road segments between intersections, and edge weights represent road distances. The goal is to find a shortest path from a given intersection in

Oceanside, New York (say, Brower Avenue and Skillman Avenue) to a given intersection in Oceanside, California (say, Topeka Street and South Horne Street).

Edge weights can represent metrics other than distances, such as time, cost, penalties, loss, or any other quantity that accumulates linearly along a path and that you want to minimize.

The breadth-first-search algorithm from Section 20.2 is a shortest-paths algorithm that works on unweighted graphs, that is, graphs in which each edge has unit weight. Because many of the concepts from breadth-first search arise in the study of shortest paths in weighted graphs, you might want to review Section 20.2 before proceeding.

Variants

This chapter focuses on the ***single-source shortest-paths problem***: given a graph $G = (V, E)$, find a shortest path from a given ***source vertex*** $s \in V$ to every vertex $v \in V$. The algorithm for the single-source problem can solve many other problems, including the following variants.

Single-destination shortest-paths problem: Find a shortest path to a given ***destination vertex*** t from each vertex v. By reversing the direction of each edge in the graph, you can reduce this problem to a single-source problem.

Single-pair shortest-path problem: Find a shortest path from u to v for given vertices u and v. If you solve the single-source problem with source vertex u, you solve this problem also. Moreover, all known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms.

All-pairs shortest-paths problem: Find a shortest path from u to v for every pair of vertices u and v. Although you can solve this problem by running a single-source algorithm once from each vertex, you often can solve it faster. Additionally, its structure is interesting in its own right. Chapter 23 addresses the all-pairs problem in detail.

Optimal substructure of a shortest path

Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it. (The Edmonds-Karp maximum-flow algorithm in Chapter 24 also relies on this property.) Recall that optimal substructure is one of the key indicators that dynamic programming (Chapter 14) and the greedy method (Chapter 15) might apply. Dijkstra's algorithm, which we shall see in Section 22.3, is a greedy algorithm, and the Floyd-Warshall algorithm, which finds a shortest path between every pair of vertices (see Sec-

tion 23.2), is a dynamic-programming algorithm. The following lemma states the optimal-substructure property of shortest paths more precisely.

Lemma 22.1 (Subpaths of shortest paths are shortest paths)
Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

Proof Decompose path p into $v_0 \overset{p_{0i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$, so that $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_0 \overset{p_{0i}}{\leadsto} v_i \overset{p'_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$ is a path from v_0 to v_k whose weight $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_0 to v_k. ■

Negative-weight edges

Some instances of the single-source shortest-paths problem may include edges whose weights are negative. If the graph $G = (V, E)$ contains no negative-weight cycles reachable from the source s, then for all $v \in V$, the shortest-path weight $\delta(s, v)$ remains well defined, even if it has a negative value. If the graph contains a negative-weight cycle reachable from s, however, shortest-path weights are not well defined. No path from s to a vertex on the cycle can be a shortest path—you can always find a path with lower weight by following the proposed "shortest" path and then traversing the negative-weight cycle. If there is a negative-weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$.

Figure 22.1 illustrates the effect of negative weights and negative-weight cycles on shortest-path weights. Because there is only one path from s to a (the path $\langle s, a \rangle$), we have $\delta(s, a) = w(s, a) = 3$. Similarly, there is only one path from s to b, and so $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$. There are infinitely many paths from s to c: $\langle s, c \rangle$, $\langle s, c, d, c \rangle$, $\langle s, c, d, c, d, c \rangle$, and so on. Because the cycle $\langle c, d, c \rangle$ has weight $6 + (-3) = 3 > 0$, the shortest path from s to c is $\langle s, c \rangle$, with weight $\delta(s, c) = w(s, c) = 5$, and the shortest path from s to d is $\langle s, c, d \rangle$, with weight $\delta(s, d) = w(s, c) + w(c, d) = 11$. Analogously, there are infinitely many paths from s to e: $\langle s, e \rangle$, $\langle s, e, f, e \rangle$, $\langle s, e, f, e, f, e \rangle$, and so on. Because the cycle $\langle e, f, e \rangle$ has weight $3 + (-6) = -3 < 0$, however, there is no shortest path from s to e. By traversing the negative-weight cycle $\langle e, f, e \rangle$ arbitrarily many times, you can find paths from s to e with arbitrarily large negative weights, and so $\delta(s, e) = -\infty$. Similarly, $\delta(s, f) = -\infty$. Because g is reachable from f, you can also find paths with arbitrarily large negative weights from s to g,

Figure 22.1 Negative edge weights in a directed graph. The shortest-path weight from source s appears within each vertex. Because vertices e and f form a negative-weight cycle reachable from s, they have shortest-path weights of $-\infty$. Because vertex g is reachable from a vertex whose shortest-path weight is $-\infty$, it, too, has a shortest-path weight of $-\infty$. Vertices such as h, i, and j are not reachable from s, and so their shortest-path weights are ∞, even though they lie on a negative-weight cycle.

and so $\delta(s, g) = -\infty$. Vertices h, i, and j also form a negative-weight cycle. They are not reachable from s, however, and so $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

Some shortest-paths algorithms, such as Dijkstra's algorithm, assume that all edge weights in the input graph are nonnegative, as in a road network. Others, such as the Bellman-Ford algorithm, allow negative-weight edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source. Typically, if there is such a negative-weight cycle, the algorithm can detect and report its existence.

Cycles

Can a shortest path contain a cycle? As we have just seen, it cannot contain a negative-weight cycle. Nor can it contain a positive-weight cycle, since removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight. That is, if $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a path and $c = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a positive-weight cycle on this path (so that $v_i = v_j$ and $w(c) > 0$), then the path $p' = \langle v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k \rangle$ has weight $w(p') = w(p) - w(c) < w(p)$, and so p cannot be a shortest path from v_0 to v_k.

That leaves only 0-weight cycles. You can remove a 0-weight cycle from any path to produce another path whose weight is the same. Thus, if there is a shortest path from a source vertex s to a destination vertex v that contains a 0-weight cycle, then there is another shortest path from s to v without this cycle. As long as a shortest path has 0-weight cycles, you can repeatedly remove these cycles from the path until you have a shortest path that is cycle-free. Therefore, without loss of

generality, assume that shortest paths have no cycles, that is, they are simple paths. Since any acyclic path in a graph $G = (V, E)$ contains at most $|V|$ distinct vertices, it also contains at most $|V| - 1$ edges. Assume, therefore, that any shortest path contains at most $|V| - 1$ edges.

Representing shortest paths

It is usually not enough to compute only shortest-path weights. Most applications of shortest paths need to know the vertices on shortest paths as well. For example, if your GPS told you the distance to your destination but not how to get there, it would not be terribly useful. We represent shortest paths similarly to how we represented breadth-first trees in Section 20.2. Given a graph $G = (V, E)$, maintain for each vertex $v \in V$ a ***predecessor*** $v.\pi$ that is either another vertex or NIL. The shortest-paths algorithms in this chapter set the π attributes so that the chain of predecessors originating at a vertex v runs backward along a shortest path from s to v. Thus, given a vertex v for which $v.\pi \neq$ NIL, the procedure PRINT-PATH(G, s, v) from Section 20.2 prints a shortest path from s to v.

In the midst of executing a shortest-paths algorithm, however, the π values might not indicate shortest paths. The ***predecessor subgraph*** $G_\pi = (V_\pi, E_\pi)$ induced by the π values is defined the same for single-source shortest paths as for breadth-first search in equations (20.2) and (20.3) on page 561:

$$\begin{aligned} V_\pi &= \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\} \ , \\ E_\pi &= \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\} \ . \end{aligned}$$

We'll prove that the π values produced by the algorithms in this chapter have the property that at termination G_π is a "shortest-paths tree"—informally, a rooted tree containing a shortest path from the source s to every vertex that is reachable from s. A shortest-paths tree is like the breadth-first tree from Section 20.2, but it contains shortest paths from the source defined in terms of edge weights instead of numbers of edges. To be precise, let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, and assume that G contains no negative-weight cycles reachable from the source vertex $s \in V$, so that shortest paths are well defined. A ***shortest-paths tree*** rooted at s is a directed subgraph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, such that

1. V' is the set of vertices reachable from s in G,
2. G' forms a rooted tree with root s, and
3. for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Figure 22.2 **(a)** A weighted, directed graph with shortest-path weights from source s. **(b)** The blue edges form a shortest-paths tree rooted at the source s. **(c)** Another shortest-paths tree with the same root.

Shortest paths are not necessarily unique, and neither are shortest-paths trees. For example, Figure 22.2 shows a weighted, directed graph and two shortest-paths trees with the same root.

Relaxation

The algorithms in this chapter use the technique of ***relaxation***. For each vertex $v \in V$, the single-source shortest paths algorithms maintain an attribute $v.d$, which is an upper bound on the weight of a shortest path from source s to v. We call $v.d$ a ***shortest-path estimate***. To initialize the shortest-path estimates and predecessors, call the $\Theta(V)$-time procedure INITIALIZE-SINGLE-SOURCE. After initialization, we have $v.\pi = \text{NIL}$ for all $v \in V$, $s.d = 0$ and $v.d = \infty$ for $v \in V - \{s\}$.

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

The process of ***relaxing*** an edge (u, v) consists of testing whether going through vertex u improves the shortest path to vertex v found so far and, if so, updating $v.d$ and $v.\pi$. A relaxation step might decrease the value of the shortest-path estimate $v.d$ and update v's predecessor attribute $v.\pi$. The RELAX procedure on the following page performs a relaxation step on edge (u, v) in $O(1)$ time. Figure 22.3 shows two examples of relaxing an edge, one in which a shortest-path estimate decreases and one in which no estimate changes.

Figure 22.3 Relaxing an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex appears within the vertex. **(a)** Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases. **(b)** Since we have $v.d \le u.d + w(u, v)$ before relaxing the edge, the relaxation step leaves $v.d$ unchanged.

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

Each algorithm in this chapter calls INITIALIZE-SINGLE-SOURCE and then repeatedly relaxes edges.[1] Moreover, relaxation is the only means by which shortest-path estimates and predecessors change. The algorithms in this chapter differ in how many times they relax each edge and the order in which they relax edges. Dijkstra's algorithm and the shortest-paths algorithm for directed acyclic graphs relax each edge exactly once. The Bellman-Ford algorithm relaxes each edge $|V| - 1$ times.

Properties of shortest paths and relaxation

To prove the algorithms in this chapter correct, we'll appeal to several properties of shortest paths and relaxation. We state these properties here, and Section 22.5 proves them formally. For your reference, each property stated here includes the appropriate lemma or corollary number from Section 22.5. The latter five of these properties, which refer to shortest-path estimates or the predecessor subgraph, im-

[1] It may seem strange that the term "relaxation" is used for an operation that tightens an upper bound. The use of the term is historical. The outcome of a relaxation step can be viewed as a relaxation of the constraint $v.d \le u.d + w(u, v)$, which, by the triangle inequality (Lemma 22.10 on page 633), must be satisfied if $u.d = \delta(s, u)$ and $v.d = \delta(s, v)$. That is, if $v.d \le u.d + w(u, v)$, there is no "pressure" to satisfy this constraint, so the constraint is "relaxed."

plicitly assume that the graph is initialized with a call to INITIALIZE-SINGLE-SOURCE(G, s) and that the only way that shortest-path estimates and the predecessor subgraph change are by some sequence of relaxation steps.

Triangle inequality (Lemma 22.10)
For any edge $(u, v) \in E$, we have $\delta(s, v) \le \delta(s, u) + w(u, v)$.

Upper-bound property (Lemma 22.11)
We always have $v.d \ge \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 22.12)
If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.

Convergence property (Lemma 22.14)
If $s \leadsto u \to v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

Path-relaxation property (Lemma 22.15)
If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and the edges of p are relaxed in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Predecessor-subgraph property (Lemma 22.17)
Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Chapter outline

Section 22.1 presents the Bellman-Ford algorithm, which solves the single-source shortest-paths problem in the general case in which edges can have negative weight. The Bellman-Ford algorithm is remarkably simple, and it has the further benefit of detecting whether a negative-weight cycle is reachable from the source. Section 22.2 gives a linear-time algorithm for computing shortest paths from a single source in a directed acyclic graph. Section 22.3 covers Dijkstra's algorithm, which has a lower running time than the Bellman-Ford algorithm but requires the edge weights to be nonnegative. Section 22.4 shows how to use the Bellman-Ford algorithm to solve a special case of linear programming. Finally, Section 22.5 proves the properties of shortest paths and relaxation stated above.

This chapter does arithmetic with infinities, and so we need some conventions for when ∞ or $-\infty$ appears in an arithmetic expression. We assume that for any real number $a \ne -\infty$, we have $a + \infty = \infty + a = \infty$. Also, to make our proofs hold in the presence of negative-weight cycles, we assume that for any real number $a \ne \infty$, we have $a + (-\infty) = (-\infty) + a = -\infty$.

All algorithms in this chapter assume that the directed graph G is stored in the adjacency-list representation. Additionally, stored with each edge is its weight, so that as each algorithm traverses an adjacency list, it can find edge weights in $O(1)$ time per edge.

22.1 The Bellman-Ford algorithm

The ***Bellman-Ford algorithm*** solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph $G = (V, E)$ with source vertex s and weight function $w : E \to \mathbb{R}$, the Bellman-Ford algorithm returns a boolean value indicating whether there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

The procedure BELLMAN-FORD relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Figure 22.4 shows the execution of the Bellman-Ford algorithm on a graph with 5 vertices. After initializing the d and π values of all vertices in line 1, the algorithm makes $|V| - 1$ passes over the edges of the graph. Each pass is one iteration of the **for** loop of lines 2–4 and consists of relaxing each edge of the graph once. Figures 22.4(b)–(e) show the state of the algorithm after each of the four passes over the edges. After making $|V| - 1$ passes, lines 5–8 check for a negative-weight cycle and return the appropriate boolean value. (We'll see a little later why this check works.)

Figure 22.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and blue edges indicate predecessor values: if edge (u, v) is blue, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. Vertices whose shortest-path estimates and predecessors have changed due to a pass are highlighted in orange. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

The Bellman-Ford algorithm runs in $O(V^2 + VE)$ time when the graph is represented by adjacency lists, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges in lines 2–4 takes $\Theta(V + E)$ time (examining $|V|$ adjacency lists to find the $|E|$ edges), and the **for** loop of lines 5–7 takes $O(V + E)$ time. Fewer than $|V| - 1$ passes over the edges sometimes suffice (see Exercise 22.1-3), which is why we say $O(V^2 + VE)$ time, rather than $\Theta(V^2 + VE)$ time. In the frequent case where $|E| = \Omega(V)$, we can express this running time as $O(VE)$. Exercise 22.1-5 asks you to make the Bellman-Ford algorithm run in $O(VE)$ time even when $|E| = o(V)$.

To prove the correctness of the Bellman-Ford algorithm, we start by showing that if there are no negative-weight cycles, the algorithm computes correct shortest-path weights for all vertices reachable from the source.

Lemma 22.2
Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight function $w : E \to \mathbb{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, after the $|V| - 1$ iterations of the **for** loop of lines 2–4 of BELLMAN-FORD, $v.d = \delta(s, v)$ for all vertices v that are reachable from s.

Proof We prove the lemma by appealing to the path-relaxation property. Consider any vertex v that is reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Because shortest paths are simple, p has at most $|V| - 1$ edges, and so $k \le |V| - 1$. Each of the $|V| - 1$ iterations of the **for** loop of lines 2–4 relaxes all $|E|$ edges. Among the edges relaxed in the ith iteration, for $i = 1, 2, \ldots, k$, is (v_{i-1}, v_i). By the path-relaxation property, therefore, $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$. ■

Corollary 22.3
Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight function $w : E \to \mathbb{R}$. Then, for each vertex $v \in V$, there is a path from s to v if and only if BELLMAN-FORD terminates with $v.d < \infty$ when it is run on G.

Proof The proof is left as Exercise 22.1-2. ■

Theorem 22.4 (Correctness of the Bellman-Ford algorithm)
Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source vertex s and weight function $w : E \to \mathbb{R}$. If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree rooted at s. If G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof Suppose that graph G contains no negative-weight cycles that are reachable from the source s. We first prove the claim that at termination, $v.d = \delta(s, v)$ for all vertices $v \in V$. If vertex v is reachable from s, then Lemma 22.2 proves this claim. If v is not reachable from s, then the claim follows from the no-path property. Thus, the claim is proven. The predecessor-subgraph property, along with the claim, implies that G_π is a shortest-paths tree. Now we use the claim to show that BELLMAN-FORD returns TRUE. At termination, for all edges $(u, v) \in E$ we have

$$\begin{aligned} v.d &= \delta(s, v) \\ &\le \delta(s, u) + w(u, v) \quad \text{(by the triangle inequality)} \\ &= u.d + w(u, v) , \end{aligned}$$

and so none of the tests in line 6 causes BELLMAN-FORD to return FALSE. Therefore, it returns TRUE.

Now, suppose that graph G contains a negative-weight cycle reachable from the source s. Let this cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$, in which case we have

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 . \tag{22.1}$$

Assume for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE. Thus, $v_i.d \le v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$. Summing the inequalities around cycle c gives

$$\begin{aligned}\sum_{i=1}^{k} v_i.d &\le \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) .\end{aligned}$$

Since $v_0 = v_k$, each vertex in c appears exactly once in each of the summations $\sum_{i=1}^{k} v_i.d$ and $\sum_{i=1}^{k} v_{i-1}.d$, and so

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d .$$

Moreover, by Corollary 22.3, $v_i.d$ is finite for $i = 1, 2, \ldots, k$. Thus,

$$0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i) ,$$

which contradicts inequality (22.1). We conclude that the Bellman-Ford algorithm returns TRUE if graph G contains no negative-weight cycles reachable from the source, and FALSE otherwise. ■

Exercises

22.1-1

Run the Bellman-Ford algorithm on the directed graph of Figure 22.4, using vertex z as the source. In each pass, relax edges in the same order as in the figure, and show the d and π values after each pass. Now, change the weight of edge (z, x) to 4 and run the algorithm again, using s as the source.

22.1-2

Prove Corollary 22.3.

22.1-3
Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v. (Here, the shortest path is by weight, not the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes, even if m is not known in advance.

22.1-4
Modify the Bellman-Ford algorithm so that it sets $v.d$ to $-\infty$ for all vertices v for which there is a negative-weight cycle on some path from the source to v.

22.1-5
Suppose that the graph given as input to the Bellman-Ford algorithm is represented with a list of $|E|$ edges, where each edge indicates the vertices it leaves and enters, along with its weight. Argue that the Bellman-Ford algorithm runs in $O(VE)$ time without the constraint that $|E| = \Omega(V)$. Modify the Bellman-Ford algorithm so that it runs in $O(VE)$ time in all cases when the input graph is represented with adjacency lists.

22.1-6
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$. Give an $O(VE)$-time algorithm to find, for all vertices $v \in V$, the value $\delta^*(v) = \min\{\delta(u, v) : u \in V\}$.

22.1-7
Suppose that a weighted, directed graph $G = (V, E)$ contains a negative-weight cycle. Give an efficient algorithm to list the vertices of one such cycle. Prove that your algorithm is correct.

22.2 Single-source shortest paths in directed acyclic graphs

In this section, we introduce one further restriction on weighted, directed graphs: they are acyclic. That is, we are concerned with weighted dags. Shortest paths are always well defined in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist. We'll see that if the edges of a weighted dag $G = (V, E)$ are relaxed according to a topological sort of its vertices, it takes only $\Theta(V + E)$ time to compute shortest paths from a single source.

The algorithm starts by topologically sorting the dag (see Section 20.4) to impose a linear ordering on the vertices. If the dag contains a path from vertex u to vertex v, then u precedes v in the topological sort. The DAG-SHORTEST-PATHS

procedure makes just one pass over the vertices in the topologically sorted order. As it processes each vertex, it relaxes each edge that leaves the vertex. Figure 22.5 shows the execution of this algorithm.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u ∈ G.V, taken in topologically sorted order
4      for each vertex v in G.Adj[u]
5          RELAX(u, v, w)
```

Let's analyze the running time of this algorithm. As shown in Section 20.4, the topological sort of line 1 takes $\Theta(V + E)$ time. The call of INITIALIZE-SINGLE-SOURCE in line 2 takes $\Theta(V)$ time. The **for** loop of lines 3–5 makes one iteration per vertex. Altogether, the **for** loop of lines 4–5 relaxes each edge exactly once. (We have used an aggregate analysis here.) Because each iteration of the inner **for** loop takes $\Theta(1)$ time, the total running time is $\Theta(V + E)$, which is linear in the size of an adjacency-list representation of the graph.

The following theorem shows that the DAG-SHORTEST-PATHS procedure correctly computes the shortest paths.

Theorem 22.5
If a weighted, directed graph $G = (V, E)$ has source vertex s and no cycles, then at the termination of the DAG-SHORTEST-PATHS procedure, $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree.

Proof We first show that $v.d = \delta(s, v)$ for all vertices $v \in V$ at termination. If v is not reachable from s, then $v.d = \delta(s, v) = \infty$ by the no-path property. Now, suppose that v is reachable from s, so that there is a shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$. Because DAG-SHORTEST-PATHS processes the vertices in topologically sorted order, it relaxes the edges on p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. The path-relaxation property implies that $v_i.d = \delta(s, v_i)$ at termination for $i = 0, 1, \ldots, k$. Finally, by the predecessor-subgraph property, G_π is a shortest-paths tree. ■

A useful application of this algorithm arises in determining critical paths in ***PERT chart***[2] analysis. A job consists of several tasks. Each task takes a certain

[2] "PERT" is an acronym for "program evaluation and review technique."

Figure 22.5 The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is s. The d values appear within the vertices, and blue edges indicate the π values. **(a)** The situation before the first iteration of the **for** loop of lines 3–5. **(b)–(g)** The situation after each iteration of the **for** loop of lines 3–5. Blue vertices have had their outgoing edges relaxed. The vertex highlighted in orange was used as u in that iteration. Each edge highlighted in orange caused a d value to change when it was relaxed in that iteration. The values shown in part (g) are the final values.

amount of time, and some tasks must be completed before others can be started. For example, if the job is to build a house, then the foundation must be completed before starting to frame the exterior walls, which must be completed before starting on the roof. Some tasks require more than one other task to be completed before they can be started: before the drywall can be installed over the wall framing, both the electrical system and plumbing must be installed. A dag models the tasks and dependencies. Edges represent tasks, with the weight of an edge indicating the time required to perform the task. Vertices represent "milestones," which are

achieved when all the tasks represented by the edges entering the vertex have been completed. If edge (u, v) enters vertex v and edge (v, x) leaves v, then task (u, v) must be completed before task (v, x) is started. A path through this dag represents a sequence of tasks that must be performed in a particular order. A ***critical path*** is a *longest* path through the dag, corresponding to the longest time to perform any sequence of tasks. Thus, the weight of a critical path provides a lower bound on the total time to perform all the tasks, even if as many tasks as possible are performed simultaneously. You can find a critical path by either

- negating the edge weights and running DAG-SHORTEST-PATHS, or
- running DAG-SHORTEST-PATHS, but replacing "∞" by "$-\infty$" in line 2 of INITIALIZE-SINGLE-SOURCE and "$>$" by "$<$" in the RELAX procedure.

Exercises

22.2-1
Show the result of running DAG-SHORTEST-PATHS on the directed acyclic graph of Figure 22.5, using vertex r as the source.

22.2-2
Suppose that you change line 3 of DAG-SHORTEST-PATHS to read

3 **for** the first $|V| - 1$ vertices, taken in topologically sorted order

Show that the procedure remains correct.

22.2-3
An alternative way to represent a PERT chart looks more like the dag of Figure 20.7 on page 574. Vertices represent tasks and edges represent sequencing constraints, that is, edge (u, v) indicates that task u must be performed before task v. Vertices, not edges, have weights. Modify the DAG-SHORTEST-PATHS procedure so that it finds a longest path in a directed acyclic graph with weighted vertices in linear time.

★ ***22.2-4***
Give an efficient algorithm to count the total number of paths in a directed acyclic graph. The count should include all paths between all pairs of vertices and all paths with 0 edges. Analyze your algorithm.

22.3 Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$, but it requires nonnegative weights on all edges: $w(u, v) \geq 0$ for each edge $(u, v) \in E$. As we shall see, with a good implementation, the running time of Dijkstra's algorithm is lower than that of the Bellman-Ford algorithm.

You can think of Dijkstra's algorithm as generalizing breadth-first search to weighted graphs. A wave emanates from the source, and the first time that a wave arrives at a vertex, a new wave emanates from that vertex. Whereas breadth-first search operates as if each wave takes unit time to traverse an edge, in a weighted graph, the time for a wave to traverse an edge is given by the edge's weight. Because a shortest path in a weighted graph might not have the fewest edges, a simple, first-in, first-out queue won't suffice for choosing the next vertex from which to send out a wave.

Instead, Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u into S, and relaxes all edges leaving u. The procedure DIJKSTRA replaces the first-in, first-out queue of breadth-first search by a min-priority queue Q of vertices, keyed by their d values.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = ∅
4  for each vertex u ∈ G.V
5      INSERT(Q, u)
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      S = S ∪ {u}
9      for each vertex v in G.Adj[u]
10         RELAX(u, v, w)
11         if the call of RELAX decreased v.d
12             DECREASE-KEY(Q, v, v.d)
```

Dijkstra's algorithm relaxes edges as shown in Figure 22.6. Line 1 initializes the d and π values in the usual way, and line 2 initializes the set S to the empty set. The algorithm maintains the invariant that $Q = V - S$ at the start of each iteration

Figure 22.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and blue edges indicate predecessor values. Blue vertices belong to the set S, and tan vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 6–12. **(b)–(f)** The situation after each successive iteration of the **while** loop. In each part, the vertex highlighted in orange was chosen as vertex u in line 7, and each edge highlighted in orange caused a d value and a predecessor to change when the edge was relaxed. The d values and predecessors shown in part (f) are the final values.

of the **while** loop of lines 6–12. Lines 3–5 initialize the min-priority queue Q to contain all the vertices in V. Since $S = \emptyset$ at that time, the invariant is true upon first reaching line 6. Each time through the **while** loop of lines 6–12, line 7 extracts a vertex u from $Q = V - S$ and line 8 adds it to set S, thereby maintaining the invariant. (The first time through this loop, $u = s$.) Vertex u, therefore, has the smallest shortest-path estimate of any vertex in $V - S$. Then, lines 9–12 relax each edge (u, v) leaving u, thus updating the estimate $v.d$ and the predecessor $v.\pi$ if the shortest path to v found so far improves by going through u. Whenever a relaxation step changes the d and π values, the call to DECREASE-KEY in line 12 updates the min-priority queue. The algorithm never inserts vertices into Q after the **for** loop of lines 4–5, and each vertex is extracted from Q and added to S exactly once, so that the **while** loop of lines 6–12 iterates exactly $|V|$ times.

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, you can think of it as using a greedy strategy. Chapter 15 explains greedy strategies in detail, but you need not have read that chapter to understand Dijkstra's algorithm. Greedy strategies do not always yield optimal

Figure 22.7 The proof of Theorem 22.6. Vertex u is selected to be added into set S in line 7 of DIJKSTRA. Vertex y is the first vertex on a shortest path from the source s to vertex u that is not in set S, and $x \in S$ is y's predecessor on that shortest path. The subpath from y to u may or may not re-enter set S.

results in general, but as the following theorem and its corollary show, Dijkstra's algorithm does indeed compute shortest paths. The key is to show that $u.d = \delta(s, u)$ each time it adds a vertex u to set S.

Theorem 22.6 (Correctness of Dijkstra's algorithm)
Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with nonnegative weight function w and source vertex s, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

Proof We will show that at the start of each iteration of the **while** loop of lines 6–12, we have $v.d = \delta(s, v)$ for all $v \in S$. The algorithm terminates when $S = V$, so that $v.d = \delta(s, v)$ for all $v \in V$.

The proof is by induction on the number of iterations of the **while** loop, which equals $|S|$ at the start of each iteration. There are two bases: for $|S| = 0$, so that $S = \emptyset$ and the claim is trivially true, and for $|S| = 1$, so that $S = \{s\}$ and $s.d = \delta(s, s) = 0$.

For the inductive step, the inductive hypothesis is that $v.d = \delta(s, v)$ for all $v \in S$. The algorithm extracts vertex u from $V - S$. Because the algorithm adds u into S, we need to show that $u.d = \delta(s, u)$ at that time. If there is no path from s to u, then we are done, by the no-path property. If there is a path from s to u, then, as Figure 22.7 shows, let y be the first vertex on a shortest path from s to u that is not in S, and let $x \in S$ be the predecessor of y on that shortest path. (We could have $y = u$ or $x = s$.) Because y appears no later than u on the shortest path and all edge weights are nonnegative, we have $\delta(s, y) \le \delta(s, u)$. Because the call of EXTRACT-MIN in line 7 returned u as having the minimum d value in $V - S$, we also have $u.d \le y.d$, and the upper-bound property gives $\delta(s, u) \le u.d$.

Since $x \in S$, the inductive hypothesis implies that $x.d = \delta(s, x)$. During the iteration of the **while** loop that added x into S, edge (x, y) was relaxed. By the convergence property, $y.d$ received the value of $\delta(s, y)$ at that time. Thus, we have

$$\delta(s, y) \le \delta(s, u) \le u.d \le y.d \quad \text{and} \quad y.d = \delta(s, y) ,$$

so that

$$\delta(s, y) = \delta(s, u) = u.d = y.d .$$

Hence, $u.d = \delta(s, u)$, and by the upper-bound property, this value never changes again. ■

Corollary 22.7
After Dijkstra's algorithm is run on a weighted, directed graph $G = (V, E)$ with nonnegative weight function w and source vertex s, the predecessor subgraph G_π is a shortest-paths tree rooted at s.

Proof Immediate from Theorem 22.6 and the predecessor-subgraph property. ■

Analysis

How fast is Dijkstra's algorithm? It maintains the min-priority queue Q by calling three priority-queue operations: INSERT (in line 5), EXTRACT-MIN (in line 7), and DECREASE-KEY (in line 12). The algorithm calls both INSERT and EXTRACT-MIN once per vertex. Because each vertex $u \in V$ is added to set S exactly once, each edge in the adjacency list $Adj[u]$ is examined in the **for** loop of lines 9–12 exactly once during the course of the algorithm. Since the total number of edges in all the adjacency lists is $|E|$, this **for** loop iterates a total of $|E|$ times, and thus the algorithm calls DECREASE-KEY at most $|E|$ times overall. (Observe once again that we are using aggregate analysis.)

Just as in Prim's algorithm, the running time of Dijkstra's algorithm depends on the specific implementation of the min-priority queue Q. A simple implementation takes advantage of the vertices being numbered 1 to $|V|$: simply store $v.d$ in the vth entry of an array. Each INSERT and DECREASE-KEY operation takes $O(1)$ time, and each EXTRACT-MIN operation takes $O(V)$ time (since it has to search through the entire array), for a total time of $O(V^2 + E) = O(V^2)$.

If the graph is sufficiently sparse—in particular, $E = o(V^2/\lg V)$—you can improve the running time by implementing the min-priority queue with a binary min-heap that includes a way to map between vertices and their corresponding heap elements. Each EXTRACT-MIN operation then takes $O(\lg V)$ time. As before, there are $|V|$ such operations. The time to build the binary min-heap is $O(V)$. (As noted in Section 21.2, you don't even need to call BUILD-MIN-HEAP.) Each DECREASE-KEY operation takes $O(\lg V)$ time, and there are still at most $|E|$ such operations. The total running time is therefore $O((V + E)\lg V)$, which is $O(E \lg V)$ in the typical case that $|E| = \Omega(V)$. This running time improves upon the straightforward $O(V^2)$-time implementation if $E = o(V^2/\lg V)$.

By implementing the min-priority queue with a Fibonacci heap (see page 478), you can improve the running time to $O(V \lg V + E)$. The amortized cost of each of the $|V|$ EXTRACT-MIN operations is $O(\lg V)$, and each DECREASE-KEY call, of which there are at most $|E|$, takes only $O(1)$ amortized time. Historically, the development of Fibonacci heaps was motivated by the observation that Dijkstra's algorithm typically makes many more DECREASE-KEY calls than EXTRACT-MIN calls, so that any method of reducing the amortized time of each DECREASE-KEY operation to $o(\lg V)$ without increasing the amortized time of EXTRACT-MIN would yield an asymptotically faster implementation than with binary heaps.

Dijkstra's algorithm resembles both breadth-first search (see Section 20.2) and Prim's algorithm for computing minimum spanning trees (see Section 21.2). It is like breadth-first search in that set S corresponds to the set of black vertices in a breadth-first search. Just as vertices in S have their final shortest-path weights, so do black vertices in a breadth-first search have their correct breadth-first distances. Dijkstra's algorithm is like Prim's algorithm in that both algorithms use a min-priority queue to find the "lightest" vertex outside a given set (the set S in Dijkstra's algorithm and the tree being grown in Prim's algorithm), add this vertex into the set, and adjust the weights of the remaining vertices outside the set accordingly.

Exercises

22.3-1

Run Dijkstra's algorithm on the directed graph of Figure 22.2, first using vertex s as the source and then using vertex z as the source. In the style of Figure 22.6, show the d and π values and the vertices in set S after each iteration of the **while** loop.

22.3-2

Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces an incorrect answer. Why doesn't the proof of Theorem 22.6 go through when negative-weight edges are allowed?

22.3-3

Suppose that you change line 6 of Dijkstra's algorithm to read

```
6  while |Q| > 1
```

This change causes the **while** loop to execute $|V| - 1$ times instead of $|V|$ times. Is this proposed algorithm correct?

22.3-4

Modify the DIJKSTRA procedure so that the priority queue Q is more like the queue in the BFS procedure in that it contains only vertices that have been reached from source s so far: $Q \subseteq V - S$ and $v \in Q$ implies $v.d \neq \infty$.

22.3-5

Professor Gaedel has written a program that he claims implements Dijkstra's algorithm. The program produces $v.d$ and $v.\pi$ for each vertex $v \in V$. Give an $O(V + E)$-time algorithm to check the output of the professor's program. It should determine whether the d and π attributes match those of some shortest-paths tree. You may assume that all edge weights are nonnegative.

22.3-6

Professor Newman thinks that he has worked out a simpler proof of correctness for Dijkstra's algorithm. He claims that Dijkstra's algorithm relaxes the edges of every shortest path in the graph in the order in which they appear on the path, and therefore the path-relaxation property applies to every vertex reachable from the source. Show that the professor is mistaken by constructing a directed graph for which Dijkstra's algorithm relaxes the edges of a shortest path out of order.

22.3-7

Consider a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \leq r(u, v) \leq 1$ that represents the reliability of a communication channel from vertex u to vertex v. Interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

22.3-8

Let $G = (V, E)$ be a weighted, directed graph with positive weight function $w : E \to \{1, 2, \ldots, W\}$ for some positive integer W, and assume that no two vertices have the same shortest-path weights from source vertex s. Now define an unweighted, directed graph $G' = (V \cup V', E')$ by replacing each edge $(u, v) \in E$ with $w(u, v)$ unit-weight edges in series. How many vertices does G' have? Now suppose that you run a breadth-first search on G'. Show that the order in which the breadth-first search of G' colors vertices in V black is the same as the order in which Dijkstra's algorithm extracts the vertices of V from the priority queue when it runs on G.

22.3-9

Let $G = (V, E)$ be a weighted, directed graph with nonnegative weight function $w : E \to \{0, 1, \ldots, W\}$ for some nonnegative integer W. Modify Dijkstra's algo-

rithm to compute the shortest paths from a given source vertex s in $O(WV + E)$ time.

22.3-10
Modify your algorithm from Exercise 22.3-9 to run in $O((V + E) \lg W)$ time. (*Hint:* How many distinct shortest-path estimates can $V - S$ contain at any point in time?)

22.3-11
Suppose that you are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra's algorithm correctly finds shortest paths from s in this graph.

22.3-12
Suppose that you have a weighted directed graph $G = (V, E)$ in which all edge weights are positive real values in the range $[C, 2C]$ for some positive constant C. Modify Dijkstra's algorithm so that it runs in $O(V + E)$ time.

22.4 Difference constraints and shortest paths

Chapter 29 studies the general linear-programming problem, showing how to optimize a linear function subject to a set of linear inequalities. This section investigates a special case of linear programming that reduces to finding shortest paths from a single source. The Bellman-Ford algorithm then solves the resulting single-source shortest-paths problem, thereby also solving the linear-programming problem.

Linear programming

In the general ***linear-programming problem***, the input is an $m \times n$ matrix A, an m-vector b, and an n-vector c. The goal is to find a vector x of n elements that maximizes the ***objective function*** $\sum_{i=1}^{n} c_i x_i$ subject to the m constraints given by $Ax \le b$.

The most popular method for solving linear programs is the ***simplex algorithm***, which Section 29.1 discusses. Although the simplex algorithm does not always run in time polynomial in the size of its input, there are other linear-programming algorithms that do run in polynomial time. We offer here two reasons to understand the setup of linear-programming problems. First, if you know that you can cast a given problem as a polynomial-sized linear-programming problem, then you im-

mediately have a polynomial-time algorithm to solve the problem. Second, faster algorithms exist for many special cases of linear programming. For example, the single-pair shortest-path problem (Exercise 22.4-4) and the maximum-flow problem (Exercise 24.1-5) are special cases of linear programming.

Sometimes the objective function does not matter: it's enough just to find any ***feasible solution***, that is, any vector x that satisfies $Ax \le b$, or to determine that no feasible solution exists. This section focuses on one such ***feasibility problem***.

Systems of difference constraints

In a ***system of difference constraints***, each row of the linear-programming matrix A contains one 1 and one -1, and all other entries of A are 0. Thus, the constraints given by $Ax \le b$ are a set of m ***difference constraints*** involving n unknowns, in which each constraint is a simple linear inequality of the form

$$x_j - x_i \le b_k ,$$

where $1 \le i, j \le n, i \ne j$, and $1 \le k \le m$.

For example, consider the problem of finding a 5-vector $x = (x_i)$ that satisfies

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \le \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix} .$$

This problem is equivalent to finding values for the unknowns x_1, x_2, x_3, x_4, x_5, satisfying the following 8 difference constraints:

$$x_1 - x_2 \le 0 , \tag{22.2}$$
$$x_1 - x_5 \le -1 , \tag{22.3}$$
$$x_2 - x_5 \le 1 , \tag{22.4}$$
$$x_3 - x_1 \le 5 , \tag{22.5}$$
$$x_4 - x_1 \le 4 , \tag{22.6}$$
$$x_4 - x_3 \le -1 , \tag{22.7}$$
$$x_5 - x_3 \le -3 , \tag{22.8}$$
$$x_5 - x_4 \le -3 . \tag{22.9}$$

One solution to this problem is $x = (-5, -3, 0, -1, -4)$, which you can verify directly by checking each inequality. In fact, this problem has more than one solution.

Another is $x' = (0, 2, 5, 4, 1)$. These two solutions are related: each component of x' is 5 larger than the corresponding component of x. This fact is not mere coincidence.

Lemma 22.8
Let $x = (x_1, x_2, \ldots, x_n)$ be a solution to a system $Ax \le b$ of difference constraints, and let d be any constant. Then $x + d = (x_1 + d, x_2 + d, \ldots, x_n + d)$ is a solution to $Ax \le b$ as well.

Proof For each x_i and x_j, we have $(x_j + d) - (x_i + d) = x_j - x_i$. Thus, if x satisfies $Ax \le b$, so does $x + d$. ■

Systems of difference constraints occur in various applications. For example, the unknowns x_i might be times at which events are to occur. Each constraint states that at least a certain amount of time, or at most a certain amount of time, must elapse between two events. Perhaps the events are jobs to be performed during the assembly of a product. If the manufacturer applies an adhesive that takes 2 hours to set at time x_1 and has to wait until it sets to install a part at time x_2, then there is a constraint that $x_2 \ge x_1 + 2$ or, equivalently, that $x_1 - x_2 \le -2$. Alternatively, the manufacturer might require the part to be installed after the adhesive has been applied but no later than the time that the adhesive has set halfway. In this case, there is a pair of constraints $x_2 \ge x_1$ and $x_2 \le x_1 + 1$ or, equivalently, $x_1 - x_2 \le 0$ and $x_2 - x_1 \le 1$.

If all the constraints have nonnegative numbers on the right-hand side—that is, if $b_i \ge 0$ for $i = 1, 2, \ldots, m$—then finding a feasible solution is trivial: just set all the unknowns x_i equal to each other. Then all the differences are 0, and every constraint is satisfied. The problem of finding a feasible solution to a system of difference constraints is interesting only if at least one constraint has $b_i < 0$.

Constraint graphs

We can interpret systems of difference constraints from a graph-theoretic point of view. For a system $Ax \le b$ of difference constraints, let's view the $m \times n$ linear-programming matrix A as the transpose of an incidence matrix (see Exercise 20.1-7) for a graph with n vertices and m edges. Each vertex v_i in the graph, for $i = 1, 2, \ldots, n$, corresponds to one of the n unknown variables x_i. Each directed edge in the graph corresponds to one of the m inequalities involving two unknowns.

More formally, given a system $Ax \le b$ of difference constraints, the corresponding ***constraint graph*** is a weighted, directed graph $G = (V, E)$, where

Figure 22.8 The constraint graph corresponding to the system (22.2)–(22.9) of difference constraints. The value of $\delta(v_0, v_i)$ appears in each vertex v_i. One feasible solution to the system is $x = (-5, -3, 0, -1, -4)$.

$$V = \{v_0, v_1, \ldots, v_n\}$$

and

$$E = \{(v_i, v_j) : x_j - x_i \le b_k \text{ is a constraint}\}$$
$$\cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_n)\} \ .$$

The constraint graph includes the additional vertex v_0, as we shall see shortly, to guarantee that the graph has some vertex that can reach all other vertices. Thus, the vertex set V consists of a vertex v_i for each unknown x_i, plus an additional vertex v_0. The edge set E contains an edge for each difference constraint, plus an edge (v_0, v_i) for each unknown x_i. If $x_j - x_i \le b_k$ is a difference constraint, then the weight of edge (v_i, v_j) is $w(v_i, v_j) = b_k$. The weight of each edge leaving v_0 is 0. Figure 22.8 shows the constraint graph for the system (22.2)–(22.9) of difference constraints.

The following theorem shows how to solve a system of difference constraints by finding shortest-path weights in the corresponding constraint graph.

Theorem 22.9

Given a system $Ax \le b$ of difference constraints, let $G = (V, E)$ be the corresponding constraint graph. If G contains no negative-weight cycles, then

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \ldots, \delta(v_0, v_n)) \tag{22.10}$$

is a feasible solution for the system. If G contains a negative-weight cycle, then there is no feasible solution for the system.

Proof We first show that if the constraint graph contains no negative-weight cycles, then equation (22.10) gives a feasible solution. Consider any edge $(v_i, v_j) \in E$. The triangle inequality implies that $\delta(v_0, v_j) \le \delta(v_0, v_i) + w(v_i, v_j)$, which is equivalent to $\delta(v_0, v_j) - \delta(v_0, v_i) \le w(v_i, v_j)$. Thus, letting $x_i = \delta(v_0, v_i)$ and $x_j = \delta(v_0, v_j)$ satisfies the difference constraint $x_j - x_i \le w(v_i, v_j)$ that corresponds to edge (v_i, v_j).

Now we show that if the constraint graph contains a negative-weight cycle, then the system of difference constraints has no feasible solution. Without loss of generality, let the negative-weight cycle be $c = \langle v_1, v_2, \ldots, v_k \rangle$, where $v_1 = v_k$. (The vertex v_0 cannot be on cycle c, because it has no entering edges.) Cycle c corresponds to the following difference constraints:

$$\begin{aligned}
x_2 - x_1 &\le w(v_1, v_2) , \\
x_3 - x_2 &\le w(v_2, v_3) , \\
&\vdots \\
x_{k-1} - x_{k-2} &\le w(v_{k-2}, v_{k-1}) , \\
x_k - x_{k-1} &\le w(v_{k-1}, v_k) .
\end{aligned}$$

We'll assume that x has a solution satisfying each of these k inequalities and then derive a contradiction. The solution must also satisfy the inequality that results from summing the k inequalities together. In summing the left-hand sides, each unknown x_i is added in once and subtracted out once (remember that $v_1 = v_k$ implies $x_1 = x_k$), so that the left-hand side sums to 0. The right-hand side sums to the weight $w(c)$ of the cycle, giving $0 \le w(c)$. But since c is a negative-weight cycle, $w(c) < 0$, and we obtain the contradiction that $0 \le w(c) < 0$. ■

Solving systems of difference constraints

Theorem 22.9 suggests how to use the Bellman-Ford algorithm to solve a system of difference constraints. Because the constraint graph contains edges from the source vertex v_0 to all other vertices, any negative-weight cycle in the constraint graph is reachable from v_0. If the Bellman-Ford algorithm returns TRUE, then the shortest-path weights give a feasible solution to the system. In Figure 22.8, for example, the shortest-path weights provide the feasible solution $x = (-5, -3, 0, -1, -4)$, and by Lemma 22.8, $x = (d - 5, d - 3, d, d - 1, d - 4)$ is also a feasible solution for any constant d. If the Bellman-Ford algorithm returns FALSE, there is no feasible solution to the system of difference constraints.

A system of difference constraints with m constraints on n unknowns produces a graph with $n + 1$ vertices and $n + m$ edges. Thus, the Bellman-Ford algorithm provides a way to solve the system in $O((n + 1)(n + m)) = O(n^2 + nm)$ time.

Exercise 22.4-5 asks you to modify the algorithm to run in $O(nm)$ time, even if m is much less than n.

Exercises

22.4-1

Find a feasible solution or determine that no feasible solution exists for the following system of difference constraints:

$$\begin{aligned}
x_1 - x_2 &\le 1 , \\
x_1 - x_4 &\le -4 , \\
x_2 - x_3 &\le 2 , \\
x_2 - x_5 &\le 7 , \\
x_2 - x_6 &\le 5 , \\
x_3 - x_6 &\le 10 , \\
x_4 - x_2 &\le 2 , \\
x_5 - x_1 &\le -1 , \\
x_5 - x_4 &\le 3 , \\
x_6 - x_3 &\le -8 .
\end{aligned}$$

22.4-2

Find a feasible solution or determine that no feasible solution exists for the following system of difference constraints:

$$\begin{aligned}
x_1 - x_2 &\le 4 , \\
x_1 - x_5 &\le 5 , \\
x_2 - x_4 &\le -6 , \\
x_3 - x_2 &\le 1 , \\
x_4 - x_1 &\le 3 , \\
x_4 - x_3 &\le 5 , \\
x_4 - x_5 &\le 10 , \\
x_5 - x_3 &\le -4 , \\
x_5 - x_4 &\le -8 .
\end{aligned}$$

22.4-3

Can any shortest-path weight from the new vertex v_0 in a constraint graph be positive? Explain.

22.4-4

Express the single-pair shortest-path problem as a linear program.

22.4-5
Show how to modify the Bellman-Ford algorithm slightly so that when using it to solve a system of difference constraints with m inequalities on n unknowns, the running time is $O(nm)$.

22.4-6
Consider adding ***equality constraints*** of the form $x_i = x_j + b_k$ to a system of difference constraints. Show how to solve this variety of constraint system.

22.4-7
Show how to solve a system of difference constraints by a Bellman-Ford-like algorithm that runs on a constraint graph without the extra vertex v_0.

★ ***22.4-8***
Let $Ax \le b$ be a system of m difference constraints in n unknowns. Show that the Bellman-Ford algorithm, when run on the corresponding constraint graph, maximizes $\sum_{i=1}^{n} x_i$ subject to $Ax \le b$ and $x_i \le 0$ for all x_i.

★ ***22.4-9***
Show that the Bellman-Ford algorithm, when run on the constraint graph for a system $Ax \le b$ of difference constraints, minimizes the quantity $(\max\{x_i\} - \min\{x_i\})$ subject to $Ax \le b$. Explain how this fact might come in handy if the algorithm is used to schedule construction jobs.

22.4-10
Suppose that every row in the matrix A of a linear program $Ax \le b$ corresponds to a difference constraint, a single-variable constraint of the form $x_i \le b_k$, or a single-variable constraint of the form $-x_i \le b_k$. Show how to adapt the Bellman-Ford algorithm to solve this variety of constraint system.

22.4-11
Give an efficient algorithm to solve a system $Ax \le b$ of difference constraints when all of the elements of b are real-valued and all of the unknowns x_i must be integers.

★ ***22.4-12***
Give an efficient algorithm to solve a system $Ax \le b$ of difference constraints when all of the elements of b are real-valued and a specified subset of some, but not necessarily all, of the unknowns x_i must be integers.

22.5 Proofs of shortest-paths properties

Throughout this chapter, our correctness arguments have relied on the triangle inequality, upper-bound property, no-path property, convergence property, path-relaxation property, and predecessor-subgraph property. We stated these properties without proof on page 611. In this section, we prove them.

The triangle inequality

In studying breadth-first search (Section 20.2), we proved as Lemma 20.1 a simple property of shortest distances in unweighted graphs. The triangle inequality generalizes the property to weighted graphs.

Lemma 22.10 (Triangle inequality)
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$ and source vertex s. Then, for all edges $(u, v) \in E$,

$$\delta(s, v) \le \delta(s, u) + w(u, v) .$$

Proof Suppose that p is a shortest path from source s to vertex v. Then p has no more weight than any other path from s to v. Specifically, path p has no more weight than the particular path that takes a shortest path from source s to vertex u and then takes edge (u, v).

Exercise 22.5-3 asks you to handle the case in which there is no shortest path from s to v. ■

Effects of relaxation on shortest-path estimates

The next group of lemmas describes how shortest-path estimates are affected by executing a sequence of relaxation steps on the edges of a weighted, directed graph that has been initialized by INITIALIZE-SINGLE-SOURCE.

Lemma 22.11 (Upper-bound property)
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$. Let $s \in V$ be the source vertex, and let the graph be initialized by INITIALIZE-SINGLE-SOURCE(G, s). Then, $v.d \ge \delta(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps on the edges of G. Moreover, once $v.d$ achieves its lower bound $\delta(s, v)$, it never changes.

Proof We prove the invariant $v.d \ge \delta(s, v)$ for all vertices $v \in V$ by induction over the number of relaxation steps.

For the base case, $v.d \geq \delta(s, v)$ holds after initialization, since if $v.d = \infty$, then $v.d \geq \delta(s, v)$ for all $v \in V - \{s\}$, and since $s.d = 0 \geq \delta(s, s)$. (Note that $\delta(s, s) = -\infty$ if s is on a negative-weight cycle and that $\delta(s, s) = 0$ otherwise.)

For the inductive step, consider the relaxation of an edge (u, v). By the inductive hypothesis, $x.d \geq \delta(s, x)$ for all $x \in V$ prior to the relaxation. The only d value that may change is $v.d$. If it changes, we have

$$\begin{aligned} v.d &= u.d + w(u, v) \\ &\geq \delta(s, u) + w(u, v) && \text{(by the inductive hypothesis)} \\ &\geq \delta(s, v) && \text{(by the triangle inequality)} , \end{aligned}$$

and so the invariant is maintained.

The value of $v.d$ never changes once $v.d = \delta(s, v)$ because, having achieved its lower bound, $v.d$ cannot decrease since we have just shown that $v.d \geq \delta(s, v)$, and it cannot increase because relaxation steps do not increase d values. ■

Corollary 22.12 (No-path property)
Suppose that in a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, no path connects a source vertex $s \in V$ to a given vertex $v \in V$. Then, after the graph is initialized by INITIALIZE-SINGLE-SOURCE(G, s), we have $v.d = \delta(s, v) = \infty$, and this equation is maintained as an invariant over any sequence of relaxation steps on the edges of G.

Proof By the upper-bound property, we always have $\infty = \delta(s, v) \leq v.d$, and thus $v.d = \infty = \delta(s, v)$. ■

Lemma 22.13
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, and let $(u, v) \in E$. Then, immediately after edge (u, v) is relaxed by a call of RELAX(u, v, w), we have $v.d \leq u.d + w(u, v)$.

Proof If, just prior to relaxing edge (u, v), we have $v.d > u.\ d + w(u, v)$, then $v.d = u.d + w(u, v)$ afterward. If, instead, $v.d \leq u.d + w(u, v)$ just before the relaxation, then neither $u.d$ nor $v.d$ changes, and so $v.d \leq u.d + w(u, v)$ afterward. ■

Lemma 22.14 (Convergence property)
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, let $s \in V$ be a source vertex, and let $s \leadsto u \to v$ be a shortest path in G for some vertices $u, v \in V$. Suppose that G is initialized by INITIALIZE-SINGLE-SOURCE(G, s) and then a sequence of relaxation steps that includes the call

RELAX(u, v, w) is executed on the edges of G. If $u.d = \delta(s, u)$ at any time prior to the call, then $v.d = \delta(s, v)$ at all times after the call.

Proof By the upper-bound property, if $u.d = \delta(s, u)$ at some point prior to relaxing edge (u, v), then this equation holds thereafter. In particular, after edge (u, v) is relaxed, we have

$$\begin{aligned} v.d &\le u.d + w(u, v) && \text{(by Lemma 22.13)} \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) && \text{(by Lemma 22.1 on page 606)} . \end{aligned}$$

The upper-bound property gives $v.d \ge \delta(s, v)$, from which we conclude that $v.d = \delta(s, v)$, and this equation is maintained thereafter. ■

Lemma 22.15 (Path-relaxation property)
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, and let $s \in V$ be a source vertex. Consider any shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$ from $s = v_0$ to v_k. If G is initialized by INITIALIZE-SINGLE-SOURCE(G, s) and then a sequence of relaxation steps occurs that includes, in order, relaxing the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$ after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur, including relaxations that are intermixed with relaxations of the edges of p.

Proof We show by induction that after the ith edge of path p is relaxed, we have $v_i.d = \delta(s, v_i)$. For the base case, $i = 0$, and before any edges of p have been relaxed, we have from the initialization that $v_0.d = s.d = 0 = \delta(s, s)$. By the upper-bound property, the value of $s.d$ never changes after initialization.

For the inductive step, assume that $v_{i-1}.d = \delta(s, v_{i-1})$. What happens when edge (v_{i-1}, v_i) is relaxed? By the convergence property, after this relaxation, we have $v_i.d = \delta(s, v_i)$, and this equation is maintained at all times thereafter. ■

Relaxation and shortest-paths trees

We now show that once a sequence of relaxations has caused the shortest-path estimates to converge to shortest-path weights, the predecessor subgraph G_π induced by the resulting π values is a shortest-paths tree for G. We start with the following lemma, which shows that the predecessor subgraph always forms a rooted tree whose root is the source.

Lemma 22.16
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, let $s \in V$ be a source vertex, and assume that G contains no negative-weight

cycles that are reachable from s. Then, after the graph is initialized by INITIALIZE-SINGLE-SOURCE(G, s), the predecessor subgraph G_π forms a rooted tree with root s, and any sequence of relaxation steps on edges of G maintains this property as an invariant.

Proof Initially, the only vertex in G_π is the source vertex, and the lemma is trivially true. Consider a predecessor subgraph G_π that arises after a sequence of relaxation steps. We first prove that G_π is acyclic. Suppose for the sake of contradiction that some relaxation step creates a cycle in the graph G_π. Let the cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_k = v_0$. Then, $v_i.\pi = v_{i-1}$ for $i = 1, 2, \ldots, k$ and, without loss of generality, assume that relaxing edge (v_{k-1}, v_k) created the cycle in G_π.

We claim that all vertices on cycle c are reachable from the source vertex s. Why? Each vertex on c has a non-NIL predecessor, and so each vertex on c was assigned a finite shortest-path estimate when it was assigned its non-NIL π value. By the upper-bound property, each vertex on cycle c has a finite shortest-path weight, which means that it is reachable from s.

We'll examine the shortest-path estimates on cycle c immediately before the call RELAX(v_{k-1}, v_k, w) and show that c is a negative-weight cycle, thereby contradicting the assumption that G contains no negative-weight cycles that are reachable from the source. Just before the call, we have $v_i.\pi = v_{i-1}$ for $i = 1, 2, \ldots, k-1$. Thus, for $i = 1, 2, \ldots, k-1$, the last update to $v_i.d$ was by the assignment $v_i.d = v_{i-1}.d + w(v_{i-1}, v_i)$. If $v_{i-1}.d$ changed since then, it decreased. Therefore, just before the call RELAX(v_{k-1}, v_k, w), we have

$$v_i.d \geq v_{i-1}.d + w(v_{i-1}, v_i) \qquad \text{for all } i = 1, 2, \ldots, k-1 . \tag{22.11}$$

Because $v_k.\pi$ is changed by the call RELAX(v_{k-1}, v_k, w), immediately beforehand we also have the strict inequality

$$v_k.d > v_{k-1}.d + w(v_{k-1}, v_k) .$$

Summing this strict inequality with the $k-1$ inequalities (22.11), we obtain the sum of the shortest-path estimates around cycle c:

$$\begin{aligned}\sum_{i=1}^{k} v_i.d &> \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) .\end{aligned}$$

But

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d ,$$

Figure 22.9 Showing that a simple path in G_π from source vertex s to vertex v is unique. If G_π contains two paths p_1 ($s \leadsto u \leadsto x \to z \leadsto v$) and p_2 ($s \leadsto u \leadsto y \to z \leadsto v$), where $x \neq y$, then $z.\pi = x$ and $z.\pi = y$, a contradiction.

since each vertex in the cycle c appears exactly once in each summation. This equation implies

$$0 > \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

Thus, the sum of weights around the cycle c is negative, which provides the desired contradiction.

We have now proven that G_π is a directed, acyclic graph. To show that it forms a rooted tree with root s, it suffices (see Exercise B.5-2 on page 1175) to prove that for each vertex $v \in V_\pi$, there is a unique simple path from s to v in G_π.

The vertices in V_π are those with non-NIL π values, plus s. Exercise 22.5-6 asks you to prove that a path from s exists to each vertex in V_π.

To complete the proof of the lemma, we now show that for any vertex $v \in V_\pi$, the graph G_π contains at most one simple path from s to v. Suppose otherwise. That is, suppose that, as Figure 22.9 illustrates, G_π contains two simple paths from s to some vertex v: p_1, which we decompose into $s \leadsto u \leadsto x \to z \leadsto v$, and p_2, which we decompose into $s \leadsto u \leadsto y \to z \leadsto v$, where $x \neq y$ (though u could be s and z could be v). But then, $z.\pi = x$ and $z.\pi = y$, which implies the contradiction that $x = y$. We conclude that G_π contains a unique simple path from s to v, and thus G_π forms a rooted tree with root s. ■

We can now show that if all vertices have been assigned their true shortest-path weights after a sequence of relaxation steps, then the predecessor subgraph G_π is a shortest-paths tree.

Lemma 22.17 (Predecessor-subgraph property)
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, let $s \in V$ be a source vertex, and assume that G contains no negative-weight cycles that are reachable from s. Then, after a call to INITIALIZE-SINGLE-SOURCE(G, s) followed by any sequence of relaxation steps on edges of G that produces $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph G_π is a shortest-paths tree rooted at s.

Proof We must prove that the three properties of shortest-paths trees given on page 608 hold for G_π. To show the first property, we must show that V_π is the set of vertices reachable from s. By definition, a shortest-path weight $\delta(s, v)$ is finite if and only if v is reachable from s, and thus the vertices that are reachable from s are exactly those with finite d values. But a vertex $v \in V - \{s\}$ has been assigned a finite value for $v.d$ if and only if $v.\pi \neq \text{NIL}$, since both assignments occur in RELAX. Thus, the vertices in V_π are exactly those reachable from s.

The second property, that G_π forms a rooted tree with root s, follows directly from Lemma 22.16.

It remains, therefore, to prove the last property of shortest-paths trees: for each vertex $v \in V_\pi$, the unique simple path $s \overset{p}{\leadsto} v$ in G_π is a shortest path from s to v in G. Let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$. Consider an edge (v_{i-1}, v_i) in path p. Because this edge belongs to G_π, the last relaxation that changed $v_i.d$ must have been of this edge. After that relaxation, we had $v_i.d = v_{i-1}.d + (v_{i-1}, v_i)$. Subsequently, an edge entering v_{i-1} could have been relaxed, causing $v_{i-1}.d$ to decrease further, but without changing $v_i.d$. Therefore, we have $v_i.d \geq v_{i-1}.d + w(v_{i-1}, v_i)$. Thus, for $i = 1, 2, \ldots, k$, we have both $v_i.d = \delta(s, v_i)$ and $v_i.d \geq v_{i-1}.d + w(v_{i-1}, v_i)$, which together imply $w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$. Summing the weights along path p yields

$$\begin{aligned}
w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\
&\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\
&= \delta(s, v_k) - \delta(s, v_0) && \text{(because the sum telescopes)} \\
&= \delta(s, v_k) && \text{(because } \delta(s, v_0) = \delta(s, s) = 0) \,.
\end{aligned}$$

Thus, we have $w(p) \leq \delta(s, v_k)$. Since $\delta(s, v_k)$ is a lower bound on the weight of any path from s to v_k, we conclude that $w(p) = \delta(s, v_k)$, and p is a shortest path from s to $v = v_k$. ■

Exercises

22.5-1

Give two shortest-paths trees for the directed graph of Figure 22.2 on page 609 other than the two shown.

22.5-2

Give an example of a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ and source vertex s such that G satisfies the following property: For

every edge $(u, v) \in E$, there is a shortest-paths tree rooted at s that contains (u, v) and another shortest-paths tree rooted at s that does not contain (u, v).

22.5-3
Modify the proof of Lemma 22.10 to handle cases in which shortest-path weights are ∞ or $-\infty$.

22.5-4
Let $G = (V, E)$ be a weighted, directed graph with source vertex s, and let G be initialized by INITIALIZE-SINGLE-SOURCE(G, s). Prove that if a sequence of relaxation steps sets $s.\pi$ to a non-NIL value, then G contains a negative-weight cycle.

22.5-5
Let $G = (V, E)$ be a weighted, directed graph with no negative-weight edges. Let $s \in V$ be the source vertex, and suppose that $v.\pi$ is allowed to be the predecessor of v on *any* shortest path to v from source s if $v \in V - \{s\}$ is reachable from s, and NIL otherwise. Give an example of such a graph G and an assignment of π values that produces a cycle in G_π. (By Lemma 22.16, such an assignment cannot be produced by a sequence of relaxation steps.)

22.5-6
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$ and no negative-weight cycles. Let $s \in V$ be the source vertex, and let G be initialized by INITIALIZE-SINGLE-SOURCE(G, s). Use induction to prove that for every vertex $v \in V_\pi$, there exists a path from s to v in G_π and that this property is maintained as an invariant over any sequence of relaxations.

22.5-7
Let $G = (V, E)$ be a weighted, directed graph that contains no negative-weight cycles. Let $s \in V$ be the source vertex, and let G be initialized by INITIALIZE-SINGLE-SOURCE(G, s). Prove that there exists a sequence of $|V| - 1$ relaxation steps that produces $v.d = \delta(s, v)$ for all $v \in V$.

22.5-8
Let G be an arbitrary weighted, directed graph with a negative-weight cycle reachable from the source vertex s. Show how to construct an infinite sequence of relaxations of the edges of G such that every relaxation causes a shortest-path estimate to change.

Problems

22-1 Yen's improvement to Bellman-Ford

The Bellman-Ford algorithm does not specify the order in which to relax edges in each pass. Consider the following method for deciding upon the order. Before the first pass, assign an arbitrary linear order $v_1, v_2, \ldots, v_{|V|}$ to the vertices of the input graph $G = (V, E)$. Then partition the edge set E into $E_f \cup E_b$, where $E_f = \{(v_i, v_j) \in E : i < j\}$ and $E_b = \{(v_i, v_j) \in E : i > j\}$. (Assume that G contains no self-loops, so that every edge belongs to either E_f or E_b.) Define $G_f = (V, E_f)$ and $G_b = (V, E_b)$.

a. Prove that G_f is acyclic with topological sort $\langle v_1, v_2, \ldots, v_{|V|}\rangle$ and that G_b is acyclic with topological sort $\langle v_{|V|}, v_{|V|-1}, \ldots, v_1\rangle$.

Suppose that each pass of the Bellman-Ford algorithm relaxes edges in the following way. First, visit each vertex in the order $v_1, v_2, \ldots, v_{|V|}$, relaxing edges of E_f that leave the vertex. Then visit each vertex in the order $v_{|V|}, v_{|V|-1}, \ldots, v_1$, relaxing edges of E_b that leave the vertex.

b. Prove that with this scheme, if G contains no negative-weight cycles that are reachable from the source vertex s, then after only $\lceil |V|/2 \rceil$ passes over the edges, $v.d = \delta(s, v)$ for all vertices $v \in V$.

c. Does this scheme improve the asymptotic running time of the Bellman-Ford algorithm?

22-2 Nesting boxes

A d-dimensional box with dimensions $(x_1, x_2, \ldots, x_d)$ ***nests*** within another box with dimensions $(y_1, y_2, \ldots, y_d)$ if there exists a permutation π on $\{1, 2, \ldots, d\}$ such that $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \ldots, x_{\pi(d)} < y_d$.

a. Argue that the nesting relation is transitive.

b. Describe an efficient method to determine whether one d-dimensional box nests inside another.

c. You are given a set of n d-dimensional boxes $\{B_1, B_2, \ldots, B_n\}$. Give an efficient algorithm to find the longest sequence $\langle B_{i_1}, B_{i_2}, \ldots, B_{i_k}\rangle$ of boxes such that B_{i_j} nests within $B_{i_{j+1}}$ for $j = 1, 2, \ldots, k-1$. Express the running time of your algorithm in terms of n and d.

22-3 Arbitrage

Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that one U.S. dollar buys 64 Indian rupees, one Indian rupee buys 1.8 Japanese yen, and one Japanese yen buys 0.009 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $64 \times 1.8 \times 0.009 = 1.0368$ U.S. dollars, thus turning a profit of 3.68%.

Suppose that you are given n currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that 1 unit of currency c_i buys $R[i, j]$ units of currency c_j.

a. Give an efficient algorithm to determine whether there exists a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1 .$$

Analyze the running time of your algorithm.

b. Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.

22-4 Gabow's scaling algorithm for single-source shortest paths

A ***scaling*** algorithm solves a problem by initially considering only the highest-order bit of each relevant input value, such as an edge weight, assuming that these values are nonnegative integers. The algorithm then refines the initial solution by looking at the two highest-order bits. It progressively looks at more and more high-order bits, refining the solution each time, until it has examined all bits and computed the correct solution.

This problem examines an algorithm for computing the shortest paths from a single source by scaling edge weights. The input is a directed graph $G = (V, E)$ with nonnegative integer edge weights w. Let $W = \max\{w(u, v) : (u, v) \in E\}$ be the maximum weight of any edge. In this problem, you will develop an algorithm that runs in $O(E \lg W)$ time. Assume that all vertices are reachable from the source.

The scaling algorithm uncovers the bits in the binary representation of the edge weights one at a time, from the most significant bit to the least significant bit. Specifically, let $k = \lceil \lg(W + 1) \rceil$ be the number of bits in the binary representation of W, and for $i = 1, 2, \ldots, k$, let $w_i(u, v) = \lfloor w(u, v)/2^{k-i} \rfloor$. That is, $w_i(u, v)$ is the "scaled-down" version of $w(u, v)$ given by the i most significant bits of $w(u, v)$. (Thus, $w_k(u, v) = w(u, v)$ for all $(u, v) \in E$.) For example, if $k = 5$ and $w(u, v) = 25$, which has the binary representation $\langle 11001 \rangle$, then $w_3(u, v) = \langle 110 \rangle = 6$. Also with $k = 5$, if $w(u, v) = \langle 00100 \rangle = 4$, then $w_4(u, v) = \langle 0010 \rangle = 2$. Define $\delta_i(u, v)$ as the shortest-path weight from vertex u

to vertex v using weight function w_i, so that $\delta_k(u, v) = \delta(u, v)$ for all $u, v \in V$. For a given source vertex s, the scaling algorithm first computes the shortest-path weights $\delta_1(s, v)$ for all $v \in V$, then computes $\delta_2(s, v)$ for all $v \in V$, and so on, until it computes $\delta_k(s, v)$ for all $v \in V$. Assume throughout that $|E| \geq |V| - 1$. You will show how to compute δ_i from δ_{i-1} in $O(E)$ time, so that the entire algorithm takes $O(kE) = O(E \lg W)$ time.

a. Suppose that for all vertices $v \in V$, we have $\delta(s, v) \leq |E|$. Show how to compute $\delta(s, v)$ for all $v \in V$ in $O(E)$ time.

b. Show how to compute $\delta_1(s, v)$ for all $v \in V$ in $O(E)$ time.

Now focus on computing δ_i from δ_{i-1}.

c. Prove that for $i = 2, 3, \ldots, k$, either $w_i(u, v) = 2w_{i-1}(u, v)$ or $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Then prove that

$$2\delta_{i-1}(s, v) \leq \delta_i(s, v) \leq 2\delta_{i-1}(s, v) + |V| - 1$$

for all $v \in V$.

d. Define, for $i = 2, 3, \ldots, k$ and all $(u, v) \in E$,

$$\widehat{w}_i(u, v) = w_i(u, v) + 2\delta_{i-1}(s, u) - 2\delta_{i-1}(s, v) .$$

Prove that for $i = 2, 3, \ldots, k$ and all $u, v \in V$, the "reweighted" value $\widehat{w}_i(u, v)$ of edge (u, v) is a nonnegative integer.

e. Now define $\widehat{\delta}_i(s, v)$ as the shortest-path weight from s to v using the weight function $\widehat{w}_i$. Prove that for $i = 2, 3, \ldots, k$ and all $v \in V$,

$$\delta_i(s, v) = \widehat{\delta}_i(s, v) + 2\delta_{i-1}(s, v)$$

and that $\widehat{\delta}_i(s, v) \leq |E|$.

f. Show how to compute $\delta_i(s, v)$ from $\delta_{i-1}(s, v)$ for all $v \in V$ in $O(E)$ time. Conclude that you can compute $\delta(s, v)$ for all $v \in V$ in $O(E \lg W)$ time.

22-5 Karp's minimum mean-weight cycle algorithm

Let $G = (V, E)$ be a directed graph with weight function $w : E \to \mathbb{R}$, and let $n = |V|$. We define the ***mean weight*** of a cycle $c = \langle e_1, e_2, \ldots, e_k \rangle$ of edges in E to be

$$\mu(c) = \frac{1}{k} \sum_{i=1}^{k} w(e_i) .$$

Let $\mu^* = \min\{\mu(c) : c \text{ is a directed cycle in } G\}$. We call a cycle c for which $\mu(c) = \mu^*$ a ***minimum mean-weight cycle***. This problem investigates an efficient algorithm for computing μ^*.

Assume without loss of generality that every vertex $v \in V$ is reachable from a source vertex $s \in V$. Let $\delta(s, v)$ be the weight of a shortest path from s to v, and let $\delta_k(s, v)$ be the weight of a shortest path from s to v consisting of *exactly* k edges. If there is no path from s to v with exactly k edges, then $\delta_k(s, v) = \infty$.

a. Show that if $\mu^* = 0$, then G contains no negative-weight cycles and $\delta(s, v) = \min\{\delta_k(s, v) : 0 \le k \le n - 1\}$ for all vertices $v \in V$.

b. Show that if $\mu^* = 0$, then

$$\max\left\{\frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} : 0 \le k \le n - 1\right\} \ge 0$$

for all vertices $v \in V$. (*Hint:* Use both properties from part (a).)

c. Let c be a 0-weight cycle, and let u and v be any two vertices on c. Suppose that $\mu^* = 0$ and that the weight of the simple path from u to v along the cycle is x. Prove that $\delta(s, v) = \delta(s, u) + x$. (*Hint:* The weight of the simple path from v to u along the cycle is $-x$.)

d. Show that if $\mu^* = 0$, then on each minimum mean-weight cycle there exists a vertex v such that

$$\max\left\{\frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} : 0 \le k \le n - 1\right\} = 0 .$$

(*Hint:* Show how to extend a shortest path to any vertex on a minimum mean-weight cycle along the cycle to make a shortest path to the next vertex on the cycle.)

e. Show that if $\mu^* = 0$, then the minimum value of

$$\max\left\{\frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} : 0 \le k \le n - 1\right\} ,$$

taken over all vertices $v \in V$, equals 0.

f. Show that if you add a constant t to the weight of each edge of G, then μ^* increases by t. Use this fact to show that μ^* equals the minimum value of

$$\max\left\{\frac{\delta_n(s,v)-\delta_k(s,v)}{n-k} : 0 \le k \le n-1\right\},$$

taken over all vertices $v \in V$.

g. Give an $O(VE)$-time algorithm to compute μ^*.

22-6 Bitonic shortest paths

A sequence is ***bitonic*** if it monotonically increases and then monotonically decreases, or if by a circular shift it monotonically increases and then monotonically decreases. For example the sequences $\langle 1, 4, 6, 8, 3, -2\rangle$, $\langle 9, 2, -4, -10, -5\rangle$, and $\langle 1, 2, 3, 4\rangle$ are bitonic, but $\langle 1, 3, 12, 4, 2, 10\rangle$ is not bitonic. (See Problem 14-3 on page 407 for the bitonic euclidean traveling-salesperson problem.)

Suppose that you are given a directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, where all edge weights are unique, and you wish to find single-source shortest paths from a source vertex s. You are given one additional piece of information: for each vertex $v \in V$, the weights of the edges along any shortest path from s to v form a bitonic sequence.

Give the most efficient algorithm you can to solve this problem, and analyze its running time.

Chapter notes

The shortest-path problem has a long history that is nicely desribed in an article by Schrijver [400]. He credits the general idea of repeatedly executing edge relaxations to Ford [148]. Dijkstra's algorithm [116] appeared in 1959, but it contained no mention of a priority queue. The Bellman-Ford algorithm is based on separate algorithms by Bellman [45] and Ford [149]. The same algorithm is also attributed to Moore [334]. Bellman describes the relation of shortest paths to difference constraints. Lawler [276] describes the linear-time algorithm for shortest paths in a dag, which he considers part of the folklore.

When edge weights are relatively small nonnegative integers, more efficient algorithms result from using min-priority queues that require integer keys and rely on the sequence of values returned by the EXTRACT-MIN calls in Dijkstra's algorithm monotonically increasing over time. Ahuja, Mehlhorn, Orlin, and Tarjan [8] give an algorithm that runs in $O(E + V\sqrt{\lg W})$ time on graphs with nonnegative edge weights, where W is the largest weight of any edge in the

graph. The best bounds are by Thorup [436], who gives an algorithm that runs in $O(E \lg \lg V)$ time, and by Raman [375], who gives an algorithm that runs in $O\left(E + V \min\left\{(\lg V)^{1/3+\epsilon}, (\lg W)^{1/4+\epsilon}\right\}\right)$ time. These two algorithms use an amount of space that depends on the word size of the underlying machine. Although the amount of space used can be unbounded in the size of the input, it can be reduced to be linear in the size of the input using randomized hashing.

For undirected graphs with integer weights, Thorup [435] gives an algorithm that runs in $O(V + E)$ time for single-source shortest paths. In contrast to the algorithms mentioned in the previous paragraph, the sequence of values returned by EXTRACT-MIN calls does not monotonically increase over time, and so this algorithm is not an implementation of Dijkstra's algorithm. Pettie and Ramachandran [357] remove the restriction of integer weights on undirected graphs. Their algorithm entails a preprocessing phase, followed by queries for specific source vertices. Preprocessing takes $O(MST(V, E) + \min\{V \lg V, V \lg \lg r\})$ time, where $MST(V, E)$ is the time to compute a minimum spanning tree and r is the ratio of the maximum edge weight to the minimum edge weight. After preprocessing, each query takes $O(E \lg \widehat{\alpha}(E, V))$ time, where $\widehat{\alpha}(E, V)$ is the inverse of Ackermann's function. (See the chapter notes for Chapter 19 for a brief discussion of Ackermann's function and its inverse.)

For graphs with negative edge weights, an algorithm due to Gabow and Tarjan [167] runs in $O(\sqrt{V} E \lg(VW))$ time, and one by Goldberg [186] runs in $O(\sqrt{V} E \lg W)$ time, where $W = \max\{|w(u, v)| : (u, v) \in E\}$. There has also been some progress based on methods that use continuous optimization and electrical flows. Cohen et al. [98] give such an algorithm, which is randomized and runs in $\widetilde{O}(E^{10/7} \lg W)$ expected time (see Problem 3-6 on page 73 for the defintion of $\widetilde{O}$-notation). There is also a pseudopolyomial-time algorithm based on fast matrix multiplication. Sankowski [394] and Yuster and Zwick [465] designed an algorithm for shortest paths that runs in $\widetilde{O}(WV^{\omega})$ time, where two $n \times n$ matrices can be multiplied in $O(n^{\omega})$ time, giving a faster algorithm than the previously mentioned algorithms for small values of W on dense graphs.

Cherkassky, Goldberg, and Radzik [89] conducted extensive experiments comparing various shortest-path algorithms. Shortest-path algorithms are widely used in real-time navigation and route-planning applications. Typically based on Dijkstra's algorithm, these algorithms use many clever ideas to be able to compute shortest paths on networks with many millions of vertices and edges in fractions of a second. Bast et al. [36] survey many of these developments.

23 All-Pairs Shortest Paths

In this chapter, we turn to the problem of finding shortest paths between all pairs of vertices in a graph. A classic application of this problem occurs in computing a table of distances between all pairs of cities for a road atlas. Classic perhaps, but not a true application of finding shortest paths between *all* pairs of vertices. After all, a road map modeled as a graph has one vertex for *every* road intersection and one edge wherever a road connects intersections. A table of intercity distances in an atlas might include distances for 100 cities, but the United States has approximately 300,000 signal-controlled intersections[1] and many more uncontrolled intersections.

A legitimate application of all-pairs shortest paths is to determine the ***diameter*** of a network: the longest of all shortest paths. If a directed graph models a communication network, with the weight of an edge indicating the time required for a message to traverse a communication link, then the diameter gives the longest possible transit time for a message in the network.

As in Chapter 22, the input is a weighted, directed graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$ that maps edges to real-valued weights. Now the goal is to find, for every pair of vertices $u, v \in V$, a shortest (least-weight) path from u to v, where the weight of a path is the sum of the weights of its constituent edges. For the all-pairs problem, the output typically takes a tabular form in which the entry in u's row and v's column is the weight of a shortest path from u to v.

You can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithm $|V|$ times, once with each vertex as the source. If all edge weights are nonnegative, you can use Dijkstra's algorithm. If you implement the min-priority queue with a linear array, the running time is $O(V^3 + VE)$ which is $O(V^3)$. The binary min-heap implementation of the min-priority queue

---

[1] According to a report cited by U.S. Department of Transportation Federal Highway Administration, "a reasonable 'rule of thumb' is one signalized intersection per 1,000 population."

yields a running time of $O(V(V + E) \lg V)$. If $|E| = \Omega(V)$, the running time becomes $O(VE \lg V)$, which is faster than $O(V^3)$ if the graph is sparse. Alternatively, you can implement the min-priority queue with a Fibonacci heap, yielding a running time of $O(V^2 \lg V + VE)$.

If the graph contains negative-weight edges, Dijkstra's algorithm doesn't work, but you can run the slower Bellman-Ford algorithm once from each vertex. The resulting running time is $O(V^2 E)$, which on a dense graph is $O(V^4)$. This chapter shows how to guarantee a much better asymptotic running time. It also investigates the relation of the all-pairs shortest-paths problem to matrix multiplication.

Unlike the single-source algorithms, which assume an adjacency-list representation of the graph, most of the algorithms in this chapter represent the graph by an adjacency matrix. (Johnson's algorithm for sparse graphs, in Section 23.3, uses adjacency lists.) For convenience, we assume that the vertices are numbered $1, 2, \ldots, |V|$, so that the input is an $n \times n$ matrix $W = (w_{ij})$ representing the edge weights of an n-vertex directed graph $G = (V, E)$, where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \ , \\ \text{the weight of directed edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E \ , \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \ . \end{cases} \tag{23.1}$$

The graph may contain negative-weight edges, but we assume for the time being that the input graph contains no negative-weight cycles.

The tabular output of each of the all-pairs shortest-paths algorithms presented in this chapter is an $n \times n$ matrix. The (i, j) entry of the output matrix contains $\delta(i, j)$, the shortest-path weight from vertex i to vertex j, as in Chapter 22.

A full solution to the all-pairs shortest-paths problem includes not only the shortest-path weights but also a ***predecessor matrix*** $\Pi = (\pi_{ij})$, where π_{ij} is NIL if either $i = j$ or there is no path from i to j, and otherwise π_{ij} is the predecessor of j on some shortest path from i. Just as the predecessor subgraph G_π from Chapter 22 is a shortest-paths tree for a given source vertex, the subgraph induced by the ith row of the Π matrix should be a shortest-paths tree with root i. For each vertex $i \in V$, the ***predecessor subgraph*** of G for i is $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, where

$$\begin{aligned} V_{\pi,i} &= \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\} \ , \\ E_{\pi,i} &= \{(\pi_{ij}, j) : j \in V_{\pi,i} - \{i\}\} \ . \end{aligned}$$

If $G_{\pi,i}$ is a shortest-paths tree, then PRINT-ALL-PAIRS-SHORTEST-PATH on the following page, which is a modified version of the PRINT-PATH procedure from Chapter 20, prints a shortest path from vertex i to vertex j.

In order to highlight the essential features of the all-pairs algorithms in this chapter, we won't cover how to compute predecessor matrices and their properties as extensively as we dealt with predecessor subgraphs in Chapter 22. Some of the exercises cover the basics.

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
if i == j
    print i
elseif π_ij == NIL
    print "no path from" i "to" j "exists"
else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
    print j
```

Chapter outline

Section 23.1 presents a dynamic-programming algorithm based on matrix multiplication to solve the all-pairs shortest-paths problem. The technique of "repeated squaring" yields a running time of $\Theta(V^3 \lg V)$. Section 23.2 gives another dynamic-programming algorithm, the Floyd-Warshall algorithm, which runs in $\Theta(V^3)$ time. Section 23.2 also covers the problem of finding the transitive closure of a directed graph, which is related to the all-pairs shortest-paths problem. Finally, Section 23.3 presents Johnson's algorithm, which solves the all-pairs shortest-paths problem in $O(V^2 \lg V + VE)$ time and is a good choice for large, sparse graphs.

Before proceeding, we need to establish some conventions for adjacency-matrix representations. First, we generally assume that the input graph $G = (V, E)$ has n vertices, so that $n = |V|$. Second, we use the convention of denoting matrices by uppercase letters, such as W, L, or D, and their individual elements by subscripted lowercase letters, such as w_{ij}, l_{ij}, or d_{ij}. Finally, some matrices have parenthesized superscripts, as in $L^{(r)} = \left(l_{ij}^{(r)}\right)$ or $D^{(r)} = \left(d_{ij}^{(r)}\right)$, to indicate iterates.

23.1 Shortest paths and matrix multiplication

This section presents a dynamic-programming algorithm for the all-pairs shortest-paths problem on a directed graph $G = (V, E)$. Each major loop of the dynamic program invokes an operation similar to matrix multiplication, so that the algorithm looks like repeated matrix multiplication. We'll start by developing a $\Theta(V^4)$-time algorithm for the all-pairs shortest-paths problem, and then we'll improve its running time to $\Theta(V^3 \lg V)$.

Before proceeding, let's briefly recap the steps given in Chapter 14 for developing a dynamic-programming algorithm:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.

We reserve the fourth step—constructing an optimal solution from computed information—for the exercises.

The structure of a shortest path

Let's start by characterizing the structure of an optimal solution. Lemma 22.1 tells us that all subpaths of a shortest path are shortest paths. Consider a shortest path p from vertex i to vertex j, and suppose that p contains at most r edges. Assuming that there are no negative-weight cycles, r is finite. If $i = j$, then p has weight 0 and no edges. If vertices i and j are distinct, then decompose path p into $i \overset{p'}{\leadsto} k \to j$, where path p' now contains at most $r - 1$ edges. Lemma 22.1 says that p' is a shortest path from i to k, and so $\delta(i, j) = \delta(i, k) + w_{kj}$.

A recursive solution to the all-pairs shortest-paths problem

Now, let $l_{ij}^{(r)}$ be the minimum weight of any path from vertex i to vertex j that contains at most r edges. When $r = 0$, there is a shortest path from i to j with no edges if and only if $i = j$, yielding

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \ , \\ \infty & \text{if } i \neq j \ . \end{cases} \tag{23.2}$$

For $r \geq 1$, one way to achieve a minimum-weight path from i to j with at most r edges is by taking a path containing at most $r - 1$ edges, so that $l_{ij}^{(r)} = l_{ij}^{(r-1)}$. Another way is by taking a path of at most $r - 1$ edges from i to some vertex k and then taking the edge (k, j), so that $l_{ij}^{(r)} = l_{ik}^{(r-1)} + w(k, j)$. Therefore, to examine paths from i to j consisting of at most r edges, try all possible predecessors k of j, giving the recursive definition

$$\begin{aligned} l_{ij}^{(r)} &= \min\left\{l_{ij}^{(r-1)}, \ \min\left\{l_{ik}^{(r-1)} + w_{kj} : 1 \leq k \leq n\right\}\right\} \\ &= \min\left\{l_{ik}^{(r-1)} + w_{kj} : 1 \leq k \leq n\right\} \ . \end{aligned} \tag{23.3}$$

The last equality follows from the observation that $w_{jj} = 0$ for all j.

What are the actual shortest-path weights $\delta(i, j)$? If the graph contains no negative-weight cycles, then whenever $\delta(i, j) < \infty$, there is a shortest path from vertex i to vertex j that is simple. (A path p from i to j that is not simple contains a cycle. Since each cycle's weight is nonnegative, removing all cycles from the path leaves a simple path with weight no greater than p's weight.) Because any simple path contains at most $n - 1$ edges, a path from vertex i to vertex j with more than $n - 1$ edges cannot have lower weight than a shortest path from i to j. The actual shortest-path weights are therefore given by

$$\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots . \tag{23.4}$$

Computing the shortest-path weights bottom up

Taking as input the matrix $W = (w_{ij})$, let's see how to compute a series of matrices $L^{(0)}, L^{(1)}, \ldots, L^{(n-1)}$, where $L^{(r)} = \left(l_{ij}^{(r)}\right)$ for $r = 0, 1, \ldots, n-1$. The initial matrix is $L^{(0)}$ given by equation (23.2). The final matrix $L^{(n-1)}$ contains the actual shortest-path weights.

The heart of the algorithm is the procedure EXTEND-SHORTEST-PATHS, which implements equation (23.3) for all i and j. The four inputs are the matrix $L^{(r-1)}$ computed so far; the edge-weight matrix W; the output matrix $L^{(r)}$, which will hold the computed result and whose elements are all initialized to ∞ before invoking the procedure; and the number n of vertices. The superscripts r and $r-1$ help to make the correspondence of the pseudocode with equation (23.3) plain, but they play no actual role in the pseudocode. The procedure extends the shortest paths computed so far by one more edge, producing the matrix $L^{(r)}$ of shortest-path weights from the matrix $L^{(r-1)}$ computed so far. Its running time is $\Theta(n^3)$ due to the three nested **for** loops.

```
EXTEND-SHORTEST-PATHS(L^(r-1), W, L^(r), n)
// Assume that the elements of L^(r) are initialized to ∞.
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            l_ij^(r) = min{l_ij^(r), l_ik^(r-1) + w_kj}
```

Let's now understand the relation of this computation to matrix multiplication. Consider how to compute the matrix product $C = A \cdot B$ of two $n \times n$ matrices A and B. The straightforward method used by MATRIX-MULTIPLY on page 81 uses a triply nested loop to implement equation (4.1), which we repeat here for convenience:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} , \tag{23.5}$$

for $i, j = 1, 2, \ldots, n$. Now make the substitutions

$$
\begin{aligned}
l^{(r-1)} &\to a\ , \\
w &\to b\ , \\
l^{(r)} &\to c\ , \\
\min &\to +\ , \\
+ &\to \cdot
\end{aligned}
$$

in equation (23.3). You get equation (23.5)! Making these changes to EXTEND-SHORTEST-PATHS, and also replacing ∞ (the identity for min) by 0 (the identity for $+$), yields the procedure MATRIX-MULTIPLY. We can see that the procedure EXTEND-SHORTEST-PATHS$(L^{(r-1)}, W, L^{(r)}, n)$ computes the matrix "product" $L^{(r)} = L^{(r-1)} \cdot W$ using this unusual definition of matrix multiplication.[2]

Thus, we can solve the all-pairs shortest-paths problem by repeatedly multiplying matrices. Each step extends the shortest-path weights computed so far by one more edge using EXTEND-SHORTEST-PATHS$(L^{(r-1)}, W, L^{(r)}, n)$ to perform the matrix multiplication. Starting with the matrix $L^{(0)}$, we produce the following sequence of $n-1$ matrices corresponding to powers of W:

$$
\begin{array}{rclcl}
L^{(1)} & = & L^{(0)} \cdot W & = & W^1\ , \\
L^{(2)} & = & L^{(1)} \cdot W & = & W^2\ , \\
L^{(3)} & = & L^{(2)} \cdot W & = & W^3\ , \\
 & & \vdots & & \\
L^{(n-1)} & = & L^{(n-2)} \cdot W & = & W^{n-1}\ .
\end{array}
$$

At the end, the matrix $L^{(n-1)} = W^{n-1}$ contains the shortest-path weights.

The procedure SLOW-APSP on the next page computes this sequence in $\Theta(n^4)$ time. The procedure takes the $n \times n$ matrices W and $L^{(0)}$ as inputs, along with n. Figure 23.1 illustrates its operation. The pseudocode uses two $n \times n$ matrices L and M to store powers of W, computing $M = L \cdot W$ on each iteration. Line 2 initializes $L = L^{(0)}$. For each iteration r, line 4 initializes $M = \infty$, where ∞ in this context is a matrix of scalar ∞ values. The rth iteration starts with the invariant $L = L^{(r-1)} = W^{r-1}$. Line 6 computes $M = L \cdot W = L^{(r-1)} \cdot W = W^{r-1} \cdot W = W^r = L^{(r)}$ so that the invariant can be restored for the next iteration by line 7, which sets $L = M$. At the end, the matrix $L = L^{(n-1)} = W^{n-1}$ of shortest-path weights is returned. The assignments to $n \times n$ matrices in lines 2, 4, and 7 implicitly run doubly nested loops that take $\Theta(n^2)$ time for each assignment.

---

[2] An algebraic ***semiring*** contains operations $\oplus$, which is commutative with identity $I_\oplus$, and $\otimes$, with identity $I_\otimes$, where $\otimes$ distributes over $\oplus$ on both the left and right, and where $I_\oplus \otimes x = x \otimes I_\oplus = I_\oplus$ for all x. Standard matrix multiplication, as in MATRIX-MULTIPLY, uses the semiring with $+$ for $\oplus$, $\cdot$ for $\otimes$, 0 for $I_\oplus$, and 1 for $I_\otimes$. The procedure EXTEND-SHORTEST-PATHS uses another semiring, known as the ***tropical semiring***, with min for $\oplus$, $+$ for $\otimes$, ∞ for $I_\oplus$, and 0 for $I_\otimes$.

$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Figure 23.1 A directed graph and the sequence of matrices $L^{(r)}$ computed by SLOW-APSP. You might want to verify that $L^{(5)}$, defined as $L^{(4)} \cdot W$, equals $L^{(4)}$, and thus $L^{(r)} = L^{(4)}$ for all $r \geq 4$.

The $n - 1$ invocations of EXTEND-SHORTEST-PATHS, each of which takes $\Theta(n^3)$ time, dominate the computation, yielding a total running time of $\Theta(n^4)$.

```
SLOW-APSP(W, L^(0), n)
let L = (l_ij) and M = (m_ij) be new n × n matrices
L = L^(0)
for r = 1 to n − 1
    M = ∞        // initialize M
    // Compute the matrix "product" M = L · W.
    EXTEND-SHORTEST-PATHS(L, W, M, n)
    L = M
return L
```

Improving the running time

Bear in mind that the goal is not to compute *all* the $L^{(r)}$ matrices: only the matrix $L^{(n-1)}$ matters. Recall that in the absence of negative-weight cycles, equation (23.4) implies $L^{(r)} = L^{(n-1)}$ for all integers $r \geq n - 1$. Just as traditional matrix multiplication is associative, so is matrix multiplication defined by the EXTEND-SHORTEST-PATHS procedure (see Exercise 23.1-4). In fact, we can compute $L^{(n-1)}$ with only $\lceil \lg(n-1) \rceil$ matrix products by using the technique of ***repeated squaring***:

$$
\begin{aligned}
L^{(1)} &= W , \\
L^{(2)} &= W^2 &&= W \cdot W , \\
L^{(4)} &= W^4 &&= W^2 \cdot W^2 \\
L^{(8)} &= W^8 &&= W^4 \cdot W^4 , \\
&\vdots \\
L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} &&= W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}} .
\end{aligned}
$$

Since $2^{\lceil \lg(n-1) \rceil} \geq n - 1$, the final product is $L^{(2^{\lceil \lg(n-1) \rceil})} = L^{(n-1)}$.

The procedure FASTER-APSP implements this idea. It takes just the $n \times n$ matrix W and the size n as inputs. Each iteration of the **while** loop of lines 4–8 starts with the invariant $L = W^r$, which it squares using EXTEND-SHORTEST-PATHS to obtain the matrix $M = L^2 = (W^r)^2 = W^{2r}$. At the end of each iteration, the value of r doubles, and L for the next iteration becomes M, restoring the invariant. Upon exiting the loop when $r \geq n - 1$, the procedure returns $L = W^r = L^{(r)} = L^{(n-1)}$ by equation (23.4). As in SLOW-APSP, the assignments to $n \times n$ matrices in lines 2, 5, and 8 implicitly run doubly nested loops, taking $\Theta(n^2)$ time for each assignment.

```
FASTER-APSP(W, n)
1  let L and M be new n × n matrices
2  L = W
3  r = 1
4  while r < n − 1
5      M = ∞          // initialize M
6      EXTEND-SHORTEST-PATHS(L, L, M, n)      // compute M = L^2
7      r = 2r
8      L = M          // ready for the next iteration
9  return L
```

Because each of the $\lceil \lg(n-1) \rceil$ matrix products takes $\Theta(n^3)$ time, FASTER-APSP runs in $\Theta(n^3 \lg n)$ time. The code is tight, containing no elaborate data structures, and the constant hidden in the Θ-notation is therefore small.

Exercises

23.1-1

Run SLOW-APSP on the weighted, directed graph of Figure 23.2, showing the matrices that result for each iteration of the loop. Then do the same for FASTER-APSP.

Figure 23.2 A weighted, directed graph for use in Exercises 23.1-1, 23.2-1, and 23.3-1.

23.1-2
Why is it convenient for both SLOW-APSP and FASTER-APSP that $w_{ii} = 0$ for $i = 1, 2, \ldots, n$?

23.1-3
What does the matrix

$$L^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$

used in the shortest-paths algorithms correspond to in regular matrix multiplication?

23.1-4
Show that matrix multiplication defined by EXTEND-SHORTEST-PATHS is associative.

23.1-5
Show how to express the single-source shortest-paths problem as a product of matrices and a vector. Describe how evaluating this product corresponds to a Bellman-Ford-like algorithm (see Section 22.1).

23.1-6
Argue that we don't need the matrix M in SLOW-APSP because by substituting L for M and leaving out the initialization of M, the code still works correctly. (*Hint:* Relate line 5 of EXTEND-SHORTEST-PATHS to RELAX on page 610.) Do we need the matrix M in FASTER-APSP?

23.1-7
Suppose that you also want to compute the vertices on shortest paths in the algorithms of this section. Show how to compute the predecessor matrix Π from the completed matrix L of shortest-path weights in $O(n^3)$ time.

23.1-8
You can also compute the vertices on shortest paths along with computing the shortest-path weights. Define $\pi_{ij}^{(r)}$ as the predecessor of vertex j on any minimum-weight path from vertex i to vertex j that contains at most r edges. Modify the EXTEND-SHORTEST-PATHS and SLOW-APSP procedures to compute the matrices $\Pi^{(1)}, \Pi^{(2)}, \ldots, \Pi^{(n-1)}$ as they compute the matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$.

23.1-9
Modify FASTER-APSP so that it can determine whether the graph contains a negative-weight cycle.

23.1-10
Give an efficient algorithm to find the length (number of edges) of a minimum-length negative-weight cycle in a graph.

23.2 The Floyd-Warshall algorithm

Having already seen one dynamic-programming solution to the all-pairs shortest-paths problem, in this section we'll see another: the ***Floyd-Warshall algorithm***, which runs in $\Theta(V^3)$ time. As before, negative-weight edges may be present, but not negative-weight cycles. As in Section 23.1, we develop the algorithm by following the dynamic-programming process. After studying the resulting algorithm, we present a similar method for finding the transitive closure of a directed graph.

The structure of a shortest path

In the Floyd-Warshall algorithm, we characterize the structure of a shortest path differently from how we characterized it in Section 23.1. The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path, where an ***intermediate*** vertex of a simple path $p = \langle v_1, v_2, \ldots, v_l \rangle$ is any vertex of p other than v_1 or v_l, that is, any vertex in the set $\{v_2, v_3, \ldots, v_{l-1}\}$.

The Floyd-Warshall algorithm relies on the following observation. Numbering the vertices of G by $V = \{1, 2, \ldots, n\}$, take a subset $\{1, 2, \ldots, k\}$ of vertices for some $1 \le k \le n$. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$, and let p be a

Figure 23.3 Optimal substructure used by the Floyd-Warshall algorithm. Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

minimum-weight path from among them. (Path p is simple.) The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The details of the relationship depend on whether k is an intermediate vertex of path p or not.

- If k is not an intermediate vertex of path p, then all intermediate vertices of path p belong to the set $\{1, 2, \ldots, k-1\}$. Thus a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.
- If k is an intermediate vertex of path p, then decompose p into $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$, as Figure 23.3 illustrates. By Lemma 22.1, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k\}$. In fact, we can make a slightly stronger statement. Because vertex k is not an *intermediate* vertex of path p_1, all intermediate vertices of p_1 belong to the set $\{1, 2, \ldots, k-1\}$. Therefore p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Likewise, p_2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$.

A recursive solution to the all-pairs shortest-paths problem

The above observations suggest a recursive formulation of shortest-path estimates that differs from the one in Section 23.1. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices belong to the set $\{1, 2, \ldots, k\}$. When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$. Following the above discussion, define $d_{ij}^{(k)}$ recursively by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 , \\ \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\} & \text{if } k \geq 1 . \end{cases} \tag{23.6}$$

Because for any path, all intermediate vertices belong to the set $\{1, 2, \ldots, n\}$, the matrix $D^{(n)} = \left(d_{ij}^{(n)}\right)$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)$ for all $i, j \in V$.

Computing the shortest-path weights bottom up

Based on recurrence (23.6), the bottom-up procedure FLOYD-WARSHALL computes the values $d_{ij}^{(k)}$ in order of increasing values of k. Its input is an $n \times n$ matrix W defined as in equation (23.1). The procedure returns the matrix $D^{(n)}$ of shortest-path weights. Figure 23.4 shows the matrices $D^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 23.1.

```
FLOYD-WARSHALL(W, n)
1  D^(0) = W
2  for k = 1 to n
3      let D^(k) = (d_ij^(k)) be a new n × n matrix
4      for i = 1 to n
5          for j = 1 to n
6              d_ij^(k) = min{d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1)}
7  return D^(n)
```

The running time of the Floyd-Warshall algorithm is determined by the triply nested **for** loops of lines 2–6. Because each execution of line 6 takes $O(1)$ time, the algorithm runs in $\Theta(n^3)$ time. As in the final algorithm in Section 23.1, the code is tight, with no elaborate data structures, and so the constant hidden in the Θ-notation is small. Thus, the Floyd-Warshall algorithm is quite practical for even moderate-sized input graphs.

Constructing a shortest path

There are a variety of different methods for constructing shortest paths in the Floyd-Warshall algorithm. One way is to compute the matrix D of shortest-path weights and then construct the predecessor matrix Π from the D matrix. Exercise 23.1-7 asks you to implement this method so that it runs in $O(n^3)$ time. Given the predecessor matrix Π, the PRINT-ALL-PAIRS-SHORTEST-PATH procedure prints the vertices on a given shortest path.

Alternatively, the predecessor matrix Π can be computed while the algorithm computes the matrices $D^{(0)}, D^{(1)}, \ldots, D^{(n)}$. Specifically, compute a sequence of

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figure 23.4 The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 23.1.

matrices $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, where $\Pi = \Pi^{(n)}$ and $\pi_{ij}^{(k)}$ is the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

Here's a recursive formulation of $\pi_{ij}^{(k)}$. When $k = 0$, a shortest path from i to j has no intermediate vertices at all, and so

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty , \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty . \end{cases} \tag{23.7}$$

For $k \geq 1$, if the path has k as an intermediate vertex, so that it is $i \leadsto k \leadsto j$ where $k \neq j$, then choose as the predecessor of j on this path the same vertex as the predecessor of j chosen on a shortest path from k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Otherwise, when the path from i to j does not have k as an intermediate vertex, choose the same predecessor of j as on a shortest path from i with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Formally, for $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \ (k \text{ is an intermediate vertex}) , \\ \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \ (k \text{ is not an intermediate vertex}) . \end{cases} \tag{23.8}$$

Exercise 23.2-3 asks you to show how to incorporate the $\Pi^{(k)}$ matrix computations into the FLOYD-WARSHALL procedure. Figure 23.4 shows the sequence of $\Pi^{(k)}$ matrices that the resulting algorithm computes for the graph of Figure 23.1. The exercise also asks for the more difficult task of proving that the predecessor subgraph $G_{\pi,i}$ is a shortest-paths tree with root i. Exercise 23.2-7 asks for yet another way to reconstruct shortest paths.

Transitive closure of a directed graph

Given a directed graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$, you might wish to determine simply whether G contains a path from i to j for all vertex pairs $i, j \in V$, without regard to edge weights. We define the ***transitive closure*** of G as the graph $G^* = (V, E^*)$, where

$$E^* = \{(i, j) : \text{there is a path from vertex } i \text{ to vertex } j \text{ in } G\} .$$

One way to compute the transitive closure of a graph in $\Theta(n^3)$ time is to assign a weight of 1 to each edge of E and run the Floyd-Warshall algorithm. If there is a path from vertex i to vertex j, you get $d_{ij} < n$. Otherwise, you get $d_{ij} = \infty$.

There is another, similar way to compute the transitive closure of G in $\Theta(n^3)$ time, which can save time and space in practice. This method substitutes the logical operations $\vee$ (logical OR) and $\wedge$ (logical AND) for the arithmetic operations min and $+$ in the Floyd-Warshall algorithm. For $i, j, k = 1, 2, \ldots, n$, define $t_{ij}^{(k)}$ to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$, and 0 otherwise. To construct the transitive closure $G^* = (V, E^*)$, put edge (i, j) into E^* if and only if $t_{ij}^{(n)} = 1$. A recursive definition of $t_{ij}^{(k)}$, analogous to recurrence (23.6), is

$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Figure 23.5 A directed graph and the matrices $T^{(k)}$ computed by the transitive-closure algorithm.

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \ , \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \ , \end{cases}$$

and for $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right) \ . \tag{23.9}$$

As in the Floyd-Warshall algorithm, the TRANSITIVE-CLOSURE procedure computes the matrices $T^{(k)} = \left(t_{ij}^{(k)}\right)$ in order of increasing k.

```
TRANSITIVE-CLOSURE(G, n)
let T^(0) = (t_ij^(0)) be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        if i == j or (i, j) ∈ G.E
            t_ij^(0) = 1
        else t_ij^(0) = 0
for k = 1 to n
    let T^(k) = (t_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
return T^(n)
```

Figure 23.5 shows the matrices $T^{(k)}$ computed by the TRANSITIVE-CLOSURE procedure on a sample graph. The TRANSITIVE-CLOSURE procedure, like the Floyd-Warshall algorithm, runs in $\Theta(n^3)$ time. On some computers, though, logical operations on single-bit values execute faster than arithmetic operations on integer words of data. Moreover, because the direct transitive-closure algorithm

uses only boolean values rather than integer values, its space requirement is less than the Floyd-Warshall algorithm's by a factor corresponding to the size of a word of computer storage.

Exercises

23.2-1
Run the Floyd-Warshall algorithm on the weighted, directed graph of Figure 23.2. Show the matrix $D^{(k)}$ that results for each iteration of the outer loop.

23.2-2
Show how to compute the transitive closure using the technique of Section 23.1.

23.2-3
Modify the FLOYD-WARSHALL procedure to compute the $\Pi^{(k)}$ matrices according to equations (23.7) and (23.8). Prove rigorously that for all $i \in V$, the predecessor subgraph $G_{\pi,i}$ is a shortest-paths tree with root i. (*Hint:* To show that $G_{\pi,i}$ is acyclic, first show that $\pi_{ij}^{(k)} = l$ implies $d_{ij}^{(k)} \geq d_{il}^{(k)} + w_{lj}$, according to the definition of $\pi_{ij}^{(k)}$. Then adapt the proof of Lemma 22.16.)

23.2-4
As it appears on page 657, the Floyd-Warshall algorithm requires $\Theta(n^3)$ space, since it creates $d_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$. Show that the procedure FLOYD-WARSHALL′, which simply drops all the superscripts, is correct, and thus only $\Theta(n^2)$ space is required.

```
FLOYD-WARSHALL′(W, n)
D = W
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            d_ij = min{d_ij, d_ik + d_kj}
return D
```

23.2-5
Consider the following change to how equation (23.8) handles equality:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \ (k \text{ is an intermediate vertex}) , \\ \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \ (k \text{ is not an intermediate vertex}) . \end{cases}$$

Is this alternative definition of the predecessor matrix Π correct?

23.2-6
Show how to use the output of the Floyd-Warshall algorithm to detect the presence of a negative-weight cycle.

23.2-7
Another way to reconstruct shortest paths in the Floyd-Warshall algorithm uses values $\phi_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$, where $\phi_{ij}^{(k)}$ is the highest-numbered intermediate vertex of a shortest path from i to j in which all intermediate vertices lie in the set $\{1, 2, \ldots, k\}$. Give a recursive formulation for $\phi_{ij}^{(k)}$, modify the FLOYD-WARSHALL procedure to compute the $\phi_{ij}^{(k)}$ values, and rewrite the PRINT-ALL-PAIRS-SHORTEST-PATH procedure to take the matrix $\Phi = \left(\phi_{ij}^{(n)}\right)$ as an input. How is the matrix Φ like the s table in the matrix-chain multiplication problem of Section 14.2?

23.2-8
Give an $O(VE)$-time algorithm for computing the transitive closure of a directed graph $G = (V, E)$. Assume that $|V| = O(E)$ and that the graph is represented with adjacency lists.

23.2-9
Suppose that it takes $f(|V|, |E|)$ time to compute the transitive closure of a directed acyclic graph, where f is a monotonically increasing function of both $|V|$ and $|E|$. Show that the time to compute the transitive closure $G^* = (V, E^*)$ of a general directed graph $G = (V, E)$ is then $f(|V|, |E|) + O(V + E^*)$.

23.3 Johnson's algorithm for sparse graphs

Johnson's algorithm finds shortest paths between all pairs in $O(V^2 \lg V + VE)$ time. For sparse graphs, it is asymptotically faster than either repeated squaring of matrices or the Floyd-Warshall algorithm. The algorithm either returns a matrix of shortest-path weights for all pairs of vertices or reports that the input graph contains a negative-weight cycle. Johnson's algorithm uses as subroutines both Dijkstra's algorithm and the Bellman-Ford algorithm, which Chapter 22 describes.

Johnson's algorithm uses the technique of ***reweighting***, which works as follows. If all edge weights w in a graph $G = (V, E)$ are nonnegative, Dijkstra's algorithm can find shortest paths between all pairs of vertices by running it once from each vertex. With the Fibonacci-heap min-priority queue, the running time of this all-pairs algorithm is $O(V^2 \lg V + VE)$. If G has negative-weight edges but no negative-weight cycles, first compute a new set of nonnegative edge weights so

that Dijkstra's algorithm applies. The new set of edge weights $\widehat{w}$ must satisfy two important properties:

1. For all pairs of vertices $u, v \in V$, a path p is a shortest path from u to v using weight function w if and only if p is also a shortest path from u to v using weight function $\widehat{w}$.
2. For all edges (u, v), the new weight $\widehat{w}(u, v)$ is nonnegative.

As we'll see in a moment, preprocessing G to determine the new weight function $\widehat{w}$ takes $O(VE)$ time.

Preserving shortest paths by reweighting

The following lemma shows how to reweight the edges to satisfy the first property above. We use δ to denote shortest-path weights derived from weight function w and $\widehat{\delta}$ to denote shortest-path weights derived from weight function $\widehat{w}$.

Lemma 23.1 (Reweighting does not change shortest paths)
Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $h : V \to \mathbb{R}$ be any function mapping vertices to real numbers. For each edge $(u, v) \in E$, define

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v) . \tag{23.10}$$

Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be any path from vertex v_0 to vertex v_k. Then p is a shortest path from v_0 to v_k with weight function w if and only if it is a shortest path with weight function $\widehat{w}$. That is, $w(p) = \delta(v_0, v_k)$ if and only if $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$. Furthermore, G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function $\widehat{w}$.

Proof We start by showing that

$$\widehat{w}(p) = w(p) + h(v_0) - h(v_k) . \tag{23.11}$$

We have

$$\begin{aligned}
\widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \qquad \text{(because the sum telescopes)} \\
&= w(p) + h(v_0) - h(v_k) .
\end{aligned}$$

Therefore, any path p from v_0 to v_k has $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$. Because $h(v_0)$ and $h(v_k)$ do not depend on the path, if one path from v_0 to v_k is shorter than another using weight function w, then it is also shorter using $\widehat{w}$. Thus, $w(p) = \delta(v_0, v_k)$ if and only if $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$.

Finally, we show that G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function $\widehat{w}$. Consider any cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$. By equation (23.11),

$$\begin{aligned} \widehat{w}(c) &= w(c) + h(v_0) - h(v_k) \\ &= w(c) , \end{aligned}$$

and thus c has negative weight using w if and only if it has negative weight using $\widehat{w}$. ■

Producing nonnegative weights by reweighting

Our next goal is to ensure that the second property holds: $\widehat{w}(u, v)$ must be nonnegative for all edges $(u, v) \in E$. Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, we'll see how to make a new graph $G' = (V', E')$, where $V' = V \cup \{s\}$ for some new vertex $s \notin V$ and $E' = E \cup \{(s, v) : v \in V\}$. To incorporate the new vertex s, extend the weight function w so that $w(s, v) = 0$ for all $v \in V$. Since no edges enter s, no shortest paths in G', other than those with source s, contain s. Moreover, G' has no negative-weight cycles if and only if G has no negative-weight cycles. Figure 23.6(a) shows the graph G' corresponding to the graph G of Figure 23.1.

Now suppose that G and G' have no negative-weight cycles. Define the function $h(v) = \delta(s, v)$ for all $v \in V'$. By the triangle inequality (Lemma 22.10 on page 633), we have $h(v) \le h(u) + w(u, v)$ for all edges $(u, v) \in E'$. Thus, by defining reweighted edge weights $\widehat{w}$ according to equation (23.10), we have $\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \ge 0$, thereby satisfying the second property. Figure 23.6(b) shows the graph G' from Figure 23.6(a) with reweighted edges.

Computing all-pairs shortest paths

Johnson's algorithm to compute all-pairs shortest paths uses the Bellman-Ford algorithm (Section 22.1) and Dijkstra's algorithm (Section 22.3) as subroutines. The pseudocode appears in the procedure JOHNSON on page 666. It assumes implicitly that the edges are stored in adjacency lists. The algorithm returns the usual $|V| \times |V|$ matrix $D = (d_{ij})$, where $d_{ij} = \delta(i, j)$, or it reports that the input graph contains a negative-weight cycle. As is typical for an all-pairs shortest-paths algorithm, it assumes that the vertices are numbered from 1 to $|V|$.

Figure 23.6 Johnson's all-pairs shortest-paths algorithm run on the graph of Figure 23.1. Vertex numbers appear outside the vertices. **(a)** The graph G' with the original weight function w. The new vertex s is blue. Within each vertex v is $h(v) = \delta(s, v)$. **(b)** After reweighting each edge (u, v) with weight function $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$. **(c)–(g)** The result of running Dijkstra's algorithm on each vertex of G using weight function $\widehat{w}$. In each part, the source vertex u is blue, and blue edges belong to the shortest-paths tree computed by the algorithm. Within each vertex v are the values $\widehat{\delta}(u, v)$ and $\delta(u, v)$, separated by a slash. The value $d_{uv} = \delta(u, v)$ is equal to $\widehat{\delta}(u, v) + h(v) - h(u)$.

```
JOHNSON(G, w)
1  compute G′, where G′.V = G.V ∪ {s},
       G′.E = G.E ∪ {(s, v) : v ∈ G.V}, and
       w(s, v) = 0 for all v ∈ G.V
2  if BELLMAN-FORD(G′, w, s) == FALSE
3      print "the input graph contains a negative-weight cycle"
4  else for each vertex v ∈ G′.V
5          set h(v) to the value of δ(s, v)
               computed by the Bellman-Ford algorithm
6      for each edge (u, v) ∈ G′.E
7          ŵ(u, v) = w(u, v) + h(u) − h(v)
8      let D = (d_uv) be a new n × n matrix
9      for each vertex u ∈ G.V
10         run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ G.V
11         for each vertex v ∈ G.V
12             d_uv = δ̂(u, v) + h(v) − h(u)
13     return D
```

The JOHNSON procedure simply performs the actions specified earlier. Line 1 produces G'. Line 2 runs the Bellman-Ford algorithm on G' with weight function w and source vertex s. If G', and hence G, contains a negative-weight cycle, line 3 reports the problem. Lines 4–12 assume that G' contains no negative-weight cycles. Lines 4–5 set $h(v)$ to the shortest-path weight $\delta(s, v)$ computed by the Bellman-Ford algorithm for all $v \in V'$. Lines 6–7 compute the new weights $\widehat{w}$. For each pair of vertices $u, v \in V$, the **for** loop of lines 9–12 computes the shortest-path weight $\widehat{\delta}(u, v)$ by calling Dijkstra's algorithm once from each vertex in V. Line 12 stores in matrix entry d_{uv} the correct shortest-path weight $\delta(u, v)$, calculated using equation (23.11). Finally, line 13 returns the completed D matrix. Figure 23.6 depicts the execution of Johnson's algorithm.

If the min-priority queue in Dijkstra's algorithm is implemented by a Fibonacci heap, Johnson's algorithm runs in $O(V^2 \lg V + VE)$ time. The simpler binary min-heap implementation yields a running time of $O(VE \lg V)$, which is still asymptotically faster than the Floyd-Warshall algorithm if the graph is sparse.

Exercises

23.3-1

Use Johnson's algorithm to find the shortest paths between all pairs of vertices in the graph of Figure 23.2. Show the values of h and $\widehat{w}$ computed by the algorithm.

23.3-2
What is the purpose of adding the new vertex s to V, yielding V'?

23.3-3
Suppose that $w(u, v) \geq 0$ for all edges $(u, v) \in E$. What is the relationship between the weight functions w and $\widehat{w}$?

23.3-4
Professor Greenstreet claims that there is a simpler way to reweight edges than the method used in Johnson's algorithm. Letting $w^* = \min\{w(u, v) : (u, v) \in E\}$, just define $\widehat{w}(u, v) = w(u, v) - w^*$ for all edges $(u, v) \in E$. What is wrong with the professor's method of reweighting?

23.3-5
Show that if G contains a 0-weight cycle c, then $\widehat{w}(u, v) = 0$ for every edge (u, v) in c.

23.3-6
Professor Michener claims that there is no need to create a new source vertex in line 1 of JOHNSON. He suggests using $G' = G$ instead and letting s be any vertex. Give an example of a weighted, directed graph G for which incorporating the professor's idea into JOHNSON causes incorrect answers. Assume that $\infty - \infty$ is undefined, and in particular, it is not 0. Then show that if G is strongly connected (every vertex is reachable from every other vertex), the results returned by JOHNSON with the professor's modification are correct.

Problems

23-1 Transitive closure of a dynamic graph
You wish to maintain the transitive closure of a directed graph $G = (V, E)$ as you insert edges into E. That is, after inserting an edge, you update the transitive closure of the edges inserted so far. Start with G having no edges initially, and represent the transitive closure by a boolean matrix.

a. Show how to update the transitive closure $G^* = (V, E^*)$ of a graph $G = (V, E)$ in $O(V^2)$ time when a new edge is added to G.

b. Give an example of a graph G and an edge e such that $\Omega(V^2)$ time is required to update the transitive closure after inserting e into G, no matter what algorithm is used.

c. Give an algorithm for updating the transitive closure as edges are inserted into the graph. For any sequence of r insertions, your algorithm should run in time $\sum_{i=1}^{r} t_i = O(V^3)$, where t_i is the time to update the transitive closure upon inserting the ith edge. Prove that your algorithm attains this time bound.

23-2 Shortest paths in ϵ-dense graphs

A graph $G = (V, E)$ is ***ϵ-dense*** if $|E| = \Theta(V^{1+\epsilon})$ for some constant ϵ in the range $0 < \epsilon \le 1$. d-ary min-heaps (see Problem 6-2 on page 179) provide a way to match the running times of Fibonacci-heap-based shortest-path algorithms on ϵ-dense graphs without using as complicated a data structure.

a. What are the asymptotic running times for the operations INSERT, EXTRACT-MIN, and DECREASE-KEY, as a function of d and the number n of elements in a d-ary min-heap? What are these running times if you choose $d = \Theta(n^\alpha)$ for some constant $0 < \alpha \le 1$? Compare these running times to the amortized costs of these operations for a Fibonacci heap.

b. Show how to compute shortest paths from a single source on an ϵ-dense directed graph $G = (V, E)$ with no negative-weight edges in $O(E)$ time. (*Hint:* Pick d as a function of ϵ.)

c. Show how to solve the all-pairs shortest-paths problem on an ϵ-dense directed graph $G = (V, E)$ with no negative-weight edges in $O(VE)$ time.

d. Show how to solve the all-pairs shortest-paths problem in $O(VE)$ time on an ϵ-dense directed graph $G = (V, E)$ that may have negative-weight edges but has no negative-weight cycles.

Chapter notes

Lawler [276] has a good discussion of the all-pairs shortest-paths problem. He attributes the matrix-multiplication algorithm to the folklore. The Floyd-Warshall algorithm is due to Floyd [144], who based it on a theorem of Warshall [450] that describes how to compute the transitive closure of boolean matrices. Johnson's algorithm is taken from [238].

Several researchers have given improved algorithms for computing shortest paths via matrix multiplication. Fredman [153] shows how to solve the all-pairs shortest paths problem using $O(V^{5/2})$ comparisons between sums of edge weights and obtains an algorithm that runs in $O(V^3(\lg\lg V/\lg V)^{1/3})$ time, which is slightly better than the running time of the Floyd-Warshall algorithm. This bound

has been improved several times, and the fastest algorithm is now by Williams [457], with a running time of $O(V^3/2^{\Omega(\lg^{1/2} V)})$.

Another line of research demonstrates how to apply algorithms for fast matrix multiplication (see the chapter notes for Chapter 4) to the all-pairs shortest paths problem. Let $O(n^\omega)$ be the running time of the fastest algorithm for multiplying two $n \times n$ matrices. Galil and Margalit [170, 171] and Seidel [403] designed algorithms that solve the all-pairs shortest paths problem in undirected, unweighted graphs in $(V^\omega p(V))$ time, where $p(n)$ denotes a particular function that is polylogarithmically bounded in n. In dense graphs, these algorithms are faster than the $O(VE)$ time needed to perform $|V|$ breadth-first searches. Several researchers have extended these results to give algorithms for solving the all-pairs shortest paths problem in undirected graphs in which the edge weights are integers in the range $\{1, 2, \ldots, W\}$. The asymptotically fastest such algorithm, by Shoshan and Zwick [410], runs in $O(WV^\omega p(VW))$ time. In directed graphs, the best algorithm to date is due to Zwick [467] and runs in $\widetilde{O}(W^{1/(4-\omega)} V^{2+1/(4-\omega)})$ time.

Karger, Koller, and Phillips [244] and independently McGeoch [320] have given a time bound that depends on E^*, the set of edges in E that participate in some shortest path. Given a graph with nonnegative edge weights, their algorithms run in $O(VE^* + V^2 \lg V)$ time and improve upon running Dijkstra's algorithm $|V|$ times when $|E^*| = o(E)$. Pettie [355] uses an approach based on component hierarchies to achieve a running time of $O(VE + V^2 \lg\lg V)$, and the same running time is also achieved by Hagerup [205].

Baswana, Hariharan, and Sen [37] examined decremental algorithms, which allow a sequence of intermixed edge deletions and queries, for maintaining all-pairs shortest paths and transitive-closure information. When a path exists, their randomized transitive-closure algorithm can fail to report it with probability $1/n^c$ for an arbitrary $c > 0$. The query times are $O(1)$ with high probability. For transitive closure, the amortized time for each update is $O(V^{4/3} \lg^{1/3} V)$. By comparison, Problem 23-1, in which edges are inserted, asks for an incremental algorithm. For all-pairs shortest paths, the update times depend on the queries. For queries just giving the shortest-path weights, the amortized time per update is $O(V^3/E \lg^2 V)$. To report the actual shortest path, the amortized update time is $\min\left\{O(V^{3/2}\sqrt{\lg V}), O(V^3/E \lg^2 V)\right\}$. Demetrescu and Italiano [111] showed how to handle update and query operations when edges are both inserted and deleted, as long as the range of edge weights is bounded.

Aho, Hopcroft, and Ullman [5] defined an algebraic structure known as a "closed semiring," which serves as a general framework for solving path problems in directed graphs. Both the Floyd-Warshall algorithm and the transitive-closure algorithm from Section 23.2 are instantiations of an all-pairs algorithm based on closed semirings. Maggs and Plotkin [309] showed how to find minimum spanning trees using a closed semiring.

24 Maximum Flow

Just as you can model a road map as a directed graph in order to find the shortest path from one point to another, you can also interpret a directed graph as a "flow network" and use it to answer questions about material flows. Imagine a material coursing through a system from a source, where the material is produced, to a sink, where it is consumed. The source produces the material at some steady rate, and the sink consumes the material at the same rate. The "flow" of the material at any point in the system is intuitively the rate at which the material moves. Flow networks can model many problems, including liquids flowing through pipes, parts through assembly lines, current through electrical networks, and information through communication networks.

You can think of each directed edge in a flow network as a conduit for the material. Each conduit has a stated capacity, given as a maximum rate at which the material can flow through the conduit, such as 200 gallons of liquid per hour through a pipe or 20 amperes of electrical current through a wire. Vertices are conduit junctions, and other than the source and sink, material flows through the vertices without collecting in them. In other words, the rate at which material enters a vertex must equal the rate at which it leaves the vertex. We call this property "flow conservation," and it is equivalent to Kirchhoff's current law when the material is electrical current.

The goal of the maximum-flow problem is to compute the greatest rate for shipping material from the source to the sink without violating any capacity constraints. It is one of the simplest problems concerning flow networks and, as we shall see in this chapter, this problem can be solved by efficient algorithms. Moreover, other network-flow problems are solvable by adapting the basic techniques used in maximum-flow algorithms.

This chapter presents two general methods for solving the maximum-flow problem. Section 24.1 formalizes the notions of flow networks and flows, formally defining the maximum-flow problem. Section 24.2 describes the classical method

of Ford and Fulkerson for finding maximum flows. We finish up with a simple application of this method, finding a maximum matching in an undirected bipartite graph, in Section 24.3. (Section 25.1 will give a more efficient algorithm that is specifically designed to find a maximum matching in a bipartite graph.)

24.1 Flow networks

This section gives a graph-theoretic definition of flow networks, discusses their properties, and defines the maximum-flow problem precisely. It also introduces some helpful notation.

Flow networks and flows

A ***flow network*** $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative ***capacity*** $c(u, v) \geq 0$. We further require that if E contains an edge (u, v), then there is no edge (v, u) in the reverse direction. (We'll see shortly how to work around this restriction.) If $(u, v) \notin E$, then for convenience we define $c(u, v) = 0$, and we disallow self-loops. Each flow network contains two distinguished vertices: a ***source*** s and a ***sink*** t. For convenience, we assume that each vertex lies on some path from the source to the sink. That is, for each vertex $v \in V$, the flow network contains a path $s \leadsto v \leadsto t$. Because each vertex other than s has at least one entering edge, we have $|E| \geq |V| - 1$. Figure 24.1 shows an example of a flow network.

Figure 24.1 **(a)** A flow network $G = (V, E)$ for the Lucky Puck Company's trucking problem. The Vancouver factory is the source s, and the Winnipeg warehouse is the sink t. The company ships pucks through intermediate cities, but only $c(u, v)$ crates per day can go from city u to city v. Each edge is labeled with its capacity. **(b)** A flow f in G with value $|f| = 19$. Each edge (u, v) is labeled by $f(u, v)/c(u, v)$. The slash notation merely separates the flow and capacity and does not indicate division.

We are now ready to define flows more formally. Let $G = (V, E)$ be a flow network with a capacity function c. Let s be the source of the network, and let t be the sink. A ***flow*** in G is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the following two properties:

Capacity constraint: For all $u, v \in V$, we require

$$0 \le f(u, v) \le c(u, v) .$$

The flow from one vertex to another must be nonnegative and must not exceed the given capacity.

Flow conservation: For all $u \in V - \{s, t\}$, we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) .$$

The total flow into a vertex other than the source or sink must equal the total flow out of that vertex—informally, "flow in equals flow out."

When $(u, v) \notin E$, there can be no flow from u to v, and $f(u, v) = 0$.

We call the nonnegative quantity $f(u, v)$ the flow from vertex u to vertex v. The ***value*** $|f|$ of a flow f is defined as

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) , \tag{24.1}$$

that is, the total flow out of the source minus the flow into the source. (Here, the $|\cdot|$ notation denotes flow value, not absolute value or cardinality.) Typically, a flow network does not have any edges into the source, and the flow into the source, given by the summation $\sum_{v \in V} f(v, s)$, is 0. We include it, however, because when we introduce residual networks later in this chapter, the flow into the source can be positive. In the ***maximum-flow problem***, the input is a flow network G with source s and sink t, and the goal is to find a flow of maximum value.

An example of flow

A flow network can model the trucking problem shown in Figure 24.1(a). The Lucky Puck Company has a factory (source s) in Vancouver that manufactures hockey pucks, and it has a warehouse (sink t) in Winnipeg that stocks them. Lucky Puck leases space on trucks from another firm to ship the pucks from the factory to the warehouse. Because the trucks travel over specified routes (edges) between cities (vertices) and have a limited capacity, Lucky Puck can ship at most $c(u, v)$ crates per day between each pair of cities u and v in Figure 24.1(a). Lucky Puck

Figure 24.2 Converting a network with antiparallel edges to an equivalent one with no antiparallel edges. **(a)** A flow network containing both the edges (v_1, v_2) and (v_2, v_1). **(b)** An equivalent network with no antiparallel edges. A new vertex v' was added, and edge (v_1, v_2) was replaced by the pair of edges (v_1, v') and (v', v_2), both with the same capacity as (v_1, v_2).

has no control over these routes and capacities, and so the company cannot alter the flow network shown in Figure 24.1(a). They need to determine the largest number p of crates per day that they can ship and then to produce this amount, since there is no point in producing more pucks than they can ship to their warehouse. Lucky Puck is not concerned with how long it takes for a given puck to get from the factory to the warehouse. They care only that p crates per day leave the factory and p crates per day arrive at the warehouse.

A flow in this network models the "flow" of shipments because the number of crates shipped per day from one city to another is subject to a capacity constraint. Additionally, the model must obey flow conservation, for in a steady state, the rate at which pucks enter an intermediate city must equal the rate at which they leave. Otherwise, crates would accumulate at intermediate cities.

Modeling problems with antiparallel edges

Suppose that the trucking firm offers Lucky Puck the opportunity to lease space for 10 crates in trucks going from Edmonton to Calgary. It might seem natural to add this opportunity to our example and form the network shown in Figure 24.2(a). This network suffers from one problem, however: it violates the original assumption that if edge $(v_1, v_2) \in E$, then $(v_2, v_1) \notin E$. We call the two edges (v_1, v_2) and (v_2, v_1) ***antiparallel***. Thus, to model a flow problem with antiparallel edges, the network must be transformed into an equivalent one containing no antiparallel edges. Figure 24.2(b) displays this equivalent network. To transform the network, choose one of the two antiparallel edges, in this case (v_1, v_2), and split it by adding a new vertex v' and replacing edge (v_1, v_2) with the pair of edges (v_1, v') and (v', v_2). Also set the capacity of both new edges to the capacity of the original edge. The resulting network satisfies the property that if an edge belongs to

Figure 24.3 Converting a multiple-source, multiple-sink maximum-flow problem into a problem with a single source and a single sink. **(a)** A flow network with three sources $S = \{s_1, s_2, s_3\}$ and two sinks $T = \{t_1, t_2\}$. **(b)** An equivalent single-source, single-sink flow network. Add a supersource s and an edge with infinite capacity from s to each of the multiple sources. Also add a supersink t and an edge with infinite capacity from each of the multiple sinks to t.

the network, the reverse edge does not. As Exercise 24.1-1 asks you to prove, the resulting network is equivalent to the original one.

Networks with multiple sources and sinks

A maximum-flow problem may have several sources and sinks, rather than just one of each. The Lucky Puck Company, for example, might actually have a set of m factories $\{s_1, s_2, \ldots, s_m\}$ and a set of n warehouses $\{t_1, t_2, \ldots, t_n\}$, as shown in Figure 24.3(a). Fortunately, this problem is no harder than ordinary maximum flow.

The problem of determining a maximum flow in a network with multiple sources and multiple sinks reduces to an ordinary maximum-flow problem. Figure 24.3(b) shows how to convert the network from (a) to an ordinary flow network with only a single source and a single sink. Add a ***supersource*** s and add a directed edge (s, s_i) with capacity $c(s, s_i) = \infty$ for each $i = 1, 2, \ldots, m$. Similarly, create a new ***supersink*** t and add a directed edge (t_i, t) with capacity $c(t_i, t) = \infty$ for each $i = 1, 2, \ldots, n$. Intuitively, any flow in the network in (a) corresponds to a flow in the network in (b), and vice versa. The single supersource s provides as much flow as desired for the multiple sources s_i, and the single supersink t likewise consumes as much flow as desired for the multiple sinks t_i. Exercise 24.1-2 asks you to prove formally that the two problems are equivalent.

Exercises

24.1-1

Show that splitting an edge in a flow network yields an equivalent network. More formally, suppose that flow network G contains edge (u, v), and define a new flow network G' by creating a new vertex x and replacing (u, v) by new edges (u, x) and (x, v) with $c(u, x) = c(x, v) = c(u, v)$. Show that a maximum flow in G' has the same value as a maximum flow in G.

24.1-2

Extend the flow properties and definitions to the multiple-source, multiple-sink problem. Show that any flow in a multiple-source, multiple-sink flow network corresponds to a flow of identical value in the single-source, single-sink network obtained by adding a supersource and a supersink, and vice versa.

24.1-3

Suppose that a flow network $G = (V, E)$ violates the assumption that the network contains a path $s \leadsto v \leadsto t$ for all vertices $v \in V$. Let u be a vertex for which there is no path $s \leadsto u \leadsto t$. Show that there must exist a maximum flow f in G such that $f(u, v) = f(v, u) = 0$ for all vertices $v \in V$.

24.1-4

Let f be a flow in a network, and let α be a real number. The ***scalar flow product***, denoted αf, is a function from $V \times V$ to $\mathbb{R}$ defined by

$$(\alpha f)(u, v) = \alpha \cdot f(u, v) .$$

Prove that the flows in a network form a ***convex set***. That is, show that if f_1 and f_2 are flows, then so is $\alpha f_1 + (1 - \alpha) f_2$ for all α in the range $0 \le \alpha \le 1$.

24.1-5

State the maximum-flow problem as a linear-programming problem.

24.1-6

Professor Adam has two children who, unfortunately, dislike each other. The problem is so severe that not only do they refuse to walk to school together, but in fact each one refuses to walk on any block that the other child has stepped on that day. The children have no problem with their paths crossing at a corner. Fortunately both the professor's house and the school are on corners, but beyond that he is not sure if it is going to be possible to send both of his children to the same school. The professor has a map of his town. Show how to formulate the problem of determining whether both his children can go to the same school as a maximum-flow problem.

24.1-7

Suppose that, in addition to edge capacities, a flow network has ***vertex capacities***. That is each vertex v has a limit $l(v)$ on how much flow can pass through v. Show how to transform a flow network $G = (V, E)$ with vertex capacities into an equivalent flow network $G' = (V', E')$ without vertex capacities, such that a maximum flow in G' has the same value as a maximum flow in G. How many vertices and edges does G' have?

24.2 The Ford-Fulkerson method

This section presents the Ford-Fulkerson method for solving the maximum-flow problem. We call it a "method" rather than an "algorithm" because it encompasses several implementations with differing running times. The Ford-Fulkerson method depends on three important ideas that transcend the method and are relevant to many flow algorithms and problems: residual networks, augmenting paths, and cuts. These ideas are essential to the important max-flow min-cut theorem (Theorem 24.6), which characterizes the value of a maximum flow in terms of cuts of the flow network. We end this section by presenting one specific implementation of the Ford-Fulkerson method and analyzing its running time.

The Ford-Fulkerson method iteratively increases the value of the flow. It starts with $f(u, v) = 0$ for all $u, v \in V$, giving an initial flow of value 0. Each iteration increases the flow value in G by finding an "augmenting path" in an associated "residual network" G_f. The edges of the augmenting path in G_f indicate on which edges in G to update the flow in order to increase the flow value. Although each iteration of the Ford-Fulkerson method increases the value of the flow, we'll see that the flow on any particular edge of G may increase or decrease. Although it might seem counterintuitive to decrease the flow on an edge, doing so may enable flow to increase on other edges, allowing more flow to travel from the source to the sink. The Ford-Fulkerson method, given in the procedure FORD-FULKERSON-METHOD, repeatedly augments the flow until the residual network has no more augmenting paths. The max-flow min-cut theorem shows that upon termination, this process yields a maximum flow.

```
FORD-FULKERSON-METHOD(G, s, t)
initialize flow f to 0
while there exists an augmenting path p in the residual network G_f
    augment flow f along p
return f
```

In order to implement and analyze the Ford-Fulkerson method, we need to introduce several additional concepts.

Residual networks

Intuitively, given a flow network G and a flow f, the residual network G_f consists of edges whose capacities represent how the flow can change on edges of G. An edge of the flow network can admit an amount of additional flow equal to the edge's capacity minus the flow on that edge. If that value is positive, that edge goes into G_f with a "residual capacity" of $c_f(u,v) = c(u,v) - f(u,v)$. The only edges of G that belong to G_f are those that can admit more flow. Those edges (u,v) whose flow equals their capacity have $c_f(u,v) = 0$, and they do not belong to G_f.

You might be surprised that the residual network G_f can also contain edges that are not in G. As an algorithm manipulates the flow, with the goal of increasing the total flow, it might need to decrease the flow on a particular edge in order to increase the flow elsewhere. In order to represent a possible decrease in the positive flow $f(u,v)$ on an edge in G, the residual network G_f contains an edge (v,u) with residual capacity $c_f(v,u) = f(u,v)$—that is, an edge that can admit flow in the opposite direction to (u,v), at most canceling out the flow on (u,v). These reverse edges in the residual network allow an algorithm to send back flow it has already sent along an edge. Sending flow back along an edge is equivalent to *decreasing* the flow on the edge, which is a necessary operation in many algorithms.

More formally, for a flow network $G = (V,E)$ with source s, sink t, and a flow f, consider a pair of vertices $u, v \in V$. We define the ***residual capacity*** $c_f(u,v)$ by

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E\ , \\ f(v,u) & \text{if } (v,u) \in E\ , \\ 0 & \text{otherwise}\ . \end{cases} \tag{24.2}$$

In a flow network, $(u,v) \in E$ implies $(v,u) \notin E$, and so exactly one case in equation (24.2) applies to each ordered pair of vertices.

As an example of equation (24.2), if $c(u,v) = 16$ and $f(u,v) = 11$, then $f(u,v)$ can increase by up to $c_f(u,v) = 5$ units before exceeding the capacity constraint on edge (u,v). Alternatively, up to 11 units of flow can return from v to u, so that $c_f(v,u) = 11$.

Given a flow network $G = (V,E)$ and a flow f, the ***residual network*** of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}\ . \tag{24.3}$$

Figure 24.4 **(a)** The flow network G and flow f of Figure 24.1(b). **(b)** The residual network G_f with augmenting path p, having residual capacity $c_f(p) = c_f(v_2, v_3) = 4$, in blue. Edges with residual capacity equal to 0, such as (v_1, v_3), are not shown, a convention we follow in the remainder of this section. **(c)** The flow in G that results from augmenting along path p by its residual capacity 4. Edges carrying no flow, such as (v_3, v_2), are labeled only by their capacity, another convention we follow throughout. **(d)** The residual network induced by the flow in (c).

That is, as promised above, each edge of the residual network, or ***residual edge***, can admit a flow that is greater than 0. Figure 24.4(a) repeats the flow network G and flow f of Figure 24.1(b), and Figure 24.4(b) shows the corresponding residual network G_f. The edges in E_f are either edges in E or their reversals, and thus

$$|E_f| \le 2\,|E| \;.$$

Observe that the residual network G_f is similar to a flow network with capacities given by c_f. It does not satisfy the definition of a flow network, however, because it could contain antiparallel edges. Other than this difference, a residual network has the same properties as a flow network, and we can define a flow in the residual network as one that satisfies the definition of a flow, but with respect to capacities c_f in the residual network G_f.

A flow in a residual network provides a roadmap for adding flow to the original flow network. If f is a flow in G and f' is a flow in the corresponding residual network G_f, we define $f \uparrow f'$, the ***augmentation*** of flow f by f', to be a function from $V \times V$ to $\mathbb{R}$, defined by

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E\ , \\ 0 & \text{otherwise}\ . \end{cases} \tag{24.4}$$

The intuition behind this definition follows the definition of the residual network. The flow on (u,v) increases by $f'(u,v)$, but decreases by $f'(v,u)$ because pushing flow on the reverse edge in the residual network signifies decreasing the flow in the original network. Pushing flow on the reverse edge in the residual network is also known as ***cancellation***. For example, suppose that 5 crates of hockey pucks go from u to v and 2 crates go from v to u. That is equivalent (from the perspective of the final result) to sending 3 crates from u to v and none from v to u. Cancellation of this type is crucial for any maximum-flow algorithm.

The following lemma shows that augmenting a flow in G by a flow in G_f yields a new flow in G with a greater flow value.

Lemma 24.1
Let $G = (V,E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G induced by f, and let f' be a flow in G_f. Then the function $f \uparrow f'$ defined in equation (24.4) is a flow in G with value $|f \uparrow f'| = |f| + |f'|$.

Proof We first verify that $f \uparrow f'$ obeys the capacity constraint for each edge in E and flow conservation at each vertex in $V - \{s,t\}$.

For the capacity constraint, first observe that if $(u,v) \in E$, then $c_f(v,u) = f(u,v)$. Because f' is a flow in G_f, we have $f'(v,u) \le c_f(v,u)$, which gives $f'(v,u) \le f(u,v)$. Therefore,

$$\begin{aligned}
(f \uparrow f')(u,v) &= f(u,v) + f'(u,v) - f'(v,u) && \text{(by equation (24.4))} \\
&\ge f(u,v) + f'(u,v) - f(u,v) && \text{(because } f'(v,u) \le f(u,v)\text{)} \\
&= f'(u,v) \\
&\ge 0\ .
\end{aligned}$$

In addition,

$$\begin{aligned}
&(f \uparrow f')(u,v) \\
&\qquad = f(u,v) + f'(u,v) - f'(v,u) && \text{(by equation (24.4))} \\
&\qquad \le f(u,v) + f'(u,v) && \text{(because flows are nonnegative)} \\
&\qquad \le f(u,v) + c_f(u,v) && \text{(capacity constraint)} \\
&\qquad = f(u,v) + c(u,v) - f(u,v) && \text{(definition of } c_f\text{)} \\
&\qquad = c(u,v)\ .
\end{aligned}$$

To show that flow conservation holds and that $|f \uparrow f'| = |f| + |f'|$, we first prove the claim that for all $u \in V$, we have

$$\sum_{v \in V} (f \uparrow f')(u,v) - \sum_{v \in V} (f \uparrow f')(v,u)$$
$$= \sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) + \sum_{v \in V} f'(u,v) - \sum_{v \in V} f'(v,u) . \qquad (24.5)$$

Because we disallow antiparallel edges in G (but not in G_f), we know that for each vertex u, there can be an edge (u,v) or (v,u) in G, but never both. For a fixed vertex u, define $V_l(u) = \{v : (u,v) \in E\}$ to be the set of vertices with edges in G leaving u, and define $V_e(u) = \{v : (v,u) \in E\}$ to be the set of vertices with edges in G entering u. We have $V_l(u) \cup V_e(u) \subseteq V$ and, because G contains no antiparallel edges, $V_l(u) \cap V_e(u) = \emptyset$. By the definition of flow augmentation in equation (24.4), only vertices v in $V_l(u)$ can have positive $(f \uparrow f')(u,v)$, and only vertices v in $V_e(u)$ can have positive $(f \uparrow f')(v,u)$. Starting from the left-hand side of equation (24.5), we use this fact and then reorder and group terms, giving

$$\begin{aligned}
&\sum_{v \in V} (f \uparrow f')(u,v) - \sum_{v \in V} (f \uparrow f')(v,u) \\
&\quad = \sum_{v \in V_l(u)} (f \uparrow f')(u,v) - \sum_{v \in V_e(u)} (f \uparrow f')(v,u) \\
&\quad = \sum_{v \in V_l(u)} (f(u,v) + f'(u,v) - f'(v,u)) - \sum_{v \in V_e(u)} (f(v,u) + f'(v,u) - f'(u,v)) \\
&\quad = \sum_{v \in V_l(u)} f(u,v) + \sum_{v \in V_l(u)} f'(u,v) - \sum_{v \in V_l(u)} f'(v,u) \\
&\qquad\qquad - \sum_{v \in V_e(u)} f(v,u) - \sum_{v \in V_e(u)} f'(v,u) + \sum_{v \in V_e(u)} f'(u,v) \\
&\quad = \sum_{v \in V_l(u)} f(u,v) - \sum_{v \in V_e(u)} f(v,u) \\
&\qquad\qquad + \sum_{v \in V_l(u)} f'(u,v) + \sum_{v \in V_e(u)} f'(u,v) - \sum_{v \in V_l(u)} f'(v,u) - \sum_{v \in V_e(u)} f'(v,u) \\
&\quad = \sum_{v \in V_l(u)} f(u,v) - \sum_{v \in V_e(u)} f(v,u) + \sum_{v \in V_l(u) \cup V_e(u)} f'(u,v) - \sum_{v \in V_l(u) \cup V_e(u)} f'(v,u) . \qquad (24.6)
\end{aligned}$$

In equation (24.6), all four summations can extend to sum over V, since each additional term has value 0. (Exercise 24.2-1 asks you to prove this formally.) Taking all four summations over V, instead of just subsets of V, proves the claim in equation (24.5).

Now we are ready to prove flow conservation for $f \uparrow f'$ and that $|f \uparrow f'| = |f| + |f'|$. For the latter property, let $u = s$ in equation (24.5). Then, we have

$$\begin{aligned}
|f \uparrow f'| &= \sum_{v \in V} (f \uparrow f')(s, v) - \sum_{v \in V} (f \uparrow f')(v, s) \\
&= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{v \in V} f'(s, v) - \sum_{v \in V} f'(v, s) \\
&= |f| + |f'| \ .
\end{aligned}$$

For flow conservation, observe that for any vertex u that is neither s nor t, flow conservation for f and f' means that the right-hand side of equation (24.5) is 0, and thus $\sum_{v \in V} (f \uparrow f')(u, v) = \sum_{v \in V} (f \uparrow f')(v, u)$. ■

Augmenting paths

Given a flow network $G = (V, E)$ and a flow f, an ***augmenting path*** p is a simple path from s to t in the residual network G_f. By the definition of the residual network, the flow on an edge (u, v) of an augmenting path may increase by up to $c_f(u, v)$ without violating the capacity constraint on whichever of (u, v) and (v, u) belongs to the original flow network G.

The blue path in Figure 24.4(b) is an augmenting path. Treating the residual network G_f in the figure as a flow network, the flow through each edge of this path can increase by up to 4 units without violating a capacity constraint, since the smallest residual capacity on this path is $c_f(v_2, v_3) = 4$. We call the maximum amount by which we can increase the flow on each edge in an augmenting path p the ***residual capacity*** of p, given by

$$c_f(p) = \min \{c_f(u, v) : (u, v) \text{ is in } p\} \ .$$

The following lemma, which Exercise 24.2-7 asks you to prove, makes the above argument more precise.

Lemma 24.2
Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Define a function $f_p : V \times V \to \mathbb{R}$ by

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p \ , \\ 0 & \text{otherwise} \ . \end{cases} \tag{24.7}$$

Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$. ■

The following corollary shows that augmenting f by f_p produces another flow in G whose value is closer to the maximum. Figure 24.4(c) shows the result of augmenting the flow f from Figure 24.4(a) by the flow f_p in Figure 24.4(b), and Figure 24.4(d) shows the ensuing residual network.

Corollary 24.3
Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be defined as in equation (24.7), and suppose that f is augmented by f_p. Then the function $f \uparrow f_p$ is a flow in G with value $|f \uparrow f_p| = |f| + |f_p| > |f|$.

Proof Immediate from Lemmas 24.1 and 24.2. ■

Cuts of flow networks

The Ford-Fulkerson method repeatedly augments the flow along augmenting paths until it has found a maximum flow. How do we know that when the algorithm terminates, it has actually found a maximum flow? The max-flow min-cut theorem, which we will prove shortly, tells us that a flow is maximum if and only if its residual network contains no augmenting path. To prove this theorem, though, we must first explore the notion of a cut of a flow network.

A ***cut*** (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. (This definition is similar to the definition of "cut" that we used for minimum spanning trees in Chapter 21, except that here we are cutting a directed graph rather than an undirected graph, and we insist that $s \in S$ and $t \in T$.) If f is a flow, then the ***net flow*** $f(S, T)$ across the cut (S, T) is defined to be

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) . \tag{24.8}$$

The ***capacity*** of the cut (S, T) is

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v) . \tag{24.9}$$

A ***minimum cut*** of a network is a cut whose capacity is minimum over all cuts of the network.

You probably noticed that the definitions of flow across a cut and capacity of a cut differ in that flow counts edges going in both directions across the cut, but capacity counts only edges going from the source side of the cut toward the sink side. This asymmetry is intentional and important. The reason for this difference will become apparent later in this section.

Figure 24.5 shows the cut $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$ in the flow network of Figure 24.1(b). The net flow across this cut is

$$\begin{aligned} f(v_1, v_3) + f(v_2, v_4) - f(v_3, v_2) &= 12 + 11 - 4 \\ &= 19 , \end{aligned}$$

and the capacity of this cut is

Figure 24.5 A cut (S, T) in the flow network of Figure 24.1(b), where $S = \{s, v_1, v_2\}$ and $T = \{v_3, v_4, t\}$. The vertices in S are orange, and the vertices in T are tan. The net flow across (S, T) is $f(S, T) = 19$, and the capacity is $c(S, T) = 26$.

$$\begin{aligned} c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\ &= 26 . \end{aligned}$$

The following lemma shows that, for a given flow f, the net flow across any cut is the same, and it equals $|f|$, the value of the flow.

Lemma 24.4
Let f be a flow in a flow network G with source s and sink t, and let (S, T) be any cut of G. Then the net flow across (S, T) is $f(S, T) = |f|$.

Proof For any vertex $u \in V - \{s, t\}$, rewrite the flow-conservation condition as

$$\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0 . \tag{24.10}$$

Taking the definition of $|f|$ from equation (24.1) and adding the left-hand side of equation (24.10), which equals 0, summed over all vertices in $S - \{s\}$, gives

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S - \{s\}} \left(\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) \right) .$$

Expanding the right-hand summation and regrouping terms yields

$$\begin{aligned} |f| &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S - \{s\}} \sum_{v \in V} f(u, v) - \sum_{u \in S - \{s\}} \sum_{v \in V} f(v, u) \\ &= \sum_{v \in V} \left(f(s, v) + \sum_{u \in S - \{s\}} f(u, v) \right) - \sum_{v \in V} \left(f(v, s) + \sum_{u \in S - \{s\}} f(v, u) \right) \\ &= \sum_{v \in V} \sum_{u \in S} f(u, v) - \sum_{v \in V} \sum_{u \in S} f(v, u) . \end{aligned}$$

Because $V = S \cup T$ and $S \cap T = \emptyset$, splitting each summation over V into summations over S and T gives

$$\begin{aligned}
|f| &= \sum_{v \in S} \sum_{u \in S} f(u,v) + \sum_{v \in T} \sum_{u \in S} f(u,v) - \sum_{v \in S} \sum_{u \in S} f(v,u) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\
&= \sum_{v \in T} \sum_{u \in S} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\
&\qquad + \left(\sum_{v \in S} \sum_{u \in S} f(u,v) - \sum_{v \in S} \sum_{u \in S} f(v,u) \right) .
\end{aligned}$$

The two summations within the parentheses are actually the same, since for all vertices $x, y \in S$, the term $f(x,y)$ appears once in each summation. Hence, these summations cancel, yielding

$$\begin{aligned}
|f| &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\
&= f(S,T) .
\end{aligned}$$

■

A corollary to Lemma 24.4 shows how cut capacities bound the value of a flow.

Corollary 24.5
The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

Proof Let (S,T) be any cut of G and let f be any flow. By Lemma 24.4 and the capacity constraint,

$$\begin{aligned}
|f| &= f(S,T) \\
&= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\
&\le \sum_{u \in S} \sum_{v \in T} f(u,v) \\
&\le \sum_{u \in S} \sum_{v \in T} c(u,v) \\
&= c(S,T) .
\end{aligned}$$

■

Corollary 24.5 yields the immediate consequence that the value of a maximum flow in a network is bounded from above by the capacity of a minimum cut of the network. The important max-flow min-cut theorem, which we now state and prove, says that the value of a maximum flow is in fact equal to the capacity of a minimum cut.

Theorem 24.6 (Max-flow min-cut theorem)
If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

Proof (1) $\Rightarrow$ (2): Suppose for the sake of contradiction that f is a maximum flow in G but that G_f has an augmenting path p. Then, by Corollary 24.3, the flow found by augmenting f by f_p, where f_p is given by equation (24.7), is a flow in G with value strictly greater than $|f|$, contradicting the assumption that f is a maximum flow.

(2) $\Rightarrow$ (3): Suppose that G_f has no augmenting path, that is, that G_f contains no path from s to t. Define

$$S = \{v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$$

and $T = V - S$. The partition (S, T) is a cut: we have $s \in S$ trivially and $t \notin S$ because there is no path from s to t in G_f. Now consider a pair of vertices $u \in S$ and $v \in T$. If $(u, v) \in E$, we must have $f(u, v) = c(u, v)$, since otherwise $(u, v) \in E_f$, which would place v in set S. If $(v, u) \in E$, we must have $f(v, u) = 0$, because otherwise $c_f(u, v) = f(v, u)$ would be positive and we would have $(u, v) \in E_f$, which again would place v in S. Of course, if neither (u, v) nor (v, u) belongs to E, then $f(u, v) = f(v, u) = 0$. We thus have

$$\begin{aligned} f(S, T) &= \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\ &= \sum_{u \in S} \sum_{v \in T} c(u, v) - \sum_{v \in T} \sum_{u \in S} 0 \\ &= c(S, T) . \end{aligned}$$

By Lemma 24.4, therefore, $|f| = f(S, T) = c(S, T)$.

(3) $\Rightarrow$ (1): By Corollary 24.5, $|f| \le c(S, T)$ for all cuts (S, T). The condition $|f| = c(S, T)$ thus implies that f is a maximum flow. ■

The basic Ford-Fulkerson algorithm

Each iteration of the Ford-Fulkerson method finds *some* augmenting path p and uses p to modify the flow f. As Lemma 24.2 and Corollary 24.3 suggest, replacing f by $f \uparrow f_p$ produces a new flow whose value is $|f| + |f_p|$. The procedure FORD-FULKERSON on the next page implements the method by updating the flow

attribute $(u, v).f$ for each edge $(u, v) \in E$.[1] It assumes implicitly that $(u, v).f = 0$ if $(u, v) \notin E$. The procedure also assumes that the capacities $c(u, v)$ come with the flow network, and that $c(u, v) = 0$ if $(u, v) \notin E$. The procedure computes the residual capacity $c_f(u, v)$ in accordance with the formula (24.2). The expression $c_f(p)$ in the code is just a temporary variable that stores the residual capacity of the path p.

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min {c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ G.E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
9  return f
```

The FORD-FULKERSON procedure simply expands on the FORD-FULKERSON-METHOD pseudocode given earlier. Figure 24.6 shows the result of each iteration in a sample run. Lines 1–2 initialize the flow f to 0. The **while** loop of lines 3–8 repeatedly finds an augmenting path p in G_f and augments flow f along p by the residual capacity $c_f(p)$. Each residual edge in path p is either an edge in the original network or the reversal of an edge in the original network. Lines 6–8 update the flow in each case appropriately, adding flow when the residual edge is an original edge and subtracting it otherwise. When no augmenting paths exist, the flow f is a maximum flow.

Analysis of Ford-Fulkerson

The running time of FORD-FULKERSON depends on the augmenting path p and how it's found in line 3. If the edge capacities are irrational numbers, it's possible to choose the augmenting path so that the algorithm never terminates: the value of the flow increases with successive augmentations, but never converges to the maximum flow value. The good news is that if the algorithm finds the augmenting path by using a breadth-first search (which we saw in Section 20.2), it runs in

[1] Recall from Section 20.1 that we represent an attribute f for edge (u, v) with the same style of notation—$(u, v).f$—that we use for an attribute of any other object.

Figure 24.6 The execution of the basic Ford-Fulkerson algorithm. **(a)–(e)** Successive iterations of the **while** loop. The left side of each part shows the residual network G_f from line 3 with a blue augmenting path p. The right side of each part shows the new flow f that results from augmenting f by f_p. The residual network in (a) is the input flow network G. **(f)** The residual network at the last **while** loop test. It has no augmenting paths, and the flow f shown in (e) is therefore a maximum flow. The value of the maximum flow found is 23.

Figure 24.7 **(a)** A flow network for which FORD-FULKERSON can take $\Theta(E\ |f^*|)$ time, where f^* is a maximum flow, shown here with $|f^*| = 2{,}000{,}000$. The blue path is an augmenting path with residual capacity 1. **(b)** The resulting residual network, with another augmenting path whose residual capacity is 1. **(c)** The resulting residual network.

polynomial time. Before proving this result, we obtain a simple bound for the case in which all capacities are integers and the algorithm finds any augmenting path.

In practice, the maximum-flow problem often arises with integer capacities. If the capacities are rational numbers, an appropriate scaling transformation can make them all integers. If f^* denotes a maximum flow in the transformed network, then a straightforward implementation of FORD-FULKERSON executes the **while** loop of lines 3–8 at most $|f^*|$ times, since the flow value increases by at least 1 unit in each iteration.

A good implementation should perform the work done within the **while** loop efficiently. It should represent the flow network $G = (V, E)$ with the right data structure and find an augmenting path by a linear-time algorithm. Let's assume that the implementation keeps a data structure corresponding to a directed graph $G' = (V, E')$, where $E' = \{(u, v) : (u, v) \in E \text{ or } (v, u) \in E\}$. Edges in the network G are also edges in G', making it straightforward to maintain capacities and flows in this data structure. Given a flow f on G, the edges in the residual network G_f consist of all edges (u, v) of G' such that $c_f(u, v) > 0$, where c_f conforms to equation (24.2). The time to find a path in a residual network is therefore $O(V + E') = O(E)$ using either depth-first search or breadth-first search. Each iteration of the **while** loop thus takes $O(E)$ time, as does the initialization in lines 1–2, making the total running time of the FORD-FULKERSON algorithm $O(E\ |f^*|)$.

When the capacities are integers and the optimal flow value $|f^*|$ is small, the running time of the Ford-Fulkerson algorithm is good. Figure 24.7(a) shows an example of what can happen on a simple flow network for which $|f^*|$ is large. A maximum flow in this network has value 2,000,000: 1,000,000 units of flow traverse the path $s \to u \to t$, and another 1,000,000 units traverse the path $s \to v \to t$. If the first augmenting path found by FORD-FULKERSON is $s \to u \to v \to t$, shown

in Figure 24.7(a), the flow has value 1 after the first iteration. The resulting residual network appears in Figure 24.7(b). If the second iteration finds the augmenting path $s \to v \to u \to t$, as shown in Figure 24.7(b), the flow then has value 2. Figure 24.7(c) shows the resulting residual network. If the algorithm continues alternately choosing the augmenting paths $s \to u \to v \to t$ and $s \to v \to u \to t$, it performs a total of 2,000,000 augmentations, increasing the flow value by only 1 unit in each.

The Edmonds-Karp algorithm

In the example of Figure 24.7, the algorithm never chooses the augmenting path with the fewest edges. It should have. By using breadth-first search to find an augmenting path in the residual network, the algorithm runs in polynomial time, independent of the maximum flow value. We call the Ford-Fulkerson method so implemented the ***Edmonds-Karp algorithm***.

Let's now prove that the Edmonds-Karp algorithm runs in $O(VE^2)$ time. The analysis depends on the distances to vertices in the residual network G_f. The notation $\delta_f(u, v)$ denotes the shortest-path distance from u to v in G_f, where each edge has unit distance.

Lemma 24.7
If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then for all vertices $v \in V - \{s, t\}$, the shortest-path distance $\delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation.

Proof We'll suppose that a flow augmentation occurs that causes the shortest-path distance from s to some vertex $v \in V - \{s, t\}$ to decrease and then derive a contradiction. Let f be the flow just before an augmentation that decreases some shortest-path distance, and let f' be the flow just afterward. Let v be a vertex with the minimum $\delta_{f'}(s, v)$ whose distance was decreased by the augmentation, so that $\delta_{f'}(s, v) < \delta_f(s, v)$. Let $p = s \leadsto u \to v$ be a shortest path from s to v in $G_{f'}$, so that $(u, v) \in E_{f'}$ and

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1 . \tag{24.11}$$

Because of how we chose v, we know that the distance of vertex u from the source s did not decrease, that is,

$$\delta_{f'}(s, u) \geq \delta_f(s, u) . \tag{24.12}$$

We claim that $(u, v) \notin E_f$. Why? If we have $(u, v) \in E_f$, then we also have

$$\begin{aligned}
\delta_f(s,v) &\le \delta_f(s,u) + 1 && \text{(by Lemma 22.10, the triangle inequality)} \\
&\le \delta_{f'}(s,u) + 1 && \text{(by inequality (24.12))} \\
&= \delta_{f'}(s,v) && \text{(by equation (24.11))} ,
\end{aligned}$$

which contradicts our assumption that $\delta_{f'}(s,v) < \delta_f(s,v)$.

How can we have $(u,v) \notin E_f$ and $(u,v) \in E_{f'}$? The augmentation must have increased the flow from v to u, so that edge (v,u) was in the augmenting path. The augmenting path was a shortest path from s to t in G_f, and since any subpath of a shortest path is itself a shortest path, this augmenting path includes a shortest path from s to u in G_f that has (v,u) as its last edge. Therefore,

$$\begin{aligned}
\delta_f(s,v) &= \delta_f(s,u) - 1 \\
&\le \delta_{f'}(s,u) - 1 && \text{(by inequality (24.12))} \\
&= \delta_{f'}(s,v) - 2 && \text{(by equation (24.11))} ,
\end{aligned}$$

so that $\delta_{f'}(s,v) > \delta_f(s,v)$, contradicting our assumption that $\delta_{f'}(s,v) < \delta_f(s,v)$. We conclude that our assumption that such a vertex v exists is incorrect. ■

The next theorem bounds the number of iterations of the Edmonds-Karp algorithm.

Theorem 24.8
If the Edmonds-Karp algorithm is run on a flow network $G = (V,E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(VE)$.

Proof We say that an edge (u,v) in a residual network G_f is ***critical*** on an augmenting path p if the residual capacity of p is the residual capacity of (u,v), that is, if $c_f(p) = c_f(u,v)$. After flow is augmented along an augmenting path, any critical edge on the path disappears from the residual network. Moreover, at least one edge on any augmenting path must be critical. We'll show that each of the $|E|$ edges can become critical at most $|V|/2$ times.

Let u and v be vertices in V that are connected by an edge in E. Since augmenting paths are shortest paths, when (u,v) is critical for the first time, we have

$$\delta_f(s,v) = \delta_f(s,u) + 1 .$$

Once the flow is augmented, the edge (u,v) disappears from the residual network. It cannot reappear later on another augmenting path until after the flow from u to v is decreased, which occurs only if (v,u) appears on an augmenting path. If f' is the flow in G when this event occurs, then we have

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1 .$$

Since $\delta_f(s, v) \le \delta_{f'}(s, v)$ by Lemma 24.7, we have

$$\begin{aligned}\delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1 \\ &\ge \delta_f(s, v) + 1 \\ &= \delta_f(s, u) + 2 .\end{aligned}$$

Consequently, from the time (u, v) becomes critical to the time when it next becomes critical, the distance of u from the source increases by at least 2. The distance of u from the source is initially at least 0. Because edge (u, v) is on an augmenting path, and augmenting paths end at t, we know that u cannot be t, so that in any residual network that has a path from s to u, the shortest such path has at most $|V| - 2$ edges. Thus, after the first time that (u, v) becomes critical, it can become critical at most $(|V| - 2)/2 = |V|/2 - 1$ times more, for a total of at most $|V|/2$ times. Since there are $O(E)$ pairs of vertices that can have an edge between them in a residual network, the total number of critical edges during the entire execution of the Edmonds-Karp algorithm is $O(VE)$. Each augmenting path has at least one critical edge, and hence the theorem follows. ■

Because each iteration of FORD-FULKERSON takes $O(E)$ time when it uses breadth-first search to find the augmenting path, the total running time of the Edmonds-Karp algorithm is $O(VE^2)$.

Exercises

24.2-1
Prove that the summations in equation (24.6) equal the summations on the right-hand side of equation (24.5).

24.2-2
In Figure 24.1(b), what is the net flow across the cut $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? What is the capacity of this cut?

24.2-3
Show the execution of the Edmonds-Karp algorithm on the flow network of Figure 24.1(a).

24.2-4
In the example of Figure 24.6, what is the minimum cut corresponding to the maximum flow shown? Of the augmenting paths appearing in the example, which one cancels flow?

24.2-5
The construction in Section 24.1 to convert a flow network with multiple sources and sinks into a single-source, single-sink network adds edges with infinite capacity. Prove that any flow in the resulting network has a finite value if the edges of the original network with multiple sources and sinks have finite capacity.

24.2-6
Suppose that each source s_i in a flow network with multiple sources and sinks produces exactly p_i units of flow, so that $\sum_{v \in V} f(s_i, v) = p_i$. Suppose also that each sink t_j consumes exactly q_j units, so that $\sum_{v \in V} f(v, t_j) = q_j$, where $\sum_i p_i = \sum_j q_j$. Show how to convert the problem of finding a flow f that obeys these additional constraints into the problem of finding a maximum flow in a single-source, single-sink flow network.

24.2-7
Prove Lemma 24.2.

24.2-8
Suppose that we redefine the residual network to disallow edges into s. Argue that the procedure FORD-FULKERSON still correctly computes a maximum flow.

24.2-9
Suppose that both f and f' are flows in a flow network. Does the augmented flow $f \uparrow f'$ satisfy the flow conservation property? Does it satisfy the capacity constraint?

24.2-10
Show how to find a maximum flow in a flow network $G = (V, E)$ by a sequence of at most $|E|$ augmenting paths. (*Hint:* Determine the paths *after* finding the maximum flow.)

24.2-11
The ***edge connectivity*** of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how to determine the edge connectivity of an undirected graph $G = (V, E)$ by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V + E)$ vertices and $O(E)$ edges.

24.2-12
You are given a flow network G, where G contains edges entering the source s. Let f be a flow in G with $|f| \geq 0$ in which one of the edges (v, s) entering

the source has $f(v, s) = 1$. Prove that there must exist another flow f' with $f'(v, s) = 0$ such that $|f| = |f'|$. Give an $O(E)$-time algorithm to compute f', given f and assuming that all edge capacities are integers.

24.2-13

Suppose that you wish to find, among all minimum cuts in a flow network G with integer capacities, one that contains the smallest number of edges. Show how to modify the capacities of G to create a new flow network G' in which any minimum cut in G' is a minimum cut with the smallest number of edges in G.

24.3 Maximum bipartite matching

Some combinatorial problems can be cast as maximum-flow problems, such as the multiple-source, multiple-sink maximum-flow problem from Section 24.1. Other combinatorial problems seem on the surface to have little to do with flow networks, but they can in fact be reduced to maximum-flow problems. This section presents one such problem: finding a maximum matching in a bipartite graph. In order to solve this problem, we'll take advantage of an integrality property provided by the Ford-Fulkerson method. We'll also see how to use the Ford-Fulkerson method to solve the maximum-bipartite-matching problem on a graph $G = (V, E)$ in $O(VE)$ time. Section 25.1 will present an algorithm specifically designed to solve this problem.

The maximum-bipartite-matching problem

Given an undirected graph $G = (V, E)$, a ***matching*** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v. We say that a vertex $v \in V$ is ***matched*** by the matching M if some edge in M is incident on v, and otherwise, v is ***unmatched***. A ***maximum matching*** is a matching of maximum cardinality, that is, a matching M such that for any matching M', we have $|M| \geq |M'|$. In this section, we restrict our attention to finding maximum matchings in bipartite graphs: graphs in which the vertex set can be partitioned into $V = L \cup R$, where L and R are disjoint and all edges in E go between L and R. We further assume that every vertex in V has at least one incident edge. Figure 24.8 illustrates the notion of a matching in a bipartite graph.

The problem of finding a maximum matching in a bipartite graph has many practical applications. As an example, consider matching a set L of machines with a set R of tasks to be performed simultaneously. An edge (u, v) in E signifies that

Figure 24.8 A bipartite graph $G = (V, E)$ with vertex partition $V = L \cup R$. **(a)** A matching with cardinality 2, indicated by blue edges. **(b)** A maximum matching with cardinality 3. **(c)** The corresponding flow network G' with a maximum flow shown. Each edge has unit capacity. Blue edges have a flow of 1, and all other edges carry no flow. The blue edges from L to R correspond to those in the maximum matching from (b).

a particular machine $u \in L$ is capable of performing a particular task $v \in R$. A maximum matching provides work for as many machines as possible.

Finding a maximum bipartite matching

The Ford-Fulkerson method provides a basis for finding a maximum matching in an undirected bipartite graph $G = (V, E)$ in time polynomial in $|V|$ and $|E|$. The trick is to construct a flow network in which flows correspond to matchings, as shown in Figure 24.8(c). We define the ***corresponding flow network*** $G' = (V', E')$ for the bipartite graph G as follows. Let the source s and sink t be new vertices not in V, and let $V' = V \cup \{s, t\}$. If the vertex partition of G is $V = L \cup R$, the directed edges of G' are the edges of E, directed from L to R, along with $|V|$ new directed edges:

$$\begin{aligned} E' = {} & \{(s, u) : u \in L\} \\ & \cup \{(u, v) : u \in L, v \in R, \text{ and } (u, v) \in E\} \\ & \cup \{(v, t) : v \in R\} \ . \end{aligned}$$

To complete the construction, assign unit capacity to each edge in E'. Since each vertex in V has at least one incident edge, $|E| \geq |V|/2$. Thus, $|E| \leq |E'| = |E| + |V| \leq 3\,|E|$, and so $|E'| = \Theta(E)$.

The following lemma shows that a matching in G corresponds directly to a flow in G's corresponding flow network G'. We say that a flow f on a flow network $G = (V, E)$ is ***integer-valued*** if $f(u, v)$ is an integer for all $(u, v) \in V \times V$.

Lemma 24.9
Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let $G' = (V', E')$ be its corresponding flow network. If M is a matching in G, then there is an integer-valued flow f in G' with value $|f| = |M|$. Conversely, if f is an integer-valued flow in G', then there is a matching M in G with cardinality $|M| = |f|$ consisting of edges $(u, v) \in E$ such that $f(u, v) > 0$.

Proof We first show that a matching M in G corresponds to an integer-valued flow f in G'. Define f as follows. If $(u, v) \in M$, then $f(s, u) = f(u, v) = f(v, t) = 1$. For all other edges $(u, v) \in E'$, define $f(u, v) = 0$. It is simple to verify that f satisfies the capacity constraint and flow conservation.

Intuitively, each edge $(u, v) \in M$ corresponds to 1 unit of flow in G' that traverses the path $s \to u \to v \to t$. Moreover, the paths induced by edges in M are vertex-disjoint, except for s and t. The net flow across cut $(L \cup \{s\}, R \cup \{t\})$ is equal to $|M|$, and thus, by Lemma 24.4, the value of the flow is $|f| = |M|$.

To prove the converse, let f be an integer-valued flow in G' and, as in the statement of the lemma, let

$$M = \{(u, v) : u \in L,\ v \in R,\ \text{and } f(u, v) > 0\} .$$

Each vertex $u \in L$ has only one entering edge, namely (s, u), and its capacity is 1. Thus, each $u \in L$ has at most 1 unit of flow entering it, and if 1 unit of flow does enter, by flow conservation, 1 unit of flow must leave. Furthermore, since the flow f is integer-valued, for each $u \in L$, the 1 unit of flow can enter on at most one edge and can leave on at most one edge. Thus, 1 unit of flow enters u if and only if there is exactly one vertex $v \in R$ such that $f(u, v) = 1$, and at most one edge leaving each $u \in L$ carries positive flow. A symmetric argument applies to each $v \in R$. The set M is therefore a matching.

To see that $|M| = |f|$, observe that of the edges $(u, v) \in E'$ such that $u \in L$ and $v \in R$,

$$f(u, v) = \begin{cases} 1 & \text{if } (u, v) \in M , \\ 0 & \text{if } (u, v) \notin M . \end{cases}$$

Consequently, $f(L \cup \{s\}, R \cup \{t\})$, the net flow across cut $(L \cup \{s\}, R \cup \{t\})$, is equal to $|M|$. Lemma 24.4 gives that $|f| = f(L \cup \{s\}, R \cup \{t\}) = |M|$. ■

Based on Lemma 24.9, we would like to conclude that a maximum matching in a bipartite graph G corresponds to a maximum flow in its corresponding flow

network G', and therefore running a maximum-flow algorithm on G' provides a maximum matching in G. The only hitch in this reasoning is that the maximum-flow algorithm might return a flow in G' for which some $f(u, v)$ is not an integer, even though the flow value $|f|$ must be an integer. The following theorem shows that the Ford-Fulkerson method cannot produce a solution with this problem.

Theorem 24.10 (Integrality theorem)
If the capacity function c takes on only integer values, then the maximum flow f produced by the Ford-Fulkerson method has the property that $|f|$ is an integer. Moreover, for all vertices u and v, the value of $f(u, v)$ is an integer.

Proof Exercise 24.3-2 asks you to provide the proof by induction on the number of iterations. ■

We can now prove the following corollary to Lemma 24.9.

Corollary 24.11
The cardinality of a maximum matching M in a bipartite graph G equals the value of a maximum flow f in its corresponding flow network G'.

Proof We use the nomenclature from Lemma 24.9. Suppose that M is a maximum matching in G and that the corresponding flow f in G' is not maximum. Then there is a maximum flow f' in G' such that $|f'| > |f|$. Since the capacities in G' are integer-valued, by Theorem 24.10, we can assume that f' is integer-valued. Thus, f' corresponds to a matching M' in G with cardinality $|M'| = |f'| > |f| = |M|$, contradicting our assumption that M is a maximum matching. In a similar manner, we can show that if f is a maximum flow in G', its corresponding matching is a maximum matching on G. ■

Thus, to find a maximum matching in a bipartite undirected graph G, create the flow network G', run the Ford-Fulkerson method on G', and convert the integer-valued maximum flow found into a maximum matching for G. Since any matching in a bipartite graph has cardinality at most $\min\{|L|, |R|\} = O(V)$, the value of the maximum flow in G' is $O(V)$. Therefore, finding a maximum matching in a bipartite graph takes $O(VE') = O(VE)$ time, since $|E'| = \Theta(E)$.

Exercises

24.3-1
Run the Ford-Fulkerson algorithm on the flow network in Figure 24.8(c) and show the residual network after each flow augmentation. Number the vertices in L top

to bottom from 1 to 5 and in R top to bottom from 6 to 9. For each iteration, pick the augmenting path that is lexicographically smallest.

24.3-2
Prove Theorem 24.10. Use induction on the number of iterations of the Ford-Fulkerson method.

24.3-3
Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let G' be its corresponding flow network. Give a good upper bound on the length of any augmenting path found in G' during the execution of FORD-FULKERSON.

Problems

24-1 Escape problem
An $n \times n$ ***grid*** is an undirected graph consisting of n rows and n columns of vertices, as shown in Figure 24.9. We denote the vertex in the ith row and the jth column by (i, j). All vertices in a grid have exactly four neighbors, except for the boundary vertices, which are the points (i, j) for which $i = 1, i = n, j = 1$, or $j = n$.

Given $m \le n^2$ starting points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ in the grid, the ***escape problem*** is to determine whether there are m vertex-disjoint paths from the starting points to any m different points on the boundary. For example, the grid in Figure 24.9(a) has an escape, but the grid in Figure 24.9(b) does not.

(a) (b)

Figure 24.9 Grids for the escape problem. Starting points are blue, and other grid vertices are tan. **(a)** A grid with an escape, shown by blue paths. **(b)** A grid with no escape.

a. Consider a flow network in which vertices, as well as edges, have capacities. That is, the total positive flow entering any given vertex is subject to a capacity constraint. Show how to reduce the problem of determining the maximum flow in a network with edge and vertex capacities to an ordinary maximum-flow problem on a flow network of comparable size.

b. Describe an efficient algorithm to solve the escape problem, and analyze its running time.

24-2 Minimum path cover

A ***path cover*** of a directed graph $G = (V, E)$ is a set P of vertex-disjoint paths such that every vertex in V is included in exactly one path in P. Paths may start and end anywhere, and they may be of any length, including 0. A ***minimum path cover*** of G is a path cover containing the fewest possible paths.

a. Give an efficient algorithm to find a minimum path cover of a directed acyclic graph $G = (V, E)$. (*Hint:* Assuming that $V = \{1, 2, \ldots, n\}$, construct a flow network based on the graph $G' = (V', E')$, where

$$V' = \{x_0, x_1, \ldots, x_n\} \cup \{y_0, y_1, \ldots, y_n\} ,$$
$$E' = \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\} ,$$

and run a maximum-flow algorithm.)

b. Does your algorithm work for directed graphs that contain cycles? Explain.

24-3 Hiring consulting experts

Professor Fieri wants to open a consulting company for the food industry. He has identified n important food categories, which he represents by the set $C = \{C_1, C_2, \ldots, C_n\}$. In each category C_k, he can hire an expert in that category for $e_k > 0$ dollars. The consulting company has lined up a set $J = \{J_1, J_2, \ldots, J_m\}$ of potential jobs. In order to perform job J_i, the company needs to have hired experts in a subset $R_i \subseteq C$ of categories. Each expert can work on multiple jobs simultaneously. If the company chooses to accept job J_i, it must have hired experts in all categories in R_i, and it takes in revenue of $p_i > 0$ dollars.

Professor Fieri's job is to determine which categories to hire experts in and which jobs to accept in order to maximize the net revenue, which is the total income from jobs accepted minus the total cost of employing the experts.

Consider the following flow network G. It contains a source vertex s, vertices $C_1, C_2, \ldots, C_n$, vertices $J_1, J_2, \ldots, J_m$, and a sink vertex t. For $k = 1, 2 \ldots, n$, the flow network contains an edge (s, C_k) with capacity $c(s, C_k) = e_k$, and for $i = 1, 2, \ldots, m$, the flow network contains an edge (J_i, t) with capacity

$c(J_i, t) = p_i$. For $k = 1, 2, \ldots, n$ and $i = 1, 2, \ldots, m$, if $C_k \in R_i$, then G contains an edge (C_k, J_i) with capacity $c(C_k, J_i) = \infty$.

a. Show that if $J_i \in T$ for a finite-capacity cut (S, T) of G, then $C_k \in T$ for each $C_k \in R_i$.

b. Show how to determine the maximum net revenue from the capacity of a minimum cut of G and the given p_i values.

c. Give an efficient algorithm to determine which jobs to accept and which experts to hire. Analyze the running time of your algorithm in terms of m, n, and $r = \sum_{i=1}^{m} |R_i|$.

24-4 Updating maximum flow

Let $G = (V, E)$ be a flow network with source s, sink t, and integer capacities. Suppose that you are given a maximum flow in G.

a. Suppose that the capacity of a single edge $(u, v) \in E$ increases by 1. Give an $O(V + E)$-time algorithm to update the maximum flow.

b. Suppose that the capacity of a single edge $(u, v) \in E$ decreases by 1. Give an $O(V + E)$-time algorithm to update the maximum flow.

24-5 Maximum flow by scaling

Let $G = (V, E)$ be a flow network with source s, sink t, and an integer capacity $c(u, v)$ on each edge $(u, v) \in E$. Let $C = \max\{c(u, v) : (u, v) \in E\}$.

a. Argue that a minimum cut of G has capacity at most $C\,|E|$.

b. For a given number K, show how to find an augmenting path of capacity at least K in $O(E)$ time, if such a path exists.

The procedure MAX-FLOW-BY-SCALING appearing on the following page modifies the basic FORD-FULKERSON-METHOD procedure to compute a maximum flow in G.

c. Argue that MAX-FLOW-BY-SCALING returns a maximum flow.

d. Show that the capacity of a minimum cut of the residual network G_f is less than $2K\,|E|$ each time line 4 executes.

e. Argue that the inner **while** loop of lines 5–6 executes $O(E)$ times for each value of K.

```
MAX-FLOW-BY-SCALING(G, s, t)
C = max {c(u, v) : (u, v) ∈ E}
initialize flow f to 0
K = 2^⌊lg C⌋
while K ≥ 1
    while there exists an augmenting path p of capacity at least K
        augment flow f along p
    K = K/2
return f
```

f. Conclude that MAX-FLOW-BY-SCALING can be implemented so that it runs in $O(E^2 \lg C)$ time.

24-6 Widest augmenting path

The Edmonds-Karp algorithm implements the Ford-Fulkerson algorithm by always choosing a shortest augmenting path in the residual network. Suppose instead that the Ford-Fulkerson algorithm chooses a ***widest augmenting path***: an augmenting path with the greatest residual capacity. Assume that $G = (V, E)$ is a flow network with source s and sink t, that all capacities are integer, and that the largest capacity is C. In this problem, you will show that choosing a widest augmenting path results in at most $|E| \ln |f^*|$ augmentations to find a maximum flow f^*.

a. Show how to adjust Dijkstra's algorithm to find the widest augmenting path in the residual network.

b. Show that a maximum flow in G can be formed by successive flow augmentations along at most $|E|$ paths from s to t.

c. Given a flow f, argue that the residual network G_f has an augmenting path p with residual capacity $c_f(p) \geq (|f^*| - |f|)/|E|$.

d. Assuming that each augmenting path is a widest augmenting path, let f_i be the flow after augmenting the flow by the ith augmenting path, where f_0 has $f(u, v) = 0$ for all edges (u, v). Show that $|f^*| - |f_i| \leq |f^*|(1 - 1/|E|)^i$.

e. Show that $|f^*| - |f_i| < |f^*| e^{-i/|E|}$.

f. Conclude that after the flow is augmented at most $|E| \ln |f^*|$ times, the flow is a maximum flow.

24-7 Global minimum cut

A ***global cut*** in an undirected graph $G = (V, E)$ is a partition (see page 1156) of V into two nonempty sets V_1 and V_2. This definition is like the definition of cut that we have used in this chapter, except that we no longer have distinguished vertices s and t. Any edge (u, v) with $u \in V_1$ and $v \in V_2$ is said to ***cross*** the cut.

We can extend this definition of a cut to a multigraph $G = (V, E)$ (see page 1167), and we denote by $c(u, v)$ the number of edges in the multigraph with endpoints u and v. A global cut in a multigraph is still a partition of the vertices, and the value of a global cut (V_1, V_2) is $c(V_1, V_2) = \sum_{u \in V_1, v \in V_2} c(u, v)$. A solution to the ***global-minimum-cut problem*** is a cut (V_1, V_2) such that $c(V_1, V_2)$ is minimum. Let $\mu(G)$ denote the value of a global minimum cut in a graph or multigraph G.

a. Show how to find a global minimum cut of a graph $G = (V, E)$ by solving $\binom{|V|}{2}$ maximum-flow problems, each with a different pair of vertices as the source and sink, and taking the mininum value of the cuts found.

b. Give an algorithm to find a global minimum cut by solving only $\Theta(V)$ maximum-flow problems. What is the running time of your algorithm?

The remainder of this problem develops an algorithm for the global-minimum-cut problem that does not use any maximum-flow computations. It uses the notion of an edge contraction, defined on page 1168, with one crucial difference. The algorithm maintains a multigraph, so that upon contracting an edge (u, v), it creates a new vertex x, and for any other vertex $y \in V$, the number of edges between x and y is $c(u, y) + c(v, y)$. The algorithm does not maintain self-loops, and so it sets $c(x, x)$ to 0. Denote by $G/(u, v)$ the multigraph that results from contracting edge (u, v) in multigraph G.

Consider what can happen to the minimum cut when an edge is contracted. Assume that, at all points, the minimum cut in a multigraph G is unique. We'll remove this assumption later.

c. Show that for any edge (u, v), we have $\mu(G/(uv)) \leq \mu(G)$. Under what conditions is $\mu(G/(uv)) < \mu(G)$?

Next, you will show that if you pick an edge uniformly at random, the probability that it belongs to the minimum cut is small.

d. Show that for any multigraph $G = (V, E)$, the value of the global minimum cut is at most the average degree of a vertex: that $\mu(G) \leq 2|E|/|V|$, where $|E|$ denotes the total number of edges in the multigraph.

e. Using the results from parts (c) and (d), show that, if we pick an edge (u, v) uniformly at random, then the probability that (u, v) belongs to the minimum cut is at most $2/V$.

Consider the algorithm that repeatedly chooses an edge at random and contracts it until the multigraph has exactly two vertices, say u and v. At that point, the multigraph corresponds to a cut in the original graph, with vertex u representing all the nodes in one side of the original graph, and v representing all the vertices on the other side. The number of edges given by $c(u, v)$ corresponds exactly to the number of edges crossing the corresponding cut in the original graph. We call this algorithm the ***contraction algorithm***.

f. Suppose that the contraction algorithm terminates with a multigraph whose only vertices are u and v. Show that $\Pr\{c(u, v) = \mu(G)\} = \Omega\left(1/\binom{|V|}{2}\right)$.

g. Prove that if the contraction algorithm repeats $\binom{|V|}{2} \ln |V|$ times, then the probability that at least one of the runs returns the minimum cut is at least $1 - 1/|V|$.

h. Give a detailed implementation of the contraction algorithm that runs in $O(V^2)$ time.

i. Combine the previous parts and remove the assumption that the minimum cut must be unique, to conclude that running the contraction algorithm $\binom{|V|}{2} \ln |V|$ times yields an algorithm that runs in $O(V^4 \lg V)$ time and returns a minimum cut with probability at least $1 - 1/V$.

Chapter notes

Ahuja, Magnanti, and Orlin [7], Even [137], Lawler [276], Papadimitriou and Steiglitz [353], Tarjan [429], and Williamson [458] are good references for network flows and related algorithms. Schrijver [399] has written an interesting review of historical developments in the field of network flows.

The Ford-Fulkerson method is due to Ford and Fulkerson [149], who originated the formal study of many of the problems in the area of network flow, including the maximum-flow and bipartite-matching problems. Many early implementations of the Ford-Fulkerson method found augmenting paths using breadth-first search. Edmonds and Karp [132], and independently Dinic [119], proved that this strategy yields a polynomial-time algorithm. A related idea, that of using "blocking flows," was also first developed by Dinic [119].

A class of algorithms known as ***push-relabel algorithms***, due to Goldberg [185] and Goldberg and Tarjan [188], takes a different approach from the Ford-Fulkerson

method. Push-relabel algorithms allow flow conservation to be violated at vertices other than the source and sink as they execute. Using an idea first developed by Karzonov [251], they allow a ***preflow*** in which the flow into a vertex may exceed the flow out of the vertex. Such a vertex is said to be ***overflowing***. Initially, every edge leaving the source is filled to capacity, so that all neighbors of the source are overflowing. In a push-relabel algorithm, each vertex is assigned an integer height. An overflowing vertex may push flow to a neighboring vertex to which it has a residual edge provided that it is higher than the neighbor. If all residual edges from an overflowing vertex go to neighbors with equal or greater heights, then the vertex may increase its height. Once all vertices other than the sink are no longer overflowing, the preflow is not only a legal flow, but also a maximum flow.

Goldberg and Tarjan [188] gave an $O(V^3)$-time algorithm that uses a queue to maintain the set of overflowing vertices, as well as an algorithm that uses dynamic trees to achieve a running time of $O(VE \lg(V^2/E + 2))$. Several other researchers developed improved variants and implementations [9, 10, 15, 86, 87, 255, 358], the fastest of which, by King, Rao, and Tarjan [255], runs in $O(VE \log_{E/(V \lg V)} V)$ time.

Another efficient algorithm for maximum flow, by Goldberg and Rao [187], runs in $O\left(\min\left\{V^{2/3}, E^{1/2}\right\} E \lg(V^2/E + 2) \lg C\right)$ time, where C is the maximum capacity of any edge. Orlin [350] gave an algorithm in the same spirit as this algorithm that runs in $O(VE + E^{31/16} \lg^2 V)$ time. Combining it with the algorithm of King, Rao, and Tarjan results in an $O(VE)$-time algorithm.

A different approach to maximum flows and related problems is to use techniques from continuous optimization including electrical flows and interior-point methods. The first breakthrough in this line of work is due to Madry [308], who gave an $\widetilde{O}(E^{10/7})$-time algorithm for unit-capacity maximum flow and bipartite maximum matching. (See Problem 3-6 on page 73 for a definition of $\widetilde{O}$.) There has been a series of papers in this area for matchings, maximum flows, and minimum-cost flows. The fastest algorithm to date in this line of work for maximum flow is due to Lee and Sidford [285], taking $\widetilde{O}(\sqrt{V} E \lg^{O(1)} C)$ time. If the capacities are not too large, this algorithm is faster than the $O(VE)$-time algorithm mentioned above. Another algorithm, due to Liu and Sidford [303] runs in $\widetilde{O}(E^{11/8} C^{1/4})$ time, where C is the maximum capacity of any edge. This algorithm does not run in polynomial time, but for small enough capacities, it is faster than the previous ones.

In practice, push-relabel algorithms currently dominate algorithms based on augmenting paths, continuous-optimization, and linear programming for the maximum-flow problem [88] .

25 Matchings in Bipartite Graphs

Many real-world problems can be modeled as finding matchings in an undirected graph. For an undirected graph $G = (V, E)$, a ***matching*** is a subset of edges $M \subseteq E$ such that every vertex in V has at most one incident edge in M.

For example, consider the following scenario. You have one or more positions to fill and several candidates to interview. According to your schedule, you are able to interview candidates at certain time slots. You ask the candidates to indicate the subsets of time slots at which they are available. How can you schedule the interviews so that each time slot has at most one candidate scheduled, while maximizing the number of candidates that you can interview? You can model this scenario as a matching problem on a bipartite graph in which each vertex represents either a candidate or a time slot, with an edge between a candidate and a time slot if the candidate is available then. If an edge is included in the matching, that means you are scheduling a particular candidate for a particular time slot. Your goal is to find a ***maximum matching***: a matching of maximum cardinality. One of the authors of this book was faced with exactly this situation when hiring teaching assistants for a large class. He used the Hopcroft-Karp algorithm in Section 25.1 to schedule the interviews.

Another application of matching is the U.S. National Resident Matching Program, in which medical students are matched to hospitals where they will be stationed as medical residents. Each student ranks the hospitals by preference, and each hospital ranks the students. The goal is to assign students to hospitals so that there is never a student and a hospital that both have regrets because the student was not assigned to the hospital, yet each ranked the other higher than who or where they were assigned. This scenario is perhaps the best-known real-world example of the "stable-marriage problem," which Section 25.2 examines.

Yet another instance where matching comes into play occurs when workers must be assigned to tasks in order to maximize the overall effectiveness of the assignment. For each worker and each task, the worker has some quantified effectiveness

for that task. Assuming that there are equal numbers of workers and tasks, the goal is to find a matching with the maximum total effectiveness. Such a situation is an example of an assignment problem, which Section 25.3 shows how to solve.

The algorithms in this chapter find matchings in ***bipartite*** graphs. As in Section 24.3, the input is an undirected graph $G = (V, E)$, where $V = L \cup R$, the vertex sets L and R are disjoint, and every edge in E is incident on one vertex in L and one vertex in R. A matching, therefore, matches vertices in L with vertices in R. In some applications, the sets L and R have equal cardinality, and in other applications they need not be the same size.

An undirected graph need not be bipartite for the concept of matching to apply. Matching in general undirected graphs has applications in areas such as scheduling and computational chemistry. It models problems in which you want to pair up entities, represented by vertices. Two vertices are adjacent if they represent compatible entities, and you need to find a large set of compatible pairs. Maximum-matching and maximum-weight matching problems on general graphs can be solved by polynomial-time algorithms whose running times are similar to those for bipartite matching, but the algorithms are significantly more complicated. Exercise 25.2-5 discusses the general version of the stable-marriage problem, known as the "stable-roommates problem." Although matching applies to general undirected graphs, this chapter deals only with bipartite graphs.

25.1 Maximum bipartite matching (revisited)

Section 24.3 demonstrated one way to find a maximum matching in a bipartite graph, by finding a maximum flow. This section provides a more efficient method, the Hopcroft-Karp algorithm, which runs in $O(\sqrt{V}E)$ time. Figure 25.1(a) shows a matching in an undirected bipartite graph. A vertex that has an incident edge in matching M is ***matched*** under M, and otherwise, it is ***unmatched***. A ***maximal matching*** is a matching M to which no other edges can be added, that is, for every edge $e \in E - M$, the edge set $M \cup \{e\}$ fails to be a matching. A maximum matching is always maximal, but the reverse does not always hold.

Many algorithms to find maximum matchings, the Hopcroft-Karp algorithm included, work by incrementally increasing the size of a matching. Given a matching M in an undirected graph $G = (V, E)$, an ***M-alternating path*** is a simple path whose edges alternate between being in M and being in $E - M$. Figure 25.1(b) depicts an ***M-augmenting path*** (sometimes called an augmenting path with respect to M): an M-alternating path whose first and last edges belong to $E - M$. Since an M-augmenting path contains one more edge in $E - M$ than in M, it must consist of an odd number of edges.

Figure 25.1 A bipartite graph, where $V = L \cup R$, $L = \{l_1, l_2, \ldots, l_7\}$, and $R = \{r_1, r_2, \ldots, r_8\}$. **(a)** A matching M with cardinality 4, highlighted in blue. Matched vertices are blue, and unmatched vertices are tan. **(b)** The five edges highlighted in orange form an M-augmenting path P going between vertices l_6 and r_8. **(c)** The set of edges $M' = M \oplus P$ highlighted in blue is a matching containing one more edge than M and adding l_6 and r_8 to the matched vertices. This matching is not a maximum matching (see Exercise 25.1-1).

Figure 25.1(c) demonstrates the following lemma, which shows that by removing from matching M the edges in an M-augmenting path that belong to M and adding to M the edges in the M-augmenting path that are not in M, the result is a new matching with one more edge than M. Since a matching is a set of edges, the lemma relies on the notion of the ***symmetric difference*** of two sets: $X \oplus Y = (X - Y) \cup (Y - X)$, that is, the elements that belong to X or Y, but not both. Alternatively, you can think of $X \oplus Y$ as $(X \cup Y) - (X \cap Y)$. The operator $\oplus$ is commutative and associative. Furthermore, $X \oplus X = \emptyset$ and $X \oplus \emptyset = \emptyset \oplus X = X$ for any set X, so that the empty set is the identity for $\oplus$.

Lemma 25.1

Let M be a matching in any undirected graph $G = (V, E)$, and let P be an M-augmenting path. Then the set of edges $M' = M \oplus P$ is also a matching in G with $|M'| = |M| + 1$.

Proof Let P contain q edges, so that $\lceil q/2 \rceil$ edges belong to $E - M$ and $\lfloor q/2 \rfloor$ edges belong to M, and let these q edges be $(v_1, v_2), (v_2, v_3), \ldots, (v_q, v_{q+1})$. Because P is an M-augmenting path, vertices v_1 and v_{q+1} are unmatched under M and all other vertices in P are matched. Edges $(v_1, v_2), (v_3, v_4), \ldots, (v_q, v_{q+1})$ belong to $E - M$, and edges $(v_2, v_3), (v_4, v_5), \ldots, (v_{q-1}, v_q)$ belong to M. The symmetric difference $M' = M \oplus P$ reverses these roles, so that edges $(v_1, v_2), (v_3, v_4), \ldots, (v_q, v_{q+1})$ belong to M' and $(v_2, v_3), (v_4, v_5), \ldots, (v_{q-1}, v_q)$ belong to $E - M'$. Each vertex $v_1, v_2, \ldots, v_q, v_{q+1}$ is matched under M', which gains one additional edge relative to M, and no other vertices or edges in G are affected by the change from M to M'. Hence, M' is a matching in G, and $|M'| = |M| + 1$. ■

Since taking the symmetric difference of a matching M with an M-augmenting path increases the size of the matching by 1, the following corollary shows that taking the symmetric difference of M with k vertex-disjoint M-augmenting paths increases the size of the matching by k.

Corollary 25.2
Let M be a matching in any undirected graph $G = (V, E)$ and $P_1, P_2, \ldots, P_k$ be vertex-disjoint M-augmenting paths. Then the set of edges $M' = M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$ is a matching in G with $|M'| = |M| + k$.

Proof Since the M-augmenting paths $P_1, P_2, \ldots, P_k$ are vertex-disjoint, we have that $P_1 \cup P_2 \cup \cdots \cup P_k = P_1 \oplus P_2 \oplus \cdots \oplus P_k$. Because the operator $\oplus$ is associative, we have

$$\begin{aligned} M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k) &= M \oplus (P_1 \oplus P_2 \oplus \cdots \oplus P_k) \\ &= (\cdots((M \oplus P_1) \oplus P_2) \oplus \cdots \oplus P_{k-1}) \oplus P_k \ . \end{aligned}$$

A simple induction on i using Lemma 25.1 shows that $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_{i-1})$ is a matching in G containing $|M| + i - 1$ edges and that path P_i is an augmenting path with respect to $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_{i-1})$. Each of these augmenting paths increases the size of the matching by 1, and so $|M'| = |M| + k$. ■

As the Hopcroft-Karp algorithm goes from matching to matching, it will be useful to consider the symmetric difference between two matchings.

Lemma 25.3
Let M and M^* be matchings in graph $G = (V, E)$, and consider the graph $G' = (V, E')$, where $E' = M \oplus M^*$. Then, G' is a disjoint union of simple paths, simple cycles, and/or isolated vertices. The edges in each such simple path or simple cycle

alternate between M and M^*. If $|M^*| > |M|$, then G' contains at least $|M^*| - |M|$ vertex-disjoint M-augmenting paths.

Proof Each vertex in G' has degree 0, 1, or 2, since at most two edges of E' can be incident on a vertex: at most one edge from M and at most one edge from M^*. Therefore, each connected component of G' is either a singleton vertex, an even-length simple cycle with edges alternately in M and M^*, or a simple path with edges alternately in M and M^*. Since

$$\begin{aligned} E' &= M \oplus M^* \\ &= (M \cup M^*) - (M \cap M^*) \end{aligned}$$

and $|M^*| > |M|$, the edge set E' must contain $|M^*| - |M|$ more edges from M^* than from M. Because each cycle in G' has an even number of edges drawn alternately from M and M^*, each cycle has an equal number of edges from M and M^*. Therefore, the simple paths in G' account for there being $|M^*| - |M|$ more edges from M^* than M. Each path containing a different number of edges from M and M^* either starts and ends with edges from M, containing one more edge from M than from M^*, or starts and ends with edges from M^*, containing one more edge from M^* than from M. Because E' contains $|M^*| - |M|$ more edges from M^* than from M, there are at least $|M^*| - |M|$ paths of the latter type, and each one is an M-augmenting path. Because each vertex has at most two incident edges from E', these paths must be vertex-disjoint. ■

If an algorithm finds a maximum matching by incrementally increasing the size of the matching, how does it determine when to stop? The following corollary gives the answer: when there are no augmenting paths.

Corollary 25.4
Matching M in graph $G = (V, E)$ is a maximum matching if and only if G contains no M-augmenting path.

Proof We prove the contrapositive of both directions of the lemma statement. The contrapositive of the forward direction is straightforward. If there is an M-augmenting path P in G, then by Lemma 25.1, the matching $M \oplus P$ contains one more edge than M, meaning that M could not be a maximum matching.

To show the contrapositive of the backward direction—if M is not a maximum matching, then G contains an M-augmenting path—let M^* be a maximum matching in Lemma 25.3, so that $|M^*| > |M|$. Then G contains at least $|M^*| - |M| > 0$ vertex-disjoint M-augmenting paths. ■

We already have learned enough to create a maximum-matching algorithm that runs in $O(VE)$ time. Start with the matching M empty. Then repeatedly run a variant of either breadth-first search or depth-first search from an unmatched vertex that takes alternating paths until you find another unmatched vertex. Use the resulting M-augmenting path to increase the size of M by 1.

The Hopcroft-Karp algorithm

The Hopcroft-Karp algorithm improves the running time to $O(\sqrt{V}E)$. The procedure HOPCROFT-KARP is given an undirected bipartite graph, and it uses Corollary 25.2 to repeatedly increase the size of the matching M it finds. Corollary 25.4 proves that the algorithm is correct, since it terminates once there are no M-augmenting paths. It remains to show that the algorithm does run in $O(\sqrt{V}E)$ time. We'll see that the **repeat** loop of lines 2–5 iterates $O(\sqrt{V})$ times and how to implement line 3 so that it runs in $O(E)$ time in each iteration.

```
HOPCROFT-KARP(G)
1  M = ∅
2  repeat
3      let 𝒫 = {P_1, P_2, ..., P_k} be a maximal set of vertex-disjoint
           shortest M-augmenting paths
4      M = M ⊕ (P_1 ∪ P_2 ∪ ··· ∪ P_k)
5  until 𝒫 == ∅
6  return M
```

Let's first see how to find a maximal set of vertex-disjoint shortest M-augmenting paths in $O(E)$ time. There are three phases. The first phase forms a directed version G_M of the undirected bipartite graph G. The second phase creates a directed acyclic graph H from G_M via a variant of breadth-first search. The third phase finds a maximal set of vertex-disjoint shortest M-augmenting paths by running a variant of depth-first search on the transpose H^{T} of H. (Recall that the transpose of a directed graph reverses the direction of each edge. Since H is acyclic, so is H^{T}.)

Given a matching M, you can think of an M-augmenting path P as starting at an unmatched vertex in L, traversing an odd number of edges, and ending at an unmatched vertex in R. The edges in P traversed from L to R must belong to $E - M$, and the edges in P traversed from R to L must belong to M. The first phase, therefore, creates the directed graph G_M by directing the edges accordingly: $G_M = (V, E_M)$, where

Figure 25.2 **(a)** The directed graph G_M created in the first phase for the undirected bipartite graph G and matching M in Figure 25.1(a). Breadth-first distances from any unmatched vertex in L appear next to each vertex. **(b)** The dag H created from G_M in the second phase. Because the smallest distance to an unmatched vertex in R is 3, vertices l_7 and r_8, with distances greater than 3, are not in H.

$$\begin{aligned} E_M = {} & \{(l,r) : l \in L, r \in R, \text{ and } (l,r) \in E - M\} && \text{(edges from } L \text{ to } R) \\ & \cup \{(r,l) : r \in R, l \in L, \text{ and } (l,r) \in M\} && \text{(edges from } R \text{ to } L) . \end{aligned}$$

Figure 25.2(a) shows the graph G_M for the graph G and matching M in Figure 25.1(a).

The dag $H = (V_H, E_H)$ created by the second phase has layers of vertices. Figure 25.2(b) shows the dag H corresponding to the directed graph G_M in part (a) of the figure. Each layer contains only vertices from L or only vertices from R, alternating from layer to layer. The layer that a vertex resides in is given by that vertex's minimum breadth-first distance in G_M from any unmatched vertex in L. Vertices in L appear in even-numbered layers, and vertices in R appear in odd-numbered layers. Let q denote the smallest distance in G_M of any unmatched vertex in R. Then, the last layer in H contains the vertices in R with distance q. Vertices whose distance exceeds q do not appear in V_H. (The graph H in Figure 25.2(b) omits vertices l_7 and r_8 because their distances from any unmatched vertex in L exceed $q = 3$.) The edges in E_H form a subset of E_M:

$$E_H = \{(l, r) \in E_M : r.d \le q \text{ and } r.d = l.d + 1\} \cup \{(r, l) \in E_M : l.d \le q\} ,$$

where the attribute d of a vertex gives the vertex's breadth-first distance in G_M from any unmatched vertex in L. Edges that do not go between two consecutive layers are omitted from E_H.

To determine the breadth-first distances of vertices, run breadth-first search on the graph G_M, but starting from all the unmatched vertices in L. (In the BFS procedure on page 556, replace the root vertex s by the set of unmatched vertices in L.) The predecessor attributes π computed by the BFS procedure are not needed here, since H is a dag and not necessarily a tree.

Every path in H from a vertex in layer 0 to an unmatched vertex in layer q corresponds to a shortest M-augmenting path in the original bipartite graph G. Just use the undirected versions of the directed edges in H. Moreover, every shortest M-augmenting path in G is present in H.

The third phase identifies a maximal set of vertex-disjoint shortest M-augmenting paths. As Figure 25.3 shows, it starts by creating the transpose H^{T} of H. Then, for each unmatched vertex r in layer q, it performs a depth-first search starting from r until it either reaches a vertex in layer 0 or has exhausted all possible paths without reaching a vertex in layer 0. Instead of maintaining discovery and finish times, the depth-first search just needs to keep track of the predecessor attributes π in the depth-first tree of each search. Upon reaching a vertex in layer 0, tracing back along the predecessors identifies an M-augmenting path. Each vertex is searched from only when it is first discovered in any search. If the search from a vertex r in layer q cannot find a path of undiscovered vertices to an undiscovered vertex in layer 0, then no M-augmenting path including r goes into the maximal set.

Figure 25.3 shows the result of the third phase. The first depth-first search starts from vertex r_1. It identifies the M-augmenting path $\langle (r_1, l_3), (l_3, r_3), (r_3, l_1) \rangle$, which is highlighted in orange, and discovers vertices r_1, l_3, r_3, and l_1. The next depth-first search starts from vertex r_4. This search first examines the edge (r_4, l_3), but because l_3 was already discovered, it backtracks and examines edge (r_4, l_5). From there, it continues and identifies the M-augmenting path $\langle (r_4, l_5), (l_5, r_7), (r_7, l_6) \rangle$, which is highlighted in yellow, and discovers vertices r_4, l_5, r_7, and l_6. The depth-first search from vertex r_6 gets stuck at vertices l_3 and l_5, which have already been discovered, and so this search fails to find a path of undiscovered vertices to a vertex in layer 0. There is no depth-first search from vertex r_5 because it is matched, and depth-first searches start from unmatched vertices. Therefore, the maximal set of vertex-disjoint shortest M-augmenting paths found contains just the two M-augmenting paths $(\langle (r_1, l_3), (l_3, r_3), (r_3, l_1) \rangle$ and $\langle (r_4, l_5), (l_5, r_7), (r_7, l_6) \rangle$.

You might have noticed that in this example, this maximal set of two vertex-disjoint shortest M-augmenting paths is not a maximum set. The graph contains three vertex-disjoint shortest M-augmenting paths: $\langle (r_1, l_2), (l_2, r_2), (r_2, l_1) \rangle$, $\langle (r_4, l_3), (l_3, r_3), (r_3, l_4) \rangle$, and $\langle (r_6, l_5), (l_5, r_7), (r_7, l_6) \rangle$. No matter: the algorithm

Figure 25.3 The transpose H^{T} of the dag H created in the third phase. The first depth-first search, starting from vertex r_1, identifies the M-augmenting path $\langle (r_1, l_3), (l_3, r_3), (r_3, l_1) \rangle$ highlighted in orange, and it discovers vertices r_1, l_3, r_3, l_1. The second depth-first search, starting from vertex r_4, identifies the M-augmenting path $\langle (r_4, l_5), (l_5, r_7), (r_7, l_6) \rangle$ highlighted in yellow, discovering vertices r_4, l_5, r_7, l_6.

requires the set of vertex-disjoint shortest M-augmenting paths found in line 3 of HOPCROFT-KARP to be only maximal, not necessarily maximum.

It remains to show that all three phases of line 3 take $O(E)$ time. We assume that in the original bipartite graph G, each vertex has at least one incident edge so that $|V| = O(E)$, which in turn implies that $|V| + |E| = O(E)$. The first phase creates the directed graph G_M by simply directing each edge of G, so that $|V_M| = |V|$ and $|E_M| = |E|$. The second phase performs a breadth-first search on G_M, taking $O(V_M + E_M) = O(E_M) = O(E)$ time. In fact, it can stop once the first distance in the queue within the breadth-first search exceeds the shortest distance q to an unmatched vertex in R. The dag H has $|V_H| \le |V_M|$ and $|E_H| \le |E_M|$, so that it takes $O(V_H + E_H) = O(E)$ time to construct. Finally, the third phase performs depth-first searches from the unmatched vertices in layer q. Once a vertex is discovered, it is not searched from again, and so the analysis of depth-first search from Section 20.3 applies here: $O(V_H + E_H) = O(E)$. Hence, all three phases take just $O(E)$ time.

Once the maximal set of vertex-disjoint shortest M-augmenting paths have been found in line 3, updating the matching in line 4 takes $O(E)$ time, as it is just a matter of going through the edges of the M-augmenting paths and adding edges to and removing edges from the matching M. Thus, each iteration of the **repeat** loop of lines 2–5 can run in $O(E)$ time.

It remains to show that the **repeat** loop iterates $O(\sqrt{V})$ times. We start with the following lemma, which shows that after each iteration of the **repeat** loop, the length of an augmenting path increases.

Lemma 25.5
Let $G = (V, E)$ be an undirected bipartite graph with matching M, and let q be the length of a shortest M-augmenting path. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ be a maximal set of vertex-disjoint M-augmenting paths of length q. Let $M' = M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$, and suppose that P is a shortest M'-augmenting path. Then P has more than q edges.

Proof We consider separately the cases in which P is vertex-disjoint from the augmenting paths in $\mathcal{P}$ and in which it is not vertex-disjoint.

First, assume that P is vertex-disjoint from the augmenting paths in $\mathcal{P}$. Then, P contains edges that are in M but are not in any of $P_1, P_2, \ldots, P_k$, so that P is also an M-augmenting path. Since P is disjoint from $P_1, P_2, \ldots, P_k$ but is also an M-augmenting path, and since $\mathcal{P}$ is a maximal set of shortest M-augmenting paths, P must be longer than any of the augmenting paths in $\mathcal{P}$, each of which has length q. Therefore, P has more than q edges.

Now, assume that P visits at least one vertex from the M-augmenting paths in $\mathcal{P}$. By Corollary 25.2, M' is a matching in G with $|M'| = |M| + k$. Since P is an M'-augmenting path, by Lemma 25.1, $M' \oplus P$ is a matching with $|M' \oplus P| = |M'| + 1 = |M| + k + 1$. Now let $A = M \oplus M' \oplus P$. We claim that $A = (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P$:

$$\begin{aligned}
A &= M \oplus M' \oplus P \\
&= M \oplus (M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)) \oplus P \\
&= (M \oplus M) \oplus (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P && \text{(associativity of } \oplus) \\
&= \emptyset \oplus (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P && (X \oplus X = \emptyset \text{ for all } X) \\
&= (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P && (\emptyset \oplus X = X \text{ for all } X)\,.
\end{aligned}$$

Lemma 25.3 with $M^* = M' \oplus P$ gives that A contains at least $|M' \oplus P| - |M| = k + 1$ vertex-disjoint M-augmenting paths. Since each such M-augmenting path has at least q edges, we have $|A| \geq (k + 1)q = kq + q$.

Now we claim that P shares at least one edge with some M-augmenting path in $\mathcal{P}$. Under the matching M', every vertex in each M-augmenting path in $\mathcal{P}$ is matched. (Only the first and last vertex in each M-augmenting path P_i is unmatched under M, and under $M \oplus P_i$, all vertices in P_i are matched. Because the M-augmenting paths in $\mathcal{P}$ are vertex-disjoint, no other path in $\mathcal{P}$ can affect whether the vertices in P_i are matched. That is, the vertices in P_i are matched under $(M \oplus P_i) \oplus P_j$ if and only if they are matched under $M \oplus P_i$, for any other

path $P_j \in \mathcal{P}$.) Suppose that P shares a vertex v with some path $P_i \in \mathcal{P}$. Vertex v cannot be an endpoint of P, because the endpoints of P are unmatched under M'. Therefore, v has an incident edge in P that belongs to M'. Since any vertex has at most one incident edge in a matching, this edge must also belong to P_i, thus proving the claim.

Because $A = (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P$ and P shares at least one edge with some $P_i \in \mathcal{P}$, we have that $|A| < |P_1 \cup P_2 \cup \cdots \cup P_k| + |P|$. Thus, we have

$$\begin{aligned} kq + q &\le |A| \\ &< |P_1 \cup P_2 \cup \cdots \cup P_k| + |P| \\ &= kq + |P| \ , \end{aligned}$$

so that $q < |P|$. We conclude that P contains more than q edges. ■

The next lemma bounds the size of a maximum matching, based on the length of a shortest augmenting path.

Lemma 25.6
Let M be a matching in graph $G = (V, E)$, and let a shortest M-augmenting path in G contain q edges. Then the size of a maximum matching in G is at most $|M| + |V|/(q + 1)$.

Proof Let M^* be a maximum matching in G. By Lemma 25.3, G contains at least $|M^*|-|M|$ vertex-disjoint M-augmenting paths. Each of these paths contains at least q edges, and hence at least $q + 1$ vertices. Because these paths are vertex-disjoint, we have $(|M^*|-|M|)(q+1) \le |V|$, so that $|M^*| \le |M|+|V|/(q+1)$. ■

The final lemma bounds the number of iterations of the **repeat** loop of lines 2–5.

Lemma 25.7
When the HOPCROFT-KARP procedure runs on an undirected bipartite graph $G = (V, E)$, the **repeat** loop of lines 2–5 iterates $O(\sqrt{V})$ times.

Proof By Lemma 25.5, the length q of the shortest M-augmenting paths found in line 3 increases from iteration to iteration. After $\lceil \sqrt{|V|} \rceil$ iterations, therefore, we must have $q \ge \lceil \sqrt{|V|} \rceil$. Consider the situation after the first time line 4 executes with M-augmenting paths whose length is at least $\lceil \sqrt{|V|} \rceil$. Since the size of a matching increases by at least one edge per iteration, Lemma 25.6 implies that the number of additional iterations before achieving a maximum matching is at most

$$\frac{|V|}{\lceil \sqrt{|V|} \rceil + 1} < \frac{|V|}{\sqrt{|V|}} = \sqrt{|V|} \ .$$

Hence, the total number of loop iterations is less than $2\sqrt{|V|}$. ■

Thus, we have the following bound on the running time of the HOPCROFT-KARP procedure.

Theorem 25.8
The procedure HOPCROFT-KARP runs in $O(\sqrt{V}E)$ time on an undirected bipartite graph $G = (V, E)$.

Proof By Lemma 25.7 the **repeat** loop iterates $O(\sqrt{V})$ times, and we have seen how to implement each iteration in $O(E)$ time. ■

Exercises

25.1-1
Use the Hopcroft-Karp algorithm to find a maximum matching for the graph in Figure 25.1.

25.1-2
How are M-augmenting paths and augmenting paths in flow networks similar? How do they differ?

25.1-3
What is the advantage of searching in the transpose H^{T} from unmatched vertices in layer q (the first layer that contains an unmatched vertex in R) to layer 0 versus searching in the dag H from layer 0 to layer q?

25.1-4
Show how to bound the number of iterations of the the **repeat** loop of lines 2–5 of HOPCROFT-KARP by $\lceil 3\sqrt{|V|}/2 \rceil$.

★ ***25.1-5***
A ***perfect matching*** is a matching under which every vertex is matched. Let $G = (V, E)$ be an undirected bipartite graph with vertex partition $V = L \cup R$, where $|L| = |R|$. For any $X \subseteq V$, define the ***neighborhood*** of X as

$$N(X) = \{y \in V : (x, y) \in E \text{ for some } x \in X\} ,$$

that is, the set of vertices adjacent to some member of X. Prove ***Hall's theorem***: there exists a perfect matching in G if and only if $|A| \le |N(A)|$ for every subset $A \subseteq L$.

25.1-6

In a ***d*-regular** graph, every vertex has degree d. If $G = (V, E)$ is bipartite with vertex partition $V = L \cup R$ and also d-regular, then $|L| = |R|$. Use Hall's theorem (see Exercise 25.1-5) to prove that every d-regular bipartite graph contains a perfect matching. Then use that result to prove that every d-regular bipartite graph contains d disjoint perfect matchings.

25.2 The stable-marriage problem

In Section 25.1, the goal was to find a maximum matching in an undirected bipartite graph. If you know that the graph $G = (V, E)$ with vertex partition $V = L \cup R$ is a ***complete bipartite graph***[1]—containing an edge from every vertex in L to every vertex in R—then you can find a maximum matching by a simple greedy algorithm.

When a graph can have several matchings, you might want to decide which matchings are most desirable. In Section 25.3, we'll add weights to the edges and find a matching of maximum weight. In this section, we will instead add some information to each vertex in a complete bipartite graph: a ranking of the vertices in the other side. That is, each vertex in L has an ordered list of all the vertices in R, and vice-versa. To keep things simple, let's assume that L and R each contain n vertices. The goal here is to match each vertex in L with a vertex in R in a "stable" way.

This problem derives its name, the ***stable-marriage problem***, from the notion of heterosexual marriage, viewing L as a set of women and R as a set of men.[2] Each woman ranks all the men in terms of desirability, and each man does the same with all the women. The goal is to pair up women and men (a matching) so that if a woman and a man are not matched to each other, then at least one of them prefers their assigned partner.

If a woman and a man are not matched to each other but each prefers the other over their assigned partner, they form a ***blocking pair***. A blocking pair has incentive to opt out of the assigned pairing and get together on their own. If that were to occur, then this pair would block the matching from being "stable." A ***stable***

[1] The definition of a complete bipartite graph differs from the definition of complete graph given on page 1167 because in a bipartite graph, there are no edges between vertices in L and no edges between vertices in R.

[2] Although marriage norms are changing, it's traditional to view the stable-marriage problem through the lens of heterosexual marriage.

matching, therefore, is a matching that has no blocking pair. If there is a blocking pair, then the matching is ***unstable***.

Let's look at an example with four women—Wanda, Emma, Lacey, and Karen—and four men—Oscar, Davis, Brent, and Hank—having the following preferences:

Wanda: Brent, Hank, Oscar, Davis
Emma: Davis, Hank, Oscar, Brent
Lacey: Brent, Davis, Hank, Oscar
Karen: Brent, Hank, Davis, Oscar

Oscar: Wanda, Karen, Lacey, Emma
Davis: Wanda, Lacey, Karen, Emma
Brent: Lacey, Karen, Wanda, Emma
Hank: Lacey, Wanda, Emma, Karen

A stable matching comprises the following pairs:

Lacey and Brent
Wanda and Hank
Karen and Davis
Emma and Oscar

You can verify that this matching has no blocking pair. For example, even though Karen prefers Brent and Hank to her partner Davis, Brent prefers his partner Lacey to Karen, and Hank prefers his partner Wanda to Karen, so that neither Karen and Brent nor Karen and Hank form a blocking pair. In fact, this stable matching is unique. Suppose instead that the last two pairs were

Emma and Davis
Karen and Oscar

Then Karen and Davis would be a blocking pair, because they were not paired together, Karen prefers Davis to Oscar, and Davis prefers Karen to Emma. Therefore, this matching is not stable.

Stable matchings need not be unique. For example, suppose that there are three women—Monica, Phoebe, and Rachel—and three men—Chandler, Joey, and Ross—with these preferences:

Monica: Chandler, Joey, Ross
Phoebe: Joey, Ross, Chandler
Rachel: Ross, Chandler, Joey

Chandler: Phoebe, Rachel, Monica
Joey: Rachel, Monica, Phoebe
Ross: Monica, Phoebe, Rachel

In this case, there are three stable matchings:

Matching 1	Matching 2	Matching 3
Monica and Chandler	Phoebe and Chandler	Rachel and Chandler
Phoebe and Joey	Rachel and Joey	Monica and Joey
Rachel and Ross	Monica and Ross	Phoebe and Ross

In matching 1, all women get their first choice and all men get their last choice. Matching 2 is the opposite, with all men getting their first choice and all women getting their last choice. When all the women or all the men get their first choice, there plainly cannot be a blocking pair. In matching 3, everyone gets their second choice. You can verify that there are no blocking pairs.

You might wonder whether it is always possible to come up with a stable matching no matter what rankings each participant provides. The answer is yes. (Exercise 25.2-3 asks you to show that even in the scenario of the National Resident Matching Program, where each hospital takes on multiple students, it is always possible to devise a stable assignment.) A simple algorithm known as the Gale-Shapley algorithm always finds a stable matching. The algorithm has two variants, which mirror each other: "woman-oriented" and "man-oriented." Let's examine the woman-oriented version. Each participant is either "free" or "engaged." Everyone starts out free. Engagements occur when a free woman proposes to a man. When a man is first proposed to, he goes from free to engaged, and he always stays engaged, though not necessarily to the same woman. If an engaged man receives a proposal from a woman whom he prefers to the woman he's currently engaged to, that engagement is broken, the woman to whom he had been engaged becomes free, and the man and the woman whom he prefers become engaged. Each woman proposes to the men in her preference list, in order, until the last time she becomes engaged. When a woman is engaged, she temporarily stops proposing, but if she becomes free again, she continues down her list. Once everyone is engaged, the algorithm terminates. The procedure GALE-SHAPLEY on the next page makes this process more concrete. The procedure allows for some choice: any free woman may be selected in line 2. We'll see that the procedure produces a stable matching regardless of the order in which line 2 chooses free women. For the man-oriented version, just reverse the roles of men and women in the procedure.

Let's see how the GALE-SHAPLEY procedure executes on the example with Wanda, Emma, Lacey, Karen, Oscar, Davis, Brent, and Hank. After everyone is initialized to free, here is one possible version of what can occur in successive iterations of the **while** loop of lines 2–9:

1. Wanda proposes to Brent. Brent is free, so that Wanda and Brent become engaged and no longer free.

```
GALE-SHAPLEY(men, women, rankings)
assign each woman and man as free
while some woman w is free
    let m be the first man on w's ranked list to whom she has not proposed
    if m is free
        w and m become engaged to each other (and not free)
    elseif m ranks w higher than the woman w' he is currently engaged to
        m breaks the engagement to w', who becomes free
        w and m become engaged to each other (and not free)
    else m rejects w, with w remaining free
return the stable matching consisting of the engaged pairs
```

2. Emma proposes to Davis. Davis is free, so that Emma and Davis become engaged and no longer free.
3. Lacey proposes to Brent. Brent is engaged to Wanda, but he prefers Lacey. Brent breaks the engagement to Wanda, who becomes free. Lacey and Brent become engaged, with Lacey no longer free.
4. Karen proposes to Brent. Brent is engaged to Lacey, whom he prefers to Karen. Brent rejects Karen, who remains free.
5. Karen proposes to Hank. Hank is free, so that Karen and Hank become engaged and no longer free.
6. Wanda proposes to Hank. Hank is engaged to Karen, but he prefers Wanda. Hank breaks the engagement with Karen, who becomes free. Wanda and Hank become engaged, with Wanda no longer free.
7. Karen proposes to Davis. Davis is engaged to Emma, but he prefers Karen. Davis breaks the engagement to Emma, who becomes free. Karen and Davis become engaged, with Karen no longer free.
8. Emma proposes to Hank. Hank is engaged to Wanda, whom he prefers to Emma. Hank rejects Emma, who remains free.
9. Emma proposes to Oscar. Oscar is free, so that Emma and Oscar become engaged and no longer free.

At this point, everyone is engaged and nobody is free, so the **while** loop terminates. The procedure returns the stable matching we saw earlier.

The following theorem shows that not only does GALE-SHAPLEY terminate, but that it always returns a stable matching, thereby proving that a stable matching always exists.

Theorem 25.9
The procedure GALE-SHAPLEY always terminates and returns a stable matching.

Proof Let's first show that the **while** loop of lines 2–9 always terminates, so that the procedure terminates. The proof is by contradiction. If the loop fails to terminate, it is because some woman remains free. In order for a woman to remain free, she must have proposed to all the men and been rejected by each one. In order for a man to reject a woman, he must be already engaged. Therefore, all the men are engaged. Once engaged, a man stays engaged (though not necessarily to the same woman). There are an equal number n of women and men, however, which means that every woman is engaged, leading to the contradiction that no women are free. We must also show that the **while** loop makes a bounded number of iterations. Since each of the n women goes through her ranking of the n men in order, possibly not reaching the end of her list, the total number of iterations is at most n^2. Therefore, the **while** loop always terminates, and the procedure returns a matching.

We need to show that there are no blocking pairs. We first observe that once a man m is engaged to a woman w, all subsequent actions for m occur in lines 6–8. Therefore, once a man is engaged, he stays engaged, and any time he breaks an engagement to a woman w, it's for a woman whom he prefers to w. Suppose that a woman w is matched with a man m, but she prefers man m'. We'll show that w and m' is not a blocking pair, because m' does not prefer w to his partner. Because w ranks m' higher than m, she must have proposed to m' before proposing to m, and m' either rejected her proposal or accepted it and later broke the engagement. If m' rejected the proposal from w, it is because he was already engaged to some woman he prefers to w. If m' accepted and later broke the engagement, he was at some point engaged to w but later accepted a proposal from a woman he prefers to w. In either case, he ultimately ends up with a partner whom he prefers to w. We conclude that even though w might prefer m' to her partner m, it is not also the case that m' prefers w to his partner. Therefore, the procedure returns a matching containing no blocking pairs. ■

Exercise 25.2-1 asks you to provide the proof of the following corollary.

Corollary 25.10
Given preference rankings for n women and n men, the Gale-Shapley algorithm can be implemented to run in $O(n^2)$ time. ■

Because line 2 can choose any free woman, you might wonder whether different choices can produce different stable matchings. The answer is no: as the following

theorem shows, every execution of the GALE-SHAPLEY produces exactly the same result. Moreover, the stable matching returned is optimal for the women.

Theorem 25.11
Regardless of how women are chosen in line 2 of GALE-SHAPLEY, the procedure always returns the same stable matching, and in this stable matching, each woman has the best partner possible in any stable matching.

Proof The proof that each woman has the best partner possible in any stable matching is by contradiction. Suppose that the GALE-SHAPLEY procedure returns a stable matching M, but that there is a different stable matching M' in which some woman w prefers her partner m' to the partner m she has in M. Because w ranks m' higher than m, she must have proposed to m' before proposing to m. Then there is a woman w' whom m' prefers to w, and m' was already engaged to w' when w proposed or m' accepted the proposal from w and later broke the engagement in favor of w'. Either way, there is a moment when m' decided against w in favor of w'. Now suppose, without loss of generality, that this moment was the first time that any man rejected a partner who belongs to some stable matching.

We claim that w' cannot have a partner m'' in a stable matching whom she prefers to m'. If there were such a man m'', then in order for w' to propose to m', she would have proposed to m'' and been rejected at some point before proposing to m'. If m' accepted the proposal from w and later broke it to accept w', then since this was the first rejection in a stable matching, we get the contradiction that m'' could not have rejected w' beforehand. If m' was already engaged to w' when w proposed, then again, m'' could not have rejected w' beforehand, thus proving the claim.

Since w' does not prefer anyone to m' in a stable matching and w' is not matched with m' in M' (because m' is matched with w in M'), w' prefers m' to her partner in M'. Since w' prefers m' over her partner in M' and m' prefers w' over his partner w in M', the pair w' and m' is a blocking pair in M'. Because M' has a blocking pair, it cannot be a stable matching, thereby contradicting the assumption that there exists some stable matching in which each woman has the best partner possible other than the matching M returned by GALE-SHAPLEY.

We put no condition on the execution of the procedure, which means that all possible orders in which line 2 selects women result in the same stable matching being returned. ■

Corollary 25.12
There can be stable matchings that the GALE-SHAPLEY procedure does not return.

Proof Theorem 25.11 says that for a given set of rankings, GALE-SHAPLEY returns just one matching, no matter how it chooses women in line 2. The earlier ex-

ample of three women and three men with three different stable matchings shows that there can be multiple stable matchings for a given set of rankings. A call of GALE-SHAPLEY is capable of returning only one of these stable matchings. ■

Although the GALE-SHAPLEY procedure gives the best possible outcome for the women, the following corollary shows that it also produces the worst possible outcome for the men.

Corollary 25.13
In the stable matching returned by the procedure GALE-SHAPLEY, each man has the worst partner possible in any stable matching.

Proof Let M be the matching returned by a call to GALE-SHAPLEY. Suppose that there is another stable matching M' and a man m who prefers his partner w in M to his partner w' in M'. Let the partner of w in M' be m'. By Theorem 25.11, m is the best partner that w can have in any stable matching, which means that w prefers m to m'. Since m prefers w to w', the pair w and m is a blocking pair in M', contradicting the assumption that M' is a stable matching. ■

Exercises

25.2-1
Describe how to implement the Gale-Shapley algorithm so that it runs in $O(n^2)$ time.

25.2-2
Is it possible to have an unstable matching with just two women and two men? If so, provide and justify an example. If not, argue why not.

25.2-3
The National Resident Matching Program differs from the scenario for the stable-marriage problem set out in this section in two ways. First, a hospital may be matched with more than one student, so that hospital h takes $r_h \geq 1$ students. Second, the number of students might not equal the number of hospitals. Describe how to modify the Gale-Shapley algorithm to fit the requirements of the National Resident Matching Program.

25.2-4
Prove the following property, which is known as ***weak Pareto optimality***:

> Let M be the stable matching produced by the GALE-SHAPLEY procedure, with women proposing to men. Then, for a given instance of the stable-marriage problem there is no matching—stable or unstable—such that every

woman has a partner whom she prefers to her partner in the stable matching M.

25.2-5

The ***stable-roommates problem*** is similar to the stable-marriage problem, except that the graph is a complete graph, not bipartite, with an even number of vertices. Each vertex represents a person, and each person ranks all the other people. The definitions of blocking pairs and stable matching extend in the natural way: a blocking pair comprises two people who both prefer each other to their current partner, and a matching is stable if there are no blocking pairs. For example, consider four people—Wendy, Xenia, Yolanda, and Zelda—with the following preference lists:

Wendy: Xenia, Yolanda, Zelda
Xenia: Wendy, Zelda, Yolanda
Yolanda: Wendy, Zelda, Xenia
Zelda: Xenia, Yolanda, Wendy

You can verify that the following matching is stable:

Wendy and Xenia
Yolanda and Zelda

Unlike the stable-marriage problem, the stable-roommates problem can have inputs for which no stable matching exists. Find such an input and explain why no stable matching exists.

25.3 The Hungarian algorithm for the assignment problem

Let us once again add some information to a complete bipartite graph $G = (V, E)$, where $V = L \cup R$. This time, instead of having the vertices of each side rank the vertices on the other side, we assign a weight to each edge. Again, let's assume that the vertex sets L and R each contain n vertices, so that the graph contains n^2 edges. For $l \in L$ and $r \in R$, denote the weight of edge (l, r) by $w(l, r)$, which represents the utility gained by matching vertex l with vertex r.

The goal is to find a perfect matching M^* (see Exercises 25.1-5 and 25.1-6) whose edges have the maximum total weight over all perfect matchings. That is, letting $w(M) = \sum_{(l,r)\in M} w(l, r)$ denote the total weight of the edges in matching M, we want to find a perfect matching M^* such that

$$w(M^*) = \max \{w(M) : M \text{ is a perfect matching}\} .$$

We call finding such a maximum-weight perfect matching the ***assignment problem***. A solution to the assignment problem is a perfect matching that maximizes the total utility. Like the stable-marriage problem, the assignment problem finds a matching that is "good," but with a different definition of good: maximizing total value rather than achieving stability.

Although you could enumerate all $n!$ perfect matchings to solve the assignment problem, an algorithm known as the ***Hungarian algorithm*** solves it much faster. This section will prove an $O(n^4)$ time bound, and Problem 25-2 asks you to refine the algorithm to reduce the running time to $O(n^3)$. Instead of working with the complete bipartite graph G, the Hungarian algorithm works with a subgraph of G called the "equality subgraph." The equality subgraph, which is defined below, changes over time and has the beneficial property that any perfect matching in the equality subgraph is also an optimal solution to the assignment problem.

The equality subgraph depends on assigning an attribute h to each vertex. We call h the ***label*** of a vertex, and we say that h is a ***feasible vertex labeling*** of G if

$$l.h + r.h \geq w(l, r) \text{ for all } l \in L \text{ and } r \in R \ .$$

A feasible vertex labeling always exists, such as the ***default vertex labeling*** given by

$$l.h = \max\{w(l, r) : r \in R\} \quad \text{for all } l \in L \ , \tag{25.1}$$

$$r.h = 0 \quad \text{for all } r \in R \ . \tag{25.2}$$

Given a feasible vertex labeling h, the ***equality subgraph*** $G_h = (V, E_h)$ of G consists of the same vertices as G and the subset of edges

$$E_h = \{(l, r) \in E : l.h + r.h = w(l, r)\} \ .$$

The following theorem ties together a perfect matching in an equality subgraph and an optimal solution to the assignment problem.

Theorem 25.14

Let $G = (V, E)$, where $V = L \cup R$, be a complete bipartite graph where each edge $(l, r) \in E$ has weight $w(l, r)$. Let h be a feasible vertex labeling of G and G_h be the equality subgraph of G. If G_h contains a perfect matching M^*, then M^* is an optimal solution to the assignment problem on G.

Proof If G_h contains a perfect matching M^*, then because G_h and G have the same sets of vertices, M^* is also a perfect matching in G. Because each edge of M^* belongs to G_h and each vertex has exactly one incident edge from any perfect matching, we have

$$\begin{aligned} w(M^*) &= \sum_{(l,r)\in M^*} w(l,r) \\ &= \sum_{(l,r)\in M^*} (l.h + r.h) \quad \text{(because all edges in } M^* \text{ belong to } G_h) \\ &= \sum_{l\in L} l.h + \sum_{r\in R} r.h \quad \text{(because } M^* \text{ is a perfect matching)} . \end{aligned}$$

Letting M be any perfect matching in G, we have

$$\begin{aligned} w(M) &= \sum_{(l,r)\in M} w(l,r) \\ &\le \sum_{(l,r)\in M} (l.h + r.h) \quad \text{(because } h \text{ is a feasible vertex labeling)} \\ &= \sum_{l\in L} l.h + \sum_{r\in R} r.h \quad \text{(because } M \text{ is a perfect matching)} . \end{aligned}$$

Thus, we have

$$w(M) \le \sum_{l\in L} l.h + \sum_{r\in R} r.h = w(M^*) , \tag{25.3}$$

so that M^* is a maximum-weight perfect matching in G. ■

The goal now becomes finding a perfect matching in an equality subgraph. Which equality subgraph? It does not matter! We have free rein to not only choose an equality subgraph, but to change which equality subgraph we choose as we go along. We just need to find *some* perfect matching in *some* equality subgraph.

To understand the equality subgraph better, consider again the proof of Theorem 25.14 and, in the second half, let M be any matching. The proof is still valid, in particular, inequality (25.3): the weight of any matching is always at most the sum of the vertex labels. If we choose any set of vertex labels that define an equality subgraph, then a maximum-cardinality matching in this equality subgraph has total value at most the sum of the vertex labels. If the set of vertex labels is the "right" one, then it will have total value equal to $w(M^*)$, and a maximum-cardinality matching in the equality subgraph is also a maximum-weight perfect matching. The Hungarian algorithm repeatedly modifies the matching and the vertex labels in order to achieve this goal.

The Hungarian algorithm starts with any feasible vertex labeling h and any matching M in the equality subgraph G_h. It repeatedly finds an M-augmenting path P in G_h and, using Lemma 25.1, updates the matching to be $M \oplus P$, thereby incrementing the size of the matching. As long as there is some equality subgraph that contains an M-augmenting path, the size of the matching can increase, until a perfect matching is achieved.

Four questions arise:

1. What initial feasible vertex labeling should the algorithm start with? Answer: the default vertex labeling given by equations (25.1) and (25.2).
2. What initial matching in G_h should the algorithm start with? Short answer: any matching, even an empty matching, but a greedy maximal matching works well.
3. If an M-augmenting path exists in G_h, how to find it? Short answer: use a variant of breadth-first search similar to the second phase of the procedure used in the Hopcroft-Karp algorithm to find a maximal set of shortest M-augmenting paths.
4. What if the search for an M-augmenting path fails? Short answer: update the feasible vertex labeling to bring in at least one new edge.

We'll elaborate on the short answers using the example that starts in Figure 25.4. Here, $L = \{l_1, l_2, \ldots, l_7\}$ and $R = \{r_1, r_2, \ldots, r_7\}$. The edge weights appear in the matrix shown in part (a), where the weight $w(l_i, r_j)$ appears in row i and column j. The feasible vertex labels, given by the default vertex labeling, appear to the left of and above the matrix. Matrix entries in red indicate edges (l_i, r_j) for which $l_i.h + r_j.h = w(l_i, r_j)$, that is, edges in the equality subgraph G_h appearing in part (b) of the figure.

Greedy maximal bipartite matching

There are several ways to implement a greedy method to find a maximal bipartite matching. The procedure GREEDY-BIPARTITE-MATCHING shows one. Edges in Figure 25.4(b) highlighted in blue indicate the initial greedy maximal matching in G_h. Exercise 25.3-2 asks you to show that the GREEDY-BIPARTITE-MATCHING procedure returns a matching that is at least half the size of a maximum matching.

```
GREEDY-BIPARTITE-MATCHING(G)
M = ∅
for each vertex l ∈ L
    if l has an unmatched neighbor in R
        choose any such unmatched neighbor r ∈ R
        M = M ∪ {(l, r)}
return M
```

	h	r_1	r_2	r_3	r_4	r_5	r_6	r_7
		0	0	0	0	0	0	0
l_1	10	4	10	10	10	2	9	3
l_2	12	6	8	5	12	9	7	2
l_3	15	11	9	6	7	9	5	15
l_4	9	3	9	6	7	5	6	3
l_5	6	2	6	5	3	2	4	2
l_6	11	10	8	11	4	11	2	11
l_7	8	3	4	5	4	3	6	8

(a)

Figure 25.4 The start of the Hungarian algorithm. **(a)** The matrix of edge weights for a bipartite graph with $L = \{l_1, l_2, \ldots, l_7\}$ and $R = \{r_1, r_2, \ldots, r_7\}$. The value in row i and column j indicates $w(l_i, r_j)$. Feasible vertex labels appear above and next to the matrix. Red entries correspond to edges in the equality subgraph. **(b)** The equality subgraph G_h. Edges highlighted in blue belong to the initial greedy maximal matching M. Blue vertices are matched, and tan vertices are unmatched. **(c)** The directed equality subgraph $G_{M,h}$ created from G_h by directing edges in M from R to L and all other edges from L to R.

Finding an M-augmenting path in G_h

To find an M-augmenting path in the equality subgraph G_h with a matching M, the Hungarian algorithm first creates the ***directed equality subgraph*** $G_{M,h}$ from G_h, just as the Hopcroft-Karp algorithm creates G_M from G. As in the Hopcroft-Karp algorithm, you can think of an M-augmenting path as starting from an unmatched vertex in L, ending at an unmatched vertex in R, taking unmatched edges from L to R, and taking matched edges from R to L. Thus, $G_{M,h} = (V, E_{M,h})$, where

$$\begin{aligned} E_{M,h} = {} & \{(l,r) : l \in L, r \in R, \text{ and } (l,r) \in E_h - M\} && \text{(edges from } L \text{ to } R) \\ & \cup \{(r,l) : r \in R, l \in L, \text{ and } (l,r) \in M\} && \text{(edges from } R \text{ to } L)\,. \end{aligned}$$

Because an M-augmenting path in the directed equality subgraph $G_{M.h}$ is also an M-augmenting path in the equality subgraph G_h, it suffices to find M-augmenting paths in $G_{M.h}$. Figure 25.4(c) shows the directed equality subgraph $G_{M,h}$ corresponding to the equality subgraph G_h and matching M from part (b) of the figure.

With the directed equality subgraph $G_{M,h}$ in hand, the Hungarian algorithm searches for an M-augmenting path from any unmatched vertex in L to any unmatched vertex in R. Any exhaustive graph-search method suffices. Here, we'll use breadth-first search, starting from all the unmatched vertices in L (just as the Hopcroft-Karp algorithm does when creating the dag H), but stopping upon first discovering some unmatched vertex in R. Figure 25.5 shows the idea. To start from all the unmatched vertices in L, initialize the first-in, first-out queue with all the unmatched vertices in L, rather than just one source vertex. Unlike the dag H in the Hopcroft-Karp algorithm, here each vertex needs just one predecessor, so that the breadth-first search creates a ***breadth-first forest*** $F = (V_F, E_F)$. Each unmatched vertex in L is a root in F.

In Figure 25.5(g), the breadth-first search has found the M-augmenting path $\langle (l_4, r_2), (r_2, l_1), (l_1, r_3), (r_3, l_6), (l_6, r_5) \rangle$. Figure 25.6(a) shows the new matching created by taking the symmetric difference of the matching M in Figure 25.5(a) with this M-augmenting path.

When the search for an M-augmenting path fails

Having updated the matching M from an M-augmenting path, the Hungarian algorithm updates the directed equality subgraph $G_{M,h}$ according to the new matching and then starts a new breadth-first search from all the unmatched vertices in L. Figure 25.6 shows the start of this process, picking up from Figure 25.5.

In Figure 25.6(d), the queue contains vertices l_4 and l_3. Neither of these vertices has an edge that leaves it, however, so that once these vertices are removed from the queue, the queue becomes empty. The search terminates at this point, before discovering an unmatched vertex in R to yield an M-augmenting path. Whenever this situation occurs, the most recently discovered vertices must belong to L. Why? Whenever an unmatched vertex in R is discovered, the search has found an M-augmenting path, and when a matched vertex in R is discovered, it has an unvisited neighbor in L, which the search can then discover.

Recall that we have the freedom to work with any equality subgraph. We can change the directed equality subgraph "on the fly," as long we do not counteract the work already done. The Hungarian algorithm updates the feasible vertex labeling h to fulfill the following criteria:

1. No edge in the breadth-first forest F leaves the directed equality subgraph.
2. No edge in the matching M leaves the directed equality subgraph.
3. At least one edge (l, r), where $l \in L \cap V_F$ and $r \in R - V_F$ goes into E_h, and hence into $E_{M,h}$. Therefore, at least one vertex in R will be newly discovered.

Thus, at least one new edge enters the directed equality subgraph, and any edge that leaves the directed equality subgraph belongs to neither the matching M nor

Figure 25.5 Finding an M-augmenting path in $G_{M,h}$ by breadth-first search. **(a)** The directed equality subgraph $G_{M,h}$ from Figure 25.4(c). **(b)–(g)** Successive versions of the breadth-first forest F, shown as the vertices at each distance from the roots—the unmatched vertices in L—are discovered. In parts (b)–(f), the layer of vertices closest to the bottom of the figure are those in the first-in, first-out queue. For example, in part (b), the queue contains the roots $\langle l_4, l_5, l_7 \rangle$, and in part (e), the queue contains $\langle r_3, r_4 \rangle$, at distance 3 from the roots. In part (g), the unmatched vertex r_5 is discovered, so the breadth-first search terminates. The path $\langle (l_4, r_2), (r_2, l_1), (l_1, r_3), (r_3, l_6), (l_6, r_5) \rangle$, highlighted in orange in parts (a) and (g), is an M-augmenting path. Taking its symmetric difference with the matching M yields a new matching with one more edge than M.

Figure 25.6 **(a)** The new matching M and the new directed equality subgraph $G_{M.h}$ after updating the matching in Figure 25.5(a) with the M-augmenting path in Figure 25.5(g). **(b)–(d)** Successive versions of the breadth-first forest F in a new breadth-first search with roots l_5 and l_7. After the vertices l_4 and l_3 in part (d) have been removed from the queue, the queue becomes empty before the search can discover an unmatched vertex in R.

the breadth-first forest F. Newly discovered vertices in R are enqueued, but their distances are not necessarily 1 greater than the distances of the most recently discovered vertices in L.

To update the feasible vertex labeling, the Hungarian algorithm first computes the value

$$\delta = \min \{l.h + r.h - w(l, r) : l \in F_L \text{ and } r \in R - F_R\} , \tag{25.4}$$

where $F_L = L \cap V_F$ and $F_R = R \cap V_F$ denote the vertices in the breadth-first forest F that belong to L and R, respectively. That is, δ is the smallest difference by which an edge incident on a vertex in F_L missed being in the current equality subgraph G_h. The Hungarian algorithm then creates a new feasible vertex labeling, say h', by subtracting δ from $l.h$ for all vertices $l \in F_L$ and adding δ to $r.h$ for all vertices $r \in F_R$:

$$v.h' = \begin{cases} v.h - \delta & \text{if } v \in F_L , \\ v.h + \delta & \text{if } v \in F_R , \\ v.h & \text{otherwise } (v \in V - V_F) . \end{cases} \tag{25.5}$$

The following lemma shows that these changes achieve the three criteria above.

Lemma 25.15
Let h be a feasible vertex labeling for the complete bipartite graph G with equality subgraph G_h, and let M be a matching for G_h and F be a breadth-first forest being constructed for the directed equality subgraph $G_{M,h}$. Then, the labeling h' in equation (25.5) is a feasible vertex labeling for G with the following properties:

1. If (u, v) is an edge in the breadth-first forest F for $G_{M,h}$, then $(u, v) \in E_{M,h'}$.
2. If (l, r) belongs to the matching M for G_h, then $(r, l) \in E_{M,h'}$.
3. There exist vertices $l \in F_L$ and $r \in R - F_R$ such that $(l, r) \notin E_{M,h}$ but $(l, r) \in E_{M,h'}$.

Proof We first show that h' is a feasible vertex labeling for G. Because h is a feasible vertex labeling, we have $l.h + r.h \geq w(l, r)$ for all $l \in L$ and $r \in R$. In order for h' to not be a feasible vertex labeling, we would need $l.h' + r.h' < l.h + r.h$ for some $l \in L$ and $r \in R$. The only way this could occur would be for some $l \in F_L$ and $r \in R - F_R$. In this instance, the amount of the decrease equals δ, so that $l.h' + r.h' = l.h - \delta + r.h$. By equation (25.4), we have that $l.h - \delta + r.h \geq w(l, r)$ for any $l \in F_L$ and $r \in R - F_R$, so that $l.h' + r.h' \geq w(l, r)$. For all other edges, we have $l.h' + r.h' \geq l.h + r.h \geq w(l, r)$. Thus, h' is a feasible vertex labeling.

Now we show that each of the three desired properties holds:

1. If $l \in F_L$ and $r \in F_R$, then we have $l.h' + r.h' = l.h + r.h$ because δ is added to the label of l and subtracted from the label of r. Therefore, if an edge belongs to F for the directed graph $G_{M,h}$, it also belongs to $G_{M,h'}$.
2. We claim that at the time the Hungarian algorithm computes the new feasible vertex labeling h', for every edge $(l, r) \in M$, we have $l \in F_L$ if and only if $r \in F_R$. To see why, consider a matched vertex r and let $(l, r) \in M$. First suppose that $r \in F_R$, so that the search discovered r and enqueued it. When r was removed from the queue, l was discovered, so $l \in F_L$. Now suppose that $r \notin F_R$, so r is undiscovered. We will show that $l \notin F_L$. The only edge in $G_{M,h}$ that enters l is (r, l), and since r is undiscovered, the search has not taken this edge; if $l \in F_L$, it is not because of the edge (r, l). The only other way that a vertex in L can be in F_L is if it is a root of the search, but only unmatched vertices in L are roots and l is matched. Thus, $l \notin F_L$, and the claim is proved.

   We already saw that $l \in F_L$ and $r \in F_R$ implies $l.h' + r.h' = l.h + r.h$. For the opposite case, when $l \in L - F_L$ and $R \in R - F_R$, we have that $l.h' = l.h$ and $r.h' = r.h$, so that again $l.h' + r.h' = l.h + r.h$. Thus, if edge (l, r) is in the matching M for the equality graph G_h, then $(r, l) \in E_{M,h'}$.

3. Let (l, r) be an edge not in E_h such that $l \in F_L$, $r \in R - F_R$, and $\delta = l.h + r.h - w(l, r)$. By the definition of δ, there is at least one such edge. Then, we have

$$\begin{aligned} l.h' + r.h' &= l.h - \delta + r.h \\ &= l.h - (l.h + r.h - w(l, r)) + r.h \\ &= w(l, r) , \end{aligned}$$

and thus $(l, r) \in E_{h'}$. Since (l, r) is not in E_h, it is not in the matching M, so that in $E_{M,h'}$ it must be directed from L to R. Thus, $(l, r) \in E_{M,h'}$. ■

It is possible for an edge to belong to $E_{M,h}$ but not to $E_{M,h'}$. By Lemma 25.15, any such edge belongs neither to the matching M nor to the breadth-first forest F at the time that the new feasible vertex labeling h' is computed. (See Exercise 25.3-3.)

Going back to Figure 25.6(d), the queue became empty before an M-augmenting path was found. Figure 25.7 shows the next steps taken by the algorithm. The value of $\delta = 1$ is achieved by the edge (l_5, r_3) because in Figure 25.4(a), $l_5.h + r_3.h - w(l_5, r_3) = 6 + 0 - 5 = 1$. In Figure 25.7(a), the values of $l_3.h$, $l_4.h$, $l_5.h$, and $l_7.h$ have decreased by 1 and the values of $r_2.h$ and $r_7.h$ have increased by 1 because these vertices are in F. As a result, the edges (l_1, r_2) and (l_6, r_7) leave $G_{M,h}$ and the edge (l_5, r_3) enters. Figure 25.7(b) shows the new directed equality subgraph $G_{M,h}$. With edge (l_5, r_3) now in $G_{M,h}$, Figure 25.7(c) shows that this edge is added to the breadth-first forest F, and r_3 is added to the queue. Parts (c)–(f) show the breadth-first forest continuing to be built until in part (f), the queue once again becomes empty after vertex l_2, which has no edges leaving, is removed. Again, the algorithm must update the feasible vertex labeling and the directed equality subgraph. Now the value of $\delta = 1$ is achieved by three edges: (l_1, r_6), (l_5, r_6), and (l_7, r_6).

As Figure 25.8 shows in parts (a) and (b), these edges enter $G_{M,h}$, and edge (l_6, r_3) leaves. Part (c) shows that edge (l_1, r_6) is added to the breadth-first forest. (Either of edges (l_5, r_6) or (l_7, r_6) could have been added instead.) Because r_6 is unmatched, the search has found the M-augmenting path $\langle (l_5, r_3), (r_3, l_1), (l_1, r_6) \rangle$, highlighted in orange.

Figure 25.9(a) shows $G_{M,h}$ after the matching M has been updated by taking its symmetric difference with the M-augmenting path. The Hungarian algorithm starts its last breadth-first search, with vertex l_7 as the only root. The search proceeds as shown in parts (b)–(h) of the figure, until the queue becomes empty after removing l_4. This time, we find that $\delta = 2$, achieved by the five edges (l_2, r_5), (l_3, r_1), (l_4, r_5), (l_5, r_1), and (l_5, r_5), each of which enters $G_{M,h}$. Figure 25.10(a) shows the results of decreasing the feasible vertex label of each vertex in F_L by 2 and increasing the feasible vertex label of each vertex in F_R

Figure 25.7 Updating the feasible vertex labeling and the directed equality subgraph $G_{M,h}$ when the queue becomes empty before finding an M-augmenting path. **(a)** With $\delta = 1$, the values of $l_3.h$, $l_4.h$, $l_5.h$, and $l_7.h$ decreased by 1 and $r_2.h$ and $r_7.h$ increased by 1. Edges (l_1, r_2) and (l_6, r_7) leave $G_{M,h}$, and edge (l_5, r_3) enters. These changes are highlighted in yellow. **(b)** The resulting directed equality subgraph $G_{M,h}$. **(c)–(f)** With edge (l_5, r_3) added to the breadth-first forest and r_3 added to the queue, the breadth-first search continues until the queue once again becomes empty in part (f).

by 2, and Figure 25.10(b) shows the resulting directed equality subgraph $G_{M,h}$. Part (c) shows that edge (l_3, r_1) is added to the breadth-first forest. Since r_1 is an unmatched vertex, the search terminates, having found the M-augmenting path $\langle (l_7, r_7), (r_7, l_3), (l_3, r_1) \rangle$, highlighted in orange. If r_1 had been matched, vertex r_5 would also have been added to the breadth-first forest, with any of l_2, l_4, or l_5 as its parent.

After updating the matching M, the algorithm arrives at the perfect matching shown for the equality subgraph G_h in Figure 25.11. By Theorem 25.14, the edges in M form an optimal solution to the original assignment problem given in the matrix. Here, the weights of edges (l_1, r_6), (l_2, r_4), (l_3, r_1), (l_4, r_2), (l_5, r_3), (l_6, r_5), and (l_7, r_7) sum to 65, which is the maximum weight of any matching.

The weight of the maximum-weight matching equals the sum of all the feasible vertex labels. These problems—maximizing the weight of a matching and mini-

	h	r_1	r_2	r_3	r_4	r_5	r_6	r_7
		0	2	1	1	0	0	2
l_1	9	4	10	10	10	2	9	3
l_2	11	6	8	5	12	9	7	2
l_3	13	11	9	6	7	9	5	15
l_4	7	3	9	6	7	5	6	3
l_5	4	2	6	5	3	2	4	2
l_6	11	10	8	11	4	11	2	11
l_7	6	3	4	5	4	3	6	8

(a)

(b) (c)

Figure 25.8 Another update to the feasible vertex labeling and directed equality subgraph $G_{M,h}$ because the queue became empty before finding an M-augmenting path. **(a)** With $\delta = 1$, the values of $l_1.h$, $l_2.h$, $l_3.h$, $l_4.h$, $l_5.h$, and $l_7.h$ decrease by 1, and $r_2.h$, $r_3.h$, $r_4.h$, and $r_7.h$ increase by 1. Edge (l_6, r_3) leaves $G_{M,h}$, and edges (l_1, r_6), (l_5, r_6) and (l_7, r_6) enter. **(b)** The resulting directed equality subgraph $G_{M,h}$. **(c)** With edge (l_1, r_6) added to the breadth-first forest and r_6 unmatched, the search terminates, having found the M-augmenting path $\langle (l_5, r_3), (r_3, l_1), (l_1, r_6) \rangle$, highlighted in orange in parts (b) and (c).

mizing the sum of the feasible vertex labels—are "duals" of each other, in a similar vein to how the value of a maximum flow equals the capacity of a minimum cut. Section 29.3 explores duality in more depth.

The Hungarian algorithm

The procedure HUNGARIAN on page 737 and its subroutine FIND-AUGMENTING-PATH on page 738 follow the steps we have just seen. The third property in Lemma 25.15 ensures that in line 23 of FIND-AUGMENTING-PATH the queue Q is nonempty. The pseudocode uses the attribute π to indicate predecessor vertices in the breadth-first forest. Instead of coloring vertices, as in the BFS procedure on page 556, the search puts the discovered vertices into the sets F_L and F_R. Because the Hungarian algorithm does not need breadth-first distances, the pseudocode omits the d attribute computed by the BFS procedure.

Figure 25.9 **(a)** The new matching M and the new directed equality subgraph $G_{M,h}$ after updating the matching in Figure 25.8 with the M-augmenting path in Figure 25.8 parts (b) and (c). **(b)–(h)** Successive versions of the breadth-first forest F in a new breadth-first search with root l_7. After the vertex l_4 in part (h) has been removed from the queue, the queue becomes empty before the search discovers an unmatched vertex in R.

Now, let's see why the Hungarian algorithm runs in $O(n^4)$ time, where $|V| = n/2$ and $|E| = n^2$ in the original graph G. (Below we outline how to reduce the running time to $O(n^3)$.) You can go through the pseudocode of HUNGARIAN to verify that lines 1–6 and 11 take $O(n^2)$ time. The **while** loop of lines 7–10 iterates at most n times, since each iteration increases the size of the matching M by 1. Each test in line 7 can take constant time by just checking whether $|M| < n$, each update of M in line 9 takes $O(n)$ time, and the updates in line 10 take $O(n^2)$ time.

To achieve the $O(n^4)$ time bound, it remains to show that each call of FIND-AUGMENTING-PATH runs in $O(n^3)$ time. Let's call each execution of lines 10–22

	h	r_1	r_2	r_3	r_4	r_5	r_6	r_7
		0	4	3	3	0	2	4
l_1	7	4	10	10	10	2	9	3
l_2	9	6	8	5	12	9	7	2
l_3	11	11	9	6	7	9	5	15
l_4	5	3	9	6	7	5	6	3
l_5	2	2	6	5	3	2	4	2
l_6	11	10	8	11	4	11	2	11
l_7	4	3	4	5	4	3	6	8

(a)

(b)

Figure 25.10 Updating the feasible vertex labeling and directed equality subgraph $G_{M,h}$. **(a)** Here, $\delta = 2$, so the values of $l_1.h$, $l_2.h$, $l_3.h$, $l_4.h$, $l_5.h$, and $l_7.h$ decreased by 2, and the values of $r_2.h$, $r_3.h$, $r_4.h$, $r_6.h$, and $r_7.h$ increased by 2. Edges (l_2, r_5), (l_3, r_1), (l_4, r_5), (l_5, r_1), and (l_5, r_5) enter $G_{M,h}$. **(b)** The resulting directed graph $G_{M,h}$. **(c)** With edge (l_3, r_1) added to the breadth-first forest and r_1 unmatched, the search terminates, having found the M-augmenting path $\langle (l_7, r_7), (r_7, l_3), (l_3, r_1) \rangle$, highlighted in orange in parts (b) and (c).

a ***growth step***. Ignoring the growth steps, you can verify that FIND-AUGMENTING-PATH is a breadth-first search. With the sets F_L and F_R represented appropriately, the breadth-first search takes $O(V + E) = O(n^2)$ time. Within a call of FIND-AUGMENTING-PATH, at most n growth steps can occur, since each growth step is guaranteed to discover at least one vertex in R. Since there are at most n^2 edges in $G_{M,h}$, the **for** loop of lines 16–22 iterates at most n^2 times per call of FIND-AUGMENTING-PATH. The bottleneck is lines 10 and 15, which take $O(n^2)$ time, so that FIND-AUGMENTING-PATH takes $O(n^3)$ time.

Exercise 25.3-5 asks you to show that reconstructing the directed equality subgraph $G_{M,h}$ in line 15 is actually unnecessary, so that its cost can be eliminated. Reducing the cost of computing δ in line 10 to $O(n)$ takes a little more effort and is the subject of Problem 25-2. With these changes, each call of FIND-AUGMENTING-PATH takes $O(n^2)$ time, so that the Hungarian algorithm runs in $O(n^3)$ time.

		r_1	r_2	r_3	r_4	r_5	r_6	r_7
	h	0	4	3	3	0	2	4
l_1	7	4	10	10	10	2	**9**	3
l_2	9	6	8	5	**12**	9	7	2
l_3	11	**11**	9	6	7	9	5	15
l_4	5	3	**9**	6	7	5	6	3
l_5	2	2	6	**5**	3	2	4	2
l_6	11	10	8	11	4	**11**	2	11
l_7	4	3	4	5	4	3	6	**8**

(a)

(b)

Figure 25.11 The final matching, shown for the equality subgraph G_h with blue edges and blue entries in the matrix. The weights of the edges in the matching sum to 65, which is the maximum for any matching in the original complete bipartite graph G, as well as the sum of all the final feasible vertex labels.

```
HUNGARIAN(G)
for each vertex l ∈ L
    l.h = max {w(l, r) : r ∈ R}      // from equation (25.1)
for each vertex r ∈ R
    r.h = 0                          // from equation (25.2)
let M be any matching in G_h (such as the matching returned by
    GREEDY-BIPARTITE-MATCHING)
from G, M, and h, form the equality subgraph G_h
    and the directed equality subgraph G_{M,h}
while M is not a perfect matching in G_h
    P = FIND-AUGMENTING-PATH(G_{M,h})
    M = M ⊕ P
    update the equality subgraph G_h
        and the directed equality subgraph G_{M,h}
return M
```

```
FIND-AUGMENTING-PATH(G_{M,h})
1  Q = ∅
2  F_L = ∅
3  F_R = ∅
4  for each unmatched vertex l ∈ L
5      l.π = NIL
6      ENQUEUE(Q, l)
7      F_L = F_L ∪ {l}          // forest F starts with unmatched vertices in L
8  repeat
9      if Q is empty            // ran out of vertices to search from?
10         δ = min {l.h + r.h − w(l, r) : l ∈ F_L and r ∈ R − F_R}
11         for each vertex l ∈ F_L
12             l.h = l.h − δ    // relabel according to equation (25.5)
13         for each vertex r ∈ F_R
14             r.h = r.h + δ    // relabel according to equation (25.5)
15         from G, M, and h, form a new directed equality graph G_{M,h}
16         for each new edge (l, r) in G_{M,h}   // continue search with new edges
17             if r ∉ F_R
18                 r.π = l                       // discover r, add it to F
19                 if r is unmatched
20                     an M-augmenting path has been found
                           (exit the repeat loop)
21                 else ENQUEUE(Q, r)            // can search from r later
22                     F_R = F_R ∪ {r}
23     u = DEQUEUE(Q)                            // search from u
24     for each neighbor v of u in G_{M,h}
25         if v ∈ L
26             v.π = u
27             F_L = F_L ∪ {v}                   // discover v, add it to F
28             ENQUEUE(Q, v)                     // can search from v later
29         elseif v ∉ F_R                        // v ∈ R, do same as lines 18–22
30             v.π = u
31             if v is unmatched
32                 an M-augmenting path has been found
                       (exit the repeat loop)
33             else ENQUEUE(Q, v)
34                 F_R = F_R ∪ {v}
35 until an M-augmenting path has been found
36 using the predecessor attributes π, construct an M-augmenting path P
       by tracing back from the unmatched vertex in R
37 return P
```

Exercises

25.3-1
The FIND-AUGMENTING-PATH procedure checks in two places (lines 19 and 31) whether a vertex it discovers in R is unmatched. Show how to rewrite the pseudocode so that it checks for an unmatched vertex in R in only one place. What is the downside of doing so?

25.3-2
Show that for any bipartite graph, the GREEDY-BIPARTITE-MATCHING procedure on page 726 returns a matching at least half the size of a maximum matching.

25.3-3
Show that if an edge (l, r) belongs to the directed equality subgraph $G_{M,h}$ but is not a member of $G_{M,h'}$, where h' is given by equation (25.5), then $l \in L - F_L$ and $r \in F_R$ at the time that h' is computed.

25.3-4
At line 29 in the FIND-AUGMENTING-PATH procedure, it has already been established that $v \in R$. This line checks to see whether v is already discovered by testing whether $v \in F_R$. Why doesn't the procedure need to check whether v is already discovered for the case when $v \in L$, in lines 26–28?

25.3-5
Professor Hrabosky asserts that the directed equality subgraph $G_{M,h}$ must be constructed and maintained by the Hungarian algorithm, so that line 6 of HUNGARIAN and line 15 of FIND-AUGMENTING-PATH are required. Argue that the professor is incorrect by showing how to determine whether an edge belongs to $E_{M,h}$ without explicitly constructing $G_{M,h}$.

25.3-6
How can you modify the Hungarian algorithm to find a matching of vertices in L to vertices in R that minimizes, rather than maximizes, the sum of the edge weights in the matching?

25.3-7
How can an assignment problem with $|L| \neq |R|$ be modified so that the Hungarian algorithm solves it?

Problems

25-1 Perfect matchings in a regular bipartite graph

a. Problem 20-3 asked about Euler tours in directed graphs. Prove that a connected, *undirected* graph $G = (V, E)$ has an Euler tour—a cycle traversing each edge exactly once, though it may visit a vertex multiple times—if and only if the degree of every vertex in V is even.

b. Assuming that G is connected, undirected, and every vertex in V has even degree, give an $O(E)$-time algorithm to find an Euler tour of G, as in Problem 20-3(b).

c. Exercise 25.1-6 states that if $G = (V, E)$ is a d-regular bipartite graph, then it contains d disjoint perfect matchings. Suppose that d is an exact power of 2. Give an algorithm to find all d disjoint perfect matchings in a d-regular bipartite graph in $\Theta(E \lg d)$ time.

25-2 Reducing the running time of the Hungarian algorithm to $O(n^3)$

In this problem, you will show how to reduce the running time of the Hungarian algorithm from $O(n^4)$ to $O(n^3)$ by showing how to reduce the running time of the FIND-AUGMENTING-PATH procedure from $O(n^3)$ to $O(n^2)$. Exercise 25.3-5 demonstrates that line 6 of HUNGARIAN and line 15 of FIND-AUGMENTING-PATH are unnecessary. Now you will show how to reduce the running time of each execution of line 10 in FIND-AUGMENTING-PATH to $O(n)$.

For each vertex $r \in R - F_R$, define a new attribute $r.\sigma$, where

$$r.\sigma = \min\{l.h + r.h - w(l, r) : l \in F_L\} .$$

That is, $r.\sigma$ indicates how close r is to being adjacent to some vertex $l \in F_L$ in the directed equality subgraph $G_{m,h}$. Initially, before placing any vertices into F_L, set $r.\sigma$ to ∞ for all $r \in R$.

a. Show how to compute δ in line 10 in $O(n)$ time, based on the σ attribute.

b. Show how to update all the σ attributes in $O(n)$ time after δ has been computed.

c. Show that updating all the σ attributes when F_L changes takes $O(n^2)$ time per call of FIND-AUGMENTING-PATH.

d. Conclude that the HUNGARIAN procedure can be implemented to run in $O(n^3)$ time.

25-3 Other matching problems

The Hungarian algorithm finds a maximum-weight perfect matching in a complete bipartite graph. It is possible to use the Hungarian algorithm to solve problems in other graphs by modifying the input graph, running the Hungarian algorithm, and then possibly modifying the output. Show how to solve the following matching problems in this manner.

a. Give an algorithm to find a maximum-weight matching in a weighted bipartite graph that is not necessarily complete and with all edge weights positive.

b. Redo part (a), but with edge weights allowed to also be 0 or negative.

c. A ***cycle cover*** in a directed graph, not necessarily bipartite, is a set of edge-disjoint directed cycles such that each vertex lies on at most one cycle. Given nonnegative edge weights $w(u, v)$, let C be the set of edges in a cycle cover, and define $w(C) = \sum_{(u,v)\in C} w(u, v)$ to be the weight of the cycle cover. Give an algorithm to find a maximum-weight cycle cover.

25-4 Fractional matchings

It is possible to define a ***fractional matching***. Given a graph $G = (V, E)$, we define a fractional matching x as a function $x : E \to [0, 1]$ (real numbers between 0 and 1, inclusive) such that for every vertex $u \in V$, we have $\sum_{(u,v)\in E} x(u, v) \le 1$. The value of a fractional matching is $\sum_{(u,v)\in E} x(u, v)$. The definition of a fractional matching is identical to that of a matching, except that a matching has the additional constraint that $x(u, v) \in \{0, 1\}$ for all edges $(u, v) \in E$. Given a graph, we let M^* denote a maximum matching and x^* denote a fractional matching with maximum value.

a. Argue that, for any bipartite graph, we must have $\sum_{(u,v)\in E} x^*(u, v) \ge |M^*|$.

b. Prove that, for any bipartite graph, we must have $\sum_{(u,v)\in E} x^*(e) \le |M^*|$. (*Hint:* Give an algorithm that converts a fractional matching with an integer value to a matching.) Conclude that the maximum value of a fractional matching in a bipartite graph is the same as the size of the maximum cardinality matching.

c. We can define a fractional matching in a weighted graph in the same manner: the value of the matching is now $\sum_{(u,v)\in E} w(u, v)x(u, v)$. Extend the results of the previous parts to show that in a weighted bipartite graph, the maximum value of a weighted fractional matching is equal to the value of a maximum weighted matching.

d. In a general graph, the analogous results do not necessarily hold. Give an example of a small graph that is not bipartite for which the fractional matching with maximum value is not a maximum matching.

25-5 Computing vertex labels

You are given a complete bipartite graph $G = (V, E)$ with edge weights $w(l, r)$ for all $(l, r) \in E$. You are also given a maximum-weight perfect matching M^* for G. You wish to compute a feasible vertex labeling h such that M^* is a perfect matching in the equality subgraph G_h. That is, you want to compute a labeling h of vertices such that

$$l.h + r.h \geq w(l, r) \quad \text{for all } l \in L \text{ and } r \in R \,, \tag{25.6}$$

$$l.h + r.h = w(l, r) \quad \text{for all } (l, r) \in M^* \,. \tag{25.7}$$

(Requirement (25.6) holds for all edges, and the stronger requirement (25.7) holds for all edges in M^*.) Give an algorithm to compute the feasible vertex labeling h, and prove that it is correct. (*Hint:* Use the similarity between conditions (25.6) and (25.7) and some of the properties of shortest paths proved in Chapter 22, in particular the triangle inequality (Lemma 22.10) and the convergence property (Lemma 22.14.))

Chapter notes

Matching algorithms have a long history and have been central to many breakthroughs in algorithm design and analysis. The book by Lovász and Plummer [306] is an excellent reference on matching problems, and the chapter on matching in the book by Ahuja, Magnanti and Orlin [10] also has extensive references.

The Hopcroft-Karp algorithm is by Hopcroft and Karp [224]. Madry [308] gave an $\widetilde{O}(E^{10/7})$-time algorithm, which is asymptotically faster than Hopcroft-Karp for sparse graphs.

Corollary 25.4 is due to Berge [53], and it also holds in graphs that are not bipartite. Matching in general graphs requires more complicated algorithms. The first polynomial-time algorithm, running in $O(V^4)$ time, is due to Edmonds [130] (in a paper that also introduced the notion of a polynomial-time algorithm). Like the bipartite case, this algorithm also uses augmenting paths, although the algorithm for finding augmenting paths in general graphs is more involved than the one for bipartite graphs. Subsequently, several $O(\sqrt{V}E)$-time algorithms appeared, including ones by Gabow and Tarjan [168] as part of an algorithm for weighted matching and a simpler one by Gabow [164].

The Hungarian algorithm is described in the book by Bondy and Murty [67] and is based on work by Kuhn [273] and Munkres [337]. Kuhn adopted the name "Hungarian algorithm" because the algorithm derived from work by the Hungarian mathematicians D. Kőnig and J. Egerváry. The algorithm is an early example of a primal-dual algorithm. A faster algorithm that runs in $O(\sqrt{V}E \log(VW))$ time, where the edge weights are integers from 0 to W, was given by Gabow and Tarjan [167], and an algorithm with the same time bound for maximum-weight matching in general graphs was given by Duan, Pettie, and Su [127].

The stable-marriage problem was first defined and analyzed by Gale and Shapley [169]. The stable-marriage problem has numerous variants. The books by Gusfield and Irving [203], Knuth [266], and Manlove [313] serve as excellent sources for cataloging and solving them.

Part VII Selected Topics

Introduction

This part contains a selection of algorithmic topics that extend and complement earlier material in this book. Some chapters introduce new models of computation such as circuits or parallel computers. Others cover specialized domains such as matrices or number theory. The last two chapters discuss some of the known limitations to the design of efficient algorithms and introduce techniques for coping with those limitations.

Chapter 26 presents an algorithmic model for parallel computing based on task-parallel computing, and more specifically, fork-join parallelism. The chapter introduces the basics of the model, showing how to quantify parallelism in terms of the measures of work and span. It then investigates several interesting fork-join algorithms, including algorithms for matrix multiplication and merge sorting.

An algorithm that receives its input over time, rather than having the entire input available at the start, is called an "online" algorithm. Chapter 27 examines techniques used in online algorithms, starting with the "toy" problem of how long to wait for an elevator before taking the stairs. It then studies the "move-to-front" heuristic for maintaining a linked list and finishes with the online version of the caching problem we saw back in Section 15.4. The analyses of these online algorithms are remarkable in that they prove that these algorithms, which do not know their future inputs, perform within a constant factor of optimal algorithms that know the future inputs.

Chapter 28 studies efficient algorithms for operating on matrices. It presents two general methods—LU decomposition and LUP decomposition—for solving linear equations by Gaussian elimination in $O(n^3)$ time. It also shows that matrix inversion and matrix multiplication can be performed equally fast. The chapter concludes by showing how to compute a least-squares approximate solution when a set of linear equations has no exact solution.

Chapter 29 studies how to model problems as linear programs, where the goal is to maximize or minimize an objective, given limited resources and competing constraints. Linear programming arises in a variety of practical application areas. The chapter also addresses the concept of "duality" which, by establishing that a maximization problem and minimization problem have the same objective value, helps to show that solutions to each are optimal.

Chapter 30 studies operations on polynomials and shows how to use a well-known signal-processing technique—the fast Fourier transform (FFT)—to multiply two degree-n polynomials in $O(n \lg n)$ time. It also derives a parallel circuit to compute the FFT.

Chapter 31 presents number-theoretic algorithms. After reviewing elementary number theory, it presents Euclid's algorithm for computing greatest common divisors. Next, it studies algorithms for solving modular linear equations and for raising one number to a power modulo another number. Then, it explores an important application of number-theoretic algorithms: the RSA public-key cryptosystem. This cryptosystem can be used not only to encrypt messages so that an adversary cannot read them, but also to provide digital signatures. The chapter finishes with the Miller-Rabin randomized primality test, which enables finding large primes efficiently—an essential requirement for the RSA system.

Chapter 32 studies the problem of finding all occurrences of a given pattern string in a given text string, a problem that arises frequently in text-editing programs. After examining the naive approach, the chapter presents an elegant approach due to Rabin and Karp. Then, after showing an efficient solution based on finite automata, the chapter presents the Knuth-Morris-Pratt algorithm, which modifies the automaton-based algorithm to save space by cleverly preprocessing the pattern. The chapter finishes by studying suffix arrays, which can not only find a pattern in a text string, but can do quite a bit more, such as finding the longest repeated substring in a text and finding the longest common substring appearing in two texts.

Chapter 33 examines three algorithms within the expansive field of machine learning. Machine-learning algorithms are designed to take in vast amounts of data, devise hypotheses about patterns in the data, and test these hypotheses. The chapter starts with k-means clustering, which groups data elements into k classes based on how similar they are to each other. It then shows how to use the technique of multiplicative weights to make predictions accurately based on a set of "experts" of varying quality. Perhaps surprisingly, even without knowing which experts are reliable and which are not, you can predict almost as accurately as the most reliable expert. The chapter finishes with gradient descent, an optimization technique that finds a local minimum value for a function. Gradient descent has many applications, including finding parameter settings for many machine-learning models.

Chapter 34 concerns NP-complete problems. Many interesting computational problems are NP-complete, but no polynomial-time algorithm is known for solving any of them. This chapter presents techniques for determining when a problem is NP-complete, using them to prove several classic problems NP-complete: determining whether a graph has a hamiltonian cycle (a cycle that includes every vertex), determining whether a boolean formula is satisfiable (whether there exists an assignment of boolean values to its variables that causes the formula to evaluate to TRUE), and determining whether a given set of numbers has a subset that adds up to a given target value. The chapter also proves that the famous traveling-salesperson problem (find a shortest route that starts and ends at the same location and visits each of a set of locations once) is NP-complete.

Chapter 35 shows how to find approximate solutions to NP-complete problems efficiently by using approximation algorithms. For some NP-complete problems, approximate solutions that are near optimal are quite easy to produce, but for others even the best approximation algorithms known work progressively more poorly as the problem size increases. Then, there are some problems for which investing increasing amounts of computation time yields increasingly better approximate solutions. This chapter illustrates these possibilities with the vertex-cover problem (unweighted and weighted versions), an optimization version of 3-CNF satisfiability, the traveling-salesperson problem, the set-covering problem, and the subset-sum problem.

26 Parallel Algorithms

The vast majority of algorithms in this book are ***serial algorithms*** suitable for running on a uniprocessor computer that executes only one instruction at a time. This chapter extends our algorithmic model to encompass ***parallel algorithms***, where multiple instructions can execute simultaneously. Specifically, we'll explore the elegant model of task-parallel algorithms, which are amenable to algorithmic design and analysis. Our study focuses on fork-join parallel algorithms, the most basic and best understood kind of task-parallel algorithm. Fork-join parallel algorithms can be expressed cleanly using simple linguistic extensions to ordinary serial code. Moreover, they can be implemented efficiently in practice.

Parallel computers—computers with multiple processing units—are ubiquitous. Handheld, laptop, desktop, and cloud machines are all ***multicore computers***, or simply, ***multicores***, containing multiple processing "cores." Each processing core is a full-fledged processor that can directly access any location in a common ***shared memory***. Multicores can be aggregated into larger systems, such as clusters, by using a network to interconnect them. These multicore clusters usually have a ***distributed memory***, where one multicore's memory cannot be accessed directly by a processor in another multicore. Instead, the processor must explicitly send a message over the cluster network to a processor in the remote multicore to request any data it requires. The most powerful clusters are supercomputers, comprising many thousands of multicores. But since shared-memory programming tends to be conceptually easier than distributed-memory programming, and multicore machines are widely available, this chapter focuses on parallel algorithms for multicores.

One approach to programming multicores is ***thread parallelism***. This processor-centric parallel-programming model employs a software abstraction of "virtual processors," or ***threads*** that share a common memory. Each thread maintains its own program counter and can execute code independently of the other threads. The operating system loads a thread onto a processing core for execution and switches it out when another thread needs to run.

Unfortunately, programming a shared-memory parallel computer using threads tends to be difficult and error-prone. One reason is that it can be complicated to dynamically partition the work among the threads so that each thread receives approximately the same load. For any but the simplest of applications, the programmer must use complex communication protocols to implement a scheduler that load-balances the work.

Task-parallel programming

The difficulty of thread programming has led to the creation of ***task-parallel platforms***, which provide a layer of software on top of threads to coordinate, schedule, and manage the processors of a multicore. Some task-parallel platforms are built as runtime libraries, but others provide full-fledged parallel languages with compiler and runtime support.

Task-parallel programming allows parallelism to be specified in a "processor-oblivious" fashion, where the programmer identifies what computational tasks may run in parallel but does not indicate which thread or processor performs the task. Thus, the programmer is freed from worrying about communication protocols, load balancing, and other vagaries of thread programming. The task-parallel platform contains a scheduler, which automatically load-balances the tasks across the processors, thereby greatly simplifying the programmer's chore. ***Task-parallel algorithms*** provide a natural extension to ordinary serial algorithms, allowing performance to be reasoned about mathematically using "work/span analysis."

Fork-join parallelism

Although the functionality of task-parallel environments is still evolving and increasing, almost all support ***fork-join parallelism***, which is typically embodied in two linguistic features: ***spawning*** and ***parallel loops***. Spawning allows a subroutine to be "forked": executed like a subroutine call, except that the caller can continue to execute while the spawned subroutine computes its result. A parallel loop is like an ordinary **for** loop, except that multiple iterations of the loop can execute at the same time.

Fork-join parallel algorithms employ spawning and parallel loops to describe parallelism. A key aspect of this parallel model, inherited from the task-parallel model but different from the thread model, is that the programmer does not specify which tasks in a computation *must* run in parallel, only which tasks *may* run in parallel. The underlying runtime system uses threads to load-balance the tasks across the processors. This chapter investigates parallel algorithms described in the fork-join model, as well as how the underlying runtime system can schedule task-parallel computations (which include fork-join computations) efficiently.

Fork-join parallelism offers several important advantages:

- The fork-join programming model is a simple extension of the familiar serial programming model used in most of this book. To describe a fork-join parallel algorithm, the pseudocode in this book needs just three added keywords: **parallel**, **spawn**, and **sync**. Deleting these parallel keywords from the parallel pseudocode results in ordinary serial pseudocode for the same problem, which we call the "serial projection" of the parallel algorithm.
- The underlying task-parallel model provides a theoretically clean way to quantify parallelism based on the notions of "work" and "span."
- Spawning allows many divide-and-conquer algorithms to be parallelized naturally. Moreover, just as serial divide-and-conquer algorithms lend themselves to analysis using recurrences, so do parallel algorithms in the fork-join model.
- The fork-join programming model is faithful to how multicore programming has been evolving in practice. A growing number of multicore environments support one variant or another of fork-join parallel programming, including Cilk [290, 291, 383, 396], Habanero-Java [466], the Java Fork-Join Framework [279], OpenMP [81], Task Parallel Library [289], Threading Building Blocks [376], and X10 [82].

Section 26.1 introduces parallel pseudocode, shows how the execution of a task-parallel computation can be modeled as a directed acyclic graph, and presents the metrics of work, span, and parallelism, which you can use to analyze parallel algorithms. Section 26.2 investigates how to multiply matrices in parallel, and Section 26.3 tackles the tougher problem of designing an efficient parallel merge sort.

26.1 The basics of fork-join parallelism

Our exploration of parallel programming begins with the problem of computing Fibonacci numbers recursively in parallel. We'll look at a straightforward serial Fibonacci calculation, which, although inefficient, serves as a good illustration of how to express parallelism in pseudocode.

Recall that the Fibonacci numbers are defined by equation (3.31) on page 69:

$$F_i = \begin{cases} 0 & \text{if } i = 0 , \\ 1 & \text{if } i = 1 , \\ F_{i-1} + F_{i-2} & \text{if } i \geq 2 . \end{cases}$$

To calculate the nth Fibonacci number recursively, you could use the ordinary serial algorithm in the procedure FIB on the facing page. You would not really want to

compute large Fibonacci numbers this way, because this computation does needless repeated work, but parallelizing it can be instructive.

```
FIB(n)
if n ≤ 1
    return n
else x = FIB(n − 1)
     y = FIB(n − 2)
     return x + y
```

To analyze this algorithm, let $T(n)$ denote the running time of FIB(n). Since FIB(n) contains two recursive calls plus a constant amount of extra work, we obtain the recurrence

$$T(n) = T(n-1) + T(n-2) + \Theta(1) .$$

This recurrence has solution $T(n) = \Theta(F_n)$, which we can establish by using the substitution method (see Section 4.3). To show that $T(n) = O(F_n)$, we'll adopt the inductive hypothesis that $T(n) \le aF_n - b$, where $a > 1$ and $b > 0$ are constants. Substituting, we obtain

$$\begin{aligned} T(n) &\le (aF_{n-1} - b) + (aF_{n-2} - b) + \Theta(1) \\ &= a(F_{n-1} + F_{n-2}) - 2b + \Theta(1) \\ &\le aF_n - b , \end{aligned}$$

if we choose b large enough to dominate the upper-bound constant in the $\Theta(1)$ term. We can then choose a large enough to upper-bound the $\Theta(1)$ base case for small n. To show that $T(n) = \Omega(F_n)$, we use the inductive hypothesis $T(n) \ge aF_n - b$. Substituting and following reasoning similar to the asymptotic upper-bound argument, we establish this hypothesis by choosing b smaller than the lower-bound constant in the $\Theta(1)$ term and a small enough to lower-bound the $\Theta(1)$ base case for small n. Theorem 3.1 on page 56 then establishes that $T(n) = \Theta(F_n)$, as desired. Since $F_n = \Theta(\phi^n)$, where $\phi = (1 + \sqrt{5})/2$ is the golden ratio, by equation (3.34) on page 69, it follows that

$$T(n) = \Theta(\phi^n) . \tag{26.1}$$

Thus this procedure is a particularly slow way to compute Fibonacci numbers, since it runs in exponential time. (See Problem 31-3 on page 954 for faster ways.)

Let's see why the algorithm is inefficient. Figure 26.1 shows the tree of recursive procedure instances created when computing F_6 with the FIB procedure. The call to FIB(6) recursively calls FIB(5) and then FIB(4). But, the call to FIB(5) also

Figure 26.1 The invocation tree for FIB(6). Each node in the tree represents a procedure instance whose children are the procedure instances it calls during its execution. Since each instance of FIB with the same argument does the same work to produce the same result, the inefficiency of this algorithm for computing the Fibonacci numbers can be seen by the vast number of repeated calls to compute the same thing. The portion of the tree shaded blue appears in task-parallel form in Figure 26.2.

results in a call to FIB(4). Both instances of FIB(4) return the same result ($F_4 = 3$). Since the FIB procedure does not memoize (recall the definition of "memoize" from page 368), the second call to FIB(4) replicates the work that the first call performs, which is wasteful.

Although the FIB procedure is a poor way to compute Fibonacci numbers, it can help us warm up to parallelism concepts. Perhaps the most basic concept is to understand is that if two parallel tasks operate on entirely different data, then—absent other interference—they each produce the same outcomes when executed at the same time as when they run serially one after the other. Within FIB(n), for example, the two recursive calls in line 3 to FIB($n - 1$) and in line 4 to FIB($n - 2$) can safely execute in parallel because the computation performed by one in no way affects the other.

Parallel keywords

The P-FIB procedure on the next page computes Fibonacci numbers, but using the ***parallel keywords*** **spawn** and **sync** to indicate parallelism in the pseudocode.

If the keywords **spawn** and **sync** are deleted from P-FIB, the resulting pseudocode text is identical to FIB (other than renaming the procedure in the header

```
P-FIB(n)
1  if n ≤ 1
2      return n
3  else x = spawn P-FIB(n − 1)      // don't wait for subroutine to return
4       y = P-FIB(n − 2)            // in parallel with spawned subroutine
5       sync                        // wait for spawned subroutine to finish
6       return x + y
```

and in the two recursive calls). We define the ***serial projection***[1] of a parallel algorithm to be the serial algorithm that results from ignoring the parallel directives, which in this case can be done by omitting the keywords **spawn** and **sync**. For **parallel for** loops, which we'll see later on, we omit the keyword **parallel**. Indeed, our parallel pseudocode possesses the elegant property that its serial projection is always ordinary serial pseudocode to solve the same problem.

Semantics of parallel keywords

Spawning occurs when the keyword **spawn** precedes a procedure call, as in line 3 of P-FIB. The semantics of a spawn differs from an ordinary procedure call in that the procedure instance that executes the spawn—the ***parent***—may continue to execute in parallel with the spawned subroutine—its ***child***—instead of waiting for the child to finish, as would happen in a serial execution. In this case, while the spawned child is computing P-FIB$(n-1)$, the parent may go on to compute P-FIB$(n-2)$ in line 4 in parallel with the spawned child. Since the P-FIB procedure is recursive, these two subroutine calls themselves create nested parallelism, as do their children, thereby creating a potentially vast tree of subcomputations, all executing in parallel.

The keyword **spawn** does not say, however, that a procedure *must* execute in parallel with its spawned children, only that it *may*. The parallel keywords express the ***logical parallelism*** of the computation, indicating which parts of the computation may proceed in parallel. At runtime, it is up to a ***scheduler*** to determine which subcomputations actually run in parallel by assigning them to available pro-

[1] In mathematics, a projection is an idempotent function, that is, a function f such that $f \circ f = f$. In this case, the function f maps the set $\mathcal{P}$ of fork-join programs to the set $\mathcal{P}_S \subset \mathcal{P}$ of serial programs, which are themselves fork-join programs with no parallelism. For a fork-join program $x \in \mathcal{P}$, since we have $f(f(x)) = f(x)$, the serial projection, as we have defined it, is indeed a mathematical projection.

cessors as the computation unfolds. We'll discuss the theory behind task-parallel schedulers shortly (on page 759).

A procedure cannot safely use the values returned by its spawned children until after it executes a **sync** statement, as in line 5. The keyword **sync** indicates that the procedure must wait as necessary for all its spawned children to finish before proceeding to the statement after the **sync**—the "join" of a fork-join parallel computation. The P-FIB procedure requires a **sync** before the **return** statement in line 6 to avoid the anomaly that would occur if x and y were summed before P-FIB$(n-1)$ had finished and its return value had been assigned to x. In addition to explicit join synchronization provided by the **sync** statement, it is convenient to assume that every procedure executes a **sync** implicitly before it returns, thus ensuring that all children finish before their parent finishes.

A graph model for parallel execution

It helps to view the execution of a parallel computation—the dynamic stream of runtime instructions executed by processors under the direction of a parallel program—as a directed acyclic graph $G = (V, E)$, called a ***(parallel) trace***.[2] Conceptually, the vertices in V are executed instructions, and the edges in E represent dependencies between instructions, where $(u, v) \in E$ means that the parallel program required instruction u to execute before instruction v.

It's sometimes inconvenient, especially if we want to focus on the parallel structure of a computation, for a vertex of a trace to represent only one executed instruction. Consequently, if a chain of instructions contains no parallel or procedural control (no **spawn**, **sync**, procedure call, or **return**—via either an explicit **return** statement or the return that happens implicitly upon reaching the end of a procedure), we group the entire chain into a single ***strand***. As an example, Figure 26.2 shows the trace that results from computing P-FIB(4) in the portion of Figure 26.1 shaded blue. Strands do not include instructions that involve parallel or procedural control. These control dependencies must be represented as edges in the trace.

When a parent procedure calls a child, the trace contains an edge (u, v) from the strand u in the parent that executes the call to the first strand v of the spawned child, as illustrated in Figure 26.2 by the edge from the orange strand in P-FIB(4) to the blue strand in P-FIB(2). When the last strand v' in the child returns, the trace contains an edge (v', u') to the strand u', where u' is the successor strand of u in the parent, as with the edge from the white strand in P-FIB(2) to the white strand in P-FIB(4).

---

[2] Also called a ***computation dag*** in the literature.

Figure 26.2 The trace of P-FIB(4) corresponding to the shaded portion of Figure 26.1. Each circle represents one strand, with blue circles representing any instructions executed in the part of the procedure (instance) up to the spawn of P-FIB($n-1$) in line 3; orange circles representing the instructions executed in the part of the procedure that calls P-FIB($n-2$) in line 4 up to the **sync** in line 5, where it suspends until the spawn of P-FIB($n-1$) returns; and white circles representing the instructions executed in the part of the procedure after the **sync**, where it sums x and y, up to the point where it returns the result. Strands belonging to the same procedure are grouped into a rounded rectangle, blue for spawned procedures and tan for called procedures. Assuming that each strand takes unit time, the work is 17 time units, since there are 17 strands, and the span is 8 time units, since the critical path—shown with blue edges—contains 8 strands.

When the parent spawns a child, however, the trace is a little different. The edge (u, v) goes from parent to child as with a call, such as the edge from the blue strand in P-FIB(4) to the blue strand in P-FIB(3), but the trace contains another edge (u, u') as well, indicating that u's successor strand u' can continue to execute while v is executing. The edge from the blue strand in P-FIB(4) to the orange strand in P-FIB(4) illustrates one such edge. As with a call, there is an edge from the last strand v' in the child, but with a spawn, it no longer goes to u's successor. Instead, the edge is (v', x), where x is the strand immediately following the **sync** in the parent that ensures that the child has finished, as with the edge from the white strand in P-FIB(3) to the white strand in P-FIB(4).

You can figure out what parallel control created a particular trace. If a strand has two successors, one of them must have been spawned, and if a strand has multiple predecessors, the predecessors joined because of a **sync** statement. Thus, in the general case, the set V forms the set of strands, and the set E of directed edges represents dependencies between strands induced by parallel and procedural

control. If G contains a directed path from strand u to strand v, we say that the two strands are ***(logically) in series***. If there is no path in G either from u to v or from v to u, the strands are ***(logically) in parallel***.

A fork-join parallel trace can be pictured as a dag of strands embedded in an ***invocation tree*** of procedure instances. For example, Figure 26.1 shows the invocation tree for FIB(6), which also serves as the invocation tree for P-FIB(6), the edges between procedure instances now representing either calls or spawns. Figure 26.2 zooms in on the subtree that is shaded blue, showing the strands that constitute each procedure instance in P-FIB(4). All directed edges connecting strands run either within a procedure or along undirected edges of the invocation tree in Figure 26.1. (More general task-parallel traces that are not fork-join traces may contain some directed edges that do not run along the undirected tree edges.)

Our analyses generally assume that parallel algorithms execute on an ***ideal parallel computer***, which consists of a set of processors and a ***sequentially consistent*** shared memory. To understand sequential consistency, you first need to know that memory is accessed by ***load instructions***, which copy data from a location in the memory to a register within a processor, and by ***store instructions***, which copy data from a processor register to a location in the memory. A single line of pseudocode can entail several such instructions. For example, the line $x = y + z$ could result in load instructions to fetch each of y and z from memory into a processor, an instruction to add them together inside the processor, and a store instruction to place the result x back into memory. In a parallel computer, several processors might need to load or store at the same time. Sequential consistency means that even if multiple processors attempt to access the memory simultaneously, the shared memory behaves as if exactly one instruction from one of the processors is executed at a time, even though the actual transfer of data may happen at the same time. It is as if the instructions were executed one at a time sequentially according to some global linear order among all the processors that preserves the individual orders in which each processor executes its own instructions.

For task-parallel computations, which are scheduled onto processors automatically by a runtime system, the sequentially consistent shared memory behaves as if a parallel computation's executed instructions were executed one by one in the order of a topological sort (see Section 20.4) of its trace. That is, you can reason about the execution by imagining that the individual instructions (not generally the strands, which may aggregate many instructions) are interleaved in some linear order that preserves the partial order of the trace. Depending on scheduling, the linear order could vary from one run of the program to the next, but the behavior of any execution is always as if the instructions executed serially in a linear order consistent with the dependencies within the trace.

In addition to making assumptions about semantics, the ideal parallel-computer model makes some performance assumptions. Specifically, it assumes that each

processor in the machine has equal computing power, and it ignores the cost of scheduling. Although this last assumption may sound optimistic, it turns out that for algorithms with sufficient "parallelism" (a term we'll define precisely a little later), the overhead of scheduling is generally minimal in practice.

Performance measures

We can gauge the theoretical efficiency of a task-parallel algorithm using ***work/span analysis***, which is based on two metrics: "work" and "span." The ***work*** of a task-parallel computation is the total time to execute the entire computation on one processor. In other words, the work is the sum of the times taken by each of the strands. If each strand takes unit time, the work is just the number of vertices in the trace. The ***span*** is the fastest possible time to execute the computation on an unlimited number of processors, which corresponds to the sum of the times taken by the strands along a longest path in the trace, where "longest" means that each strand is weighted by its execution time. Such a longest path is called the ***critical path*** of the trace, and thus the span is the weight of the longest (weighted) path in the trace. (Section 22.2, pages 617–619 shows how to find a critical path in a dag $G = (V, E)$ in $\Theta(V + E)$ time.) For a trace in which each strand takes unit time, the span equals the number of strands on the critical path. For example, the trace of Figure 26.2 has 17 vertices in all and 8 vertices on its critical path, so that if each strand takes unit time, its work is 17 time units and its span is 8 time units.

The actual running time of a task-parallel computation depends not only on its work and its span, but also on how many processors are available and how the scheduler allocates strands to processors. To denote the running time of a task-parallel computation on P processors, we subscript by P. For example, we might denote the running time of an algorithm on P processors by T_P. The work is the running time on a single processor, or T_1. The span is the running time if we could run each strand on its own processor—in other words, if we had an unlimited number of processors—and so we denote the span by T_∞.

The work and span provide lower bounds on the running time T_P of a task-parallel computation on P processors:

- In one step, an ideal parallel computer with P processors can do at most P units of work, and thus in T_P time, it can perform at most $P T_P$ work. Since the total work to do is T_1, we have $P T_P \geq T_1$. Dividing by P yields the ***work law***:

  $$T_P \geq T_1/P. \tag{26.2}$$

- A P-processor ideal parallel computer cannot run any faster than a machine with an unlimited number of processors. Looked at another way, a machine

with an unlimited number of processors can emulate a P-processor machine by using just P of its processors. Thus, the ***span law*** follows:

$$T_P \geq T_\infty \, . \tag{26.3}$$

We define the ***speedup*** of a computation on P processors by the ratio T_1/T_P, which says how many times faster the computation runs on P processors than on one processor. By the work law, we have $T_P \geq T_1/P$, which implies that $T_1/T_P \leq P$. Thus, the speedup on a P-processor ideal parallel computer can be at most P. When the speedup is linear in the number of processors, that is, when $T_1/T_P = \Theta(P)$, the computation exhibits ***linear speedup***. ***Perfect linear speedup*** occurs when $T_1/T_P = P$.

The ratio T_1/T_∞ of the work to the span gives the ***parallelism*** of the parallel computation. We can view the parallelism from three perspectives. As a ratio, the parallelism denotes the average amount of work that can be performed in parallel for each step along the critical path. As an upper bound, the parallelism gives the maximum possible speedup that can be achieved on any number of processors. Perhaps most important, the parallelism provides a limit on the possibility of attaining perfect linear speedup. Specifically, once the number of processors exceeds the parallelism, the computation cannot possibly achieve perfect linear speedup. To see this last point, suppose that $P > T_1/T_\infty$, in which case the span law implies that the speedup satisfies $T_1/T_P \leq T_1/T_\infty < P$. Moreover, if the number P of processors in the ideal parallel computer greatly exceeds the parallelism—that is, if $P \gg T_1/T_\infty$—then $T_1/T_P \ll P$, so that the speedup is much less than the number of processors. In other words, if the number of processors exceeds the parallelism, adding even more processors makes the speedup less perfect.

As an example, consider the computation P-FIB(4) in Figure 26.2, and assume that each strand takes unit time. Since the work is $T_1 = 17$ and the span is $T_\infty = 8$, the parallelism is $T_1/T_\infty = 17/8 = 2.125$. Consequently, achieving much more than double the performance is impossible, no matter how many processors execute the computation. For larger input sizes, however, we'll see that P-FIB(n) exhibits substantial parallelism.

We define the ***(parallel) slackness*** of a task-parallel computation executed on an ideal parallel computer with P processors to be the ratio $(T_1/T_\infty)/P = T_1/(PT_\infty)$, which is the factor by which the parallelism of the computation exceeds the number of processors in the machine. Restating the bounds on speedup, if the slackness is less than 1, perfect linear speedup is impossible, because $T_1/(PT_\infty) < 1$ and the span law imply that $T_1/T_P \leq T_1/T_\infty < P$. Indeed, as the slackness decreases from 1 and approaches 0, the speedup of the computation diverges further and further from perfect linear speedup. If the slackness is less than 1, additional parallelism in an algorithm can have a great impact on its

execution efficiency. If the slackness is greater than 1, however, the work per processor is the limiting constraint. We'll see that as the slackness increases from 1, a good scheduler can achieve closer and closer to perfect linear speedup. But once the slackness is much greater than 1, the advantage of additional parallelism shows diminishing returns.

Scheduling

Good performance depends on more than just minimizing the work and span. The strands must also be scheduled efficiently onto the processors of the parallel machine. Our fork-join parallel-programming model provides no way for a programmer to specify which strands to execute on which processors. Instead, we rely on the runtime system's scheduler to map the dynamically unfolding computation to individual processors. In practice, the scheduler maps the strands to static threads, and the operating system schedules the threads on the processors themselves. But this extra level of indirection is unnecessary for our understanding of scheduling. We can just imagine that the scheduler maps strands to processors directly.

A task-parallel scheduler must schedule the computation without knowing in advance when procedures will be spawned or when they will finish—that is, it must operate ***online***. Moreover, a good scheduler operates in a distributed fashion, where the threads implementing the scheduler cooperate to load-balance the computation. Provably good online, distributed schedulers exist, but analyzing them is complicated. Instead, to keep our analysis simple, we'll consider an online ***centralized*** scheduler that knows the global state of the computation at any moment.

In particular, we'll analyze ***greedy schedulers***, which assign as many strands to processors as possible in each time step, never leaving a processor idle if there is work that can be done. We'll classify each step of a greedy scheduler as follows:

- ***Complete step***: At least P strands are ***ready*** to execute, meaning that all strands on which they depend have finished execution. A greedy scheduler assigns any P of the ready strands to the processors, completely utilizing all the processor resources.
- ***Incomplete step***: Fewer than P strands are ready to execute. A greedy scheduler assigns each ready strand to its own processor, leaving some processors idle for the step, but executing all the ready strands.

The work law tells us that the fastest running time T_P that we can hope for on P processors must be at least T_1/P. The span law tells us that the fastest possible running time must be at least T_∞. The following theorem shows that greedy scheduling is provably good in that it achieves the sum of these two lower bounds as an upper bound.

Theorem 26.1
On an ideal parallel computer with P processors, a greedy scheduler executes a task-parallel computation with work T_1 and span T_∞ in time

$$T_P \le T_1/P + T_\infty . \tag{26.4}$$

Proof Without loss of generality, assume that each strand takes unit time. (If necessary, replace each longer strand by a chain of unit-time strands.) We'll consider complete and incomplete steps separately.

In each complete step, the P processors together perform a total of P work. Thus, if the number of complete steps is k, the total work executing all the complete steps is kP. Since the greedy scheduler doesn't execute any strand more than once and only T_1 work needs to be performed, it follows that $kP \le T_1$, from which we can conclude that the number k of complete steps is at most T_1/P.

Now, let's consider an incomplete step. Let G be the trace for the entire computation, let G' be the subtrace of G that has yet to be executed at the start of the incomplete step, and let G'' be the subtrace remaining to be executed after the incomplete step. Consider the set R of strands that are ready at the beginning of the incomplete step, where $|R| < P$. By definition, if a strand is ready, all its predecessors in trace G have executed. Thus the predecessors of strands in R do not belong to G'. A longest path in G' must necessarily start at a strand in R, since every other strand in G' has a predecessor and thus could not start a longest path. Because the greedy scheduler executes all ready strands during the incomplete step, the strands of G'' are exactly those in G' minus the strands in R. Consequently, the length of a longest path in G'' must be 1 less than the length of a longest path in G'. In other words, every incomplete step decreases the span of the trace remaining to be executed by 1. Hence, the number of incomplete steps can be at most T_∞.

Since each step is either complete or incomplete, the theorem follows. ■

The following corollary shows that a greedy scheduler always performs well.

Corollary 26.2
The running time T_P of any task-parallel computation scheduled by a greedy scheduler on a P-processor ideal parallel computer is within a factor of 2 of optimal.

Proof Let T_P^* be the running time produced by an optimal scheduler on a machine with P processors, and let T_1 and T_∞ be the work and span of the computation, respectively. Since the work and span laws—inequalities (26.2) and (26.3)—give $T_P^* \ge \max\{T_1/P, T_\infty\}$, Theorem 26.1 implies that

$$\begin{aligned} T_P &\le T_1/P + T_\infty \\ &\le 2 \cdot \max\{T_1/P, T_\infty\} \\ &\le 2T_P^* . \end{aligned}$$

■

The next corollary shows that, in fact, a greedy scheduler achieves near-perfect linear speedup on any task-parallel computation as the slackness grows.

Corollary 26.3
Let T_P be the running time of a task-parallel computation produced by a greedy scheduler on an ideal parallel computer with P processors, and let T_1 and T_∞ be the work and span of the computation, respectively. Then, if $P \ll T_1/T_\infty$, or equivalently, the parallel slackness is much greater than 1, we have $T_P \approx T_1/P$, a speedup of approximately P.

Proof If we suppose that $P \ll T_1/T_\infty$, then it follows that $T_\infty \ll T_1/P$, and hence Theorem 26.1 gives $T_P \le T_1/P + T_\infty \approx T_1/P$. Since the work law (26.2) dictates that $T_P \ge T_1/P$, we conclude that $T_P \approx T_1/P$, which is a speedup of $T_1/T_P \approx P$. ■

The $\ll$ symbol denotes "much less," but how much is "much less"? As a rule of thumb, a slackness of at least 10—that is, 10 times more parallelism than processors—generally suffices to achieve good speedup. Then, the span term in the greedy bound, inequality (26.4), is less than 10% of the work-per-processor term, which is good enough for most engineering situations. For example, if a computation runs on only 10 or 100 processors, it doesn't make sense to value parallelism of, say 1,000,000, over parallelism of 10,000, even with the factor of 100 difference. As Problem 26-2 shows, sometimes reducing extreme parallelism yields algorithms that are better with respect to other concerns and which still scale up well on reasonable numbers of processors.

Analyzing parallel algorithms

We now have all the tools we need to analyze parallel algorithms using work/span analysis, allowing us to bound an algorithm's running time on any number of processors. Analyzing the work is relatively straightforward, since it amounts to nothing more than analyzing the running time of an ordinary serial algorithm, namely, the serial projection of the parallel algorithm. You should already be familiar with analyzing work, since that is what most of this textbook is about! Analyzing the span is the new thing that parallelism engenders, but it's generally no harder once you get the hang of it. Let's investigate the basic ideas using the P-FIB program.

Analyzing the work $T_1(n)$ of P-FIB(n) poses no hurdles, because we've already done it. The serial projection of P-FIB is effectively the original FIB procedure, and hence, we have $T_1(n) = T(n) = \Theta(\phi^n)$ from equation (26.1).

Figure 26.3 illustrates how to analyze the span. If two traces are joined in series, their spans add to form the span of their composition, whereas if they are joined

Figure 26.3 Series-parallel composition of parallel traces. **(a)** When two traces are joined in series, the work of the composition is the sum of their work, and the span of the composition is the sum of their spans. **(b)** When two traces are joined in parallel, the work of the composition remains the sum of their work, but the span of the composition is only the maximum of their spans.

in parallel, the span of their composition is the maximum of the spans of the two traces. As it turns out, the trace of any fork-join parallel computation can be built up from single strands by series-parallel composition.

Armed with an understanding of series-parallel composition, we can analyze the span of P-FIB(n). The spawned call to P-FIB$(n-1)$ in line 3 runs in parallel with the call to P-FIB$(n-2)$ in line 4. Hence, we can express the span of P-FIB(n) as the recurrence

$$\begin{aligned} T_\infty(n) &= \max\{T_\infty(n-1), T_\infty(n-2)\} + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1) , \end{aligned}$$

which has solution $T_\infty(n) = \Theta(n)$. (The second equality above follows from the first because P-FIB$(n-1)$ uses P-FIB$(n-2)$ in its computation, so that the span of P-FIB$(n-1)$ must be at least as large as the span of P-FIB$(n-2)$.)

The parallelism of P-FIB(n) is $T_1(n)/T_\infty(n) = \Theta(\phi^n/n)$, which grows dramatically as n gets large. Thus, Corollary 26.3 tells us that on even the largest parallel computers, a modest value for n suffices to achieve near perfect linear speedup for P-FIB(n), because this procedure exhibits considerable parallel slackness.

Parallel loops

Many algorithms contain loops for which all the iterations can operate in parallel. Although the **spawn** and **sync** keywords can be used to parallelize such loops, it is more convenient to specify directly that the iterations of such loops can run in parallel. Our pseudocode provides this functionality via the **parallel** keyword, which precedes the **for** keyword in a **for** loop statement.

As an example, consider the problem of multiplying a square $n \times n$ matrix $A = (a_{ij})$ by an n-vector $x = (x_j)$. The resulting n-vector $y = (y_i)$ is given by the equation

$$y_i = \sum_{j=1}^{n} a_{ij} x_j ,$$

for $i = 1, 2, \ldots, n$. The P-MAT-VEC procedure performs matrix-vector multiplication (actually, $y = y + Ax$) by computing all the entries of y in parallel. The **parallel for** keywords in line 1 of P-MAT-VEC indicate that the n iterations of the loop body, which includes a serial **for** loop, may be run in parallel. The initialization $y = 0$, if desired, should be performed before calling the procedure (and can be done with a **parallel for** loop).

```
P-MAT-VEC(A, x, y, n)
1  parallel for i = 1 to n          // parallel loop
2      for j = 1 to n               // serial loop
3          y_i = y_i + a_ij x_j
```

Compilers for fork-join parallel programs can implement **parallel for** loops in terms of **spawn** and **sync** by using recursive spawning. For example, for the **parallel for** loop in lines 1–3, a compiler can generate the auxiliary subroutine P-MAT-VEC-RECURSIVE and call P-MAT-VEC-RECURSIVE$(A, x, y, n, 1, n)$ in the place where the loop would be in the compiled code. As Figure 26.4 illustrates, this procedure recursively spawns the first half of the iterations of the loop to execute in parallel (line 5) with the second half of the iterations (line 6) and then executes a **sync** (line 7), thereby creating a binary tree of parallel execution. Each leaf represents a base case, which is the serial **for** loop of lines 2–3.

```
P-MAT-VEC-RECURSIVE(A, x, y, n, i, i')
1  if i == i'                       // just one iteration to do?
2      for j = 1 to n               // mimic P-MAT-VEC serial loop
3          y_i = y_i + a_ij x_j
4  else mid = ⌊(i + i')/2⌋          // parallel divide-and-conquer
5      spawn P-MAT-VEC-RECURSIVE(A, x, y, n, i, mid)
6      P-MAT-VEC-RECURSIVE(A, x, y, n, mid + 1, i')
7      sync
```

To calculate the work $T_1(n)$ of P-MAT-VEC on an $n \times n$ matrix, simply compute the running time of its serial projection, which comes from replacing the **parallel**

Figure 26.4 A trace for the computation of P-MAT-VEC-RECURSIVE$(A, x, y, 8, 1, 8)$. The two numbers within each rounded rectangle give the values of the last two parameters (i and i' in the procedure header) in the invocation (spawn, in blue, or call, in tan) of the procedure. The blue circles represent strands corresponding to the part of the procedure up to the spawn of P-MAT-VEC-RECURSIVE in line 5. The orange circles represent strands corresponding to the part of the procedure that calls P-MAT-VEC-RECURSIVE in line 6 up to the **sync** in line 7, where it suspends until the spawned subroutine in line 5 returns. The white circles represent strands corresponding to the (negligible) part of the procedure after the **sync** up to the point where it returns.

for loop in line 1 with an ordinary **for** loop. The running time of the resulting serial pseudocode is $\Theta(n^2)$, which means that $T_1(n) = \Theta(n^2)$. This analysis seems to ignore the overhead for recursive spawning in implementing the parallel loops, however. Indeed, the overhead of recursive spawning does increase the work of a parallel loop compared with that of its serial projection, but not asymptotically. To see why, observe that since the tree of recursive procedure instances is a full binary tree, the number of internal nodes is one less than the number of leaves (see Exercise B.5-3 on page 1175). Each internal node performs constant work to divide the iteration range, and each leaf corresponds to a base case, which takes at least constant time ($\Theta(n)$ time in this case). Thus, by amortizing the overhead of recursive spawning over the work of the iterations in the leaves, we see that the overall work increases by at most a constant factor.

To reduce the overhead of recursive spawning, task-parallel platforms sometimes ***coarsen*** the leaves of the recursion by executing several iterations in a single leaf, either automatically or under programmer control. This optimization comes at the expense of reducing the parallelism. If the computation has sufficient parallel slackness, however, near-perfect linear speedup won't be sacrificed.

Although recursive spawning doesn't affect the work of a parallel loop asymptotically, we must take it into account when analyzing the span. Consider a parallel loop with n iterations in which the ith iteration has span $iter_\infty(i)$. Since the depth of recursion is logarithmic in the number of iterations, the parallel loop's span is

$$T_\infty(n) = \Theta(\lg n) + \max\{iter_\infty(i) : 1 \le i \le n\} .$$

For example, let's compute the span of the doubly nested loops in lines 1–3 of P-MAT-VEC. The span for the **parallel for** loop control is $\Theta(\lg n)$. For each iteration of the outer parallel loop, the inner serial **for** loop contains n iterations of line 3. Since each iteration takes constant time, the total span for the inner serial **for** loop is $\Theta(n)$, no matter which iteration of the outer **parallel for** loop it's in. Thus, taking the maximum over all iterations of the outer loop and adding in the $\Theta(\lg n)$ for loop control yields an overall span of $T_\infty n = \Theta(n) + \Theta(\lg n) = \Theta(n)$ for the procedure. Since the work is $\Theta(n^2)$, the parallelism is $\Theta(n^2)/\Theta(n) = \Theta(n)$. (Exercise 26.1-7 asks you to provide an implementation with even more parallelism.)

Race conditions

A parallel algorithm is ***deterministic*** if it always does the same thing on the same input, no matter how the instructions are scheduled on the multicore computer. It is ***nondeterministic*** if its behavior might vary from run to run when the input is the same. A parallel algorithm that is intended to be deterministic may nevertheless act nondeterministically, however, if it contains a difficult-to-diagnose bug called a "determinacy race."

Famous race bugs include the Therac-25 radiation therapy machine, which killed three people and injured several others, and the Northeast Blackout of 2003, which left over 50 million people in the United States without power. These pernicious bugs are notoriously hard to find. You can run tests in the lab for days without a failure, only to discover that your software sporadically crashes in the field, sometimes with dire consequences.

A ***determinacy race*** occurs when two logically parallel instructions access the same memory location and at least one of the instructions modifies the value stored in the location. The toy procedure RACE-EXAMPLE on the following page illustrates a determinacy race. After initializing x to 0 in line 1, RACE-EXAMPLE creates two parallel strands, each of which increments x in line 3. Although it might seem that a call of RACE-EXAMPLE should always print the value 2 (its serial projection certainly does), it could instead print the value 1. Let's see how this anomaly might occur.

When a processor increments x, the operation is not indivisible, but is composed of a sequence of instructions:

(a)

step	x	r_1	r_2
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

(b)

Figure 26.5 Illustration of the determinacy race in RACE-EXAMPLE. **(a)** A trace showing the dependencies among individual instructions. The processor registers are r_1 and r_2. Instructions unrelated to the race, such as the implementation of loop control, are omitted. **(b)** An execution sequence that elicits the bug, showing the values of x in memory and registers r_1 and r_2 for each step in the execution sequence.

```
RACE-EXAMPLE()
x = 0
parallel for i = 1 to 2
    x = x + 1            // determinacy race
print x
```

- Load x from memory into one of the processor's registers.
- Increment the value in the register.
- Store the value in the register back into x in memory.

Figure 26.5(a) illustrates a trace representing the execution of RACE-EXAMPLE, with the strands broken down to individual instructions. Recall that since an ideal parallel computer supports sequential consistency, you can view the parallel execution of a parallel algorithm as an interleaving of instructions that respects the dependencies in the trace. Part (b) of the figure shows the values in an execution of the computation that elicits the anomaly. The value x is kept in memory, and r_1 and r_2 are processor registers. In step 1, one of the processors sets x to 0. In steps 2 and 3, processor 1 loads x from memory into its register r_1 and increments it, producing the value 1 in r_1. At that point, processor 2 comes into the picture, executing instructions 4–6. Processor 2 loads x from memory into register r_2; increments it, producing the value 1 in r_2; and then stores this value into x, setting x to 1. Now, processor 1 resumes with step 7, storing the value 1 in r_1 into x, which

leaves the value of x unchanged. Therefore, step 8 prints the value 1, rather than the value 2 that the serial projection would print.

Let's recap what happened. By sequential consistency, the effect of the parallel execution is as if the executed instructions of the two processors are interleaved. If processor 1 executes all its instructions before processor 2, a trivial interleaving, the value 2 is printed. Conversely, if processor 2 executes all its instructions before processor 1, the value 2 is still printed. When the instructions of the two processors interleave nontrivially, however, it is possible, as in this example execution, that one of the updates to x is lost, resulting in the value 1 being printed.

Of course, many executions do not elicit the bug. That's the problem with determinacy races. Generally, most instruction orderings produce correct results, such as any where the instructions on the left branch execute before the instructions on the right branch, or vice versa. But some orderings generate improper results when the instructions interleave. Consequently, races can be extremely hard to test for. Your program may fail, but you may be unable to reliably reproduce the failure in subsequent tests, confounding your attempts to locate the bug in your code and fix it. Task-parallel programming environments often provide race-detection productivity tools to help you isolate race bugs.

Many parallel programs in the real world are intentionally nondeterministic. They contain determinacy races, but they mitigate the dangers of nondeterminism through the use of mutual-exclusion locks and other methods of synchronization. For our purposes, however, we'll insist on an absence of determinacy races in the algorithms we develop. Nondeterministic programs are indeed interesting, but nondeterministic programming is a more advanced topic and unnecessary for a wide swath of interesting parallel algorithms.

To ensure that algorithms are deterministic, any two strands that operate in parallel should be ***mutually noninterfering***: they only read, and do not modify, any memory locations accessed by both of them. Consequently, in a **parallel for** construct, such as the outer loop of P-MAT-VEC, we want all the iterations of the body, including any code an iteration executes in subroutines, to be mutually noninterfering. And between a **spawn** and its corresponding **sync**, we want the code executed by the spawned child and the code executed by the parent to be mutually noninterfering, once again including invoked subroutines.

As an example of how easy it is to write code with unintentional races, the P-MAT-VEC-WRONG procedure on the next page is a faulty parallel implementation of matrix-vector multiplication that achieves a span of $\Theta(\lg n)$ by parallelizing the inner **for** loop. This procedure is incorrect, unfortunately, due to determinacy races when updating y_i in line 3, which executes in parallel for all n values of j.

Index variables of **parallel for** loops, such as i in line 1 and j in line 2, do not cause races between iterations. Conceptually, each iteration of the loop creates an independent variable to hold the index of that iteration during that iteration's

```
P-MAT-VEC-WRONG(A, x, y, n)
parallel for i = 1 to n
    parallel for j = 1 to n
        y_i = y_i + a_ij x_j        // determinacy race
```

execution of the loop body. Even if two parallel iterations both access the same index variable, they really are accessing different variable instances—hence different memory locations—and no race occurs.

A parallel algorithm with races can sometimes be deterministic. As an example, two parallel threads might store the same value into a shared variable, and it wouldn't matter which stored the value first. For simplicity, however, we generally prefer code without determinacy races, even if the races are benign. And good parallel programmers frown on code with determinacy races that cause nondeterministic behavior, if deterministic code that performs comparably is an option.

But nondeterministic code does have its place. For example, you can't implement a parallel hash table, a highly practical data structure, without writing code containing determinacy races. Much research has centered around how to extend the fork-join model to incorporate limited "structured" nondeterminism while avoiding the full measure of complications that arise when nondeterminism is completely unrestricted.

A chess lesson

To illustrate the power of work/span analysis, this section closes with a true story that occurred during the development of one of the first world-class parallel chess-playing programs [106] many years ago. The timings below have been simplified for exposition.

The chess program was developed and tested on a 32-processor computer, but it was designed to run on a supercomputer with 512 processors. Since the supercomputer availability was limited and expensive, the developers ran benchmarks on the small computer and extrapolated performance to the large computer.

At one point, the developers incorporated an optimization into the program that reduced its running time on an important benchmark on the small machine from $T_{32} = 65$ seconds to $T'_{32} = 40$ seconds. Yet, the developers used the work and span performance measures to conclude that the optimized version, which was faster on 32 processors, would actually be slower than the original version on the 512 processors of the large machine. As a result, they abandoned the "optimization."

Here is their work/span analysis. The original version of the program had work $T_1 = 2048$ seconds and span $T_\infty = 1$ second. Let's treat inequality (26.4) on

page 760 as the equation $T_P = T_1/P + T_\infty$, which we can use as an approximation to the running time on P processors. Then indeed we have $T_{32} = 2048/32 + 1 = 65$. With the optimization, the work becomes $T'_1 = 1024$ seconds, and the span becomes $T'_\infty = 8$ seconds. Our approximation gives $T'_{32} = 1024/32 + 8 = 40$.

The relative speeds of the two versions switch when we estimate their running times on 512 processors, however. The first version has a running time of $T_{512} = 2048/512+1 = 5$ seconds, and the second version runs in $T'_{512} = 1024/512+8 = 10$ seconds. The optimization that speeds up the program on 32 processors makes the program run for twice as long on 512 processors! The optimized version's span of 8, which is not the dominant term in the running time on 32 processors, becomes the dominant term on 512 processors, nullifying the advantage from using more processors. The optimization does not scale up.

The moral of the story is that work/span analysis, and measurements of work and span, can be superior to measured running times alone in extrapolating an algorithm's scalability.

Exercises

26.1-1

What does a trace for the execution of a serial algorithm look like?

26.1-2

Suppose that line 4 of P-FIB spawns P-FIB$(n-2)$, rather than calling it as is done in the pseudocode. How would the trace of P-FIB(4) in Figure 26.2 change? What is the impact on the asymptotic work, span, and parallelism?

26.1-3

Draw the trace that results from executing P-FIB(5). Assuming that each strand in the computation takes unit time, what are the work, span, and parallelism of the computation? Show how to schedule the trace on 3 processors using greedy scheduling by labeling each strand with the time step in which it is executed.

26.1-4

Prove that a greedy scheduler achieves the following time bound, which is slightly stronger than the bound proved in Theorem 26.1:

$$T_P \le \frac{T_1 - T_\infty}{P} + T_\infty \,. \tag{26.5}$$

26.1-5

Construct a trace for which one execution by a greedy scheduler can take nearly twice the time of another execution by a greedy scheduler on the same number of processors. Describe how the two executions would proceed.

26.1-6

Professor Karan measures her deterministic task-parallel algorithm on 4, 10, and 64 processors of an ideal parallel computer using a greedy scheduler. She claims that the three runs yielded $T_4 = 80$ seconds, $T_{10} = 42$ seconds, and $T_{64} = 10$ seconds. Argue that the professor is either lying or incompetent. (*Hint:* Use the work law (26.2), the span law (26.3), and inequality (26.5) from Exercise 26.1-4.)

26.1-7

Give a parallel algorithm to multiply an $n \times n$ matrix by an n-vector that achieves $\Theta(n^2/\lg n)$ parallelism while maintaining $\Theta(n^2)$ work.

26.1-8

Analyze the work, span, and parallelism of the procedure P-TRANSPOSE, which transposes an $n \times n$ matrix A in place.

```
P-TRANSPOSE(A, n)
1  parallel for j = 2 to n
2      parallel for i = 1 to j - 1
3          exchange a_ij with a_ji
```

26.1-9

Suppose that instead of a **parallel for** loop in line 2, the P-TRANSPOSE procedure in Exercise 26.1-8 had an ordinary **for** loop. Analyze the work, span, and parallelism of the resulting algorithm.

26.1-10

For what number of processors do the two versions of the chess program run equally fast, assuming that $T_P = T_1/P + T_\infty$?

26.2 Parallel matrix multiplication

In this section, we'll explore how to parallelize the three matrix-multiplication algorithms from Sections 4.1 and 4.2. We'll see that each algorithm can be parallelized in a straightforward fashion using either parallel loops or recursive spawning. We'll analyze them using work/span analysis, and we'll see that each parallel algorithm attains the same performance on one processor as its corresponding serial algorithm, while scaling up to large numbers of processors.

A parallel algorithm for matrix multiplication using parallel loops

The first algorithm we'll study is P-MATRIX-MULTIPLY, which simply parallelizes the two outer loops in the procedure MATRIX-MULTIPLY on page 81.

```
P-MATRIX-MULTIPLY(A, B, C, n)
1  parallel for i = 1 to n                  // compute entries in each of n rows
2      parallel for j = 1 to n              // compute n entries in row i
3          for k = 1 to n
4              c_ij = c_ij + a_ik · b_kj    // add in another term of equation (4.1)
```

Let's analyze P-MATRIX-MULTIPLY. Since the serial projection of the algorithm is just MATRIX-MULTIPLY, the work is the same as the running time of MATRIX-MULTIPLY: $T_1(n) = \Theta(n^3)$. The span is $T_\infty(n) = \Theta(n)$, because it follows a path down the tree of recursion for the **parallel for** loop starting in line 1, then down the tree of recursion for the **parallel for** loop starting in line 2, and then executes all n iterations of the ordinary **for** loop starting in line 3, resulting in a total span of $\Theta(\lg n) + \Theta(\lg n) + \Theta(n) = \Theta(n)$. Thus the parallelism is $\Theta(n^3)/\Theta(n) = \Theta(n^2)$. (Exercise 26.2-3 asks you to parallelize the inner loop to obtain a parallelism of $\Theta(n^3/\lg n)$, which you cannot do straightforwardly using **parallel for**, because you would create races.)

A parallel divide-and-conquer algorithm for matrix multiplication

Section 4.1 shows how to multiply $n \times n$ matrices serially in $\Theta(n^3)$ time using a divide-and-conquer strategy. Let's see how to parallelize that algorithm using recursive spawning instead of calls.

The serial MATRIX-MULTIPLY-RECURSIVE procedure on page 83 takes as input three $n \times n$ matrices A, B, and C and performs the matrix calculation $C = C + A \cdot B$ by recursively performing eight multiplications of $n/2 \times n/2$ submatrices of A and B. The P-MATRIX-MULTIPLY-RECURSIVE procedure on the following page implements the same divide-and-conquer strategy, but it uses spawning to perform the eight multiplications in parallel. To avoid determinacy races in updating the elements of C, it creates a temporary matrix D to store four of the submatrix products. At the end, it adds C and D together to produce the final result. (Problem 26-2 asks you to eliminate the temporary matrix D at the expense of some parallelism.)

Lines 2–3 of P-MATRIX-MULTIPLY-RECURSIVE handle the base case of multiplying 1×1 matrices. The remainder of the procedure deals with the recursive case. Line 4 allocates a temporary matrix D, and lines 5–7 zero it. Line 8 partitions each of the four matrices A, B, C, and D into $n/2 \times n/2$ submatrices. (As

```
P-MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
1  if n == 1                          // just one element in each matrix?
2      c11 = c11 + a11 · b11
3      return
4  let D be a new n × n matrix        // temporary matrix
5  parallel for i = 1 to n            // set D = 0
6      parallel for j = 1 to n
7          dij = 0
8  partition A, B, C, and D into n/2 × n/2 submatrices
       A11, A12, A21, A22; B11, B12, B21, B22; C11, C12, C21, C22;
       and D11, D12, D21, D22; respectively
9  spawn P-MATRIX-MULTIPLY-RECURSIVE(A11, B11, C11, n/2)
10 spawn P-MATRIX-MULTIPLY-RECURSIVE(A11, B12, C12, n/2)
11 spawn P-MATRIX-MULTIPLY-RECURSIVE(A21, B11, C21, n/2)
12 spawn P-MATRIX-MULTIPLY-RECURSIVE(A21, B12, C22, n/2)
13 spawn P-MATRIX-MULTIPLY-RECURSIVE(A12, B21, D11, n/2)
14 spawn P-MATRIX-MULTIPLY-RECURSIVE(A12, B22, D12, n/2)
15 spawn P-MATRIX-MULTIPLY-RECURSIVE(A22, B21, D21, n/2)
16 spawn P-MATRIX-MULTIPLY-RECURSIVE(A22, B22, D22, n/2)
17 sync                               // wait for spawned submatrix products
18 parallel for i = 1 to n            // update C = C + D
19     parallel for j = 1 to n
20         cij = cij + dij
```

with MATRIX-MULTIPLY-RECURSIVE on page 83, we're glossing over the subtle issue of how to use index calculations to represent submatrix sections of a matrix.) The spawned recursive call in line 9 sets $C_{11} = C_{11} + A_{11} \cdot B_{11}$, so that C_{11} accumulates the first of the two terms in equation (4.5) on page 82. Similarly, lines 10–12 cause each of C_{12}, C_{21}, and C_{22} in parallel to accumulate the first of the two terms in equations (4.6)–(4.8), respectively. Line 13 sets the submatrix D_{11} to the submatrix product $A_{12} \cdot B_{21}$, so that D_{11} equals the second of the two terms in equation (4.5). Lines 14–16 set each of D_{12}, D_{21}, and D_{22} in parallel to the second of the two terms in equations (4.6)–(4.8), respectively. The **sync** statement in line 17 ensures that all the spawned submatrix products in lines 9–16 have been computed, after which the doubly nested **parallel for** loops in lines 18–20 add the elements of D to the corresponding elements of C.

Let's analyze the P-MATRIX-MULTIPLY-RECURSIVE procedure. We start by analyzing the work $M_1(n)$, echoing the serial running-time analysis of its progenitor MATRIX-MULTIPLY-RECURSIVE. The recursive case allocates and zeros the

temporary matrix D in $\Theta(n^2)$ time, partitions in $\Theta(1)$ time, performs eight recursive multiplications of $n/2 \times n/2$ matrices, and finishes up with the $\Theta(n^2)$ work from adding two $n \times n$ matrices. Thus the work outside the spawned recursive calls is $\Theta(n^2)$, and the recurrence for the work $M_1(n)$ becomes

$$\begin{aligned} M_1(n) &= 8M_1(n/2) + \Theta(n^2) \\ &= \Theta(n^3) \end{aligned}$$

by case 1 of the master theorem (Theorem 4.1). Not surprisingly, the work of this parallel algorithm is asymptotically the same as the running time of the procedure MATRIX-MULTIPLY on page 81, with its triply nested loops.

Let's determine the span $M_\infty(n)$ of P-MATRIX-MULTIPLY-RECURSIVE. Because the eight parallel recursive spawns all execute on matrices of the same size, the maximum span for any recursive spawn is just the span of a single one of them, or $M_\infty(n/2)$. The span for the doubly nested **parallel for** loops in lines 5–7 is $\Theta(\lg n)$ because each loop control adds $\Theta(\lg n)$ to the constant span of line 7. Similarly, the doubly nested **parallel for** loops in lines 18–20 add another $\Theta(\lg n)$. Matrix partitioning by index calculation has $\Theta(1)$ span, which is dominated by the $\Theta(\lg n)$ span of the nested loops. We obtain the recurrence

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) \; . \tag{26.6}$$

Since this recurrence falls under case 2 of the master theorem with $k = 1$, the solution is $M_\infty(n) = \Theta(\lg^2 n)$.

The parallelism of P-MATRIX-MULTIPLY-RECURSIVE is $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$, which is huge. (Problem 26-2 asks you to simplify this parallel algorithm at the expense of just a little less parallelism.)

Parallelizing Strassen's method

To parallelize Strassen's algorithm, we can follow the same general outline as on pages 86–87, but use spawning. You may find it helpful to compare each step below with the corresponding step there. We'll analyze costs as we go along to develop recurrences $T_1(n)$ and $T_\infty(n)$ for the overall work and span, respectively.

1. If $n = 1$, the matrices each contain a single element. Perform a single scalar multiplication and a single scalar addition, and return. Otherwise, partition the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.2) on page 82. This step takes $\Theta(1)$ work and $\Theta(1)$ span by index calculation.

2. Create $n/2 \times n/2$ matrices $S_1, S_2, \ldots, S_{10}$, each of which is the sum or difference of two submatrices from step 1. Create and zero the entries of seven $n/2 \times n/2$ matrices $P_1, P_2, \ldots, P_7$ to hold seven $n/2 \times n/2$ matrix products. All

17 matrices can be created, and the P_i initialized, with doubly nested **parallel for** loops using $\Theta(n^2)$ work and $\Theta(\lg n)$ span.

3. Using the submatrices from step 1 and the matrices $S_1, S_2, \ldots, S_{10}$ created in step 2, recursively spawn computations of each of the seven $n/2 \times n/2$ matrix products $P_1, P_2, \ldots, P_7$, taking $7T_1(n/2)$ work and $T_\infty(n/2)$ span.
4. Update the four submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding or subtracting various P_i matrices. Using doubly nested **parallel for** loops, computing all four submatrices takes $\Theta(n^2)$ work and $\Theta(\lg n)$ span.

Let's analyze this algorithm. Since the serial projection is the same as the original serial algorithm, the work is just the running time of the serial projection, namely, $\Theta(n^{\lg 7})$. As we did with P-MATRIX-MULTIPLY-RECURSIVE, we can devise a recurrence for the span. In this case, seven recursive calls execute in parallel, but since they all operate on matrices of the same size, we obtain the same recurrence (26.6) as we did for P-MATRIX-MULTIPLY-RECURSIVE, with solution $\Theta(\lg^2 n)$. Thus the parallel version of Strassen's method has parallelism $\Theta(n^{\lg 7}/\lg^2 n)$, which is large. Although the parallelism is slightly less than that of P-MATRIX-MULTIPLY-RECURSIVE, that's just because the work is also less.

Exercises

26.2-1

Draw the trace for computing P-MATRIX-MULTIPLY on 2×2 matrices, labeling how the vertices in your diagram correspond to strands in the execution of the algorithm. Assuming that each strand executes in unit time, analyze the work, span, and parallelism of this computation.

26.2-2

Repeat Exercise 26.2-1 for P-MATRIX-MULTIPLY-RECURSIVE.

26.2-3

Give pseudocode for a parallel algorithm that multiplies two $n \times n$ matrices with work $\Theta(n^3)$ but span only $\Theta(\lg n)$. Analyze your algorithm.

26.2-4

Give pseudocode for an efficient parallel algorithm that multiplies a $p \times q$ matrix by a $q \times r$ matrix. Your algorithm should be highly parallel even if any of p, q, and r equal 1. Analyze your algorithm.

26.2-5
Give pseudocode for an efficient parallel version of the Floyd-Warshall algorithm (see Section 23.2), which computes shortest paths between all pairs of vertices in an edge-weighted graph. Analyze your algorithm.

26.3 Parallel merge sort

We first saw serial merge sort in Section 2.3.1, and in Section 2.3.2 we analyzed its running time and showed it to be $\Theta(n \lg n)$. Because merge sort already uses the divide-and-conquer method, it seems like a terrific candidate for implementing using fork-join parallelism.

The procedure P-MERGE-SORT modifies merge sort to spawn the first recursive call. Like its serial counterpart MERGE-SORT on page 39, the P-MERGE-SORT procedure sorts the subarray $A[p:r]$. After the **sync** statement in line 8 ensures that the two recursive spawns in lines 5 and 7 have finished, P-MERGE-SORT calls the P-MERGE procedure, a parallel merging algorithm, which is on page 779, but you don't need to bother looking at it right now.

```
P-MERGE-SORT(A, p, r)
1  if p ≥ r                    // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋             // midpoint of A[p : r]
4  // Recursively sort A[p : q] in parallel.
5  spawn P-MERGE-SORT(A, p, q)
6  // Recursively sort A[q + 1 : r] in parallel.
7  spawn P-MERGE-SORT(A, q + 1, r)
8  sync                        // wait for spawns
9  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
10 P-MERGE(A, p, q, r)
```

First, let's use work/span analysis to get some intuition for why we need a parallel merge procedure. After all, it may seem as though there should be plenty of parallelism just by parallelizing MERGE-SORT without worrying about parallelizing the merge. But what would happen if the call to P-MERGE in line 10 of P-MERGE-SORT were replaced by a call to the serial MERGE procedure on page 36? Let's call the pseudocode so modified P-NAIVE-MERGE-SORT.

Let $T_1(n)$ be the (worst-case) work of P-NAIVE-MERGE-SORT on an n-element subarray, where $n = r - p + 1$ is the number of elements in $A[p:r]$, and let $T_\infty(n)$

be the span. Because MERGE is serial with running time $\Theta(n)$, both its work and span are $\Theta(n)$. Since the serial projection of P-NAIVE-MERGE-SORT is exactly MERGE-SORT, its work is $T_1(n) = \Theta(n \lg n)$. The two recursive calls in lines 5 and 7 run in parallel, and so its span is given by the recurrence

$$\begin{aligned} T_\infty(n) &= T_\infty(n/2) + \Theta(n) \\ &= \Theta(n) , \end{aligned}$$

by case 1 of the master theorem. Thus the parallelism of P-NAIVE-MERGE-SORT is $T_1(n)/T_\infty(n) = \Theta(\lg n)$, which is an unimpressive amount of parallelism. To sort a million elements, for example, since $\lg 10^6 \approx 20$, it might achieve linear speedup on a few processors, but it would not scale up to dozens of processors.

The parallelism bottleneck in P-NAIVE-MERGE-SORT is plainly the MERGE procedure. If we asymptotically reduce the span of merging, the master theorem dictates that the span of parallel merge sort will also get smaller. When you look at the pseudocode for MERGE, it may seem that merging is inherently serial, but it's not. We can fashion a parallel merging algorithm. The goal is to reduce the span of parallel merging asymptotically, but if we want an efficient parallel algorithm, we must ensure that the $\Theta(n)$ bound on work doesn't increase.

Figure 26.6 depicts the divide-and-conquer strategy that we'll use in P-MERGE. The heart of the algorithm is a recursive auxiliary procedure P-MERGE-AUX that merges two sorted subarrays of an array A into a subarray of another array B in parallel. Specifically, P-MERGE-AUX merges $A[p_1 : r_1]$ and $A[p_2 : r_2]$ into subarray $B[p_3 : r_3]$, where $r_3 = p_3 + (r_1 - p_1 + 1) + (r_2 - p_2 + 1) - 1 = p_3 + (r_1 - p_1) + (r_2 - p_2) + 1$.

The key idea of the recursive merging algorithm in P-MERGE-AUX is to split each of the two sorted subarrays of A around a pivot x, such that all the elements in the lower part of each subarray are at most x and all the elements in the upper part of each subarray are at least x. The procedure can then recurse in parallel on two subtasks: merging the two lower parts, and merging the two upper parts. The trick is to find a pivot x so that the recursion is not too lopsided. We don't want a situation such as that in QUICKSORT on page 183, where bad partitioning elements lead to a dramatic loss of asymptotic efficiency. We could opt to partition around a random element, as RANDOMIZED-QUICKSORT on page 192 does, but because the input subarrays are sorted, P-MERGE-AUX can quickly determine a pivot that always works well.

Specifically, the recursive merging algorithm picks the pivot x as the middle element of the larger of the two input subarrays, which we can assume without loss of generality is $A[p_1 : r_1]$, since otherwise, the two subarrays can just switch roles. That is, $x = A[q_1]$, where $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$. Because $A[p_1 : r_1]$ is sorted, x is a median of the subarray elements: every element in $A[p_1 : q_1 - 1]$ is no more than x, and every element in $A[q_1 + 1 : r_1]$ is no less than x. Then the

Figure 26.6 The idea behind P-MERGE-AUX, which merges two sorted subarrays $A[p_1 : r_1]$ and $A[p_2 : r_2]$ into the subarray $B[p_3 : r_3]$ in parallel. Letting $x = A[q_1]$ (shown in yellow) be a median of $A[p_1 : r_1]$ and q_2 be a place in $A[p_2 : r_2]$ such that x would fall between $A[q_2 - 1]$ and $A[q_2]$, every element in the subarrays $A[p_1 : q_1 - 1]$ and $A[p_2 : q_2 - 1]$ (shown in orange) is at most x, and every element in the subarrays $A[q_1 + 1 : r_1]$ and $A[q_2 + 1 : r_2]$ (shown in blue) is at least x. To merge, compute the index q_3 where x belongs in $B[p_3 : r_3]$, copy x into $B[q_3]$, and then recursively merge $A[p_1 : q_1 - 1]$ with $A[p_2 : q_2 - 1]$ into $B[p_3 : q_3 - 1]$ and $A[q_1 + 1 : r_1]$ with $A[q_2 : r_2]$ into $B[q_3 + 1 : r_3]$.

algorithm finds the "split point" q_2 in the smaller subarray $A[p_2 : r_2]$ such that all the elements in $A[p_2 : q_2 - 1]$ (if any) are at most x and all the elements in $A[q_2 : r_2]$ (if any) are at least x. Intuitively, the subarray $A[p_2 : r_2]$ would still be sorted if x were inserted between $A[q_2 - 1]$ and $A[q_2]$ (although the algorithm doesn't do that). Since $A[p_2 : r_2]$ is sorted, a minor variant of binary search (see Exercise 2.3-6) with x as the search key can find the split point q_2 in $\Theta(\lg n)$ time in the worst case. As we'll see when we get to the analysis, even if x splits $A[p_2 : r_2]$ badly—x is either smaller than all the subarray elements or larger—we'll still have at least $1/4$ of the elements in each of the two recursive merges. Thus the larger of the recursive merges operates on at most $3/4$ elements, and the recursion is guaranteed to bottom out after $\Theta(\lg n)$ recursive calls.

Now let's put these ideas into pseudocode. We start with the serial procedure FIND-SPLIT-POINT(A, p, r, x) on the next page, which takes as input a sorted subarray $A[p : r]$ and a key x. The procedure returns a split point of $A[p : r]$: an index q in the range $p \le q \le r + 1$ such that all the elements in $A[p : q - 1]$ (if any) are at most x and all the elements in $A[q : r]$ (if any) are at least x.

The FIND-SPLIT-POINT procedure uses binary search to find the split point. Lines 1 and 2 establish the range of indices for the search. Each time through the **while** loop, line 5 compares the middle element of the range with the search key x, and lines 6 and 7 narrow the search range to either the lower half or the upper half of the subarray, depending on the result of the test. In the end, after the range has been narrowed to a single index, line 8 returns that index as the split point.

```
FIND-SPLIT-POINT(A, p, r, x)
low = p                          // low end of search range
high = r + 1                     // high end of search range
while low < high                 // more than one element?
    mid = ⌊(low + high)/2⌋       // midpoint of range
    if x ≤ A[mid]                // is answer q ≤ mid?
        high = mid               // narrow search to A[low : mid]
    else low = mid + 1           // narrow search to A[mid + 1 : high]
return low
```

Because FIND-SPLIT-POINT contains no parallelism, its span is just its serial running time, which is also its work. On a subarray $A[p:r]$ of size $n = r - p + 1$, each iteration of the **while** loop halves the search range, which means that the loop terminates after $\Theta(\lg n)$ iterations. Since each iteration takes constant time, the algorithm runs in $\Theta(\lg n)$ (worst-case) time. Thus the procedure has work and span $\Theta(\lg n)$.

Let's now look at the pseudocode for the parallel merging procedure P-MERGE on the next page. Most of the pseudocode is devoted to the recursive procedure P-MERGE-AUX. The procedure P-MERGE itself is just a "wrapper" that sets up for P-MERGE-AUX. It allocates a new array $B[p:r]$ to hold the output of P-MERGE-AUX in line 1. It then calls P-MERGE-AUX in line 2, passing the indices of the two subarrays to be merged and providing B as the output destination of the merged result, starting at index p. After P-MERGE-AUX returns, lines 3–4 perform a parallel copy of the output $B[p:r]$ into the subarray $A[p:r]$, which is where P-MERGE-SORT expects it.

The P-MERGE-AUX procedure is the interesting part of the algorithm. Let's start by understanding the parameters of this recursive parallel procedure. The input array A and the four indices p_1, r_1, p_2, r_2 specify the subarrays $A[p_1:r_1]$ and $A[p_2:r_2]$ to be merged. The array B and the index p_3 indicate that the merged result should be stored into $B[p_3:r_3]$, where $r_3 = p_3 + (r_1 - p_1) + (r_2 - p_2) + 1$, as we saw earlier. The end index r_3 of the output subarray is not needed by the pseudocode, but it helps conceptually to name the end index, as in the comment in line 13.

The procedure begins by checking the base case of the recursion and doing some bookkeeping to simplify the rest of the pseudocode. Lines 1 and 2 test whether the two subarrays are both empty, in which case the procedure returns. Line 3 checks whether the first subarray contains fewer elements than the second subarray. Since the number of elements in the first subarray is $r_1 - p_1 + 1$ and the number in the second subarray is $r_2 - p_2 + 1$, the test omits the two "+1's." If the first subarray

```
P-MERGE(A, p, q, r)
let B[p : r] be a new array                    // allocate scratch array
P-MERGE-AUX(A, p, q, q + 1, r, B, p)           // merge from A into B
parallel for i = p to r                        // copy B back to A in parallel
    A[i] = B[i]

P-MERGE-AUX(A, p1, r1, p2, r2, B, p3)
1  if p1 > r1 and p2 > r2                      // are both subarrays empty?
2      return
3  if r1 − p1 < r2 − p2                        // second subarray bigger?
4      exchange p1 with p2                     // swap subarray roles
5      exchange r1 with r2
6  q1 = ⌊(p1 + r1)/2⌋                          // midpoint of A[p1 : r1]
7  x = A[q1]                                   // median of A[p1 : r1] is pivot x
8  q2 = FIND-SPLIT-POINT(A, p2, r2, x)         // split A[p2 : r2] around x
9  q3 = p3 + (q1 − p1) + (q2 − p2)             // where x belongs in B ...
10 B[q3] = x                                   // ... put it there
11 // Recursively merge A[p1 : q1 − 1] and A[p2 : q2 − 1] into B[p3 : q3 − 1].
12 spawn P-MERGE-AUX(A, p1, q1 − 1, p2, q2 − 1, B, p3)
13 // Recursively merge A[q1 + 1 : r1] and A[q2 : r2] into B[q3 + 1 : r3].
14 spawn P-MERGE-AUX(A, q1 + 1, r1, q2, r2, B, q3 + 1)
15 sync                                        // wait for spawns
```

is the smaller of the two, lines 4 and 5 switch the roles of the subarrays so that $A[p_1, r_1]$ refers to the larger subarray for the balance of the procedure.

We're now at the crux of P-MERGE-AUX: implementing the parallel divide-and-conquer strategy. As we continue our pseudocode walk, you may find it helpful to refer again to Figure 26.6.

First the divide step. Line 6 computes the midpoint q_1 of $A[p_1 : r_1]$, which indexes a median $x = A[q_1]$ of this subarray to be used as the pivot, and line 7 determines x itself. Next, line 8 uses the FIND-SPLIT-POINT procedure to find the index q_2 in $A[p_2 : r_2]$ such that all elements in $A[p_2 : q_2 - 1]$ are at most x and all the elements in $A[q_2 : r_2]$ are at least x. Line 9 computes the index q_3 of the element that divides the output subarray $B[p_3 : r_3]$ into $B[p_3 : q_3 - 1]$ and $B[q_3 + 1 : r_3]$, and then line 10 puts x directly into $B[q_3]$, which is where it belongs in the output.

Next is the conquer step, which is where the parallel recursion occurs. Lines 12 and 14 each spawn P-MERGE-AUX to recursively merge from A into B, the first to merge the smaller elements and the second to merge the larger elements. The

sync statement in line 15 ensures that the subproblems finish before the procedure returns.

There is no combine step, as $B[p:r]$ already contains the correct sorted output.

Work/span analysis of parallel merging

Let's first analyze the worst-case span $T_\infty(n)$ of P-MERGE-AUX on input subarrays that together contain a total of n elements. The call to FIND-SPLIT-POINT in line 8 contributes $\Theta(\lg n)$ to the span in the worst case, and the procedure performs at most a constant amount of additional serial work outside of the two recursive spawns in lines 12 and 14.

Because the two recursive spawns operate logically in parallel, only one of them contributes to the overall worst-case span. We claimed earlier that neither recursive invocation ever operates on more than $3n/4$ elements. Let's see why. Let $n_1 = r_1 - p_1 + 1$ and $n_2 = r_2 - p_2 + 1$, where $n = n_1 + n_2$, be the sizes of the two subarrays when line 6 starts executing, that is, after we have established that $n_2 \le n_1$ by swapping the roles of the two subarrays, if necessary. Since the pivot x is a median of of $A[p_1 : r_1]$, in the worst case, a recursive merge involves at most $n_1/2$ elements of $A[p_1 : r_1]$, but it might involve all n_2 of the elements of $A[p_2 : r_2]$. Thus we can bound the number of elements involved in a recursive invocation of P-MERGE-AUX by

$$\begin{aligned} n_1/2 + n_2 &= (2n_1 + 4n_2)/4 \\ &\le (3n_1 + 3n_2)/4 \quad (\text{since } n_2 \le n_1) \\ &= 3n/4 , \end{aligned}$$

proving the claim.

The worst-case span of P-MERGE-AUX can therefore be described by the following recurrence:

$$T_\infty(n) = T_\infty(3n/4) + \Theta(\lg n) . \tag{26.7}$$

Because this recurrence falls under case 2 of the master theorem with $k = 1$, its solution is $T_\infty(n) = \Theta(\lg^2 n)$.

Now let's verify that the work $T_1(n)$ of P-MERGE-AUX on n elements is linear. A lower bound of $\Omega(n)$ is straightforward, since each of the n elements is copied from array A to array B. We'll show that $T_1(n) = O(n)$ by deriving a recurrence for the worst-case work. The binary search in line 8 costs $\Theta(\lg n)$ in the worst case, which dominates the other work outside of the recursive spawns. For the recursive spawns, observe that although lines 12 and 14 might merge different numbers of elements, the two recursive spawns together merge at most $n - 1$ elements (since $x = A[q]$ is not merged). Moreover, as we saw when analyzing the span, a recursive spawn operates on at most $3n/4$ elements. We therefore obtain the recurrence

$$T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n) , \tag{26.8}$$

where α lies in the range $1/4 \le \alpha \le 3/4$. The value of α can vary from one recursive invocation to another.

We'll use the substitution method (see Section 4.3) to prove that the above recurrence (26.8) has solution $T_1(n) = O(n)$. (You could also use the Akra-Bazzi method from Section 4.7.) Assume that $T_1(n) \le c_1 n - c_2 \lg n$ for some positive constants c_1 and c_2. Using the properties of logarithms on pages 66–67—in particular, to deduce that $\lg \alpha + \lg(1-\alpha) = -\Theta(1)$—substitution yields

$$\begin{aligned}
T_1(n) &\le (c_1 \alpha n - c_2 \lg(\alpha n)) + (c_1(1-\alpha)n - c_2 \lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1(\alpha + (1-\alpha))n - c_2(\lg(\alpha n) + \lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1 n - c_2(\lg \alpha + \lg n + \lg(1-\alpha) + \lg n) + \Theta(\lg n) \\
&= c_1 n - c_2 \lg n - c_2(\lg n + \lg \alpha + \lg(1-\alpha)) + \Theta(\lg n) \\
&= c_1 n - c_2 \lg n - c_2(\lg n - \Theta(1)) + \Theta(\lg n) \\
&\le c_1 n - c_2 \lg n ,
\end{aligned}$$

if we choose c_2 large enough that the $c_2(\lg n - \Theta(1))$ term dominates the $\Theta(\lg n)$ term for sufficiently large n. Furthermore, we can choose c_1 large enough to satisfy the implied $\Theta(1)$ base cases of the recurrence, completing the induction. The lower and upper bounds of $\Omega(n)$ and $O(n)$ give $T_1(n) = \Theta(n)$, asymptotically the same work as for serial merging.

The execution of the pseudocode in the P-MERGE procedure itself does not add asymptotically to the work and span of P-MERGE-AUX. The **parallel for** loop in lines 3–4 has $\Theta(\lg n)$ span due to the loop control, and each iteration runs in constant time. Thus the $\Theta(\lg^2 n)$ span of P-MERGE-AUX dominates, yielding $\Theta(\lg^2 n)$ span overall for P-MERGE. The **parallel for** loop contains $\Theta(n)$ work, matching the asymptotic work of P-MERGE-AUX and yielding $\Theta(n)$ work overall for P-MERGE.

Analysis of parallel merge sort

The "heavy lifting" is done. Now that we have determined the work and span of P-MERGE, we can analyze P-MERGE-SORT. Let $T_1(n)$ and $T_\infty(n)$ be the work and span, respectively, of P-MERGE-SORT on an array of n elements. The call to P-MERGE in line 10 of P-MERGE-SORT dominates the costs of lines 1–3, for both work and span. Thus we obtain the recurrence

$$T_1(n) = 2T_1(n/2) + \Theta(n)$$

for the work of P-MERGE-SORT, and we obtain the recurrence

$$T_\infty(n) = T_\infty(n/2) + \Theta(\lg^2 n)$$

for its span. The work recurrence has solution $T_1(n) = \Theta(n \lg n)$ by case 2 of the master theorem with $k = 0$. The span recurrence has solution $T_\infty(n) = \Theta(\lg^3 n)$, also by case 2 of the master theorem, but with $k = 2$.

Parallel merging gives P-MERGE-SORT a parallelism advantage over P-NAIVE-MERGE-SORT. The parallelism of P-NAIVE-MERGE-SORT, which calls the serial MERGE procedure, is only $\Theta(\lg n)$. For P-MERGE-SORT, the parallelism is

$$\begin{aligned} T_1(n)/T_\infty(n) &= \Theta(n \lg n)/\Theta(\lg^3 n) \\ &= \Theta(n/\lg^2 n) , \end{aligned}$$

which is much better, both in theory and in practice. A good implementation in practice would sacrifice some parallelism by coarsening the base case in order to reduce the constants hidden by the asymptotic notation. For example, you could switch to an efficient serial sort, perhaps quicksort, when the number of elements to be sorted is sufficiently small.

Exercises

26.3-1
Explain how to coarsen the base case of P-MERGE.

26.3-2
Instead of finding a median element in the larger subarray, as P-MERGE does, suppose that the merge procedure finds a median of all the elements in the two sorted subarrays using the result of Exercise 9.3-10. Give pseudocode for an efficient parallel merging procedure that uses this median-finding procedure. Analyze your algorithm.

26.3-3
Give an efficient parallel algorithm for partitioning an array around a pivot, as is done by the PARTITION procedure on page 184. You need not partition the array in place. Make your algorithm as parallel as possible. Analyze your algorithm. (*Hint:* You might need an auxiliary array and might need to make more than one pass over the input elements.)

26.3-4
Give a parallel version of FFT on page 890. Make your implementation as parallel as possible. Analyze your algorithm.

★ ***26.3-5***
Show how to parallelize SELECT from Section 9.3. Make your implementation as parallel as possible. Analyze your algorithm.

Problems

26-1 Implementing parallel loops using recursive spawning

Consider the parallel procedure SUM-ARRAYS for performing pairwise addition on n-element arrays $A[1:n]$ and $B[1:n]$, storing the sums in $C[1:n]$.

```
SUM-ARRAYS(A, B, C, n)
parallel for i = 1 to n
    C[i] = A[i] + B[i]
```

a. Rewrite the parallel loop in SUM-ARRAYS using recursive spawning in the manner of P-MAT-VEC-RECURSIVE. Analyze the parallelism.

Consider another implementation of the parallel loop in SUM-ARRAYS given by the procedure SUM-ARRAYS′, where the value *grain-size* must be specified.

```
SUM-ARRAYS′(A, B, C, n)
grain-size = ?                    // to be determined
r = ⌈n/grain-size⌉
for k = 0 to r − 1
    spawn ADD-SUBARRAY(A, B, C, k · grain-size + 1,
                       min {(k + 1) · grain-size, n})
sync

ADD-SUBARRAY(A, B, C, i, j)
for k = i to j
    C[k] = A[k] + B[k]
```

b. Suppose that you set $grain\text{-}size = 1$. What is the resulting parallelism?

c. Give a formula for the span of SUM-ARRAYS′ in terms of n and *grain-size*. Derive the best value for *grain-size* to maximize parallelism.

26-2 Avoiding a temporary matrix in recursive matrix multiplication

The P-MATRIX-MULTIPLY-RECURSIVE procedure on page 772 must allocate a temporary matrix D of size $n \times n$, which can adversely affect the constants hidden by the Θ-notation. The procedure has high parallelism, however: $\Theta(n^3/\log^2 n)$.

For example, ignoring the constants in the Θ-notation, the parallelism for multiplying 1000×1000 matrices comes to approximately $1000^3/10^2 = 10^7$, since $\lg 1000 \approx 10$. Most parallel computers have far fewer than 10 million processors.

a. Parallelize MATRIX-MULTIPLY-RECURSIVE without using temporary matrices so that it retains its $\Theta(n^3)$ work. (*Hint:* Spawn the recursive calls, but insert a **sync** in a judicious location to avoid races.)

b. Give and solve recurrences for the work and span of your implementation.

c. Analyze the parallelism of your implementation. Ignoring the constants in the Θ-notation, estimate the parallelism on 1000×1000 matrices. Compare with the parallelism of P-MATRIX-MULTIPLY-RECURSIVE, and discuss whether the trade-off would be worthwhile.

26-3 Parallel matrix algorithms

Before attempting this problem, it may be helpful to read Chapter 28.

a. Parallelize the LU-DECOMPOSITION procedure on page 827 by giving pseudocode for a parallel version of this algorithm. Make your implementation as parallel as possible, and analyze its work, span, and parallelism.

b. Do the same for LUP-DECOMPOSITION on page 830.

c. Do the same for LUP-SOLVE on page 824.

d. Using equation (28.14) on page 835, write pseudocode for a parallel algorithm to invert a symmetric positive-definite matrix. Make your implementation as parallel as possible, and analyze its work, span, and parallelism.

26-4 Parallel reductions and scan (prefix) computations

A ***$\otimes$-reduction*** of an array $x[1:n]$, where $\otimes$ is an associative operator, is the value $y = x[1] \otimes x[2] \otimes \cdots \otimes x[n]$. The REDUCE procedure computes the $\otimes$-reduction of a subarray $x[i:j]$ serially.

```
REDUCE(x, i, j)
y = x[i]
for k = i + 1 to j
    y = y ⊗ x[k]
return y
```

a. Design and analyze a parallel algorithm P-REDUCE that uses recursive spawning to perform the same function with $\Theta(n)$ work and $\Theta(\lg n)$ span.

A related problem is that of computing a ***⊗-scan***, sometimes called a ***⊗-prefix computation***, on an array $x[1:n]$, where $\otimes$ is once again an associative operator. The $\otimes$-scan, implemented by the serial procedure SCAN, produces the array $y[1:n]$ given by

$$\begin{aligned}
y[1] &= x[1] , \\
y[2] &= x[1] \otimes x[2] , \\
y[3] &= x[1] \otimes x[2] \otimes x[3] , \\
&\vdots \\
y[n] &= x[1] \otimes x[2] \otimes x[3] \otimes \cdots \otimes x[n] ,
\end{aligned}$$

that is, all prefixes of the array x "summed" using the $\otimes$ operator.

```
SCAN(x, n)
let y[1:n] be a new array
y[1] = x[1]
for i = 2 to n
    y[i] = y[i - 1] ⊗ x[i]
return y
```

Parallelizing SCAN is not straightforward. For example, simply changing the **for** loop to a **parallel for** loop would create races, since each iteration of the loop body depends on the previous iteration. The procedures P-SCAN-1 and P-SCAN-1-AUX perform the $\otimes$-scan in parallel, albeit inefficiently.

```
P-SCAN-1(x, n)
let y[1:n] be a new array
P-SCAN-1-AUX(x, y, 1, n)
return y

P-SCAN-1-AUX(x, y, i, j)
parallel for l = i to j
    y[l] = P-REDUCE(x, 1, l)
```

b. Analyze the work, span, and parallelism of P-SCAN-1.

The procedures P-SCAN-2 and P-SCAN-2-AUX use recursive spawning to perform a more efficient $\otimes$-scan.

```
P-SCAN-2(x, n)
let y[1 : n] be a new array
P-SCAN-2-AUX(x, y, 1, n)
return y

P-SCAN-2-AUX(x, y, i, j)
if i == j
    y[i] = x[i]
else k = ⌊(i + j)/2⌋
    spawn P-SCAN-2-AUX(x, y, i, k)
    P-SCAN-2-AUX(x, y, k + 1, j)
    sync
    parallel for l = k + 1 to j
        y[l] = y[k] ⊗ y[l]
```

c. Argue that P-SCAN-2 is correct, and analyze its work, span, and parallelism.

To improve on both P-SCAN-1 and P-SCAN-2, perform the $\otimes$-scan in two distinct passes over the data. The first pass gathers the terms for various contiguous subarrays of x into a temporary array t, and the second pass uses the terms in t to compute the final result y. The pseudocode in the procedures P-SCAN-3, P-SCAN-UP, and P-SCAN-DOWN on the facing page implements this strategy, but certain expressions have been omitted.

d. Fill in the three missing expressions in line 8 of P-SCAN-UP and lines 5 and 6 of P-SCAN-DOWN. Argue that with the expressions you supplied, P-SCAN-3 is correct. (*Hint:* Prove that the value v passed to P-SCAN-DOWN(v, x, t, y, i, j) satisfies $v = x[1] \otimes x[2] \otimes \cdots \otimes x[i-1]$.)

e. Analyze the work, span, and parallelism of P-SCAN-3.

f. Describe how to rewrite P-SCAN-3 so that it doesn't require the use of the temporary array t.

★ ***g.*** Give an algorithm P-SCAN-4(x, n) for a scan that operates in place. It should place its output in x and require only constant auxiliary storage.

h. Describe an efficient parallel algorithm that uses a $+$-scan to determine whether a string of parentheses is well formed. For example, the string (( ) ( )) ( )

is well formed, but the string (( ))) (( ) is not. (*Hint:* Interpret (as a 1 and) as a -1, and then perform a $+$-scan.)

```
P-SCAN-3(x, n)
let y[1 : n] and t[1 : n] be new arrays
y[1] = x[1]
if n > 1
    P-SCAN-UP(x, t, 2, n)
    P-SCAN-DOWN(x[1], x, t, y, 2, n)
return y

P-SCAN-UP(x, t, i, j)
if i == j
    return x[i]
else
    k = ⌊(i + j)/2⌋
    t[k] = spawn P-SCAN-UP(x, t, i, k)
    right = P-SCAN-UP(x, t, k + 1, j)
    sync
    return _____          // fill in the blank

P-SCAN-DOWN(v, x, t, y, i, j)
if i == j
    y[i] = v ⊗ x[i]
else
    k = ⌊(i + j)/2⌋
    spawn P-SCAN-DOWN(_____, x, t, y, i, k)     // fill in the blank
    P-SCAN-DOWN(_____, x, t, y, k + 1, j)       // fill in the blank
    sync
```

26-5 Parallelizing a simple stencil calculation

Computational science is replete with algorithms that require the entries of an array to be filled in with values that depend on the values of certain already computed neighboring entries, along with other information that does not change over the course of the computation. The pattern of neighboring entries does not change during the computation and is called a ***stencil***. For example, Section 14.4 presents a stencil algorithm to compute a longest common subsequence, where the value in entry $c[i, j]$ depends only on the values in $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$,

as well as the elements x_i and y_j within the two sequences given as inputs. The input sequences are fixed, but the algorithm fills in the two-dimensional array c so that it computes entry $c[i, j]$ after computing all three entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$.

This problem examines how to use recursive spawning to parallelize a simple stencil calculation on an $n \times n$ array A in which the value placed into entry $A[i, j]$ depends only on values in $A[i', j']$, where $i' \le i$ and $j' \le j$ (and of course, $i' \ne i$ or $j' \ne j$). In other words, the value in an entry depends only on values in entries that are above it and/or to its left, along with static information outside of the array. Furthermore, we assume throughout this problem that once the entries upon which $A[i, j]$ depends have been filled in, the entry $A[i, j]$ can be computed in $\Theta(1)$ time (as in the LCS-LENGTH procedure of Section 14.4).

Partition the $n \times n$ array A into four $n/2 \times n/2$ subarrays as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} . \tag{26.9}$$

You can immediately fill in subarray A_{11} recursively, since it does not depend on the entries in the other three subarrays. Once the computation of A_{11} finishes, you can fill in A_{12} and A_{21} recursively in parallel, because although they both depend on A_{11}, they do not depend on each other. Finally, you can fill in A_{22} recursively.

a. Give parallel pseudocode that performs this simple stencil calculation using a divide-and-conquer algorithm SIMPLE-STENCIL based on the decomposition (26.9) and the discussion above. (Don't worry about the details of the base case, which depends on the specific stencil.) Give and solve recurrences for the work and span of this algorithm in terms of n. What is the parallelism?

b. Modify your solution to part (a) to divide an $n \times n$ array into nine $n/3 \times n/3$ subarrays, again recursing with as much parallelism as possible. Analyze this algorithm. How much more or less parallelism does this algorithm have compared with the algorithm from part (a)?

c. Generalize your solutions to parts (a) and (b) as follows. Choose an integer $b \ge 2$. Divide an $n \times n$ array into b^2 subarrays, each of size $n/b \times n/b$, recursing with as much parallelism as possible. In terms of n and b, what are the work, span, and parallelism of your algorithm? Argue that, using this approach, the parallelism must be $o(n)$ for any choice of $b \ge 2$. (*Hint:* For this argument, show that the exponent of n in the parallelism is strictly less than 1 for any choice of $b \ge 2$.)

d. Give pseudocode for a parallel algorithm for this simple stencil calculation that achieves $\Theta(n/\lg n)$ parallelism. Argue using notions of work and span that

the problem has $\Theta(n)$ inherent parallelism. Unfortunately, simple fork-join parallelism does not let you achieve this maximal parallelism.

26-6 Randomized parallel algorithms

Like serial algorithms, parallel algorithms can employ random-number generators. This problem explores how to adapt the measures of work, span, and parallelism to handle the expected behavior of randomized task-parallel algorithms. It also asks you to design and analyze a parallel algorithm for randomized quicksort.

a. Explain how to modify the work law (26.2), span law (26.3), and greedy scheduler bound (26.4) to work with expectations when T_P, T_1, and T_∞ are all random variables.

b. Consider a randomized parallel algorithm for which 1% of the time, $T_1 = 10^4$ and $T_{10,000} = 1$, but for the remaining 99% of the time, $T_1 = T_{10,000} = 10^9$. Argue that the ***speedup*** of a randomized parallel algorithm should be defined as $\mathrm{E}[T_1]/\mathrm{E}[T_P]$, rather than $\mathrm{E}[T_1/T_P]$.

c. Argue that the ***parallelism*** of a randomized task-parallel algorithm should be defined as the ratio $\mathrm{E}[T_1]/\mathrm{E}[T_\infty]$.

d. Parallelize the RANDOMIZED-QUICKSORT algorithm on page 192 by using recursive spawning to produce P-RANDOMIZED-QUICKSORT. (Do not parallelize RANDOMIZED-PARTITION.)

e. Analyze your parallel algorithm for randomized quicksort. (*Hint:* Review the analysis of RANDOMIZED-SELECT on page 230.)

f. Parallelize RANDOMIZED-SELECT on page 230. Make your implementation as parallel as possible. Analyze your algorithm. (*Hint:* Use the partitioning algorithm from Exercise 26.3-3.)

Chapter notes

Parallel computers and algorithmic models for parallel programming have been around in various forms for years. Prior editions of this book included material on sorting networks and the PRAM (Parallel Random-Access Machine) model. The data-parallel model [58, 217] is another popular algorithmic programming model, which features operations on vectors and matrices as primitives. The notion of sequential consistency is due to Lamport [275].

Graham [197] and Brent [71] showed that there exist schedulers achieving the bound of Theorem 26.1. Eager, Zahorjan, and Lazowska [129] showed that

any greedy scheduler achieves this bound and proposed the methodology of using work and span (although not by those names) to analyze parallel algorithms. Blelloch [57] developed an algorithmic programming model based on work and span (which he called "depth") for data-parallel programming. Blumofe and Leiserson [63] gave a distributed scheduling algorithm for task-parallel computations based on randomized "work-stealing" and showed that it achieves the bound $\mathrm{E}[T_P] \leq T_1/P + O(T_\infty)$. Arora, Blumofe, and Plaxton [20] and Blelloch, Gibbons, and Matias [61] also provided provably good algorithms for scheduling task-parallel computations. The recent literature contains many algorithms and strategies for scheduling parallel programs.

The parallel pseudocode and programming model were influenced by Cilk [290, 291, 383, 396]. The open-source project OpenCilk (www.opencilk.org) provides Cilk programming as an extension to the C and C++ programming languages. All of the parallel algorithms in this chapter can be coded straightforwardly in Cilk.

Concerns about nondeterministic parallel programs were expressed by Lee [281] and Bocchino, Adve, Adve, and Snir [64]. The algorithms literature contains many algorithmic strategies (see, for example, [60, 85, 118, 140, 160, 282, 283, 412, 461]) for detecting races and extending the fork-join model to avoid or safely embrace various kinds of nondeterminism. Blelloch, Fineman, Gibbons, and Shun [59] showed that deterministic parallel algorithms can often be as fast as, or even faster than, their nondeterministic counterparts.

Several of the parallel algorithms in this chapter appeared in unpublished lecture notes by C. E. Leiserson and H. Prokop and were originally implemented in Cilk. The parallel merge-sorting algorithm was inspired by an algorithm due to Akl [12].

27 Online Algorithms

Most problems described in this book have assumed that the entire input was available before the algorithm executes. In many situations, however, the input becomes available not in advance, but only as the algorithm executes. This idea was implicit in much of the discussion of data structures in Part III. The reason that you want to design, for example, a data structure that can handle n INSERT, DELETE, and SEARCH operations in $O(\lg n)$ time per operation is most likely because you are going to receive n such operation requests without knowing in advance what operations will be coming. This idea was also implicit in amortized analysis in Chapter 16, where we saw how to maintain a table that can grow or shrink in response to a sequence of insertion and deletion operations, yet with a constant amortized cost per operation.

An ***online algorithm*** receives its input progressively over time, rather than having the entire input available at the start, as in an ***offline algorithm***. Online algorithms pertain to many situations in which information arrives gradually. A stock trader must make decisions today, without knowing what the prices will be tomorrow, yet wants to achieve good returns. A computer system must schedule arriving jobs without knowing what work will need to be done in the future. A store must decide when to order more inventory without knowing what the future demand will be. A driver for a ride-hailing service must decide whether to pick up a fare without knowing who will request rides in the future. In each of these situations, and many more, algorithmic decisions must be made without knowledge of the future.

There are several approaches for dealing with unknown future inputs. One approach is to form a probabilistic model of future inputs and design an algorithm that assumes future inputs conform to the model. This technique is common, for example, in the field of queuing theory, and it is also related to machine learning. Of course, you might not be able to develop a workable probabilistic model, or even if you can, some inputs might not conform to it. This chapter takes a differ-

ent approach. Instead of assuming anything about the future input, we employ a conservative strategy of limiting how poor a solution any input can entail.

This chapter, therefore, adopts a worst-case approach, designing online algorithms that guarantee the quality of the solution for all possible future inputs. We'll analyze online algorithms by comparing the solution produced by the online algorithm with a solution produced by an optimal algorithm that knows the future inputs, and taking a worst-case ratio over all possible instances. We call this methodology ***competitive analysis***. We'll use a similar approach when we study approximation algorithms in Chapter 35, where we'll compare the solution returned by an algorithm that might be suboptimal with the value of the optimal solution, and determine a worst-case ratio over all possible instances.

We start with a "toy" problem: deciding between whether to take the elevator or the stairs. This problem will introduce the basic methodology of thinking about online algorithms and how to analyze them via competitive analysis. We will then look at two problems that use competitive analysis. The first is how to maintain a search list so that the access time is not too large, and the second is about strategies for deciding which cache blocks to evict from a cache or other kind of fast computer memory.

27.1 Waiting for an elevator

Our first example of an online algorithm models a problem that you likely have encountered yourself: whether you should wait for an elevator to arrive or just take the stairs. Suppose that you enter a building and wish to visit an office that is k floors up. You have two choices: walk up the stairs or take the elevator. Let's assume, for convenience, that you can climb the stairs at the rate of one floor per minute. The elevator travels much faster than you can climb the stairs: it can ascend all k floors in just one minute. Your dilemma is that you do not know how long it will take for the elevator to arrive at the ground floor and pick you up. Should you take the elevator or the stairs? How do you decide?

Let's analyze the problem. Taking the stairs takes k minutes, no matter what. Suppose you know that the elevator takes at most $B - 1$ minutes to arrive for some value of B that is considerably higher than k. (The elevator could be going up when you call for it and then stop at several floors on its way down.) To keep things simple, let's also assume that the number of minutes for the elevator to arrive is an integer. Therefore, waiting for the elevator and taking it k floors up takes anywhere from one minute (if the elevator is already at the ground floor) to $(B-1)+1 = B$ minutes (the worst case). Although you know B and k, you don't know how long the elevator will take to arrive this time. You can use competitive

analysis to inform your decision regarding whether to take the stairs or elevator. In the spirit of competitive analysis, you want to be sure that, no matter what the future brings (i.e., how long the elevator takes to arrive), you will not wait much longer than a seer who knows when the elevator will arrive.

Let us first consider what the seer would do. If the seer knows that the elevator is going to arrive in at most $k - 1$ minutes, the seer waits for the elevator, and otherwise, the seer takes the stairs. Letting m denote the number of minutes it takes for the elevator to arrive at the ground floor, we can express the time that the seer spends as the function

$$t(m) = \begin{cases} m + 1 & \text{if } m \le k - 1 \ , \\ k & \text{if } m \ge k \ . \end{cases} \tag{27.1}$$

We typically evaluate online algorithms by their ***competitive ratio***. Let $\mathcal{U}$ denote the set (universe) of all possible inputs, and consider some input $I \in \mathcal{U}$. For a minimization problem, such as the stairs-versus-elevator problem, if an online algorithm A produces a solution with value $A(I)$ on input I and the solution from an algorithm F that knows the future has value $F(I)$ on the same input, then the competitive ratio of algorithm A is

$$\max \{A(I)/F(I) : I \in \mathcal{U}\} \ .$$

If an online algorithm has a competitive ratio of c, we say that it is ***c-competitive***. The competitive ratio is always at least 1, so that we want an online algorithm with a competitive ratio as close to 1 as possible.

In the stairs-versus-elevator problem, the only input is the time for the elevator to arrive. Algorithm F knows this information, but an online algorithm has to make a decision without knowing when the elevator will arrive. Consider the algorithm "always take the stairs," which always takes exactly k minutes. Using equation (27.1), the competitive ratio is

$$\max \{k/t(m) : 0 \le m \le B - 1\} \ . \tag{27.2}$$

Enumerating the terms in equation (27.2) gives the competitive ratio as

$$\max \left\{ \frac{k}{1}, \frac{k}{2}, \frac{k}{3}, \ldots, \frac{k}{(k-1)}, \frac{k}{k}, \frac{k}{k}, \ldots, \frac{k}{k} \right\} = k \ ,$$

so that the competitive ratio is k. The maximum is achieved when the elevator arrives immediately. In this case, taking the stairs requires k minutes, but the optimal solution takes just 1 minute.

Now let's consider the opposite approach: "always take the elevator." If it takes m minutes for the elevator to arrive at the ground floor, then this algorithm will always take $m + 1$ minutes. Thus the competitive ratio becomes

$$\max \{(m + 1)/t(m) : 0 \le m \le B - 1\} \ ,$$

which we can again enumerate as

$$\max\left\{\frac{1}{1}, \frac{2}{2}, \ldots, \frac{k}{k}, \frac{k+1}{k}, \frac{k+2}{k}, \ldots, \frac{B}{k}\right\} = \frac{B}{k} .$$

Now the maximum is achieved when the elevator takes $B - 1$ minutes to arrive, compared with the optimal approach of taking the stairs, which requires k minutes.

Hence, the algorithm "always take the stairs" has competitive ratio k, and the algorithm "always take the elevator" has competitive ratio B/k. Because we prefer the algorithm with smaller competitive ratio, if $k = 10$ and $B = 300$, we prefer "always take the stairs," with competitive ratio 10, over "always take the elevator," with competitive ratio 30. Taking the stairs is not always better, or necessarily more often better. It's just that taking the stairs guards better against the worst-case future.

These two approaches of always taking the stairs and always taking the elevator are extreme solutions, however. Instead, you can "hedge your bets" and guard even better against a worst-case future. In particular, you can wait for the elevator for a while, and then if it doesn't arrive, take the stairs. How long is "a while"? Let's say that "a while" is k minutes. Then the time $h(m)$ required by this hedging strategy, as a function of the number m of minutes before the elevator arrives, is

$$h(m) = \begin{cases} m + 1 & \text{if } m \le k , \\ 2k & \text{if } m > k . \end{cases}$$

In the second case, $h(m) = 2k$ because you wait for k minutes and then climb the stairs for k minutes. The competitive ratio is now

$$\max\{h(m)/t(m) : 0 \le m \le B - 1\} .$$

Enumerating this ratio yields

$$\max\left\{\frac{1}{1}, \frac{2}{2}, \ldots, \frac{k}{k}, \frac{k+1}{k}, \frac{2k}{k}, \frac{2k}{k}, \frac{2k}{k}, \ldots, \frac{2k}{k}\right\} = 2 .$$

The competitive ratio is now *independent* of k and B.

This example illustrates a common philosophy in online algorithms: we want an algorithm that guards against any possible worst case. Initially, waiting for the elevator guards against the case when the elevator arrives quickly, but eventually switching to the stairs guards against the case when the elevator takes a long time to arrive.

Exercises

27.1-1
Suppose that when hedging your bets, you wait for p minutes, instead of for k minutes, before taking the stairs. What is the competitive ratio as a function of p and k? How should you choose p to minimize the competitive ratio?

27.1-2
Imagine that you decide to take up downhill skiing. Suppose that a pair of skis costs r dollars to rent for a day and b dollars to buy, where $b > r$. If you knew in advance how many days you would ever ski, your decision whether to rent or buy would be easy. If you'll ski for at least $\lceil b/r \rceil$ days, then you should buy skis, and otherwise you should rent. This strategy minimizes the total that you ever spend. In reality, you don't know in advance how many days you'll eventually ski. Even after you have skied several times, you still don't know how many more times you'll ever ski. Yet you don't want to waste your money. Give and analyze an algorithm that has a competitive ratio of 2, that is, an algorithm guaranteeing that, no matter how many times you ski, you never spend more than twice what you would have spent if you knew from the outset how many times you'll ski.

27.1-3
In "concentration solitaire," a game for one person, you have n pairs of matching cards. The backs of the cards are all the same, but the fronts contain pictures of animals. One pair has pictures of aardvarks, one pair has pictures of bears, one pair has pictures of camels, and so on. At the start of the game, the cards are all placed face down. In each round, you can turn two cards face up to reveal their pictures. If the pictures match, then you remove that pair from the game. If they don't match, then you turn both of them over, hiding their pictures once again. The game ends when you have removed all n pairs, and your score is how many rounds you needed to do so. Suppose that you can remember the picture on every card that you have seen. Give an algorithm to play concentration solitaire that has a competitive ratio of 2.

27.2 Maintaining a search list

The next example of an online algorithm pertains to maintaining the order of elements in a linked list, as in Section 10.2. This problem often arises in practice for hash tables when collisions are resolved by chaining (see Section 11.2), since each slot contains a linked list. Reordering the linked list of elements in each slot of the hash table can boost the performance of searches measurably.

The list-maintenance problem can be set up as follows. You are given a list L of n elements $\{x_1, x_2, \ldots, x_n\}$. We'll assume that the list is doubly linked, although the algorithms and analysis work just as well for singly linked lists. Denote the position of element x_i in the list L by $r_L(x_i)$, where $1 \le r_L(x_i) \le n$. Calling LIST-SEARCH(L, x_i) on page 260 thus takes $\Theta(r_L(x_i))$ time.

If you know in advance something about the distribution of search requests, then it makes sense to arrange the list ahead of time to put the more frequently searched elements closer to the front, which minimizes the total cost (see Exercise 27.2-1). If instead you don't know anything about the search sequence, then no matter how you arrange the list, it is possible that every search is for whatever element appears at the tail of the list. The total searching time would then be $\Theta(nm)$, where m is the number of searches.

If you notice patterns in the access sequence or you observe differences in the frequencies in which elements are accessed, then you might want to rearrange the list as you perform searches. For example, if you discover that every search is for a particular element, you could move that element to the front of the list. In general, you could rearrange the list after each call to LIST-SEARCH. But how would you do so without knowing the future? After all, no matter how you move elements around, every search could be for the last element.

But it turns out that some search sequences are "easier" than others. Rather than just evaluate performance on the worst-case sequence, let's compare a reorganization scheme with whatever an optimal offline algorithm would do if it knew the search sequence in advance. That way, if the sequence is fundamentally hard, the optimal offline algorithm will also find it hard, but if the sequence is easy, we can hope to do reasonably well.

To ease analysis, we'll drop the asymptotic notation and say that the cost is just i to search for the ith element in the list. Let's also assume that the only way to reorder the elements in the list is by swapping two adjacent elements in the list. Because the list is doubly linked, each swap incurs a cost of 1. Thus, for example, a search for the sixth element followed by moving it forward two places (entailing two swaps) incurs a total cost 8. The goal is to minimize the total cost of calls to LIST-SEARCH plus the total number of swaps performed.

The online algorithm that we'll explore is MOVE-TO-FRONT(L, x). This procedure first searches for x in the doubly linked list L, and then it moves x to the front of the list.[1] If x is located at position $r = r_L(x)$ before the call, MOVE-TO-FRONT swaps x with the element in position $r - 1$, then with the element in position $r - 2$,

[1] The path-compression heuristic in Section 19.3 resembles MOVE-TO-FRONT, although it would be more accurately expressed as "move-to-next-to-front." Unlike MOVE-TO-FRONT in a doubly linked list, path compression can relocate multiple elements to become "next-to-front."

	FORESEE					MOVE-TO-FRONT				
element searched	L	search cost	swap cost	search + swap cost	cumulative cost	L	search cost	swap cost	search + swap cost	cumulative cost
5	$\langle 1, 2, 3, 4, 5\rangle$	5	0	5	5	$\langle 1, 2, 3, 4, 5\rangle$	5	4	9	9
3	$\langle 1, 2, 3, 4, 5\rangle$	3	3	6	11	$\langle 5, 1, 2, 3, 4\rangle$	4	3	7	16
4	$\langle 4, 1, 2, 3, 5\rangle$	1	0	1	12	$\langle 3, 5, 1, 2, 4\rangle$	5	4	9	25
4	$\langle 4, 1, 2, 3, 5\rangle$	1	0	1	13	$\langle 4, 3, 5, 1, 2\rangle$	1	0	1	26

Figure 27.1 The costs incurred by the procedures FORESEE and MOVE-TO-FRONT when searching for the elements 5, 3, 4, and 4, starting with the list $L = \langle 1, 2, 3, 4, 5\rangle$. If FORESEE instead moved 3 to the front after the search for 5, the cumulative cost would not change, nor would the cumulative cost change if 4 moved to the second position after the search for 5.

and so on, until it finally swaps x with the element in position 1. Thus if the call MOVE-TO-FRONT$(L, 8)$ executes on the list $L = \langle 5, 3, 12, 4, 8, 9, 22\rangle$, the list becomes $\langle 8, 5, 3, 12, 4, 9, 22\rangle$. The call MOVE-TO-FRONT$(L, k)$ costs $2r_L(k) - 1$: it costs $r_L(k)$ to search for k, and it costs 1 for each of the $r_L(k) - 1$ swaps that move k to the front of the list.

We'll see that MOVE-TO-FRONT has a competitive ratio of 4. Let's think about what this means. MOVE-TO-FRONT performs a series of operations on a doubly linked list, accumulating cost. For comparison, suppose that there is an algorithm FORESEE that knows the future. Like MOVE-TO-FRONT, it also searches the list and moves elements around, but after each call it optimally rearranges the list for the future. (There may be more than one optimal order.) Thus FORESEE and MOVE-TO-FRONT maintain different lists of the same elements.

Consider the example shown in Figure 27.1. Starting with the list $\langle 1, 2, 3, 4, 5\rangle$, four searches occur, for the elements 5, 3, 4, and 4. The hypothetical procedure FORESEE, after searching for 3, moves 4 to the front of the list, knowing that a search for 4 is imminent. It thus incurs a swap cost of 3 upon its second call, after which no further swap costs accrue. MOVE-TO-FRONT incurs swap costs in each step, moving the found element to the front. In this example, MOVE-TO-FRONT has a higher cost in each step, but that is not necessarily always the case.

The key to proving the competitive bound is to show that at any point, the total cost of MOVE-TO-FRONT is not much higher than that of FORESEE. Surprisingly, we can determine a bound on the costs incurred by MOVE-TO-FRONT relative to FORESEE even though MOVE-TO-FRONT cannot see the future.

If we compare any particular step, MOVE-TO-FRONT and FORESEE may be operating on very different lists and do very different things. If we focus on the search for 4 above, we observe that FORESEE actually moves it to the front of the list early, paying to move the element to the front before it is accessed. To capture this con-

cept, we use the idea of an ***inversion***: a pair of elements, say a and b, in which a appears before b in one list, but b appears before a in another list. For two lists L and L', let $I(L, L')$, called the ***inversion count***, denote the number of inversions between the two lists, that is, the number of pairs of elements whose order differs in the two lists. For example, with lists $L = \langle 5, 3, 1, 4, 2 \rangle$ and $L' = \langle 3, 1, 2, 4, 5 \rangle$, then out of the $\binom{5}{2} = 10$ pairs, exactly five of them—$(1, 5), (2, 4), (2, 5), (3, 5), (4, 5)$—are inversions, since these pairs, and only these pairs, appear in different orders in the two lists. Thus the inversion count is $I(L, L') = 5$.

In order to analyze the algorithm, we define the following notation. Let L_i^M be the list maintained by MOVE-TO-FRONT immediately after the ith search, and similarly, let L_i^F be FORESEE's list immediately after the ith search. Let c_i^M and c_i^F be the costs incurred by MOVE-TO-FRONT and FORESEE on their ith calls, respectively. We don't know how many swaps FORESEE performs in its ith call, but we'll denote that number by t_i. Therefore, if the ith operation is a search for element x, then

$$c_i^M = 2r_{L_{i-1}^M}(x) - 1 , \tag{27.3}$$

$$c_i^F = r_{L_{i-1}^F}(x) + t_i . \tag{27.4}$$

In order to compare these costs more carefully, let's break down the elements into subsets, depending on their positions in the two lists before the ith search, relative to the element x being searched for in the ith search. We define three sets:

$$\begin{aligned} BB &= \{\text{elements before } x \text{ in both } L_{i-1}^M \text{ and } L_{i-1}^F\} , \\ BA &= \{\text{elements before } x \text{ in } L_{i-1}^M \text{ but after } x \text{ in } L_{i-1}^F\} , \\ AB &= \{\text{elements after } x \text{ in } L_{i-1}^M \text{ but before } x \text{ in } L_{i-1}^F\} . \end{aligned}$$

We can now relate the position of element x in L_{i-1}^F and L_{i-1}^M to the sizes of these sets:

$$r_{L_{i-1}^M}(x) = |BB| + |BA| + 1 , \tag{27.5}$$

$$r_{L_{i-1}^F}(x) = |BB| + |AB| + 1 . \tag{27.6}$$

When a swap occurs in one of the lists, it changes the relative positions of the two elements involved, which in turn changes the inversion count. Suppose that elements x and y are swapped in some list. Then the only possible difference in the inversion count between this list and *any* other list depends on whether (x, y) is an inversion. In fact, the inversion count of (x, y) with respect to any other list *must* change. If (x, y) is an inversion before the swap, it no longer is afterward, and vice versa. Therefore, if two consecutive elements x and y swap positions in a list L, then for any other list L', the value of the inversion count $I(L, L')$ either increases by 1 or decreases by 1.

As we compare MOVE-TO-FRONT and FORESEE searching and modifying their lists, we'll think about MOVE-TO-FRONT executing on its list for the ith time and then FORESEE executing on its list for the ith time. After MOVE-TO-FRONT has executed for the ith time and before FORESEE has executed for the ith time, we'll compare $I(L_{i-1}^M, L_{i-1}^F)$ (the inversion count immediately before the ith call of MOVE-TO-FRONT) with $I(L_i^M, L_{i-1}^F)$ (the inversion count after the ith call of MOVE-TO-FRONT but before the ith call of FORESEE). We'll concern ourselves later with what FORESEE does.

Let us analyze what happens to the inversion count after executing the ith call of MOVE-TO-FRONT, and suppose that it searches for element x. More precisely, we'll compute $I(L_i^M, L_{i-1}^F) - I(L_{i-1}^M, L_{i-1}^F)$, the change in the inversion count, which gives a rough idea of how much MOVE-TO-FRONT's list becomes more or less like FORESEE's list. After searching, MOVE-TO-FRONT performs a series of swaps with each of the elements on the list L_{i-1}^M that precedes x. Using the notation above, the number of such swaps is $|BB| + |BA|$. Bearing in mind that the list L_{i-1}^F has yet to be changed by the ith call of FORESEE, let's see how the inversion count changes.

Consider a swap with an element $y \in BB$. Before the swap, y precedes x in both L_{i-1}^M and L_{i-1}^F. After the swap, x precedes y in L_i^M, and L_{i-1}^F does not change. Therefore, the inversion count increases by 1 for each element in BB. Now consider a swap with an element $z \in BA$. Before the swap, z precedes x in L_{i-1}^M but x precedes z in L_{i-1}^F. After the swap, x precedes z in both lists. Therefore, the inversion count decreases by 1 for each element in BA. Thus altogether, the inversion count increases by

$$I(L_i^M, L_{i-1}^F) - I(L_{i-1}^M, L_{i-1}^F) = |BB| - |BA| \ . \tag{27.7}$$

We have laid the groundwork needed to analyze MOVE-TO-FRONT.

Theorem 27.1
Algorithm MOVE-TO-FRONT has a competitive ratio of 4.

Proof The proof uses a potential function, as described in Chapter 16 on amortized analysis. The value Φ_i of the potential function after the ith calls of MOVE-TO-FRONT and FORESEE depends on the inversion count:

$$\Phi_i = 2I(L_i^M, L_i^F) \ .$$

(Intuitively, the factor of 2 embodies the notion that each inversion represents a cost of 2 for MOVE-TO-FRONT relative to FORESEE: 1 for searching and 1 for swapping.) By equation (27.7), after the ith call of MOVE-TO-FRONT, but before the ith call of FORESEE, the potential increases by $2(|BB| - |BA|)$. Since the inversion count of the two lists is nonnegative, we have $\Phi_i \geq 0$ for all $i \geq 0$.

Assuming that MOVE-TO-FRONT and FORESEE start with the same list, the initial potential Φ_0 is 0, so that $\Phi_i \geq \Phi_0$ for all i.

Drawing from equation (16.2) on page 456, the amortized cost $\widehat{c}_i^M$ of the ith MOVE-TO-FRONT operation is

$$\widehat{c}_i^M = c_i^M + \Phi_i - \Phi_{i-1} ,$$

where c_i^M, the actual cost of the ith MOVE-TO-FRONT operation, is given by equation (27.3):

$$c_i^M = 2r_{L_{i-1}^M}(x) - 1 .$$

Now, let's consider the potential change $\Phi_i - \Phi_{i-1}$. Since both L^M and L^F change, let's consider the changes to one list at a time. Recall that when MOVE-TO-FRONT moves element x to the front, it increases the potential by exactly $2(|BB| - |BA|)$. We now consider how the optimal algorithm FORESEE changes its list L^F: it performs t_i swaps. Each swap performed by FORESEE either increases or decreases the potential by 2, and thus the increase in potential by FORESEE in the ith call can be at most $2t_i$. We therefore have

$$\begin{aligned}
\widehat{c}_i^M &= c_i^M + \Phi_i - \Phi_{i-1} \\
&\leq 2r_{L_{i-1}^M}(x) - 1 + 2(|BB| - |BA| + t_i) \\
&= 2r_{L_{i-1}^M}(x) - 1 + 2(|BB| - (r_{L_{i-1}^M}(x) - 1 - |BB|) + t_i) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{(by equation (27.5))} \\
&= 4\,|BB| + 1 + 2t_i \\
&\leq 4\,|BB| + 4\,|AB| + 4 + 4t_i \qquad \text{(increasing some terms)} \\
&= 4(|BB| + |AB| + 1 + t_i) \\
&= 4(r_{L_{i-1}^F}(x) + t_i) \qquad \text{(by equation (27.6))} \\
&= 4c_i^F \qquad \text{(by equation (27.4))} .
\end{aligned} \tag{27.8}$$

We now finish the proof as in Chapter 16 by showing that the total amortized cost provides an upper bound on the total actual cost, because the initial potential function is 0 and the potential function is always nonnegative. By equation (16.3) on page 456, for any sequence of m MOVE-TO-FRONT operations, we have

$$\begin{aligned}
\sum_{i=1}^{m} \widehat{c}_i^M &= \sum_{i=1}^{m} c_i^M + \Phi_m - \Phi_0 \\
&\geq \sum_{i=1}^{m} c_i^M \qquad \text{(because } \Phi_m \geq \Phi_0) .
\end{aligned} \tag{27.9}$$

Therefore, we have

$$\begin{aligned}\sum_{i=1}^{m} c_i^M &\le \sum_{i=1}^{m} \widehat{c}_i^M && \text{(by equation (27.9))}\\ &\le \sum_{i=1}^{m} 4c_i^F && \text{(by equation (27.8))}\\ &= 4\sum_{i=1}^{m} c_i^F \ .\end{aligned}$$

Thus the total cost of the m MOVE-TO-FRONT operations is at most 4 times the total cost of the m FORESEE operations, so MOVE-TO-FRONT is 4-competitive. ■

Isn't it amazing that we can compare MOVE-TO-FRONT with the optimal algorithm FORESEE when we have no idea of the swaps that FORESEE makes? We were able to relate the performance of MOVE-TO-FRONT to the optimal algorithm by capturing how particular properties (swaps in this case) must evolve relative to the optimal algorithm, without actually knowing the optimal algorithm.

The online algorithm MOVE-TO-FRONT has a competitive ratio of 4: on any input sequence, it incurs a cost at most 4 times that of any other algorithm. On a particular input sequence, it could cost much less than 4 times the optimal algorithm, perhaps even matching the optimal algorithm.

Exercises

27.2-1

You are given a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements, and you wish to make a static list L (no rearranging once the list is created) containing the elements of S that is good for searching. Suppose that you have a probability distribution, where $p(x_i)$ is the probability that a given search searches for element x_i. Argue that the expected cost for m searches is

$$m \sum_{i=1}^{n} p(x_i) \cdot r_L(x_i) \ .$$

Prove that this sum is minimized when the elements of L are sorted in decreasing order with respect to $p(x_i)$.

27.2-2

Professor Carnac claims that since FORESEE is an optimal algorithm that knows the future, then at each step it must incur no more cost than MOVE-TO-FRONT. Either prove that Professor Carnac is correct or provide a counterexample.

27.2-3

Another way to maintain a linked list for efficient searching is for each element to maintain a ***frequency count***: the number of times that the element has been searched for. The idea is to rearrange list elements after searches so that the list is always sorted by decreasing frequency count, from largest to smallest. Either show that this algorithm is $O(1)$-competitive, or prove that it is not.

27.2-4

The model in this section charged a cost of 1 for each swap. We can consider an alternative cost model in which, after accessing x, you can move x anywhere earlier in the list, and there is no cost for doing so. The only cost is the cost of the actual accesses. Show that MOVE-TO-FRONT is 2-competitive in this cost model, assuming that the number requests is sufficiently large. (*Hint:* Use the potential function $\Phi_i = I(L_i^M, L_i^F)$.)

27.3 Online caching

In Section 15.4, we studied the caching problem, in which ***blocks*** of data from the main memory of a computer are stored in the ***cache***: a small but faster memory. In that section, we studied the offline version of the problem, in which we assumed that we knew the sequence of memory requests in advance, and we designed an algorithm to minimize the number of cache misses. In almost all computer systems, caching is, in fact, an online problem. We do not generally know the series of cache requests in advance; they are presented to the algorithm only as the requests for blocks are actually made. To gain a better understanding of this more realistic scenario, we analyze online algorithms for caching. We will first see that all deterministic online algorithms for caching have a lower bound of $\Omega(k)$ for the competitive ratio, where k is the size of the cache. We will then present an algorithm with a competitive ratio of $\Theta(n)$, where the input size is n, and one with a competitive ratio of $O(k)$, which matches the lower bound. We will end by showing how to use randomization to design an algorithm with a much better competitive ratio of $\Theta(\lg k)$. We will also discuss the assumptions that underlie randomized online algorithms, via the notion of an adversary, such as we saw in Chapter 11 and will see in Chapter 31.

You can find the terminology used to describe the caching problem in Section 15.4, which you might wish to review before proceeding.

27.3.1 Deterministic caching algorithms

In the caching problem, the input comprises a sequence of n memory requests, for data in blocks $b_1, b_2, \ldots, b_n$, in that order. The blocks requested are not necessarily distinct: each block may appear multiple times within the request sequence. After block b_i is requested, it resides in a cache that can hold up to k blocks, where k is a fixed cache size. We assume that $n > k$, since otherwise we are assured that the cache can hold all the requested blocks at once. When a block b_i is requested, if it is already in the cache, then a ***cache hit*** occurs and the cache remains unchanged. If b_i is not in the cache, then a ***cache miss*** occurs. If the cache contains fewer than k blocks upon a cache miss, block b_i is placed into the cache, which now contains one block more than before. If a cache miss occurs with an already full cache, however, some block must be evicted from the cache before b_i can enter. Thus, a caching algorithm must decide which block to evict from the cache upon a cache miss when the cache is full. The goal is to minimize the number of cache misses over the entire request sequence. The caching algorithms considered in this chapter differ only in which block they decide to evict upon a cache miss. We do not consider abilities such as prefetching, in which a block is brought into the cache before an upcoming request in order to avert a future cache miss.

There are many online caching policies to determine which block to evict, including the following:

- First-in, first-out (FIFO): evict the block that has been in the cache the longest time.
- Last-in, first-out (LIFO): evict the block that has been in the cache the shortest time.
- Least Recently Used (LRU): evict the block whose last use is furthest in the past.
- Least Frequently Used (LFU): evict the block that has been accessed the fewest times, breaking ties by choosing the block that has been in the cache the longest.

To analyze these algorithms, we assume that the cache starts out empty, so that no evictions occur during the first k requests. We wish to compare the performance of an online algorithm to an optimal offline algorithm that knows the future requests. As we will soon see, all these deterministic online algorithms have a lower bound of $\Omega(k)$ for their competitive ratio. Some deterministic algorithms also have a competitive ratio with an $O(k)$ upper bound, but some other deterministic algorithms are considerably worse, having a competitive ratio of $\Theta(n/k)$.

We now proceed to analyze the LIFO and LRU policies. In addition to assuming that $n > k$, we will assume that at least k distinct blocks are requested. Otherwise, the cache never fills up and no blocks are evicted, so that all algorithms exhibit the same behavior. We begin by showing that LIFO has a large competitive ratio.

Theorem 27.2
LIFO has a competitive ratio of $\Theta(n/k)$ for the online caching problem with n requests and a cache of size k.

Proof We first show a lower bound of $\Omega(n/k)$. Suppose that the input consists of $k+1$ blocks, numbered $1, 2, \ldots, k+1$, and the request sequence is

$$1,\ 2,\ 3,\ 4,\ \ldots,\ k,\ k+1, k, k+1, k, k+1,\ \ldots,$$

where after the initial $1, 2, \ldots, k, k+1$, the remainder of the sequence alternates between k and $k+1$, with a total of n requests. The sequence ends on block k if n and k are either both even or both odd, and otherwise, the sequence ends on block $k+1$. That is, $b_i = i$ for $i = 1, 2, \ldots k-1$, $b_i = k+1$ for $i = k+1, k+3, \ldots$ and $b_i = k$ for $i = k, k+2, \ldots$. How many blocks does LIFO evict? After the first k requests (which are considered to be cache misses), the cache is filled with blocks $1, 2, \ldots, k$. The $(k+1)$st request, which is for block $k+1$, causes block k to be evicted. The $(k+2)$nd request, which is for block k, forces block $k+1$ to be evicted, since that block was just placed into the cache. This behavior continues, alternately evicting blocks k and $k+1$ for the remaining requests. LIFO, therefore, suffers a cache miss on every one of the n requests.

The optimal offline algorithm knows the entire sequence of requests in advance. Upon the first request of block $k+1$, it just evicts any block except block k, and then it never evicts another block. Thus, the optimal offline algorithm evicts only once. Since the first k requests are considered cache misses, the total number of cache misses is $k+1$. The competitive ratio, therefore, is $n/(k+1)$, or $\Omega(n/k)$.

For the upper bound, observe that on any input of size n, any caching algorithm incurs at most n cache misses. Because the input contains at least k distinct blocks, any caching algorithm, including the optimal offline algorithm, must incur at least k cache misses. Therefore, LIFO has a competitive ratio of $O(n/k)$. ■

We call such a competitive ratio ***unbounded***, because it grows with the input size. Exercise 27.3-2 asks you to show that LFU also has an unbounded competitive ratio.

FIFO and LRU have a much better competitive ratio of $\Theta(k)$. There is a big difference between competitive ratios of $\Theta(n/k)$ and $\Theta(k)$. The cache size k is independent of the input sequence and does not grow as more requests arrive over time. A competitive ratio that depends on n, on the other hand, does grow with the size of the input sequence and thus can get quite large. It is preferable to use an algorithm with a competitive ratio that does not grow with the input sequence's size, when possible.

We now show that LRU has a competitive ratio of $\Theta(k)$, first showing the upper bound.

Theorem 27.3
LRU has a competitive ratio of $O(k)$ for the online caching problem with n requests and a cache of size k.

Proof To analyze LRU, we will divide the sequence of requests into ***epochs***. Epoch 1 begins with the first request. Epoch i, for $i > 1$, begins upon encountering the $(k+1)$st distinct request since the beginning of epoch $i-1$. Consider the following example of requests with $k = 3$:

$$1,\ 2,\ 1,\ 5,\ 4,\ 4,\ 1,\ 2,\ 4,\ 2,\ 3,\ 4,\ 5,\ 2,\ 2,\ 1,\ 2,\ 2\,. \tag{27.10}$$

The first $k = 3$ distinct requests are for blocks 1, 2 and 5, so epoch 2 begins with the first request for block 4. In epoch 2, the first 3 distinct requests are for blocks 4, 1, and 2. Requests for these blocks recur until the request for block 3, and with this request epoch 3 begins. Thus, this example has four epochs:

$$1{,}2{,}1{,}5 \qquad 4,\ 4,\ 1,\ 2,\ 4,\ 2 \qquad 3{,}4{,}5 \qquad 2,\ 2,\ 1,\ 2,\ 2\,. \tag{27.11}$$

Now we consider the behavior of LRU. In each epoch, the first time a request for a particular block appears, it may cause a cache miss, but subsequent requests for that block within the epoch cannot cause a cache miss, since the block is now one of the k most recently used. For example, in epoch 2, the first request for block 4 causes a cache miss, but the subsequent requests for block 4 do not. (Exercise 27.3-1 asks you to show the contents of the cache after each request.) In epoch 3, requests for blocks 3 and 5 cause cache misses, but the request for block 4 does not, because it was recently accessed in epoch 2. Since only the first request for a block within an epoch can cause a cache miss and the cache holds k blocks, each epoch incurs at most k cache misses.

Now consider the behavior of the optimal algorithm. The first request in each epoch must cause a cache miss, even for an optimal algorithm. The miss occurs because, by the definition of an epoch, there *must* have been k other blocks accessed since the last access to this block.

Since, for each epoch, the optimal algorithm incurs at least one miss and LRU incurs at most k, the competitive ratio is at most $k/1 = O(k)$. ■

Exercise 27.3-3 asks you to show that FIFO also has a competitive ratio of $O(k)$.

We could show lower bounds of $\Omega(k)$ on LRU and FIFO, but in fact, we can make a much stronger statement: *any* deterministic online caching algorithm must have a competitive ratio of $\Omega(k)$. The proof relies on an adversary who knows the online algorithm being used and can tailor the future requests to cause the online algorithm to incur more cache misses than the optimal offline algorithm.

Consider a scenario in which the cache has size k and the set of possible blocks to request is $\{1, 2, \ldots, k+1\}$. The first k requests are for blocks $1, 2, \ldots, k$, so

that both the adversary and the deterministic online algorithm place these blocks into the cache. The next request is for block $k + 1$. In order to make room in the cache for block $k + 1$, the online algorithm evicts some block b_1 from the cache. The adversary, knowing that the online algorithm has just evicted block b_1, makes the next request be for b_1, so that the online algorithm must evict some other block b_2 to clear room in the cache for b_1. As you might have guessed, the adversary makes the next request be for block b_2, so that the online algorithm evicts some other block b_3 to make room for b_2. The online algorithm and the adversary continue in this manner. The online algorithm incurs a cache miss on every request and therefore incurs n cache misses over the n requests.

Now let's consider an optimal offline algorithm, which knows the future. As discussed in Section 15.4, this algorithm is known as furthest-in-future, and it always evicts the block whose next request is furthest in the future. Since there are only $k + 1$ unique blocks, when furthest-in-future evicts a block, we know that it will not be accessed during at least the next k requests. Thus, after the first k cache misses, the optimal algorithm incurs a cache miss at most once every k requests. Therefore, the number of cache misses over n requests is at most $k + n/k$.

Since the deterministic online algorithm incurs n cache misses and the optimal offline algorithm incurs at most $k + n/k$ cache misses, the competitive ratio is at least

$$\frac{n}{k + n/k} = \frac{nk}{n + k^2} .$$

For $n \geq k^2$, the above expression is at least

$$\frac{nk}{n + k^2} \geq \frac{nk}{2n} = \frac{k}{2} .$$

Thus, for sufficiently long request sequences, we have shown the following:

Theorem 27.4
Any deterministic online algorithm for caching with a cache size of k has competitive ratio $\Omega(k)$. ■

Although we can analyze the common caching strategies from the point of view of competitive analysis, the results are somewhat unsatisfying. Yes, we can distinguish between algorithms with a competitive ratio of $\Theta(k)$ and those with unbounded competitive ratios. In the end, however, all of these competitive ratios are rather high. The online algorithms we have seen so far are deterministic, and it is this property that the adversary is able to exploit.

27.3.2 Randomized caching algorithms

If we don't limit ourselves to deterministic online algorithms, we can use randomization to develop an online caching algorithm with a significantly smaller competitive ratio. Before describing the algorithm, let's discuss randomization in online algorithms in general. Recall that we analyze online algorithms with respect to an adversary who knows the online algorithm and can design requests knowing the decisions made by the online algorithm. With randomization, we must ask whether the adversary also knows the random choices made by the online algorithm. An adversary who does not know the random choices is ***oblivious***, and an adversary who knows the random choices is ***nonoblivious***. Ideally, we prefer to design algorithms against a nonoblivious adversary, as this adversary is stronger than an oblivious one. Unfortunately, a nonoblivious adversary mitigates much of the power of randomness, as an adversary who knows the outcome of random choices typically can act as if the online algorithm is deterministic. The oblivious adversary, on the other hand, does not know the random choices of the online algorithm, and that is the adversary we typically use.

As a simple illustration of the difference between an oblivious and nonoblivious adversary, imagine that you are flipping a fair coin n times, and the adversary wants to know how many heads you flipped. A nonoblivious adversary knows, after each flip, whether the coin came up heads or tails, and hence knows how many heads you flipped. An oblivious adversary, on the other hand, knows only that you are flipping a fair coin n times. The oblivious adversary, therefore, can reason that the number of heads follows a binomial distribution, so that the expected number of heads is $n/2$ (by equation (C.41) on page 1199) and the variance is $n/4$ (by equation (C.44) on page 1200). But the oblivious adversary has no way of knowing exactly how many heads you actually flipped.

Let's return to caching. We'll start with a deterministic algorithm and then randomize it. The algorithm we'll use is an approximation of LRU called MARKING. Rather than "least recently used," think of MARKING as simply "recently used." MARKING maintains a 1-bit attribute *mark* for each block in the cache. Initially, all blocks in the cache are unmarked. When a block is requested, if it is already in the cache, it is marked. If the request is a cache miss, MARKING checks to see whether there are any unmarked blocks in the cache. If all blocks are marked, then they are all changed to unmarked. Now, regardless of whether all blocks in the cache were marked when the request occurred, there is at least one unmarked block in the cache, and so an arbitrary unmarked block is evicted, and the requested block is placed into the cache and marked.

How should the block to evict from among the unmarked blocks in the cache be chosen? The procedure RANDOMIZED-MARKING on the next page shows the

process when the block is chosen randomly. The procedure takes as input a block b being requested.

```
RANDOMIZED-MARKING(b)
1  if block b resides in the cache,
2      b.mark = 1
3  else
4      if all blocks b' in the cache have b'.mark = 1
5          unmark all blocks b' in the cache, setting b'.mark = 0
6      select an unmarked block u with u.mark = 0 uniformly at random
7      evict block u
8      place block b into the cache
9      b.mark = 1
```

For the purpose of analysis, we say that a new epoch begins immediately after each time line 5 executes. An epoch starts with no marked blocks in the cache. The first time a block is requested during an epoch, the number of marked blocks increases by 1, and any subsequent requests to that block do not change the number of marked blocks. Therefore, the number of marked blocks monotonically increases within an epoch. Under this view, epochs are the same as in the proof of Theorem 27.3: with a cache that holds k blocks, an epoch comprises requests for k distinct blocks (possibly fewer for the final epoch), and the next epoch begins upon a request for a block not in those k.

Because we are going to analyze a randomized algorithm, we will compute the expected competitive ratio. Recall that for an input I, we denote the solution value of an online algorithm A by $A(I)$ and the solution value of an optimal algorithm F by $F(I)$. Online algorithm A has an ***expected competitive ratio*** c if for all inputs I, we have

$$\mathrm{E}\,[A(I)] \le cF(I)\,, \tag{27.12}$$

where the expectation is taken over the random choices made by A.

Although the deterministic MARKING algorithm has a competitive ratio of $\Theta(k)$ (Theorem 27.4 provides the lower bound and see Exercise 27.3-4 for the upper bound), RANDOMIZED-MARKING has a much smaller expected competitive ratio, namely $O(\lg k)$. The key to the improved competitive ratio is that the adversary cannot always make a request for a block that is not in the cache, since an oblivious adversary does not know which blocks are in the cache.

Theorem 27.5
RANDOMIZED-MARKING has an expected competitive ratio of $O(\lg k)$ for the online caching problem with n requests and a cache of size k, against an oblivious adversary.

Before proving Theorem 27.5, we prove a basic probabilistic fact.

Lemma 27.6
Suppose that a bag contains $x + y$ balls: $x - 1$ blue balls, y white balls, and 1 red ball. You repeatedly choose a ball at random and remove it from the bag until you have chosen a total of m balls that are either blue or red, where $m \le x$. You set aside each white ball you choose. Then, one of the balls chosen is the red ball with probability m/x.

Proof Choosing a white ball does not affect how many blue or red balls are chosen in any way. Therefore, we can continue the analysis as if there were no white balls and the bag contains just $x - 1$ blue balls and 1 red ball.

Let A be the event that the red ball is not chosen, and let A_i be the event that the ith draw does not choose the red ball. By equation (C.22) on page 1190, we have

$$\begin{aligned}\Pr\{A\} &= \Pr\{A_1 \cap A_2 \cap \cdots \cap A_m\} \\ &= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ &\quad \Pr\{A_m \mid A_1 \cap A_2 \cap \cdots \cap A_{m-1}\} \ . \end{aligned} \tag{27.13}$$

The probability $\Pr\{A_1\}$ that the first ball is blue equals $(x - 1)/x$, since initially there are $x - 1$ blue balls and 1 red ball. More generally, we have

$$\Pr\{A_i \mid A_1 \cap \cdots \cap A_{i-1}\} = \frac{x - i}{x - i + 1} \ , \tag{27.14}$$

since the ith draw is from $x - i$ blue balls and 1 red ball. Equations (27.13) and (27.14) give

$$\Pr\{A\} = \left(\frac{x-1}{x}\right)\left(\frac{x-2}{x-1}\right)\left(\frac{x-3}{x-2}\right)\cdots\left(\frac{x-m+1}{x-m+2}\right)\left(\frac{x-m}{x-m+1}\right) \ . \tag{27.15}$$

The right-hand side of equation (27.15) is a telescoping product, similar to the telescoping series in equation (A.12) on page 1143. The numerator of one term equals the denominator of the next, so that everything except the first denominator and last numerator cancel, and we obtain $\Pr\{A\} = (x - m)/x$. Since we actually want to compute $\Pr\{\bar{A}\} = 1 - \Pr\{A\}$, that is, the probability that the red ball *is* chosen, we get $\Pr\{\bar{A}\} = 1 - (x - m)/x = m/x$. ■

Now we can prove Theorem 27.5.

Proof We'll analyze RANDOMIZED-MARKING one epoch at a time. Within epoch i, any request for a block b that is not the first request for block b in epoch i must result in a cache hit, since after the first request in epoch i, block b resides in the cache and is marked, so that it cannot be evicted during the epoch. Therefore, since we are counting cache misses, we'll consider only the first request for each block within each epoch, disregarding all other requests.

We can classify the requests in an epoch as either old or new. If block b resides in the cache at the start of epoch i, each request for block b during epoch i is an ***old request***. Old requests in epoch i are for blocks requested in epoch $i - 1$. If a request in epoch i is not old, it is a ***new request***, and it is for a block not requested in epoch $i - 1$. All requests in epoch 1 are new. For example, let's look again at the request sequence in example (27.11):

$$1,2,1,5 \qquad 4,\ 4,\ 1,\ 2,\ 4,\ 2 \qquad 3,4,5 \qquad 2,\ 2,\ 1,\ 2,\ 2\ .$$

Since we can disregard all requests for a block within an epoch other than the first request, to analyze the cache behavior, we can view this request sequence as just

$$1,2,5 \qquad 4,1,2 \qquad 3,4,5 \qquad 2,\ 1\ .$$

All three requests in epoch 1 are new. In epoch 2, the requests for blocks 1 and 2 are old, but the request for block 4 is new. In epoch 3, the request for block 4 is old, and the requests for blocks 3 and 5 are new. Both requests in epoch 4 are new.

Within an epoch, each new request must cause a cache miss since, by definition, the block is not already in the cache. An old request, on the other hand, may or may not cause a cache miss. The old block is in the cache at the beginning of the epoch, but other requests might cause it to be evicted. Returning to our example, in epoch 2, the request for block 4 must cause a cache miss, as this request is new. The request for block 1, which is old, may or may not cause a cache miss. If block 1 was evicted when block 4 was requested, then a cache miss occurs and block 1 must be brought back into the cache. If instead block 1 was not evicted when block 4 was requested, then the request for block 1 results in a cache hit. The request for block 2 could incur a cache miss under two scenarios. One is if block 2 was evicted when block 4 was requested. The other is if block 1 was evicted when block 4 was requested, and then block 2 was evicted when block 1 was requested. We see that, within an epoch, each ensuing old request has an increasing chance of causing a cache miss.

Because we consider only the first request for each block within an epoch, we assume that each epoch contains exactly k requests, and each request within an epoch is for a unique block. (The last epoch might contain fewer than k requests. If it does, just add dummy requests to fill it out to k requests.) In epoch i, denote the number of new requests by $r_i \geq 1$ (an epoch must contain at least one new

request), so that the number of old requests is $k - r_i$. As mentioned above, a new request always incurs a cache miss.

Let us now focus on an arbitrary epoch i to obtain a bound on the expected number of cache misses within that epoch. In particular, let's think about the jth old request within the epoch, where $1 \le j < k$. Denote by b_{ij} the block requested in the jth old request of epoch i, and denote by n_{ij} and o_{ij} the number of new and old requests, respectively, that occur within epoch i but before the jth old request. Because $j - 1$ old requests occur before the jth old request, we have $o_{ij} = j - 1$. We will show that the probability of a cache miss upon the jth old request is $n_{ij}/(k - o_{ij})$, or $n_{ij}/(k - j + 1)$.

Start by considering the first old request, for block $b_{i,1}$. What is the probability that this request causes a cache miss? It causes a cache miss precisely when one of the $n_{i,1}$ previous requests resulted in $b_{i,1}$ being evicted. We can determine the probability that $b_{i,1}$ was chosen for eviction by using Lemma 27.6: consider the k blocks in the cache to be k balls, with block $b_{i,1}$ as the red ball, the other $k - 1$ blocks as the $k - 1$ blue balls, and no white balls. Each of the $n_{i,1}$ requests chooses a block to evict with equal probability, corresponding to drawing balls $n_{i,1}$ times. Thus, we can apply Lemma 27.6 with $x = k$, $y = 0$, and $m = n_{i,1}$, deriving the probability of a cache miss upon the first old request as $n_{i,1}/k$, which equals $n_{ij}/(k - j + 1)$ since $j = 1$.

In order to determine the probability of a cache miss for subsequent old requests, we'll need an additional observation. Let's consider the second old request, which is for block $b_{i,2}$. This request causes a cache miss precisely when one of the previous requests evicts $b_{i,2}$. Let's consider two cases, based on the request for $b_{i,1}$. In the first case, suppose that the request for $b_{i,1}$ did not cause an eviction, because $b_{i,1}$ was already in the cache. Then, the only way that $b_{i,2}$ could have been evicted is by one of the $n_{i,2}$ new requests that precedes it. What is the probability that this eviction happens? There are $n_{i,2}$ chances for $b_{i,2}$ to be evicted, but we also know that there is one block in the cache, namely $b_{i,1}$, that is not evicted. Thus, we can again apply Lemma 27.6, but with $b_{i,1}$ as the white ball, $b_{i,2}$ as the red ball, the remaining blocks as the blue balls, and drawing balls $n_{i,2}$ times. Applying Lemma 27.6, with $x = k - 1$, $y = 1$, and $m = n_{i,2}$, we find that the probability of a cache miss is $n_{i,2}/(k - 1)$. In the second case, the request for $b_{i,1}$ does cause an eviction, which can happen only if one of the new requests preceding the request for $b_{i,1}$ evicts $b_{i,1}$. Then, the request for $b_{i,1}$ brings $b_{i,1}$ back into the cache and evicts some other block. In this case, we know that of the new requests, one of them did not result in $b_{i,2}$ being evicted, since $b_{i,1}$ was evicted. Therefore, $n_{i,2} - 1$ new requests could evict $b_{i,2}$, as could the request for $b_{i,1}$, so that the number of requests that could evict $b_{i,2}$ is $n_{i,2}$. Each such request evicts a block chosen from among $k - 1$ blocks, since the request that resulted in evicting $b_{i,1}$ did not also cause $b_{i,2}$ to be evicted. Therefore, we can apply Lemma 27.6, with $x = k - 1$,

$y = 1$, and $m = n_{i,2}$, and get that the probability of a miss is $n_{i,2}/(k-1)$. In both cases the probability is the same, and it equals $n_{ij}/(k-j+1)$ since $j = 2$.

More generally, o_{ij} old requests occur before the jth old request. Each of these prior old requests either caused an eviction or did not. For those that caused an eviction, it is because they were evicted by a previous request, and for those that did not cause an eviction, it is because they were not evicted by any previous request. In either case, we can decrease the number of blocks that the random process is choosing from by 1 for each old request, and thus o_{ij} requests cannot cause b_{ij} to be evicted. Therefore, we can use Lemma 27.6 to determine the probability that b_{ij} was evicted by a previous request, with $x = k - o_{ij}$, $y = o_{ij}$ and $m = n_{ij}$. Thus, we have proven our claim that the probability of a cache miss on the jth request for an old block is $n_{ij}/(k-o_{ij})$, or $n_{ij}/(k-j+1)$. Since $n_{ij} \le r_i$ (recall that r_i is the number of new requests during epoch i), we have an upper bound of $r_i/(k-j+1)$ on the probability that the jth old request incurs a cache miss.

We can now compute the expected number of misses during epoch i using indicator random variables, as introduced in Section 5.2. We define indicator random variables

$$\begin{aligned} Y_{ij} &= \mathrm{I}\{\text{the } j\text{th old request in epoch } i \text{ incurs a cache miss}\} , \\ Z_{ij} &= \mathrm{I}\{\text{the } j\text{th new request in epoch } i \text{ incurs a cache miss}\} . \end{aligned}$$

We have $Z_{ij} = 1$ for $j = 1, 2, \ldots, r_i$, since every new request results in a cache miss. Let X_i be the random variable denoting the number of cache misses during epoch i, so that

$$X_i = \sum_{j=1}^{k-r_i} Y_{ij} + \sum_{j=1}^{r_i} Z_{ij} ,$$

and so

$$\begin{aligned} \mathrm{E}[X_i] &= \mathrm{E}\left[\sum_{j=1}^{k-r_i} Y_{ij} + \sum_{j=1}^{r_i} Z_{ij}\right] \\ &= \sum_{j=1}^{k-r_i} \mathrm{E}[Y_{ij}] + \sum_{j=1}^{r_i} \mathrm{E}[Z_{ij}] \qquad \text{(by linearity of expectation)} \\ &\le \sum_{j=1}^{k-r_i} \frac{r_i}{k-j+1} + \sum_{j=1}^{r_i} 1 \qquad \text{(by Lemma 5.1 on page 130)} \\ &= r_i \left(\sum_{j=1}^{k-r_i} \frac{1}{k-j+1} + 1 \right) \end{aligned}$$

$$
\begin{aligned}
&\le r_i \left(\sum_{j=1}^{k-1} \frac{1}{k-j+1} + 1 \right) \\
&= r_i H_k \qquad \text{(by equation (A.8) on page 1142)} , \qquad (27.16)
\end{aligned}
$$

where H_k is the kth harmonic number.

To compute the expected total number of cache misses, we sum over all epochs. Let p denote the number of epochs and X be the random variable denoting the number of cache misses. Then, we have $X = \sum_{i=1}^{p} X_i$, so that

$$
\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{p} X_i \right] \\
&= \sum_{i=1}^{p} \mathrm{E}[X_i] \qquad \text{(by linearity of expectation)} \\
&\le \sum_{i=1}^{p} r_i H_k \qquad \text{(by inequality (27.16))} \\
&= H_k \sum_{i=1}^{p} r_i \ . \qquad (27.17)
\end{aligned}
$$

To complete the analysis, we need to understand the behavior of the optimal offline algorithm. It could make a completely different set of decisions from those made by RANDOMIZED-MARKING, and at any point its cache may look nothing like the cache of the randomized algorithm. Yet, we want to relate the number of cache misses of the optimal offline algorithm to the value in inequality (27.17), in order to have a competitive ratio that does not depend on $\sum_{i=1}^{p} r_i$. Focusing on individual epochs won't suffice. At the beginning of any epoch, the offline algorithm might have loaded the cache with exactly the blocks that will be requested in that epoch. Therefore, we cannot take any one epoch in isolation and claim that an offline algorithm must suffer any cache misses during that epoch.

If we consider two consecutive epochs, however, we can better analyze the optimal offline algorithm. Consider two consecutive epochs, $i-1$ and i. Each contains k requests for k different blocks. (Recall our assumption that all requests are first requests in an epoch.) Epoch i contains r_i requests for new blocks, that is, blocks that were not requested during epoch $i-1$. Therefore, the number of distinct requests during epochs $i-1$ and i is exactly $k+r_i$. No matter what the cache contents were at the beginning of epoch $i-1$, after $k+r_i$ distinct requests, there must be at least r_i cache misses. There could be more, but there is no way to have fewer. Letting m_i denote the number of cache misses of the offline algorithm during epoch i, we have just argued that

$$
m_{i-1} + m_i \ge r_i \ . \qquad (27.18)
$$

The total number of cache misses of the offline algorithm is

$$\begin{aligned}
\sum_{i=1}^{p} m_i &= \frac{1}{2}\sum_{i=1}^{p} 2m_i \\
&= \frac{1}{2}\left(m_1 + \sum_{i=2}^{p}(m_{i-1} + m_i) + m_p\right) \\
&\geq \frac{1}{2}\left(m_1 + \sum_{i=2}^{p}(m_{i-1} + m_i)\right) \\
&\geq \frac{1}{2}\left(m_1 + \sum_{i=2}^{p} r_i\right) && \text{(by inequality (27.18))} \\
&= \frac{1}{2}\sum_{i=1}^{p} r_i && \text{(because } m_1 = r_1\text{)} .
\end{aligned}$$

The justification $m_1 = r_1$ for the last equality follows because, by our assumptions, the cache starts out empty and every request incurs a cache miss in the first epoch, even for the optimal offline adversary.

To conclude the analysis, because we have an upper bound of $H_k \sum_{i=1}^{p} r_i$ on the expected number of cache misses for RANDOMIZED-MARKING and a lower bound of $\frac{1}{2}\sum_{i=1}^{p} r_i$ on the number of cache misses for the optimal offline algorithm, the expected competitive ratio is at most

$$\begin{aligned}
\frac{H_k \sum_{i=1}^{p} r_i}{\frac{1}{2}\sum_{i=1}^{p} r_i} &= 2H_k \\
&= 2\ln k + O(1) \quad \text{(by equation (A.9) on page 1142)} \\
&= O(\lg k) .
\end{aligned}$$

■

Exercises

27.3-1

For the cache sequence (27.10), show the contents of the cache after each request and count the number of cache misses. How many misses does each epoch incur?

27.3-2

Show that LFU has a competitive ratio of $\Theta(n/k)$ for the online caching problem with n requests and a cache of size k.

27.3-3

Show that FIFO has a competitive ratio of $O(k)$ for the online caching problem with n requests and a cache of size k.

27.3-4

Show that the deterministic MARKING algorithm has a competitive ratio of $O(k)$ for the online caching problem with n requests and a cache of size k.

27.3-5

Theorem 27.4 shows that any deterministic online algorithm for caching has a competitive ratio of $\Omega(k)$, where k is the cache size. One way in which an algorithm might be able to perform better is to have some ability to know what the next few requests will be. We say that an algorithm is ***l-lookahead*** if it has the ability to look ahead at the next l requests. Prove that for every constant $l \geq 0$ and every cache size $k \geq 1$, every deterministic l-lookahead algorithm has competitive ratio $\Omega(k)$.

Problems

27-1 Cow-path problem

The Appalachian Trail (AT) is a marked hiking trail in the eastern United States extending between Springer Mountain in Georgia and Mount Katahdin in Maine. The trail is about 2,190 miles long. You decide that you are going to hike the AT from Georgia to Maine and back. You plan to learn more about algorithms while on the trail, and so you bring along your copy of *Introduction to Algorithms* in your backpack.[2] You have already read through this chapter before starting out. Because the beauty of the trail distracts you, you forget about reading this book until you have reached Maine and hiked halfway back to Georgia. At that point, you decide that you have already seen the trail and want to continue reading the rest of the book, starting with Chapter 28. Unfortunately, you find that the book is no longer in your pack. You must have left it somewhere along the trail, but you don't know where. It could be anywhere between Georgia and Maine. You want to find the book, but now that you have learned something about online algorithms, you want your algorithm for finding it to have a good competitive ratio. That is, no matter where the book is, if its distance from you is x miles away, you would like to be sure that you do not walk more than cx miles to find it, for some constant c. You do not know x, though you may assume that $x \geq 1$.[3]

[2] This book is heavy. We do not recommend that you carry it on a long hike.

[3] In case you're wondering what this problem has to do with cows, some papers about it frame the problem as a cow looking for a field in which to graze.

What algorithm should you use, and what constant c can you prove bounds the total distance cx that you would have to walk? Your algorithm should work for a trail of any length, not just the 2,190-mile-long AT.

27-2 Online scheduling to minimize average completion time

Problem 15-2 discusses scheduling to minimize average completion time on one machine, without release times and preemption and with release times and preemption. Now you will develop an online algorithm for nonpreemptively scheduling a set of tasks with release times. Suppose you are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of tasks, where task a_i has ***release time*** r_i, before which it cannot start, and requires p_i units of processing time to complete once it has started. You have one computer on which to run the tasks. Tasks cannot be ***preempted***, which is to say that once started, a task must run to completion without interruption. (See Problem 15-2 on page 446 for a more detailed description of this problem.) Given a schedule, let C_i be the ***completion time*** of task a_i, that is, the time at which task a_i completes processing. Your goal is to find a schedule that minimizes the average completion time, that is, to minimize $(1/n)\sum_{i=1}^{n} C_i$.

In the online version of this problem, you learn about task i only when it arrives at its release time r_i, and at that point, you know its processing time p_i. The offline version of this problem is NP-hard (see Chapter 34), but you will develop a 2-competitive online algorithm.

a. Show that, if there are release times, scheduling by shortest processing time (when the machine becomes idle, start the already released task with the smallest processing time that has not yet run) is not d-competitive for any constant d.

In order to develop an online algorithm, consider the preemptive version of this problem, which is discussed in Problem 15-2(b). One way to schedule is to run the tasks according to the shortest remaining processing time (SRPT) order. That is, at any point, the machine is running the available task with the smallest amount of remaining processing time.

b. Explain how to run SRPT as an online algorithm.

c. Suppose that you run SRPT and obtain completion times $C_1^P, \ldots, C_n^P$. Show that

$$\sum_{i=1}^{n} C_i^P \le \sum_{i=1}^{n} C_i^* ,$$

where the C_i^* are the completion times in an optimal nonpreemptive schedule.

Consider the (offline) algorithm COMPLETION-TIME-SCHEDULE.

```
COMPLETION-TIME-SCHEDULE(S)
compute an optimal schedule for the preemptive version of the problem
renumber the tasks so that the completion times in the optimal
    preemptive schedule are ordered by their completion times
    C_1^P < C_2^P < ··· < C_n^P in SRPT order
greedily schedule the tasks nonpreemptively in the renumbered
    order a_1, ..., a_n
let C_1, ..., C_n be the completion times of renumbered tasks a_1, ..., a_n
    in this nonpreemptive schedule
return C_1, ..., C_n
```

d. Prove that $C_i^P \geq \max\left\{\sum_{j=1}^{i} p_j, \max\{r_j : j \leq i\}\right\}$ for $i = 1, \ldots, n$.

e. Prove that $C_i \leq \max\{r_j : j \leq i\} + \sum_{j=1}^{i} p_j$ for $i = 1, \ldots, n$.

f. Algorithm COMPLETION-TIME-SCHEDULE is an offline algorithm. Explain how to modify it to produce an online algorithm.

g. Combine parts (c)–(f) to show that the online version of COMPLETION-TIME-SCHEDULE is 2-competitive.

Chapter notes

Online algorithms are widely used in many domains. Some good overviews include the textbook by Borodin and El-Yaniv [68], the collection of surveys edited by Fiat and Woeginger [142], and the survey by Albers [14].

The move-to-front heuristic from Section 27.2 was analyzed by Sleator and Tarjan [416, 417] as part of their early work on amortized analysis. This rule works quite well in practice.

Competitive analysis of online caching also originated with Sleator and Tarjan [417]. The randomized marking algorithm was proposed and analyzed by Fiat et al. [141]. Young [464] surveys online caching and paging algorithms, and Buchbinder and Naor [76] survey primal-dual online algorithms.

Specific types of online algorithms are described using other names. ***Dynamic graph algorithms*** are online algorithms on graphs, where at each step a vertex or edge undergoes modification. Typically a vertex or edge is either inserted or

deleted, or some associated property, such as edge weight, changes. Some graph problems need to be solved again after each change to the graph, and a good dynamic graph algorithm will not need to solve from scratch. For example, edges are inserted and deleted, and after each change to the graph, the minimum spanning tree is recomputed. Exercise 21.2-8 asks such a question. Similar questions can be asked for other graph algorithms, such as shortest paths, connectivity, or matching. The first paper in this field is credited to Even and Shiloach [138], who study how to maintain a shortest-path tree as edges are being deleted from a graph. Since then hundreds of papers have been published. Demetrescu et al. [110] survey early developments in dynamic graph algorithms.

For massive data sets, the input data might be too large to store. ***Streaming algorithms*** model this situation by requiring the memory used by an algorithm to be significantly smaller than the input size. For example, you may have a graph with n vertices and m edges with $m \gg n$, but the memory allowed may be only $O(n)$. Or you may have n numbers, but the memory allowed may only be $O(\lg n)$ or $O(\sqrt{n})$. A streaming algorithm is measured by the number of passes made over the data in addition to the running time of the algorithm. McGregor [322] surveys streaming algorithms for graphs and Muthukrishnan [341] surveys general streaming algorithms.

28 Matrix Operations

Because operations on matrices lie at the heart of scientific computing, efficient algorithms for working with matrices have many practical applications. This chapter focuses on how to multiply matrices and solve sets of simultaneous linear equations. Appendix D reviews the basics of matrices.

Section 28.1 shows how to solve a set of linear equations using LUP decompositions. Then, Section 28.2 explores the close relationship between multiplying and inverting matrices. Finally, Section 28.3 discusses the important class of symmetric positive-definite matrices and shows how to use them to find a least-squares solution to an overdetermined set of linear equations.

One important issue that arises in practice is ***numerical stability***. Because actual computers have limits to how precisely they can represent floating-point numbers, round-off errors in numerical computations may become amplified over the course of a computation, leading to incorrect results. Such computations are called ***numerically unstable***. Although we'll briefly consider numerical stability on occasion, we won't focus on it in this chapter. We refer you to the excellent book by Higham [216] for a thorough discussion of stability issues.

28.1 Solving systems of linear equations

Numerous applications need to solve sets of simultaneous linear equations. A linear system can be cast as a matrix equation in which each matrix or vector element belongs to a field, typically the real numbers $\mathbb{R}$. This section discusses how to solve a system of linear equations using a method called LUP decomposition.

The process starts with a set of linear equations in n unknowns $x_1, x_2, \ldots, x_n$:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 , \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 , \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n .
\end{aligned} \tag{28.1}$$

A ***solution*** to the equations (28.1) is a set of values for $x_1, x_2, \ldots, x_n$ that satisfy all of the equations simultaneously. In this section, we treat only the case in which there are exactly n equations in n unknowns.

Next, rewrite equations (28.1) as the matrix-vector equation

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

or, equivalently, letting $A = (a_{ij})$, $x = (x_i)$, and $b = (b_i)$, as

$$Ax = b . \tag{28.2}$$

If A is nonsingular, it possesses an inverse A^{-1}, and

$$x = A^{-1}b \tag{28.3}$$

is the solution vector. We can prove that x is the unique solution to equation (28.2) as follows. If there are two solutions, x and x', then $Ax = Ax' = b$ and, letting I denote an identity matrix,

$$\begin{aligned}
x &= Ix \\
&= (A^{-1}A)x \\
&= A^{-1}(Ax) \\
&= A^{-1}(Ax') \\
&= (A^{-1}A)x' \\
&= Ix' \\
&= x' .
\end{aligned}$$

This section focuses on the case in which A is nonsingular or, equivalently (by Theorem D.1 on page 1220), the rank of A equals the number n of unknowns. There are other possibilities, however, which merit a brief discussion. If the number of equations is less than the number n of unknowns—or, more generally, if the rank of A is less than n—then the system is ***underdetermined***. An underdetermined system typically has infinitely many solutions, although it may have no

solutions at all if the equations are inconsistent. If the number of equations exceeds the number n of unknowns, the system is ***overdetermined***, and there may not exist any solutions. Section 28.3 addresses the important problem of finding good approximate solutions to overdetermined systems of linear equations.

Let's return to the problem of solving the system $Ax = b$ of n equations in n unknowns. One option is to compute A^{-1} and then, using equation (28.3), multiply b by A^{-1}, yielding $x = A^{-1}b$. This approach suffers in practice from numerical instability. Fortunately, another approach—LUP decomposition—is numerically stable and has the further advantage of being faster in practice.

Overview of LUP decomposition

The idea behind LUP decomposition is to find three $n \times n$ matrices L, U, and P such that

$$PA = LU, \tag{28.4}$$

where

- L is a unit lower-triangular matrix,
- U is an upper-triangular matrix, and
- P is a permutation matrix.

We call matrices L, U, and P satisfying equation (28.4) an ***LUP decomposition*** of the matrix A. We'll show that every nonsingular matrix A possesses such a decomposition.

Computing an LUP decomposition for the matrix A has the advantage that linear systems can be efficiently solved when they are triangular, as is the case for both matrices L and U. If you have an LUP decomposition for A, you can solve equation (28.2), $Ax = b$, by solving only triangular linear systems, as follows. Multiply both sides of $Ax = b$ by P, yielding the equivalent equation $PAx = Pb$. By Exercise D.1-4 on page 1219, multiplying both sides by a permutation matrix amounts to permuting the equations (28.1). By the decomposition (28.4), substituting LU for PA gives

$$LUx = Pb\,.$$

You can now solve this equation by solving two triangular linear systems. Define $y = Ux$, where x is the desired solution vector. First, solve the lower-triangular system

$$Ly = Pb \tag{28.5}$$

for the unknown vector y by a method called "forward substitution." Having solved for y, solve the upper-triangular system

$$Ux = y \tag{28.6}$$

for the unknown x by a method called "back substitution." Why does this process solve $Ax = b$? Because the permutation matrix P is invertible (see Exercise D.2-3 on page 1223), multiplying both sides of equation (28.4) by P^{-1} gives $P^{-1}PA = P^{-1}LU$, so that

$$A = P^{-1}LU. \tag{28.7}$$

Hence, the vector x that satisfies $Ux = y$ is the solution to $Ax = b$:

$$\begin{aligned} Ax &= P^{-1}LUx && \text{(by equation (28.7))} \\ &= P^{-1}Ly && \text{(by equation (28.6))} \\ &= P^{-1}Pb && \text{(by equation (28.5))} \\ &= b\,. \end{aligned}$$

The next step is to show how forward and back substitution work and then attack the problem of computing the LUP decomposition itself.

Forward and back substitution

Forward substitution can solve the lower-triangular system (28.5) in $\Theta(n^2)$ time, given L, P, and b. An array $\pi[1:n]$ provides a more compact format to represent the permutation P than an $n \times n$ matrix that is mostly 0s. For $i = 1, 2, \ldots, n$, the entry $\pi[i]$ indicates that $P_{i,\pi[i]} = 1$ and $P_{ij} = 0$ for $j \neq \pi[i]$. Thus, PA has $a_{\pi[i],j}$ in row i and column j, and Pb has $b_{\pi[i]}$ as its ith element. Since L is unit lower-triangular, the matrix equation $Ly = Pb$ is equivalent to the n equations

$$\begin{aligned} y_1 \qquad\qquad\qquad\qquad\qquad &= b_{\pi[1]}\,, \\ l_{21}y_1 + \quad y_2 \qquad\qquad\qquad &= b_{\pi[2]}\,, \\ l_{31}y_1 + l_{32}y_2 + \quad y_3 \qquad\qquad &= b_{\pi[3]}\,, \\ &\vdots \\ l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \cdots + y_n &= b_{\pi[n]}\,. \end{aligned}$$

The first equation gives $y_1 = b_{\pi[1]}$ directly. Knowing the value of y_1, you can substitute it into the second equation, yielding

$$y_2 = b_{\pi[2]} - l_{21}y_1\,.$$

Next, you can substitute both y_1 and y_2 into the third equation, obtaining

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2)\,.$$

In general, you substitute $y_1, y_2, \ldots, y_{i-1}$ "forward" into the ith equation to solve for y_i:

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j \ .$$

Once you've solved for y, you can solve for x in equation (28.6) using ***back substitution***, which is similar to forward substitution. This time, you solve the nth equation first and work backward to the first equation. Like forward substitution, this process runs in $\Theta(n^2)$ time. Since U is upper-triangular, the matrix equation $Ux = y$ is equivalent to the n equations

$$\begin{aligned}
u_{11}x_1 + u_{12}x_2 + \cdots + \quad u_{1,n-2}x_{n-2} + \quad u_{1,n-1}x_{n-1} + \quad u_{1n}x_n &= y_1 \ , \\
u_{22}x_2 + \cdots + \quad u_{2,n-2}x_{n-2} + \quad u_{2,n-1}x_{n-1} + \quad u_{2n}x_n &= y_2 \ , \\
&\vdots \\
u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n &= y_{n-2} \ , \\
u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1} \ , \\
u_{n,n}x_n &= y_n \ .
\end{aligned}$$

Thus, you can solve for $x_n, x_{n-1}, \ldots, x_1$ successively as follows:

$$\begin{aligned}
x_n &= y_n/u_{n,n} \ , \\
x_{n-1} &= (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} \ , \\
x_{n-2} &= (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2} \ , \\
&\vdots
\end{aligned}$$

or, in general,

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j \right) / u_{ii} \ .$$

Given P, L, U, and b, the procedure LUP-SOLVE on the next page solves for x by combining forward and back substitution. The permutation matrix P is represented by the array π. The procedure first solves for y using forward substitution in lines 2–3, and then it solves for x using backward substitution in lines 4–5. Since the summation within each of the **for** loops includes an implicit loop, the running time is $\Theta(n^2)$.

As an example of these methods, consider the system of linear equations defined by $Ax = b$, where

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} \text{ and } b = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix} ,$$

```
LUP-SOLVE(L, U, π, b, n)
let x and y be new vectors of length n
for i = 1 to n
    y_i = b_{π[i]} − Σ_{j=1}^{i−1} l_ij y_j
for i = n downto 1
    x_i = (y_i − Σ_{j=i+1}^{n} u_ij x_j) / u_ii
return x
```

and we want to solve for the unknown x. The LUP decomposition is

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix}, \quad \text{and} \quad P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

(You might want to verify that $PA = LU$.) Using forward substitution, solve $Ly = Pb$ for y:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix},$$

obtaining

$$y = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix}$$

by computing first y_1, then y_2, and finally y_3. Then, using back substitution, solve $Ux = y$ for x:

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix},$$

thereby obtaining the desired answer

$$x = \begin{pmatrix} -1.4 \\ 2.2 \\ 0.6 \end{pmatrix}$$

by computing first x_3, then x_2, and finally x_1.

Computing an LU decomposition

Given an LUP decomposition for a nonsingular matrix A, you can use forward and back substitution to solve the system $Ax = b$ of linear equations. Now let's see

how to efficiently compute an LUP decomposition for A. We start with the simpler case in which A is an $n \times n$ nonsingular matrix and P is absent (or, equivalently, $P = I_n$, the $n \times n$ identity matrix), so that $A = LU$. We call the two matrices L and U an ***LU decomposition*** of A.

To create an LU decomposition, we'll use a process known as ***Gaussian elimination***. Start by subtracting multiples of the first equation from the other equations in order to remove the first variable from those equations. Then subtract multiples of the second equation from the third and subsequent equations so that now the first and second variables are removed from them. Continue this process until the system that remains has an upper-triangular form—this is the matrix U. The matrix L comprises the row multipliers that cause variables to be eliminated.

To implement this strategy, let's start with a recursive formulation. The input is an $n \times n$ nonsingular matrix A. If $n = 1$, then nothing needs to be done: just choose $L = I_1$ and $U = A$. For $n > 1$, break A into four parts:

$$\begin{aligned} A &= \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right) \\ &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix}, \end{aligned} \tag{28.8}$$

where $v = (a_{21}, a_{31}, \ldots, a_{n1})$ is a column $(n-1)$-vector, $w^{\mathrm{T}} = (a_{12}, a_{13}, \ldots, a_{1n})^{\mathrm{T}}$ is a row $(n-1)$-vector, and A' is an $(n-1) \times (n-1)$ matrix. Then, using matrix algebra (verify the equations by simply multiplying through), factor A as

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}. \end{aligned} \tag{28.9}$$

The 0s in the first and second matrices of equation (28.9) are row and column $(n-1)$-vectors, respectively. The term vw^{T}/a_{11} is an $(n-1) \times (n-1)$ matrix formed by taking the outer product of v and w and dividing each element of the result by a_{11}. Thus it conforms in size to the matrix A' from which it is subtracted. The resulting $(n-1) \times (n-1)$ matrix

$$A' - vw^{\mathrm{T}}/a_{11} \tag{28.10}$$

is called the ***Schur complement*** of A with respect to a_{11}.

We claim that if A is nonsingular, then the Schur complement is nonsingular, too. Why? Suppose that the Schur complement, which is $(n-1) \times (n-1)$, is singular. Then by Theorem D.1, it has row rank strictly less than $n-1$. Because the bottom $n-1$ entries in the first column of the matrix

$$\begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}$$

are all 0, the bottom $n-1$ rows of this matrix must have row rank strictly less than $n-1$. The row rank of the entire matrix, therefore, is strictly less than n. Applying Exercise D.2-8 on page 1223 to equation (28.9), A has rank strictly less than n, and from Theorem D.1, we derive the contradiction that A is singular.

Because the Schur complement is nonsingular, it, too, has an LU decomposition, which we can find recursively. Let's say that

$$A' - vw^{\mathrm{T}}/a_{11} = L'U' ,$$

where L' is unit lower-triangular and U' is upper-triangular. The LU decomposition of A is then $A = LU$, with

$$L = \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \text{ and } U = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} ,$$

as shown by

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} \quad \text{(by equation (28.9))} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & vw^{\mathrm{T}}/a_{11} + L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} \\ &= LU. \end{aligned}$$

Because L' is unit lower-triangular, so is L, and because U' is upper-triangular, so is U.

Of course, if $a_{11} = 0$, this method doesn't work, because it divides by 0. It also doesn't work if the upper leftmost entry of the Schur complement $A' - vw^{\mathrm{T}}/a_{11}$ is 0, since the next step of the recursion will divide by it. The denominators in each step of LU decomposition are called ***pivots***, and they occupy the diagonal elements of the matrix U. The permutation matrix P included in LUP decomposition provides a way to avoid dividing by 0, as we'll see below. Using permutations to avoid division by 0 (or by small numbers, which can contribute to numerical instability), is called ***pivoting***.

An important class of matrices for which LU decomposition always works correctly is the class of symmetric positive-definite matrices. Such matrices require no pivoting to avoid dividing by 0 in the recursive strategy outlined above. We will prove this result, as well as several others, in Section 28.3.

The pseudocode in the procedure LU-DECOMPOSITION follows the recursive strategy, except that an iteration loop replaces the recursion. (This transformation is a standard optimization for a "tail-recursive" procedure—one whose last operation is a recursive call to itself. See Problem 7-5 on page 202.) The procedure initializes the matrix U with 0s below the diagonal and matrix L with 1s on its diagonal and 0s above the diagonal. Each iteration works on a square submatrix, using its upper leftmost element as the pivot to compute the v and w vectors and the Schur complement, which becomes the square submatrix worked on by the next iteration.

```
LU-DECOMPOSITION(A, n)
1  let L and U be new n × n matrices
2  initialize U with 0s below the diagonal
3  initialize L with 1s on the diagonal and 0s above the diagonal
4  for k = 1 to n
5      u_kk = a_kk
6      for i = k + 1 to n
7          l_ik = a_ik / a_kk          // a_ik holds v_i
8          u_ki = a_ki                 // a_ki holds w_i
9      for i = k + 1 to n              // compute the Schur complement ...
10         for j = k + 1 to n
11             a_ij = a_ij − l_ik u_kj // ... and store it back into A
12 return L and U
```

Each recursive step in the description above takes place in one iteration of the outer **for** loop of lines 4–11. Within this loop, line 5 determines the pivot to be $u_{kk} = a_{kk}$. The **for** loop in lines 6–8 (which does not execute when $k = n$) uses the v and w vectors to update L and U. Line 7 determines the below-diagonal elements of L, storing v_i/a_{kk} in l_{ik}, and line 8 computes the above-diagonal elements of U, storing w_i in u_{ki}. Finally, lines 9–11 compute the elements of the Schur complement and store them back into the matrix A. (There is no need to divide by a_{kk} in line 11 because that already happened when line 7 computed l_{ik}.) Because line 11 is triply nested, LU-DECOMPOSITION runs in $\Theta(n^3)$ time.

Figure 28.1 illustrates the operation of LU-DECOMPOSITION. It shows a standard optimization of the procedure that stores the significant elements of L and U in place in the matrix A. Each element a_{ij} corresponds to either l_{ij} (if $i > j$) or u_{ij} (if $i \le j$), so that the matrix A holds both L and U when the procedure terminates. To obtain the pseudocode for this optimization from the pseudocode for the LU-DECOMPOSITION procedure, just replace each reference to l or u by a. You can verify that this transformation preserves correctness.

$$\begin{array}{cccc} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{array}$$

(a)

$$\begin{array}{c|ccc} 2 & 3 & 1 & 5 \\ \hline 3 & 4 & 2 & 4 \\ 1 & 16 & 9 & 18 \\ 2 & 4 & 9 & 21 \end{array}$$

(b)

$$\begin{array}{c|ccc} 2 & 3 & 1 & 5 \\ \hline 3 & 4 & 2 & 4 \\ 1 & 4 & 1 & 2 \\ 2 & 1 & 7 & 17 \end{array}$$

(c)

$$\begin{array}{c|c|cc} 2 & 3 & 1 & 5 \\ \hline 3 & 4 & 2 & 4 \\ \hline 1 & 4 & 1 & 2 \\ 2 & 1 & 7 & 3 \end{array}$$

(d)

$$\underset{A}{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{pmatrix}} = \underset{L}{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 2 & 1 & 7 & 1 \end{pmatrix}} \underset{U}{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 0 & 4 & 2 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{pmatrix}}$$

(e)

Figure 28.1 The operation of LU-DECOMPOSITION. **(a)** The matrix A. **(b)** The result of the first iteration of the outer **for** loop of lines 4–11. The element $a_{11} = 2$ highlighted in blue is the pivot, the tan column is v/a_{11}, and the tan row is w^{T}. The elements of U computed thus far are above the horizontal line, and the elements of L are to the left of the vertical line. The Schur complement matrix $A' - vw^{\mathrm{T}}/a_{11}$ occupies the lower right. **(c)** The result of the next iteration of the outer **for** loop, on the Schur complement matrix from part (b). The element $a_{22} = 4$ highlighted in blue is the pivot, and the tan column and row are v/a_{22} and w^{T} (in the partitioning of the Schur complement), respectively. Lines divide the matrix into the elements of U computed so far (above), the elements of L computed so far (left), and the new Schur complement (lower right). **(d)** After the next iteration, the matrix A is factored. The element 3 in the new Schur complement becomes part of U when the recursion terminates.) **(e)** The factorization $A = LU$.

Computing an LUP decomposition

If the diagonal of the matrix given to LU-DECOMPOSITION contains any 0s, then the procedure will attempt to divide by 0, which would cause disaster. Even if the diagonal contains no 0s, but does have numbers with small absolute values, dividing by such numbers can cause numerical instabilities. Therefore, LUP decomposition pivots on entries with the largest absolute values that it can find.

In LUP decomposition, the input is an $n \times n$ nonsingular matrix A, with a goal of finding a permutation matrix P, a unit lower-triangular matrix L, and an upper-triangular matrix U such that $PA = LU$. Before partitioning the matrix A, as LU decomposition does, LUP decomposition moves a nonzero element, say a_{k1}, from somewhere in the first column to the $(1, 1)$ position of the matrix. For the greatest numerical stability, LUP decomposition chooses the element in the first column with the greatest absolute value as a_{k1}. (The first column cannot contain only 0s, for then A would be singular, because its determinant would be 0, by Theorems D.4 and D.5 on page 1221.) In order to preserve the set of equations, LUP decomposition exchanges row 1 with row k, which is equivalent to multiplying A by a

permutation matrix Q on the left (Exercise D.1-4 on page 1219). Thus, the analog to equation (28.8) expresses QA as

$$QA = \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix},$$

where $v = (a_{21}, a_{31}, \ldots, a_{n1})$, except that a_{11} replaces a_{k1}; $w^{\mathrm{T}} = (a_{k2}, a_{k3}, \ldots, a_{kn})^{\mathrm{T}}$; and A' is an $(n-1) \times (n-1)$ matrix. Since $a_{k1} \neq 0$, the analog to equation (28.9) guarantees no division by 0:

$$\begin{aligned} QA &= \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix}. \end{aligned}$$

Just as in LU decomposition, if A is nonsingular, then the Schur complement $A' - vw^{\mathrm{T}}/a_{k1}$ is nonsingular, too. Therefore, you can recursively find an LUP decomposition for it, with unit lower-triangular matrix L', upper-triangular matrix U', and permutation matrix P', such that

$$P'(A' - vw^{\mathrm{T}}/a_{k1}) = L'U' .$$

Define

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q ,$$

which is a permutation matrix, since it is the product of two permutation matrices (Exercise D.1-4 on page 1219). This definition of P gives

$$\begin{aligned} PA &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA \\ &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & P'(A' - vw^{\mathrm{T}}/a_{k1}) \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} \\ &= LU, \end{aligned}$$

which yields the LUP decomposition. Because L' is unit lower-triangular, so is L, and because U' is upper-triangular, so is U.

Notice that in this derivation, unlike the one for LU decomposition, both the column vector v/a_{k1} and the Schur complement $A' - vw^{\mathrm{T}}/a_{k1}$ are multiplied by the permutation matrix P'. The procedure LUP-DECOMPOSITION gives the pseudocode for LUP decomposition.

```
LUP-DECOMPOSITION(A, n)
1  let π[1 : n] be a new array
2  for i = 1 to n
3      π[i] = i                          // initialize π to the identity permutation
4  for k = 1 to n
5      p = 0
6      for i = k to n                    // find largest absolute value in column k
7          if |a_ik| > p
8              p = |a_ik|
9              k' = i                    // row number of the largest found so far
10     if p == 0
11         error "singular matrix"
12     exchange π[k] with π[k']
13     for i = 1 to n
14         exchange a_ki with a_k'i      // exchange rows k and k'
15     for i = k + 1 to n
16         a_ik = a_ik/a_kk
17         for j = k + 1 to n
18             a_ij = a_ij − a_ik a_kj   // compute L and U in place in A
```

Like LU-DECOMPOSITION, the LUP-DECOMPOSITION procedure replaces the recursion with an iteration loop. As an improvement over a direct implementation of the recursion, the procedure dynamically maintains the permutation matrix P as an array π, where $\pi[i] = j$ means that the ith row of P contains a 1 in column j. The LUP-DECOMPOSITION procedure also implements the improvement mentioned earlier, computing L and U in place in the matrix A. Thus, when the procedure terminates,

$$a_{ij} = \begin{cases} l_{ij} & \text{if } i > j \ , \\ u_{ij} & \text{if } i \le j \ . \end{cases}$$

Figure 28.2 illustrates how LUP-DECOMPOSITION factors a matrix. Lines 2–3 initialize the array π to represent the identity permutation. The outer **for** loop of lines 4–18 implements the recursion, finding an LUP decomposition of

1	2	0	2	0.6
2	3	3	4	–2
3	5	5	4	2
4	–1	–2	3.4	–1

(a)

3	5	5	4	2
2	3	3	4	–2
1	2	0	2	0.6
4	–1	–2	3.4	–1

(b)

3	5	5	4	2
2	0.6	0	1.6	–3.2
1	0.4	–2	0.4	–.2
4	–0.2	–1	4.2	–0.6

(c)

3	5	5	4	2
2	0.6	0	1.6	–3.2
1	0.4	–2	0.4	–0.2
4	–0.2	–1	4.2	–0.6

(d)

3	5	5	4	2
1	0.4	–2	0.4	–0.2
2	0.6	0	1.6	–3.2
4	–0.2	–1	4.2	–0.6

(e)

3	5	5	4	2
1	0.4	–2	0.4	–0.2
2	0.6	0	1.6	–3.2
4	–0.2	0.5	4	–0.5

(f)

3	5	5	4	2
1	0.4	–2	0.4	–0.2
2	0.6	0	1.6	–3.2
4	–0.2	0.5	4	–0.5

(g)

3	5	5	4	2
1	0.4	–2	0.4	–0.2
4	–0.2	0.5	4	–0.5
2	0.6	0	1.6	–3.2

(h)

3	5	5	4	2
1	0.4	–2	0.4	–0.2
4	–0.2	0.5	4	–0.5
2	0.6	0	0.4	–3

(i)

$$\underset{P}{\begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}} \underset{A}{\begin{pmatrix} 2 & 0 & 2 & 0.6 \\ 3 & 3 & 4 & -2 \\ 5 & 5 & 4 & 2 \\ -1 & -2 & 3.4 & -1 \end{pmatrix}} = \underset{L}{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.4 & 1 & 0 & 0 \\ -0.2 & 0.5 & 1 & 0 \\ 0.6 & 0 & 0.4 & 1 \end{pmatrix}} \underset{U}{\begin{pmatrix} 5 & 5 & 4 & 2 \\ 0 & -2 & 0.4 & -0.2 \\ 0 & 0 & 4 & -0.5 \\ 0 & 0 & 0 & -3 \end{pmatrix}}$$

(j)

Figure 28.2 The operation of LUP-DECOMPOSITION. **(a)** The input matrix A with the identity permutation of the rows in yellow on the left. The first step of the algorithm determines that the element 5 highlighted in blue in the third row is the pivot for the first column. **(b)** Rows 1 and 3 are swapped and the permutation is updated. The tan column and row represent v and w^{T}. **(c)** The vector v is replaced by $v/5$, and the lower right of the matrix is updated with the Schur complement. Lines divide the matrix into three regions: elements of U (above), elements of L (left), and elements of the Schur complement (lower right). **(d)–(f)** The second step. **(g)–(i)** The third step. No further changes occur on the fourth (final) step. **(j)** The LUP decomposition $PA = LU$.

the $(n-k+1) \times (n-k+1)$ submatrix whose upper left is in row k and column k. Each time through the outer loop, lines 5–9 determine the element $a_{k'k}$ with the largest absolute value of those in the current first column (column k) of the $(n-k+1) \times (n-k+1)$ submatrix that the procedure is currently working on. If all elements in the current first column are 0, lines 10–11 report that the matrix is singular. To pivot, line 12 exchanges $\pi[k']$ with $\pi[k]$, and lines 13–14 exchange the kth and k'th rows of A, thereby making the pivot element a_{kk}. (The entire rows are swapped because in the derivation of the method above, not only is $A' - vw^{\mathrm{T}}/a_{k1}$ multiplied by P', but so is v/a_{k1}.) Finally, the Schur complement is computed by lines 15–18 in much the same way as it is computed by lines 6–11 of LU-DECOMPOSITION, except that here the operation is written to work in place.

Because of its triply nested loop structure, LUP-DECOMPOSITION has a running time of $\Theta(n^3)$, which is the same as that of LU-DECOMPOSITION. Thus, pivoting costs at most a constant factor in time.

Exercises

28.1-1

Solve the equation

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

by using forward substitution.

28.1-2

Find an LU decomposition of the matrix

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix}.$$

28.1-3

Solve the equation

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

by using an LUP decomposition.

28.1-4

Describe the LUP decomposition of a diagonal matrix.

28.1-5
Describe the LUP decomposition of a permutation matrix, and prove that it is unique.

28.1-6
Show that for all $n \geq 1$, there exists a singular $n \times n$ matrix that has an LU decomposition.

28.1-7
In LU-DECOMPOSITION, is it necessary to perform the outermost **for** loop iteration when $k = n$? How about in LUP-DECOMPOSITION?

28.2 Inverting matrices

Although you can use equation (28.3) to solve a system of linear equations by computing a matrix inverse, in practice you are better off using more numerically stable techniques, such as LUP decomposition. Sometimes, however, you really do need to compute a matrix inverse. This section shows how to use LUP decomposition to compute a matrix inverse. It also proves that matrix multiplication and computing the inverse of a matrix are equivalently hard problems, in that (subject to technical conditions) an algorithm for one can solve the other in the same asymptotic running time. Thus, you can use Strassen's algorithm (see Section 4.2) for matrix multiplication to invert a matrix. Indeed, Strassen's original paper was motivated by the idea that a set of a linear equations could be solved more quickly than by the usual method.

Computing a matrix inverse from an LUP decomposition

Suppose that you have an LUP decomposition of a matrix A in the form of three matrices L, U, and P such that $PA = LU$. Using LUP-SOLVE, you can solve an equation of the form $Ax = b$ in $\Theta(n^2)$ time. Since the LUP decomposition depends on A but not b, you can run LUP-SOLVE on a second set of equations of the form $Ax = b'$ in $\Theta(n^2)$ additional time. In general, once you have the LUP decomposition of A, you can solve, in $\Theta(kn^2)$ time, k versions of the equation $Ax = b$ that differ only in the vector b.

Let's think of the equation

$$AX = I_n \,, \tag{28.11}$$

which defines the matrix X, the inverse of A, as a set of n distinct equations of the form $Ax = b$. To be precise, let X_i denote the ith column of X, and recall that the

unit vector e_i is the ith column of I_n. You can then solve equation (28.11) for X by using the LUP decomposition for A to solve each equation

$$AX_i = e_i$$

separately for X_i. Once you have the LUP decomposition, you can compute each of the n columns X_i in $\Theta(n^2)$ time, and so you can compute X from the LUP decomposition of A in $\Theta(n^3)$ time. Since you find the LUP decomposition of A in $\Theta(n^3)$ time, you can compute the inverse A^{-1} of a matrix A in $\Theta(n^3)$ time.

Matrix multiplication and matrix inversion

Now let's see how the theoretical speedups obtained for matrix multiplication translate to speedups for matrix inversion. In fact, we'll prove something stronger: matrix inversion is equivalent to matrix multiplication, in the following sense. If $M(n)$ denotes the time to multiply two $n \times n$ matrices, then a nonsingular $n \times n$ matrix can be inverted in $O(M(n))$ time. Moreover, if $I(n)$ denotes the time to invert a nonsingular $n \times n$ matrix, then two $n \times n$ matrices can be multiplied in $O(I(n))$ time. We prove these results as two separate theorems.

Theorem 28.1 (Multiplication is no harder than inversion)
If an $n \times n$ matrix can be inverted in $I(n)$ time, where $I(n) = \Omega(n^2)$ and $I(n)$ satisfies the regularity condition $I(3n) = O(I(n))$, then two $n \times n$ matrices can be multiplied in $O(I(n))$ time.

Proof Let A and B be $n \times n$ matrices. To compute their product $C = AB$, define the $3n \times 3n$ matrix D by

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix} .$$

The inverse of D is

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix} ,$$

and thus to compute the product AB, just take the upper right $n \times n$ submatrix of D^{-1}.

Constructing matrix D takes $\Theta(n^2)$ time, which is $O(I(n))$ from the assumption that $I(n) = \Omega(n^2)$, and inverting D takes $O(I(3n)) = O(I(n))$ time, by the regularity condition on $I(n)$. We thus have $M(n) = O(I(n))$. ■

Note that $I(n)$ satisfies the regularity condition whenever $I(n) = \Theta(n^c \lg^d n)$ for any constants $c > 0$ and $d \geq 0$.

The proof that matrix inversion is no harder than matrix multiplication relies on some properties of symmetric positive-definite matrices proved in Section 28.3.

Theorem 28.2 (Inversion is no harder than multiplication)
Suppose that two $n \times n$ real matrices can be multiplied in $M(n)$ time, where $M(n) = \Omega(n^2)$ and $M(n)$ satisfies the following two regularity conditions:

1. $M(n+k) = O(M(n))$ for any k in the range $0 \le k < n$, and
2. $M(n/2) \le cM(n)$ for some constant $c < 1/2$.

Then the inverse of any real nonsingular $n \times n$ matrix can be computed in $O(M(n))$ time.

Proof Let A be an $n \times n$ matrix with real-valued entries that is nonsingular. Assume that n is an exact power of 2 (i.e., $n = 2^l$ for some integer l); we'll see at the end of the proof what to do if n is not an exact power of 2.

For the moment, assume that the $n \times n$ matrix A is symmetric and positive-definite. Partition each of A and its inverse A^{-1} into four $n/2 \times n/2$ submatrices:

$$A = \begin{pmatrix} B & C^{\mathrm{T}} \\ C & D \end{pmatrix} \text{ and } A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix} . \tag{28.12}$$

Then, if we let

$$S = D - CB^{-1}C^{\mathrm{T}} \tag{28.13}$$

be the Schur complement of A with respect to B (we'll see more about this form of Schur complement in Section 28.3), we have

$$A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix} = \begin{pmatrix} B^{-1} + B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1} & -B^{-1}C^{\mathrm{T}}S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix} , \tag{28.14}$$

since $AA^{-1} = I_n$, as you can verify by performing the matrix multiplication. Because A is symmetric and positive-definite, Lemmas 28.4 and 28.5 in Section 28.3 imply that B and S are both symmetric and positive-definite. By Lemma 28.3 in Section 28.3, therefore, the inverses B^{-1} and S^{-1} exist, and by Exercise D.2-6 on page 1223, B^{-1} and S^{-1} are symmetric, so that $(B^{-1})^{\mathrm{T}} = B^{-1}$ and $(S^{-1})^{\mathrm{T}} = S^{-1}$. Therefore, to compute the submatrices

$$\begin{aligned} R &= B^{-1} + B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1} , \\ T &= -B^{-1}C^{\mathrm{T}}S^{-1} , \\ U &= -S^{-1}CB^{-1} , \quad \text{and} \\ V &= S^{-1} \end{aligned}$$

of A^{-1}, do the following, where all matrices mentioned are $n/2 \times n/2$:

1. Form the submatrices B, C, C^{T}, and D of A.
2. Recursively compute the inverse B^{-1} of B.
3. Compute the matrix product $W = CB^{-1}$, and then compute its transpose W^{T}, which equals $B^{-1}C^{\mathrm{T}}$ (by Exercise D.1-2 on page 1219 and $(B^{-1})^{\mathrm{T}} = B^{-1}$).
4. Compute the matrix product $X = WC^{\mathrm{T}}$, which equals $CB^{-1}C^{\mathrm{T}}$, and then compute the matrix $S = D - X = D - CB^{-1}C^{\mathrm{T}}$.
5. Recursively compute the inverse S^{-1} of S.
6. Compute the matrix product $Y = S^{-1}W$, which equals $S^{-1}CB^{-1}$, and then compute its transpose Y^{T}, which equals $B^{-1}C^{\mathrm{T}}S^{-1}$ (by Exercise D.1-2, $(B^{-1})^{\mathrm{T}} = B^{-1}$, and $(S^{-1})^{\mathrm{T}} = S^{-1}$).
7. Compute the matrix product $Z = W^{\mathrm{T}}Y$, which equals $B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1}$.
8. Set $R = B^{-1} + Z$.
9. Set $T = -Y^{\mathrm{T}}$.
10. Set $U = -Y$.
11. Set $V = S^{-1}$.

Thus, to invert an $n \times n$ symmetric positive-definite matrix, invert two $n/2 \times n/2$ matrices in steps 2 and 5; perform four multiplications of $n/2 \times n/2$ matrices in steps 3, 4, 6, and 7; plus incur an additional cost of $O(n^2)$ for extracting submatrices from A, inserting submatrices into A^{-1}, and performing a constant number of additions, subtractions, and transposes on $n/2 \times n/2$ matrices. The running time is given by the recurrence

$$\begin{aligned} I(n) &\le 2I(n/2) + 4M(n/2) + O(n^2) \\ &= 2I(n/2) + \Theta(M(n)) \\ &= O(M(n)) \,. \end{aligned} \tag{28.15}$$

The second line follows from the assumption that $M(n) = \Omega(n^2)$ and from the second regularity condition in the statement of the theorem, which implies that $4M(n/2) < 2M(n)$. Because $M(n) = \Omega(n^2)$, case 3 of the master theorem (Theorem 4.1) applies to the recurrence (28.15), giving the $O(M(n))$ result.

It remains to prove how to obtain the same asymptotic running time for matrix multiplication as for matrix inversion when A is invertible but not symmetric and positive-definite. The basic idea is that for any nonsingular matrix A, the matrix $A^{\mathrm{T}}A$ is symmetric (by Exercise D.1-2) and positive-definite (by Theorem D.6 on page 1222). The trick, then, is to reduce the problem of inverting A to the problem of inverting $A^{\mathrm{T}}A$.

The reduction is based on the observation that when A is an $n \times n$ nonsingular matrix, we have

$$A^{-1} = (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}} ,$$

since $((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})A = (A^{\mathrm{T}}A)^{-1}(A^{\mathrm{T}}A) = I_n$ and a matrix inverse is unique. Therefore, to compute A^{-1}, first multiply A^{T} by A to obtain $A^{\mathrm{T}}A$, then invert the symmetric positive-definite matrix $A^{\mathrm{T}}A$ using the above divide-and-conquer algorithm, and finally multiply the result by A^{T}. Each of these three steps takes $O(M(n))$ time, and thus any nonsingular matrix with real entries can be inverted in $O(M(n))$ time.

The above proof assumed that A is an $n \times n$ matrix, where n is an exact power of 2. If n is not an exact power of 2, then let $k < n$ be such that $n + k$ is an exact power of 2, and define the $(n + k) \times (n + k)$ matrix A' as

$$A' = \begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix} .$$

Then the inverse of A' is

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix} ,$$

Apply the method of the proof to A' to compute the inverse of A', and take the first n rows and n columns of the result as the desired answer A^{-1}. The first regularity condition on $M(n)$ ensures that enlarging the matrix in this way increases the running time by at most a constant factor. ■

The proof of Theorem 28.2 suggests how to solve the equation $Ax = b$ by using LU decomposition without pivoting, so long as A is nonsingular. Let $y = A^{\mathrm{T}}b$. Multiply both sides of the equation $Ax = b$ by A^{T}, yielding $(A^{\mathrm{T}}A)x = A^{\mathrm{T}}b = y$. This transformation doesn't affect the solution x, since A^{T} is invertible. Because $A^{\mathrm{T}}A$ is symmetric positive-definite, it can be factored by computing an LU decomposition. Then, use forward and back substitution to solve for x in the equation $(A^{\mathrm{T}}A)x = y$. Although this method is theoretically correct, in practice the procedure LUP-DECOMPOSITION works much better. LUP decomposition requires fewer arithmetic operations by a constant factor, and it has somewhat better numerical properties.

Exercises

28.2-1

Let $M(n)$ be the time to multiply two $n \times n$ matrices, and let $S(n)$ denote the time required to square an $n \times n$ matrix. Show that multiplying and squaring matrices have essentially the same difficulty: an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time squaring algorithm, and an $S(n)$-time squaring algorithm implies an $O(S(n))$-time matrix-multiplication algorithm.

28.2-2
Let $M(n)$ be the time to multiply two $n \times n$ matrices. Show that an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time LUP-decomposition algorithm. (The LUP decomposition your method produces need not be the same as the result produced by the LUP-DECOMPOSITION procedure.)

28.2-3
Let $M(n)$ be the time to multiply two $n \times n$ boolean matrices, and let $T(n)$ be the time to find the transitive closure of an $n \times n$ boolean matrix. (See Section 23.2.) Show that an $M(n)$-time boolean matrix-multiplication algorithm implies an $O(M(n) \lg n)$-time transitive-closure algorithm, and a $T(n)$-time transitive-closure algorithm implies an $O(T(n))$-time boolean matrix-multiplication algorithm.

28.2-4
Does the matrix-inversion algorithm based on Theorem 28.2 work when matrix elements are drawn from the field of integers modulo 2? Explain.

★ ***28.2-5***
Generalize the matrix-inversion algorithm of Theorem 28.2 to handle matrices of complex numbers, and prove that your generalization works correctly. (*Hint:* Instead of the transpose of A, use the ***conjugate transpose*** A^*, which you obtain from the transpose of A by replacing every entry with its complex conjugate. Instead of symmetric matrices, consider ***Hermitian*** matrices, which are matrices A such that $A = A^*$.)

28.3 Symmetric positive-definite matrices and least-squares approximation

Symmetric positive-definite matrices have many interesting and desirable properties. An $n \times n$ matrix A is ***symmetric positive-definite*** if $A = A^{\mathrm{T}}$ (A is symmetric) and $x^{\mathrm{T}}Ax > 0$ for all n-vectors $x \neq 0$ (A is positive-definite). Symmetric positive-definite matrices are nonsingular, and an LU decomposition on them will not divide by 0. This section proves these and several other important properties of symmetric positive-definite matrices. We'll also see an interesting application to curve fitting by a least-squares approximation.

The first property we prove is perhaps the most basic.

Lemma 28.3
Any positive-definite matrix is nonsingular.

Proof Suppose that a matrix A is singular. Then by Corollary D.3 on page 1221, there exists a nonzero vector x such that $Ax = 0$. Hence, $x^{\mathrm{T}}Ax = 0$, and A cannot be positive-definite. ■

The proof that an LU decomposition on a symmetric positive-definite matrix A won't divide by 0 is more involved. We begin by proving properties about certain submatrices of A. Define the kth ***leading submatrix*** of A to be the matrix A_k consisting of the intersection of the first k rows and first k columns of A.

Lemma 28.4
If A is a symmetric positive-definite matrix, then every leading submatrix of A is symmetric and positive-definite.

Proof Since A is symmetric, each leading submatrix A_k is also symmetric. We'll prove that A_k is positive-definite by contradiction. If A_k is not positive-definite, then there exists a k-vector $x_k \neq 0$ such that $x_k^{\mathrm{T}} A_k x_k \leq 0$. Let A be $n \times n$, and

$$A = \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \tag{28.16}$$

for submatrices B (which is $(n-k)\times k$) and C (which is $(n-k)\times(n-k)$). Define the n-vector $x = (\, x_k^{\mathrm{T}} \quad 0\,)^{\mathrm{T}}$, where $n-k$ 0s follow x_k. Then we have

$$\begin{aligned} x^{\mathrm{T}}Ax &= (\, x_k^{\mathrm{T}} \quad 0\,)\begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix}\begin{pmatrix} x_k \\ 0 \end{pmatrix} \\ &= (\, x_k^{\mathrm{T}} \quad 0\,)\begin{pmatrix} A_k x_k \\ B x_k \end{pmatrix} \\ &= x_k^{\mathrm{T}} A_k x_k \\ &\leq 0\,, \end{aligned}$$

which contradicts A being positive-definite. ■

We now turn to some essential properties of the Schur complement. Let A be a symmetric positive-definite matrix, and let A_k be a leading $k \times k$ submatrix of A. Partition A once again according to equation (28.16). Equation (28.10) generalizes to define the ***Schur complement*** S of A with respect to A_k as

$$S = C - BA_k^{-1}B^{\mathrm{T}}\,. \tag{28.17}$$

(By Lemma 28.4, A_k is symmetric and positive-definite, and therefore, A_k^{-1} exists by Lemma 28.3, and S is well defined.) The earlier definition (28.10) of the Schur complement is consistent with equation (28.17) by letting $k = 1$.

The next lemma shows that the Schur-complement matrices of symmetric positive-definite matrices are themselves symmetric and positive-definite. We used this

result in Theorem 28.2, and its corollary will help prove that LU decomposition works for symmetric positive-definite matrices.

Lemma 28.5 (Schur complement lemma)
If A is a symmetric positive-definite matrix and A_k is a leading $k \times k$ submatrix of A, then the Schur complement S of A with respect to A_k is symmetric and positive-definite.

Proof Because A is symmetric, so is the submatrix C. By Exercise D.2-6 on page 1223, the product $BA_k^{-1}B^{\mathrm{T}}$ is symmetric. Since C and $BA_k^{-1}B^{\mathrm{T}}$ are symmetric, then by Exercise D.1-1 on page 1219, so is S.

It remains to show that S is positive-definite. Consider the partition of A given in equation (28.16). For any nonzero vector x, we have $x^{\mathrm{T}}Ax > 0$ by the assumption that A is positive-definite. Let the subvectors y and z consist of the first k and last $n - k$ elements in x, respectively, and thus they are compatible with A_k and C, respectively. Because A_k^{-1} exists, we have

$$\begin{aligned}
x^{\mathrm{T}}Ax &= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \\
&= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k y + B^{\mathrm{T}} z \\ By + Cz \end{pmatrix} \\
&= y^{\mathrm{T}} A_k y + y^{\mathrm{T}} B^{\mathrm{T}} z + z^{\mathrm{T}} B y + z^{\mathrm{T}} C z \\
&= (y + A_k^{-1} B^{\mathrm{T}} z)^{\mathrm{T}} A_k (y + A_k^{-1} B^{\mathrm{T}} z) + z^{\mathrm{T}} (C - B A_k^{-1} B^{\mathrm{T}}) z \,, \qquad (28.18)
\end{aligned}$$

This last equation, which you can verify by multiplying through, amounts to "completing the square" of the quadratic form. (See Exercise 28.3-2.)

Since $x^{\mathrm{T}}Ax > 0$ holds for any nonzero x, pick any nonzero z and then choose $y = -A_k^{-1}B^{\mathrm{T}}z$, which causes the first term in equation (28.18) to vanish, leaving

$$z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z = z^{\mathrm{T}}Sz$$

as the value of the expression. For any $z \neq 0$, we therefore have $z^{\mathrm{T}}Sz = x^{\mathrm{T}}Ax > 0$, and thus S is positive-definite. ■

Corollary 28.6
LU decomposition of a symmetric positive-definite matrix never causes a division by 0.

Proof Let A be an $n \times n$ symmetric positive-definite matrix. In fact, we'll prove a stronger result than the statement of the corollary: every pivot is strictly positive. The first pivot is a_{11}. Let e_1 be the length-n unit vector $(1\ 0\ 0\ \cdots\ 0)^{\mathrm{T}}$, so that $a_{11} = e_1^{\mathrm{T}}Ae_1$, which is positive because e_1 is nonzero and A is positive

definite. Since the first step of LU decomposition produces the Schur complement of A with respect to $A_1 = (a_{11})$, Lemma 28.5 implies by induction that all pivots are positive. ■

Least-squares approximation

One important application of symmetric positive-definite matrices arises in fitting curves to given sets of data points. You are given a set of m data points

$$(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m) ,$$

where you know that the y_i are subject to measurement errors. You wish to determine a function $F(x)$ such that the approximation errors

$$\eta_i = F(x_i) - y_i \tag{28.19}$$

are small for $i = 1, 2, \ldots, m$. The form of the function F depends on the problem at hand. Let's assume that it has the form of a linearly weighted sum

$$F(x) = \sum_{j=1}^{n} c_j f_j(x) ,$$

where the number n of summands and the specific ***basis functions*** f_j are chosen based on knowledge of the problem at hand. A common choice is $f_j(x) = x^{j-1}$, which means that

$$F(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_n x^{n-1}$$

is a polynomial of degree $n - 1$ in x. Thus, if you are given m data points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$, you need to calculate n coefficients $c_1, c_2, \ldots, c_n$ that minimize the approximation errors $\eta_1, \eta_2, \ldots, \eta_m$.

By choosing $n = m$, you can calculate each y_i *exactly* in equation (28.19). Such a high-degree polynomial F "fits the noise" as well as the data, however, and generally gives poor results when used to predict y for previously unseen values of x. It is usually better to choose n significantly smaller than m and hope that by choosing the coefficients c_j well, you can obtain a function F that finds the significant patterns in the data points without paying undue attention to the noise. Some theoretical principles exist for choosing n, but they are beyond the scope of this text. In any case, once you choose a value of n that is less than m, you end up with an overdetermined set of equations whose solution you wish to approximate. Let's see how to do so.

Let

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix}$$

denote the matrix of values of the basis functions at the given points, that is, $a_{ij} = f_j(x_i)$. Let $c = (c_k)$ denote the desired n-vector of coefficients. Then,

$$\begin{aligned} Ac &= \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \\ &= \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix} \end{aligned}$$

is the m-vector of "predicted values" for y. Thus,

$$\eta = Ac - y$$

is the m-vector of ***approximation errors***.

To minimize approximation errors, let's minimize the norm of the error vector η, which gives a ***least-squares solution***, since

$$\|\eta\| = \left(\sum_{i=1}^{m} \eta_i^2 \right)^{1/2} .$$

Because

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right)^2 ,$$

to minimize $\|\eta\|$, differentiate $\|\eta\|^2$ with respect to each c_k and then set the result to 0:

$$\frac{d\,\|\eta\|^2}{dc_k} = \sum_{i=1}^{m} 2 \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right) a_{ik} = 0 . \tag{28.20}$$

The n equations (28.20) for $k = 1, 2, \dots, n$ are equivalent to the single matrix equation

$$(Ac - y)^{\mathrm{T}} A = 0$$

or, equivalently (using Exercise D.1-2 on page 1219), to

$$A^{\mathrm{T}}(Ac - y) = 0 ,$$

which implies

$$A^{\mathrm{T}} A c = A^{\mathrm{T}} y . \tag{28.21}$$

In statistics, equation (28.21) is called the ***normal equation***. The matrix $A^{\mathrm{T}}A$ is symmetric by Exercise D.1-2, and if A has full column rank, then by Theorem D.6 on page 1222, $A^{\mathrm{T}}A$ is positive-definite as well. Hence, $(A^{\mathrm{T}}A)^{-1}$ exists, and the solution to equation (28.21) is

$$\begin{aligned} c &= \left((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}\right) y \\ &= A^{+} y , \end{aligned} \tag{28.22}$$

where the matrix $A^{+} = ((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})$ is the ***pseudoinverse*** of the matrix A. The pseudoinverse naturally generalizes the notion of a matrix inverse to the case in which A is not square. (Compare equation (28.22) as the approximate solution to $Ac = y$ with the solution $A^{-1}b$ as the exact solution to $Ax = b$.)

As an example of producing a least-squares fit, suppose that you have five data points

$$\begin{aligned} (x_1, y_1) &= (-1, 2) , \\ (x_2, y_2) &= (1, 1) , \\ (x_3, y_3) &= (2, 1) , \\ (x_4, y_4) &= (3, 0) , \\ (x_5, y_5) &= (5, 3) , \end{aligned}$$

shown as orange dots in Figure 28.3, and you want to fit these points with a quadratic polynomial

$$F(x) = c_1 + c_2 x + c_3 x^2 .$$

Start with the matrix of basis-function values

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix} ,$$

whose pseudoinverse is

Figure 28.3 The least-squares fit of a quadratic polynomial to the set of five data points $\{(-1, 2), (1, 1), (2, 1), (3, 0), (5, 3)\}$. The orange dots are the data points, and the blue dots are their estimated values predicted by the polynomial $F(x) = 1.2 - 0.757x + 0.214x^2$, the quadratic polynomial that minimizes the sum of the squared errors, plotted in blue. Each orange line shows the error for one data point.

$$A^+ = \begin{pmatrix} 0.500 & 0.300 & 0.200 & 0.100 & -0.100 \\ -0.388 & 0.093 & 0.190 & 0.193 & -0.088 \\ 0.060 & -0.036 & -0.048 & -0.036 & 0.060 \end{pmatrix} .$$

Multiplying y by A^+ gives the coefficient vector

$$c = \begin{pmatrix} 1.200 \\ -0.757 \\ 0.214 \end{pmatrix} ,$$

which corresponds to the quadratic polynomial

$$F(x) = 1.200 - 0.757x + 0.214x^2$$

as the closest-fitting quadratic to the given data, in a least-squares sense.

As a practical matter, you would typically solve the normal equation (28.21) by multiplying y by A^{T} and then finding an LU decomposition of $A^{\mathrm{T}}A$. If A has full rank, the matrix $A^{\mathrm{T}}A$ is guaranteed to be nonsingular, because it is symmetric and positive-definite. (See Exercise D.1-2 and Theorem D.6.)

Figure 28.4 A least-squares fit of a curve of the form

$$c_1 + c_2 x + c_3 x^2 + c_4 \sin(2\pi x) + c_5 \cos(2\pi x)$$

for the carbon-dioxide concentrations measured in Mauna Loa, Hawaii from 1990[1] to 2019, where x is the number of years elapsed since 1990. This curve is the famous "Keeling curve," illustrating curve-fitting to nonpolynomial formulas. The sine and cosine terms allow modeling of seasonal variations in CO_2 concentrations. The red curve shows the measured CO_2 concentrations. The best fit, shown in black, has the form

$$352.83 + 1.39x + 0.02x^2 + 2.83 \sin(2\pi x) - 0.94 \cos(2\pi x) .$$

We close this section with an example in Figure 28.4, illustrating that a curve can also fit a nonpolynomial function. The curve confirms one aspect of climate change: that carbon dioxide (CO_2) concentrations have steadily increased over a period of 29 years. Linear and quadratic terms model the annual increase, and sine and cosine terms model seasonal variations.

[1] The year in which *Introduction to Algorithms* was first published.

Exercises

28.3-1
Prove that every diagonal element of a symmetric positive-definite matrix is positive.

28.3-2
Let $A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ be a 2×2 symmetric positive-definite matrix. Prove that its determinant $ac - b^2$ is positive by "completing the square" in a manner similar to that used in the proof of Lemma 28.5.

28.3-3
Prove that the maximum element in a symmetric positive-definite matrix lies on the diagonal.

28.3-4
Prove that the determinant of each leading submatrix of a symmetric positive-definite matrix is positive.

28.3-5
Let A_k denote the kth leading submatrix of a symmetric positive-definite matrix A. Prove that $\det(A_k)/\det(A_{k-1})$ is the kth pivot during LU decomposition, where, by convention, $\det(A_0) = 1$.

28.3-6
Find the function of the form

$$F(x) = c_1 + c_2 x \lg x + c_3 e^x$$

that is the best least-squares fit to the data points

$$(1, 1), (2, 1), (3, 3), (4, 8) .$$

28.3-7
Show that the pseudoinverse A^+ satisfies the following four equations:

$$\begin{aligned} AA^+A &= A , \\ A^+AA^+ &= A^+ , \\ (AA^+)^{\mathrm{T}} &= AA^+ , \\ (A^+A)^{\mathrm{T}} &= A^+A . \end{aligned}$$

Problems

28-1 Tridiagonal systems of linear equations

Consider the tridiagonal matrix

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix} .$$

a. Find an LU decomposition of A.

b. Solve the equation $Ax = (1\ 1\ 1\ 1\ 1)^{\mathrm{T}}$ by using forward and back substitution.

c. Find the inverse of A.

d. Show how to solve the equation $Ax = b$ for any $n \times n$ symmetric positive-definite, tridiagonal matrix A and any n-vector b in $O(n)$ time by performing an LU decomposition. Argue that any method based on forming A^{-1} is asymptotically more expensive in the worst case.

e. Show how to solve the equation $Ax = b$ for any $n \times n$ nonsingular, tridiagonal matrix A and any n-vector b in $O(n)$ time by performing an LUP decomposition.

28-2 Splines

A practical method for interpolating a set of points with a curve is to use ***cubic splines***. You are given a set $\{(x_i, y_i) : i = 0, 1, \ldots, n\}$ of $n + 1$ point-value pairs, where $x_0 < x_1 < \cdots < x_n$. Your goal is to fit a piecewise-cubic curve (spline) $f(x)$ to the points. That is, the curve $f(x)$ is made up of n cubic polynomials $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ for $i = 0, 1, \ldots, n-1$, where if x falls in the range $x_i \le x \le x_{i+1}$, then the value of the curve is given by $f(x) = f_i(x - x_i)$. The points x_i at which the cubic polynomials are "pasted" together are called ***knots***. For simplicity, assume that $x_i = i$ for $i = 0, 1, \ldots, n$.

To ensure continuity of $f(x)$, require that

$$\begin{aligned} f(x_i) &= f_i(0) = y_i , \\ f(x_{i+1}) &= f_i(1) = y_{i+1} \end{aligned}$$

for $i = 0, 1, \ldots, n-1$. To ensure that $f(x)$ is sufficiently smooth, also require the first derivative to be continuous at each knot:

$$f'(x_{i+1}) = f_i'(1) = f_{i+1}'(0)$$

for $i = 0, 1, \ldots, n-2$.

a. Suppose that for $i = 0, 1, \ldots, n$, in addition to the point-value pairs $\{(x_i, y_i)\}$, you are also given the first derivative $D_i = f'(x_i)$ at each knot. Express each coefficient a_i, b_i, c_i, and d_i in terms of the values y_i, y_{i+1}, D_i, and D_{i+1}. (Remember that $x_i = i$.) How quickly can you compute the $4n$ coefficients from the point-value pairs and first derivatives?

The question remains of how to choose the first derivatives of $f(x)$ at the knots. One method is to require the second derivatives to be continuous at the knots:

$$f''(x_{i+1}) = f_i''(1) = f_{i+1}''(0)$$

for $i = 0, 1, \ldots, n-2$. At the first and last knots, assume that $f''(x_0) = f_0''(0) = 0$ and $f''(x_n) = f_{n-1}''(1) = 0$. These assumptions make $f(x)$ a ***natural*** cubic spline.

b. Use the continuity constraints on the second derivative to show that for $i = 1, 2, \ldots, n-1$,

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}) . \tag{28.23}$$

c. Show that

$$2D_0 + D_1 = 3(y_1 - y_0) , \tag{28.24}$$

$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}) . \tag{28.25}$$

d. Rewrite equations (28.23)–(28.25) as a matrix equation involving the vector $D = (\ D_0 \quad D_1 \quad D_2 \quad \cdots \quad D_n\ )^{\mathrm{T}}$ of unknowns. What attributes does the matrix in your equation have?

e. Argue that a natural cubic spline can interpolate a set of $n+1$ point-value pairs in $O(n)$ time (see Problem 28-1).

f. Show how to determine a natural cubic spline that interpolates a set of $n+1$ points (x_i, y_i) satisfying $x_0 < x_1 < \cdots < x_n$, even when x_i is not necessarily equal to i. What matrix equation must your method solve, and how quickly does your algorithm run?

Chapter notes

Many excellent texts describe numerical and scientific computation in much greater detail than we have room for here. The following are especially readable: George

and Liu [180], Golub and Van Loan [192], Press, Teukolsky, Vetterling, and Flannery [365, 366], and Strang [422, 423].

Golub and Van Loan [192] discuss numerical stability. They show why $\det(A)$ is not necessarily a good indicator of the stability of a matrix A, proposing instead to use $\|A\|_\infty \|A^{-1}\|_\infty$, where $\|A\|_\infty = \max\left\{\sum_{j=1}^{n} |a_{ij}| : 1 \le i \le n\right\}$. They also address the question of how to compute this value without actually computing A^{-1}.

Gaussian elimination, upon which the LU and LUP decompositions are based, was the first systematic method for solving linear systems of equations. It was also one of the earliest numerical algorithms. Although it was known earlier, its discovery is commonly attributed to C. F. Gauss (1777–1855). In his famous paper [424], Strassen showed that an $n \times n$ matrix can be inverted in $O(n^{\lg 7})$ time. Winograd [460] originally proved that matrix multiplication is no harder than matrix inversion, and the converse is due to Aho, Hopcroft, and Ullman [5].

Another important matrix decomposition is the ***singular value decomposition***, or ***SVD***. The SVD factors an $m \times n$ matrix A into $A = Q_1 \Sigma Q_2^{\mathrm{T}}$, where Σ is an $m \times n$ matrix with nonzero values only on the diagonal, Q_1 is $m \times m$ with mutually orthonormal columns, and Q_2 is $n \times n$, also with mutually orthonormal columns. Two vectors are ***orthonormal*** if their inner product is 0 and each vector has a norm of 1. The books by Strang [422, 423] and Golub and Van Loan [192] contain good treatments of the SVD.

Strang [423] has an excellent presentation of symmetric positive-definite matrices and of linear algebra in general.

29 Linear Programming

Many problems take the form of maximizing or minimizing an objective, given limited resources and competing constraints. If you can specify the objective as a linear function of certain variables, and if you can specify the constraints on resources as equalities or inequalities on those variables, then you have a ***linear-programming problem***. Linear programs arise in a variety of practical applications. We begin by studying an application in electoral politics.

A political problem

Suppose that you are a politician trying to win an election. Your district has three different types of areas—urban, suburban, and rural. These areas have, respectively, 100,000, 200,000, and 50,000 registered voters. Although not all the registered voters actually go to the polls, you decide that to govern effectively, you would like at least half the registered voters in each of the three regions to vote for you. You are honorable and would never consider supporting policies you don't believe in. You realize, however, that certain issues may be more effective in winning votes in certain places. Your primary issues are preparing for a zombie apocalypse, equipping sharks with lasers, building highways for flying cars, and allowing dolphins to vote.

According to your campaign staff's research, you can estimate how many votes you win or lose from each population segment by spending $1,000 on advertising on each issue. This information appears in the table of Figure 29.1. In this table, each entry indicates the number of thousands of either urban, suburban, or rural voters who would be won over by spending $1,000 on advertising in support of a particular issue. Negative entries denote votes that would be lost. Your task is to figure out the minimum amount of money that you need to spend in order to win 50,000 urban votes, 100,000 suburban votes, and 25,000 rural votes.

You could, by trial and error, devise a strategy that wins the required number of votes, but the strategy you come up with might not be the least expensive one. For example, you could devote $20,000 of advertising to preparing for a zombie

policy	urban	suburban	rural
zombie apocalypse	−2	5	3
sharks with lasers	8	2	−5
highways for flying cars	0	0	10
dolphins voting	10	0	−2

Figure 29.1 The effects of policies on voters. Each entry describes the number of thousands of urban, suburban, or rural voters who could be won over by spending \$1,000 on advertising support of a policy on a particular issue. Negative entries denote votes that would be lost.

apocalypse, \$0 to equipping sharks with lasers, \$4,000 to building highways for flying cars, and \$9,000 to allowing dolphins to vote. In this case, you would win $(20 \cdot -2) + (0 \cdot 8) + (4 \cdot 0) + (9 \cdot 10) = 50$ thousand urban votes, $(20 \cdot 5) + (0 \cdot 2) + (4 \cdot 0) + (9 \cdot 0) = 100$ thousand suburban votes, and $(20 \cdot 3) + (0 \cdot -5) + (4 \cdot 10) + (9 \cdot -2) = 82$ thousand rural votes. You would win the exact number of votes desired in the urban and suburban areas and more than enough votes in the rural area. (In fact, according to your model, in the rural area you would receive more votes than there are voters.) In order to garner these votes, you would have paid for $20 + 0 + 4 + 9 = 33$ thousand dollars of advertising.

It's natural to wonder whether this strategy is the best possible. That is, can you achieve your goals while spending less on advertising? Additional trial and error might help you to answer this question, but a better approach is to formulate (or ***model***) this question mathematically.

The first step is to decide what decisions you have to make and to introduce variables that capture these decisions. Since you have four decisions, you introduce four ***decision variables***:

- x_1 is the number of thousands of dollars spent on advertising on preparing for a zombie apocalypse,
- x_2 is the number of thousands of dollars spent on advertising on equipping sharks with lasers,
- x_3 is the number of thousands of dollars spent on advertising on building highways for flying cars, and
- x_4 is the number of thousands of dollars spent on advertising on allowing dolphins to vote.

You then think about ***constraints***, which are limits, or restrictions, on the values that the decision variables can take. You can write the requirement that you win at least 50,000 urban votes as

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \,. \tag{29.1}$$

Similarly, you can write the requirements that you win at least 100,000 suburban votes and 25,000 rural votes as

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \tag{29.2}$$

and

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \; . \tag{29.3}$$

Any setting of the variables x_1, x_2, x_3, x_4 that satisfies inequalities (29.1)–(29.3) yields a strategy that wins a sufficient number of each type of vote.

Finally, you think about your ***objective***, which is the quantity that you wish to either minimize or maximize. In order to keep costs as small as possible, you would like to minimize the amount spent on advertising. That is, you want to minimize the expression

$$x_1 + x_2 + x_3 + x_4 \; . \tag{29.4}$$

Although negative advertising often occurs in political campaigns, there is no such thing as negative-cost advertising. Consequently, you require that

$$x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, \text{ and } x_4 \geq 0 \; . \tag{29.5}$$

Combining inequalities (29.1)–(29.3) and (29.5) with the objective of minimizing (29.4) produces what is known as a "linear program." We can format this problem tabularly as

$$\text{minimize} \quad x_1 + x_2 + x_3 + x_4 \tag{29.6}$$

subject to

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \tag{29.7}$$

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \tag{29.8}$$

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \tag{29.9}$$

$$x_1, x_2, x_3, x_4 \geq 0 \; . \tag{29.10}$$

The solution to this linear program yields your optimal strategy.

The remainder of this chapter covers how to formulate linear programs and is an introduction to modeling in general. Modeling refers to the general process of converting a problem into a mathematical form amenable to solution by an algorithm. Section 29.1 discusses briefly the algorithmic aspects of linear programming, although it does not include the details of a linear-programming algorithm. Throughout this book, we have seen ways to model problems, such as by shortest paths and connectivity in a graph. When modeling a problem as a linear program, you go through the steps used in this political example—identifying the decision variables, specifying the constraints, and formulating the objective function. In order to model a problem as a linear program, the constraints and objectives must be

linear. In Section 29.2, we will see several other examples of modeling via linear programs. Section 29.3 discusses duality, an important concept in linear programming and other optimization algorithms.

29.1 Linear programming formulations and algorithms

Linear programs take a particular form, which we will examine in this section. Multiple algorithms have been developed to solve linear programs. Some run in polynomial time, some do not, but they are all too complicated to show here. Instead, we will give an example that demonstrates some ideas behind the simplex algorithm, which is currently the most commonly deployed solution method.

General linear programs

In the general linear-programming problem, we wish to optimize a linear function subject to a set of linear inequalities. Given a set of real numbers $a_1, a_2, \ldots, a_n$ and a set of variables $x_1, x_2, \ldots, x_n$, we define a ***linear function*** f on those variables by

$$f(x_1, x_2, \ldots, x_n) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = \sum_{j=1}^{n} a_j x_j \ .$$

If b is a real number and f is a linear function, then the equation

$$f(x_1, x_2, \ldots, x_n) = b$$

is a ***linear equality*** and the inequalities

$$f(x_1, x_2, \ldots, x_n) \le b \quad \text{and} \quad f(x_1, x_2, \ldots, x_n) \ge b$$

are ***linear inequalities***. We use the general term ***linear constraints*** to denote either linear equalities or linear inequalities. Linear programming does not allow strict inequalities. Formally, a ***linear-programming problem*** is the problem of either minimizing or maximizing a linear function subject to a finite set of linear constraints. If minimizing, we call the linear program a ***minimization linear program***, and if maximizing, we call the linear program a ***maximization linear program***.

In order to discuss linear-programming algorithms and properties, it will be helpful to use a standard notation for the input. By convention, a maximization linear program takes as input n real numbers $c_1, c_2, \ldots, c_n$; m real numbers $b_1, b_2, \ldots, b_m$; and mn real numbers a_{ij} for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$.

The goal is to find n real numbers $x_1, x_2, \ldots, x_n$ that

$$\text{maximize} \quad \sum_{j=1}^{n} c_j x_j \tag{29.11}$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \le b_i \quad \text{for } i = 1, 2, \ldots, m \tag{29.12}$$

$$x_j \ge 0 \quad \text{for } j = 1, 2, \ldots, n \ . \tag{29.13}$$

We call expression (29.11) the ***objective function*** and the $n + m$ inequalities in lines (29.12) and (29.13) the ***constraints***. The n constraints in line (29.13) are the ***nonnegativity constraints***. It can sometimes be more convenient to express a linear program in a more compact form. If we create an $m \times n$ matrix $A = (a_{ij})$, an m-vector $b = (b_i)$, an n-vector $c = (c_j)$, and an n-vector $x = (x_j)$, then we can rewrite the linear program defined in (29.11)–(29.13) as

$$\text{maximize} \quad c^{\mathrm{T}} x \tag{29.14}$$

subject to

$$Ax \le b \tag{29.15}$$

$$x \ge 0 \ . \tag{29.16}$$

In line (29.14), $c^{\mathrm{T}} x$ is the inner product of two n-vectors. In inequality (29.15), Ax is the m-vector that is the product of an $m \times n$ matrix and an n-vector, and in inequality (29.16), $x \ge 0$ means that each entry of the vector x must be nonnegative. We call this representation the ***standard form*** for a linear program, and we adopt the convention that A, b, and c always have the dimensions given above.

The standard form above may not naturally correspond to real-life situations you are trying to model. For example, you might have equality constraints or variables that can take on negative values. Exercises 29.1-6 and 29.1-7 ask you to show how to convert any linear program into this standard form.

We now introduce terminology to describe solutions to linear programs. We denote a particular setting of the values in a variable, say x, by putting a bar over the variable name: $\bar{x}$. If $\bar{x}$ satisfies all the constraints, then it is a ***feasible solution***, but if it fails to satisfy at least one constraint, then it is an ***infeasible solution***. We say that a solution $\bar{x}$ has ***objective value*** $c^{\mathrm{T}} \bar{x}$. A feasible solution $\bar{x}$ whose objective value is maximum over all feasible solutions is an ***optimal solution***, and we call its objective value $c^{\mathrm{T}} \bar{x}$ the ***optimal objective value***. If a linear program has no feasible solutions, we say that the linear program is ***infeasible***, and otherwise, it is ***feasible***. The set of points that satisfy all the constraints is the ***feasible region***. If a linear program has some feasible solutions but does not have a finite optimal objective value, then the feasible region is ***unbounded*** and so is the linear program. Exercise 29.1-5 asks you to show that a linear program can have a finite optimal objective value even if the feasible region is unbounded.

One of the reasons for the power and popularity of linear programming is that linear programs can, in general, be solved efficiently. There are two classes of algorithms, known as ellipsoid algorithms and interior-point algorithms, that solve linear programs in polynomial time. In addition, the simplex algorithm is widely used. Although it does not run in polynomial time in the worst case, it tends to perform well in practice.

We will not give a detailed algorithm for linear programming, but will discuss a few important ideas. First, we will give an example of using a geometric procedure to solve a two-variable linear program. Although this example does not immediately generalize to an efficient algorithm for larger problems, it introduces some important concepts for linear programming and for optimization in general.

A two-variable linear program

Let us first consider the following linear program with two variables:

$$\text{maximize} \quad x_1 + x_2 \tag{29.17}$$

subject to

$$4x_1 - x_2 \le 8 \tag{29.18}$$

$$2x_1 + x_2 \le 10 \tag{29.19}$$

$$5x_1 - 2x_2 \ge -2 \tag{29.20}$$

$$x_1, x_2 \ge 0 \ . \tag{29.21}$$

Figure 29.2(a) graphs the constraints in the (x_1, x_2)-Cartesian coordinate system. The feasible region in the two-dimensional space (highlighted in blue in the figure) is convex.[1] Conceptually, you could evaluate the objective function $x_1 + x_2$ at each point in the feasible region, and then identify a point that has the maximum objective value as an optimal solution. For this example (and for most linear programs), however, the feasible region contains an infinite number of points, and so to solve this linear program, you need an efficient way to find a point that achieves the maximum objective value without explicitly evaluating the objective function at every point in the feasible region.

In two dimensions, you can optimize via a graphical procedure. The set of points for which $x_1 + x_2 = z$, for any z, is a line with a slope of -1. Plotting $x_1 + x_2 = 0$ produces the line with slope -1 through the origin, as in Figure 29.2(b). The intersection of this line and the feasible region is the set of feasible solutions that have an objective value of 0. In this case, that intersection of the line with the feasible region is the single point $(0, 0)$. More generally, for any value z, the

[1] An intuitive definition of a convex region is that it fulfills the requirement that for any two points in the region, all points on a line segment between them are also in the region.

Figure 29.2 **(a)** The linear program given in (29.18)–(29.21). Each constraint is represented by a line and a direction. The intersection of the constraints, which is the feasible region, is highlighted in blue. **(b)** The red lines show, respectively, the points for which the objective value is 0, 4, and 8. The optimal solution to the linear program is $x_1 = 2$ and $x_2 = 6$ with objective value 8.

intersection of the line $x_1 + x_2 = z$ and the feasible region is the set of feasible solutions that have objective value z. Figure 29.2(b) shows the lines $x_1 + x_2 = 0$, $x_1 + x_2 = 4$, and $x_1 + x_2 = 8$. Because the feasible region in Figure 29.2 is bounded, there must be some maximum value z for which the intersection of the line $x_1 + x_2 = z$ and the feasible region is nonempty. Any point in the feasible region that maximizes $x_1 + x_2$ is an optimal solution to the linear program, which in this case is the vertex of the feasible region at $x_1 = 2$ and $x_2 = 6$, with objective value 8.

It is no accident that an optimal solution to the linear program occurs at a vertex of the feasible region. The maximum value of z for which the line $x_1 + x_2 = z$ intersects the feasible region must be on the boundary of the feasible region, and thus the intersection of this line with the boundary of the feasible region is either a single vertex or a line segment. If the intersection is a single vertex, then there is just one optimal solution, and it is that vertex. If the intersection is a line segment, every point on that line segment must have the same objective value. In particular, both endpoints of the line segment are optimal solutions. Since each endpoint of a line segment is a vertex, there is an optimal solution at a vertex in this case as well.

Although you cannot easily graph linear programs with more than two variables, the same intuition holds. If you have three variables, then each constraint corresponds to a half-space in three-dimensional space. The intersection of these half-

spaces forms the feasible region. The set of points for which the objective function obtains a given value z is now a plane (assuming no degenerate conditions). If all coefficients of the objective function are nonnegative, and if the origin is a feasible solution to the linear program, then as you move this plane away from the origin, in a direction normal to the objective function, you find points of increasing objective value. (If the origin is not feasible or if some coefficients in the objective function are negative, the intuitive picture becomes slightly more complicated.) As in two dimensions, because the feasible region is convex, the set of points that achieve the optimal objective value must include a vertex of the feasible region. Similarly, if you have n variables, each constraint defines a half-space in n-dimensional space. We call the feasible region formed by the intersection of these half-spaces a ***simplex***. The objective function is now a hyperplane and, because of convexity, an optimal solution still occurs at a vertex of the simplex. Any algorithm for linear programming must also identify linear programs that have no solutions, as well as linear programs that have no finite optimal solution.

The ***simplex algorithm*** takes as input a linear program and returns an optimal solution. It starts at some vertex of the simplex and performs a sequence of iterations. In each iteration, it moves along an edge of the simplex from a current vertex to a neighboring vertex whose objective value is no smaller than that of the current vertex (and usually is larger.) The simplex algorithm terminates when it reaches a local maximum, which is a vertex from which all neighboring vertices have a smaller objective value. Because the feasible region is convex and the objective function is linear, this local optimum is actually a global optimum. In Section 29.3, we'll see an important concept called "duality," which we'll use to prove that the solution returned by the simplex algorithm is indeed optimal.

The simplex algorithm, when implemented carefully, often solves general linear programs quickly in practice. With some carefully contrived inputs, however, the simplex algorithm can require exponential time. The first polynomial-time algorithm for linear programming was the ***ellipsoid algorithm***, which runs slowly in practice. A second class of polynomial-time algorithms are known as ***interior-point methods***. In contrast to the simplex algorithm, which moves along the exterior of the feasible region and maintains a feasible solution that is a vertex of the simplex at each iteration, these algorithms move through the interior of the feasible region. The intermediate solutions, while feasible, are not necessarily vertices of the simplex, but the final solution is a vertex. For large inputs, interior-point algorithms can run as fast as, and sometimes faster than, the simplex algorithm. The chapter notes point you to more information about these algorithms.

If you add to a linear program the additional requirement that all variables take on integer values, you have an ***integer linear program***. Exercise 34.5-3 on page 1098 asks you to show that just finding a feasible solution to this problem is NP-hard. Since no polynomial-time algorithms are known for any NP-hard prob-

lems, there is no known polynomial-time algorithm for integer linear programming. In contrast, a general linear-programming problem can be solved in polynomial time.

Exercises

29.1-1
Consider the linear program

$$\begin{aligned}
&\text{minimize} \quad -2x_1 + 3x_2 \\
&\text{subject to} \\
&\qquad\qquad x_1 + x_2 = 7 \\
&\qquad\qquad x_1 - 2x_2 \le 4 \\
&\qquad\qquad x_1 \ge 0 \quad .
\end{aligned}$$

Give three feasible solutions to this linear program. What is the objective value of each one?

29.1-2
Consider the following linear program, which has a nonpositivity constraint:

$$\begin{aligned}
&\text{minimize} \quad 2x_1 + 7x_2 + x_3 \\
&\text{subject to} \\
&\qquad\qquad x_1 - x_3 = 7 \\
&\qquad\qquad 3x_1 + x_2 \ge 24 \\
&\qquad\qquad x_2 \ge 0 \\
&\qquad\qquad x_3 \le 0 \quad .
\end{aligned}$$

Give three feasible solutions to this linear program. What is the objective value of each one?

29.1-3
Show that the following linear program is infeasible:

$$\begin{aligned}
&\text{maximize} \quad 3x_1 - 2x_2 \\
&\text{subject to} \\
&\qquad\qquad x_1 + x_2 \le 2 \\
&\qquad\qquad -2x_1 - 2x_2 \le -10 \\
&\qquad\qquad x_1, x_2 \ge 0 \quad .
\end{aligned}$$

29.1-4

Show that the following linear program is unbounded:

$$\begin{array}{lrcl} \text{maximize} & x_1 - x_2 & & \\ \text{subject to} & & & \\ & -2x_1 + x_2 & \le & -1 \\ & -x_1 - 2x_2 & \le & -2 \\ & x_1, x_2 & \ge & 0 \quad . \end{array}$$

29.1-5

Give an example of a linear program for which the feasible region is not bounded, but the optimal objective value is finite.

29.1-6

Sometimes, in a linear program, you need to convert constraints from one form to another.

a. Show how to convert an equality constraint into an equivalent set of inequalities. That is, given a constraint $\sum_{j=1}^{n} a_{ij} x_j = b_i$, give a set of inequalities that will be satisfied if and only if $\sum_{j=1}^{n} a_{ij} x_j = b_i$,

b. Show how to convert an inequality constraint $\sum_{j=1}^{n} a_{ij} x_j \le b_i$ into an equality constraint and a nonnegativity constraint. You will need to introduce an additional variable s, and use the constraint that $s \ge 0$.

29.1-7

Explain how to convert a minimization linear program to an equivalent maximization linear program, and argue that your new linear program is equivalent to the original one.

29.1-8

In the political problem at the beginning of this chapter, there are feasible solutions that correspond to winning more voters than there actually are in the district. For example, you can set x_2 to 200, x_3 to 200, and $x_1 = x_4 = 0$. That solution is feasible, yet it seems to say that you will win 400,000 suburban voters, even though there are only 200,000 actual suburban voters. What constraints can you add to the linear program to ensure that you never seem to win more voters than there actually are? Even if you don't add these constraints, argue that the optimal solution to this linear program can never win more voters than there actually are in the district.

29.2 Formulating problems as linear programs

Linear programming has many applications. Any textbook on operations research is filled with examples of linear programming, and linear programming has become a standard tool taught to students in most business schools. The election scenario is one typical example. Here are two more examples:

- An airline wishes to schedule its flight crews. The Federal Aviation Administration imposes several constraints, such as limiting the number of consecutive hours that each crew member can work and insisting that a particular crew work only on one model of aircraft during each month. The airline wants to schedule crews on all of its flights using as few crew members as possible.
- An oil company wants to decide where to drill for oil. Siting a drill at a particular location has an associated cost and, based on geological surveys, an expected payoff of some number of barrels of oil. The company has a limited budget for locating new drills and wants to maximize the amount of oil it expects to find, given this budget.

Linear programs also model and solve graph and combinatorial problems, such as those appearing in this book. We have already seen a special case of linear programming used to solve systems of difference constraints in Section 22.4. In this section, we'll study how to formulate several graph and network-flow problems as linear programs. Section 35.4 uses linear programming as a tool to find an approximate solution to another graph problem.

Perhaps the most important aspect of linear programming is to be able to recognize when you can formulate a problem as a linear program. Once you cast a problem as a polynomial-sized linear program, you can solve it in polynomial time by the ellipsoid algorithm or interior-point methods. Several linear-programming software packages can solve problems efficiently, so that once the problem is in the form of a linear program, such a package can solve it.

We'll look at several concrete examples of linear-programming problems. We start with two problems that we have already studied: the single-source shortest-paths problem from Chapter 22 and the maximum-flow problem from Chapter 24. We then describe the minimum-cost-flow problem. (Although the minimum-cost-flow problem has a polynomial-time algorithm that is not based on linear programming, we won't describe the algorithm.) Finally, we describe the multicommodity-flow problem, for which the only known polynomial-time algorithm is based on linear programming.

When we solved graph problems in Part VI, we used attribute notation, such as $v.d$ and $(u, v).f$. Linear programs typically use subscripted variables rather than

objects with attached attributes, however. Therefore, when we express variables in linear programs, we indicate vertices and edges through subscripts. For example, we denote the shortest-path weight for vertex v not by $v.d$ but by d_v, and we denote the flow from vertex u to vertex v not by $(u, v).f$ but by f_{uv}. For quantities that are given as inputs to problems, such as edge weights or capacities, we continue to use notations such as $w(u, v)$ and $c(u, v)$.

Shortest paths

We can formulate the single-source shortest-paths problem as a linear program. We'll focus on how to formulate the single-pair shortest-path problem, leaving the extension to the more general single-source shortest-paths problem as Exercise 29.2-2.

In the single-pair shortest-path problem, the input is a weighted, directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights, a source vertex s, and destination vertex t. The goal is to compute the value d_t, which is the weight of a shortest path from s to t. To express this problem as a linear program, you need to determine a set of variables and constraints that define when you have a shortest path from s to t. The triangle inequality (Lemma 22.10 on page 633) gives $d_v \le d_u + w(u, v)$ for each edge $(u, v) \in E$. The source vertex initially receives a value $d_s = 0$, which never changes. Thus the following linear program expresses the shortest-path weight from s to t:

$$\text{maximize} \quad d_t \tag{29.22}$$

subject to

$$d_v \le d_u + w(u, v) \quad \text{for each edge } (u, v) \in E \tag{29.23}$$

$$d_s = 0 \ . \tag{29.24}$$

You might be surprised that this linear program maximizes an objective function when it is supposed to compute shortest paths. Minimizing the objective function would be a mistake, because when all the edge weights are nonnegative, setting $\bar{d}_v = 0$ for all $v \in V$ (recall that a bar over a variable name denotes a specific setting of the variable's value) would yield an optimal solution to the linear program without solving the shortest-paths problem. Maximizing is the right thing to do because an optimal solution to the shortest-paths problem sets each $\bar{d}_v$ to $\min\{\bar{d}_u + w(u, v) : u \in V \text{ and } (u, v) \in E\}$, so that $\bar{d}_v$ is the largest value that is less than or equal to all of the values in the set $\{\bar{d}_u + w(u, v)\}$. Therefore, it makes sense to maximize d_v for all vertices v on a shortest path from s to t subject to these constraints, and maximizing d_t achieves this goal.

This linear program has $|V|$ variables d_v, one for each vertex $v \in V$. It also has $|E| + 1$ constraints: one for each edge, plus the additional constraint that the source vertex's shortest-path weight always has the value 0.

Maximum flow

Next, let's express the maximum-flow problem as a linear program. Recall that the input is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$, and two distinguished vertices: a source s and a sink t. As defined in Section 24.1, a flow is a nonnegative real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the capacity constraint and flow conservation. A maximum flow is a flow that satisfies these constraints and maximizes the flow value, which is the total flow coming out of the source minus the total flow into the source. A flow, therefore, satisfies linear constraints, and the value of a flow is a linear function. Recalling also that we assume that $c(u, v) = 0$ if $(u, v) \notin E$ and that there are no antiparallel edges, the maximum-flow problem can be expressed as a linear program:

$$\text{maximize} \quad \sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} \tag{29.25}$$

subject to

$$f_{uv} \leq c(u, v) \quad \text{for each } u, v \in V \tag{29.26}$$

$$\sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv} \quad \text{for each } u \in V - \{s, t\} \tag{29.27}$$

$$f_{uv} \geq 0 \quad \text{for each } u, v \in V \ . \tag{29.28}$$

This linear program has $|V|^2$ variables, corresponding to the flow between each pair of vertices, and it has $2|V|^2 + |V| - 2$ constraints.

It is usually more efficient to solve a smaller-sized linear program. The linear program in (29.25)–(29.28) has, for ease of notation, a flow and capacity of 0 for each pair of vertices u, v with $(u, v) \notin E$. It is more efficient to rewrite the linear program so that it has $O(V + E)$ constraints. Exercise 29.2-4 asks you to do so.

Minimum-cost flow

In this section, we have used linear programming to solve problems for which we already knew efficient algorithms. In fact, an efficient algorithm designed specifically for a problem, such as Dijkstra's algorithm for the single-source shortest-paths problem, will often be more efficient than linear programming, both in theory and in practice.

The real power of linear programming comes from the ability to solve new problems. Recall the problem faced by the politician in the beginning of this chapter. The problem of obtaining a sufficient number of votes, while not spending too much money, is not solved by any of the algorithms that we have studied in this book, yet it can be solved by linear programming. Books abound with such real-world problems that linear programming can solve. Linear programming is also

Figure 29.3 **(a)** An example of a minimum-cost-flow problem. Capacities are denoted by c and costs by a. Vertex s is the source, and vertex t is the sink. The goal is to send 4 units of flow from s to t. **(b)** A solution to the minimum-cost flow problem in which 4 units of flow are sent from s to t. For each edge, the flow and capacity are written as flow/capacity.

particularly useful for solving variants of problems for which we may not already know of an efficient algorithm.

Consider, for example, the following generalization of the maximum-flow problem. Suppose that, in addition to a capacity $c(u, v)$ for each edge (u, v), you are given a real-valued cost $a(u, v)$. As in the maximum-flow problem, assume that $c(u, v) = 0$ if $(u, v) \notin E$ and that there are no antiparallel edges. If you send f_{uv} units of flow over edge (u, v), you incur a cost of $a(u, v) \cdot f_{uv}$. You are also given a flow demand d. You wish to send d units of flow from s to t while minimizing the total cost $\sum_{(u,v)\in E} a(u, v) \cdot f_{uv}$ incurred by the flow. This problem is known as the ***minimum-cost-flow problem***.

Figure 29.3(a) shows an example of the minimum-cost-flow problem, with a goal of sending 4 units of flow from s to t while incurring the minimum total cost. Any particular legal flow, that is, a function f satisfying constraints (29.26)–(29.28), incurs a total cost of $\sum_{(u,v)\in E} a(u, v) \cdot f_{uv}$. What is the particular 4-unit flow that minimizes this cost? Figure 29.3(b) shows an optimal solution, with total cost $\sum_{(u,v)\in E} a(u, v) \cdot f_{uv} = (2 \cdot 2) + (5 \cdot 2) + (3 \cdot 1) + (7 \cdot 1) + (1 \cdot 3) = 27$.

There are polynomial-time algorithms specifically designed for the minimum-cost-flow problem, but they are beyond the scope of this book. The minimum-cost-flow problem can be expressed as a linear program, however. The linear program looks similar to the one for the maximum-flow problem with the additional constraint that the value of the flow must be exactly d units, and with the new objective function of minimizing the cost:

$$\text{minimize} \quad \sum_{(u,v)\in E} a(u,v) \cdot f_{uv} \tag{29.29}$$

subject to

$$\begin{aligned}
f_{uv} &\le c(u,v) && \text{for each } u, v \in V \\
\sum_{v\in V} f_{vu} - \sum_{v\in V} f_{uv} &= 0 && \text{for each } u \in V - \{s,t\} \\
\sum_{v\in V} f_{sv} - \sum_{v\in V} f_{vs} &= d \\
f_{uv} &\ge 0 && \text{for each } u, v \in V \quad .
\end{aligned} \tag{29.30}$$

Multicommodity flow

As a final example, let's consider another flow problem. Suppose that the Lucky Puck company from Section 24.1 decides to diversify its product line and ship not only hockey pucks, but also hockey sticks and hockey helmets. Each piece of equipment is manufactured in its own factory, has its own warehouse, and must be shipped, each day, from factory to warehouse. The sticks are manufactured in Vancouver and are needed in Saskatoon, and the helmets are manufactured in Edmonton and must be shipped to Regina. The capacity of the shipping network does not change, however, and the different items, or ***commodities***, must share the same network.

This example is an instance of a ***multicommodity-flow problem***. The input to this problem is once again a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \ge 0$. As in the maximum-flow problem, implicitly assume that $c(u, v) = 0$ for $(u, v) \notin E$ and that the graph has no antiparallel edges. In addition, there are k different commodities, $K_1, K_2, \ldots, K_k$, with commodity i specified by the triple $K_i = (s_i, t_i, d_i)$. Here, vertex s_i is the source of commodity i, vertex t_i is the sink of commodity i, and d_i is the demand for commodity i, which is the desired flow value for the commodity from s_i to t_i. We define a flow for commodity i, denoted by f_i, (so that f_{iuv} is the flow of commodity i from vertex u to vertex v) to be a real-valued function that satisfies the flow-conservation and capacity constraints. We define f_{uv}, the ***aggregate flow***, to be the sum of the various commodity flows, so that $f_{uv} = \sum_{i=1}^{k} f_{iuv}$. The aggregate flow on edge (u, v) must be no more than the capacity of edge (u, v). This problem has no objective function: the question is to determine whether such a flow exists. Thus the linear program for this problem has a "null" objective function:

$$
\begin{array}{lrcll}
\text{minimize} & & & 0 & \\
\text{subject to} & & & & \\
& \displaystyle\sum_{i=1}^{k} f_{iuv} & \le & c(u,v) & \text{for each } u, v \in V \\
& \displaystyle\sum_{v \in V} f_{iuv} - \sum_{v \in V} f_{ivu} & = & 0 & \text{for each } i = 1, 2, \ldots, k \text{ and} \\
& & & & \text{for each } u \in V - \{s_i, t_i\} \\
& \displaystyle\sum_{v \in V} f_{i,s_i,v} - \sum_{v \in V} f_{i,v,s_i} & = & d_i & \text{for each } i = 1, 2, \ldots, k \\
& f_{iuv} & \ge & 0 & \text{for each } u, v \in V \text{ and} \\
& & & & \text{for each } i = 1, 2, \ldots, k \ .
\end{array}
$$

The only known polynomial-time algorithm for this problem expresses it as a linear program and then solves it with a polynomial-time linear-programming algorithm.

Exercises

29.2-1
Write out explicitly the linear program corresponding to finding the shortest path from vertex s to vertex x in Figure 22.2(a) on page 609.

29.2-2
Given a graph G, write a linear program for the single-source shortest-paths problem. The solution should have the property that d_v is the shortest-path weight from the source vertex s to v for each vertex $v \in V$.

29.2-3
Write out explicitly the linear program corresponding to finding the maximum flow in Figure 24.1(a).

29.2-4
Rewrite the linear program for maximum flow (29.25)–(29.28) so that it uses only $O(V + E)$ constraints.

29.2-5
Write a linear program that, given a bipartite graph $G = (V, E)$, solves the maximum-bipartite-matching problem.

29.2-6
There can be more than one way to model a particular problem as a linear program. This exercise gives an alternative formulation for the maximum-flow problem. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_p\}$ be the set of *all* possible directed simple paths from source s

to sink t. Using decision variables $x_1, \ldots, x_p$, where x_i is the amount of flow on path i, formulate a linear program for the maximum-flow problem. What is an upper bound on p, the number of directed simple paths from s to t?

29.2-7

In the ***minimum-cost multicommodity-flow problem***, the input is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$ and a cost $a(u, v)$. As in the multicommodity-flow problem, there are k different commodities, $K_1, K_2, \ldots, K_k$, with commodity i specified by the triple $K_i = (s_i, t_i, d_i)$. We define the flow f_i for commodity i and the aggregate flow f_{uv} on edge (u, v) as in the multicommodity-flow problem. A feasible flow is one in which the aggregate flow on each edge (u, v) is no more than the capacity of edge (u, v). The cost of a flow is $\sum_{u,v \in V} a(u, v) \cdot f_{uv}$, and the goal is to find the feasible flow of minimum cost. Express this problem as a linear program.

29.3 Duality

We will now introduce a powerful concept called ***linear-programming duality***. In general, given a maximization problem, duality allows you to formulate a related minimization problem that has the same objective value. The idea of duality is actually more general than linear programming, but we restrict our attention to linear programming in this section.

Duality enables us to prove that a solution is indeed optimal. We saw an example of duality in Chapter 24 with Theorem 24.6, the max-flow min-cut theorem. Suppose that, given an instance of a maximum-flow problem, you find a flow f with value $|f|$. How do you know whether f is a maximum flow? By the max-flow min-cut theorem, if you can find a cut whose value is also $|f|$, then you have verified that f is indeed a maximum flow. This relationship provides an example of duality: given a maximization problem, define a related minimization problem such that the two problems have the same optimal objective values.

Given a linear program in standard form in which the objective is to maximize, let's see how to formulate a ***dual*** linear program in which the objective is to minimize and whose optimal value is identical to that of the original linear program. When referring to dual linear programs, we call the original linear program the ***primal***.

Given the primal linear program

$$\text{maximize} \quad \sum_{j=1}^{n} c_j x_j \tag{29.31}$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \le b_i \quad \text{for } i = 1, 2, \ldots, m \tag{29.32}$$

$$x_j \ge 0 \quad \text{for } j = 1, 2, \ldots, n \ , \tag{29.33}$$

its dual is

$$\text{minimize} \quad \sum_{i=1}^{m} b_i y_i \tag{29.34}$$

subject to

$$\sum_{i=1}^{m} a_{ij} y_i \ge c_j \quad \text{for } j = 1, 2, \ldots, n \tag{29.35}$$

$$y_i \ge 0 \quad \text{for } i = 1, 2, \ldots, m \ . \tag{29.36}$$

Mechanically, to form the dual, change the maximization to a minimization, exchange the roles of coefficients on the right-hand sides and in the objective function, and replace each $\le$ by $\ge$. Each of the m constraints in the primal corresponds to a variable y_i in the dual. Likewise, each of the n constraints in the dual corresponds to a variable x_j in the primal. For example, consider the following primal linear program:

$$\text{maximize} \quad 3x_1 + x_2 + 4x_3 \tag{29.37}$$

subject to

$$x_1 + x_2 + 3x_3 \le 30 \tag{29.38}$$

$$2x_1 + 2x_2 + 5x_3 \le 24 \tag{29.39}$$

$$4x_1 + x_2 + 2x_3 \le 36 \tag{29.40}$$

$$x_1, x_2, x_3 \ge 0 \ . \tag{29.41}$$

Its dual is

$$\text{minimize} \quad 30y_1 + 24y_2 + 36y_3 \tag{29.42}$$

subject to

$$y_1 + 2y_2 + 4y_3 \ge 3 \tag{29.43}$$

$$y_1 + 2y_2 + y_3 \ge 1 \tag{29.44}$$

$$3y_1 + 5y_2 + 2y_3 \ge 4 \tag{29.45}$$

$$y_1, y_2, y_3 \ge 0 \ . \tag{29.46}$$

Although forming the dual can be considered a mechanical operation, there is an intuitive explanation. Consider the primal maximization problem (29.37)–(29.41). Each constraint gives an upper bound on the objective function. In addition, if you

take one or more constraints and add together nonnegative multiples of them, you get a valid constraint. For example, you can add constraints (29.38) and (29.39) to obtain the constraint $3x_1 + 3x_2 + 8x_3 \le 54$. Any feasible solution to the primal must satisfy this new constraint, but there is something else interesting about it. Comparing this new constraint to the objective function (29.37), you can see that for each variable, the corresponding coefficient is at least as large as the coefficient in the objective function. Thus, since the variables x_1, x_2 and x_3 are nonnegative, we have that

$$3x_1 + x_2 + 4x_3 \;\le\; 3x_1 + 3x_2 + 8x_3 \;\le\; 54 ,$$

and so the solution value to the primal is at most 54. In other words, adding these two constraints together has generated an upper bound on the objective value.

In general, for any nonnegative multipliers y_1, y_2, and y_3, you can generate a constraint

$$y_1(x_1+x_2+3x_3)+y_2(2x_1+2x_2+5x_3)+y_3(4x_1+x_2+2x_3) \le 30y_1+24y_2+36y_3$$

from the primal constraints or, by distributing and regrouping,

$$(y_1+2y_2+4y_3)x_1+(y_1+2y_2+y_3)x_2+(3y_1+5y_2+2y_3)x_3 \le 30y_1+24y_2+36y_3 .$$

Now, as long as this constraint has coefficients of x_1, x_2, and x_3 that are at least their objective-function coefficients, it is a valid upper bound. That is, as long as

$$\begin{aligned} y_1 + 2y_2 + 4y_3 &\ge 3 , \\ y_1 + 2y_2 + y_3 &\ge 1 , \\ 3y_1 + 5y_2 + 2y_3 &\ge 4 , \end{aligned}$$

you have a valid upper bound of $30y_1+24y_2+36y_3$. The multipliers y_1, y_2, and y_3 must be nonnegative, because otherwise you cannot combine the inequalities. Of course, you would like the upper bound to be as small as possible, and so you want to choose y to minimize $30y_1 + 24y_2 + 36y_3$. Observe that we have just described the dual linear program as the problem of finding the smallest possible upper bound on the primal.

We'll formalize this idea and show in Theorem 29.4 that, if the linear program and its dual are feasible and bounded, then the optimal value of the dual linear program is always equal to the optimal value of the primal linear program. We begin by demonstrating ***weak duality***, which states that any feasible solution to the primal linear program has a value no greater than that of any feasible solution to the dual linear program.

Lemma 29.1 (Weak linear-programming duality)
Let $\bar{x}$ be any feasible solution to the primal linear program in (29.31)–(29.33), and let $\bar{y}$ be any feasible solution to its dual linear program in (29.34)–(29.36). Then

$$\sum_{j=1}^{n} c_j \bar{x}_j \le \sum_{i=1}^{m} b_i \bar{y}_i \ .$$

Proof We have

$$\begin{aligned}
\sum_{j=1}^{n} c_j \bar{x}_j &\le \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j \quad \text{(by inequalities (29.35))} \\
&= \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} \bar{x}_j \right) \bar{y}_i \\
&\le \sum_{i=1}^{m} b_i \bar{y}_i \qquad\qquad \text{(by inequalities (29.32))} \ .
\end{aligned}$$

■

Corollary 29.2
Let $\bar{x}$ be a feasible solution to the primal linear program in (29.31)–(29.33), and let $\bar{y}$ be a feasible solution to its dual linear program in (29.34)–(29.36). If

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i \ ,$$

then $\bar{x}$ and $\bar{y}$ are optimal solutions to the primal and dual linear programs, respectively.

Proof By Lemma 29.1, the objective value of a feasible solution to the primal cannot exceed that of a feasible solution to the dual. The primal linear program is a maximization problem and the dual is a minimization problem. Thus, if feasible solutions $\bar{x}$ and $\bar{y}$ have the same objective value, neither can be improved. ■

We now show that, at optimality, the primal and dual objective values are indeed equal. To prove linear programming duality, we will require one lemma from linear algebra, known as Farkas's lemma, the proof of which Problem 29-4 asks you to provide. Farkas's lemma can take several forms, each of which is about when a set of linear equalities has a solution. In stating the lemma, we use $m + 1$ as a dimension because it matches our use below.

Lemma 29.3 (Farkas's lemma)
Given $M \in \mathbb{R}^{(m+1)\times n}$ and $g \in \mathbb{R}^{m+1}$, exactly one of the following statements is true:

1. There exists $v \in \mathbb{R}^n$ such that $Mv \le g$,
2. There exists $w \in \mathbb{R}^{m+1}$ such that $w \ge 0$, $w^{\mathrm{T}} M = 0$ (an n-vector of all zeros), and $w^{\mathrm{T}} g < 0$. ■

Theorem 29.4 (Linear-programming duality)
Given the primal linear program in (29.31)–(29.33) and its corresponding dual in (29.34)–(29.36), if both are feasible and bounded, then for optimal solutions x^* and y^*, we have $c^{\mathrm{T}}x^* = b^{\mathrm{T}}y^*$.

Proof Let $\mu = b^{\mathrm{T}}y^*$ be the optimal value of the dual linear program given in (29.34)–(29.36). Consider an augmented set of primal constraints in which we add a constraint to (29.31)–(29.33) that the objective value is at least μ. We write out this ***augmented primal*** as

$$Ax \le b\,, \tag{29.47}$$

$$c^{\mathrm{T}}x \ge \mu\,. \tag{29.48}$$

We can multiply (29.48) through by -1 and rewrite (29.47)–(29.48) as

$$\begin{pmatrix} A \\ -c^{\mathrm{T}} \end{pmatrix} x \le \begin{pmatrix} b \\ -\mu \end{pmatrix}. \tag{29.49}$$

Here, $\begin{pmatrix} A \\ -c^{\mathrm{T}} \end{pmatrix}$ denotes an $(m+1)\times n$ matrix, x is an n-vector, and $\begin{pmatrix} b \\ -\mu \end{pmatrix}$ denotes an $(m+1)$-vector.

We claim that if there is a feasible solution $\bar{x}$ to the augmented primal, then the theorem is proved. To establish this claim, observe that $\bar{x}$ is also a feasible solution to the original primal and that it has objective value at least μ. We can then apply Lemma 29.1, which states that the objective value of the primal is at most μ, to complete the proof of the theorem.

It therefore remains to show that the augmented primal has a feasible solution. Suppose, for the purpose of contradiction, that the augmented primal is infeasible, which means that there is no $v \in \mathbb{R}^n$ such that $\begin{pmatrix} A \\ -c^{\mathrm{T}} \end{pmatrix} v \le \begin{pmatrix} b \\ -\mu \end{pmatrix}$. We can apply Farkas's lemma, Lemma 29.3, to inequalty (29.49) with

$$M = \begin{pmatrix} A \\ -c^{\mathrm{T}} \end{pmatrix} \text{ and } g = \begin{pmatrix} b \\ -\mu \end{pmatrix}.$$

Because the augmented primal is infeasible, condition 1 of Farkas's lemma does not hold. Therefore, condition 2 must apply, so that there must exist a $w \in \mathbb{R}^{m+1}$ such that $w \ge 0$, $w^{\mathrm{T}}M = 0$, and $w^{\mathrm{T}}g < 0$. Let's write w as $w = \begin{pmatrix} \bar{y} \\ \lambda \end{pmatrix}$ for some $\bar{y} \in \mathbb{R}^m$ and $\lambda \in \mathbb{R}$, where $\bar{y} \ge 0$ and $\lambda \ge 0$. Substituting for w, M, and g in condition 2 gives

$$\begin{pmatrix} \bar{y} \\ \lambda \end{pmatrix}^{\mathrm{T}} \begin{pmatrix} A \\ -c^{\mathrm{T}} \end{pmatrix} = 0 \text{ and } \begin{pmatrix} \bar{y} \\ \lambda \end{pmatrix}^{\mathrm{T}} \begin{pmatrix} b \\ -\mu \end{pmatrix} < 0\,.$$

Unpacking the matrix notation gives

$$\bar{y}^{\mathrm{T}} A - \lambda c^{\mathrm{T}} = 0 \text{ and } \bar{y}^{\mathrm{T}} b - \lambda \mu < 0 . \tag{29.50}$$

We now show that the requirements in (29.50) contradict the assumption that μ is the optimal solution value for the dual linear program. We consider two cases.

The first case is when $\lambda = 0$. In this case, (29.50) simplifies to

$$\bar{y}^{\mathrm{T}} A = 0 \text{ and } \bar{y}^{\mathrm{T}} b < 0 . \tag{29.51}$$

We'll now construct a dual feasible solution y' with an objective value smaller than $b^{\mathrm{T}} y^*$. Set $y' = y^* + \epsilon \bar{y}$, for any $\epsilon > 0$. Since

$$\begin{aligned} y'^{\mathrm{T}} A &= (y^* + \epsilon \bar{y})^{\mathrm{T}} A \\ &= y^{*\mathrm{T}} A + \epsilon \bar{y}^{\mathrm{T}} A \\ &= y^{*\mathrm{T}} A && \text{(by (29.51))} \\ &\geq c^{\mathrm{T}} && \text{(because } y^* \text{ is feasible)} , \end{aligned}$$

y' is feasible. Now consider the objective value

$$\begin{aligned} b^{\mathrm{T}} y' &= b^{\mathrm{T}} (y^* + \epsilon \bar{y}) \\ &= b^{\mathrm{T}} y^* + \epsilon b^{\mathrm{T}} \bar{y} \\ &< b^{\mathrm{T}} y^* , \end{aligned}$$

where the last inequality follows because $\epsilon > 0$ and, by (29.51), $\bar{y}^{\mathrm{T}} b = b^{\mathrm{T}} \bar{y} < 0$ (since both $\bar{y}^{\mathrm{T}} b$ and $b^{\mathrm{T}} \bar{y}$ are the inner product of b and $\bar{y}$), and so their product is negative. Thus we have a feasible dual solution of value less than μ, which contradicts μ being the optimal objective value.

We now consider the second case, where $\lambda > 0$. In this case, we can take (29.50) and divide through by λ to obtain

$$(\bar{y}^{\mathrm{T}}/\lambda) A - (\lambda/\lambda) c^{\mathrm{T}} = 0 \text{ and } (\bar{y}^{\mathrm{T}}/\lambda) b - (\lambda/\lambda) \mu < 0 . \tag{29.52}$$

Now set $y' = \bar{y}/\lambda$ in (29.52), giving

$$y'^{\mathrm{T}} A = c^{\mathrm{T}} \text{ and } y'^{\mathrm{T}} b < \mu .$$

Thus, y' is a feasible dual solution with objective value strictly less than μ, a contradiction. We conclude that the augmented primal has a feasible solution, and the theorem is proved. ■

Fundamental theorem of linear programming

We conclude this chapter by stating the fundamental theorem of linear programming, which extends Theorem 29.4 to the cases when the linear program may be either feasible or unbounded. Exercise 29.3-8 asks you to provide the proof.

Theorem 29.5 (Fundamental theorem of linear programming)
Any linear program, given in standard form, either

1. has an optimal solution with a finite objective value,
2. is infeasible, or
3. is unbounded. ■

Exercises

29.3-1
Formulate the dual of the linear program given in lines (29.6)–(29.10) on page 852.

29.3-2
You have a linear program that is not in standard form. You could produce the dual by first converting it to standard form, and then taking the dual. It would be more convenient, however, to produce the dual directly. Explain how to directly take the dual of an arbitrary linear program.

29.3-3
Write down the dual of the maximum-flow linear program, as given in lines (29.25)–(29.28) on page 862. Explain how to interpret this formulation as a minimum-cut problem.

29.3-4
Write down the dual of the minimum-cost-flow linear program, as given in lines (29.29)–(29.30) on page 864. Explain how to interpret this problem in terms of graphs and flows.

29.3-5
Show that the dual of the dual of a linear program is the primal linear program.

29.3-6
Which result from Chapter 24 can be interpreted as weak duality for the maximum-flow problem?

29.3-7
Consider the following 1-variable primal linear program:

$$\begin{aligned} &\text{maximize} \quad tx \\ &\text{subject to} \\ &\qquad\qquad rx \le s \\ &\qquad\qquad\ \ x \ge 0 \ , \end{aligned}$$

where r, s, and t are arbitrary real numbers. State for which values of r, s, and t you can assert that

1. Both the primal linear program and its dual have optimal solutions with finite objective values.
2. The primal is feasible, but the dual is infeasible.
3. The dual is feasible, but the primal is infeasible.
4. Neither the primal nor the dual is feasible.

29.3-8
Prove the fundamental theorem of linear programming, Theorem 29.5.

Problems

29-1 Linear-inequality feasibility
Given a set of m linear inequalities on n variables $x_1, x_2, \ldots, x_n$, the ***linear-inequality feasibility problem*** asks whether there is a setting of the variables that simultaneously satisfies each of the inequalities.

a. Given an algorithm for the linear-programming problem, show how to use it to solve a linear-inequality feasibility problem. The number of variables and constraints that you use in the linear-programming problem should be polynomial in n and m.

b. Given an algorithm for the linear-inequality feasibility problem, show how to use it to solve a linear-programming problem. The number of variables and linear inequalities that you use in the linear-inequality feasibility problem should be polynomial in n and m, the number of variables and constraints in the linear program.

29-2 Complementary slackness
Complementary slackness describes a relationship between the values of primal variables and dual constraints and between the values of dual variables and primal constraints. Let $\bar{x}$ be a feasible solution to the primal linear program given in (29.31)–(29.33), and let $\bar{y}$ be a feasible solution to the dual linear program given in (29.34)–(29.36). Complementary slackness states that the following conditions are necessary and sufficient for $\bar{x}$ and $\bar{y}$ to be optimal:

$$\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j \quad \text{or} \quad \bar{x}_j = 0 \quad \text{for } j = 1, 2, \ldots, n$$

and

$$\sum_{j=1}^{n} a_{ij}\bar{x}_j = b_i \quad \text{or} \quad \bar{y}_i = 0 \quad \text{for } i = 1, 2, \ldots, m\ .$$

a. Verify that complementary slackness holds for the linear program in lines (29.37)–(29.41).

b. Prove that complementary slackness holds for any primal linear program and its corresponding dual.

c. Prove that a feasible solution $\bar{x}$ to a primal linear program given in lines (29.31)–(29.33) is optimal if and only if there exist values $\bar{y} = (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$ such that

1. $\bar{y}$ is a feasible solution to the dual linear program given in (29.34)–(29.36),
2. $\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j$ for all j such that $\bar{x}_j > 0$, and
3. $\bar{y}_i = 0$ for all i such that $\sum_{j=1}^{n} a_{ij}\bar{x}_j < b_i$.

29-3 Integer linear programming

An ***integer linear-programming problem*** is a linear-programming problem with the additional constraint that the variables x must take on integer values. Exercise 34.5-3 on page 1098 shows that just determining whether an integer linear program has a feasible solution is NP-hard, which means that there is no known polynomial-time algorithm for this problem.

a. Show that weak duality (Lemma 29.1) holds for an integer linear program.

b. Show that duality (Theorem 29.4) does not always hold for an integer linear program.

c. Given a primal linear program in standard form, let P be the optimal objective value for the primal linear program, D be the optimal objective value for its dual, IP be the optimal objective value for the integer version of the primal (that is, the primal with the added constraint that the variables take on integer values), and ID be the optimal objective value for the integer version of the dual. Assuming that both the primal integer program and the dual integer program are feasible and bounded, show that

$$IP \le P = D \le ID\ .$$

29-4 Farkas's lemma

Prove Farkas's lemma, Lemma 29.3.

29-5 Minimum-cost circulation

This problem considers a variant of the minimum-cost-flow problem from Section 29.2 in which there is no demand, source, or sink. Instead, the input, as before, contains a flow network, capacity constraints $c(u, v)$, and edge costs $a(u, v)$. A flow is feasible if it satisfies the capacity constraint on every edge and flow conservation at *every* vertex. The goal is to find, among all feasible flows, the one of minimum cost. We call this problem the ***minimum-cost-circulation problem.***

a. Formulate the minimum-cost-circulation problem as a linear program.

b. Suppose that for all edges $(u, v) \in E$, we have $a(u, v) > 0$. What does an optimal solution to the minimum-cost-circulation problem look like?

c. Formulate the maximum-flow problem as a minimum-cost-circulation problem linear program. That is, given a maximum-flow problem instance $G = (V, E)$ with source s, sink t and edge capacities c, create a minimum-cost-circulation problem by giving a (possibly different) network $G' = (V', E')$ with edge capacities c' and edge costs a' such that you can derive a solution to the maximum-flow problem from a solution to the minimum-cost-circulation problem.

d. Formulate the single-source shortest-path problem as a minimum-cost-circulation problem linear program.

Chapter notes

This chapter only begins to study the wide field of linear programming. A number of books are devoted exclusively to linear programming, including those by Chvátal [94], Gass [178], Karloff [246], Schrijver [398], and Vanderbei [444]. Many other books give a good coverage of linear programming, including those by Papadimitriou and Steiglitz [353] and Ahuja, Magnanti, and Orlin [7]. The coverage in this chapter draws on the approach taken by Chvátal.

The simplex algorithm for linear programming was invented by G. Dantzig in 1947. Shortly after, researchers discovered how to formulate a number of problems in a variety of fields as linear programs and solve them with the simplex algorithm. As a result, applications of linear programming flourished, along with several algorithms. Variants of the simplex algorithm remain the most popular

methods for solving linear-programming problems. This history appears in a number of places, including the notes in [94] and [246].

The ellipsoid algorithm was the first polynomial-time algorithm for linear programming and is due to L. G. Khachian in 1979. It was based on earlier work by N. Z. Shor, D. B. Judin, and A. S. Nemirovskii. Grötschel, Lovász, and Schrijver [201] describe how to use the ellipsoid algorithm to solve a variety of problems in combinatorial optimization. To date, the ellipsoid algorithm does not appear to be competitive with the simplex algorithm in practice.

Karmarkar's paper [247] includes a description of the first interior-point algorithm. Many subsequent researchers designed interior-point algorithms. Good surveys appear in the article of Goldfarb and Todd [189] and the book by Ye [463].

Analysis of the simplex algorithm remains an active area of research. V. Klee and G. J. Minty constructed an example on which the simplex algorithm runs through $2^n - 1$ iterations. The simplex algorithm usually performs well in practice, and many researchers have tried to give theoretical justification for this empirical observation. A line of research begun by K. H. Borgwardt, and carried on by many others, shows that under certain probabilistic assumptions on the input, the simplex algorithm converges in expected polynomial time. Spielman and Teng [421] made progress in this area, introducing the "smoothed analysis of algorithms" and applying it to the simplex algorithm.

The simplex algorithm is known to run efficiently in certain special cases. Particularly noteworthy is the network-simplex algorithm, which is the simplex algorithm, specialized to network-flow problems. For certain network problems, including the shortest-paths, maximum-flow, and minimum-cost-flow problems, variants of the network-simplex algorithm run in polynomial time. See, for example, the article by Orlin [349] and the citations therein.

30 Polynomials and the FFT

The straightforward method of adding two polynomials of degree n takes $\Theta(n)$ time, but the straightforward method of multiplying them takes $\Theta(n^2)$ time. This chapter will show how the fast Fourier transform, or FFT, can reduce the time to multiply polynomials to $\Theta(n \lg n)$.

The most common use for Fourier transforms, and hence the FFT, is in signal processing. A signal is given in the ***time domain***: as a function mapping time to amplitude. Fourier analysis expresses the signal as a weighted sum of phase-shifted sinusoids of varying frequencies. The weights and phases associated with the frequencies characterize the signal in the ***frequency domain***. Among the many everyday applications of FFT's are compression techniques used to encode digital video and audio information, including MP3 files. Many fine books delve into the rich area of signal processing, and the chapter notes reference a few of them.

Polynomials

A ***polynomial*** in the variable x over an algebraic field F represents a function $A(x)$ as a formal sum:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \ .$$

The values $a_0, a_1, \ldots, a_{n-1}$ are the ***coefficients*** of the polynomial. The coefficients and x are drawn from a field F, typically the set $\mathbb{C}$ of complex numbers. A polynomial $A(x)$ has ***degree*** k if its highest nonzero coefficient is a_k, in which case we say that $\text{degree}(A) = k$. Any integer strictly greater than the degree of a polynomial is a ***degree-bound*** of that polynomial. Therefore, the degree of a polynomial of degree-bound n may be any integer between 0 and $n - 1$, inclusive.

A variety of operations extend to polynomials. For ***polynomial addition***, if $A(x)$ and $B(x)$ are polynomials of degree-bound n, their ***sum*** is a polynomial $C(x)$, also

of degree-bound n, such that $C(x) = A(x) + B(x)$ for all x in the underlying field. That is, if

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \quad \text{and} \quad B(x) = \sum_{j=0}^{n-1} b_j x^j ,$$

then

$$C(x) = \sum_{j=0}^{n-1} c_j x^j ,$$

where $c_j = a_j + b_j$ for $j = 0, 1, \ldots, n-1$. For example, given the polynomials $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$, their sum is $C(x) = 4x^3 + 7x^2 - 6x + 4$.

For ***polynomial multiplication***, if $A(x)$ and $B(x)$ are polynomials of degree-bound n, their ***product*** $C(x)$ is a polynomial of degree-bound $2n - 1$ such that $C(x) = A(x)B(x)$ for all x in the underlying field. You probably have multiplied polynomials before, by multiplying each term in $A(x)$ by each term in $B(x)$ and then combining terms with equal powers. For example, you can multiply $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$ as follows:

$$\begin{array}{rrrrrrrl}
 & & & 6x^3 & +\ 7x^2 & -\ 10x & +\ 9 & \\
 & & & -\ 2x^3 & & +\ 4x & -\ 5 & \\
\hline
 & & & -\ 30x^3 & -\ 35x^2 & +\ 50x & -\ 45 & \text{(multiply } A(x) \text{ by } -5) \\
 & & 24x^4 & +\ 28x^3 & -\ 40x^2 & +\ 36x & & \text{(multiply } A(x) \text{ by } 4x) \\
-\ 12x^6 & -\ 14x^5 & +\ 20x^4 & -\ 18x^3 & & & & \text{(multiply } A(x) \text{ by } -2x^3) \\
\hline
-\ 12x^6 & -\ 14x^5 & +\ 44x^4 & -\ 20x^3 & -\ 75x^2 & +\ 86x & -\ 45 &
\end{array}$$

Another way to express the product $C(x)$ is

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j , \tag{30.1}$$

where

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} , \tag{30.2}$$

(By the definition of degree, $a_k = 0$ for all $k > \text{degree}(A)$ and $b_k = 0$ for all $k > \text{degree}(B)$.) If A is a polynomial of degree-bound n_a and B is a polynomial of degree-bound n_b, then C must be a polynomial of degree-bound $n_a + n_b - 1$, because $\text{degree}(C) = \text{degree}(A) + \text{degree}(B)$. Since a polynomial of degree-bound k is also a polynomial of degree-bound $k + 1$, we normally make the somewhat simpler statement that the product polynomial C is a polynomial of degree-bound $n_a + n_b$.

Chapter outline

Section 30.1 presents two ways to represent polynomials: the coefficient representation and the point-value representation. The straightforward method for multiplying polynomials of degree n—equations (30.1) and (30.2)—takes $\Theta(n^2)$ time with polynomials represented in coefficient form, but only $\Theta(n)$ time with point-value form. Converting between the two representations, however, reduces the time to multiply polynomials to just $\Theta(n \lg n)$. To see why this approach works, you must first understand complex roots of unity, which Section 30.2 covers. Section 30.2 then uses the FFT and its inverse to perform the conversions. Because the FFT is used so often in signal processing, it is often implemented as a circuit in hardware, and Section 30.3 illustrates the structure of such circuits.

This chapter relies on complex numbers, and within this chapter the symbol i denotes $\sqrt{-1}$ exclusively.

30.1 Representing polynomials

The coefficient and point-value representations of polynomials are in a sense equivalent: a polynomial in point-value form has a unique counterpart in coefficient form. This section introduces the two representations and shows how to combine them in order to multiply two degree-bound n polynomials in $\Theta(n \lg n)$ time.

Coefficient representation

A ***coefficient representation*** of a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ of degree-bound n is a vector of coefficients $a = (a_0, a_1, \ldots, a_{n-1})$. Matrix equations in this chapter generally treat vectors as column vectors.

The coefficient representation is convenient for certain operations on polynomials. For example, the operation of ***evaluating*** the polynomial $A(x)$ at a given point x_0 consists of computing the value of $A(x_0)$. To evaluate a polynomial in $\Theta(n)$ time, use ***Horner's rule***:

$$A(x_0) = a_0 + x_0 \left(a_1 + x_0 \left(a_2 + \cdots + x_0 \left(a_{n-2} + x_0 (a_{n-1}) \right) \cdots \right) \right) .$$

Similarly, adding two polynomials represented by the coefficient vectors $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$ takes $\Theta(n)$ time: just produce the coefficient vector $c = (c_0, c_1, \ldots, c_{n-1})$, where $c_j = a_j + b_j$ for $j = 0, 1, \ldots, n-1$.

Now, consider multiplying two degree-bound n polynomials $A(x)$ and $B(x)$ represented in coefficient form. The method described by equations (30.1) and (30.2) takes $\Theta(n^2)$ time, since it multiplies each coefficient in the vector a by each co-

efficient in the vector b. The operation of multiplying polynomials in coefficient form seems to be considerably more difficult than that of evaluating a polynomial or adding two polynomials. The resulting coefficient vector c, given by equation (30.2), is also called the ***convolution*** of the input vectors a and b, denoted $c = a \otimes b$. Since multiplying polynomials and computing convolutions are fundamental computational problems of considerable practical importance, this chapter concentrates on efficient algorithms for them.

Point-value representation

A ***point-value representation*** of a polynomial $A(x)$ of degree-bound n is a set of n ***point-value pairs***

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

such that all of the x_k are distinct and

$$y_k = A(x_k) \tag{30.3}$$

for $k = 0, 1, \ldots, n-1$. A polynomial has many different point-value representations, since any set of n distinct points $x_0, x_1, \ldots, x_{n-1}$ can serve as a basis for the representation.

Computing a point-value representation for a polynomial given in coefficient form is in principle straightforward, since all you have to do is select n distinct points $x_0, x_1, \ldots, x_{n-1}$ and then evaluate $A(x_k)$ for $k = 0, 1, \ldots, n-1$. With Horner's method, evaluating a polynomial at n points takes $\Theta(n^2)$ time. We'll see later that if you choose the points x_k cleverly, you can accelerate this computation to run in $\Theta(n \lg n)$ time.

The inverse of evaluation—determining the coefficient form of a polynomial from a point-value representation—is ***interpolation***. The following theorem shows that interpolation is well defined when the desired interpolating polynomial must have a degree-bound equal to the given number of point-value pairs.

Theorem 30.1 (Uniqueness of an interpolating polynomial)
For any set $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ of n point-value pairs such that all the x_k values are distinct, there is a unique polynomial $A(x)$ of degree-bound n such that $y_k = A(x_k)$ for $k = 0, 1, \ldots, n-1$.

Proof The proof relies on the existence of the inverse of a certain matrix. Equation (30.3) is equivalent to the matrix equation

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} . \tag{30.4}$$

The matrix on the left is denoted $V(x_0, x_1, \ldots, x_{n-1})$ and is known as a ***Vandermonde matrix***. By Problem D-1 on page 1223, this matrix has determinant

$$\prod_{0 \le j < k \le n-1} (x_k - x_j) ,$$

and therefore, by Theorem D.5 on page 1221, it is invertible (that is, nonsingular) if the x_k are distinct. To solve for the coefficients a_j uniquely given the point-value representation, use the inverse of the Vandermonde matrix:

$$a = V(x_0, x_1, \ldots, x_{n-1})^{-1} y . \qquad \blacksquare$$

The proof of Theorem 30.1 describes an algorithm for interpolation based on solving the set (30.4) of linear equations. Section 28.1 shows how to solve these equations in $O(n^3)$ time.

A faster algorithm for n-point interpolation is based on ***Lagrange's formula***:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \ne k} (x - x_j)}{\prod_{j \ne k} (x_k - x_j)} . \tag{30.5}$$

You might want to verify that the right-hand side of equation (30.5) is a polynomial of degree-bound n that satisfies $A(x_k) = y_k$ for all k. Exercise 30.1-5 asks you how to compute the coefficients of A using Lagrange's formula in $\Theta(n^2)$ time.

Thus, n-point evaluation and interpolation are well-defined inverse operations that transform between the coefficient representation of a polynomial and a point-value representation.[1] The algorithms described above for these problems take $\Theta(n^2)$ time.

The point-value representation is quite convenient for many operations on polynomials. For addition, if $C(x) = A(x) + B(x)$, then $C(x_k) = A(x_k) + B(x_k)$ for any point x_k. More precisely, given point-value representations for A,

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\} ,$$

[1] Interpolation is a notoriously tricky problem from the point of view of numerical stability. Although the approaches described here are mathematically correct, small differences in the inputs or round-off errors during computation can cause large differences in the result.

and for B,

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\} ,$$

where A and B are evaluated at the *same* n points, then a point-value representation for C is

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})\} .$$

Thus the time to add two polynomials of degree-bound n in point-value form is $\Theta(n)$.

Similarly, the point-value representation is convenient for multiplying polynomials. If $C(x) = A(x)B(x)$, then $C(x_k) = A(x_k)B(x_k)$ for any point x_k, and to obtain a point-value representation for C, just pointwise multiply a point-value representation for A by a point-value representation for B. Polynomial multiplication differs from polynomial addition in one key aspect, however: $\text{degree}(C) = \text{degree}(A) + \text{degree}(B)$, so that if A and B have degree-bound n, then C has degree-bound $2n$. A standard point-value representation for A and B consists of n point-value pairs for each polynomial. Multiplying these together gives n point-value pairs, but $2n$ pairs are necessary to interpolate a unique polynomial C of degree-bound $2n$. (See Exercise 30.1-4.) Instead, begin with "extended" point-value representations for A and for B consisting of $2n$ point-value pairs each. Given an extended point-value representation for A,

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\} ,$$

and a corresponding extended point-value representation for B,

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\} ,$$

then a point-value representation for C is

$$\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \ldots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\} .$$

Given two input polynomials in extended point-value form, multiplying them to obtain the point-value form of the result takes just $\Theta(n)$ time, much less than the $\Theta(n^2)$ time required to multiply polynomials in coefficient form.

Finally, let's consider how to evaluate a polynomial given in point-value form at a new point. For this problem, the simplest approach known is to first convert the polynomial to coefficient form and then evaluate it at the new point.

Fast multiplication of polynomials in coefficient form

Can the linear-time multiplication method for polynomials in point-value form expedite polynomial multiplication in coefficient form? The answer hinges on

Figure 30.1 A graphical outline of an efficient polynomial-multiplication process. Representations on the top are in coefficient form, and those on the bottom are in point-value form. The arrows from left to right correspond to the multiplication operation. The ω_{2n} terms are complex $(2n)$th roots of unity.

whether it is possible convert a polynomial quickly from coefficient form to point-value form (evaluate) and vice versa (interpolate).

Any points can serve as evaluation points, but certain evaluation points allow conversion between representations in only $\Theta(n \lg n)$ time. As we'll see in Section 30.2, if "complex roots of unity" are the evaluation points, then the discrete Fourier transform (or DFT) evaluates and the inverse DFT interpolates. Section 30.2 shows how the FFT accomplishes the DFT and inverse DFT operations in $\Theta(n \lg n)$ time.

Figure 30.1 shows this strategy graphically. One minor detail concerns degree-bounds. The product of two polynomials of degree-bound n is a polynomial of degree-bound $2n$. Before evaluating the input polynomials A and B, therefore, first double their degree-bounds to $2n$ by adding n high-order coefficients of 0. Because the vectors have $2n$ elements, use "complex $(2n)$th roots of unity," which are denoted by the ω_{2n} terms in Figure 30.1.

The following procedure takes advantage of the FFT to multiply two polynomials $A(x)$ and $B(x)$ of degree-bound n in $\Theta(n \lg n)$-time, where the input and output representations are in coefficient form. The procedure assumes that n is an exact power of 2, so if it isn't, just add high-order zero coefficients.

1. **Double degree-bound:** Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-order zero coefficients to each.

2. **Evaluate:** Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ by applying the FFT of order $2n$ on each polynomial. These representations contain the values of the two polynomials at the $(2n)$th roots of unity.
3. **Pointwise multiply:** Compute a point-value representation for the polynomial $C(x) = A(x)B(x)$ by multiplying these values together pointwise. This representation contains the value of $C(x)$ at each $(2n)$th root of unity.
4. **Interpolate:** Create the coefficient representation of the polynomial $C(x)$ by applying the FFT on $2n$ point-value pairs to compute the inverse DFT.

Steps (1) and (3) take $\Theta(n)$ time, and steps (2) and (4) take $\Theta(n \lg n)$ time. Thus, once we show how to use the FFT, we will have proven the following.

Theorem 30.2
Two polynomials of degree-bound n with both the input and output representations in coefficient form can be multiplied in $\Theta(n \lg n)$ time. ■

Exercises

30.1-1
Multiply the polynomials $A(x) = 7x^3 - x^2 + x - 10$ and $B(x) = 8x^3 - 6x + 3$ using equations (30.1) and (30.2).

30.1-2
Another way to evaluate a polynomial $A(x)$ of degree-bound n at a given point x_0 is to divide $A(x)$ by the polynomial $(x - x_0)$, obtaining a quotient polynomial $q(x)$ of degree-bound $n - 1$ and a remainder r, such that

$$A(x) = q(x)(x - x_0) + r \ .$$

Then we have $A(x_0) = r$. Show how to compute the remainder r and the coefficients of $q(x)$ from x_0 and the coefficients of A in $\Theta(n)$ time.

30.1-3
Given a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$, define $A^{\text{rev}}(x) = \sum_{j=0}^{n-1} a_{n-1-j} x^j$. Show how to derive a point-value representation for $A^{\text{rev}}(x)$ from a point-value representation for $A(x)$, assuming that none of the points is 0.

30.1-4
Prove that n distinct point-value pairs are necessary to uniquely specify a polynomial of degree-bound n, that is, if fewer than n distinct point-value pairs are given, they fail to specify a unique polynomial of degree-bound n. (*Hint:* Using Theorem 30.1, what can you say about a set of $n - 1$ point-value pairs to which you add one more arbitrarily chosen point-value pair?)

30.1-5
Show how to use equation (30.5) to interpolate in $\Theta(n^2)$ time. (*Hint:* First compute the coefficient representation of the polynomial $\prod_j (x - x_j)$ and then divide by $(x - x_k)$ as necessary for the numerator of each term (see Exercise 30.1-2). You can compute each of the n denominators in $O(n)$ time.)

30.1-6
Explain what is wrong with the "obvious" approach to polynomial division using a point-value representation: dividing the corresponding y values. Discuss separately the case in which the division comes out exactly and the case in which it doesn't.

30.1-7
Consider two sets A and B, each having n integers in the range from 0 to $10n$. The ***Cartesian sum*** of A and B is defined by

$$C = \{x + y : x \in A \text{ and } y \in B\} .$$

The integers in C lie in the range from 0 to $20n$. Show how, in $O(n \lg n)$ time, to find the elements of C and the number of times each element of C is realized as a sum of elements in A and B. (*Hint:* Represent A and B as polynomials of degree at most $10n$.)

30.2 The DFT and FFT

In Section 30.1, we claimed that by computing the DFT and its inverse by using the FFT, it is possible to evaluate and interpolate a degree n polynomial at the complex roots of unity in $\Theta(n \lg n)$ time. This section defines complex roots of unity, studies their properties, defines the DFT, and then shows how the FFT computes the DFT and its inverse in $\Theta(n \lg n)$ time.

Complex roots of unity

A ***complex nth root of unity*** is a complex number ω such that

$$\omega^n = 1 .$$

There are exactly n complex nth roots of unity: $e^{2\pi i k/n}$ for $k = 0, 1, \ldots, n-1$. To interpret this formula, use the definition of the exponential of a complex number:

$$e^{iu} = \cos(u) + i \sin(u) .$$

Figure 30.2 shows that the n complex roots of unity are equally spaced around the circle of unit radius centered at the origin of the complex plane. The value

Figure 30.2 The values of $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ in the complex plane, where $\omega_8 = e^{2\pi i/8}$ is the principal 8th root of unity.

$$\omega_n = e^{2\pi i/n} \tag{30.6}$$

is the ***principal nth root of unity***.[2] All other complex nth roots of unity are powers of ω_n.

The n complex nth roots of unity,

$$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1} ,$$

form a group under multiplication (see Section 31.3). This group has the same structure as the additive group $(\mathbb{Z}_n, +)$ modulo n, since $\omega_n^n = \omega_n^0 = 1$ implies that $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$. Similarly, $\omega_n^{-1} = \omega_n^{n-1}$. The following lemmas furnish some essential properties of the complex nth roots of unity.

Lemma 30.3 (Cancellation lemma)
For any integers $n > 0$, $k \geq 0$, and $d > 0$,

$$\omega_{dn}^{dk} = \omega_n^k . \tag{30.7}$$

Proof The lemma follows directly from equation (30.6), since

$$\begin{aligned} \omega_{dn}^{dk} &= \left(e^{2\pi i/dn}\right)^{dk} \\ &= \left(e^{2\pi i/n}\right)^{k} \\ &= \omega_n^k . \end{aligned}$$

■

[2] Many other authors define ω_n differently: $\omega_n = e^{-2\pi i/n}$. This alternative definition tends to be used for signal-processing applications. The underlying mathematics is substantially the same with either definition of ω_n.

Corollary 30.4
For any even integer $n > 0$,

$$\omega_n^{n/2} = \omega_2 = -1 .$$

Proof The proof is left as Exercise 30.2-1. ■

Lemma 30.5 (Halving lemma)
If $n > 0$ is even, then the squares of the n complex nth roots of unity are the $n/2$ complex $(n/2)$th roots of unity.

Proof By the cancellation lemma, $(\omega_n^k)^2 = \omega_{n/2}^k$ for any nonnegative integer k. Squaring all of the complex nth roots of unity produces each $(n/2)$th root of unity exactly twice, since

$$\begin{aligned}(\omega_n^{k+n/2})^2 &= \omega_n^{2k+n} \\ &= \omega_n^{2k}\omega_n^n \\ &= \omega_n^{2k} \\ &= (\omega_n^k)^2 .\end{aligned}$$

Thus ω_n^k and $\omega_n^{k+n/2}$ have the same square. We could also have used Corollary 30.4 to prove this property, since $\omega_n^{n/2} = -1$ implies $\omega_n^{k+n/2} = \omega_n^k\omega_n^{n/2} = -\omega_n^k$, and thus $(\omega_n^{k+n/2})^2 = (-\omega_n^k)^2 = (\omega_n^k)^2$. ■

As we'll see, the halving lemma is essential to the divide-and-conquer approach for converting between coefficient and point-value representations of polynomials, since it guarantees that the recursive subproblems are only half as large.

Lemma 30.6 (Summation lemma)
For any integer $n \geq 1$ and nonzero integer k not divisible by n,

$$\sum_{j=0}^{n-1} \left(\omega_n^k\right)^j = 0 .$$

Proof Equation (A.6) on page 1142 applies to complex values as well as to reals, giving

$$\begin{aligned}
\sum_{j=0}^{n-1} \left(\omega_n^k\right)^j &= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} \\
&= \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} \\
&= \frac{(1)^k - 1}{\omega_n^k - 1} \\
&= 0 .
\end{aligned}$$

To see that the denominator is not 0, note that $\omega_n^k = 1$ only when k is divisible by n, which the lemma statement prohibits. ■

The DFT

Recall the goal of evaluating a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

of degree-bound n at $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$ (that is, at the n complex nth roots of unity).[3] The polynomial A is given in coefficient form: $a = (a_0, a_1, \ldots, a_{n-1})$. Let us define the results y_k, for $k = 0, 1, \ldots, n-1$, by

$$\begin{aligned}
y_k &= A(\omega_n^k) \\
&= \sum_{j=0}^{n-1} a_j \omega_n^{kj} . \qquad (30.8)
\end{aligned}$$

The vector $y = (y_0, y_1, \ldots, y_{n-1})$ is the ***discrete Fourier transform (DFT)*** of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$. We also write $y = \mathrm{DFT}_n(a)$.

The FFT

The ***fast Fourier transform (FFT)*** takes advantage of the special properties of the complex roots of unity to compute $\mathrm{DFT}_n(a)$ in $\Theta(n \lg n)$ time, as opposed to the $\Theta(n^2)$ time of the straightforward method. Assume throughout that n is an exact power of 2. Although strategies for dealing with sizes that are not exact powers of 2 are known, they are beyond the scope of this book.

---

[3] The length n is actually what Section 30.1 referred to as $2n$, since the degree-bound of the given polynomials doubles prior to evaluation. In the context of polynomial multiplication, therefore, we are actually working with complex $(2n)$th roots of unity.

The FFT method employs a divide-and-conquer strategy, using the even-indexed and odd-indexed coefficients of $A(x)$ separately to define the two new polynomials $A^{\text{even}}(x)$ and $A^{\text{odd}}(x)$ of degree-bound $n/2$:

$$\begin{aligned} A^{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} , \\ A^{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} . \end{aligned}$$

Note that A^{even} contains all the even-indexed coefficients of A (the binary representation of the index ends in 0) and A^{odd} contains all the odd-indexed coefficients (the binary representation of the index ends in 1). It follows that

$$A(x) = A^{\text{even}}(x^2) + x A^{\text{odd}}(x^2) , \tag{30.9}$$

so that the problem of evaluating $A(x)$ at $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ reduces to

1. evaluating the degree-bound $n/2$ polynomials $A^{\text{even}}(x)$ and $A^{\text{odd}}(x)$ at the points

$$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2 , \tag{30.10}$$

   and then

2. combining the results according to equation (30.9).

By the halving lemma, the list of values (30.10) consists not of n distinct values but only of the $n/2$ complex $(n/2)$th roots of unity, with each root occurring exactly twice. Therefore, the FFT recursively evaluates the polynomials A^{even} and A^{odd} of degree-bound $n/2$ at the $n/2$ complex $(n/2)$th roots of unity. These subproblems have exactly the same form as the original problem, but are half the size, dividing an n-element DFT_n computation into two $n/2$-element $\text{DFT}_{n/2}$ computations. This decomposition is the basis for the FFT procedure on the next page, which computes the DFT of an n-element vector $a = (a_0, a_1, \ldots, a_{n-1})$, where n is an exact power of 2.

The FFT procedure works as follows. Lines 1–2 represent the base case of the recursion. The DFT of 1 element is the element itself, since in this case

$$\begin{aligned} y_0 &= a_0 \omega_1^0 \\ &= a_0 \cdot 1 \\ &= a_0 . \end{aligned}$$

Lines 5–6 define the coefficient vectors for the polynomials A^{even} and A^{odd}. Lines 3, 4, and 12 guarantee that ω is updated properly so that whenever lines 10–11 are executed, $\omega = \omega_n^k$. (Keeping a running value of ω from iteration to iteration saves

```
FFT(a, n)
1  if n == 1
2      return a                          // DFT of 1 element is the element itself
3  ω_n = e^(2πi/n)
4  ω = 1
5  a^even = (a_0, a_2, ..., a_{n-2})
6  a^odd = (a_1, a_3, ..., a_{n-1})
7  y^even = FFT(a^even, n/2)
8  y^odd = FFT(a^odd, n/2)
9  for k = 0 to n/2 - 1                  // at this point, ω = ω_n^k
10     y_k = y_k^even + ω y_k^odd
11     y_{k+(n/2)} = y_k^even - ω y_k^odd
12     ω = ω ω_n
13 return y
```

time over computing ω_n^k from scratch each time through the **for** loop.[4]) Lines 7–8 perform the recursive $\mathrm{DFT}_{n/2}$ computations, setting, for $k = 0, 1, \ldots, n/2 - 1$,

$$\begin{aligned} y_k^{\text{even}} &= A^{\text{even}}(\omega_{n/2}^k) , \\ y_k^{\text{odd}} &= A^{\text{odd}}(\omega_{n/2}^k) , \end{aligned}$$

or, since $\omega_{n/2}^k = \omega_n^{2k}$ by the cancellation lemma,

$$\begin{aligned} y_k^{\text{even}} &= A^{\text{even}}(\omega_n^{2k}) , \\ y_k^{\text{odd}} &= A^{\text{odd}}(\omega_n^{2k}) . \end{aligned}$$

Lines 10–11 combine the results of the recursive $\mathrm{DFT}_{n/2}$ calculations. For the first $n/2$ results $y_0, y_1, \ldots, y_{n/2-1}$, line 10 yields

$$\begin{aligned} y_k &= y_k^{\text{even}} + \omega_n^k y_k^{\text{odd}} \\ &= A^{\text{even}}(\omega_n^{2k}) + \omega_n^k A^{\text{odd}}(\omega_n^{2k}) \\ &= A(\omega_n^k) \qquad\qquad \text{(by equation (30.9))} . \end{aligned}$$

For $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$, letting $k = 0, 1, \ldots, n/2 - 1$, line 11 yields

---

[4] The downside of iteratively updating ω is that round-off errors can accumulate, especially for larger input sizes. Several techniques to limit the magnitude of FFT round-off errors have been proposed, but are beyond the scope of this book. If several FFTs are going to be run on inputs of the same size, then it might be worthwhile to directly precompute a table of all $n/2$ values of ω_n^k.

$$\begin{aligned}
y_{k+(n/2)} &= y_k^{\text{even}} - \omega_n^k y_k^{\text{odd}} \\
&= y_k^{\text{even}} + \omega_n^{k+(n/2)} y_k^{\text{odd}} && (\text{since } \omega_n^{k+(n/2)} = -\omega_n^k) \\
&= A^{\text{even}}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{\text{odd}}(\omega_n^{2k}) \\
&= A^{\text{even}}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{\text{odd}}(\omega_n^{2k+n}) && (\text{since } \omega_n^{2k+n} = \omega_n^{2k}) \\
&= A(\omega_n^{k+(n/2)}) && (\text{by equation (30.9)}) .
\end{aligned}$$

Thus the vector y returned by FFT is indeed the DFT of the input vector a.

Lines 10 and 11 multiply each value y_k^{odd} by ω_n^k, for $k = 0, 1, \ldots, n/2 - 1$. Line 10 adds this product to y_k^{even}, and line 11 subtracts it. Because each factor ω_n^k appears in both its positive and negative forms, we call the factors ω_n^k ***twiddle factors***.

To determine the running time of the procedure FFT, note that exclusive of the recursive calls, each invocation takes $\Theta(n)$ time, where n is the length of the input vector. The recurrence for the running time is therefore

$$\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n) \\
&= \Theta(n \lg n) ,
\end{aligned}$$

by case 2 of the master theorem (Theorem 4.1). Thus the FFT can evaluate a polynomial of degree-bound n at the complex nth roots of unity in $\Theta(n \lg n)$ time.

Interpolation at the complex roots of unity

The polynomial multiplication scheme entails converting from coefficient form to point-value form by evaluating the polynomial at the complex roots of unity, point-wise multiplying, and finally converting from point-value form back to coefficient form by interpolating. We've just seen how to evaluate, so now we'll see how to interpolate the complex roots of unity by a polynomial. To interpolate, we'll write the DFT as a matrix equation and then look at the form of the matrix inverse.

From equation (30.4), we can write the DFT as the matrix product $y = V_n a$, where V_n is a Vandermonde matrix containing the appropriate powers of ω_n:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} .$$

The (k, j) entry of V_n is ω_n^{kj}, for $j, k = 0, 1, \ldots, n - 1$. The exponents of the entries of V_n form a multiplication table for factors 0 to $n - 1$.

For the inverse operation, which we write as $a = \mathrm{DFT}_n^{-1}(y)$, multiply y by the matrix V_n^{-1}, the inverse of V_n.

Theorem 30.7
For $j, k = 0, 1, \ldots, n-1$, the (j, k) entry of V_n^{-1} is ω_n^{-jk}/n.

Proof We show that $V_n^{-1} V_n = I_n$, the $n \times n$ identity matrix. Consider the (k, k') entry of $V_n^{-1} V_n$:

$$\begin{aligned}
[V_n^{-1} V_n]_{kk'} &= \sum_{j=0}^{n-1} (\omega_n^{-jk}/n)(\omega_n^{jk'}) \\
&= \sum_{j=0}^{n-1} \omega_n^{j(k'-k)}/n \ .
\end{aligned}$$

This summation equals 1 if $k' = k$, and it is 0 otherwise by the summation lemma (Lemma 30.6). Note that in order for the summation lemma to apply, $k' - k$ must not be divisible by n. Indeed, it is not, since $-(n-1) \le k' - k \le n - 1$. ■

With the inverse matrix V_n^{-1} defined, $\mathrm{DFT}_n^{-1}(y)$ is given by

$$\begin{aligned}
a_j &= \sum_{k=0}^{n-1} y_k \frac{\omega^{-jk}}{n} \\
&= \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj} \qquad (30.11)
\end{aligned}$$

for $j = 0, 1, \ldots, n-1$. By comparing equations (30.8) and (30.11), you can see that if you modify the FFT algorithm to switch the roles of a and y, replace ω_n by ω_n^{-1}, and divide each element of the result by n, you get the inverse DFT (see Exercise 30.2-4). Thus, DFT_n^{-1} is computable in $\Theta(n \lg n)$ time as well.

Thus, the FFT and the inverse FFT provide a way to transform a polynomial of degree-bound n back and forth between its coefficient representation and a point-value representation in only $\Theta(n \lg n)$ time. In the context of polynomial multiplication, we have shown the following about the convolution $a \otimes b$ of vectors a and b:

Theorem 30.8 (Convolution theorem)
For any two vectors a and b of length n, where n is an exact power of 2,

$$a \otimes b = \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(a) \cdot \mathrm{DFT}_{2n}(b)) \ ,$$

where the vectors a and b are padded with 0s to length $2n$ and $\cdot$ denotes the componentwise product of two $2n$-element vectors. ■

Exercises

30.2-1
Prove Corollary 30.4.

30.2-2
Compute the DFT of the vector $(0, 1, 2, 3)$.

30.2-3
Do Exercise 30.1-1 by using the $\Theta(n \lg n)$-time scheme.

30.2-4
Write pseudocode to compute DFT_n^{-1} in $\Theta(n \lg n)$ time.

30.2-5
Describe the generalization of the FFT procedure to the case in which n is an exact power of 3. Give a recurrence for the running time, and solve the recurrence.

★ ***30.2-6***
Instead of performing an n-element FFT over the field of complex numbers (where n is an exact power of 2), let's use the ring $\mathbb{Z}_m$ of integers modulo m, where $m = 2^{tn/2} + 1$ and t is an arbitrary positive integer. We can use $\omega = 2^t$ instead of ω_n as a principal nth root of unity, modulo m. Prove that the DFT and the inverse DFT are well defined in this system.

30.2-7
Given a list of values $z_0, z_1, \ldots, z_{n-1}$ (possibly with repetitions), show how to find the coefficients of a polynomial $P(x)$ of degree-bound $n + 1$ that has zeros only at $z_0, z_1, \ldots, z_{n-1}$ (possibly with repetitions). Your procedure should run in $O(n \lg^2 n)$ time. (*Hint:* The polynomial $P(x)$ has a zero at z_j if and only if $P(x)$ is a multiple of $(x - z_j)$.)

★ ***30.2-8***
The ***chirp transform*** of a vector $a = (a_0, a_1, \ldots, a_{n-1})$ is the vector $y = (y_0, y_1, \ldots, y_{n-1})$, where $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$ and z is any complex number. The DFT is therefore a special case of the chirp transform, obtained by taking $z = \omega_n$. Show how to evaluate the chirp transform for any complex number z in $O(n \lg n)$ time. (*Hint:* Use the equation

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} \left(a_j z^{j^2/2}\right) \left(z^{-(k-j)^2/2}\right)$$

to view the chirp transform as a convolution.)

30.3 FFT circuits

Many of the FFT's applications in signal processing require the utmost speed, and so the FFT is often implemented as a circuit in hardware. The FFT's divide-and-conquer structure enables the circuit to have a parallel structure so that the ***depth*** of the circuit—the maximum number of computational elements between any output and any input that can reach it—is $\Theta(\lg n)$. Moreover, the structure of the FFT circuit has several interesting mathematical properties, which we won't go into here.

Butterfly operations

Notice that the **for** loop of lines 9–12 of the FFT procedure computes the value $\omega_n^k\, y_k^{\text{odd}}$ twice per iteration: once in line 10 and once in line 11. A good optimizing compiler produces code that evaluates this ***common subexpression*** just once, storing its value into a temporary variable, so that lines 10–11 are treated like the three lines

$$\begin{aligned} t &= \omega\, y_k^{\text{odd}} \\ y_k &= y_k^{\text{even}} + t \\ y_{k+(n/2)} &= y_k^{\text{even}} - t \end{aligned}$$

This operation, multiplying the twiddle factor $\omega = \omega_n^k$ by y_k^{odd}, storing the product into the temporary variable t, and adding and subtracting t from y_k^{even}, is known as a ***butterfly operation***. Figure 30.3 shows it as a circuit, and you can see how it vaguely resembles the shape of a butterfly. (Although less colorfully, it could have been called a "bowtie" operation.)

Figure 30.3 A circuit for a butterfly operation. **(a)** The two input values enter from the left, the twiddle factor ω_n^k is multiplied by y_k^{odd}, and the sum and difference are output on the right. **(b)** A simplified drawing of a butterfly operation, which we'll use when drawing the parallel FFT circuit.

Figure 30.4 The schema for the conquer and combine steps of an n-input, n-output FFT circuit, FFT_n, shown for $n = 8$. Inputs enter from the left, and outputs exit from the right. The input values first go through two $\text{FFT}_{n/2}$ circuits, and then $n/2$ butterfly circuits combine the results. Only the top and bottom wires entering a butterfly interact with it: wires that pass through the middle of a butterfly do not affect that butterfly, nor are their values changed by that butterfly.

Recursive circuit structure

The FFT procedure follows the divide-and-conquer strategy that we first saw in Section 2.3.1:

Divide the n-element input vector into its $n/2$ even-indexed and $n/2$ odd-indexed elements.

Conquer by recursively computing the DFTs of the two subproblems, each of size $n/2$.

Combine by performing $n/2$ butterfly operations. These butterfly operations work with twiddle factors $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n/2-1}$.

The circuit schema in Figure 30.4 follows the conquer and combine steps of this pattern for an FFT circuit with n inputs and n outputs, denoted by FFT_n. Each line is a wire that carries a value. Inputs enter from the left, one per wire, and outputs exit from the right. The conquer step runs the inputs through two $\text{FFT}_{n/2}$ circuits, which are also constructed recursively. The values produced by the two $\text{FFT}_{n/2}$ circuits feed into $n/2$ butterfly circuits, with twiddle factors $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n/2} - 1$,

Figure 30.5 The tree of input vectors to the recursive calls of the FFT procedure. The initial invocation is for $n = 8$.

to combine the results. The base case of the recursion occurs when $n = 1$, where the sole output value equals the sole input value. An FFT_1 circuit, therefore, does nothing, and so the smallest nontrivial FFT circuit is FFT_2, a single butterfly operation whose twiddle factor is $\omega_2^0 = 1$.

Permuting the inputs

How does the divide step enter into the circuit design? Let's examine how input vectors to the various recursive calls of the FFT procedure relate to the original input vector, so that the circuit can emulate the divide step at the start for all levels of recursion. Figure 30.5 arranges the input vectors to the recursive calls in an invocation of FFT in a tree structure, where the initial call is for $n = 8$. The tree has one node for each call of the procedure, labeled by the elements of the initial call as they appear in the corresponding input vector. Each FFT invocation makes two recursive calls, unless it has received a 1-element vector. The first call appears in the left child, and the second call appears in the right child.

Looking at the tree, observe that if you arrange the elements of the initial vector a into the order in which they appear in the leaves, you can trace the execution of the FFT procedure, but bottom up instead of top down. First, take the elements in pairs, compute the DFT of each pair using one butterfly operation, and replace the pair with its DFT. The vector then holds $n/2$ two-element DFTs. Next, take these $n/2$ DFTs in pairs and compute the DFT of the four vector elements they come from by executing two butterfly operations, replacing two two-element DFTs with one four-element DFT. The vector then holds $n/4$ four-element DFTs. Continue in this manner until the vector holds two $(n/2)$-element DFTs, which $n/2$ butterfly operations combine into the final n-element DFT. In other words, you can start with the elements of the initial vector a, but rearranged as in the leaves of Figure 30.5, and then feed them directly into a circuit that follows the schema in Figure 30.4.

Let's think about the permutation that rearranges the input vector. The order in which the leaves appear in Figure 30.5 is a ***bit-reversal permutation***. That is, letting $\text{rev}(k)$ be the $\lg n$-bit integer formed by reversing the bits of the binary representation of k, then vector element a_k moves to position $\text{rev}(k)$. In Figure 30.5, for example, the leaves appear in the order $0, 4, 2, 6, 1, 5, 3, 7$. This sequence in binary is $000, 100, 010, 110, 001, 101, 011, 111$, and you can obtain it by reversing the bits of each number in the sequence $0, 1, 2, 3, 4, 6, 7$ or, in binary, $000, 001, 010, 011, 100, 101, 110, 111$. To see in general that the input vector should be rearranged by a bit-reversal permutation, note that at the top level of the tree, indices whose low-order bit is 0 go into the left subtree and indices whose low-order bit is 1 go into the right subtree. Stripping off the low-order bit at each level, continue this process down the tree, until you get the order given by the bit-reversal permutation at the leaves.

The full FFT circuit

Figure 30.6 depicts the entire circuit for $n = 8$. The circuit begins with a bit-reversal permutation of the inputs, followed by $\lg n$ stages, each stage consisting of $n/2$ butterflies executed in parallel. Assuming that each butterfly circuit has constant depth, the full circuit has depth $\Theta(\lg n)$. The butterfly operations at each level of recursion in the FFT procedure are independent, and so the circuit performs them in parallel. The figure shows wires running from left to right, carrying values through the $\lg n$ stages. For $s = 1, 2, \ldots, \lg n$, stage s consists of $n/2^s$ groups of butterflies, with 2^{s-1} butterflies per group. The twiddle factors in stage s are $\omega_m^0, \omega_m^1, \ldots, \omega_m^{m/2-1}$, where $m = 2^s$.

Exercises

30.3-1
Show the values on the wires for each butterfly input and output in the FFT circuit of Figure 30.6, given the input vector $(0, 2, 3, -1, 4, 5, 7, 9)$.

30.3-2
Consider an FFT_n circuit, such as in Figure 30.6, with wires $0, 1, \ldots, n-1$ (wire j has output y_j) and stages numbered as in the figure. Stage s, for $s = 1, 2 \ldots, \lg n$, consists of $n/2^s$ groups of butterflies. Which two wires are inputs and outputs for the jth butterfly circuit in the gth group in stage s?

30.3-3
Consider a b-bit integer k in the range $0 \le k < 2^b$. Treating k as a b-element vector over $\{0, 1\}$, describe a $b \times b$ matrix M such that the matrix-vector product Mk is the binary representation of $\text{rev}(k)$.

Figure 30.6 A full circuit that computes the FFT in parallel, here shown for $n = 8$ inputs. It has $\lg n$ stages, and each stage comprises $n/2$ butterflies that can operate in parallel. As in Figure 30.4, only the top and bottom wires entering a butterfly interact with it. For example, the top butterfly in stage 2 has inputs and outputs only on wires 0 and 2 (the wires with outputs y_0 and y_2, respectively). This circuit has depth $\Theta(\lg n)$ and performs $\Theta(n \lg n)$ butterfly operations altogether.

30.3-4

Write pseudocode for the procedure BIT-REVERSE-PERMUTATION(a, n), which performs the bit-reversal permutation on a vector a of length n in-place. Assume that you may call the procedure BIT-REVERSE-OF(k, b), which returns an integer that is the b-bit reversal of the nonnegative integer k, where $0 \le k < 2^b$.

★ *30.3-5*

Suppose that the adders within the butterfly operations of a given FFT circuit sometimes fail in such a manner that they always produce a 0 output, independent of their inputs. In addition, suppose that exactly one adder has failed, but you don't know which one. Describe how you can identify the failed adder by supplying inputs to the overall FFT circuit and observing the outputs. How efficient is your method?

Problems

30-1 Divide-and-conquer multiplication

a. Show how to multiply two linear polynomials $ax + b$ and $cx + d$ using only three multiplications. (*Hint:* One of the multiplications is $(a + b) \cdot (c + d)$.)

b. Give two divide-and-conquer algorithms for multiplying two polynomials of degree-bound n in $\Theta(n^{\lg 3})$ time. The first algorithm should divide the input polynomial coefficients into a high half and a low half, and the second algorithm should divide them according to whether their index is odd or even.

c. Show how to multiply two n-bit integers in $O(n^{\lg 3})$ steps, where each step operates on at most a constant number of 1-bit values.

30-2 Multidimensional fast Fourier transform

The 1-dimensional discrete Fourier transform defined by equation (30.8) generalizes to d dimensions. The input is a d-dimensional array $A = (a_{j_1, j_2, \ldots, j_d})$ whose dimensions are $n_1, n_2, \ldots, n_d$, where $n_1 n_2 \cdots n_d = n$. The d-dimensional discrete Fourier transform is defined by the equation

$$y_{k_1, k_2, \ldots, k_d} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1, j_2, \ldots, j_d} \omega_{n_1}^{j_1 k_1} \omega_{n_2}^{j_2 k_2} \cdots \omega_{n_d}^{j_d k_d}$$

for $0 \le k_1 < n_1, 0 \le k_2 < n_2, \ldots, 0 \le k_d < n_d$.

a. Show how to produce a d-dimensional DFT by computing 1-dimensional DFTs on each dimension in turn. That is, first compute n/n_1 separate 1-dimensional DFTs along dimension 1. Then, using the result of the DFTs along dimension 1 as the input, compute n/n_2 separate 1-dimensional DFTs along dimension 2. Using this result as the input, compute n/n_3 separate 1-dimensional DFTs along dimension 3, and so on, through dimension d.

b. Show that the ordering of dimensions does not matter, so that if you compute the 1-dimensional DFTs in any order of the d dimensions, you compute the d-dimensional DFT.

c. Show that if you compute each 1-dimensional DFT by computing the fast Fourier transform, the total time to compute a d-dimensional DFT is $O(n \lg n)$, independent of d.

30-3 Evaluating all derivatives of a polynomial at a point

Given a polynomial $A(x)$ of degree-bound n, we define its tth derivative by

$$A^{(t)}(x) = \begin{cases} A(x) & \text{if } t = 0 , \\ \frac{d}{dx} A^{(t-1)}(x) & \text{if } 1 \le t \le n-1 , \\ 0 & \text{if } t \ge n . \end{cases}$$

In this problem, you will show how to determine $A^{(t)}(x_0)$ for $t = 0, 1, \ldots, n-1$, given the coefficient representation $(a_0, a_1, \ldots, a_{n-1})$ of $A(x)$ and a point x_0.

a. Given coefficients $b_0, b_1, \ldots, b_{n-1}$ such that

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j ,$$

show how to compute $A^{(t)}(x_0)$, for $t = 0, 1, \ldots, n-1$, in $O(n)$ time.

b. Explain how to find $b_0, b_1, \ldots, b_{n-1}$ in $O(n \lg n)$ time, given $A(x_0 + \omega_n^k)$ for $k = 0, 1, \ldots, n-1$.

c. Prove that

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \left(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j) \right) ,$$

where $f(j) = a_j \cdot j!$ and

$$g(l) = \begin{cases} x_0^{-l}/(-l)! & \text{if } -(n-1) \le l \le 0 , \\ 0 & \text{if } 1 \le l \le n-1 . \end{cases}$$

d. Explain how to evaluate $A(x_0 + \omega_n^k)$ for $k = 0, 1, \ldots, n-1$ in $O(n \lg n)$ time. Conclude that you can evaluate all nontrivial derivatives of $A(x)$ at x_0 in $O(n \lg n)$ time.

30-4 Polynomial evaluation at multiple points

Problem 2-3 showed how to evaluate a polynomial of degree-bound n at a single point in $O(n)$ time using Horner's rule. This chapter described how to evaluate such a polynomial at all n complex roots of unity in $O(n \lg n)$ time using the FFT. Now, you will show how to evaluate a polynomial of degree-bound n at n arbitrary points in $O(n \lg^2 n)$ time.

To do so, assume that you can compute the polynomial remainder when one such polynomial is divided by another in $O(n \lg n)$ time. For example, the remainder of $3x^3 + x^2 - 3x + 1$ when divided by $x^2 + x + 2$ is

$$(3x^3 + x^2 - 3x + 1) \bmod (x^2 + x + 2) = -7x + 5 .$$

Given the coefficient representation of a polynomial $A(x) = \sum_{k=0}^{n-1} a_k x^k$ and n points $x_0, x_1, \ldots, x_{n-1}$, your goal is to compute the n values $A(x_0), A(x_1), \ldots, A(x_{n-1})$. For $0 \le i \le j \le n-1$, define the polynomials $P_{ij}(x) = \prod_{k=i}^{j}(x - x_k)$ and $Q_{ij}(x) = A(x) \bmod P_{ij}(x)$. Note that $Q_{ij}(x)$ has degree at most $j - i$.

a. Prove that $A(x) \bmod (x - z) = A(z)$ for any point z.

b. Prove that $Q_{kk}(x) = A(x_k)$ and that $Q_{0,n-1}(x) = A(x)$.

c. Prove that for $i \le k \le j$, we have both $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$ and $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.

d. Give an $O(n \lg^2 n)$-time algorithm to evaluate $A(x_0), A(x_1), \ldots, A(x_{n-1})$.

30-5 FFT using modular arithmetic

As defined, the discrete Fourier transform requires computation with complex numbers, which can result in a loss of precision due to round-off errors. For some problems, the answer is known to contain only integers, and a variant of the FFT based on modular arithmetic can guarantee that the answer is calculated exactly. An example of such a problem is that of multiplying two polynomials with integer coefficients. Exercise 30.2-6 gives one approach, using a modulus of length $\Omega(n)$ bits to handle a DFT on n points. This problem explores another approach that uses a modulus of the more reasonable length $O(\lg n)$, but it requires that you understand the material of Chapter 31. Let n be an exact power of 2.

a. You wish to search for the smallest k such that $p = kn + 1$ is prime. Give a simple heuristic argument why you might expect k to be approximately $\ln n$. (The value of k might be much larger or smaller, but you can reasonably expect to examine $O(\lg n)$ candidate values of k on average.) How does the expected length of p compare to the length of n?

Let g be a generator of $\mathbb{Z}_p^*$, and let $w = g^k \bmod p$.

b. Argue that the DFT and the inverse DFT are well-defined inverse operations modulo p, where w is used as a principal nth root of unity.

c. Show how to make the FFT and its inverse work modulo p in $O(n \lg n)$ time, where operations on words of $O(\lg n)$ bits take unit time. Assume that the algorithm is given p and w.

d. Compute the DFT modulo $p = 17$ of the vector $(0, 5, 3, 7, 7, 2, 1, 6)$. (*Hint:* Verify and use the fact that $g = 3$ is a generator of $\mathbb{Z}_{17}^*$.)

Chapter notes

Van Loan's book [442] provides an outstanding treatment of the fast Fourier transform. Press, Teukolsky, Vetterling, and Flannery [365, 366] offer a good description of the fast Fourier transform and its applications. For an excellent introduction to signal processing, a popular FFT application area, see the texts by Oppenheim and Schafer [347] and Oppenheim and Willsky [348]. The Oppenheim and Schafer book also shows how to handle cases in which n is not an exact power of 2.

Fourier analysis is not limited to 1-dimensional data. It is widely used in image processing to analyze data in two or more dimensions. The books by Gonzalez and Woods [194] and Pratt [363] discuss multidimensional Fourier transforms and their use in image processing, and books by Tolimieri, An, and Lu [439] and Van Loan [442] discuss the mathematics of multidimensional fast Fourier transforms.

Cooley and Tukey [101] are widely credited with devising the FFT in the 1960s. The FFT had in fact been discovered many times previously, but its importance was not fully realized before the advent of modern digital computers. Although Press, Teukolsky, Vetterling, and Flannery attribute the origins of the method to Runge and König in 1924, an article by Heideman, Johnson, and Burrus [211] traces the history of the FFT as far back as C. F. Gauss in 1805.

Frigo and Johnson [161] developed a fast and flexible implementation of the FFT, called FFTW ("fastest Fourier transform in the West"). FFTW is designed for situations requiring multiple DFT computations on the same problem size. Before actually computing the DFTs, FFTW executes a "planner," which, by a series of trial runs, determines how best to decompose the FFT computation for the given problem size on the host machine. FFTW adapts to use the hardware cache efficiently, and once subproblems are small enough, FFTW solves them with optimized, straight-line code. Moreover, FFTW has the advantage of taking $\Theta(n \lg n)$ time for any problem size n, even when n is a large prime.

Although the standard Fourier transform assumes that the input represents points that are uniformly spaced in the time domain, other techniques can approximate the FFT on "nonequispaced" data. The article by Ware [449] provides an overview.

31 Number-Theoretic Algorithms

Number theory was once viewed as a beautiful but largely useless subject in pure mathematics. Today number-theoretic algorithms are used widely, due in large part to the invention of cryptographic schemes based on large prime numbers. These schemes are feasible because we can find large primes quickly, and they are secure because we do not know how to factor the product of large primes (or solve related problems, such as computing discrete logarithms) efficiently. This chapter presents some of the number theory and related algorithms that underlie such applications.

We start in Section 31.1 by introducing basic concepts of number theory, such as divisibility, modular equivalence, and unique prime factorization. Section 31.2 studies one of the world's oldest algorithms: Euclid's algorithm for computing the greatest common divisor of two integers, and Section 31.3 reviews concepts of modular arithmetic. Section 31.4 then explores the set of multiples of a given number a, modulo n, and shows how to find all solutions to the equation $ax = b \pmod{n}$ by using Euclid's algorithm. The Chinese remainder theorem is presented in Section 31.5. Section 31.6 considers powers of a given number a, modulo n, and presents a repeated-squaring algorithm for efficiently computing $a^b \bmod n$, given a, b, and n. This operation is at the heart of efficient primality testing and of much modern cryptography, such as the RSA public-key cryptosystem described in Section 31.7. We wrap up in Section 31.8, which examines a randomized primality test. This test finds large primes efficiently, an essential step in creating keys for the RSA cryptosystem.

Size of inputs and cost of arithmetic computations

Because we'll be working with large integers, we need to adjust how to think about the size of an input and about the cost of elementary arithmetic operations.

In this chapter, a "large input" typically means an input containing "large integers" rather than an input containing "many integers" (as for sorting). Thus, the size of an input depends on the *number of bits* required to represent that input, not just the number of integers in the input. An algorithm with integer in-

puts $a_1, a_2, \ldots, a_k$ is a ***polynomial-time algorithm*** if it runs in time polynomial in $\lg a_1, \lg a_2, \ldots, \lg a_k$, that is, polynomial in the lengths of its binary-encoded inputs.

Most of this book considers the elementary arithmetic operations (multiplications, divisions, or computing remainders) as primitive operations that take one unit of time. Counting the number of such arithmetic operations that an algorithm performs provides a basis for making a reasonable estimate of the algorithm's actual running time on a computer. Elementary operations can be time-consuming, however, when their inputs are large. It thus becomes appropriate to measure how many ***bit operations*** a number-theoretic algorithm requires. In this model, multiplying two β-bit integers by the ordinary method uses $\Theta(\beta^2)$ bit operations. Similarly, dividing a β-bit integer by a shorter integer or taking the remainder of a β-bit integer when divided by a shorter integer requires $\Theta(\beta^2)$ time by simple algorithms. (See Exercise 31.1-12.) Faster methods are known. For example, a simple divide-and-conquer method for multiplying two β-bit integers has a running time of $\Theta(\beta^{\lg 3})$, and $O(\beta \lg \beta \lg \lg \beta)$ time is possible. For practical purposes, however, the $\Theta(\beta^2)$ algorithm is often best, and we use this bound as a basis for our analyses. In this chapter, we'll usually analyze algorithms in terms of both the number of arithmetic operations and the number of bit operations they require.

31.1 Elementary number-theoretic notions

This section provides a brief review of notions from elementary number theory concerning the set $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ of integers and the set $\mathbb{N} = \{0, 1, 2, \ldots\}$ of natural numbers.

Divisibility and divisors

The notation $d \mid a$ (read "d ***divides*** a") means that $a = kd$ for some integer k. Every integer divides 0. If $a > 0$ and $d \mid a$, then $|d| \le |a|$. If $d \mid a$, then we also say that a is a ***multiple*** of d. If d does not divide a, we write $d \nmid a$.

If $d \mid a$ and $d \ge 0$, then d is a ***divisor*** of a. Since $d \mid a$ if and only if $-d \mid a$, without loss of generality, we define the divisors of a to be nonnegative, with the understanding that the negative of any divisor of a also divides a. A divisor of a nonzero integer a is at least 1 but not greater than $|a|$. For example, the divisors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.

Every positive integer a is divisible by the ***trivial divisors*** 1 and a. The nontrivial divisors of a are the ***factors*** of a. For example, the factors of 20 are 2, 4, 5, and 10.

Prime and composite numbers

An integer $a > 1$ whose only divisors are the trivial divisors 1 and a is a ***prime number*** or, more simply, a ***prime***. Primes have many special properties and play a critical role in number theory. The first 20 primes, in order, are

2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71.

Exercise 31.1-2 asks you to prove that there are infinitely many primes. An integer $a > 1$ that is not prime is a ***composite number*** or, more simply, a ***composite***. For example, 39 is composite because $3 \mid 39$. We call the integer 1 a ***unit***, and it is neither prime nor composite. Similarly, the integer 0 and all negative integers are neither prime nor composite.

The division theorem, remainders, and modular equivalence

Given an integer n, we can partition the integers into those that are multiples of n and those that are not multiples of n. Much number theory is based upon refining this partition by classifying the integers that are not multiples of n according to their remainders when divided by n. The following theorem provides the basis for this refinement. We omit the proof (but see, for example, Niven and Zuckerman [345]).

Theorem 31.1 (Division theorem)
For any integer a and any positive integer n, there exist unique integers q and r such that $0 \le r < n$ and $a = qn + r$. ■

The value $q = \lfloor a/n \rfloor$ is the ***quotient*** of the division. The value $r = a \bmod n$ is the ***remainder*** (or ***residue***) of the division, so that $n \mid a$ if and only if $a \bmod n = 0$.

The integers partition into n equivalence classes according to their remainders modulo n. The ***equivalence class modulo n*** containing an integer a is

$$[a]_n = \{a + kn : k \in \mathbb{Z}\} .$$

For example, $[3]_7 = \{\ldots, -11, -4, 3, 10, 17, \ldots\}$, and $[-4]_7$ and $[10]_7$ also denote this set. With the notation defined on page 64, writing $a \in [b]_n$ is the same as writing $a = b \pmod{n}$. The set of all such equivalence classes is

$$\mathbb{Z}_n = \{[a]_n : 0 \le a \le n - 1\} . \tag{31.1}$$

When you see the definition

$$\mathbb{Z}_n = \{0, 1, \ldots, n - 1\} , \tag{31.2}$$

you should read it as equivalent to equation (31.1) with the understanding that 0 represents $[0]_n$, 1 represents $[1]_n$, and so on. Each class is represented by its

smallest nonnegative element. You should keep the underlying equivalence classes in mind, however. For example, if we refer to -1 as a member of $\mathbb{Z}_n$, we are really referring to $[n-1]_n$, since $-1 = n - 1 \pmod{n}$.

Common divisors and greatest common divisors

If d is a divisor of a and d is also a divisor of b, then d is a ***common divisor*** of a and b. For example, the divisors of 30 are 1, 2, 3, 5, 6, 10, 15, and 30, and so the common divisors of 24 and 30 are 1, 2, 3, and 6. Any pair of integers has a common divisor of 1.

An important property of common divisors is that

$$\text{if } d \mid a \text{ and } d \mid b, \text{ then } d \mid (a+b) \text{ and } d \mid (a-b) . \tag{31.3}$$

More generally, for any integers x and y,

$$\text{if } d \mid a \text{ and } d \mid b, \text{ then } d \mid (ax+by) . \tag{31.4}$$

Also, if $a \mid b$, then either $|a| \le |b|$ or $b = 0$, which implies that

$$\text{if } a \mid b \text{ and } b \mid a, \text{ then } a = \pm b . \tag{31.5}$$

The ***greatest common divisor*** of two integers a and b which are not both 0, denoted by $\gcd(a,b)$, is the largest of the common divisors of a and b. For example, $\gcd(24,30) = 6$, $\gcd(5,7) = 1$, and $\gcd(0,9) = 9$. If a and b are both nonzero, then $\gcd(a,b)$ is an integer between 1 and $\min\{|a|,|b|\}$. We define $\gcd(0,0)$ to be 0, so that standard properties of the gcd function (such as equation (31.9) below) hold universally.

Exercise 31.1-9 asks you to prove the following elementary properties of the gcd function:

$$\gcd(a,b) = \gcd(b,a) , \tag{31.6}$$

$$\gcd(a,b) = \gcd(-a,b) , \tag{31.7}$$

$$\gcd(a,b) = \gcd(|a|,|b|) , \tag{31.8}$$

$$\gcd(a,0) = |a| , \tag{31.9}$$

$$\gcd(a,ka) = |a| \qquad \text{for any } k \in \mathbb{Z} . \tag{31.10}$$

The following theorem provides an alternative and useful way to characterize $\gcd(a,b)$.

Theorem 31.2

If a and b are any integers, not both zero, then $\gcd(a,b)$ is the smallest positive element of the set $\{ax+by : x, y \in \mathbb{Z}\}$ of linear combinations of a and b.

Proof Let s be the smallest positive such linear combination of a and b, and let $s = ax + by$ for some $x, y \in \mathbb{Z}$. Let $q = \lfloor a/s \rfloor$. Equation (3.11) on page 64 then implies

$$\begin{aligned} a \bmod s &= a - qs \\ &= a - q(ax + by) \\ &= a\,(1 - qx) + b\,(-qy), \end{aligned}$$

so that $a \bmod s$ is a linear combination of a and b as well. Because s is the smallest *positive* such linear combination and $0 \le a \bmod s < s$ (inequality (3.12) on page 64), $a \bmod s$ cannot be positive. Hence, $a \bmod s = 0$. Therefore, we have that $s \mid a$ and, by analogous reasoning, $s \mid b$. Thus, s is a common divisor of a and b, so that $\gcd(a, b) \ge s$. By definition, $\gcd(a, b)$ divides both a and b, and s is defined as a linear combination of a and b. Equation (31.4) therefore implies that $\gcd(a, b) \mid s$. But $\gcd(a, b) \mid s$ and $s > 0$ imply that $\gcd(a, b) \le s$. Combining $\gcd(a, b) \ge s$ and $\gcd(a, b) \le s$ yields $\gcd(a, b) = s$. We conclude that s, the smallest positive linear combination of a and b, is also their greatest common divisor. ■

Theorem 31.2 engenders three useful corollaries.

Corollary 31.3
For any integers a and b, if $d \mid a$ and $d \mid b$, then $d \mid \gcd(a, b)$.

Proof This corollary follows from equation (31.4) and Theorem 31.2, because $\gcd(a, b)$ is a linear combination of a and b, ■

Corollary 31.4
For all integers a and b and any nonnegative integer n, we have

$$\gcd(an, bn) = n \gcd(a, b) .$$

Proof If $n = 0$, the corollary is trivial. If $n > 0$, then $\gcd(an, bn)$ is the smallest positive element of the set $\{anx + bny : x, y \in \mathbb{Z}\}$, which in turn is n times the smallest positive element of the set $\{ax + by : x, y \in \mathbb{Z}\}$. ■

Corollary 31.5
For all positive integers n, a, and b, if $n \mid ab$ and $\gcd(a, n) = 1$, then $n \mid b$.

Proof Exercise 31.1-5 asks you to provide the proof. ■

Relatively prime integers

Two integers a and b are ***relatively prime*** if their only common divisor is 1, that is, if $\gcd(a, b) = 1$. For example, 8 and 15 are relatively prime, since the divisors of 8 are 1, 2, 4, and 8, and the divisors of 15 are 1, 3, 5, and 15. The following theorem states that if two integers are each relatively prime to an integer p, then their product is relatively prime to p.

Theorem 31.6
For any integers a, b, and p, we have $\gcd(ab, p) = 1$ if and only if $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$ both hold.

Proof If $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$, then it follows from Theorem 31.2 that there exist integers x, y, x', and y' such that

$$\begin{aligned} ax + py &= 1 , \\ bx' + py' &= 1 . \end{aligned}$$

Multiplying these equations and rearranging gives

$$ab(xx') + p(ybx' + y'ax + pyy') = 1 .$$

Since 1 is thus a positive linear combination of ab and p, it is the smallest positive linear combination. Applying Theorem 31.2 implies $\gcd(ab, p) = 1$, completing the proof in this direction.

Conversely, if $\gcd(ab, p) = 1$, then Theorem 31.2 implies that there exist integers x and y such that

$$abx + py = 1 .$$

Writing abx as $a(bx)$ and applying Theorem 31.2 again proves that $\gcd(a, p) = 1$. Proving that $\gcd(b, p) = 1$ is similar. ■

Integers $n_1, n_2, \ldots, n_k$ are ***pairwise relatively prime*** if $\gcd(n_i, n_j) = 1$ for $1 \le i < j \le k$.

Unique prime factorization

An elementary but important fact about divisibility by primes is the following.

Theorem 31.7
For all primes p and all integers a and b, if $p \mid ab$, then $p \mid a$ or $p \mid b$ (or both).

Proof Assume for the purpose of contradiction that $p \mid ab$, but that $p \nmid a$ and $p \nmid b$. Because $p > 1$ and $ab = kp$ for some $k \in \mathbb{Z}$, equation (31.10) gives

that $\gcd(ab, p) = p$. We also have that $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$, since the only divisors of p are 1 and p, and we assumed that p divides neither a nor b. Theorem 31.6 then implies that $\gcd(ab, p) = 1$, contradicting $\gcd(ab, p) = p$. This contradiction completes the proof. ■

A consequence of Theorem 31.7 is that any composite integer can be uniquely factored into a product of primes. Exercise 31.1-11 asks you to provide a proof.

Theorem 31.8 (Unique prime factorization)
There is exactly one way to write any composite integer a as a product of the form

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} ,$$

where the p_i are prime, $p_1 < p_2 < \cdots < p_r$, and the e_i are positive integers. ■

As an example, the unique prime factorization of the number 6000 is $2^4 \cdot 3^1 \cdot 5^3$.

Exercises

31.1-1
Prove that if $a > b > 0$ and $c = a + b$, then $c \bmod a = b$.

31.1-2
Prove that there are infinitely many primes. (*Hint:* Show that none of the primes $p_1, p_2, \ldots, p_k$ divide $(p_1 p_2 \cdots p_k) + 1$.)

31.1-3
Prove that if $a \mid b$ and $b \mid c$, then $a \mid c$.

31.1-4
Prove that if p is prime and $0 < k < p$, then $\gcd(k, p) = 1$.

31.1-5
Prove Corollary 31.5.

31.1-6
Prove that if p is prime and $0 < k < p$, then $p \mid \binom{p}{k}$. Conclude that for all integers a and b and all primes p,

$$(a + b)^p = a^p + b^p \pmod{p} .$$

31.1-7
Prove that if a and b are any positive integers such that $a \mid b$, then

$$(x \bmod b) \bmod a = x \bmod a$$

for any x. Prove, under the same assumptions, that

$$x = y \pmod b \quad \text{implies} \quad x = y \pmod a$$

for any integers x and y.

31.1-8
For any integer $k > 0$, an integer n is a ***kth power*** if there exists an integer a such that $a^k = n$. Furthermore, $n > 1$ is a ***nontrivial power*** if it is a kth power for some integer $k > 1$. Show how to determine whether a given β-bit integer n is a nontrivial power in time polynomial in β.

31.1-9
Prove equations (31.6)–(31.10).

31.1-10
Show that the gcd operator is associative. That is, prove that for all integers a, b, and c, we have

$$\gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c) .$$

★ ***31.1-11***
Prove Theorem 31.8.

31.1-12
Give efficient algorithms for the operations of dividing a β-bit integer by a shorter integer and of taking the remainder of a β-bit integer when divided by a shorter integer. Your algorithms should run in $\Theta(\beta^2)$ time.

31.1-13
Give an efficient algorithm to convert a given β-bit (binary) integer to a decimal representation. Argue that if multiplication or division of integers whose length is at most β takes $M(\beta)$ time, where $M(\beta) = \Omega(\beta)$, then you can convert binary to decimal in $O(M(\beta) \lg \beta)$ time. (*Hint:* Use a divide-and-conquer approach, obtaining the top and bottom halves of the result with separate recursions.)

31.1-14
Professor Marshall sets up n lightbulbs in a row. The lightbulbs all have switches, so that if he presses a bulb, it toggles on if it was off and off if it was on. The lightbulbs all start off. For $i = 1, 2, 3, \ldots, n$, the professor presses bulb $i, 2i, 3i, \ldots$. After the last press, which lightbulbs are on? Prove your answer.

31.2 Greatest common divisor

In this section, we describe Euclid's algorithm for efficiently computing the greatest common divisor of two integers. When we analyze the running time, we'll see a surprising connection with the Fibonacci numbers, which yield a worst-case input for Euclid's algorithm.

We restrict ourselves in this section to nonnegative integers. This restriction is justified by equation (31.8), which states that $\gcd(a, b) = \gcd(|a|, |b|)$.

In principle, for positive integers a and b, their prime factorizations suffice to compute $\gcd(a, b)$. Indeed, if

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} , \tag{31.11}$$

$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r} , \tag{31.12}$$

with 0 exponents being used to make the set of primes $p_1, p_2, \ldots, p_r$ the same for both a and b, then, as Exercise 31.2-1 asks you to show,

$$\gcd(a, b) = p_1^{\min\{e_1, f_1\}} p_2^{\min\{e_2, f_2\}} \cdots p_r^{\min\{e_r, f_r\}} . \tag{31.13}$$

The best algorithms to date for factoring do not run in polynomial time. Thus, this approach to computing greatest common divisors seems unlikely to yield an efficient algorithm.

Euclid's algorithm for computing greatest common divisors relies on the following theorem.

Theorem 31.9 (GCD recursion theorem)
For any nonnegative integer a and any positive integer b,

$$\gcd(a, b) = \gcd(b, a \bmod b) .$$

Proof We will show that $\gcd(a, b)$ and $\gcd(b, a \bmod b)$ divide each other. Since they are both nonnegative, equation (31.5) then implies that they must be equal.

We first show that $\gcd(a, b) \mid \gcd(b, a \bmod b)$. If we let $d = \gcd(a, b)$, then $d \mid a$ and $d \mid b$. By equation (3.11) on page 64, $a \bmod b = a - qb$, where $q = \lfloor a/b \rfloor$. Since $a \bmod b$ is thus a linear combination of a and b, equation (31.4) implies that $d \mid (a \bmod b)$. Therefore, since $d \mid b$ and $d \mid (a \bmod b)$, Corollary 31.3 implies that $d \mid \gcd(b, a \bmod b)$, that is,

$$\gcd(a, b) \mid \gcd(b, a \bmod b) . \tag{31.14}$$

Showing that $\gcd(b, a \bmod b) \mid \gcd(a, b)$ is almost the same. If we now let $d = \gcd(b, a \bmod b)$, then $d \mid b$ and $d \mid (a \bmod b)$. Since $a = qb + (a \bmod b)$,

where $q = \lfloor a/b \rfloor$, we have that a is a linear combination of b and $(a \bmod b)$. By equation (31.4), we conclude that $d \mid a$. Since $d \mid b$ and $d \mid a$, we have that $d \mid \gcd(a, b)$ by Corollary 31.3, so that

$$\gcd(b, a \bmod b) \mid \gcd(a, b) . \tag{31.15}$$

Using equation (31.5) to combine equations (31.14) and (31.15) completes the proof. ■

Euclid's algorithm

Euclid's *Elements* (circa 300 B.C.E.) describes the following gcd algorithm, although its origin might be even earlier. The recursive procedure EUCLID implements Euclid's algorithm, based directly on Theorem 31.9. The inputs a and b are arbitrary nonnegative integers.

```
EUCLID(a, b)
1  if b == 0
2      return a
3  else return EUCLID(b, a mod b)
```

For example, here is how the procedure computes $\gcd(30, 21)$:

$$\begin{aligned}
\text{EUCLID}(30, 21) &= \text{EUCLID}(21, 9) \\
&= \text{EUCLID}(9, 3) \\
&= \text{EUCLID}(3, 0) \\
&= 3 .
\end{aligned}$$

This computation calls EUCLID recursively three times.

The correctness of EUCLID follows from Theorem 31.9 and the property that if the algorithm returns a in line 2, then $b = 0$, so that by equation (31.9), $\gcd(a, b) = \gcd(a, 0) = a$. The algorithm cannot recurse indefinitely, since the second argument strictly decreases in each recursive call and is always nonnegative. Therefore, EUCLID always terminates with the correct answer.

The running time of Euclid's algorithm

Let's analyze the worst-case running time of EUCLID as a function of the size of a and b. The overall running time of EUCLID is proportional to the number of recursive calls it makes. The analysis assumes that $a > b \geq 0$, that is, the first argument is greater than the second argument. Why? If $b = a > 0$, then $a \bmod b = 0$ and the procedure terminates after one recursive call. If $b > a \geq 0$,

then the procedure makes just one more recursive call than when $a > b$, because in this case EUCLID(a, b) immediately makes the recursive call EUCLID(b, a), and now the first argument is greater than the second.

Our analysis relies on the Fibonacci numbers F_k, defined by the recurrence equation (3.31) on page 69.

Lemma 31.10
If $a > b \geq 1$ and the call EUCLID(a, b) performs $k \geq 1$ recursive calls, then $a \geq F_{k+2}$ and $b \geq F_{k+1}$.

Proof The proof proceeds by induction on k. For the base case of the induction, let $k = 1$. Then, $b \geq 1 = F_2$, and since $a > b$, we must have $a \geq 2 = F_3$. Since $b > (a \bmod b)$, in each recursive call the first argument is strictly larger than the second. The assumption that $a > b$ therefore holds for each recursive call.

Assuming inductively that the lemma holds if the procedure makes $k - 1$ recursive calls, we shall prove that the lemma holds for k recursive calls. Since $k > 0$, we have $b > 0$, and EUCLID(a, b) calls EUCLID$(b, a \bmod b)$ recursively, which in turn makes $k - 1$ recursive calls. The inductive hypothesis then implies that $b \geq F_{k+1}$ (thus proving part of the lemma), and $a \bmod b \geq F_k$. We have

$$\begin{aligned} b + (a \bmod b) &= b + (a - b\lfloor a/b \rfloor) \qquad \text{(by equation (3.11))} \\ &\leq a , \end{aligned}$$

since $a > b > 0$ implies $\lfloor a/b \rfloor \geq 1$. Thus,

$$\begin{aligned} a &\geq b + (a \bmod b) \\ &\geq F_{k+1} + F_k \\ &= F_{k+2} . \end{aligned}$$

■

The following theorem is an immediate corollary of this lemma.

Theorem 31.11 (Lamé's theorem)
For any integer $k \geq 1$, if $a > b \geq 1$ and $b < F_{k+1}$, then the call EUCLID(a, b) makes fewer than k recursive calls. ■

To show that the upper bound of Theorem 31.11 is the best possible, we'll show that the call EUCLID(F_{k+1}, F_k) makes exactly $k - 1$ recursive calls when $k \geq 2$. We use induction on k. For the base case, $k = 2$, and the call EUCLID(F_3, F_2) makes exactly one recursive call, to EUCLID$(1, 0)$. (We have to start at $k = 2$, because when $k = 1$ we do not have $F_2 > F_1$.) For the inductive step, assume that EUCLID(F_k, F_{k-1}) makes exactly $k - 2$ recursive calls. For $k > 2$, we have $F_k > F_{k-1} > 0$ and $F_{k+1} = F_k + F_{k-1}$, and so by Exercise 31.1-1, we

have $F_{k+1} \bmod F_k = F_{k-1}$. Because EUCLID$(a, b)$ calls EUCLID$(b, a \bmod b)$ when $b > 0$, the call EUCLID(F_{k+1}, F_k) recurses one time more than the call EUCLID(F_k, F_{k-1}), or exactly $k - 1$ times, which meets the upper bound given by Theorem 31.11.

Since F_k is approximately $\phi^k/\sqrt{5}$, where ϕ is the golden ratio $(1 + \sqrt{5})/2$ defined by equation (3.32) on page 69, the number of recursive calls in EUCLID is $O(\lg b)$. (See Exercise 31.2-5 for a tighter bound.) Therefore, a call of EUCLID on two β-bit numbers performs $O(\beta)$ arithmetic operations and $O(\beta^3)$ bit operations (assuming that multiplication and division of β-bit numbers take $O(\beta^2)$ bit operations). Problem 31-2 asks you to prove an $O(\beta^2)$ bound on the number of bit operations.

The extended form of Euclid's algorithm

By rewriting Euclid's algorithm, we can gain additional useful information. Specifically, let's extend the algorithm to compute the integer coefficients x and y such that

$$d = \gcd(a, b) = ax + by \ , \tag{31.16}$$

where either or both of x and y may be zero or negative. These coefficients will prove useful later for computing modular multiplicative inverses. The procedure EXTENDED-EUCLID takes as input a pair of nonnegative integers and returns a triple of the form (d, x, y) that satisfies equation (31.16). As an example, Figure 31.1 traces out the call EXTENDED-EUCLID$(99, 78)$.

```
EXTENDED-EUCLID(a, b)
1  if b == 0
2      return (a, 1, 0)
3  else (d', x', y') = EXTENDED-EUCLID(b, a mod b)
4      (d, x, y) = (d', y', x' − ⌊a/b⌋ y')
5      return (d, x, y)
```

The EXTENDED-EUCLID procedure is a variation of the EUCLID procedure. Line 1 is equivalent to the test "b == 0" in line 1 of EUCLID. If $b = 0$, then EXTENDED-EUCLID returns not only $d = a$ in line 2, but also the coefficients $x = 1$ and $y = 0$, so that $a = ax + by$. If $b \neq 0$, EXTENDED-EUCLID first computes (d', x', y') such that $d' = \gcd(b, a \bmod b)$ and

$$d' = bx' + (a \bmod b)y' \ . \tag{31.17}$$

As in the EUCLID procedure, we have $d = \gcd(a, b) = d' = \gcd(b, a \bmod b)$. To obtain x and y such that $d = ax + by$, let's rewrite equation (31.17), setting

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	−11	14
78	21	3	3	3	−11
21	15	1	3	−2	3
15	6	2	3	1	−2
6	3	2	3	0	1
3	0	—	3	1	0

Figure 31.1 How EXTENDED-EUCLID computes $\gcd(99, 78)$. Each line shows one level of the recursion: the values of the inputs a and b, the computed value $\lfloor a/b \rfloor$, and the values d, x, and y returned. The triple (d, x, y) returned becomes the triple (d', x', y') used at the next higher level of recursion. The call EXTENDED-EUCLID$(99, 78)$ returns $(3, -11, 14)$, so that $\gcd(99, 78) = 3 = 99 \cdot (-11) + 78 \cdot 14$.

$d = d'$ and using equation (3.11):

$$\begin{aligned} d &= bx' + (a - b\lfloor a/b \rfloor)y' \\ &= ay' + b(x' - \lfloor a/b \rfloor y') . \end{aligned}$$

Thus, choosing $x = y'$ and $y = x' - \lfloor a/b \rfloor y'$ satisfies the equation $d = ax + by$, thereby proving the correctness of EXTENDED-EUCLID.

Since the number of recursive calls made in EUCLID is equal to the number of recursive calls made in EXTENDED-EUCLID, the running times of EUCLID and EXTENDED-EUCLID are the same, to within a constant factor. That is, for $a > b > 0$, the number of recursive calls is $O(\lg b)$.

Exercises

31.2-1
Prove that equations (31.11) and (31.12) imply equation (31.13).

31.2-2
Compute the values (d, x, y) that the call EXTENDED-EUCLID$(899, 493)$ returns.

31.2-3
Prove that for all integers a, k, and n,

$$\gcd(a, n) = \gcd(a + kn, n) . \tag{31.18}$$

Use equation (31.18) to show that $a = 1 \pmod{n}$ implies $\gcd(a, n) = 1$.

31.2-4
Rewrite EUCLID in an iterative form that uses only a constant amount of memory (that is, stores only a constant number of integer values).

31.2-5
If $a > b \geq 0$, show that the call EUCLID(a, b) makes at most $1 + \log_\phi b$ recursive calls. Improve this bound to $1 + \log_\phi(b/\gcd(a, b))$.

31.2-6
What does EXTENDED-EUCLID(F_{k+1}, F_k) return? Prove your answer correct.

31.2-7
Define the gcd function for more than two arguments by the recursive equation $\gcd(a_0, a_1, \ldots, a_n) = \gcd(a_0, \gcd(a_1, a_2, \ldots, a_n))$. Show that the gcd function returns the same answer independent of the order in which its arguments are specified. Also show how to find integers $x_0, x_1, \ldots, x_n$ such that $\gcd(a_0, a_1, \ldots, a_n) = a_0x_0 + a_1x_1 + \cdots + a_nx_n$. Show that the number of divisions performed by your algorithm is $O(n + \lg(\max\{a_0, a_1, \ldots, a_n\}))$.

31.2-8
The ***least common multiple*** $\operatorname{lcm}(a_1, a_2, \ldots, a_n)$ of integers $a_1, a_2, \ldots, a_n$ is the smallest nonnegative integer that is a multiple of each a_i. Show how to compute $\operatorname{lcm}(a_1, a_2, \ldots, a_n)$ efficiently using the (two-argument) gcd operation as a subroutine.

31.2-9
Prove that n_1, n_2, n_3, and n_4 are pairwise relatively prime if and only if

$$\gcd(n_1n_2, n_3n_4) = \gcd(n_1n_3, n_2n_4) = 1 .$$

More generally, show that $n_1, n_2, \ldots, n_k$ are pairwise relatively prime if and only if a set of $\lceil \lg k \rceil$ pairs of numbers derived from the n_i are relatively prime.

31.3 Modular arithmetic

Informally, you can think of modular arithmetic as arithmetic as usual over the integers, except that when working modulo n, then every result x is replaced by the element of $\{0, 1, \ldots, n-1\}$ that is equivalent to x, modulo n (so that x is replaced by $x \bmod n$). This informal model suffices if you stick to the operations of addition, subtraction, and multiplication. A more formal model for modular arithmetic, which follows, is best described within the framework of group theory.

Finite groups

A ***group*** $(S, \oplus)$ is a set S together with a binary operation $\oplus$ defined on S for which the following properties hold:

1. **Closure:** For all $a, b \in S$, we have $a \oplus b \in S$.
2. **Identity:** There exists an element $e \in S$, called the ***identity*** of the group, such that $e \oplus a = a \oplus e = a$ for all $a \in S$.
3. **Associativity:** For all $a, b, c \in S$, we have $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.
4. **Inverses:** For each $a \in S$, there exists a unique element $b \in S$, called the ***inverse*** of a, such that $a \oplus b = b \oplus a = e$.

As an example, consider the familiar group $(\mathbb{Z}, +)$ of the integers $\mathbb{Z}$ under the operation of addition: 0 is the identity, and the inverse of a is $-a$. An ***abelian group*** $(S, \oplus)$ satisfies the ***commutative law*** $a \oplus b = b \oplus a$ for all $a, b \in S$. The ***size*** of group $(S, \oplus)$ is $|S|$, and if $|S| < \infty$, then $(S, \oplus)$ is a ***finite group***.

The groups defined by modular addition and multiplication

We can form two finite abelian groups by using addition and multiplication modulo n, where n is a positive integer. These groups are based on the equivalence classes of the integers modulo n, defined in Section 31.1.

To define a group on $\mathbb{Z}_n$, we need suitable binary operations, which we obtain by redefining the ordinary operations of addition and multiplication. We can define addition and multiplication operations for $\mathbb{Z}_n$, because the equivalence class of two integers uniquely determines the equivalence class of their sum or product. That is, if $a = a' \pmod{n}$ and $b = b' \pmod{n}$, then

$$\begin{aligned} a + b &= a' + b' \pmod{n} , \\ ab &= a'b' \pmod{n} . \end{aligned}$$

Thus, we define addition and multiplication modulo n, denoted $+_n$ and $\cdot_n$, by

$$\begin{aligned} [a]_n +_n [b]_n &= [a + b]_n , \\ [a]_n \cdot_n [b]_n &= [ab]_n . \end{aligned} \tag{31.19}$$

(We can define subtraction similarly on $\mathbb{Z}_n$ by $[a]_n -_n [b]_n = [a - b]_n$, but division is more complicated, as we'll see.) These facts justify the common and convenient practice of using the smallest nonnegative element of each equivalence class as its representative when performing computations in $\mathbb{Z}_n$. We add, subtract, and multiply as usual on the representatives, but we replace each result x by the representative of its class, that is, by $x \bmod n$.

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

(a)

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(b)

Figure 31.2 Two finite groups. Equivalence classes are denoted by their representative elements. **(a)** The group $(\mathbb{Z}_6, +_6)$. **(b)** The group $(\mathbb{Z}_{15}^*, \cdot_{15})$.

Using this definition of addition modulo n, we define the ***additive group modulo n*** as $(\mathbb{Z}_n, +_n)$. The size of the additive group modulo n is $|\mathbb{Z}_n| = n$. Figure 31.2(a) gives the operation table for the group $(\mathbb{Z}_6, +_6)$.

Theorem 31.12
The system $(\mathbb{Z}_n, +_n)$ is a finite abelian group.

Proof Equation (31.19) shows that $(\mathbb{Z}_n, +_n)$ is closed. Associativity and commutativity of $+_n$ follow from the associativity and commutativity of $+$:

$$\begin{aligned}
([a]_n +_n [b]_n) +_n [c]_n &= [a+b]_n +_n [c]_n \\
&= [(a+b)+c]_n \\
&= [a+(b+c)]_n \\
&= [a]_n +_n [b+c]_n \\
&= [a]_n +_n ([b]_n +_n [c]_n) ,
\end{aligned}$$

$$\begin{aligned}
[a]_n +_n [b]_n &= [a+b]_n \\
&= [b+a]_n \\
&= [b]_n +_n [a]_n .
\end{aligned}$$

The identity element of $(\mathbb{Z}_n, +_n)$ is 0 (that is, $[0]_n$). The (additive) inverse of an element a (that is, of $[a]_n$) is the element $-a$ (that is, $[-a]_n$ or $[n-a]_n$), since $[a]_n +_n [-a]_n = [a-a]_n = [0]_n$. ■

Using the definition of multiplication modulo n, we define the ***multiplicative group modulo n*** as $(\mathbb{Z}_n^*, \cdot_n)$. The elements of this group are the set $\mathbb{Z}_n^*$ of elements in $\mathbb{Z}_n$ that are relatively prime to n, so that each one has a unique inverse, modulo n:

$$\mathbb{Z}_n^* = \{[a]_n \in \mathbb{Z}_n : \gcd(a, n) = 1\} .$$

To see that $\mathbb{Z}_n^*$ is well defined, note that for $0 \le a < n$, we have $a = (a + kn) \pmod{n}$ for all integers k. By Exercise 31.2-3, therefore, $\gcd(a, n) = 1$ implies $\gcd(a + kn, n) = 1$ for all integers k. Since $[a]_n = \{a + kn : k \in \mathbb{Z}\}$, the set $\mathbb{Z}_n^*$ is well defined. An example of such a group is

$$\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\} ,$$

where the group operation is multiplication modulo 15. (We have denoted an element $[a]_{15}$ as a, and thus, for example, we denote $[7]_{15}$ as 7.) Figure 31.2(b) shows the group $(\mathbb{Z}_{15}^*, \cdot_{15})$. For example, $8 \cdot 11 = 13 \pmod{15}$, working in $\mathbb{Z}_{15}^*$. The identity for this group is 1.

Theorem 31.13
The system $(\mathbb{Z}_n^*, \cdot_n)$ is a finite abelian group.

Proof Theorem 31.6 implies that $(\mathbb{Z}_n^*, \cdot_n)$ is closed. Associativity and commutativity can be proved for $\cdot_n$ as they were for $+_n$ in the proof of Theorem 31.12. The identity element is $[1]_n$. To show the existence of inverses, let a be an element of $\mathbb{Z}_n^*$ and let (d, x, y) be returned by EXTENDED-EUCLID(a, n). Then we have $d = 1$, since $a \in \mathbb{Z}_n^*$, and

$$ax + ny = 1 , \tag{31.20}$$

or equivalently,

$$ax = 1 \pmod{n} .$$

Thus $[x]_n$ is a multiplicative inverse of $[a]_n$, modulo n. Furthermore, we claim that $[x]_n \in \mathbb{Z}_n^*$. To see why, equation (31.20) demonstrates that the smallest positive linear combination of x and n must be 1. Therefore, Theorem 31.2 implies that $\gcd(x, n) = 1$. We defer the proof that inverses are uniquely defined until Corollary 31.26 in Section 31.4. ■

As an example of computing multiplicative inverses, suppose that $a = 5$ and $n = 11$. Then EXTENDED-EUCLID(a, n) returns $(d, x, y) = (1, -2, 1)$, so that $1 = 5 \cdot (-2) + 11 \cdot 1$. Thus, $[-2]_{11}$ (i.e., $[9]_{11}$) is the multiplicative inverse of $[5]_{11}$.

When working with the groups $(\mathbb{Z}_n, +_n)$ and $(\mathbb{Z}_n^*, \cdot_n)$ in the remainder of this chapter, we follow the convenient practice of denoting equivalence classes by their representative elements and denoting the operations $+_n$ and $\cdot_n$ by the usual

arithmetic notations $+$ and $\cdot$ (or juxtaposition, so that $ab = a \cdot b$) respectively. Furthermore, equivalences modulo n may also be interpreted as equations in $\mathbb{Z}_n$. For example, the following two statements are equivalent:

$$ax = b \pmod{n}$$

and

$$[a]_n \cdot_n [x]_n = [b]_n .$$

As a further convenience, we sometimes refer to a group $(S, \oplus)$ merely as S when the operation $\oplus$ is understood from context. We may thus refer to the groups $(\mathbb{Z}_n, +_n)$ and $(\mathbb{Z}_n^*, \cdot_n)$ as just $\mathbb{Z}_n$ and $\mathbb{Z}_n^*$, respectively.

We denote the (multiplicative) inverse of an element a by $(a^{-1} \bmod n)$. Division in $\mathbb{Z}_n^*$ is defined by the equation $a/b = ab^{-1} \pmod{n}$. For example, in $\mathbb{Z}_{15}^*$ we have that $7^{-1} = 13 \pmod{15}$, since $7 \cdot 13 = 91 = 1 \pmod{15}$, so that $2/7 = 2 \cdot 13 = 11 \pmod{15}$.

The size of $\mathbb{Z}_n^*$ is denoted $\phi(n)$. This function, known as ***Euler's phi function***, satisfies the equation

$$\phi(n) = n \prod_{p \text{ prime such that } p \mid n} \left(1 - \frac{1}{p}\right) , \tag{31.21}$$

so that p runs over all the primes dividing n (including n itself, if n is prime). We won't prove this formula here. Intuitively, begin with a list of the n remainders $\{0, 1, \ldots, n-1\}$ and then, for each prime p that divides n, cross out every multiple of p in the list. For example, since the prime divisors of 45 are 3 and 5,

$$\begin{aligned} \phi(45) &= 45\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{5}\right) \\ &= 45\left(\frac{2}{3}\right)\left(\frac{4}{5}\right) \\ &= 24 . \end{aligned}$$

If p is prime, then $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$, and

$$\begin{aligned} \phi(p) &= p\left(1 - \frac{1}{p}\right) \\ &= p - 1 . \end{aligned} \tag{31.22}$$

If n is composite, then $\phi(n) < n - 1$, although it can be shown that

$$\phi(n) > \frac{n}{e^{\gamma} \ln\ln n + 3/\ln\ln n} \tag{31.23}$$

for $n \geq 3$, where $\gamma = 0.5772156649\ldots$ is ***Euler's constant***. A somewhat simpler (but looser) lower bound for $n > 5$ is

$$\phi(n) > \frac{n}{6 \ln \ln n} . \tag{31.24}$$

The lower bound (31.23) is essentially the best possible, since

$$\liminf_{n \to \infty} \frac{\phi(n)}{n / \ln \ln n} = e^{-\gamma} . \tag{31.25}$$

Subgroups

If $(S, \oplus)$ is a group, $S' \subseteq S$, and $(S', \oplus)$ is also a group, then $(S', \oplus)$ is a ***subgroup*** of $(S, \oplus)$. For example, the even integers form a subgroup of the integers under the operation of addition. The following theorem, whose proof we leave as Exercise 31.3-3, provides a useful tool for recognizing subgroups.

Theorem 31.14 (A nonempty closed subset of a finite group is a subgroup)
If $(S, \oplus)$ is a finite group and S' is any nonempty subset of S such that $a \oplus b \in S'$ for all $a, b \in S'$, then $(S', \oplus)$ is a subgroup of $(S, \oplus)$. ■

For example, the set $\{0, 2, 4, 6\}$ forms a subgroup of $\mathbb{Z}_8$, since it is nonempty and closed under the operation $+$ (that is, it is closed under $+_8$).

The following theorem, whose proof is omitted, provides an extremely useful constraint on the size of a subgroup.

Theorem 31.15 (Lagrange's theorem)
If $(S, \oplus)$ is a finite group and $(S', \oplus)$ is a subgroup of $(S, \oplus)$, then $|S'|$ is a divisor of $|S|$. ■

A subgroup S' of a group S is a ***proper*** subgroup if $S' \neq S$. We'll use the following corollary in the analysis in Section 31.8 of the Miller-Rabin primality test procedure.

Corollary 31.16
If S' is a proper subgroup of a finite group S, then $|S'| \leq |S|/2$. ■

Subgroups generated by an element

Theorem 31.14 affords us a straightforward way to produce a subgroup of a finite group $(S, \oplus)$: choose an element a and take all elements that can be generated from a using the group operation. Specifically, define $a^{(k)}$ for $k \geq 1$ by

$$a^{(k)} = \bigoplus_{i=1}^{k} a = \underbrace{a \oplus a \oplus \cdots \oplus a}_{k} .$$

For example, taking $a = 2$ in the group $\mathbb{Z}_6$ yields the sequence

$$a^{(1)}, a^{(2)}, a^{(3)}, \ldots = 2, 4, 0, 2, 4, 0, 2, 4, 0, \ldots .$$

We have $a^{(k)} = ka \bmod n$ in the group $\mathbb{Z}_n$, and $a^{(k)} = a^k \bmod n$ in the group $\mathbb{Z}_n^*$. We define the ***subgroup generated by a***, denoted $\langle a \rangle$ or $(\langle a \rangle, \oplus)$, by

$$\langle a \rangle = \left\{ a^{(k)} : k \geq 1 \right\} .$$

We say that a ***generates*** the subgroup $\langle a \rangle$ or that a is a ***generator*** of $\langle a \rangle$. Since S is finite, $\langle a \rangle$ is a finite subset of S, possibly including all of S. Since the associativity of $\oplus$ implies

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)} ,$$

$\langle a \rangle$ is closed and therefore, by Theorem 31.14, $\langle a \rangle$ is a subgroup of S. For example, in $\mathbb{Z}_6$, we have

$$\begin{aligned} \langle 0 \rangle &= \{0\} , \\ \langle 1 \rangle &= \{0, 1, 2, 3, 4, 5\} , \\ \langle 2 \rangle &= \{0, 2, 4\} . \end{aligned}$$

Similarly, in $\mathbb{Z}_7^*$, we have

$$\begin{aligned} \langle 1 \rangle &= \{1\} , \\ \langle 2 \rangle &= \{1, 2, 4\} , \\ \langle 3 \rangle &= \{1, 2, 3, 4, 5, 6\} . \end{aligned}$$

The ***order*** of a (in the group S), denoted $\mathrm{ord}(a)$, is defined as the smallest positive integer t such that $a^{(t)} = e$. (Recall that $e \in S$ is the group identity.)

Theorem 31.17
For any finite group $(S, \oplus)$ and any $a \in S$, the order of a is equal to the size of the subgroup it generates, or $\mathrm{ord}(a) = |\langle a \rangle|$.

Proof Let $t = \mathrm{ord}(a)$. Since $a^{(t)} = e$ and $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$ for $k \geq 1$, if $i > t$, then $a^{(i)} = a^{(j)}$ for some $j < i$. Therefore, as we generate elements by a, we see no new elements after $a^{(t)}$. Thus, $\langle a \rangle = \left\{ a^{(1)}, a^{(2)}, \ldots, a^{(t)} \right\}$, and so $|\langle a \rangle| \leq t$. To show that $|\langle a \rangle| \geq t$, we show that each element of the sequence $a^{(1)}, a^{(2)}, \ldots, a^{(t)}$ is distinct. Suppose for the purpose of contradiction that $a^{(i)} = a^{(j)}$ for some i and j satisfying $1 \leq i < j \leq t$. Then, $a^{(i+k)} = a^{(j+k)}$ for

$k \geq 0$. But this equation implies that $a^{(i+(t-j))} = a^{(j+(t-j))} = e$, a contradiction, since $i + (t - j) < t$ but t is the least positive value such that $a^{(t)} = e$. Therefore, each element of the sequence $a^{(1)}, a^{(2)}, \ldots, a^{(t)}$ is distinct, and $|\langle a \rangle| \geq t$. We conclude that $\text{ord}(a) = |\langle a \rangle|$. ■

Corollary 31.18
The sequence $a^{(1)}, a^{(2)}, \ldots$ is periodic with period $t = \text{ord}(a)$, that is, $a^{(i)} = a^{(j)}$ if and only if $i = j \pmod{t}$. ■

Consistent with the above corollary, we define $a^{(0)}$ as e and $a^{(i)}$ as $a^{(i \bmod t)}$, where $t = \text{ord}(a)$, for all integers i.

Corollary 31.19
If $(S, \oplus)$ is a finite group with identity e, then for all $a \in S$,

$$a^{(|S|)} = e .$$

Proof Lagrange's theorem (Theorem 31.15) implies that $\text{ord}(a) \mid |S|$, and so $|S| = 0 \pmod{t}$, where $t = \text{ord}(a)$. Therefore, $a^{(|S|)} = a^{(0)} = e$. ■

Exercises

31.3-1
Draw the group operation tables for the groups $(\mathbb{Z}_4, +_4)$ and $(\mathbb{Z}_5^*, \cdot_5)$. Show that these groups are isomorphic by exhibiting a one-to-one correspondence f between $\mathbb{Z}_4$ and $\mathbb{Z}_5^*$ such that $a+b = c \pmod 4$ if and only if $f(a) \cdot f(b) = f(c) \pmod 5$.

31.3-2
List all subgroups of $\mathbb{Z}_9$ and of $\mathbb{Z}_{13}^*$.

31.3-3
Prove Theorem 31.14.

31.3-4
Show that if p is prime and e is a positive integer, then

$$\phi(p^e) = p^{e-1}(p - 1) .$$

31.3-5
Show that for any integer $n > 1$ and for any $a \in \mathbb{Z}_n^*$, the function $f_a : \mathbb{Z}_n^* \to \mathbb{Z}_n^*$ defined by $f_a(x) = ax \bmod n$ is a permutation of $\mathbb{Z}_n^*$.

31.4 Solving modular linear equations

We now consider the problem of finding solutions to the equation

$$ax = b \pmod{n}, \tag{31.26}$$

where $a > 0$ and $n > 0$. This problem has several applications. For example, we'll use it in Section 31.7 as part of the procedure to find keys in the RSA public-key cryptosystem. We assume that a, b, and n are given, and we wish to find all values of x, modulo n, that satisfy equation (31.26). The equation may have zero, one, or more than one such solution.

Let $\langle a \rangle$ denote the subgroup of $\mathbb{Z}_n$ generated by a. Since $\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \bmod n : x > 0\}$, equation (31.26) has a solution if and only if $[b] \in \langle a \rangle$. Lagrange's theorem (Theorem 31.15) tells us that $|\langle a \rangle|$ must be a divisor of n. The following theorem gives us a precise characterization of $\langle a \rangle$.

Theorem 31.20
For any positive integers a and n, if $d = \gcd(a, n)$, then we have

$$\begin{aligned}\langle a \rangle &= \langle d \rangle \\ &= \{0, d, 2d, \ldots, ((n/d) - 1)d\}\end{aligned}$$

in $\mathbb{Z}_n$, and thus

$$|\langle a \rangle| = n/d.$$

Proof We begin by showing that $d \in \langle a \rangle$. Recall that EXTENDED-EUCLID(a, n) returns a triple (d, x, y) such that $ax + ny = d$. Thus, $ax = d \pmod{n}$, so that $d \in \langle a \rangle$. In other words, d is a multiple of a in $\mathbb{Z}_n$.

Since $d \in \langle a \rangle$, it follows that every multiple of d belongs to $\langle a \rangle$, because any multiple of a multiple of a is itself a multiple of a. Thus, $\langle a \rangle$ contains every element in $\{0, d, 2d, \ldots, ((n/d) - 1)d\}$. That is, $\langle d \rangle \subseteq \langle a \rangle$.

We now show that $\langle a \rangle \subseteq \langle d \rangle$. If $m \in \langle a \rangle$, then $m = ax \bmod n$ for some integer x, and so $m = ax + ny$ for some integer y. Because $d = \gcd(a, n)$, we know that $d \mid a$ and $d \mid n$, and so $d \mid m$ by equation (31.4). Therefore, $m \in \langle d \rangle$.

Combining these results, we have that $\langle a \rangle = \langle d \rangle$. To see that $|\langle a \rangle| = n/d$, observe that there are exactly n/d multiples of d between 0 and $n - 1$, inclusive. ■

Corollary 31.21
The equation $ax = b \pmod{n}$ is solvable for the unknown x if and only if $d \mid b$, where $d = \gcd(a, n)$.

Proof The equation $ax = b \pmod{n}$ is solvable if and only if $[b] \in \langle a \rangle$, which is the same as saying

$$(b \bmod n) \in \{0, d, 2d, \ldots, ((n/d) - 1)d\} ,$$

by Theorem 31.20. If $0 \le b < n$, then $b \in \langle a \rangle$ if and only if $d \mid b$, since the members of $\langle a \rangle$ are precisely the multiples of d. If $b < 0$ or $b \ge n$, the corollary then follows from the observation that $d \mid b$ if and only if $d \mid (b \bmod n)$, since b and $b \bmod n$ differ by a multiple of n, which is itself a multiple of d. ■

Corollary 31.22
The equation $ax = b \pmod{n}$ either has d distinct solutions modulo n, where $d = \gcd(a, n)$, or it has no solutions.

Proof If $ax = b \pmod{n}$ has a solution, then $b \in \langle a \rangle$. By Theorem 31.17, $\mathrm{ord}(a) = |\langle a \rangle|$, and so Corollary 31.18 and Theorem 31.20 imply that the sequence $ai \bmod n$, for $i = 0, 1, \ldots$, is periodic with period $|\langle a \rangle| = n/d$. If $b \in \langle a \rangle$, then b appears exactly d times in the sequence $ai \bmod n$, for $i = 0, 1, \ldots, n-1$, since the length-(n/d) block of values $\langle a \rangle$ repeats exactly d times as i increases from 0 to $n-1$. The indices x of the d positions for which $ax \bmod n = b$ are the solutions of the equation $ax = b \pmod{n}$. ■

Theorem 31.23
Let $d = \gcd(a, n)$, and suppose that $d = ax' + ny'$ for some integers x' and y' (for example, as computed by EXTENDED-EUCLID). If $d \mid b$, then the equation $ax = b \pmod{n}$ has as one of its solutions the value x_0, where

$$x_0 = x'(b/d) \bmod n .$$

Proof We have

$$\begin{aligned} ax_0 &= ax'(b/d) \pmod{n} \\ &= d(b/d) \pmod{n} \qquad (\text{because } ax' = d \pmod{n}) \\ &= b \pmod{n} , \end{aligned}$$

and thus x_0 is a solution to $ax = b \pmod{n}$. ■

Theorem 31.24
Suppose that the equation $ax = b \pmod{n}$ is solvable (that is, $d \mid b$, where $d = \gcd(a, n)$) and that x_0 is any solution to this equation. Then, this equation has exactly d distinct solutions, modulo n, given by $x_i = x_0 + i(n/d)$ for $i = 0, 1, \ldots, d-1$.

Proof Because $n/d > 0$ and $0 \le i(n/d) < n$ for $i = 0, 1, \ldots, d-1$, the values $x_0, x_1, \ldots, x_{d-1}$ are all distinct, modulo n. Since x_0 is a solution of $ax = b \pmod{n}$, we have $ax_0 \bmod n = b \pmod{n}$. Thus, for $i = 0, 1, \ldots, d-1$, we have

$$\begin{aligned} ax_i \bmod n &= a(x_0 + in/d) \bmod n \\ &= (ax_0 + ain/d) \bmod n \\ &= ax_0 \bmod n \quad \text{(because } d \mid a \text{ implies that } ain/d \text{ is a multiple of } n\text{)} \\ &= b \pmod{n}\,, \end{aligned}$$

and hence $ax_i = b \pmod{n}$, making x_i a solution, too. By Corollary 31.22, the equation $ax = b \pmod{n}$ has exactly d solutions, so that $x_0, x_1, \ldots, x_{d-1}$ must be all of them. ■

We have now developed the mathematics needed to solve the equation $ax = b \pmod{n}$. The procedure MODULAR-LINEAR-EQUATION-SOLVER prints all solutions to this equation. The inputs a and n are arbitrary positive integers, and b is an arbitrary integer.

```
MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)
1  (d, x', y') = EXTENDED-EUCLID(a, n)
2  if d | b
3      x0 = x'(b/d) mod n
4      for i = 0 to d - 1
5          print (x0 + i(n/d)) mod n
6  else print "no solutions"
```

As an example of the operation of MODULAR-LINEAR-EQUATION-SOLVER, consider the equation $14x = 30 \pmod{100}$ (and thus $a = 14$, $b = 30$, and $n = 100$). Calling EXTENDED-EUCLID in line 1 gives $(d, x', y') = (2, -7, 1)$. Since $2 \mid 30$, lines 3–5 execute. Line 3 computes $x_0 = (-7)(15) \bmod 100 = 95$. The **for** loop of lines 4–5 prints the two solutions, 95 and 45.

The procedure MODULAR-LINEAR-EQUATION-SOLVER works as follows. The call to EXTENDED-EUCLID in line 1 returns a triple (d, x', y') such that $d = \gcd(a, n)$ and $d = ax' + ny'$. Therefore, x' is a solution to the equation $ax' = d \pmod{n}$. If d does not divide b, then the equation $ax = b \pmod{n}$ has no solution, by Corollary 31.21. Line 2 checks to see whether $d \mid b$, and if not, line 6 reports that there are no solutions. Otherwise, line 3 computes a solution x_0 to $ax = b \pmod{n}$, as Theorem 31.23 suggests. Given one solution, Theorem 31.24 states that adding multiples of (n/d), modulo n, yields the other

$d - 1$ solutions. The **for** loop of lines 4–5 prints out all d solutions, beginning with x_0 and spaced n/d apart, modulo n.

MODULAR-LINEAR-EQUATION-SOLVER performs $O(\lg n + \gcd(a, n))$ arithmetic operations, since EXTENDED-EUCLID performs $O(\lg n)$ arithmetic operations, and each iteration of the **for** loop of lines 4–5 performs a constant number of arithmetic operations.

The following corollaries of Theorem 31.24 give specializations of particular interest.

Corollary 31.25
For any $n > 1$, if $\gcd(a, n) = 1$, then the equation $ax = b \pmod{n}$ has a unique solution, modulo n. ■

If $b = 1$, a common case of considerable interest, the x that solves the equation is a ***multiplicative inverse*** of a, modulo n.

Corollary 31.26
For any $n > 1$, if $\gcd(a, n) = 1$, then the equation $ax = 1 \pmod{n}$ has a unique solution, modulo n. Otherwise, it has no solution. ■

Thanks to Corollary 31.26, the notation $a^{-1} \bmod n$ refers to *the* multiplicative inverse of a, modulo n, when a and n are relatively prime. If $\gcd(a, n) = 1$, then the unique solution to the equation $ax = 1 \pmod{n}$ is the integer x returned by EXTENDED-EUCLID, since the equation

$$\gcd(a, n) = 1 = ax + ny$$

implies $ax = 1 \pmod{n}$. Thus, EXTENDED-EUCLID can compute $a^{-1} \bmod n$ efficiently.

Exercises

31.4-1
Find all solutions to the equation $35x = 10 \pmod{50}$.

31.4-2
Prove that the equation $ax = ay \pmod{n}$ implies $x = y \pmod{n}$ whenever $\gcd(a, n) = 1$. Show that the condition $\gcd(a, n) = 1$ is necessary by supplying a counterexample with $\gcd(a, n) > 1$.

31.4-3
Consider the following change to line 3 of the procedure MODULAR-LINEAR-EQUATION-SOLVER:

3 $\quad x_0 = x'(b/d) \bmod (n/d)$

With this change, will the procedure still work? Explain why or why not.

★ ***31.4-4***
Let p be prime and $f(x) = (f_0 + f_1 x + \cdots + f_t x^t) \pmod p$ be a polynomial of degree t, with coefficients f_i drawn from $\mathbb{Z}_p$. We say that $a \in \mathbb{Z}_p$ is a ***zero*** of f if $f(a) = 0 \pmod p$. Prove that if a is a zero of f, then $f(x) = (x - a)g(x) \pmod p$ for some polynomial $g(x)$ of degree $t - 1$. Prove by induction on t that if p is prime, then a polynomial $f(x)$ of degree t can have at most t distinct zeros modulo p.

31.5 The Chinese remainder theorem

Around 100 C.E., the Chinese mathematician Sun-Tsŭ solved the problem of finding those integers x that leave remainders 2, 3, and 2 when divided by 3, 5, and 7 respectively. One such solution is $x = 23$, and all solutions are of the form $23 + 105k$ for arbitrary integers k. The "Chinese remainder theorem" provides a correspondence between a system of equations modulo a set of pairwise relatively prime moduli (for example, 3, 5, and 7) and an equation modulo their product (for example, 105).

The Chinese remainder theorem has two major applications. Let the integer n be factored as $n = n_1 n_2 \cdots n_k$, where the factors n_i are pairwise relatively prime. First, the Chinese remainder theorem is a descriptive "structure theorem" that describes the structure of $\mathbb{Z}_n$ as identical to that of the Cartesian product $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$ with componentwise addition and multiplication modulo n_i in the ith component. Second, this description helps in designing efficient algorithms, since working in each of the systems $\mathbb{Z}_{n_i}$ can be more efficient (in terms of bit operations) than working modulo n.

Theorem 31.27 (Chinese remainder theorem)
Let $n = n_1 n_2 \cdots n_k$, where the n_i are pairwise relatively prime. Consider the correspondence

$$a \leftrightarrow (a_1, a_2, \ldots, a_k) , \tag{31.27}$$

where $a \in \mathbb{Z}_n$, $a_i \in \mathbb{Z}_{n_i}$, and

$$a_i = a \bmod n_i$$

for $i = 1, 2, \ldots, k$. Then, mapping (31.27) is a one-to-one mapping (bijection) between $\mathbb{Z}_n$ and the Cartesian product $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$. Operations performed on the elements of $\mathbb{Z}_n$ can be equivalently performed on the corresponding k-tuples by performing the operations independently in each coordinate position in the appropriate system. That is, if

$$a \leftrightarrow (a_1, a_2, \ldots, a_k) ,$$
$$b \leftrightarrow (b_1, b_2, \ldots, b_k) ,$$

then

$$(a + b) \bmod n \leftrightarrow ((a_1 + b_1) \bmod n_1, \ldots, (a_k + b_k) \bmod n_k) , \tag{31.28}$$
$$(a - b) \bmod n \leftrightarrow ((a_1 - b_1) \bmod n_1, \ldots, (a_k - b_k) \bmod n_k) , \tag{31.29}$$
$$(ab) \bmod n \leftrightarrow (a_1 b_1 \bmod n_1, \ldots, a_k b_k \bmod n_k) . \tag{31.30}$$

Proof Let's see how to translate between the two representations. Going from a to $(a_1, a_2, \ldots, a_k)$ requires only k "mod" operations. The reverse—computing a from inputs $(a_1, a_2, \ldots, a_k)$—is only slightly more complicated.

We begin by defining $m_i = n/n_i$ for $i = 1, 2, \ldots, k$. Thus, m_i is the product of all of the n_j's other than n_i: $m_i = n_1 n_2 \cdots n_{i-1} n_{i+1} \cdots n_k$. We next define

$$c_i = m_i (m_i^{-1} \bmod n_i) \tag{31.31}$$

for $i = 1, 2, \ldots, k$. Equation (31.31) is well defined: since m_i and n_i are relatively prime (by Theorem 31.6), Corollary 31.26 guarantees that $m_i^{-1} \bmod n_i$ exists. Here is how to compute a as a function of the a_i and c_i:

$$a = (a_1 c_1 + a_2 c_2 + \cdots + a_k c_k) \pmod{n} . \tag{31.32}$$

We now show that equation (31.32) ensures that $a = a_i \pmod{n_i}$ for $i = 1, 2, \ldots, k$. If $j \neq i$, then $m_j = 0 \pmod{n_i}$, which implies that $c_j = m_j = 0 \pmod{n_i}$. Note also that $c_i = 1 \pmod{n_i}$, from equation (31.31). We thus have the appealing and useful correspondence

$$c_i \leftrightarrow (0, 0, \ldots, 0, 1, 0, \ldots, 0) ,$$

a vector that has 0s everywhere except in the ith coordinate, where it has a 1. The c_i thus form a "basis" for the representation, in a certain sense. For each i, therefore, we have

$$\begin{aligned} a &= a_i c_i && \pmod{n_i} \\ &= a_i m_i (m_i^{-1} \bmod n_i) && \pmod{n_i} \\ &= a_i && \pmod{n_i} , \end{aligned}$$

which is what we wished to show: our method of computing a from the a_i's produces a result a that satisfies the constraints $a = a_i \pmod{n_i}$ for $i = 1, 2, \ldots, k$. The correspondence is one-to-one, since we can transform in both directions. Finally, equations (31.28)–(31.30) follow directly from Exercise 31.1-7, since $x \bmod n_i = (x \bmod n) \bmod n_i$ for any x and $i = 1, 2, \ldots, k$. ■

We'll use the following corollaries later in this chapter.

Corollary 31.28
If $n_1, n_2, \ldots, n_k$ are pairwise relatively prime and $n = n_1 n_2 \cdots n_k$, then for any integers $a_1, a_2, \ldots, a_k$, the set of simultaneous equations

$$x = a_i \pmod{n_i} ,$$

for $i = 1, 2, \ldots, k$, has a unique solution modulo n for the unknown x. ■

Corollary 31.29
If $n_1, n_2, \ldots, n_k$ are pairwise relatively prime and $n = n_1 n_2 \cdots n_k$, then for all integers x and a,

$$x = a \pmod{n_i}$$

for $i = 1, 2, \ldots, k$ if and only if

$$x = a \pmod{n} .$$ ■

As an example of the application of the Chinese remainder theorem, suppose that you are given the two equations

$$\begin{aligned} a &= 2 \pmod{5} , \\ a &= 3 \pmod{13} , \end{aligned}$$

so that $a_1 = 2$, $n_1 = m_2 = 5$, $a_2 = 3$, and $n_2 = m_1 = 13$, and you wish to compute a mod 65, since $n = n_1 n_2 = 65$. Because $13^{-1} = 2 \pmod{5}$ and $5^{-1} = 8 \pmod{13}$, you compute

$$\begin{aligned} c_1 &= 13 \cdot (2 \bmod 5) = 26 , \\ c_2 &= 5 \cdot (8 \bmod 13) = 40 , \end{aligned}$$

and

$$\begin{aligned} a &= 2 \cdot 26 + 3 \cdot 40 \pmod{65} \\ &= 52 + 120 \pmod{65} \\ &= 42 \pmod{65} . \end{aligned}$$

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

Figure 31.3 An illustration of the Chinese remainder theorem for $n_1 = 5$ and $n_2 = 13$. For this example, $c_1 = 26$ and $c_2 = 40$. In row i, column j is shown the value of a, modulo 65, such that $a \bmod 5 = i$ and $a \bmod 13 = j$. Note that row 0, column 0 contains a 0. Similarly, row 4, column 12 contains a 64 (equivalent to -1). Since $c_1 = 26$, moving down a row increases a by 26. Similarly, $c_2 = 40$ means that moving right by a column increases a by 40. Increasing a by 1 corresponds to moving diagonally downward and to the right, wrapping around from the bottom to the top and from the right to the left.

See Figure 31.3 for an illustration of the Chinese remainder theorem, modulo 65.

Thus, you can work modulo n by working modulo n directly or by working in the transformed representation using separate modulo n_i computations, as convenient. The computations are entirely equivalent.

Exercises

31.5-1

Find all solutions to the equations $x = 4 \pmod 5$ and $x = 5 \pmod{11}$.

31.5-2

Find all integers x that leave remainders 1, 2, and 3 when divided by 9, 8, and 7, respectively.

31.5-3

Argue that, under the definitions of Theorem 31.27, if $\gcd(a, n) = 1$, then

$$(a^{-1} \bmod n) \leftrightarrow ((a_1^{-1} \bmod n_1), (a_2^{-1} \bmod n_2), \ldots, (a_k^{-1} \bmod n_k)) .$$

31.5-4

Under the definitions of Theorem 31.27, prove that for any polynomial f, the number of roots of the equation $f(x) = 0 \pmod n$ equals the product of the number of roots of each of the equations $f(x) = 0 \pmod{n_1}$, $f(x) = 0 \pmod{n_2}$, $\ldots$, $f(x) = 0 \pmod{n_k}$.

31.6 Powers of an element

Along with considering the multiples of a given element a, modulo n, we often consider the sequence of powers of a, modulo n, where $a \in \mathbb{Z}_n^*$:

$$a^0, a^1, a^2, a^3, \ldots,$$

modulo n. Indexing from 0, the 0th value in this sequence is $a^0 \bmod n = 1$, and the ith value is $a^i \bmod n$. For example, the powers of 3 modulo 7 are

i	0	1	2	3	4	5	6	7	8	9	10	11	$\cdots$
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2	6	4	5	$\cdots$

and the powers of 2 modulo 7 are

i	0	1	2	3	4	5	6	7	8	9	10	11	$\cdots$
$2^i \bmod 7$	1	2	4	1	2	4	1	2	4	1	2	4	$\cdots$

In this section, let $\langle a \rangle$ denote the subgroup of $\mathbb{Z}_n^*$ generated by a through repeated multiplication, and let $\mathrm{ord}_n(a)$ (the "order of a, modulo n") denote the order of a in $\mathbb{Z}_n^*$. For example, $\langle 2 \rangle = \{1, 2, 4\}$ in $\mathbb{Z}_7^*$, and $\mathrm{ord}_7(2) = 3$. Using the definition of the Euler phi function $\phi(n)$ as the size of $\mathbb{Z}_n^*$ (see Section 31.3), we now translate Corollary 31.19 into the notation of $\mathbb{Z}_n^*$ to obtain Euler's theorem and specialize it to $\mathbb{Z}_p^*$, where p is prime, to obtain Fermat's theorem.

Theorem 31.30 (Euler's theorem)
For any integer $n > 1$,

$$a^{\phi(n)} = 1 \pmod{n} \text{ for all } a \in \mathbb{Z}_n^* . \qquad \blacksquare$$

Theorem 31.31 (Fermat's theorem)
If p is prime, then

$$a^{p-1} = 1 \pmod{p} \text{ for all } a \in \mathbb{Z}_p^* .$$

Proof By equation (31.22), $\phi(p) = p - 1$ if p is prime. $\blacksquare$

Fermat's theorem applies to every element in $\mathbb{Z}_p$ except 0, since $0 \notin \mathbb{Z}_p^*$. For all $a \in \mathbb{Z}_p$, however, we have $a^p = a \pmod{p}$ if p is prime.

If $\mathrm{ord}_n(g) = |\mathbb{Z}_n^*|$, then every element in $\mathbb{Z}_n^*$ is a power of g, modulo n, and g is a ***primitive root*** or a ***generator*** of $\mathbb{Z}_n^*$. For example, 3 is a primitive root, modulo 7, but 2 is not a primitive root, modulo 7. If $\mathbb{Z}_n^*$ possesses a primitive root, the group $\mathbb{Z}_n^*$ is ***cyclic***. We omit the proof of the following theorem, which is proven by Niven and Zuckerman [345].

Theorem 31.32
The values of $n > 1$ for which $\mathbb{Z}_n^*$ is cyclic are 2, 4, p^e, and $2p^e$, for all primes $p > 2$ and all positive integers e. ■

If g is a primitive root of $\mathbb{Z}_n^*$ and a is any element of $\mathbb{Z}_n^*$, then there exists a z such that $g^z = a \pmod{n}$. This z is a ***discrete logarithm*** or an ***index*** of a, modulo n, to the base g. We denote this value as $\mathrm{ind}_{n,g}(a)$.

Theorem 31.33 (Discrete logarithm theorem)
If g is a primitive root of $\mathbb{Z}_n^*$, then the equation $g^x = g^y \pmod{n}$ holds if and only if the equation $x = y \pmod{\phi(n)}$ holds.

Proof Suppose first that $x = y \pmod{\phi(n)}$. Then, we have $x = y + k\phi(n)$ for some integer k, and thus

$$\begin{aligned} g^x &= g^{y+k\phi(n)} && \pmod{n} \\ &= g^y \cdot (g^{\phi(n)})^k && \pmod{n} \\ &= g^y \cdot 1^k && \pmod{n} \qquad \text{(by Euler's theorem)} \\ &= g^y && \pmod{n}\ . \end{aligned}$$

Conversely, suppose that $g^x = g^y \pmod{n}$. Because the sequence of powers of g generates every element of $\langle g \rangle$ and $|\langle g \rangle| = \phi(n)$, Corollary 31.18 implies that the sequence of powers of g is periodic with period $\phi(n)$. Therefore, if $g^x = g^y \pmod{n}$, we must have $x = y \pmod{\phi(n)}$. ■

Let's now turn our attention to the square roots of 1, modulo a prime power. The following properties will be useful to justify the primality-testing algorithm in Section 31.8.

Theorem 31.34
If p is an odd prime and $e \geq 1$, then the equation

$$x^2 = 1 \pmod{p^e} \tag{31.33}$$

has only two solutions, namely $x = 1$ and $x = -1$.

Proof By Exercise 31.6-2, equation (31.33) is equivalent to

$$p^e \mid (x-1)(x+1)\ .$$

Since $p > 2$, we can have $p \mid (x-1)$ or $p \mid (x+1)$, but not both. (Otherwise, by property (31.3), p would also divide their difference $(x+1)-(x-1) = 2$.) If $p \nmid (x-1)$, then $\gcd(p^e, x-1) = 1$, and by Corollary 31.5, we would have $p^e \mid (x+1)$. That is, $x = -1 \pmod{p^e}$. Symmetrically, if $p \nmid (x+1)$,

then $\gcd(p^e, x+1) = 1$, and Corollary 31.5 implies that $p^e \mid (x-1)$, so that $x = 1 \pmod{p^e}$. Therefore, either $x = -1 \pmod{p^e}$ or $x = 1 \pmod{p^e}$. ■

A number x is a ***nontrivial square root of 1, modulo n***, if it satisfies the equation $x^2 = 1 \pmod{n}$ but x is equivalent to neither of the two "trivial" square roots: 1 or -1, modulo n. For example, 6 is a nontrivial square root of 1, modulo 35. We'll use the following corollary to Theorem 31.34 in Section 31.8 to prove the Miller-Rabin primality-testing procedure correct.

Corollary 31.35
If there exists a nontrivial square root of 1, modulo n, then n is composite.

Proof By the contrapositive of Theorem 31.34, if there exists a nontrivial square root of 1, modulo n, then n cannot be an odd prime or a power of an odd prime. Nor can n be 2, because if $x^2 = 1 \pmod{2}$, then $x = 1 \pmod{2}$, and therefore, all square roots of 1, modulo 2, are trivial. Thus, n cannot be prime. Finally, we must have $n > 1$ for a nontrivial square root of 1 to exist. Therefore, n must be composite. ■

Raising to powers with repeated squaring

A frequently occurring operation in number-theoretic computations is raising one number to a power modulo another number, also known as ***modular exponentiation***. More precisely, we would like an efficient way to compute $a^b \bmod n$, where a and b are nonnegative integers and n is a positive integer. Modular exponentiation is an essential operation in many primality-testing routines and in the RSA public-key cryptosystem. The method of ***repeated squaring*** solves this problem efficiently.

Repeated squaring is based on the following formula to compute a^b for nonnegative integers a and b:

$$a^b = \begin{cases} 1 & \text{if } b = 0 , \\ (a^{b/2})^2 & \text{if } b > 0 \text{ and } b \text{ is even} , \\ a \cdot a^{b-1} & \text{if } b > 0 \text{ and } b \text{ is odd} . \end{cases} \tag{31.34}$$

The last case, where b is odd, reduces to the one of the first two cases, since if b is odd, then $b-1$ is even. The recursive procedure MODULAR-EXPONENTIATION on the next page computes $a^b \bmod n$ using equation (31.34), but performing all computations modulo n. The term "repeated squaring" comes from squaring the intermediate result $d = a^{b/2}$ in line 5. Figure 31.4 shows the values of the parameter b, the local variable d, and the value returned at each level of the recursion for the call MODULAR-EXPONENTIATION$(7, 560, 561)$, which returns the result 1.

b	560	280	140	70	35	34	17	16			8	4	2	1	0
d	67	166	298	241	355	160	103	526	157	49	7	1			–
returned value	1	67	166	298	241	355	160	103	526	157	49	7	1		

Figure 31.4 The values of the parameter b, the local variable d, and the value returned for recursive calls of MODULAR-EXPONENTIATION with parameter values $a = 7$, $b = 560$, and $n = 561$. The value returned by each recursive call is assigned directly to d. The result of the call with $a = 7$, $b = 560$, and $n = 561$ is 1.

```
MODULAR-EXPONENTIATION(a, b, n)
1  if b == 0
2      return 1
3  elseif b mod 2 == 0
4      d = MODULAR-EXPONENTIATION(a, b/2, n)     // b is even
5      return (d · d) mod n
6  else d = MODULAR-EXPONENTIATION(a, b − 1, n)  // b is odd
7      return (a · d) mod n
```

The total number of recursive calls depends on the number of bits of b and the values of these bits. Assume that $b > 0$ and that the most significant bit of b is a 1. Each 0 generates one recursive call (in line 4), and each 1 generates two recursive calls (one in line 6 followed by one in line 4 because if b is odd, then $b - 1$ is even). If the inputs a, b, and n are β-bit numbers, then there are between β and $2\beta - 1$ recursive calls altogether, the total number of arithmetic operations required is $O(\beta)$, and the total number of bit operations required is $O(\beta^3)$.

Exercises

31.6-1

Draw a table showing the order of every element in $\mathbb{Z}_{11}^*$. Pick the smallest primitive root g and compute a table giving $\text{ind}_{11,g}(x)$ for all $x \in \mathbb{Z}_{11}^*$.

31.6-2

Show that $x^2 = 1 \pmod{p^e}$ is equivalent to $p^e \mid (x-1)(x+1)$.

31.6-3

Rewrite the third case of MODULAR-EXPONENTIATION, where b is odd, so that if b has β bits and the most significant bit is 1, then there are always exactly β recursive calls.

31.6-4
Give a nonrecursive (i.e., iterative) version of MODULAR-EXPONENTIATION.

31.6-5
Assuming that you know $\phi(n)$, explain how to compute $a^{-1} \bmod n$ for any $a \in \mathbb{Z}_n^*$ using the procedure MODULAR-EXPONENTIATION.

31.7 The RSA public-key cryptosystem

With a public-key cryptosystem, you can ***encrypt*** messages sent between two communicating parties so that an eavesdropper who overhears the encrypted messages will not be able to decode, or ***decrypt***, them. A public-key cryptosystem also enables a party to append an unforgeable "digital signature" to the end of an electronic message. Such a signature is the electronic version of a handwritten signature on a paper document. It can be easily checked by anyone, forged by no one, yet loses its validity if any bit of the message is altered. It therefore provides authentication of both the identity of the signer and the contents of the signed message. It is the perfect tool for electronically signed business contracts, electronic checks, electronic purchase orders, and other electronic communications that parties wish to authenticate.

The RSA public-key cryptosystem relies on the dramatic difference between the ease of finding large prime numbers and the difficulty of factoring the product of two large prime numbers. Section 31.8 describes an efficient procedure for finding large prime numbers.

Public-key cryptosystems

In a public-key cryptosystem, each participant has both a ***public key*** and a ***secret key***. Each key is a piece of information. For example, in the RSA cryptosystem, each key consists of a pair of integers. The participants "Alice" and "Bob" are traditionally used in cryptography examples. We denote the public keys for Alice and Bob as P_A and P_B, respectively, and likewise the secret keys are S_A for Alice and S_B for Bob.

Each participant creates his or her own public and secret keys. Secret keys are kept secret, but public keys can be revealed to anyone or even published. In fact, it is often convenient to assume that everyone's public key is available in a public directory, so that any participant can easily obtain the public key of any other participant.

Figure 31.5 Encryption in a public key system. Bob encrypts the message M using Alice's public key P_A and transmits the resulting ciphertext $C = P_A(M)$ over a communication channel to Alice. An eavesdropper who captures the transmitted ciphertext gains no information about M. Alice receives C and decrypts it using her secret key to obtain the original message $M = S_A(C)$.

The public and secret keys specify functions that can be applied to any message. Let $\mathcal{D}$ denote the set of permissible messages. For example, $\mathcal{D}$ might be the set of all finite-length bit sequences. The simplest, and original, formulation of public-key cryptography requires one-to-one functions from $\mathcal{D}$ to itself, based on the public and secret keys. We denote the function based on Alice's public key P_A by $P_A()$ and the function based on her secret key S_A by $S_A()$. The functions $P_A()$ and $S_A()$ are thus permutations of $\mathcal{D}$. We assume that the functions $P_A()$ and $S_A()$ are efficiently computable given the corresponding keys P_A and S_A.

The public and secret keys for any participant are a "matched pair" in that they specify functions that are inverses of each other. That is,

$$M = S_A(P_A(M)) , \tag{31.35}$$

$$M = P_A(S_A(M)) \tag{31.36}$$

for any message $M \in \mathcal{D}$. Transforming M with the two keys P_A and S_A successively, in either order, yields back the original message M.

A public-key cryptosystem requires that Alice, and only Alice, be able to compute the function $S_A()$ in any practical amount of time. This assumption is crucial to keeping encrypted messages sent to Alice private and to knowing that Alice's digital signatures are authentic. Alice must keep her key S_A secret. If she does not, whoever else has access to S_A can decrypt messages intended only for Alice and can also forge her digital signature. The assumption that only Alice can reasonably compute $S_A()$ must hold even though everyone knows P_A and can compute $P_A()$, the inverse function to $S_A()$, efficiently. These requirements appear formidable, but we'll see how to satisfy them.

In a public-key cryptosystem, encryption works as shown in Figure 31.5. Suppose that Bob wishes to send Alice a message M encrypted so that it looks like

Figure 31.6 Digital signatures in a public-key system. Alice signs the message M' by appending her digital signature $\sigma = S_A(M')$ to it. She transmits the message/signature pair (M', σ) to Bob, who verifies it by checking the equation $M' = P_A(\sigma)$. If the equation holds, he accepts (M', σ) as a message that Alice has signed.

unintelligible gibberish to an eavesdropper. The scenario for sending the message goes as follows.

- Bob obtains Alice's public key P_A, perhaps from a public directory or perhaps directly from Alice.
- Bob computes the ***ciphertext*** $C = P_A(M)$ corresponding to the message M and sends C to Alice.
- When Alice receives the ciphertext C, she applies her secret key S_A to retrieve the original message: $S_A(C) = S_A(P_A(M)) = M$.

Because $S_A()$ and $P_A()$ are inverse functions, Alice can compute M from C. Because only Alice is able to compute $S_A()$, only Alice can compute M from C. Because Bob encrypts M using $P_A()$, only Alice can understand the transmitted message.

Digital signatures can be implemented within this formulation of a public-key cryptosystem. (There are other ways to construct digital signatures, but we won't go into them here.) Suppose now that Alice wishes to send Bob a digitally signed response M'. Figure 31.6 shows how the digital-signature scenario proceeds.

- Alice computes her ***digital signature*** σ for the message M' using her secret key S_A and the equation $\sigma = S_A(M')$.
- Alice sends the message/signature pair (M', σ) to Bob.
- When Bob receives (M', σ), he can verify that it originated from Alice by using Alice's public key to verify the equation $M' = P_A(\sigma)$. (Presumably, M' contains Alice's name, so that Bob knows whose public key to use.) If the equation holds, then Bob concludes that the message M' was actually signed by Alice. If

the equation fails to hold, Bob concludes either that the information he received was corrupted by transmission errors or that the pair (M', σ) is an attempted forgery.

Because a digital signature provides both authentication of the signer's identity and authentication of the contents of the signed message, it is analogous to a handwritten signature at the end of a written document.

A digital signature must be verifiable by anyone who has access to the signer's public key. A signed message can be verified by one party and then passed on to other parties who can also verify the signature. For example, the message might be an electronic check from Alice to Bob. After Bob verifies Alice's signature on the check, he can give the check to his bank, who can then also verify the signature and effect the appropriate funds transfer.

A signed message may or may not be encrypted. The message can be "in the clear" and not protected from disclosure. By composing the above protocols for encryption and for signatures, Alice can create a message to Bob that is both signed and encrypted. Alice first appends her digital signature to the message and then encrypts the resulting message/signature pair with Bob's public key. Bob decrypts the received message with his secret key to obtain both the original message and its digital signature. Bob can then verify the signature using Alice's public key. The corresponding combined process using paper-based systems would be to sign the paper document and then seal the document inside a paper envelope that is opened only by the intended recipient.

The RSA cryptosystem

In the ***RSA public-key cryptosystem***, a participant creates a public key and a secret key with the following procedure:

1. Select at random two large prime numbers p and q such that $p \neq q$. The primes p and q might be, say, 1024 bits each.
2. Compute $n = pq$.
3. Select a small odd integer e that is relatively prime to $\phi(n)$, which, by equation (31.21), equals $(p-1)(q-1)$.
4. Compute d as the multiplicative inverse of e, modulo $\phi(n)$. (Corollary 31.26 guarantees that d exists and is uniquely defined. You can use the technique of Section 31.4 to compute d, given e and $\phi(n)$.)
5. Publish the pair $P = (e, n)$ as the participant's ***RSA public key***.
6. Keep secret the pair $S = (d, n)$ as the participant's ***RSA secret key***.

For this scheme, the domain $\mathcal{D}$ is the set $\mathbb{Z}_n$. To transform a message M associated with a public key $P = (e, n)$, compute

$$P(M) = M^e \bmod n \ . \tag{31.37}$$

To transform a ciphertext C associated with a secret key $S = (d, n)$, compute

$$S(C) = C^d \bmod n \ . \tag{31.38}$$

These equations apply to both encryption and signatures. To create a signature, the signer's secret key is applied to the message to be signed, rather than to a ciphertext. To verify a signature, the public key of the signer is applied to the signature rather than to a message to be encrypted.

To implement the public-key and secret-key operations (31.37) and (31.38), you can use the procedure MODULAR-EXPONENTIATION described in Section 31.6. To analyze the running time of these operations, assume that the public key (e, n) and secret key (d, n) satisfy $\lg e = O(1)$, $\lg d \le \beta$, and $\lg n \le \beta$. Then, applying a public key requires $O(1)$ modular multiplications and uses $O(\beta^2)$ bit operations. Applying a secret key requires $O(\beta)$ modular multiplications, using $O(\beta^3)$ bit operations.

Theorem 31.36 (Correctness of RSA)
The RSA equations (31.37) and (31.38) define inverse transformations of $\mathbb{Z}_n$ satisfying equations (31.35) and (31.36).

Proof From equations (31.37) and (31.38), we have that for any $M \in \mathbb{Z}_n$,

$$P(S(M)) = S(P(M)) = M^{ed} \pmod{n} \ .$$

Since e and d are multiplicative inverses modulo $\phi(n) = (p-1)(q-1)$,

$$ed = 1 + k(p-1)(q-1)$$

for some integer k. But then, if $M \ne 0 \pmod{p}$, we have

$$\begin{aligned} M^{ed} &= M(M^{p-1})^{k(q-1)} && \pmod{p} \\ &= M((M \bmod p)^{p-1})^{k(q-1)} && \pmod{p} \\ &= M(1)^{k(q-1)} && \pmod{p} \qquad \text{(by Theorem 31.31)} \\ &= M && \pmod{p} \ . \end{aligned}$$

Also, $M^{ed} = M \pmod{p}$ if $M = 0 \pmod{p}$. Thus,

$$M^{ed} = M \pmod{p}$$

for all M. Similarly,

$$M^{ed} = M \pmod{q}$$

for all M. Thus, by Corollary 31.29 to the Chinese remainder theorem,

$$M^{ed} = M \pmod{n}$$

for all M. ■

The security of the RSA cryptosystem rests in large part on the difficulty of factoring large integers. If an adversary can factor the modulus n in a public key, then the adversary can derive the secret key from the public key, using the knowledge of the factors p and q in the same way that the creator of the public key used them. Therefore, if factoring large integers is easy, then breaking the RSA cryptosystem is easy. The converse statement, that if factoring large integers is hard, then breaking RSA is hard, is unproven. After two decades of research, however, no easier method has been found to break the RSA public-key cryptosystem than to factor the modulus n. And factoring large integers is surprisingly difficult. By randomly selecting and multiplying together two 1024-bit primes, you can create a public key that cannot be "broken" in any feasible amount of time with current technology. In the absence of a fundamental breakthrough in the design of number-theoretic algorithms, and when implemented with care following recommended standards, the RSA cryptosystem is capable of providing a high degree of security in applications.

In order to achieve security with the RSA cryptosystem, however, you should use integers that are quite long—more than 1000 bits—to resist possible advances in the art of factoring. In 2021, RSA moduli are commonly in the range of 2048 to 4096 bits. To create moduli of such sizes, you must find large primes efficiently. Section 31.8 addresses this problem.

For efficiency, RSA is often used in a "hybrid" or "key-management" mode with fast cryptosystems that are not public-key cryptosystems. With such a ***symmetric-key*** system, the encryption and decryption keys are identical. If Alice wishes to send a long message M to Bob privately, she selects a random key K for the fast symmetric-key cryptosystem and encrypts M using K, obtaining ciphertext C, where C is as long as M, but K is quite short. Then she encrypts K using Bob's public RSA key. Since K is short, computing $P_B(K)$ is fast (much faster than computing $P_B(M)$). She then transmits $(C, P_B(K))$ to Bob, who decrypts $P_B(K)$ to obtain K and then uses K to decrypt C, obtaining M.

A similar hybrid approach creates digital signatures efficiently. This approach combines RSA with a public ***collision-resistant hash function*** h—a function that is easy to compute but for which it is computationally infeasible to find two messages M and M' such that $h(M) = h(M')$. The value $h(M)$ is a short (say, 256-bit) "fingerprint" of the message M. If Alice wishes to sign a message M, she first applies h to M to obtain the fingerprint $h(M)$, which she then encrypts with her secret key. She sends $(M, S_A(h(M)))$ to Bob as her signed version of M.

Bob can verify the signature by computing $h(M)$ and verifying that P_A applied to $S_A(h(M))$ as received equals $h(M)$. Because no one can create two messages with the same fingerprint, it is computationally infeasible to alter a signed message and preserve the validity of the signature.

One way to distribute public keys uses ***certificates***. For example, assume that there is a "trusted authority" T whose public key is known by everyone. Alice can obtain from T a signed message (her certificate) stating that "Alice's public key is P_A." This certificate is "self-authenticating" since everyone knows P_T. Alice can include her certificate with her signed messages, so that the recipient has Alice's public key immediately available in order to verify her signature. Because her key was signed by T, the recipient knows that Alice's key is really Alice's.

Exercises

31.7-1
Consider an RSA key set with $p = 11$, $q = 29$, $n = 319$, and $e = 3$. What value of d should be used in the secret key? What is the encryption of the message $M = 100$?

31.7-2
Prove that if Alice's public exponent e is 3 and an adversary obtains Alice's secret exponent d, where $0 < d < \phi(n)$, then the adversary can factor Alice's modulus n in time polynomial in the number of bits in n. (Although you are not asked to prove it, you might be interested to know that this result remains true even if the condition $e = 3$ is removed. See Miller [327].)

★ ***31.7-3***
Prove that RSA is multiplicative in the sense that

$$P_A(M_1)P_A(M_2) = P_A(M_1M_2) \pmod{n} .$$

Use this fact to prove that if an adversary had a procedure that could efficiently decrypt 1% of messages from $\mathbb{Z}_n$ encrypted with P_A, then the adversary could employ a probabilistic algorithm to decrypt every message encrypted with P_A with high probability.

★ 31.8 Primality testing

This section shows how to find large primes. We begin with a discussion of the density of primes, proceed to examine a plausible, but incomplete, approach to

primality testing, and then present an effective randomized primality test due to Miller and Rabin.

The density of prime numbers

Many applications, such as cryptography, call for finding large "random" primes. Fortunately, large primes are not too rare, so that it is feasible to test random integers of the appropriate size until you find one that is prime. The ***prime distribution function*** $\pi(n)$ specifies the number of primes that are less than or equal to n. For example, $\pi(10) = 4$, since there are 4 prime numbers less than or equal to 10, namely, 2, 3, 5, and 7. The prime number theorem gives a useful approximation to $\pi(n)$.

Theorem 31.37 (Prime number theorem)

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1 . \qquad \blacksquare$$

The approximation $n/\ln n$ gives reasonably accurate estimates of $\pi(n)$ even for small n. For example, it is off by less than 6% at $n = 10^9$, where $\pi(n) = 50{,}847{,}534$ and $n/\ln n \approx 48{,}254{,}942$. (To a number theorist, 10^9 is a small number.)

The process of randomly selecting an integer n and determining whether it is prime is really just a Bernoulli trial (see Section C.4). By the prime number theorem, the probability of a success—that is, the probability that n is prime—is approximately $1/\ln n$. The geometric distribution says how many trials must occur to obtain a success, and by equation (C.36) on page 1197, the expected number of trials is approximately $\ln n$. Thus, in order to find a prime that has the same length as n by testing integers chosen randomly near n, the expected number examined would be approximately $\ln n$. For example, the expectation is that finding a 1024-bit prime would require testing approximately $\ln 2^{1024} \approx 710$ randomly chosen 1024-bit numbers for primality. (Of course, to cut this figure in half, choose only odd integers.)

The remainder of this section shows how to determine whether a large odd integer n is prime. For notational convenience, we assume that n has the prime factorization

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} ,$$

where $r \geq 1$, $p_1, p_2, \ldots, p_r$ are the prime factors of n, and $e_1, e_2, \ldots, e_r$ are positive integers. The integer n is prime if and only if $r = 1$ and $e_1 = 1$.

One simple approach to the problem of testing for primality is ***trial division***: try dividing n by each integer $2, 3, 5, 7, 9, \ldots, \lfloor\sqrt{n}\rfloor$, skipping even integers greater

than 2. We can conclude that n is prime if and only if none of the trial divisors divides n. Assuming that each trial division takes constant time, the worst-case running time is $\Theta(\sqrt{n})$, which is exponential in the length of n. (Recall that if n is encoded in binary using β bits, then $\beta = \lceil \lg(n+1) \rceil$, and so $\sqrt{n} = \Theta(2^{\beta/2})$.) Thus, trial division works well only if n is very small or happens to have a small prime factor. When it works, trial division has the advantage that it not only determines whether n is prime or composite, it also determines one of n's prime factors if n is composite.

This section focuses on finding out whether a given number n is prime. If n is composite, we won't worry about finding its prime factorization. Computing the prime factorization of a number is computationally expensive. You might be surprised that it turns out to be much easier to ascertain whether a given number is prime than it is to determine the prime factorization of the number if it is not prime.

Pseudoprimality testing

We'll start with a method for primality testing that "almost works" and, in fact, is good enough for many practical applications. Later on, we'll refine this method to remove the small defect. Let $\mathbb{Z}_n^+$ denote the nonzero elements of $\mathbb{Z}_n$:

$$\mathbb{Z}_n^+ = \{1, 2, \ldots, n-1\} .$$

If n is prime, then $\mathbb{Z}_n^+ = \mathbb{Z}_n^*$.

We say that n is a ***base-a pseudoprime*** if n is composite and

$$a^{n-1} = 1 \pmod{n} . \tag{31.39}$$

Fermat's theorem (Theorem 31.31 on page 932) implies that if n is prime, then n satisfies equation (31.39) for every a in $\mathbb{Z}_n^+$. Thus, if there is any $a \in \mathbb{Z}_n^+$ such that n does *not* satisfy equation (31.39), then n is certainly composite. Surprisingly, the converse *almost* holds, so that this criterion forms an almost perfect test for primality. Instead of trying every value of $a \in \mathbb{Z}_n^+$, test to see whether n satisfies equation (31.39) for just $a = 2$. If not, then declare n to be composite by returning COMPOSITE. Otherwise, return PRIME, guessing that n is prime (when, in fact, all we know is that n is either prime or a base-2 pseudoprime).

The procedure PSEUDOPRIME on the next page pretends in this manner to check whether n is prime. It uses the procedure MODULAR-EXPONENTIATION from Section 31.6. It assumes that the input n is an odd integer greater than 2. This procedure can make errors, but only of one type. That is, if it says that n is composite, then it is always correct. If it says that n is prime, however, then it makes an error only if n is a base-2 pseudoprime.

How often does PSEUDOPRIME err? Surprisingly rarely. There are only 22 values of n less than 10,000 for which it errs, the first four of which are 341, 561, 645,

```
PSEUDOPRIME(n)
if MODULAR-EXPONENTIATION(2, n − 1, n) ≠ 1 (mod n)
    return COMPOSITE          // definitely
else return PRIME             // we hope!
```

and 1105. We won't prove it, but the probability that this program makes an error on a randomly chosen β-bit number goes to 0 as β approaches ∞. Using more precise estimates due to Pomerance [361] of the number of base-2 pseudoprimes of a given size, a randomly chosen 512-bit number that is called prime by PSEUDOPRIME has less than one chance in 10^{20} of being a base-2 pseudoprime, and a randomly chosen 1024-bit number that is called prime has less than one chance in 10^{41} of being a base-2 pseudoprime. Thus, if you are merely trying to find a large prime for some application, for all practical purposes you almost never go wrong by choosing large numbers at random until one of them causes PSEUDOPRIME to return PRIME. But when the numbers being tested for primality are not randomly chosen, you might need a better approach for testing primality. As we'll see, a little more cleverness, and some randomization, will yield a primality-testing method that works well on all inputs.

Since PSEUDOPRIME checks equation (31.39) for only $a = 2$, you might think that you could eliminate all the errors by simply checking equation (31.39) for a second base number, say $a = 3$. Better yet, you could check equation (31.39) for even more values of a. Unfortunately, even checking for several values of a does not eliminate all errors, because there exist composite integers n, known as ***Carmichael numbers***, that satisfy equation (31.39) for *all* $a \in \mathbb{Z}_n^*$. (The equation does fail when $\gcd(a, n) > 1$—that is, when $a \notin \mathbb{Z}_n^*$—but demonstrating that n is composite by finding such an a can be difficult if n has only large prime factors.) The first three Carmichael numbers are 561, 1105, and 1729. Carmichael numbers are extremely rare. For example, only 255 of them are less than 100,000,000. Exercise 31.8-2 helps explain why they are so rare.

Let's see how to improve the primality test so that Carmichael numbers won't fool it.

The Miller-Rabin randomized primality test

The Miller-Rabin primality test overcomes the problems of the simple procedure PSEUDOPRIME with two modifications:

- It tries several randomly chosen base values a instead of just one base value.
- While computing each modular exponentiation, it looks for a nontrivial square root of 1, modulo n, during the final set of squarings. If it finds one, it stops

and returns COMPOSITE. Corollary 31.35 from Section 31.6 justifies detecting composites in this manner.

The pseudocode for the Miller-Rabin primality test appears in the procedures MILLER-RABIN and WITNESS. The input $n > 2$ to MILLER-RABIN is the odd number to be tested for primality, and s is the number of randomly chosen base values from $\mathbb{Z}_n^+$ to be tried. The code uses the random-number generator RANDOM described on page 129: RANDOM$(2, n-2)$ returns a randomly chosen integer a satisfying $2 \le a \le n-2$. (This range of values avoids having $a = \pm 1 \pmod{n}$.) The call of the auxiliary procedure WITNESS(a, n) returns TRUE if and only if a is a "witness" to the compositeness of n—that is, if it is possible using a to prove (in a manner that we will see) that n is composite. The test WITNESS(a, n) is an extension of, but more effective than, the test in equation (31.39) that formed the basis for PSEUDOPRIME, using $a = 2$.

Let's first understand how WITNESS works, and then we'll see how the Miller-Rabin primality test uses it. Let $n - 1 = 2^t u$ where $t \ge 1$ and u is odd. That is, the binary representation of $n - 1$ is the binary representation of the odd integer u followed by exactly t zeros. Therefore, $a^{n-1} = (a^u)^{2^t} \pmod{n}$, so that one way to compute $a^{n-1} \bmod n$ is to first compute $a^u \bmod n$ and then square the result t times successively.

```
MILLER-RABIN(n, s)                    // n > 2 is odd
for j = 1 to s
    a = RANDOM(2, n − 2)
    if WITNESS(a, n)
        return COMPOSITE              // definitely
return PRIME                          // almost surely

WITNESS(a, n)
let t and u be such that t ≥ 1, u is odd, and n − 1 = 2^t u
x_0 = MODULAR-EXPONENTIATION(a, u, n)
for i = 1 to t
    x_i = x_{i−1}^2 mod n
    if x_i == 1 and x_{i−1} ≠ 1 and x_{i−1} ≠ n − 1
        return TRUE                   // found a nontrivial square root of 1
if x_t ≠ 1
    return TRUE                       // composite, as in PSEUDOPRIME
return FALSE
```

This pseudocode for WITNESS computes $a^{n-1} \bmod n$ by first computing the value $x_0 = a^u \bmod n$ in line 2 and then repeatedly squaring the result t times in the **for** loop of lines 3–6. By induction on i, the sequence $x_0, x_1, \ldots, x_t$ of values computed satisfies the equation $x_i = a^{2^i u} \pmod{n}$ for $i = 0, 1, \ldots, t$, so that in particular $x_t = a^{n-1} \pmod{n}$. After line 4 performs a squaring step, however, the loop will terminate early if lines 5–6 detect that a nontrivial square root of 1 has just been discovered. (We'll explain these tests shortly.) If so, the procedure stops and returns TRUE. Lines 7–8 return TRUE if the value computed for $x_t = a^{n-1} \pmod{n}$ is not equal to 1, just as the PSEUDOPRIME procedure returns COMPOSITE in this case. Line 9 returns FALSE if lines 6 or 8 have not returned TRUE.

The following lemma proves the correctness of WITNESS.

Lemma 31.38
If WITNESS(a, n) returns TRUE, then a proof that n is composite can be constructed using a as a witness.

Proof If WITNESS returns TRUE from line 8, it's because line 7 determined that $x_t = a^{n-1} \bmod n \neq 1$. If n is prime, however, Fermat's theorem (Theorem 31.31) says that $a^{n-1} = 1 \pmod{n}$ for all $a \in \mathbb{Z}_n^*$. Since $\mathbb{Z}_n^+ = \mathbb{Z}_n^*$ if n is prime, Fermat's theorem also says that $a^{n-1} = 1 \pmod{n}$ for all $a \in \mathbb{Z}_n^+$. Therefore, n cannot be prime, and the equation $a^{n-1} \bmod n \neq 1$ proves this fact.

If WITNESS returns TRUE from line 6, then it has discovered that x_{i-1} is a nontrivial square root of 1, modulo n, since we have that $x_{i-1} \neq \pm 1 \pmod{n}$ yet $x_i = x_{i-1}^2 = 1 \pmod{n}$. Corollary 31.35 on page 934 states that only if n is composite can there exist a nontrivial square root of 1, modulo n, so that demonstrating that x_{i-1} is a nontrivial square root of 1, modulo n proves that n is composite. ■

Thus, if the call WITNESS(a, n) returns TRUE, then n is surely composite, and the witness a, along with the reason that the procedure returns TRUE (did it return from line 6 or from line 8?), provides a proof that n is composite.

Let's explore an alternative view of the behavior of WITNESS as a function of the sequence $X = \langle x_0, x_1, \ldots, x_t \rangle$. We'll find this view useful later on, when we analyze the error rate of the Miller-Rabin primality test. Note that if $x_i = 1$ for some $0 \leq i < t$, WITNESS might not compute the rest of the sequence. If it were to do so, however, each value $x_{i+1}, x_{i+2}, \ldots, x_t$ would be 1, so we can consider these positions in the sequence X as being all 1s. There are four cases:

1. $X = \langle \ldots, d \rangle$, where $d \neq 1$: the sequence X does not end in 1. Return TRUE in line 8, since a is a witness to the compositeness of n (by Fermat's Theorem).

2. $X = \langle 1, 1, \ldots, 1 \rangle$: the sequence X is all 1s. Return FALSE, since a is not a witness to the compositeness of n.
3. $X = \langle \ldots, -1, 1, \ldots, 1 \rangle$: the sequence X ends in 1, and the last non-1 is equal to -1. Return FALSE, since a is not a witness to the compositeness of n.
4. $X = \langle \ldots, d, 1, \ldots, 1 \rangle$, where $d \neq \pm 1$: the sequence X ends in 1, but the last non-1 is not -1. Return TRUE in line 6: a is a witness to the compositeness of n, since d is a nontrivial square root of 1.

Now, let's examine the Miller-Rabin primality test based on how it uses the WITNESS procedure. As before, assume that n is an odd integer greater than 2.

The procedure MILLER-RABIN is a probabilistic search for a proof that n is composite. The main loop (beginning on line 1) picks up to s random values of a from $\mathbb{Z}_n^+$, except for 1 and $n - 1$ (line 2). If it picks a value of a that is a witness to the compositeness of n, then MILLER-RABIN returns COMPOSITE on line 4. Such a result is always correct, by the correctness of WITNESS. If MILLER-RABIN finds no witness in s trials, then the procedure assumes that it found no witness because no witnesses exist, and therefore it assumes that n is prime. We'll see that this result is likely to be correct if s is large enough, but there is still a tiny chance that the procedure could be unlucky in its choice of s random values of a, so that even though the procedure failed to find a witness, at least one witness exists.

To illustrate the operation of MILLER-RABIN, let n be the Carmichael number 561, so that $n - 1 = 560 = 2^4 \cdot 35$, $t = 4$, and $u = 35$. If the procedure chooses $a = 7$ as a base, the column for $b = 35$ in Figure 31.4 (Section 31.6) shows that WITNESS computes $x_0 = a^{35} = 241 \pmod{561}$. Because of how the MODULAR-EXPONENTIATION procedure operates recursively on its parameter b, the first four columns in Figure 31.4 represent the factor 2^4 of 560—the rightmost four zeros in the binary representation of 560—reading these four zeros from right to left in the binary representation. Thus WITNESS computes the sequence $X = \langle 241, 298, 166, 67, 1 \quad \rangle$. Then, in the last squaring step, WITNESS discovers that a^{280} is a nontrivial square root of 1 since $a^{280} = 67 \pmod{n}$ and $(a^{280})^2 = a^{560} = 1 \pmod{n}$. Therefore, $a = 7$ is a witness to the compositeness of n, WITNESS$(7, n)$ returns TRUE, and MILLER-RABIN returns COMPOSITE.

If n is a β-bit number, MILLER-RABIN requires $O(s\beta)$ arithmetic operations and $O(s\beta^3)$ bit operations, since it requires asymptotically no more work than s modular exponentiations.

Error rate of the Miller-Rabin primality test

If MILLER-RABIN returns PRIME, then there is a very slim chance that it has made an error. Unlike PSEUDOPRIME, however, the chance of error does not depend on n: there are no bad inputs for this procedure. Rather, it depends on the size of s

and the "luck of the draw" in choosing base values a. Moreover, since each test is more stringent than a simple check of equation (31.39), we can expect on general principles that the error rate should be small for randomly chosen integers n. The following theorem presents a more precise argument.

Theorem 31.39
If n is an odd composite number, then the number of witnesses to the compositeness of n is at least $(n-1)/2$.

Proof The proof shows that the number of nonwitnesses is at most $(n-1)/2$, which implies the theorem.

We start by claiming that any nonwitness must be a member of $\mathbb{Z}_n^*$. Why? Consider any nonwitness a. It must satisfy $a^{n-1} = 1 \pmod{n}$ or, equivalently, $a \cdot a^{n-2} = 1 \pmod{n}$. Thus the equation $ax = 1 \pmod{n}$ has a solution, namely a^{n-2}. By Corollary 31.21 on page 924, $\gcd(a, n) \mid 1$, which in turn implies that $\gcd(a, n) = 1$. Therefore, a is a member of $\mathbb{Z}_n^*$, and all nonwitnesses belong to $\mathbb{Z}_n^*$.

To complete the proof, we show that not only are all nonwitnesses contained in $\mathbb{Z}_n^*$, they are all contained in a proper subgroup B of $\mathbb{Z}_n^*$ (recall that B is a *proper* subgroup of $\mathbb{Z}_n^*$ when B is subgroup of $\mathbb{Z}_n^*$ but B is not equal to $\mathbb{Z}_n^*$). By Corollary 31.16 on page 921, we then have $|B| \le |\mathbb{Z}_n^*|/2$. Since $|\mathbb{Z}_n^*| \le n-1$, we obtain $|B| \le (n-1)/2$. Therefore, if all nonwitnesses are contained in a proper subgroup of $\mathbb{Z}_n^*$, then the number of nonwitnesses is at most $(n-1)/2$, so that the number of witnesses must be at least $(n-1)/2$.

To find a proper subgroup B of $\mathbb{Z}_n^*$ containing all of the nonwitnesses, we consider two cases.

Case 1: There exists an $x \in \mathbb{Z}_n^*$ such that

$$x^{n-1} \ne 1 \pmod{n} .$$

In other words, n is not a Carmichael number. Since, as noted earlier, Carmichael numbers are extremely rare, case 1 is the more typical case (e.g., when n has been chosen randomly and is being tested for primality).

Let $B = \{b \in \mathbb{Z}_n^* : b^{n-1} = 1 \pmod{n}\}$. The set B must be nonempty, since $1 \in B$. The set B is closed under multiplication modulo n, and so B is a subgroup of $\mathbb{Z}_n^*$ by Theorem 31.14. Every nonwitness belongs to B, since a nonwitness a satisfies $a^{n-1} = 1 \pmod{n}$. Since $x \in \mathbb{Z}_n^* - B$, we have that B is a proper subgroup of $\mathbb{Z}_n^*$.

Case 2: For all $x \in \mathbb{Z}_n^*$,

$$x^{n-1} = 1 \pmod{n} . \tag{31.40}$$

In other words, n is a Carmichael number. This case is extremely rare in practice. Unlike a pseudoprimality test, however, the Miller-Rabin test can efficiently determine that Carmichael numbers are composite, as we're about to see.

In this case, n cannot be a prime power. To see why, suppose to the contrary that $n = p^e$, where p is a prime and $e > 1$. We derive a contradiction as follows. Since we assume that n is odd, p must also be odd. Theorem 31.32 on page 933 implies that $\mathbb{Z}_n^*$ is a cyclic group: it contains a generator g such that $\text{ord}_n(g) = |\mathbb{Z}_n^*| = \phi(n) = p^e(1 - 1/p) = (p-1)p^{e-1}$. (The formula for $\phi(n)$ comes from equation (31.21) on page 920.) By equation (31.40), we have $g^{n-1} = 1 \pmod{n}$. Then the discrete logarithm theorem (Theorem 31.33 on page 933, taking $y = 0$) implies that $n - 1 = 0 \pmod{\phi(n)}$, or

$$(p-1)p^{e-1} \mid p^e - 1 .$$

This statement is a contradiction for $e > 1$, since $(p-1)p^{e-1}$ is divisible by the prime p, but $p^e - 1$ is not. Thus n is not a prime power.

Since the odd composite number n is not a prime power, we decompose it into a product $n_1 n_2$, where n_1 and n_2 are odd numbers greater than 1 that are relatively prime to each other. (There may be several ways to decompose n, and it does not matter which one we choose. For example, if $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, then we can choose $n_1 = p_1^{e_1}$ and $n_2 = p_2^{e_2} p_3^{e_3} \cdots p_r^{e_r}$.)

Recall that t and u are such that $n - 1 = 2^t u$, where $t \geq 1$ and u is odd, and that for an input a, the procedure WITNESS computes the sequence

$$X = \langle a^u, a^{2u}, a^{2^2 u}, \ldots, a^{2^t u} \rangle$$

where all computations are performed modulo n.

Let us call a pair (v, j) of integers ***acceptable*** if $v \in \mathbb{Z}_n^*$, $j \in \{0, 1, \ldots, t\}$, and

$$v^{2^j u} = -1 \pmod{n} .$$

Acceptable pairs certainly exist, since u is odd. Choose $v = n - 1$ and $j = 0$, and let $u = 2k + 1$, so that $v^{2^j u} = (n-1)^u = (n-1)^{2k+1}$. Taking this number modulo n gives $(n-1)^{2k+1} = (n-1)^{2k} \cdot (n-1) = (-1)^{2k} \cdot -1 = -1 \pmod{n}$. Thus, $(n-1, 0)$ is an acceptable pair. Now pick the largest possible j such that there exists an acceptable pair (v, j), and fix v so that (v, j) is an acceptable pair. Let

$$B = \{x \in \mathbb{Z}_n^* : x^{2^j u} = \pm 1 \pmod{n}\} .$$

Since B is closed under multiplication modulo n, it is a subgroup of $\mathbb{Z}_n^*$. By Theorem 31.15 on page 921, therefore, $|B|$ divides $|\mathbb{Z}_n^*|$. Every nonwitness must be a member of B, since the sequence X produced by a nonwitness must either be all 1s or else contain a -1 no later than the jth position, by the maximality of j.

(If (a, j') is acceptable, where a is a nonwitness, we must have $j' \le j$ by how we chose j.)

We now use the existence of v to demonstrate that there exists a $w \in \mathbb{Z}_n^* - B$, and hence that B is a proper subgroup of $\mathbb{Z}_n^*$. Since $v^{2^j u} = -1 \pmod{n}$, we also have $v^{2^j u} = -1 \pmod{n_1}$ by Corollary 31.29 to the Chinese remainder theorem. By Corollary 31.28, there exists a w simultaneously satisfying the equations

$$\begin{aligned} w &= v \pmod{n_1} , \\ w &= 1 \pmod{n_2} . \end{aligned}$$

Therefore,

$$\begin{aligned} w^{2^j u} &= -1 \pmod{n_1} , \\ w^{2^j u} &= 1 \pmod{n_2} . \end{aligned}$$

Corollary 31.29 gives that $w^{2^j u} \ne 1 \pmod{n_1}$ implies $w^{2^j u} \ne 1 \pmod{n}$ and also that $w^{2^j u} \ne -1 \pmod{n_2}$ implies $w^{2^j u} \ne -1 \pmod{n}$. Hence, we conclude that $w^{2^j u} \ne \pm 1 \pmod{n}$, and so $w \notin B$.

It remains to show that $w \in \mathbb{Z}_n^*$. We start by working separately modulo n_1 and modulo n_2. Working modulo n_1, since $v \in \mathbb{Z}_n^*$, we have that $\gcd(v, n) = 1$. Also, we have $\gcd(v, n_1) = 1$, since if v does not have any common divisors with n, then it certainly does not have any common divisors with n_1. Since $w = v \pmod{n_1}$, we see that $\gcd(w, n_1) = 1$. Working modulo n_2, we have $w = 1 \pmod{n_2}$ implies $\gcd(w, n_2) = 1$ by Exercise 31.2-3. Since $\gcd(w, n_1) = 1$ and $\gcd(w, n_2) = 1$, Theorem 31.6 on page 908 yields $\gcd(w, n_1 n_2) = \gcd(w, n) = 1$. That is, $w \in \mathbb{Z}_n^*$.

Therefore, we have $w \in \mathbb{Z}_n^* - B$, and we can conclude in case 2 that B, which includes all nonwitnesses, is a proper subgroup of $\mathbb{Z}_n^*$ and therefore has size at most $(n-1)/2$.

In either case, the number of witnesses to the compositeness of n is at least $(n-1)/2$. ■

Theorem 31.40

For any odd integer $n > 2$ and positive integer s, the probability that MILLER-RABIN(n, s) errs is at most 2^{-s}.

Proof By Theorem 31.39, if n is composite, then each execution of the **for** loop of lines 1–4 of MILLER-RABIN has a probability of at least $1/2$ of discovering a witness to the compositeness of n. MILLER-RABIN makes an error only if it is so unlucky as to miss discovering a witness to the compositeness of n on each of the s iterations of the main loop. The probability of such a sequence of misses is at most 2^{-s}. ■

If n is prime, MILLER-RABIN always reports PRIME, and if n is composite, the chance that MILLER-RABIN reports PRIME is at most 2^{-s}.

When applying MILLER-RABIN to a large randomly chosen integer n, however, we need to consider as well the prior probability that n is prime, in order to correctly interpret MILLER-RABIN's result. Suppose that we fix a bit length β and choose at random an integer n of length β bits to be tested for primality, so that $\beta \approx \lg n \approx 1.443 \ln n$. Let A denote the event that n is prime. By the prime number theorem (Theorem 31.37), the probability that n is prime is approximately

$$\begin{aligned} \Pr\{A\} &\approx 1/\ln n \\ &\approx 1.443/\beta. \end{aligned}$$

Now let B denote the event that MILLER-RABIN returns PRIME. We have that $\Pr\left\{\overline{B} \mid A\right\} = 0$ (or equivalently, that $\Pr\{B \mid A\} = 1$) and $\Pr\left\{B \mid \overline{A}\right\} \le 2^{-s}$ (or equivalently, that $\Pr\left\{\overline{B} \mid \overline{A}\right\} > 1 - 2^{-s}$).

But what is $\Pr\{A \mid B\}$, the probability that n is prime, given that MILLER-RABIN has returned PRIME? By the alternate form of Bayes's theorem (equation (C.20) on page 1189) and approximating $\Pr\left\{B \mid \overline{A}\right\}$ by 2^{-s}, we have

$$\begin{aligned} \Pr\{A \mid B\} &= \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{A\}\Pr\{B \mid A\} + \Pr\left\{\overline{A}\right\}\Pr\left\{B \mid \overline{A}\right\}} \\ &\approx \frac{(1/\ln n)\cdot 1}{(1/\ln n)\cdot 1 + (1 - 1/\ln n)\cdot 2^{-s}} \\ &\approx \frac{1}{1 + 2^{-s}(\ln n - 1)} . \end{aligned}$$

This probability does not exceed $1/2$ until s exceeds $\lg(\ln n - 1)$. Intuitively, that many initial trials are needed just for the confidence derived from failing to find a witness to the compositeness of n to overcome the prior bias in favor of n being composite. For a number with $\beta = 1024$ bits, this initial testing requires about

$$\begin{aligned} \lg(\ln n - 1) &\approx \lg(\beta/1.443) \\ &\approx 9 \end{aligned}$$

trials. In any case, choosing $s = 50$ should suffice for almost any imaginable application.

In fact, the situation is much better. If you are trying to find large primes by applying MILLER-RABIN to large randomly chosen odd integers, then choosing a small value of s (say 3) is unlikely to lead to erroneous results, though we won't prove it here. The reason is that for a randomly chosen odd composite integer n, the expected number of nonwitnesses to the compositeness of n is likely to be considerably smaller than $(n-1)/2$.

If the integer n is not chosen randomly, however, the best that can be proven is that the number of nonwitnesses is at most $(n-1)/4$, using an improved version of Theorem 31.39. Furthermore, there do exist integers n for which the number of nonwitnesses is $(n-1)/4$.

Exercises

31.8-1
Prove that if an odd integer $n > 1$ is not a prime or a prime power, then there exists a nontrivial square root of 1, modulo n.

★ ***31.8-2***
It is possible to strengthen Euler's theorem (Theorem 31.30) slightly to the form

$$a^{\lambda(n)} = 1 \pmod{n} \quad \text{for all } a \in \mathbb{Z}_n^* ,$$

where $n = p_1^{e_1} \cdots p_r^{e_r}$ and $\lambda(n)$ is defined by

$$\lambda(n) = \operatorname{lcm}(\phi(p_1^{e_1}), \ldots, \phi(p_r^{e_r})) .$$

Prove that $\lambda(n) \mid \phi(n)$. A composite number n is a Carmichael number if $\lambda(n) \mid n-1$. The smallest Carmichael number is $561 = 3 \cdot 11 \cdot 17$, for which $\lambda(n) = \operatorname{lcm}(2, 10, 16) = 80$, which divides 560. Prove that Carmichael numbers must be both "square-free" (not divisible by the square of any prime) and the product of at least three primes. (For this reason, they are not common.)

31.8-3
Prove that if x is a nontrivial square root of 1, modulo n, then $\gcd(x-1, n)$ and $\gcd(x+1, n)$ are both nontrivial divisors of n.

Problems

31-1 Binary gcd algorithm
Most computers can perform the operations of subtraction, testing the parity (odd or even) of a binary integer, and halving more quickly than computing remainders. This problem investigates the ***binary gcd algorithm***, which avoids the remainder computations used in Euclid's algorithm.

a. Prove that if a and b are both even, then $\gcd(a, b) = 2 \cdot \gcd(a/2, b/2)$.

b. Prove that if a is odd and b is even, then $\gcd(a, b) = \gcd(a, b/2)$.

c. Prove that if a and b are both odd, then $\gcd(a, b) = \gcd((a-b)/2, b)$.

d. Design an efficient binary gcd algorithm for input integers a and b, where $a \geq b$, that runs in $O(\lg a)$ time. Assume that each subtraction, parity test, and halving takes unit time.

31-2 Analysis of bit operations in Euclid's algorithm

a. Consider the ordinary "paper and pencil" algorithm for long division: dividing a by b, which yields a quotient q and remainder r. Show that this method requires $O((1 + \lg q) \lg b)$ bit operations.

b. Define $\mu(ab) = (1 + \lg a)(1 + \lg b)$. Show that the number of bit operations performed by EUCLID in reducing the problem of computing $\gcd(a, b)$ to that of computing $\gcd(b, a \bmod b)$ is at most $c(\mu(a, b) - \mu(b, a \bmod b))$ for some sufficiently large constant $c > 0$.

c. Show that EUCLID(a, b) requires $O(\mu(a, b))$ bit operations in general and $O(\beta^2)$ bit operations when applied to two β-bit inputs.

31-3 Three algorithms for Fibonacci numbers

This problem compares the efficiency of three methods for computing the nth Fibonacci number F_n, given n. Assume that the cost of adding, subtracting, or multiplying two numbers is $O(1)$, independent of the size of the numbers.

a. Show that the running time of the straightforward recursive method for computing F_n based on recurrence (3.31) on page 69 is exponential in n. (See, for example, the FIB procedure on page 751.)

b. Show how to compute F_n in $O(n)$ time using memoization.

c. Show how to compute F_n in $O(\lg n)$ time using only integer addition and multiplication. (*Hint:* Consider the matrix $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ and its powers.)

d. Assume now that adding two β-bit numbers takes $\Theta(\beta)$ time and that multiplying two β-bit numbers takes $\Theta(\beta^2)$ time. What is the running time of these three methods under this more reasonable cost measure for the elementary arithmetic operations?

31-4 Quadratic residues

Let p be an odd prime. A number $a \in Z_p^*$ is a ***quadratic residue*** modulo p, if the equation $x^2 = a \pmod{p}$ has a solution for the unknown x.

a. Show that there are exactly $(p - 1)/2$ quadratic residues, modulo p.

b. If p is prime, we define the ***Legendre symbol*** $\left(\frac{a}{p}\right)$, for $a \in \mathbb{Z}_p^*$, to be 1 if a is a quadratic residue, modulo p, and -1 otherwise. Prove that if $a \in \mathbb{Z}_p^*$, then

$$\left(\frac{a}{p}\right) = a^{(p-1)/2} \pmod{p} .$$

Give an efficient algorithm that determines whether a given number a is a quadratic residue, modulo p. Analyze the efficiency of your algorithm.

c. Prove that if p is a prime of the form $4k + 3$ and a is a quadratic residue in $\mathbb{Z}_p^*$, then $a^{k+1} \bmod p$ is a square root of a, modulo p. How much time is required to find the square root of a quadratic residue a, modulo p?

d. Describe an efficient randomized algorithm for finding a nonquadratic residue, modulo an arbitrary prime p, that is, a member of $\mathbb{Z}_p^*$ that is not a quadratic residue. How many arithmetic operations does your algorithm require on average?

Chapter notes

Knuth [260] contains a good discussion of algorithms for finding the greatest common divisor, as well as other basic number-theoretic algorithms. Dixon [121] gives an overview of factorization and primality testing. Bach [33], Riesel [378], and Bach and Shallit [34] provide overviews of the basics of computational number theory; Shoup [411] provides a more recent survey. The conference proceedings edited by Pomerance [362] contains several excellent survey articles.

Knuth [260] discusses the origin of Euclid's algorithm. It appears in Book 7, Propositions 1 and 2, of the Greek mathematician Euclid's *Elements*, which was written around 300 B.C.E. Euclid's description may have been derived from an algorithm due to Eudoxus around 375 B.C.E. Euclid's algorithm may hold the honor of being the oldest nontrivial algorithm, rivaled only by an algorithm for multiplication known to the ancient Egyptians. Shallit [407] chronicles the history of the analysis of Euclid's algorithm.

Knuth attributes a special case of the Chinese remainder theorem (Theorem 31.27) to the Chinese mathematician Sun-Tsŭ, who lived sometime between 200 B.C.E. and 200 C.E.—the date is quite uncertain. The same special case was given by the Greek mathematician Nichomachus around 100 C.E. It was generalized by Qin Jiushao in 1247. The Chinese remainder theorem was finally stated and proved in its full generality by L. Euler in 1734.

The randomized primality-testing algorithm presented here is due to Miller [327] and Rabin [373] and is the fastest randomized primality-testing algorithm known,

to within constant factors. The proof of Theorem 31.40 is a slight adaptation of one suggested by Bach [32]. A proof of a stronger result for MILLER-RABIN was given by Monier [332, 333]. For many years primality-testing was the classic example of a problem where randomization appeared to be necessary to obtain an efficient (polynomial-time) algorithm. In 2002, however, Agrawal, Kayal, and Saxena [4] surprised everyone with their deterministic polynomial-time primality-testing algorithm. Until then, the fastest deterministic primality testing algorithm known, due to Cohen and Lenstra [97], ran in $(\lg n)^{O(\lg \lg \lg n)}$ time on input n, which is just slightly superpolynomial. Nonetheless, for practical purposes, randomized primality-testing algorithms remain more efficient and are generally preferred.

Beauchemin, Brassard, Crépeau, Goutier, and Pomerance [40] nicely discuss the problem of finding large "random" primes.

The concept of a public-key cryptosystem is due to Diffie and Hellman [115]. The RSA cryptosystem was proposed in 1977 by Rivest, Shamir, and Adleman [380]. Since then, the field of cryptography has blossomed. Our understanding of the RSA cryptosystem has deepened, and modern implementations use significant refinements of the basic techniques presented here. In addition, many new techniques have been developed for proving cryptosystems to be secure. For example, Goldwasser and Micali [190] show that randomization can be an effective tool in the design of secure public-key encryption schemes. For signature schemes, Goldwasser, Micali, and Rivest [191] present a digital-signature scheme for which every conceivable type of forgery is provably as difficult as factoring. Katz and Lindell [253] provide an overview of modern cryptography.

The best algorithms for factoring large numbers have a running time that grows roughly exponentially with the cube root of the length of the number n to be factored. The general number-field sieve factoring algorithm (as developed by Buhler, Lenstra, and Pomerance [77] as an extension of the ideas in the number-field sieve factoring algorithm by Pollard [360] and Lenstra et al. [295] and refined by Coppersmith [102] and others) is perhaps the most efficient such algorithm in general for large inputs. Although it is difficult to give a rigorous analysis of this algorithm, under reasonable assumptions we can derive a running-time estimate of $L(1/3, n)^{1.902+o(1)}$, where $L(\alpha, n) = e^{(\ln n)^{\alpha}(\ln \ln n)^{1-\alpha}}$.

The elliptic-curve method due to Lenstra [296] may be more effective for some inputs than the number-field sieve method, since it can find a small prime factor p quite quickly. With this method, the time to find p is estimated to be $L(1/2, p)^{\sqrt{2}+o(1)}$.

32 String Matching

Text-editing programs frequently need to find all occurrences of a pattern in the text. Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user. Efficient algorithms for this problem —called "string matching"—can greatly aid the responsiveness of the text-editing program. Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences. Internet search engines also use them to find web pages relevant to queries.

The string-matching problem can be stated formally as follows. The text is given as an array $T[1:n]$ of length n, and the pattern is an array $P[1:m]$ of length $m \le n$. The elements of P and T are characters drawn from an alphabet Σ, which is a finite set of characters. For example, Σ could be the set $\{\mathtt{0}, \mathtt{1}\}$, or it could be the set $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. The character arrays P and T are often called ***strings*** of characters.

As Figure 32.1 shows, pattern P ***occurs with shift s*** in text T (or, equivalently, that pattern P ***occurs beginning at position s + 1*** in text T) if $0 \le s \le n - m$ and $T[s+1:s+m] = P[1:m]$, that is, if $T[s+j] = P[j]$, for $1 \le j \le m$. If P occurs with shift s in T, then s is a ***valid shift***, and otherwise, s is an ***invalid shift***. The ***string-matching problem*** is the problem of finding all valid shifts with which a given pattern P occurs in a given text T.

Figure 32.1 An example of the string-matching problem to find all occurrences of the pattern $P = \mathtt{abaa}$ in the text $T = \mathtt{abcabaabcabac}$. The pattern occurs only once in the text, at shift $s = 3$, which is a valid shift. A vertical line connects each character of the pattern to its matching character in the text, and all matched characters are shaded blue.

Except for the naive brute-force algorithm in Section 32.1, each string-matching algorithm in this chapter performs some preprocessing based on the pattern and then finds all valid shifts. We call this latter phase "matching." Here are the preprocessing and matching times for each of the string-matching algorithms in this chapter. The total running time of each algorithm is the sum of the preprocessing and matching times:

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite automaton	$O(m\,\lvert\Sigma\rvert)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$
Suffix array[1]	$O(n \lg n)$	$O(m \lg n + km)$

Section 32.2 presents an interesting string-matching algorithm, due to Rabin and Karp. Although the $\Theta((n-m+1)m)$ worst-case running time of this algorithm is no better than that of the naive method, it works much better on average and in practice. It also generalizes nicely to other pattern-matching problems. Section 32.3 then describes a string-matching algorithm that begins by constructing a finite automaton specifically designed to search for occurrences of the given pattern P in a text. This algorithm takes $O(m\,\lvert\Sigma\rvert)$ preprocessing time, but only $\Theta(n)$ matching time. Section 32.4 presents the similar, but much cleverer, Knuth-Morris-Pratt (or KMP) algorithm, which has the same $\Theta(n)$ matching time, but it reduces the preprocessing time to only $\Theta(m)$.

A completely different approach appears in Section 32.5, which examines suffix arrays and the longest common prefix array. You can use these arrays not only to find a pattern in a text, but also to answer other questions, such as what is the longest repeated substring in the text and what is the longest common substring between two texts. The algorithm to form the suffix array in Section 32.5 takes $O(n \lg n)$ time and, given the suffix array, the section shows how to compute the longest common prefix array in $O(n)$ time.

Notation and terminology

We denote by Σ^* (read "sigma-star") the set of all finite-length strings formed using characters from the alphabet Σ. This chapter considers only finite-length

[1] For suffix arrays, the preprocessing time of $O(n \lg n)$ comes from the algorithm presented in Section 32.5. It can be reduced to $\Theta(n)$ by using the algorithm in Problem 32-2. The factor k in the matching time denotes the number of occurrences of the pattern in the text.

Figure 32.2 A graphical proof of Lemma 32.1. Suppose that $x \sqsupset z$ and $y \sqsupset z$. The three parts of the figure illustrate the three cases of the lemma. Vertical lines connect matching regions (shown in blue) of the strings. **(a)** If $|x| \le |y|$, then $x \sqsupset y$. **(b)** If $|x| \ge |y|$, then $y \sqsupset x$. **(c)** If $|x| = |y|$, then $x = y$.

strings. The 0-length ***empty string***, denoted ε, also belongs to Σ^*. The length of a string x is denoted $|x|$. The ***concatenation*** of two strings x and y, denoted xy, has length $|x| + |y|$ and consists of the characters from x followed by the characters from y.

A string w is a ***prefix*** of a string x, denoted $w \sqsubset x$, if $x = wy$ for some string $y \in \Sigma^*$. Note that if $w \sqsubset x$, then $|w| \le |x|$. Similarly, a string w is a ***suffix*** of a string x, denoted $w \sqsupset x$, if $x = yw$ for some $y \in \Sigma^*$. As with a prefix, $w \sqsupset x$ implies $|w| \le |x|$. For example, $\texttt{ab} \sqsubset \texttt{abcca}$ and $\texttt{cca} \sqsupset \texttt{abcca}$. A string w is a ***proper prefix*** of x if $w \sqsubset x$ and $|w| < |x|$, and likewise for a ***proper suffix***. The empty string ε is both a suffix and a prefix of every string. For any strings x and y and any character a, we have $x \sqsupset y$ if and only if $xa \sqsupset ya$. The $\sqsubset$ and $\sqsupset$ relations are transitive. The following lemma will be useful later.

Lemma 32.1 (Overlapping-suffix lemma)
Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \le |y|$, then $x \sqsupset y$. If $|x| \ge |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.

Proof See Figure 32.2 for a graphical proof. ■

For convenience, denote the k-character prefix $P[1:k]$ of the pattern $P[1:m]$ by $P[:k]$. Thus, we can write $P[:0] = \varepsilon$ and $P[:m] = P = P[1:m]$. Similarly, denote the k-character prefix of the text T by $T[:k]$. Using this notation, we

can state the string-matching problem as that of finding all shifts s in the range $0 \le s \le n - m$ such that $P \sqsupset T[:s + m]$.

Our pseudocode allows two equal-length strings to be compared for equality as a primitive operation. If the strings are compared from left to right and the comparison stops when a mismatch is discovered, we assume that the time taken by such a test is a linear function of the number of matching characters discovered. To be precise, the test "$x == y$" is assumed to take $\Theta(t)$ time, where t is the length of the longest string z such that $z \sqsubset x$ and $z \sqsubset y$.

32.1 The naive string-matching algorithm

The NAIVE-STRING-MATCHER procedure finds all valid shifts using a loop that checks the condition $P[1:m] = T[s+1:s+m]$ for each of the $n-m+1$ possible values of s.

```
NAIVE-STRING-MATCHER(T, P, n, m)
1  for s = 0 to n − m
2      if P[1 : m] == T[s + 1 : s + m]
3          print "Pattern occurs with shift" s
```

Figure 32.3 portrays the naive string-matching procedure as sliding a "template" containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text. The **for** loop of lines 1–3 considers each possible shift explicitly. The test in line 2 determines whether the current shift is valid. This test implicitly loops to check corresponding character positions until all positions match successfully or a mismatch is found. Line 3 prints out each valid shift s.

Procedure NAIVE-STRING-MATCHER takes $O((n - m + 1)m)$ time, and this bound is tight in the worst case. For example, consider the text string $\mathtt{a}^n$ (a string of n a's) and the pattern $\mathtt{a}^m$. For each of the $n-m+1$ possible values of the shift s, the implicit loop on line 2 to compare corresponding characters must execute m times to validate the shift. The worst-case running time is thus $\Theta((n - m + 1)m)$, which is $\Theta(n^2)$ if $m = \lfloor n/2 \rfloor$. Because it requires no preprocessing, NAIVE-STRING-MATCHER's running time equals its matching time.

NAIVE-STRING-MATCHER is far from an optimal procedure for this problem. Indeed, this chapter will show that the Knuth-Morris-Pratt algorithm is much better in the worst case. The naive string-matcher is inefficient because it entirely ignores information gained about the text for one value of s when it considers other values of s. Such information can be quite valuable, however. For example, if $P = \mathtt{aaab}$

Figure 32.3 The operation of the NAIVE-STRING-MATCHER procedure for the pattern $P = \texttt{aab}$ and the text $T = \texttt{acaabc}$. Imagine the pattern P as a template that slides next to the text. **(a)–(d)** The four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown in blue), and a red jagged line connects the first mismatched character found, if any. The algorithm finds one occurrence of the pattern, at shift $s = 2$, shown in part (c).

and $s = 0$ is valid, then none of the shifts 1, 2, or 3 are valid, since $T[4] = \texttt{b}$. The following sections examine several ways to make effective use of this sort of information.

Exercises

32.1-1

Show the comparisons the naive string matcher makes for the pattern $P = \texttt{0001}$ in the text $T = \texttt{000010001010001}$.

32.1-2

Suppose that all characters in the pattern P are different. Show how to accelerate NAIVE-STRING-MATCHER to run in $O(n)$ time on an n-character text T.

32.1-3

Suppose that pattern P and text T are *randomly* chosen strings of length m and n, respectively, from the d-ary alphabet $\Sigma_d = \{0, 1, \ldots, d-1\}$, where $d \geq 2$. Show that the *expected* number of character-to-character comparisons made by the implicit loop in line 2 of the naive algorithm is

$$(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1)$$

over all executions of this loop. (Assume that the naive algorithm stops comparing characters for a given shift once it finds a mismatch or matches the entire pattern.) Thus, for randomly chosen strings, the naive algorithm is quite efficient.

32.1-4

Suppose that the pattern P may contain occurrences of a ***gap character*** $\diamond$ that can match an *arbitrary* string of characters (even one of 0 length). For example, the pattern $\texttt{ab}\diamond\texttt{ba}\diamond\texttt{c}$ occurs in the text $\texttt{cabccbacbacab}$ as

$$\mathtt{c}\underbrace{\mathtt{ab}}_{\mathtt{ab}}\underbrace{\mathtt{cc}}_{\diamond}\underbrace{\mathtt{ba}}_{\mathtt{ba}}\underbrace{\mathtt{cba}}_{\diamond}\underbrace{\mathtt{c}}_{\mathtt{c}}\mathtt{ab}$$

and as

$$\mathtt{c}\underbrace{\mathtt{ab}}_{\mathtt{ab}}\underbrace{\mathtt{ccbac}}_{\diamond}\underbrace{\mathtt{ba}}_{\mathtt{ba}}\underbrace{\phantom{\mathtt{cba}}}_{\diamond}\underbrace{\mathtt{c}}_{\mathtt{c}}\mathtt{ab}.$$

The gap character may occur an arbitrary number of times in the pattern but not at all in the text. Give a polynomial-time algorithm to determine whether such a pattern P occurs in a given text T, and analyze the running time of your algorithm.

32.2 The Rabin-Karp algorithm

Rabin and Karp proposed a string-matching algorithm that performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching. The Rabin-Karp algorithm uses $\Theta(m)$ preprocessing time, and its worst-case running time is $\Theta((n-m+1)m)$. Based on certain assumptions, however, its average-case running time is better.

This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number. You might want to refer to Section 31.1 for the relevant definitions.

For expository purposes, let's assume that $\Sigma = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots, \mathtt{9}\}$, so that each character is a decimal digit. (In the general case, you can assume that each character is a digit in radix-d notation, so that it has a numerical value in the range 0 to $d-1$, where $d = |\Sigma|$.) You can then view a string of k consecutive characters as representing a length-k decimal number. For example, the character string `31415` corresponds to the decimal number 31,415. Because we interpret the input characters as both graphical symbols and digits, it will be convenient in this section to denote them as digits in standard text font.

Given a pattern $P[1:m]$, let p denote its corresponding decimal value. In a similar manner, given a text $T[1:n]$, let t_s denote the decimal value of the length-m substring $T[s+1:s+m]$, for $s = 0, 1, \ldots, n-m$. Certainly, $t_s = p$ if and only if $T[s+1:s+m] = P[1:m]$, and thus, s is a valid shift if and only if $t_s = p$. If you could compute p in $\Theta(m)$ time and all the t_s values in a total of $\Theta(n-m+1)$ time,[2] then you could determine all valid shifts s in $\Theta(m) + \Theta(n-m+1) = \Theta(n)$

[2] We write $\Theta(n-m+1)$ instead of $\Theta(n-m)$ because s takes on $n-m+1$ different values. The "+1" is significant in an asymptotic sense because when $m = n$, computing the lone t_s value takes $\Theta(1)$ time, not $\Theta(0)$ time.

time by comparing p with each of the t_s values. (For the moment, let's not worry about the possibility that p and the t_s values might be very large numbers.)

Indeed, you can compute p in $\Theta(m)$ time using Horner's rule (see Problem 2-3):

$$p = P[m] + 10\Big(P[m-1] + 10\big(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots\big)\Big) .$$

Similarly, you can compute t_0 from $T[1:m]$ in $\Theta(m)$ time.

To compute the remaining values $t_1, t_2, \ldots, t_{n-m}$ in $\Theta(n-m)$ time, observe that you can compute t_{s+1} from t_s in constant time, since

$$t_{s+1} = 10\,(t_s - 10^{m-1}T[s+1]) + T[s+m+1] . \tag{32.1}$$

Subtracting $10^{m-1}T[s+1]$ removes the high-order digit from t_s, multiplying the result by 10 shifts the number left by one digit position, and adding $T[s+m+1]$ brings in the appropriate low-order digit. For example, suppose that $m = 5$, $t_s = 31415$, and the new low-order digit is $T[s+5+1] = 2$. The high-order digit to remove is $T[s+1] = 3$, and so

$$\begin{aligned} t_{s+1} &= 10\,(31415 - 10000 \cdot 3) + 2 \\ &= 14152 \end{aligned}$$

If you precompute the constant 10^{m-1} (which you can do in $O(\lg m)$ time using the techniques of Section 31.6, although for this application a straightforward $O(m)$-time method suffices), then each execution of equation (32.1) takes a constant number of arithmetic operations. Thus, you can compute p in $\Theta(m)$ time, and you can compute all of $t_0, t_1, \ldots, t_{n-m}$ in $\Theta(n-m+1)$ time. Therefore, you can find all occurrences of the pattern $P[1:m]$ in the text $T[1:n]$ with $\Theta(m)$ preprocessing time and $\Theta(n-m+1)$ matching time.

This scheme works well if P is short enough and the alphabet Σ is small enough that arithmetic operations on p and t_s take constant time. But what if P is long, or if the size of Σ means that instead of powers of 10 in equation (32.1) you have to use powers of a larger number (such as powers of 256 for the extended ASCII character set)? Then the values of p and t_s might be too large to work with in constant time. Fortunately, this problem can be solved, as Figure 32.4 shows: compute p and the t_s values modulo a suitable modulus q. You can compute p modulo q in $\Theta(m)$ time and all the t_s values modulo q in $\Theta(n-m+1)$ time. With $|\Sigma| = 10$, if you choose the modulus q as a prime such that $10q$ just fits within one computer word, then you can perform all the necessary computations with single-precision arithmetic. In general, with a d-ary alphabet $\{0, 1, \ldots, d-1\}$, choose q so that dq fits within a computer word and adjust the recurrence equation (32.1) to work modulo q, so that it becomes

$$t_{s+1} = \big(d(t_s - T[s+1]h) + T[s+m+1]\big) \bmod q , \tag{32.2}$$

Figure 32.4 The Rabin-Karp algorithm. Each character is a decimal digit. Values are computed modulo 13. **(a)** A text string. A window of length 5 is shaded blue. The numerical value of the blue number, computed modulo 13, yields the value 7. **(b)** The same text string with values computed modulo 13 for each possible position of a length-5 window. Assuming the pattern $P = 31415$, look for windows whose value modulo 13 is 7, since $31415 = 7 \pmod{13}$. The algorithm finds two such windows, shaded blue in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern. The second window, beginning at text position 13, is a spurious hit. **(c)** How to compute the value for a window in constant time, given the value for the previous window. The first window has value 31415. Dropping the high-order digit 3, shifting left (multiplying by 10), and then adding in the low-order digit 2 gives the new value 14152. Because all computations are performed modulo 13, the value for the first window is 7, and the value for the new window is 8.

where $h = d^{m-1} \bmod q$ is the value of the digit "1" in the high-order position of an m-digit text window.

The solution of working modulo q is not perfect, however: $t_s = p \pmod q$ does not automatically mean that $t_s = p$. On the other hand, if $t_s \neq p \pmod q$, then you definitely know that $t_s \neq p$, so that shift s is invalid. Thus you can use the test $t_s = p \pmod q$ as a fast heuristic test to rule out invalid shifts. If $t_s = p \pmod q$—a ***hit***—then you need to test further to see whether s is really valid or you just have a ***spurious hit***. This additional test explicitly checks the condition $P[1:m] = T[s+1:s+m]$. If q is large enough, then you would hope that spurious hits occur infrequently enough that the cost of the extra checking is low.

The procedure RABIN-KARP-MATCHER on the next page makes these ideas precise. The inputs to the procedure are the text T, the pattern P, their lengths n and m, the radix d to use (which is typically taken to be $|\Sigma|$), and the prime q to use. The procedure works as follows. All characters are interpreted as radix-d digits. The subscripts on t are provided only for clarity: the procedure works correctly if all the subscripts are dropped. Line 1 initializes h to the value of the high-order digit position of an m-digit window. Lines 2–6 compute p as the value of $P[1:m] \bmod q$ and t_0 as the value of $T[1:m] \bmod q$. The **for** loop of lines 7–12 iterates through all possible shifts s, maintaining the following invariant:

Whenever line 8 is executed, $t_s = T[s+1:s+m] \bmod q$.

If a hit occurs because $p = t_s$ in line 8, then line 9 determines whether s is a valid shift or the hit was spurious via the test $P[1:m] == T[s+1:s+m]$. Line 10 prints out any valid shifts that are found. If $s < n - m$ (checked in line 11), then the **for** loop will iterate at least one more time, and so line 12 first executes to ensure that the loop invariant holds upon the next iteration. Line 12 computes the value of $t_{s+1} \bmod q$ from the value of $t_s \bmod q$ in constant time using equation (32.2) directly.

RABIN-KARP-MATCHER takes $\Theta(m)$ preprocessing time, and its matching time is $\Theta((n-m+1)m)$ in the worst case, since (like the naive string-matching algorithm) the Rabin-Karp algorithm explicitly verifies every valid shift. If $P = \mathtt{a}^m$ and $T = \mathtt{a}^n$, then verifying takes $\Theta((n-m+1)m)$ time, since each of the $n-m+1$ possible shifts is valid.

In many applications, you expect few valid shifts—perhaps some constant c of them. In such applications, the expected matching time of the algorithm is only $O((n-m+1)+cm) = O(n+m)$, plus the time required to process spurious hits. We can base a heuristic analysis on the assumption that reducing values modulo q acts like a random mapping from Σ^* to $\mathbb{Z}_q$. The expected number of spurious hits is then $O(n/q)$, because we can estimate the chance that an arbitrary t_s will be equivalent to p, modulo q, as $1/q$. Since there are $O(n)$ positions at which the

```
RABIN-KARP-MATCHER(T, P, n, m, d, q)
1  h = d^(m-1) mod q
2  p = 0
3  t_0 = 0
4  for i = 1 to m                          // preprocessing
5      p = (dp + P[i]) mod q
6      t_0 = (dt_0 + T[i]) mod q
7  for s = 0 to n - m                      // matching—try all possible shifts
8      if p == t_s                         // a hit?
9          if P[1 : m] == T[s + 1 : s + m] // valid shift?
10             print "Pattern occurs with shift" s
11     if s < n - m
12         t_(s+1) = (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

test of line 8 fails (actually, at most $n - m + 1$ positions) and checking each hit takes $O(m)$ time in line 9, the expected matching time taken by the Rabin-Karp algorithm is

$$O(n) + O(m(v + n/q)) ,$$

where v is the number of valid shifts. This running time is $O(n)$ if $v = O(1)$ and you choose $q \geq m$. That is, if the expected number of valid shifts is small ($O(1)$) and you choose the prime q to be larger than the length of the pattern, then you can expect the Rabin-Karp procedure to use only $O(n + m)$ matching time. Since $m \leq n$, this expected matching time is $O(n)$.

Exercises

32.2-1

Working modulo $q = 11$, how many spurious hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$ when looking for the pattern $P = 26$?

32.2-2

Describe how to extend the Rabin-Karp method to the problem of searching a text string for an occurrence of any one of a given set of k patterns. Start by assuming that all k patterns have the same length. Then generalize your solution to allow the patterns to have different lengths.

32.2-3

Show how to extend the Rabin-Karp method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters. (The pattern may be shifted vertically and horizontally, but it may not be rotated.)

32.2-4

Alice has a copy of a long n-bit file $A = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$, and Bob similarly has an n-bit file $B = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. Alice and Bob wish to know if their files are identical. To avoid transmitting all of A or B, they use the following fast probabilistic check. Together, they select a prime $q > 1000n$ and randomly select an integer x from $\{0, 1, \ldots, q-1\}$. Letting

$$A(x) = \left(\sum_{i=0}^{n-1} a_i x^i \right) \bmod q \quad \text{and} \quad B(x) = \left(\sum_{i=0}^{n-1} b_i x^i \right) \bmod q \ ,$$

Alice evaluates $A(x)$ and Bob evaluates $B(x)$. Prove that if $A \neq B$, there is at most one chance in 1000 that $A(x) = B(x)$, whereas if the two files are the same, $A(x)$ is necessarily the same as $B(x)$. (*Hint:* See Exercise 31.4-4.)

32.3 String matching with finite automata

Many string-matching algorithms build a finite automaton—a simple machine for processing information—that scans the text string T for all occurrences of the pattern P. This section presents a method for building such an automaton. These string-matching automata are efficient: they examine each text character *exactly once*, taking constant time per text character. The matching time used—after preprocessing the pattern to build the automaton—is therefore $\Theta(n)$. The time to build the automaton, however, can be large if Σ is large. Section 32.4 describes a clever way around this problem.

We begin this section with the definition of a finite automaton. We then examine a special string-matching automaton and show how to use it to find occurrences of a pattern in a text. Finally, we'll see how to construct the string-matching automaton for a given input pattern.

Finite automata

A ***finite automaton*** M, illustrated in Figure 32.5, is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of ***states***,
- $q_0 \in Q$ is the ***start state***,
- $A \subseteq Q$ is a distinguished set of ***accepting states***,
- Σ is a finite ***input alphabet***,
- δ is a function from $Q \times \Sigma$ into Q, called the ***transition function*** of M.

state	input a	input b
0	1	0
1	0	0

(a)

(b)

Figure 32.5 A simple two-state finite automaton with state set $Q = \{0, 1\}$, start state $q_0 = 0$, and input alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. **(a)** A tabular representation of the transition function δ. **(b)** An equivalent state-transition diagram. State 1, in orange, is the only accepting state. Directed edges represent transitions. For example, the edge from state 1 to state 0 labeled b indicates that $\delta(1, \mathtt{b}) = 0$. This automaton accepts those strings that end in an odd number of a's. More precisely, it accepts a string x if and only if $x = yz$, where $y = \varepsilon$ or y ends with a b, and $z = \mathtt{a}^k$, where k is odd. For example, on input abaaa, including the start state, this automaton enters the sequence of states $\langle 0, 1, 0, 1, 0, 1 \rangle$, and so it accepts this input. For input abbaa, it enters the sequence of states $\langle 0, 1, 0, 0, 1, 0 \rangle$, and so it rejects this input.

The finite automaton begins in state q_0 and reads the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves ("makes a transition") from state q to state $\delta(q, a)$. Whenever its current state q is a member of A, the machine M has ***accepted*** the string read so far. An input that is not accepted is ***rejected***.

A finite automaton M induces a function ϕ, called the ***final-state function***, from Σ^* to Q such that $\phi(w)$ is the state M ends up in after reading the string w. Thus, M accepts a string w if and only if $\phi(w) \in A$. We define the function ϕ recursively, using the transition function:

$$\begin{aligned} \phi(\varepsilon) &= q_0 , \\ \phi(wa) &= \delta(\phi(w), a) \quad \text{for } w \in \Sigma^*, a \in \Sigma . \end{aligned}$$

String-matching automata

For a given pattern P, a preprocessing step constructs a string-matching automaton specific to P. The automaton then searches the text string for occurrences of P. Figure 32.6 illustrates the automaton for the pattern $P = \mathtt{ababaca}$. From now on, let's assume that P is fixed, and for brevity, we won't bother to indicate the dependence upon P in our notation.

In order to specify the string-matching automaton corresponding to a given pattern $P[1:m]$, we first define an auxiliary function σ, called the ***suffix function*** corresponding to the pattern P. The function σ maps Σ^* to $\{0, 1, \ldots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is also a suffix of x:

$$\sigma(x) = \max\{k : P[:k] \sqsupset x\} . \tag{32.3}$$

(a)

state	input a	input b	input c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

Figure 32.6 **(a)** A state-transition diagram for the string-matching automaton that accepts all strings ending in the string `ababaca`. State 0 is the start state, and state 7 (in orange) is the only accepting state. The transition function δ is defined by equation (32.4), and a directed edge from state i to state j labeled a represents $\delta(i, a) = j$. The right-going edges forming the "spine" of the automaton, shown in blue, correspond to successful matches between pattern and input characters. Except for the edges from state 7 to states 1 and 2, the left-going edges correspond to mismatches. Some edges corresponding to mismatches are omitted: by convention, if a state i has no outgoing edge labeled a for some $a \in \Sigma$, then $\delta(i, a) = 0$. **(b)** The corresponding transition function δ, and the pattern string $P =$ `ababaca`. The entries corresponding to successful matches between pattern and input characters are shown in blue. **(c)** The operation of the automaton on the text $T =$ `abababacaba`. Under each text character $T[i]$ appears the state $\phi(T[:i])$ that the automaton is in after processing the prefix $T[:i]$. The substring of the pattern that occurs in the text is highlighted in blue. The automaton finds this one occurrence of the pattern, ending in position 9.

The suffix function σ is well defined since the empty string $P[:0] = \varepsilon$ is a suffix of every string. As examples, for the pattern $P =$ `ab`, we have $\sigma(\varepsilon) = 0$, $\sigma(\texttt{ccaca}) = 1$, and $\sigma(\texttt{ccab}) = 2$. For a pattern P of length m, we have $\sigma(x) = m$ if and only if $P \sqsupset x$. From the definition of the suffix function, $x \sqsupset y$ implies $\sigma(x) \le \sigma(y)$ (see Exercise 32.3-4).

We are now ready to define the string-matching automaton that corresponds to a given pattern $P[1:m]$:

- The state set Q is $\{0, 1, \ldots, m\}$. The start state q_0 is state 0, and state m is the only accepting state.
- The transition function δ is defined, for any state q and character a, by

$$\delta(q, a) = \sigma(P[:q]a) \; . \tag{32.4}$$

As the automaton consumes characters of the text T, it is trying to build a match of the pattern P against the most recently seen characters of T. At any time, the state number q gives the length of the longest prefix of P that matches the most recently seen text characters. Whenever the automaton reaches state m, the m most recently seen text characters match the first m characters of P. Since P has length m, reaching state m means that the m most recently seen text characters match the entire pattern, so that the automaton has found a match.

With this intuition behind the design of the automaton, here is the reasoning behind defining $\delta(q, a) = \sigma(P[:q]a)$. Suppose that the automaton is in state q after reading the first i characters of the text, that is, $q = \phi(T[:i])$. The intuitive idea then says that q also equals the length of the longest prefix of P that matches a suffix of $T[:i]$ or, equivalently, that $q = \sigma(T[:i])$. Thus, since $\phi(T[:i])$ and $\sigma(T[:i])$ both equal q, we will see (in Theorem 32.4 on page 973) that the automaton maintains the following invariant:

$$\phi(T[:i]) = \sigma(T[:i]) \; . \tag{32.5}$$

If the automaton is in state q and reads the next character $T[i+1] = a$, then the transition should lead to the state corresponding to the longest prefix of P that is a suffix of $T[:i]a$. That state is $\sigma(T[:i]a)$, and equation (32.5) gives $\phi(T[:i]a) = \sigma(T[:i]a)$. Because $P[:q]$ is the longest prefix of P that is a suffix of $T[:i]$, the longest prefix of P that is a suffix of $T[:i]a$ has length not only $\sigma(T[:i]a)$, but also $\sigma(P[:q]a)$, and so $\phi(T[:i]a) = \sigma(P[:q]a)$. (Lemma 32.3 on page 972 will prove that $\sigma(T[:i]a) = \sigma(P[:q]a)$.) Thus, when the automaton is in state q, the transition function δ on character a should take the automaton to state $\delta(q, a) = \delta(\phi(T[:i]), a) = \phi(T[:i]a) = \sigma(P[:q]a)$ (with the last equality following from equation (32.5)).

There are two cases to consider, depending on whether the next character continues to match the pattern. In the first case, $a = P[q+1]$, so that the character a continues to match the pattern. In this case, because $\delta(q, a) = q + 1$, the transition continues to go along the "spine" of the automaton (the blue edges in Figure 32.6(a)). In the second case, $a \neq P[q+1]$, so that a does not extend the match being built. In this case, we need to find the longest prefix of P that is also a suffix of $T[:i]a$, which will have length at most q. The preprocessing step matches the pattern against itself when creating the string-matching automaton, so that the transition function can quickly identify the longest such smaller prefix of P.

Let's look at an example. Consider state 5 in the string-matching automaton of Figure 32.6. In state 5, the five most recently read characters of T are `ababa`, the characters along the spine of the automaton that reach state 5. If the next character of T is `c`, then the most recently read characters of T are `ababac`, which is the prefix of P with length 6. The automaton should continue along the spine to state 6. This is the first case, in which the match continues, and $\delta(5, \mathtt{c}) = 6$. To illustrate the second case, suppose that in state 5, the next character of T is `b`, so the most recently read characters of T are `ababab`. Here, the longest prefix of P that matches the most recently read characters of T—that is, a suffix of the portion of T read so far—is `abab`, with length 4, so $\delta(5, \mathtt{b}) = 4$.

To clarify the operation of a string-matching automaton, the simple and efficient procedure FINITE-AUTOMATON-MATCHER simulates the behavior of such an automaton (represented by its transition function δ) in finding occurrences of a pattern P of length m in an input text $T[1:n]$. As for any string-matching automaton for a pattern of length m, the state set Q is $\{0, 1, \ldots, m\}$, the start state is 0, and the only accepting state is state m. From the simple loop structure of FINITE-AUTOMATON-MATCHER, you can see that its matching time on a text string of length n is $\Theta(n)$, assuming that each lookup of the transition function δ takes constant time. This matching time, however, does not include the preprocessing time required to compute the transition function. We address this problem later, after first proving that the procedure FINITE-AUTOMATON-MATCHER operates correctly.

```
FINITE-AUTOMATON-MATCHER(T, δ, n, m)
q = 0
for i = 1 to n
    q = δ(q, T[i])
    if q == m
        print "Pattern occurs with shift" i − m
```

Let's examine how the automaton operates on an input text $T[1:n]$. We will prove that the automaton is in state $\sigma(T[:i])$ after reading character $T[i]$. Since $\sigma(T[:i]) = m$ if and only if $P \sqsupset T[:i]$, the machine is in the accepting state m if and only if it has just read the pattern P. We start with two lemmas about the suffix function σ.

Lemma 32.2 (Suffix-function inequality)
For any string x and character a, we have $\sigma(xa) \le \sigma(x) + 1$.

Figure 32.7 An illustration for the proof of Lemma 32.2. The figure shows that $r \le \sigma(x) + 1$, where $r = \sigma(xa)$.

Figure 32.8 An illustration for the proof of Lemma 32.3. The figure shows that $r = \sigma(P[:q]a)$, where $q = \sigma(x)$ and $r = \sigma(xa)$.

Proof Referring to Figure 32.7, let $r = \sigma(xa)$. If $r = 0$, then the conclusion $\sigma(xa) = r \le \sigma(x)+1$ is trivially satisfied since $\sigma(x)$ is nonnegative. Now assume that $r > 0$. Then, $P[:r] \sqsupset xa$, by the definition of σ. Thus, $P[:r-1] \sqsupset x$, by dropping the a from both the end of $P[:r]$ and the end of xa. Therefore, $r - 1 \le \sigma(x)$, since $\sigma(x)$ is the largest k such that $P[:k] \sqsupset x$, and thus $\sigma(xa) = r \le \sigma(x) + 1$. ■

Lemma 32.3 (Suffix-function recursion lemma)
For any string x and character a, if $q = \sigma(x)$, then $\sigma(xa) = \sigma(P[:q]a)$.

Proof The definition of σ gives that $P[:q] \sqsupset x$. As Figure 32.8 shows, we also have $P[:q]a \sqsupset xa$. Let $r = \sigma(xa)$. Then $P[:r] \sqsupset xa$ and, by Lemma 32.2, $r \le q + 1$. Thus, we have $|P[:r]| = r \le q + 1 = |P[:q]a|$. Since $P[:q]a \sqsupset xa$, $P[:r] \sqsupset xa$, and $|P[:r]| \le |P[:q]a|$, Lemma 32.1 on page 959 implies that $P[:r] \sqsupset P[:q]a$. Therefore, $r \le \sigma(P[:q]a)$, that is, $\sigma(xa) \le \sigma(P[:q]a)$. But we also have $\sigma(P[:q]a) \le \sigma(xa)$, since $P[:q]a \sqsupset xa$. Thus, $\sigma(xa) = \sigma(P[:q]a)$. ■

We are now ready to prove the main theorem characterizing the behavior of a string-matching automaton on a given input text. As noted above, this theorem shows that the automaton is merely keeping track, at each step, of the longest prefix of the pattern that is a suffix of what has been read so far. In other words, the automaton maintains the invariant (32.5).

Theorem 32.4
If ϕ is the final-state function of a string-matching automaton for a given pattern P and $T[1:n]$ is an input text for the automaton, then

$$\phi(T[:i]) = \sigma(T[:i])$$

for $i = 0, 1, \ldots, n$.

Proof The proof is by induction on i. For $i = 0$, the theorem is trivially true, since $T[:0] = \varepsilon$. Thus, $\phi(T[:0]) = 0 = \sigma(T[:0])$.

Now assume that $\phi(T[:i]) = \sigma(T[:i])$. We will prove that $\phi(T[:i+1]) = \sigma(T[:i+1])$. Let q denote $\phi(T[:i])$, so that $q = \sigma(T[:i])$, and let a denote $T[i+1]$. Then,

$$\begin{aligned}
\phi(T[:i+1]) &= \phi(T[:i]a) && \text{(by the definitions of } T[:i+1] \text{ and } a\text{)} \\
&= \delta(\phi(T[:i]), a) && \text{(by the definition of } \phi\text{)} \\
&= \delta(q, a) && \text{(by the definition of } q\text{)} \\
&= \sigma(P[:q]a) && \text{(by the definition (32.4) of } \delta\text{)} \\
&= \sigma(T[:i]a) && \text{(by Lemma 32.3)} \\
&= \sigma(T[:i+1]) && \text{(by the definition of } T[:i+1]\text{)} .
\end{aligned}$$

■

By Theorem 32.4, if the machine enters state q on line 3, then q is the largest value such that $P[:q] \sqsupset T[:i]$. Thus, in line 4, $q = m$ if and only if the machine has just read an occurrence of the pattern P. Therefore, FINITE-AUTOMATON-MATCHER operates correctly.

Computing the transition function

The procedure COMPUTE-TRANSITION-FUNCTION on the following page computes the transition function δ from a given pattern $P[1:m]$. It computes $\delta(q, a)$ in a straightforward manner according to its definition in equation (32.4). The nested loops beginning on lines 1 and 2 consider all states q and all characters a, and lines 3–6 set $\delta(q, a)$ to be the largest k such that $P[:k] \sqsupset P[:q]a$. The code starts with the largest conceivable value of k, which is $q+1$, unless $q = m$, in which case k cannot be larger than m. It then decreases k until $P[:k]$ is a suffix of $P[:q]a$, which must eventually occur, since $P[:0] = \varepsilon$ is a suffix of every string.

```
COMPUTE-TRANSITION-FUNCTION(P, Σ, m)
1  for q = 0 to m
2      for each character a ∈ Σ
3          k = min{m, q + 1}
4          while P[:k] is not a suffix of P[:q]a
5              k = k − 1
6          δ(q, a) = k
7  return δ
```

The running time of COMPUTE-TRANSITION-FUNCTION is $O(m^3 |\Sigma|)$, because the outer loops contribute a factor of $m|\Sigma|$, the inner **while** loop can run at most $m + 1$ times, and the test for whether $P[:k]$ is a suffix of $P[:q]a$ on line 4 can require comparing up to m characters. Much faster procedures exist. By utilizing some cleverly computed information about the pattern P (see Exercise 32.4-8), the time required to compute δ from P improves to $O(m|\Sigma|)$. This improved procedure for computing δ provides a way to find all occurrences of a length-m pattern in a length-n text over an alphabet Σ with $O(m|\Sigma|)$ preprocessing time and $\Theta(n)$ matching time.

Exercises

32.3-1

Draw a state-transition diagram for the string-matching automaton for the pattern $P = \texttt{aabab}$ over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$ and illustrate its operation on the text string $T = \texttt{aaababaabaababaab}$.

32.3-2

Draw a state-transition diagram for the string-matching automaton for the pattern $P = \texttt{ababbabbababbababbabb}$ over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$.

32.3-3

A pattern P is ***nonoverlappable*** if $P[:k] \sqsupset P[:q]$ implies $k = 0$ or $k = q$. Describe the state-transition diagram of the string-matching automaton for a nonoverlappable pattern.

32.3-4

Let x and y be prefixes of the pattern P. Prove that $x \sqsupset y$ implies $\sigma(x) \le \sigma(y)$.

★ ***32.3-5***
Given two patterns P and P', describe how to construct a finite automaton that determines all occurrences of *either* pattern. Try to minimize the number of states in your automaton.

32.3-6
Given a pattern P containing gap characters (see Exercise 32.1-4), show how to build a finite automaton that can find an occurrence of P in a text T in $O(n)$ matching time, where $n = |T|$.

★ 32.4 The Knuth-Morris-Pratt algorithm

Knuth, Morris, and Pratt developed a linear-time string matching algorithm that avoids computing the transition function δ altogether. Instead, the KMP algorithm uses an auxiliary function π, which it precomputes from the pattern in $\Theta(m)$ time and stores in an array $\pi[1:m]$. The array π allows the algorithm to compute the transition function δ efficiently (in an amortized sense) "on the fly" as needed. Loosely speaking, for any state $q = 0, 1, \ldots, m$ and any character $a \in \Sigma$, the value $\pi[q]$ contains the information needed to compute $\delta(q, a)$ but that does not depend on a. Since the array π has only m entries, whereas δ has $\Theta(m|\Sigma|)$ entries, the KMP algorithm saves a factor of $|\Sigma|$ in the preprocessing time by computing π rather than δ. Like the procedure FINITE-AUTOMATON-MATCHER, once preprocessing has completed, the KMP algorithm uses $\Theta(n)$ matching time.

The prefix function for a pattern

The prefix function π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. The KMP algorithm takes advantage of this information to avoid testing useless shifts in the naive pattern-matching algorithm and to avoid precomputing the full transition function δ for a string-matching automaton.

Consider the operation of the naive string matcher. Figure 32.9(a) shows a particular shift s of a template containing the pattern $P = \texttt{ababaca}$ against a text T. For this example, $q = 5$ of the characters have matched successfully, but the 6th pattern character fails to match the corresponding text character. The information that q characters have matched successfully determines the corresponding text characters. Because these q text characters match, certain shifts must be invalid. In the example of the figure, the shift $s + 1$ is necessarily invalid, since the first pattern character (a) would be aligned with a text character that does not match the first pattern character, but does match the second pattern character (b). The shift

$s' = s + 2$ shown in part (b) of the figure, however, aligns the first three pattern characters with three text characters that necessarily match.

More generally, suppose that you know that $P[:q] \sqsupset T[:s+q]$ or, equivalently, that $P[1:q] = T[s+1:s+q]$. You want to shift P so that some shorter prefix $P[:k]$ of P matches a suffix of $T[:s+q]$, if possible. You might have more than one choice for how much to shift, however. In Figure 32.9(b), shifting P by 2 positions works, so that $P[:3] \sqsupset T[:s+q]$, but so does shifting P by 4 positions, so that $P[:1] \sqsupset T[:s+q]$ in Figure 32.9(c). If more than one shift amount works, you should choose the smallest shift amount so that you do not miss any potential matches. Put more precisely, you want to answer this question:

> Given that pattern characters $P[1:q]$ match text characters $T[s+1:s+q]$ (that is, $P[:q] \sqsupset T[:s+q]$), what is the least shift $s' > s$ such that for some $k < q$,
>
> $$P[1:k] = T[s'+1:s'+k] \,, \tag{32.6}$$
>
> (that is, $P[:k] \sqsupset T[:s'+k]$), where $s'+k = s+q$?

Here's another way to look at this question. If you know $P[:q] \sqsupset T[:s+q]$, then how do you find the longest proper prefix $P[:k]$ of $P[:q]$ that is also a suffix of $T[:s+q]$? These questions are equivalent because given s and q, requiring $s'+k = s+q$ means that finding the smallest shift s' (2 in Figure 32.9(b)) is tantamount to finding the longest prefix length k (3 in Figure 32.9(b)). If you add the difference $q-k$ in the lengths of these prefixes of P to the shift s, you get the new shift s', so that $s' = s + (q-k)$. In the best case, $k = 0$, so that $s' = s+q$, immediately ruling out shifts $s+1, s+2, \ldots, s+q-1$. In any case, at the new shift s', it is redundant to compare the first k characters of P with the corresponding characters of T, since equation (32.6) guarantees that they match.

As Figure 32.9(d) demonstrates, you can precompute the necessary information by comparing the pattern against itself. Since $T[s'+1:s'+k]$ is part of the matched portion of the text, it is a suffix of the string $P[:q]$. Therefore, think of equation (32.6) as asking for the greatest $k < q$ such that $P[:k] \sqsupset P[:q]$. Then, the new shift $s' = s + (q-k)$ is the next potentially valid shift. It will be convenient to store, for each value of q, the number k of matching characters at the new shift s', rather than storing, say, the amount $s' - s$ to shift by.

Let's look at the precomputed information a little more formally. For a given pattern $P[1:m]$, the ***prefix function*** for P is the function $\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$ such that

$$\pi[q] = \max\{k : k < q \text{ and } P[:k] \sqsupset P[:q]\} \,.$$

That is, $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of $P[:q]$. Here is the complete prefix function π for the pattern `ababaca`:

Figure 32.9 The prefix function π. **(a)** The pattern $P = \texttt{ababaca}$ aligns with a text T so that the first $q = 5$ characters match. Matching characters, in blue, are connected by blue lines. **(b)** Knowing these particular 5 matched characters ($P[:5]$) suffices to deduce that a shift of $s + 1$ is invalid, but that a shift of $s' = s + 2$ is consistent with everything known about the text and therefore is potentially valid. The prefix $P[:k]$, where $k = 3$, aligns with the text seen so far. **(c)** A shift of $s + 4$ is also potentially valid, but it leaves only the prefix $P[:1]$ aligned with the text seen so far. **(d)** To precompute useful information for such deductions, compare the pattern with itself. Here, the longest prefix of P that is also a proper suffix of $P[:5]$ is $P[:3]$. The array π represents this precomputed information, so that $\pi[5] = 3$. Given that q characters have matched successfully at shift s, the next potentially valid shift is at $s' = s + (q - \pi[q])$ as shown in part (b).

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

The procedure KMP-MATCHER on the following page gives the Knuth-Morris-Pratt matching algorithm. The procedure follows from FINITE-AUTOMATON-MATCHER for the most part. To compute π, KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION. These two procedures have much in common, because both match a string against the pattern P: KMP-MATCHER matches the text T against P, and COMPUTE-PREFIX-FUNCTION matches P against itself.

Next, let's analyze the running times of these procedures. Then we'll prove them correct, which will be more complicated.

Running-time analysis

The running time of COMPUTE-PREFIX-FUNCTION is $\Theta(m)$, which we show by using the aggregate method of amortized analysis (see Section 16.1). The only tricky part is showing that the **while** loop of lines 5–6 executes $O(m)$ times alto-

```
KMP-MATCHER(T, P, n, m)
1  π = COMPUTE-PREFIX-FUNCTION(P, m)
2  q = 0                          // number of characters matched
3  for i = 1 to n                 // scan the text from left to right
4      while q > 0 and P[q + 1] ≠ T[i]
5          q = π[q]               // next character does not match
6      if P[q + 1] == T[i]
7          q = q + 1              // next character matches
8      if q == m                  // is all of P matched?
9          print "Pattern occurs with shift" i − m
10         q = π[q]               // look for the next match
```

```
COMPUTE-PREFIX-FUNCTION(P, m)
1  let π[1 : m] be a new array
2  π[1] = 0
3  k = 0
4  for q = 2 to m
5      while k > 0 and P[k + 1] ≠ P[q]
6          k = π[k]
7      if P[k + 1] == P[q]
8          k = k + 1
9      π[q] = k
10 return π
```

gether. Starting with some observations about k, we'll show that it makes at most $m-1$ iterations. First, line 3 starts k at 0, and the only way that k increases is by the increment operation in line 8, which executes at most once per iteration of the **for** loop of lines 4–9. Thus, the total increase in k is at most $m-1$. Second, since $k < q$ upon entering the **for** loop and each iteration of the loop increments q, we always have $k < q$. Therefore, the assignments in lines 2 and 9 ensure that $\pi[q] < q$ for all $q = 1, 2, \ldots, m$, which means that each iteration of the **while** loop decreases k. Third, k never becomes negative. Putting these facts together, we see that the total decrease in k from the **while** loop is bounded from above by the total increase in k over all iterations of the **for** loop, which is $m - 1$. Thus, the **while** loop iterates at most $m - 1$ times in all, and COMPUTE-PREFIX-FUNCTION runs in $\Theta(m)$ time.

Exercise 32.4-4 asks you to show, by a similar aggregate analysis, that the matching time of KMP-MATCHER is $\Theta(n)$.

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)

(b)

Figure 32.10 An illustration of Lemma 32.5 for the pattern $P = \texttt{ababaca}$ and $q = 5$. **(a)** The π function for the given pattern. Since $\pi[5] = 3$, $\pi[3] = 1$, and $\pi[1] = 0$, iterating π gives $\pi^*[5] = \{3, 1, 0\}$. **(b)** Sliding the template containing the pattern P to the right and noting when some prefix $P[:k]$ of P matches up with some proper suffix of $P[:5]$. Matches occur when $k = 3$, 1, and 0. In the figure, the first row gives P, and the vertical red line is drawn just after $P[:5]$. Successive rows show all the shifts of P that cause some prefix $P[:k]$ of P to match some suffix of $P[:5]$. Successfully matched characters are shown in blue. Blue lines connect aligned matching characters. Thus, $\{k : k < 5 \text{ and } P[:k] \sqsupset P[:5]\} = \{3, 1, 0\}$. Lemma 32.5 claims that $\pi^*[q] = \{k : k < q \text{ and } P[:k] \sqsupset P[:q]\}$ for all q.

Compared with FINITE-AUTOMATON-MATCHER, by using π rather than δ, the KMP algorithm reduces the time for preprocessing the pattern from $O(m|\Sigma|)$ to $\Theta(m)$, while keeping the actual matching time bounded by $\Theta(n)$.

Correctness of the prefix-function computation

We'll see a little later that the prefix function π helps to simulate the transition function δ in a string-matching automaton. But first, we need to prove that the procedure COMPUTE-PREFIX-FUNCTION does indeed compute the prefix function correctly. Doing so requires finding all prefixes $P[:k]$ that are proper suffixes of a given prefix $P[:q]$. The value of $\pi[q]$ gives us the length of the longest such prefix, but the following lemma, illustrated in Figure 32.10, shows that iterating the prefix function π generates all the prefixes $P[:k]$ that are proper suffixes of $P[:q]$. Let

$$\pi^*[q] = \left\{\pi[q], \pi^{(2)}[q], \pi^{(3)}[q], \ldots, \pi^{(t)}[q]\right\} ,$$

where $\pi^{(i)}[q]$ is defined in terms of functional iteration, so that $\pi^{(0)}[q] = q$ and $\pi^{(i)}[q] = \pi[\pi^{(i-1)}[q]]$ for $i \geq 1$ (so that $\pi[q] = \pi^{(1)}[q]$), and where the sequence in $\pi^*[q]$ stops upon reaching $\pi^{(t)}[q] = 0$ for some $t \geq 1$.

Lemma 32.5 (Prefix-function iteration lemma)
Let P be a pattern of length m with prefix function π. Then, for $q = 1, 2, \ldots, m$, we have $\pi^*[q] = \{k : k < q \text{ and } P[:k] \sqsupset P[:q]\}$.

Proof We first prove that $\pi^*[q] \subseteq \{k : k < q \text{ and } P[:k] \sqsupset P[:q]\}$ or, equivalently,

$$i \in \pi^*[q] \text{ implies } P[:i] \sqsupset P[:q] . \tag{32.7}$$

If $i \in \pi^*[q]$, then $i = \pi^{(u)}[q]$ for some $u > 0$. We prove equation (32.7) by induction on u. For $u = 1$, we have $i = \pi[q]$, and the claim follows since $i < q$ and $P[:\pi[q]] \sqsupset P[:q]$ by the definition of π. Now consider some $u \geq 1$ such that both $\pi^{(u)}[q]$ and $\pi^{(u+1)}[q]$ belong to $\pi^*[q]$. Let $i = \pi^{(u)}[q]$, so that $\pi[i] = \pi^{(u+1)}[q]$. The inductive hypothesis is that $P[:i] \sqsupset P[:q]$. Because the relations $<$ and $\sqsupset$ are transitive, we have $\pi[i] < i < q$ and $P[:\pi[i]] \sqsupset P[:i] \sqsupset P[:q]$, which establishes equation (32.7) for all i in $\pi^*[q]$. Therefore, $\pi^*[q] \subseteq \{k : k < q \text{ and } P[:k] \sqsupset P[:q]\}$.

We now prove that $\{k : k < q \text{ and } P[:k] \sqsupset P[:q]\} \subseteq \pi^*[q]$ by contradiction. Suppose to the contrary that the set $\{k : k < q \text{ and } P[:k] \sqsupset P[:q]\} - \pi^*[q]$ is nonempty, and let j be the largest number in the set. Because $\pi[q]$ is the largest value in $\{k : k < q \text{ and } P[:k] \sqsupset P[:q]\}$ and $\pi[q] \in \pi^*[q]$, it must be the case that $j < \pi[q]$. Having established that $\pi^*[q]$ contains at least one integer greater than j, let j' denote the smallest such integer. (We can choose $j' = \pi[q]$ if no other number in $\pi^*[q]$ is greater than j.) We have $P[:j] \sqsupset P[:q]$ because $j \in \{k : k < q \text{ and } P[:k] \sqsupset P[:q]\}$, and from $j' \in \pi^*[q]$ and equation (32.7), we have $P[:j'] \sqsupset P[:q]$. Thus, $P[:j] \sqsupset P[:j']$ by Lemma 32.1, and j is the largest value less than j' with this property. Therefore, we must have $\pi[j'] = j$ and, since $j' \in \pi^*[q]$, we must have $j \in \pi^*[q]$ as well. This contradiction proves the lemma. ■

The algorithm COMPUTE-PREFIX-FUNCTION computes $\pi[q]$, in order, for $q = 1, 2, \ldots, m$. Setting $\pi[1]$ to 0 in line 2 of COMPUTE-PREFIX-FUNCTION is certainly correct, since $\pi[q] < q$ for all q. We'll use the following lemma and its corollary to prove that COMPUTE-PREFIX-FUNCTION computes $\pi[q]$ correctly for $q > 1$.

Lemma 32.6
Let P be a pattern of length m, and let π be the prefix function for P. For $q = 1, 2, \ldots, m$, if $\pi[q] > 0$, then $\pi[q] - 1 \in \pi^*[q - 1]$.

Proof Let $r = \pi[q] > 0$, so that $r < q$ and $P[:r] \sqsupset P[:q]$, and thus, $r - 1 < q - 1$ and $P[:r - 1] \sqsupset P[:q - 1]$ (by dropping the last character from

$P[:r]$ and $P[:q]$, which we can do because $r > 0$). By Lemma 32.5, therefore, $r - 1 \in \pi^*[q-1]$. Thus, we have $\pi[q] - 1 = r - 1 \in \pi^*[q-1]$. ■

For $q = 2, 3, \ldots, m$, define the subset $E_{q-1} \subseteq \pi^*[q-1]$ by

$$\begin{aligned} E_{q-1} &= \{k \in \pi^*[q-1] : P[k+1] = P[q]\} \\ &= \{k : k < q-1 \text{ and } P[:k] \sqsupset P[:q-1] \text{ and } P[k+1] = P[q]\} \\ &\qquad\qquad \text{(by Lemma 32.5)} \\ &= \{k : k < q-1 \text{ and } P[:k+1] \sqsupset P[:q]\} \ . \end{aligned}$$

The set E_{q-1} consists of the values $k < q-1$ for which $P[:k] \sqsupset P[:q-1]$ and for which, because $P[k+1] = P[q]$, we have $P[:k+1] \sqsupset P[:q]$. Thus, E_{q-1} consists of those values $k \in \pi^*[q-1]$ such that extending $P[:k]$ to $P[:k+1]$ produces a proper suffix of $P[:q]$.

Corollary 32.7
Let P be a pattern of length m, and let π be the prefix function for P. Then, for $q = 2, 3, \ldots, m$,

$$\pi[q] = \begin{cases} 0 & \text{if } E_{q-1} = \emptyset \ , \\ 1 + \max E_{q-1} & \text{if } E_{q-1} \neq \emptyset \ . \end{cases}$$

Proof If E_{q-1} is empty, there is no $k \in \pi^*[q-1]$ (including $k = 0$) such that extending $P[:k]$ to $P[:k+1]$ produces a proper suffix of $P[:q]$. Therefore, $\pi[q] = 0$.

If, instead, E_{q-1} is nonempty, then for each $k \in E_{q-1}$, we have $k + 1 < q$ and $P[:k+1] \sqsupset P[:q]$. Therefore, the definition of $\pi[q]$ gives

$$\pi[q] \geq 1 + \max E_{q-1} \ . \tag{32.8}$$

Note that $\pi[q] > 0$. Let $r = \pi[q] - 1$, so that $r + 1 = \pi[q] > 0$, and therefore $P[:r+1] \sqsupset P[:q]$. If a nonempty string is a suffix of another, then the two strings must have the same last character. Since $r + 1 > 0$, the prefix $P[:r+1]$ is nonempty, and so $P[r+1] = P[q]$. Furthermore, $r \in \pi^*[q-1]$ by Lemma 32.6. Therefore, $r \in E_{q-1}$, and so $\pi[q] - 1 = r \leq \max E_{q-1}$ or, equivalently,

$$\pi[q] \leq 1 + \max E_{q-1} \ . \tag{32.9}$$

Combining equations (32.8) and (32.9) completes the proof. ■

We now finish the proof that COMPUTE-PREFIX-FUNCTION computes π correctly. The key is to combine the definition of E_{q-1} with the statement of Corollary 32.7, so that $\pi[q]$ equals 1 plus the greatest value of k in $\pi^*[q-1]$ such that

$P[k+1] = P[q]$. First, in COMPUTE-PREFIX-FUNCTION, $k = \pi[q-1]$ at the start of each iteration of the **for** loop of lines 4–9. This condition is enforced by lines 2 and 3 when the loop is first entered, and it remains true in each successive iteration because of line 9. Lines 5–8 adjust k so that it becomes the correct value of $\pi[q]$. The **while** loop of lines 5–6 searches through all values $k \in \pi^*[q-1]$ in decreasing order to find the value of $\pi[q]$. The loop terminates either because k reaches 0 or $P[k+1] = P[q]$. Because the "and" operator short-circuits, if the loop terminates because $P[k+1] = P[q]$, then k must have also been positive, and so k is the greatest value in E_{q-1}. In this case, lines 7–9 set $\pi[q]$ to $k+1$, according to Corollary 32.7. If, instead, the **while** loop terminates because $k = 0$, then there are two possibilities. If $P[1] = P[q]$, then $E_{q-1} = \{0\}$, and lines 7–9 set both k and $\pi[q]$ to 1. If $k = 0$ and $P[1] \neq P[q]$, however, then $E_{q-1} = \emptyset$. In this case, line 9 sets $\pi[q]$ to 0, again according to Corollary 32.7, which completes the proof of the correctness of COMPUTE-PREFIX-FUNCTION.

Correctness of the Knuth-Morris-Pratt algorithm

You can think of the procedure KMP-MATCHER as a reimplemented version of the procedure FINITE-AUTOMATON-MATCHER, but using the prefix function π to compute state transitions. Specifically, we'll prove that in the ith iteration of the **for** loops of both KMP-MATCHER and FINITE-AUTOMATON-MATCHER, the state q has the same value upon testing for equality with m (at line 8 in KMP-MATCHER and at line 4 in FINITE-AUTOMATON-MATCHER). Once we have argued that KMP-MATCHER simulates the behavior of FINITE-AUTOMATON-MATCHER, the correctness of KMP-MATCHER follows from the correctness of FINITE-AUTOMATON-MATCHER (though we'll see a little later why line 10 in KMP-MATCHER is necessary).

Before formally proving that KMP-MATCHER correctly simulates FINITE-AUTOMATON-MATCHER, let's take a moment to understand how the prefix function π replaces the δ transition function. Recall that when a string-matching automaton is in state q and it scans a character $a = T[i]$, it moves to a new state $\delta(q, a)$. If $a = P[q+1]$, so that a continues to match the pattern, then the state number is incremented: $\delta(q, a) = q + 1$. Otherwise, $a \neq P[q+1]$, so that a does not continue to match the pattern, and the state number does not increase: $0 \leq \delta(q, a) \leq q$. In the first case, when a continues to match, KMP-MATCHER moves to state $q + 1$ without referring to the π function: the **while** loop test in line 4 immediately comes up false, the test in line 6 comes up true, and line 7 increments q.

The π function comes into play when the character a does not continue to match the pattern, so that the new state $\delta(q, a)$ is either q or to the left of q along the spine of the automaton. The **while** loop of lines 4–5 in KMP-MATCHER iterates through

the states in $\pi^*[q]$, stopping either when it arrives in a state, say q', such that a matches $P[q'+1]$ or q' has gone all the way down to 0. If a matches $P[q'+1]$, then line 7 sets the new state to $q'+1$, which should equal $\delta(q,a)$ for the simulation to work correctly. In other words, the new state $\delta(q,a)$ should be either state 0 or a state numbered 1 more than some state in $\pi^*[q]$.

Let's look at the example in Figures 32.6 and 32.10, which are for the pattern $P = \texttt{ababaca}$. Suppose that the automaton is in state $q = 5$, having matched `ababa`. The states in $\pi^*[5]$ are, in descending order, 3, 1, and 0. If the next character scanned is `c`, then you can see that the automaton moves to state $\delta(5,\texttt{c}) = 6$ in both FINITE-AUTOMATON-MATCHER (line 3) and KMP-MATCHER (line 7). Now suppose that the next character scanned is instead `b`, so that the automaton should move to state $\delta(5,\texttt{b}) = 4$. The **while** loop in KMP-MATCHER exits after executing line 5 once, and the automaton arrives in state $q' = \pi[5] = 3$. Since $P[q'+1] = P[4] = \texttt{b}$, the test in line 6 comes up true, and the automaton moves to the new state $q'+1 = 4 = \delta(5,\texttt{b})$. Finally, suppose that the next character scanned is instead `a`, so that the automaton should move to state $\delta(5,\texttt{a}) = 1$. The first three times that the test in line 4 executes, the test comes up true. The first time finds that $P[6] = \texttt{c} \neq \texttt{a}$, and the automaton moves to state $\pi[5] = 3$ (the first state in $\pi^*[5]$). The second time finds that $P[4] = \texttt{b} \neq \texttt{a}$, and the automaton moves to state $\pi[3] = 1$ (the second state in $\pi^*[5]$). The third time finds that $P[2] = \texttt{b} \neq \texttt{a}$, and the automaton moves to state $\pi[1] = 0$ (the last state in $\pi^*[5]$). The **while** loop exits once it arrives in state $q' = 0$. Now line 6 finds that $P[q'+1] = P[1] = \texttt{a}$, and line 7 moves the automaton to the new state $q'+1 = 1 = \delta(5,\texttt{a})$.

Thus, the intuition is that KMP-MATCHER iterates through the states in $\pi^*[q]$ in decreasing order, stopping at some state q' and then possibly moving to state $q'+1$. Although that might seem like a lot of work just to simulate computing $\delta(q,a)$, bear in mind that asymptotically, KMP-MATCHER is no slower than FINITE-AUTOMATON-MATCHER.

We are now ready to formally prove the correctness of the Knuth-Morris-Pratt algorithm. By Theorem 32.4, we have that $q = \sigma(T[:i])$ after each time line 3 of FINITE-AUTOMATON-MATCHER executes. Therefore, it suffices to show that the same property holds with regard to the **for** loop in KMP-MATCHER. The proof proceeds by induction on the number of loop iterations. Initially, both procedures set q to 0 as they enter their respective **for** loops for the first time. Consider iteration i of the **for** loop in KMP-MATCHER. By the inductive hypothesis, the state number q equals $\sigma(T[:i-1])$ at the start of the loop iteration. We need to show that when line 8 is reached, the new value of q is $\sigma(T[:i])$. (Again, we'll handle line 10 separately.)

Considering q to be the state number at the start of the **for** loop iteration, when KMP-MATCHER considers the character $T[i]$, the longest prefix of P that is a suffix of $T[:i]$ is either $P[:q+1]$ (if $P[q+1] = T[i]$) or some prefix (not

necessarily proper, and possibly empty) of $P[:q]$. We consider separately the three cases in which $\sigma(T[:i]) = 0$, $\sigma(T[:i]) = q + 1$, and $0 < \sigma(T[:i]) \le q$.

- If $\sigma(T[:i]) = 0$, then $P[:0] = \varepsilon$ is the only prefix of P that is a suffix of $T[:i]$. The **while** loop of lines 4–5 iterates through each value q' in $\pi^*[q]$, but although $P[:q'] \sqsupset P[:q] \sqsupset T[:i-1]$ for every $q' \in \pi^*[q]$ (because $<$ are $\sqsupset$ are transitive relations), the loop never finds a q' such that $P[q'+1] = T[i]$. The loop terminates when q reaches 0, and of course line 7 does not execute. Therefore, $q = 0$ at line 8, so that now $q = \sigma(T[:i])$.
- If $\sigma(T[:i]) = q+1$, then $P[q+1] = T[i]$, and the **while** loop test in line 4 fails the first time through. Line 7 executes, incrementing the state number to $q + 1$, which equals $\sigma(T[:i])$.
- If $0 < \sigma(T[:i]) \le q'$, then the **while** loop of lines 4–5 iterates at least once, checking in decreasing order each value in $\pi^*[q]$ until it stops at some $q' < q$. Thus, $P[:q']$ is the longest prefix of $P[:q]$ for which $P[q'+1] = T[i]$, so that when the **while** loop terminates, $q' + 1 = \sigma(P[:q]T[i])$. Since $q = \sigma(T[:i-1])$, Lemma 32.3 implies that $\sigma(T[:i-1]T[i]) = \sigma(P[:q]T[i])$. Thus we have

  $$\begin{aligned} q' + 1 &= \sigma(P[:q]T[i]) \\ &= \sigma(T[:i-1]T[i]) \\ &= \sigma(T[:i]) \end{aligned}$$

  when the **while** loop terminates. After line 7 increments q, the new state number q equals $\sigma(T[:i])$.

Line 10 is necessary in KMP-MATCHER, because otherwise, line 4 might try to reference $P[m+1]$ after finding an occurrence of P. (The argument that $q = \sigma(T[:i-1])$ upon the next execution of line 4 remains valid by the hint given in Exercise 32.4-8: that $\delta(m, a) = \delta(\pi[m]a)$ or, equivalently, $\sigma(Pa) = \sigma(P[:\pi[m]]a)$ for any $a \in \Sigma$.) The remaining argument for the correctness of the Knuth-Morris-Pratt algorithm follows from the correctness of FINITE-AUTOMATON-MATCHER, since we have shown that KMP-MATCHER simulates the behavior of FINITE-AUTOMATON-MATCHER.

Exercises

32.4-1

Compute the prefix function π for the pattern `ababbabbabbababbabb`.

32.4-2

Give an upper bound on the size of $\pi^*[q]$ as a function of q. Give an example to show that your bound is tight.

32.4-3
Explain how to determine the occurrences of pattern P in the text T by examining the π function for the string PT (the string of length $m+n$ that is the concatenation of P and T).

32.4-4
Use an aggregate analysis to show that the running time of KMP-MATCHER is $\Theta(n)$.

32.4-5
Use a potential function to show that the running time of KMP-MATCHER is $\Theta(n)$.

32.4-6
Show how to improve KMP-MATCHER by replacing the occurrence of π in line 5 (but not line 10) by π', where π' is defined recursively for $q = 1, 2, \ldots, m-1$ by the equation

$$\pi'[q] = \begin{cases} 0 & \text{if } \pi[q] = 0 , \\ \pi'[\pi[q]] & \text{if } \pi[q] \neq 0 \text{ and } P[\pi[q]+1] = P[q+1] , \\ \pi[q] & \text{if } \pi[q] \neq 0 \text{ and } P[\pi[q]+1] \neq P[q+1] . \end{cases}$$

Explain why the modified algorithm is correct, and explain in what sense this change constitutes an improvement.

32.4-7
Give a linear-time algorithm to determine whether a text T is a cyclic rotation of another string T'. For example, `braze` and `zebra` are cyclic rotations of each other.

★ ***32.4-8***
Give an $O(m\,|\Sigma|)$-time algorithm for computing the transition function δ for the string-matching automaton corresponding to a given pattern P. (*Hint:* Prove that $\delta(q, a) = \delta(\pi[q]a)$ if $q = m$ or $P[q+1] \neq a$.)

32.5 Suffix arrays

The algorithms we have seen thus far in this chapter can efficiently find all occurrences of a pattern in a text. That is, however, all they can do. This section presents a different approach—suffix arrays—with which you can find all occurrences of a pattern in a text, but also quite a bit more. A suffix array won't find all occurrences

i	1	2	3	4	5	6	7
$T[i]$	r	a	t	a	t	a	t

i	$SA[i]$	$rank[i]$	$LCP[i]$	suffix $T[SA[i]:]$
1	6	4	0	at
2	4	3	2	atat
3	2	7	4	atatat
4	1	2	0	ratatat
5	7	6	0	t
6	5	1	1	tat
7	3	5	3	tatat

Figure 32.11 The suffix array SA, rank array $rank$, longest common prefix array LCP, and lexicographically sorted suffixes of the text $T =$ `ratatat` with length $n = 7$. The value of $rank[i]$ indicates the position of the suffix $T[i:]$ in the lexicographically sorted order: $rank[SA[i]] = i$ for $i = 1, 2, \ldots, n$. The $rank$ array is used to compute the LCP array.

of a pattern as quickly as, say, the Knuth-Morris-Pratt algorithm, but its additional flexibility makes it well worth studying.

A suffix array is simply a compact way to represent the lexicographically sorted order of all n suffixes of a length-n text. Given a text $T[1:n]$, let $T[i:]$ denote the suffix $T[i:n]$. The ***suffix array*** $SA[1:n]$ of T is defined such that if $SA[i] = j$, then $T[j:]$ is the ith suffix of T in lexicographic order.[3] That is, the ith suffix of T in lexicographic order is $T[SA[i]:]$. Along with the suffix array, another useful array is the ***longest common prefix array*** $LCP[1:n]$. The entry $LCP[i]$ gives the length of the longest common prefix between the ith and $(i-1)$st suffixes in the sorted order (with $LCP[SA[1]]$ defined to be 0, since there is no prefix lexicographically smaller than $T[SA[1]:]$). Figure 32.11 shows the suffix array and longest common prefix array for the 7-character text `ratatat`.

Given the suffix array for a text, you can search for a pattern via binary search on the suffix array. Each occurrence of a pattern in the text starts some suffix of the text, and because the suffix array is in lexicographically sorted order, all occurrences of a pattern will appear at the start of consecutive entries of the suffix array. For example, in Figure 32.11, the three occurrences of `at` in `ratatat` appear in entries 1 through 3 of the suffix array. If you find the length-m pattern in the length-n suffix array via binary search (taking $O(m \lg n)$ time because each comparison takes $O(m)$ time), then you can find all occurrences of the pattern in the text by searching backward and forward from that spot until you find a suffix that does not start with the pattern (or you go beyond the bounds of the suffix array). If the pattern occurs k times, then the time to find all k occurrences is $O(m \lg n + km)$.

[3] Informally, lexicographic order is "alphabetical order" in the underlying character set. A more precise definition of lexicographic order appears in Problem 12-2 on page 327.

With the longest common prefix array, you can find a longest repeated substring, that is, the longest substring that occurs more than once in the text. If $LCP[i]$ contains a maximum value in the LCP array, then a longest repeated substring appears in $T[SA[i] : SA[i] + LCP[i] - 1]$. In the example of Figure 32.11, the LCP array has one maximum value: $LCP[3] = 4$. Therefore, since $SA[3] = 2$, the longest repeated substring is $T[2:5] = \texttt{atat}$. Exercise 32.5-3 asks you to use the suffix array and longest common prefix array to find the longest common substrings between two texts. Next, we'll see how to compute the suffix array for an n-character text in $O(n \lg n)$ time and, given the suffix array and the text, how to compute the longest common prefix array in $\Theta(n)$ time.

Computing the suffix array

There are several algorithms to compute the suffix array of a length-n text. Some run in linear time, but are rather complicated. One such algorithm is given in Problem 32-2. Here we'll explore a simpler algorithm that runs in $\Theta(n \lg n)$ time.

The idea behind the $O(n \lg n)$-time procedure COMPUTE-SUFFIX-ARRAY on the following page is to lexicographically sort substrings of the text with increasing lengths. The procedure makes several passes over the text, with the substring length doubling each time. By the $\lceil \lg n \rceil$th pass, the procedure is sorting all the suffixes, thereby gaining the information needed to construct the suffix array. The key to attaining an $O(n \lg n)$-time algorithm will be to have each pass after the first sort in linear time, which will indeed be possible by using radix sort.

Let's start with a simple observation. Consider any two strings, s_1 and s_2. Decompose s_1 into s_1' and s_1'', so that s_1 is s_1' concatenated with s_1''. Likewise, let s_2 be s_2' concatenated with s_2''. Now, suppose that s_1' is lexicographically smaller than s_2'. Then, regardless of s_1'' and s_2'', it must be the case that s_1 is lexicographically smaller than s_2. For example, let $s_1 = \texttt{aaz}$ and $s_2 = \texttt{aba}$, and decompose s_1 into $s_1' = \texttt{aa}$ and $s_1'' = \texttt{z}$ and s_2 into $s_2' = \texttt{ab}$ and $s_2'' = \texttt{a}$. Because s_1' is lexicographically smaller than s_2', it follows that s_1 is lexicographically smaller than s_2, even though s_2'' is lexicographically smaller than s_1''.

Instead of comparing substrings directly, COMPUTE-SUFFIX-ARRAY represents substrings of the text with integer ***ranks***. Ranks have the simple property that one substring is lexicographically smaller then another if and only if it has a smaller rank. Identical substrings have equal ranks.

Where do these ranks come from? Initially, the substrings being considered are just single characters from the text. Assume that, as in many programming languages, there is a function, ord, that maps a character to its underlying encoding, which is a positive integer. The ord function could be the ASCII or Unicode encodings or any other function that produces a relative ordering of the characters. For example if all the characters are known to be lowercase letters, then $\text{ord}(\texttt{a}) = 1$,

```
COMPUTE-SUFFIX-ARRAY(T, n)
allocate arrays substr-rank[1 : n], rank[1 : n], and SA[1 : n]
for i = 1 to n
    substr-rank[i].left-rank = ord(T[i])
    if i < n
        substr-rank[i].right-rank = ord(T[i + 1])
    else substr-rank[i].right-rank = 0
    substr-rank[i].index = i
sort the array substr-rank into monotonically increasing order based
    on the left-rank attributes, using the right-rank attributes to break ties;
    if still a tie, the order does not matter
l = 2
while l < n
    MAKE-RANKS(substr-rank, rank, n)
    for i = 1 to n
        substr-rank[i].left-rank = rank[i]
        if i + l ≤ n
            substr-rank[i].right-rank = rank[i + l]
        else substr-rank[i].right-rank = 0
        substr-rank[i].index = i
    sort the array substr-rank into monotonically increasing order based
        on the left-rank attributes, using the right-rank attributes
        to break ties; if still a tie, the order does not matter
    l = 2l
for i = 1 to n
    SA[i] = substr-rank[i].index
return SA
```

```
MAKE-RANKS(substr-rank, rank, n)
r = 1
rank[substr-rank[1].index] = r
for i = 2 to n
    if substr-rank[i].left-rank ≠ substr-rank[i − 1].left-rank
            or substr-rank[i].right-rank ≠ substr-rank[i − 1].right-rank
        r = r + 1
    rank[substr-rank[i].index] = r
```

	After lines 2–7					After line 8			
i	*left-rank*	*right-rank*	*index*	substring	i	*left-rank*	*right-rank*	*index*	substring
1	114	97	1	`ra`	1	97	116	2	`at`
2	97	116	2	`at`	2	97	116	4	`at`
3	116	97	3	`ta`	3	97	116	6	`at`
4	97	116	4	`at`	4	114	97	1	`ra`
5	116	97	5	`ta`	5	116	0	7	`t`
6	97	116	6	`at`	6	116	97	3	`ta`
7	116	0	7	`t`	7	116	97	5	`ta`

Figure 32.12 The *substr-rank* array for indices $i = 1, 2, \ldots, 7$ after the **for** loop of lines 2–7 and after the sorting step in line 8 for input string $T =$ `ratatat`.

ord(`b`) $= 2, \ldots,$ ord(`z`) $= 26$ would work. Once the substrings being considered contain multiple characters, their ranks will be positive integers less than or equal to n, coming from their relative order after being sorted. An empty substring always has rank 0, since it is lexicographically less than any nonempty substring.

The COMPUTE-SUFFIX-ARRAY procedure uses objects internally to keep track of the relative ordering of the substrings according to their ranks. When considering substrings of a given length, the procedure creates and sorts an array *substr-rank*$[1:n]$ of n objects, each with the following attributes:

- *left-rank* contains the rank of the left part of the substring.
- *right-rank* contains the rank of the right part of the substring.
- *index* contains the index into the text T of where the substring starts.

Before delving into the details of how the procedure works, let's look at how it operates on the input text `ratatat`, with $n = 7$. Assuming that the ord function returns the ASCII code for a character, Figure 32.12 shows the *substr-rank* array after the **for** loop of lines 2–7 and then after the sorting step in line 8. The *left-rank* and *right-rank* values after lines 2–7 are the ranks of length-1 substrings in positions i and $i + 1$, for $i = 1, 2, \ldots, n$. These initial ranks are the ASCII values of the characters. At this point, the *left-rank* and *right-rank* values give the ranks of the left and right part of each substring of length 2. Because the substring starting at index 7 consists of only one character, its right part is empty and so its *right-rank* is 0. After the sorting step in line 8, the *substr-rank* array gives the relative lexicographic order of all the substrings of length 2, with starting points of these substrings in the *index* attribute. For example, the lexicographically smallest length-2 substring is `at`, which starts at position *substr-rank*$[1].$*index*, which equals 2. This substring also occurs at positions *substr-rank*$[2].$*index* $= 4$ and *substr-rank*$[3].$*index* $= 6$.

The procedure then enters the **while** loop of lines 10–19. The loop variable l gives an upper bound on the length of substrings that have been sorted thus far.

After line 11

i	*rank*
1	2
2	1
3	4
4	1
5	4
6	1
7	3

After lines 12–17

i	*left-rank*	*right-rank*	*index*	substring
1	2	4	1	`rata`
2	1	1	2	`atat`
3	4	4	3	`tata`
4	1	1	4	`atat`
5	4	3	5	`tat`
6	1	0	6	`at`
7	3	0	7	`t`

After line 18

i	*left-rank*	*right-rank*	*index*	substring
1	1	0	6	`at`
2	1	1	2	`atat`
3	1	1	4	`atat`
4	2	4	1	`rata`
5	3	0	7	`t`
6	4	3	5	`tat`
7	4	4	3	`tata`

Figure 32.13 The *rank* array after line 11 and the *substr-rank* array after lines 12–17 and after line 18 in the first iteration of the **while** loop of lines 10–19, where $l = 2$.

Entering the **while** loop, therefore, the substrings of length at most $l = 2$ are sorted. The call of MAKE-RANKS in line 11 gives each of these substrings its rank in the sorted order, from 1 up to the number of unique length-2 substrings, based on the values it finds in the *substr-rank* array. With $l = 2$, MAKE-RANKS sets $rank[i]$ to be the rank of the length-2 substring $T[i : i + 1]$. Figure 32.13 shows these new ranks, which are not necessarily unique. For example, since the length-2 substring `at` occurs at positions 2, 4, and 6, MAKE-RANKS finds that $substr\text{-}rank[1]$, $substr\text{-}rank[2]$, and $substr\text{-}rank[3]$ have equal values in *left-rank* and in *right-rank*. Since $substr\text{-}rank[1].index = 2$, $substr\text{-}rank[2].index = 4$, and $substr\text{-}rank[3].index = 6$, and since `at` is the smallest substring in lexicographic order, MAKE-RANKS sets $rank[2] = rank[4] = rank[6] = 1$.

This iteration of the **while** loop will sort the substrings of length at most 4 based on the ranks from sorting the substrings of length at most 2. The **for** loop of lines 12–17 reconstitutes the *substr-rank* array, with $substr\text{-}rank[i].left\text{-}rank$ based on $rank[i]$ (the rank of the length-2 substring $T[i : i+1]$) and $substr\text{-}rank[i].right\text{-}rank$ based on $rank[i+2]$ (the rank of the length-2 substring $T[i+2 : i+3]$, which is 0 if this substring starts beyond the end of the length-n text). Together, these two ranks give the relative rank of the length-4 substring $T[i : i + 3]$. Figure 32.13 shows the effect of lines 12–17. The figure also shows the result of sorting the *substr-rank* array in line 18, based on the *left-rank* attribute, and using the *right-rank* attribute to break ties. Now *substr-rank* gives the lexicographically sorted order of all substrings with length at most 4.

The next iteration of the **while** loop, with $l = 4$, sorts the substrings of length at most 8 based on the ranks from sorting the substrings of length at most[4] 4. Figure 32.14 shows the ranks of the length-4 substrings and the *substr-rank* array

[4] Why keep saying "length at most"? Because for a given value of l, a substring of length l starting at position i is $T[i : i + l - 1]$. If $i + l - 1 > n$, then the substring cuts off at the end of the text.

After line 11

i	*rank*
1	3
2	2
3	6
4	2
5	5
6	1
7	4

After lines 12–17

i	*left-rank*	*right-rank*	*index*	substring
1	3	5	1	`ratatat`
2	2	1	2	`atatat`
3	6	4	3	`tatat`
4	2	0	4	`atat`
5	5	0	5	`tat`
6	1	0	6	`at`
7	4	0	7	`t`

After line 18

i	*left-rank*	*right-rank*	*index*	substring
1	1	0	6	`at`
2	2	0	4	`atat`
3	2	1	2	`atatat`
4	3	5	1	`ratatat`
5	4	0	7	`t`
6	5	0	5	`tat`
7	6	4	3	`tatat`

Figure 32.14 The *rank* array after line 11 and the *substr-rank* array after lines 12–17 and after line 18 in the second—and final—iteration of the **while** loop of lines 10–19, where $l = 4$.

before and after sorting. This iteration is the final one, since with the length n of the text equaling 7, the procedure has sorted all substrings.

In general, as the loop variable l increases, more and more of the right parts of the substrings are empty. Therefore, more of the *right-rank* values are 0. Because i is at most n within the loop of lines 12–17, the left part of each substring is always nonempty, and so all *left-rank* values are always positive.

This example illuminates why the COMPUTE-SUFFIX-ARRAY procedure works. The initial ranks established in lines 2–7 are simply the ord values of the characters in the text, and so when line 8 sorts the *substr-rank* array, its ordering corresponds to the lexicographic ordering of the length-2 substrings. Each iteration of the **while** loop of lines 10–19 takes sorted substrings of length l and produces sorted substrings of length $2l$. Once l reaches or exceeds n, all substrings have been sorted.

Within an iteration of the **while** loop, the MAKE-RANKS procedure "re-ranks" the substrings that were sorted, either by line 8 before the first iteration or by line 18 in the previous iteration. MAKE-RANKS takes a *substr-rank* array, which has been sorted, and fills in an array $rank[1:n]$ so that $rank[i]$ is the rank of the ith substring represented in the *substr-rank* array. Each rank is a positive integer, starting from 1, and going up to the number of unique substrings of length $2l$. Substrings with equal values of *left-rank* and *right-rank* receive the same rank. Otherwise, a substring that is lexicographically smaller than another appears earlier in the *substr-rank* array, and it receives a smaller rank. Once the substrings of length $2l$ are re-ranked, line 18 sorts them by rank, preparing for the next iteration of the **while** loop.

Once l reaches or exceeds n and all substrings are sorted, the values in the *index* attributes give the starting positions of the sorted substrings. These indices are precisely the values that constitute the suffix array.

Let's analyze the running time of COMPUTE-SUFFIX-ARRAY. Lines 1–7 take $\Theta(n)$ time. Line 8 takes $O(n \lg n)$ time, using either merge sort (see Section 2.3.1) or heapsort (see Chapter 6). Because the value of l doubles in each iteration of

the **while** loop of lines 10–19, this loop makes $\lceil \lg n \rceil - 1$ iterations. Within each iteration, the call of MAKE-RANKS takes $\Theta(n)$ time, as does the **for** loop of lines 12–17. Line 18, like line 8, takes $O(n \lg n)$ time, using either merge sort or heapsort. Finally, the **for** loop of lines 20–21 takes $\Theta(n)$ time. The total time works out to $O(n \lg^2 n)$.

A simple observation allows us to reduce the running time to $\Theta(n \lg n)$. The values of *left-rank* and *right-rank* being sorted in line 18 are always integers in the range 0 to n. Therefore, radix sort can sort the *substr-rank* array in $\Theta(n)$ time by first running counting sort (see Chapter 8) based on *right-rank* and then running counting sort based on *left-rank*. Now each iteration of the **while** loop of lines 10–19 takes only $\Theta(n)$ time, giving a total time of $\Theta(n \lg n)$.

Exercise 32.5-2 asks you to make a simple modification to COMPUTE-SUFFIX-ARRAY that allows the **while** loop of lines 10–19 to iterate fewer than $\lceil \lg n \rceil - 1$ times for certain inputs.

Computing the *LCP* array

Recall that $LCP[i]$ is defined as the length of the longest common prefix of the $(i-1)$st and ith lexicographically smallest suffixes $T[SA[i-1]:]$ and $T[SA[i]:]$. Because $T[SA[1]:]$ is the lexicographically smallest suffix, we define $LCP[1]$ to be 0.

In order to compute the *LCP* array, we need an array *rank* that is the inverse of the *SA* array, just like the final *rank* array in COMPUTE-SUFFIX-ARRAY: if $SA[i] = j$, then $rank[j] = i$. That is, we have $rank[SA[i]] = i$ for $i = 1, 2, \ldots, n$. For a suffix $T[i:]$, the value of $rank[i]$ gives the position of this suffix in the lexicographically sorted order. Figure 32.11 includes the *rank* array for the `ratatat` example. For example, the suffix `tat` is $T[5:]$. To find this suffix's position in the sorted order, look up $rank[5] = 6$.

To compute the *LCP* array, we will need to determine where in the lexicographically sorted order a suffix appears, but with its first character removed. The *rank* array helps. Consider the ith smallest suffix, which is $T[SA[i]:]$. Dropping its first character gives the suffix $T[SA[i]+1:]$, that is, the suffix starting at position $SA[i]+1$ in the text. The location of this suffix in the sorted order is given by $rank[SA[i]+1]$. For example, for the suffix `atat`, let's see where to find `tat` (`atat` with its first character removed) in the lexicographically sorted order. The suffix `atat` appears in position 2 of the suffix array, and $SA[2] = 4$. Thus, $rank[SA[2]+1] = rank[5] = 6$, and sure enough the suffix `tat` appears in location 6 in the sorted order.

The procedure COMPUTE-LCP on the next page produces the *LCP* array. The following lemma helps show that the procedure is correct.

```
COMPUTE-LCP(T, SA, n)
allocate arrays rank[1 : n] and LCP[1 : n]
for i = 1 to n
    rank[SA[i]] = i                  // by definition
LCP[1] = 0                           // also by definition
l = 0                                // initialize length of LCP
for i = 1 to n
    if rank[i] > 1
        j = SA[rank[i] − 1]          // T[j :] precedes T[i :] lexicographically
        m = max {i, j}
        while m + l ≤ n and T[i + l] == T[j + l]
            l = l + 1                // next character is in common prefix
        LCP[rank[i]] = l             // length of LCP of T[j :] and T[i :]
        if l > 0
            l = l − 1                // peel off first character of common prefix
return LCP
```

Lemma 32.8
Consider suffixes $T[i-1:]$ and $T[i:]$, which appear at positions $rank[i-1]$ and $rank[i]$, respectively, in the lexicographically sorted order of suffixes. If $LCP[rank[i-1]] = l > 1$, then the suffix $T[i:]$, which is $T[i-1:]$ with its first character removed, has $LCP[rank[i]] \geq l-1$.

Proof The suffix $T[i-1:]$ appears at position $rank[i-1]$ in the lexicographically sorted order. The suffix immediately preceding it in the sorted order appears at position $rank[i-1]-1$ and is $T[SA[rank[i-1]-1]:]$. By assumption and the definition of the LCP array, these two suffixes, $T[SA[rank[i-1]-1]:]$ and $T[i-1:]$, have a longest common prefix of length $l > 1$. Removing the first character from each of these suffixes gives the suffixes $T[SA[rank[i-1]-1]+1:]$ and $T[i:]$, respectively. These suffixes have a longest common prefix of length $l-1$. If $T[SA[rank[i-1]-1]+1:]$ immediately precedes $T[i:]$ in the lexicographically sorted order (that is, if $rank[SA[rank[i-1]-1]+1] = rank[i]-1$), then the lemma is proven.

So now assume that $T[SA[rank[i-1]-1]+1:]$ does not immediately precede $T[i:]$ in the sorted order. Since $T[SA[rank[i-1]-1]:]$ immediately precedes $T[i-1:]$ and they have the same first $l > 1$ characters, $T[SA[rank[i-1]-1]+1:]$ must appear in the sorted order somewhere before $T[i:]$, with one or more other suffixes intervening. Each of these suffixes must start with the same $l-1$ characters as $T[SA[rank[i-1]-1]+1:]$ and $T[i:]$, for otherwise it would appear either before

$T[SA[rank[i-1]-1]+1:]$ or after $T[i:]$. Therefore, whichever suffix appears in position $rank[i]-1$, immediately before $T[i:]$, has at least its first $l-1$ characters in common with $T[i:]$. Thus, $LCP[rank[i]] \geq l-1$. ■

The COMPUTE-LCP procedure works as follows. After allocating the *rank* and *LCP* arrays in line 1, lines 2–3 fill in the *rank* array and line 4 pegs $LCP[1]$ to 0, per the definition of the *LCP* array.

The **for** loop of lines 6–14 fills in the rest of the *LCP* array going by decreasing-length suffixes. That is, it fills the position of the *LCP* array in the order $rank[1]$, $rank[2], rank[3], \ldots, rank[n]$, with the assignment occurring in line 12. Upon considering a suffix $T[i:]$, line 8 determines the suffix $T[j:]$ that immediately precedes $T[i:]$ in the lexicographically sorted order. At this point, the longest common prefix of $T[j:]$ and $T[i:]$ has length at least l. This property certainly holds upon the first iteration of the **for** loop, when $l = 0$. Assuming that line 12 sets $LCP[rank[i]]$ correctly, line 14 (which decrements l if it is positive) and Lemma 32.8 maintain this property for the next iteration. The longest common prefix of $T[j:]$ and $T[i:]$ might be even longer than the value of l at the start of the iteration, however. Lines 9–11 increment l for each additional character the prefixes have in common so that it achieves the length of the longest common prefix. The index m is set in line 9 and used in the test in line 10 to make sure that the test $T[i+l] == T[j+l]$ for extending the longest common prefix does not run off the end of the text T. When the **while** loop of lines 10–11 terminates, l is the length of the longest common prefix of $T[j:]$ and $T[i:]$.

As a simple aggregate analysis shows, the COMPUTE-LCP procedure runs in $\Theta(n)$ time. Each of the two **for** loops iterates n times, and so it remains only to bound the total number of iterations by the **while** loop of lines 10–11. Each iteration increases l by 1, and the test $m + l \leq n$ ensures that l is always less than n. Because l has an initial value of 0 and decreases at most $n-1$ times in line 14, line 11 increments l fewer than $2n$ times. Thus, COMPUTE-LCP takes $\Theta(n)$ time.

Exercises

32.5-1

Show the *substr-rank* and *rank* arrays before each iteration of the **while** loop of lines 10–19 and after the last iteration of the **while** loop, the suffix array *SA* returned, and the sorted suffixes when COMPUTE-SUFFIX-ARRAY is run on the text `hippityhoppity`. Use the position of each letter in the alphabet as its ord value, so that ord(`b`) = 2. Then show the *LCP* array after each iteration of the **for** loop of lines 6–14 of COMPUTE-LCP given the text `hippityhoppity` and its suffix array.

32.5-2

For some inputs, the COMPUTE-SUFFIX-ARRAY procedure can produce the correct result with fewer than $\lceil \lg n \rceil - 1$ iterations of the **while** loop of lines 10–19. Modify COMPUTE-SUFFIX-ARRAY (and, if necessary, MAKE-RANKS) so that the procedure can stop before making all $\lceil \lg n \rceil - 1$ iterations in some cases. Describe an input that allows the procedure to make $O(1)$ iterations. Describe an input that forces the procedure to make the maximum number of iterations.

32.5-3

Given two texts, T_1 of length n_1 and T_2 of length n_2, show how to use the suffix array and longest common prefix array to find all of the ***longest common substrings***, that is, the longest substrings that appear in both T_1 and T_2. Your algorithm should run in $O(n \lg n + kl)$ time, where $n = n_1 + n_2$ and there are k such longest substrings, each with length l.

32.5-4

Professor Markram proposes the following method to find the longest palindromes in a string $T[1:n]$ by using its suffix array and LCP array. (Recall from Problem 14-2 that a palindrome is a nonempty string that reads the same forward and backward.)

> Let `@` be a character that does not appear in T. Construct the text T' as the concatenation of T, `@`, and the reverse of T. Denote the length of T' by $n' = 2n + 1$. Create the suffix array SA and LCP array LCP for T'. Since the indices for a palindrome and its reverse appear in consecutive positions in the suffix array, find the entries with the maximum LCP value $LCP[i]$ such that $SA[i-1] = n' - SA[i] - LCP[i] + 2$. (This constraint prevents a substring—and its reverse—from being construed as a palindrome unless it really is one.) For each such index i, one of the longest palindromes is $T'[SA[i] : SA[i] + LCP[i] - 1]$.

For example, if the text T is `unreferenced`, with $n = 12$, then the text T' is `unreferenced@decnerefernu`, with $n' = 25$ and the following suffix array and LCP array:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$T'[i]$	u	n	r	e	f	e	r	e	n	c	e	d	@	d	e	c	n	e	r	e	f	e	r	n	u
$SA[i]$	13	10	16	12	14	15	11	4	20	8	18	6	22	5	21	9	17	2	24	3	19	7	23	25	1
$LCP[i]$	0	0	1	0	1	0	1	1	4	1	1	3	2	0	3	0	1	1	1	0	5	2	1	0	1

The maximum LCP value is achieved at $LCP[21] = 5$, and $SA[20] = 3 = n' - SA[21] - LCP[21] + 2$. The suffixes of T' starting at indices $SA[20]$ and $SA[21]$ are `referenced@decnerefernu` and `refernu`, both of which start with the length-5 palindrome `refer`.

Alas, this method is not foolproof. Give an input string T that causes this method to give results that are shorter than the longest palindrome contained within T, and explain why your input causes the method to fail.

Problems

32-1 String matching based on repetition factors

Let y^i denote the concatenation of string y with itself i times. For example, $(\texttt{ab})^3 = \texttt{ababab}$. We say that a string $x \in \Sigma^*$ has ***repetition factor*** r if $x = y^r$ for some string $y \in \Sigma^*$ and some $r > 0$. Let $\rho(x)$ denote the largest r such that x has repetition factor r.

a. Give an efficient algorithm that takes as input a pattern $P[1:m]$ and computes the value $\rho(P[1:i])$ for $i = 1, 2, \ldots, m$. What is the running time of your algorithm?

b. For any pattern $P[1:m]$, let $\rho^*(P)$ be defined as $\max\{\rho(P[1:i]) : 1 \le i \le m\}$. Prove that if the pattern P is chosen randomly from the set of all binary strings of length m, then the expected value of $\rho^*(P)$ is $O(1)$.

c. Argue that the procedure REPETITION-MATCHER correctly finds all occurrences of pattern $P[1:m]$ in text $T[1:n]$ in $O(\rho^*(P)n + m)$ time. (This algorithm is due to Galil and Seiferas. By extending these ideas greatly, they obtained a linear-time string-matching algorithm that uses only $O(1)$ storage beyond what is required for P and T.)

```
REPETITION-MATCHER(T, P, n, m)
k = 1 + ρ*(P)
q = 0
s = 0
while s ≤ n − m
    if T[s + q + 1] == P[q + 1]
        q = q + 1
        if q == m
            print "Pattern occurs with shift" s
    if q == m or T[s + q + 1] ≠ P[q + 1]
        s = s + max{1, ⌈q/k⌉}
        q = 0
```

32-2 A linear-time suffix-array algorithm

In this problem, you will develop and analyze a linear-time divide-and-conquer algorithm to compute the suffix array of a text $T[1:n]$. As in Section 32.5, assume that each character in the text is represented by an underlying encoding, which is a positive integer.

The idea behind the linear-time algorithm is to compute the suffix array for the suffixes starting at $2/3$ of the positions in the text, recursing as needed, use the resulting information to sort the suffixes starting at the remaining $1/3$ of the positions, and then merge the sorted information in linear time to produce the full suffix array.

For $i = 1, 2, \ldots, n$, if $i \bmod 3$ equals 1 or 2, then i is a ***sample position***, and the suffixes starting at such positions are ***sample suffixes***. Positions $3, 6, 9, \ldots$ are ***nonsample positions***, and the suffixes starting at nonsample positions are ***nonsample suffixes***.

The algorithm sorts the sample suffixes, sorts the nonsample suffixes (aided by the result of sorting the sample suffixes), and merges the sorted sample and nonsample suffixes. Using the example text $T = \texttt{bippityboppityboo}$, here is the algorithm in detail, listing substeps of each of the above steps:

1. The sample suffixes comprise about $2/3$ of the suffixes. Sort them by the following substeps, which work with a heavily modified version of T and may require recursion. In part (a) of this problem on page 999, you will show that the orders of the suffixes of T and the suffixes of the modified version of T are the same.

   A. Construct two texts P_1 and P_2 made up of "metacharacters" that are actually substrings of three consecutive characters from T. We delimit each such metacharacter with parentheses. Construct

   $$P_1 = (T[1:3])\,(T[4:6])\,(T[7:9]) \cdots (T[n':n'+2])\ ,$$

   where n' is the largest integer congruent to 1, modulo 3, that is less than or equal to n and T is extended beyond position n with the special character $\varnothing$, with encoding 0. With the example text $T = \texttt{bippityboppityboo}$, we get that

   $$P_1 = (\texttt{bip})\,(\texttt{pit})\,(\texttt{ybo})\,(\texttt{ppi})\,(\texttt{tyb})\,(\texttt{oo}\varnothing)\ .$$

   Similarly, construct

   $$P_2 = (T[2:4])\,(T[5:7])\,(T[8:10]) \cdots (T[n'':n''+2])\ ,$$

   where n'' is the largest integer congruent to 2, modulo 3, that is less than or equal to n. For our example, we have

   $$P_2 = (\texttt{ipp})\,(\texttt{ity})\,(\texttt{bop})\,(\texttt{pit})\,(\texttt{ybo})\,(\texttt{o}\varnothing\varnothing)\ .$$

position in T	1	4	7	10	13	16	2	5	8	11	14	17
metacharacter in P	(bip)	(pit)	(ybo)	(ppi)	(tyb)	(oo∅)	(ipp)	(ity)	(bop)	(pit)	(ybo)	(o∅∅)
character in P'	1	7	10	8	9	6	3	4	2	7	10	5
position in P'	1	2	3	4	5	6	7	8	9	10	11	12
$SA_{P'}$	1	9	7	8	12	6	10	2	4	5	11	3
positions in T of sorted sample suffixes of T	1	8	2	5	17	16	11	4	10	13	14	7

Figure 32.15 Computed values when sorting the sample suffixes of the linear-time suffix-array algorithm for the text $T =$ `bippityboppityboo`.

If n is a multiple of 3, append the metacharacter (∅∅∅) to the end of P_1. In this way, P_1 is guaranteed to end with a metacharacter containing ∅. (This property helps in part (a) of this problem.) The text P_2 may or may not end with a metacharacter containing ∅.

B. Concatenate P_1 and P_2 to form a new text P. Figure 32.15 shows P for our example, along with the corresponding positions of T.

C. Sort and rank the unique metacharacters of P, with ranks starting from 1. In the example, P has 10 unique metacharacters: in sorted order, they are (`bip`), (`bop`), (`ipp`), (`ity`), (o∅∅), (oo∅), (`pit`), (`ppi`), (`tyb`), (`ybo`). The metacharacters (`pit`) and (`ybo`) each appear twice.

D. As Figure 32.15 shows, construct a new "text" P' by renaming each metacharacter in P by its rank. If P contains k unique metacharacters, then each "character" in P' is an integer from 1 to k. The suffix arrays for P and P' are identical.

E. Compute the suffix array $SA_{P'}$ of P'. If the characters of P' (i.e., the ranks of metacharacters in P) are unique, then you can compute its suffix array directly, since the ordering of the individual characters gives the suffix array. Otherwise, recurse to compute the suffix array of P', treating the ranks in P' as the input characters in the recursive call. Figure 32.15 shows the suffix array $SA_{P'}$ for our example. Since the number of metacharacters in P, and hence the length of P', is approximately $2n/3$, this recursive subproblem is smaller than the current problem.

F. From $SA_{P'}$ and the positions in T corresponding to the sample positions, compute the list of positions of the sorted sample suffixes of the original text T. Figure 32.15 shows the list of positions in T of the sorted sample suffixes in our example.

2. The nonsample suffixes comprise about $1/3$ of the suffixes. Using the sorted sample suffixes, sort the nonsample suffixes by the following substeps.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$T[i]$	b	i	p	p	i	t	y	b	o	p	p	i	t	y	b	o	o	$\varnothing$	$\varnothing$
r_i	1	3	$\square$	8	4	$\square$	12	2	$\square$	9	7	$\square$	10	11	$\square$	6	5	0	0

Figure 32.16 The ranks r_1 through r_{n+3} for the text $T =$ `bippityboppityboo` with $n = 17$.

G. Extending the text T by the two special characters $\varnothing\varnothing$, so that T now has $n + 2$ characters, consider each suffix $T[i:]$ for $i = 1, 2, \ldots, n + 2$. Assign a rank r_i to each suffix $T[i:]$. For the two special characters $\varnothing\varnothing$, set $r_{n+1} = r_{n+2} = 0$. For the sample positions of T, base the rank on the list of sorted sample positions of T. The rank is currently undefined for the nonsample positions of T. For these positions, set $r_i = \square$. Figure 32.16 shows the ranks for $T =$ `bippityboppityboo` with $n = 17$.

H. Sort the nonsample suffixes by comparing tuples $(T[i], r_{i+1})$. In our example, we get $T[15:] < T[12:] < T[9:] < T[3:] < T[6:]$ because $(\texttt{b}, 6) < (\texttt{i}, 10) < (\texttt{o}, 9) < (\texttt{p}, 8) < (\texttt{t}, 12)$.

3. Merge the sorted sets of suffixes. From the sorted set of suffixes, determine the suffix array of T.

This completes the description of a linear-time algorithm for computing suffix arrays. The following parts of this problem ask you to show that certain steps of this algorithm are correct and to analyze the algorithm's running time.

a. Define a ***nonempty suffix*** at position i of the text P created in substep B as all metacharacters from position i of P up to and including the first metacharacter of P in which $\varnothing$ appears or the end of P. In the example shown in Figure 32.15, the nonempty suffixes of P starting at positions 1, 4, and 11 of P are (`bip`) (`pit`) (`ybo`) (`ppi`) (`tyb`) (`oo`$\varnothing$), (`ppi`) (`tyb`) (`oo`$\varnothing$), and (`ybo`) (`o`$\varnothing\varnothing$), respectively. Prove that the order of suffixes of P is the same as the order of its nonempty suffixes. Conclude that the order of suffixes of P gives the order of the sample suffixes of T. (*Hint:* If P contains duplicate metacharacters, consider separately the cases in which two suffixes both start in P_1, both start in P_2, and one starts in P_1 and the other starts in P_2. Use the property that $\varnothing$ appears in the last metacharacter of P_1.)

b. Show how to perform substep C in $\Theta(n)$ time, bearing in mind that in a recursive call, the characters in T are actually ranks in P' in the caller.

c. Argue that the tuples in substep H are unique. Then show how to perform this substep in $\Theta(n)$ time.

d. Consider two suffixes $T[i:]$ and $T[j:]$, where $T[i:]$ is a sample suffix and $T[j:]$ is a nonsample suffix. Show how to determine in $\Theta(1)$ time whether $T[i:]$ is lexicographically smaller than $T[j:]$. (*Hint:* Consider separately the cases in which $i \bmod 3 = 1$ and $i \bmod 3 = 2$. Compare tuples whose elements are characters in T and ranks as shown in Figure 32.16. The number of elements per tuple may depend on whether $i \bmod 3$ equals 1 or 2.) Conclude that step 3 can be performed in $\Theta(n)$ time.

e. Justify the recurrence $T(n) \le T(2n/3 + 2) + \Theta(n)$ for the running time of the full algorithm, and show that its solution is $O(n)$. Conclude that the algorithm runs in $\Theta(n)$ time.

32-3 Burrows-Wheeler transform

The ***Burrows-Wheeler transform***, or ***BWT***, for a text T is defined as follows. First, append a new character that compares as lexicographically less than every character of T, and denote this character by `$` and the resulting string by T'. Letting n be the length of T', create n rows of characters, where each row is one of the n cyclic rotations of T'. Next, sort the rows lexicographically. The BWT is then the string of n characters in the rightmost column, read top to bottom.

For example, let T = `rutabaga`, so that T' = `rutabaga$`. The cyclic rotations are

```
rutabaga$
utabaga$r
tabaga$ru
abaga$rut
baga$ruta
aga$rutab
ga$rutaba
a$rutabag
$rutabaga
```

Sorting the rows and numbering the sorted rows gives

```
1  $rutabaga
2  a$rutabag
3  abaga$rut
4  aga$rutab
5  baga$ruta
6  ga$rutaba
7  rutabaga$
8  tabaga$ru
9  utabaga$r
```

The BWT is the rightmost column, `agtbaa$ur`. (The row numbering will be helpful in understanding how to compute the inverse BWT.)

The BWT has applications in bioinformatics, and it can also be a step in text compression. That is because it tends to place identical characters together, as in the BWT of `rutabaga`, which places two of the instances of `a` together. When identical characters are placed together, or even nearby, additional means of compressing become available. Following the BWT, combinations of move-to-front encoding, run-length encoding, and Huffman coding (see Section 15.3) can provide significant text compression. Compression ratios with the BWT tend improve as the text length increases.

a. Given the suffix array for T', show how to compute the BWT in $\Theta(n)$ time.

In order to decompress, the BWT must be invertible. Assuming that the alphabet size is constant, the inverse BWT can be computed in $\Theta(n)$ time from the BWT. Let's look at the BWT of `rutabaga`, denoting it by $BWT[1:n]$. Each character in the BWT has a unique lexicographic rank from 1 to n. Denote the rank of $BWT[i]$ by $rank[i]$. If a character appears multiple times in the BWT, each instance of the character has a rank 1 greater than the previous instance of the character. Here are *BWT* and *rank* for `rutabaga`:

i	1	2	3	4	5	6	7	8	9
$BWT[i]$	a	g	t	b	a	a	$	u	r
$rank[i]$	2	6	8	5	3	4	1	9	7

For example, $rank[1] = 2$ because $BWT[1] =$ `a` and the only character that precedes the first `a` lexicographically is `$` (which we defined to precede all other characters, so that `$` has rank 1). Next, we have $rank[2] = 6$ because $BWT[2] =$ `g` and five characters in the BWT precede `g` lexicographically: `$`, the three instances of `a`, and `b`. Jumping ahead to $rank[5] = 3$, that is because $BWT[5] =$ `a`, and because this `a` is the second instance of `a` in the BWT, its *rank* value is 1 greater than the *rank* value for the previous instance of `a`, in position 1.

There is enough information in *BWT* and *rank* to reconstruct T' from back to front. Suppose that you know the rank r of a character c in T'. Then c is the first character in row r of the sorted cyclic rotations. The last character in row r must be the character that precedes c in T'. But you know which character is the last character in row r, because it is $BWT[r]$. To reconstruct T' from back to front, start with `$`, which you can find in *BWT*. Then work backward using *BWT* and *rank* to reconstruct T'.

Let's see how this strategy works for `rutabaga`. The last character of T', `$`, appears in position 7 of *BWT*. Since $rank[7] = 1$, row 1 of the sorted cyclic rotations of T' begins with `$`. The character that precedes `$` in T' is the last character in row 1, which is $BWT[1]$: `a`. Now we know that the last two characters of T'

are a$. Looking up *rank*[1], it equals 2, so that row 2 of the sorted cyclic rotations of T' begins with a. The last character in row 2 precedes a in T', and that character is $BWT[2] =$ g. Now we know that the last three characters of T' are ga$. Continuing on, we have $rank[2] = 6$, so that row 6 of the sorted cyclic rotations begins with g. The character preceding g in T' is $BWT[6] =$ a, and so the last four characters of T' are aga$. Because $rank[6] = 4$, a begins row 4 of the sorted cyclic rotations of T'. The character preceding a in T' is the last character in row 4, $BWT[4] =$ b, and the last five characters of T' are baga$. And so on, until all n characters of T' have been identified, from back to front.

b. Given the array $BWT[1:n]$, write pseudcode to compute the array $rank[1:n]$ in $\Theta(n)$ time, assuming that the alphabet size is constant.

c. Given the arrays $BWT[1:n]$ and $rank[1:n]$, write pseudocode to compute T' in $\Theta(n)$ time.

Chapter notes

The relation of string matching to the theory of finite automata is discussed by Aho, Hopcroft, and Ullman [5]. The Knuth-Morris-Pratt algorithm [267] was invented independently by Knuth and Pratt and by Morris, but they published their work jointly. Matiyasevich [317] earlier discovered a similar algorithm, which applied only to an alphabet with two characters and was specified for a Turing machine with a two-dimensional tape. Reingold, Urban, and Gries [377] give an alternative treatment of the Knuth-Morris-Pratt algorithm. The Rabin-Karp algorithm was proposed by Karp and Rabin [250]. Galil and Seiferas [173] give an interesting deterministic linear-time string-matching algorithm that uses only $O(1)$ space beyond that required to store the pattern and text.

The suffix-array algorithm in Section 32.5 is by Manber and Myers [312], who first proposed the notion of suffix arrays. The linear-time algorithm to compute the longest common prefix array presented here is by Kasai et al. [252]. Problem 32-2 is based on the DC3 algorithm by Kärkkäinen, Sanders, and Burkhardt [245]. For a survey of suffix-array algorithms, see the article by Puglisi, Smyth, and Turpin [370]. To learn more about the Burrows-Wheeler transform from Problem 32-3, see the articles by Burrows and Wheeler [78] and Manzini [314].

33 Machine-Learning Algorithms

Machine learning may be viewed as a subfield of artificial intelligence. Broadly speaking, artificial intelligence aims to enable computers to carry out complex perception and information-processing tasks with human-like performance. The field of AI is vast and uses many different algorithmic methods.

Machine learning is rich and fascinating, with strong ties to statistics and optimization. Technology today produces enormous amounts of data, providing myriad opportunities for machine-learning algorithms to formulate and test hypotheses about patterns within the data. These hypotheses can then be used to make predictions about the characteristics or classifications in new data. Because machine learning is particularly good with challenging tasks involving uncertainty, where observed data follows unknown rules, it has markedly transformed fields such as medicine, advertising, and speech recognition.

This chapter presents three important machine-learning algorithms: k-means clustering, multiplicative weights, and gradient descent. You can view each of these tasks as a learning problem, whereby an algorithm uses the data collected so far to produce a hypothesis that describes the regularities learned and/or makes predictions about new data. The boundaries of machine learning are imprecise and evolving—some might say that the k-means clustering algorithm should be called "data science" and not "machine learning," and gradient descent, though an immensely important algorithm for machine learning, also has a multitude of applications outside of machine learning (most notably for optimization problems).

Machine learning typically starts with a ***training phase*** followed by a ***prediction phase*** in which predictions are made about new data. For ***online learning***, the training and prediction phases are intermingled. The training phase takes as input ***training data***, where each input data point has an associated output or ***label***; the label might be a category name or some real-valued attribute. It then produces as an output one or more ***hypotheses*** about how the labels depend on the attributes of the input data points. Hypotheses can take many forms, typically some type of formula or algorithm. The learning algorithm used is often a form of gradient

descent. The prediction phase then uses the hypothesis on new data in order to make ***predictions*** regarding the labels of new data points.

The type of learning just described is known as ***supervised learning***, since it starts with a set of inputs that are each labeled. As an example, consider a machine-learning algorithm to recognize spam emails. The training data comprises a collection of emails, each of which is labeled either "spam" or "not spam." The machine-learning algorithm frames a hypothesis, possibly a rule of the form "if an email has one of a set of words, then it is likely to be spam." Or it might learn rules that assign a spam score to each word and then evaluates a document by the sum of the spam scores of its constituent words, so that a document with a total score above a certain threshold value is classified as spam. The machine-learning algorithm can then predict whether a new email is spam or not.

A second form of machine learning is ***unsupervised learning***, where the training data is unlabeled, as in the clustering problem of Section 33.1. Here the machine-learning algorithm produces hypotheses regarding the centers of groups of input data points.

A third form of machine learning (not covered further here) is ***reinforcement learning***, where the machine-learning algorithm takes actions in an environment, receives feedback for those actions from the environment, and then updates its model of the environment based on the feedback. The learner is in an environment that has some state, and the actions of the learner have an effect on that state. Reinforcement learning is a natural choice for situations such as game playing or operating a self-driving car.

Sometimes the goal in a supervised machine-learning application is not making accurate predictions of labels for new examples, but rather performing causal ***inference***: finding an explanatory model that describes how the various features of an input data point affect its associated label. Finding a model that fits a given set of training data well can be tricky. It may involve sophisticated optimization methods that need to balance between producing a hypothesis that fits the data well and producing a hypothesis that is simple.

This chapter focuses on three problem domains: finding hypotheses that group the input data points well (using a clustering algorithm), learning which predictors (experts) to rely upon for making predictions in an online learning problem (using the multiplicative-weights algorithm), and fitting a model to data (using gradient descent).

Section 33.1 considers the clustering problem: how to divide a given set of n training data points into a given number k of groups, or "clusters," based on a measure of how similar (or more accurately, how dissimilar) points are to each other. The approach is iterative, beginning with an arbitrary initial clustering and incorporating successive improvements until no further improvements occur. Clustering

is often used as an initial step when working on a machine-learning problem to discover what structure exists in the data.

Section 33.2 shows how to make online predictions quite accurately when you have a set of predictors, often called "experts," to rely on, many of which might be poor predictors, but some of which are good predictors. At first, you do not know which predictors are poor and which are good. The goal is to make predictions on new examples that are nearly as good as the predictions made by the best predictor. We study an effective multiplicative-weights prediction method that associates a positive real weight with each predictor and multiplicatively decreases the weights associated with predictors when they make poor predictions. The model in this section is online (see Chapter 27): at each step, we do not know anything about the future examples. In addition, we are able to make predictions even in the presence of adversarial experts, who are collaborating against us, a situation that actually happens in game-playing settings.

Finally, Section 33.3 introduces gradient descent, a powerful optimization technique used to find parameter settings in machine-learning models. Gradient descent also has many applications outside of machine learning. Intuitively, gradient descent finds the value that produces a local minimum for a function by "walking downhill." In a learning application, a "downhill step" is a step that adjusts hypothesis parameters so that the hypothesis does better on the given set of labeled examples.

This chapter makes extensive use of vectors. In contrast to the rest of the book, vector names in this chapter appear in boldface, such as $\mathbf{x}$, to more clearly delineate which quantities are vectors. Components of vectors do not appear in boldface, so if vector $\mathbf{x}$ has d dimensions, we might write $\mathbf{x} = (x_1, x_2, \ldots, x_d)$.

33.1 Clustering

Suppose that you have a large number of data points (examples), and you wish to group them into classes based on how similar they are to each other. For example, each data point might represent a celestial star, giving its temperature, size, and spectral characteristics. Or, each data point might represent a fragment of recorded speech. Grouping these speech fragments appropriately might reveal the set of accents of the fragments. Once a grouping of the training data points is found, new data can be placed into an appropriate group, facilitating star-type recognition or speech recognition.

These situations, along with many others, fall under the umbrella of clustering. The input to a ***clustering*** problem is a set of n examples (objects) and an integer k, with the goal of dividing the examples into at most k disjoint clusters such that

the examples in each cluster are similar to each other. The clustering problem has several variations. For example, the integer k might not be given, but instead arises out of the clustering procedure. In this section we presume that k is given.

Feature vectors and similarity

Let's formally define the clustering problem. The input is a set of n ***examples***. Each example has a set of ***attributes*** in common with all other examples, though the attribute values may vary among examples. For example, the clustering problem shown in Figure 33.1 clusters $n = 49$ examples—48 state capitals plus the District of Columbia—into $k = 4$ clusters. Each example has two attributes: the latitude and longitude of the capital. In a given clustering problem, each example has d attributes, with an example $\mathbf{x}$ specified by a d-dimensional ***feature vector***

$$\mathbf{x} = (x_1, x_2, \ldots, x_d) .$$

Here, x_a for $a = 1, 2, \ldots, d$ is a real number giving the value of attribute a for example $\mathbf{x}$. We call $\mathbf{x}$ the ***point*** in $\mathbb{R}^d$ representing the example. For the example in Figure 33.1, each capital $\mathbf{x}$ has its latitude in x_1 and its longitude in x_2.

In order to cluster similar points together, we need to define similarity. Instead, let's define the opposite: the ***dissimilarity*** $\Delta(\mathbf{x}, \mathbf{y})$ of points $\mathbf{x}$ and $\mathbf{y}$ is the squared Euclidean distance between them:

$$\begin{aligned} \Delta(\mathbf{x}, \mathbf{y}) &= \|\mathbf{x} - \mathbf{y}\|^2 \\ &= \sum_{a=1}^{d} (x_a - y_a)^2 . \end{aligned} \tag{33.1}$$

Of course, for $\Delta(\mathbf{x}, \mathbf{y})$ to be well defined, all attribute values must be present. If any are missing, then you might just ignore that example, or you could fill in a missing attribute value with the median value for that attribute.

The attribute values are often "messy" in other ways, so that some "data cleaning" is necessary before the clustering algorithm is run. For example, the scale of attribute values can vary widely across attributes. In the example of Figure 33.1, the scales of the two attributes vary by a factor of 2, since latitude ranges from -90 to $+90$ degrees but longitude ranges from -180 to $+180$ degrees. You can imagine other scenarios where the differences in scales are even greater. If the examples contain information about students, one attribute might be grade-point average but another might be family income. Therefore, the attribute values are usually scaled or normalized, so that no single attribute can dominate the others when computing dissimilarities. One way to do so is by scaling attribute values with a linear transform so that the minimum value becomes 0 and the maximum value becomes 1. If the attribute values are binary values, then no scaling may be needed. Another

Figure 33.1 The iterations of Lloyd's procedure when clustering the capitals of the lower 48 states and the District of Columbia into $k = 4$ clusters. Each capital has two attributes: latitude and longitude. Each iteration reduces the value f, measuring the sum of squares of distances of all capitals to their cluster centers, until the value of f does not change. **(a)** The initial four clusters, with the capitals of Arkansas, Kansas, Louisiana, and Tennessee chosen as centers. **(b)–(k)** Iterations of Lloyd's procedure. **(l)** The 11th iteration results in the same value of f as the 10th iteration in part (k), and so the procedure terminates.

option is scaling so that the values for each attribute have mean 0 and unit variance. Sometimes it makes sense to choose the same scaling rule for several related attributes (for example, if they are lengths measured to the same scale).

Also, the choice of dissimilarity measure is somewhat arbitrary. The use of the sum of squared differences as in equation (33.1) is not required, but it is a conventional choice and mathematically convenient. For the example of Figure 33.1, you might use the actual distance between capitals rather than equation (33.1).

Clusterings

With the notion of similarity (actually, *dis*similarity) defined, let's see how to define clusters of similar points. Let S denote the given set of n points in $\mathbb{R}^d$. In some applications the points are not necessarily distinct, so that S is a multiset rather than a set.

Because the goal is to create k clusters, we define a ***k-clustering*** of S as a decomposition of S into a sequence $\langle S^{(1)}, S^{(2)}, \ldots, S^{(k)} \rangle$ of k disjoint subsets, or ***clusters***, so that

$$S = S^{(1)} \cup S^{(2)} \cup \cdots \cup S^{(k)} .$$

A cluster may be empty, for example if $k > 1$ but all of the points in S have the same attribute values.

There are many ways to define a k-clustering of S and many ways to evaluate the quality of a given k-clustering. We consider here only k-clusterings of S that are defined by a sequence C of k ***centers***

$$C = \langle \mathbf{c}^{(1)}, \mathbf{c}^{(2)}, \ldots, \mathbf{c}^{(k)} \rangle ,$$

where each center is a point in $\mathbb{R}^d$, and the ***nearest-center rule*** says that a point $\mathbf{x}$ may belong to cluster $S^{(\ell)}$ if the center of no other cluster is closer to $\mathbf{x}$ than the center $\mathbf{c}^{(\ell)}$ of $S^{(\ell)}$:

$$\mathbf{x} \in S^{(\ell)} \text{ only if } \Delta(\mathbf{x}, \mathbf{c}^{(\ell)}) = \min \{ \Delta(\mathbf{x}, \mathbf{c}^{(j)}) : 1 \le j \le k \} .$$

A center can be anywhere, and not necessarily a point in S.

Ties are possible and must be broken so that each point lies in exactly one cluster. In general, ties may be broken arbitrarily, although we'll need the property that we never change which cluster a point $\mathbf{x}$ is assigned to unless the distance from $\mathbf{x}$ to its new cluster center is *strictly smaller* than the distance from $\mathbf{x}$ to its old cluster center. That is, if the current cluster has a center that is one of the closest cluster centers to $\mathbf{x}$, then don't change which cluster $\mathbf{x}$ is assigned to.

The ***k-means problem*** is then the following: given a set S of n points and a positive integer k, find a sequence $C = \langle \mathbf{c}^{(1)}, \mathbf{c}^{(2)}, \ldots, \mathbf{c}^{(k)} \rangle$ of k center points

minimizing the sum $f(S, C)$ of the squared distance from each point to its nearest center, where

$$\begin{aligned} f(S, C) &= \sum_{\mathbf{x} \in S} \min \{\Delta(\mathbf{x}, \mathbf{c}^{(j)}) : 1 \le j \le k\} \\ &= \sum_{\ell=1}^{k} \sum_{\mathbf{x} \in S^{(\ell)}} \Delta(\mathbf{x}, \mathbf{c}^{(\boldsymbol{\ell})}) . \end{aligned} \tag{33.2}$$

In the second line, the k-clustering $\langle S^{(1)}, S^{(2)}, \ldots, S^{(k)} \rangle$ is defined by the centers C and the nearest-center rule. See Exercise 33.1-1 for an alternative formulation based on pairwise interpoint distances.

Is there a polynomial-time algorithm for the k-means problem? Probably not, because it is NP-hard [310]. As we'll see in Chapter 34, NP-hard problems have no known polynomial-time algorithm, but nobody has ever proven that polynomial-time algorithms for NP-hard problems cannot exist. Although we know of no polynomial-time algorithm that finds the global minimum over all clusterings (according to equation (33.2)), we *can* find a local minimum.

Lloyd [304] proposed a simple procedure that finds a sequence C of k centers that yields a local minimum of $f(S, C)$. A local minimum in the k-means problem satisfies two simple properties: each cluster has an optimal center (defined below), and each point is assigned to the cluster (or one of the clusters) with the closest center. Lloyd's procedure finds a good clustering—possibly optimal—that satisfies these two properties. These properties are necessary, but not sufficient, for optimality.

Optimal center for a given cluster

In an optimal solution to the k-means problem, each center point must be the ***centroid***, or ***mean***, of the points in its cluster. The centroid is a d-dimensional point, where the value in each dimension is the mean of the values of all the points in the cluster in that dimension (that is, the mean of the corresponding attribute values in the cluster). That is, if $\mathbf{c}^{(\boldsymbol{\ell})}$ is the centroid for cluster $S^{(\ell)}$, then for attributes $a = 1, 2, \ldots, d$, we have

$$c_a^{(\ell)} = \frac{1}{|S^{(\ell)}|} \sum_{\mathbf{x} \in S^{(\ell)}} x_a .$$

Over all attributes, we write

$$\mathbf{c}^{(\boldsymbol{\ell})} = \frac{1}{|S^{(\ell)}|} \sum_{\mathbf{x} \in S^{(\ell)}} \mathbf{x} . \tag{33.3}$$

Theorem 33.1
Given a nonempty cluster $S^{(\ell)}$, its centroid (or mean) is the unique choice for the cluster center $\mathbf{c}^{(\boldsymbol{\ell})} \in \mathbb{R}^d$ that minimizes

$$\sum_{\mathbf{x} \in S^{(\ell)}} \Delta(\mathbf{x}, \mathbf{c}^{(\boldsymbol{\ell})}) .$$

Proof We wish to minimize, by choosing $\mathbf{c}^{(\boldsymbol{\ell})} \in \mathbb{R}^d$, the sum

$$\begin{aligned}\sum_{\mathbf{x} \in S^{(\ell)}} \Delta(\mathbf{x}, \mathbf{c}^{(\boldsymbol{\ell})}) &= \sum_{\mathbf{x} \in S^{(\ell)}} \sum_{a=1}^{d} (x_a - c_a^{(\ell)})^2 \\ &= \sum_{a=1}^{d} \left(\sum_{\mathbf{x} \in S^{(\ell)}} x_a^2 - 2 \left(\sum_{\mathbf{x} \in S^{(\ell)}} x_a \right) c_a^{(\ell)} + \left| S^{(\ell)} \right| (c_a^{(\ell)})^2 \right) .\end{aligned}$$

For each attribute a, the term summed is a convex quadratic function in $c_a^{(\ell)}$. To minimize this function, take its derivative with respect to $c_a^{(\ell)}$ and set it to 0:

$$-2 \sum_{\mathbf{x} \in S^{(\ell)}} x_a + 2 \left| S^{(\ell)} \right| c_a^{(\ell)} = 0$$

or, equivalently,

$$c_a^{(\ell)} = \frac{1}{|S^{(\ell)}|} \sum_{\mathbf{x} \in S^{(\ell)}} x_a .$$

Since the minimum is obtained uniquely when each coordinate of $c_a^{(\ell)}$ is the average of the corresponding coordinate for $\mathbf{x} \in S^{(\ell)}$, the overall minimum is obtained when $\mathbf{c}^{(\boldsymbol{\ell})}$ is the centroid of the points $\mathbf{x}$, as in equation (33.3). ■

Optimal clusters for given centers

The following theorem shows that the nearest-center rule—assigning each point $\mathbf{x}$ to one of the clusters whose center is nearest to $\mathbf{x}$—yields an optimal solution to the k-means problem.

Theorem 33.2
Given a set S of n points and a sequence $\langle \mathbf{c}^{(\mathbf{1})}, \mathbf{c}^{(\mathbf{2})}, \ldots, \mathbf{c}^{(\boldsymbol{k})} \rangle$ of k centers, a clustering $\langle S^{(1)}, S^{(2)}, \ldots, S^{(k)} \rangle$ minimizes

$$\sum_{\ell=1}^{k} \sum_{\mathbf{x} \in S^{(\ell)}} \Delta(\mathbf{x}, \mathbf{c}^{(\boldsymbol{\ell})}) \tag{33.4}$$

if and only if it assigns each point $\mathbf{x} \in S$ to a cluster $S^{(\ell)}$ that minimizes $\Delta(\mathbf{x}, \mathbf{c}^{(\boldsymbol{\ell})})$.

Proof The proof is straightforward: each point $\mathbf{x} \in S$ contributes exactly once to the sum (33.4), and choosing to put $\mathbf{x}$ in a cluster whose center is nearest minimizes the contribution from $\mathbf{x}$. ■

Lloyd's procedure

Lloyd's procedure just iterates two operations—assigning points to clusters based on the nearest-center rule, followed by recomputing the centers of clusters to be their centroids—until the results converge. Here is Lloyd's procedure:

Input: A set S of points in $\mathbb{R}^d$, and a positive integer k.

Output: A k-clustering $\langle S^{(1)}, S^{(2)}, \ldots, S^{(k)} \rangle$ of S with a sequence of centers $\langle \mathbf{c}^{(1)}, \mathbf{c}^{(2)}, \ldots, \mathbf{c}^{(k)} \rangle$.

1. **Initialize centers:** Generate an initial sequence $\langle \mathbf{c}^{(1)}, \mathbf{c}^{(2)}, \ldots, \mathbf{c}^{(k)} \rangle$ of k centers by picking k points independently from S at random. (If the points are not necessarily distinct, see Exercise 33.1-3.) Assign all points to cluster $S^{(1)}$ to begin.
2. **Assign points to clusters:** Use the nearest-center rule to define the clustering $\langle S^{(1)}, S^{(2)}, \ldots, S^{(k)} \rangle$. That is, assign each point $\mathbf{x} \in S$ to a cluster $S^{(\ell)}$ having a nearest center (breaking ties arbitrarily, but not changing the assignment for a point $\mathbf{x}$ unless the new cluster center is strictly closer to $\mathbf{x}$ than the old one).
3. **Stop if no change:** If step 2 did not change the assignments of any points to clusters, then stop and return the clustering $\langle S^{(1)}, S^{(2)}, \ldots, S^{(k)} \rangle$ and the associated centers $\langle \mathbf{c}^{(1)}, \mathbf{c}^{(2)}, \ldots, \mathbf{c}^{(k)} \rangle$. Otherwise, go to step 4.
4. **Recompute centers as centroids:** For $\ell = 1, 2, \ldots, k$, compute the center $\mathbf{c}^{(\ell)}$ of cluster $S^{(\ell)}$ as the centroid of the points in $S^{(\ell)}$. (If $S^{(\ell)}$ is empty, let $\mathbf{c}^{(\ell)}$ be the zero vector.) Then go to step 2.

It is possible for some of the clusters returned to be empty, particularly if many of the input points are identical.

Lloyd's procedure always terminates. By Theorem 33.1, recomputing the centers of each cluster as the cluster centroid cannot increase $f(S, C)$. Lloyd's procedure ensures that a point is reassigned to a different cluster only when such an operation strictly decreases $f(S, C)$. Thus each iteration of Lloyd's procedure, except the last iteration, must strictly decrease $f(S, C)$. Since there are only a finite number of possible k-clusterings of S (at most k^n), the procedure must terminate. Furthermore, once one iteration of Lloyd's procedure yields no decrease in f, further iterations would not change anything, and the procedure can stop at this locally optimum assignment of points to clusters.

If Lloyd's procedure really required k^n iterations, it would be impractical. In practice, it sometimes suffices to terminate the procedure when the percentage decrease in $f(S, C)$ in the latest iteration falls below a predetermined threshold. Because Lloyd's procedure is guaranteed to find only a locally optimal clustering, one approach to finding a good clustering is to run Lloyd's procedure many times with different randomly chosen initial centers, taking the best result.

The running time of Lloyd's procedure is proportional to the number T of iterations. In one iteration, assigning points to clusters based on the nearest-center rule requires $O(dkn)$ time, and recomputing new centers for each cluster requires $O(dn)$ time (because each point is in one cluster). The overall running time of the k-means procedure is thus $O(Tdkn)$.

Lloyd's algorithm illustrates an approach common to many machine-learning algorithms:

- First, define a hypothesis space in terms an appropriate sequence θ of parameters, so that each θ is associated with a specific hypothesis h_θ. (For the k-means problem, θ is a dk-dimensional vector, equivalent to C, containing the d-dimensional center of each of the k clusters, and h_θ is the hypothesis that each data point $\mathbf{x}$ should be grouped with a cluster having a center closest to $\mathbf{x}$.)
- Second, define a measure $f(E, \theta)$ describing how poorly hypothesis h_θ fits the given training data E. Smaller values of $f(E, \theta)$ are better, and a (locally) optimal solution (locally) minimizes $f(E, \theta)$. (For the k-means problem, $f(E, \theta)$ is just $f(S, C)$.)
- Third, given a set of training data E, use a suitable optimization procedure to find a value of θ^* that minimizes $f(E, \theta^*)$, at least locally. (For the k-means problem, this value of θ^* is the sequence C of k center points returned by Lloyd's algorithm.)
- Return θ^* as the answer.

In this framework, we see that optimization becomes a powerful tool for machine learning. Using optimization in this way is flexible. For example, ***regularization*** terms can be incorporated in the function to be minimized, in order to penalize hypotheses that are "too complicated" and that "overfit" the training data. (Regularization is a complex topic that isn't pursued further here.)

Examples

Figure 33.1 demonstrates Lloyd's procedure on a set of $n = 49$ cities: 48 U.S. state capitals and the District of Columbia. Each city has $d = 2$ dimensions: latitude and longitude. The initial clustering in part (a) of the figure has the initial cluster centers arbitrarily chosen as the capitals of Arkansas, Kansas, Louisiana,

and Tennessee. As the procedure iterates, the value of the function f decreases, until the 11th iteration in part (l), where it remains the same as in the 10th iteration in part (k). Lloyd's procedure then terminates with the clusters shown in part (l).

As Figure 33.2 shows, Lloyd's procedure can also apply to "vector quantization." Here, the goal is to reduce the number of distinct colors required to represent a photograph, thereby allowing the photograph to be greatly compressed (albeit in a lossy manner). In part (a) of the figure, an original photograph 700 pixels wide and 500 pixels high uses 24 bits (three bytes) per pixel to encode a triple of red, green, and blue (RGB) primary color intensities. Parts (b)–(e) of the figure show the results of using Lloyd's procedure to compress the picture from a initial space of 2^{24} possible values per pixel to a space of only $k = 4, k = 16, k = 64$, or $k = 256$ possible values per pixel; these k values are the cluster centers. The photograph can then be represented with only 2, 4, 6, or 8 bits per pixel, respectively, instead of the 24-bits per pixel needed by the initial photograph. An auxiliary table, the "palette," accompanies the compressed image; it holds the k 24-bit cluster centers and is used to map each pixel value to its 24-bit cluster center when the photo is decompressed.

Exercises

33.1-1

Show that the objective function $f(S, C)$ of equation (33.2) may be alternatively written as

$$f(S, C) = \sum_{\ell=1}^{k} \frac{1}{2\,|S^{(\ell)}|} \sum_{\mathbf{x} \in S^{(\ell)}} \sum_{\mathbf{y} \in S^{(\ell)} : \mathbf{x} \neq \mathbf{y}} \Delta(\mathbf{x}, \mathbf{y}) \; .$$

33.1-2

Give an example in the plane with $n = 4$ points and $k = 2$ clusters where an iteration of Lloyd's procedure does not improve $f(S, C)$, yet the k-clustering is not optimal.

33.1-3

When the input to Lloyd's procedure contains many repeated points, a different initialization procedure might be used. Describe a way to pick a number of centers at random that maximizes the number of distinct centers picked. (*Hint:* See Exercise 5.3-5.)

33.1-4

Show how to find an optimal k-clustering in polynomial time when there is just one attribute ($d = 1$).

(a) Original

(b) $k = 4$ ($f = 1.29 \times 10^9$; 31 iterations)

(c) $k = 16$ ($f = 3.31 \times 10^8$; 36 iterations)

(d) $k = 64$ ($f = 5.50 \times 10^7$; 59 iterations)

(e) $k = 256$ ($f = 1.52 \times 10^7$; 104 iterations)

Figure 33.2 Using Lloyd's procedure for vector quantization to compress a photo by using fewer colors. **(a)** The original photo has 350,000 pixels (700×500), each a 24-bit RGB (red/blue/green) triple of 8-bit values; these pixels (colors) are the "points" to be clustered. Points repeat, so there are only 79,083 distinct colors (less than 2^{24}). After compression, only k distinct colors are used, so each pixel is represented by only $\lceil \lg k \rceil$ bits instead of 24. A "palette" maps these values back to 24-bit RGB values (the cluster centers). **(b)–(e)** The same photo with $k = 4, 16, 64$, and 256 colors. (Photo from standuppaddle, pixabay.com.)

33.2 Multiplicative-weights algorithms

This section considers problems that require you to make a series of decisions. After each decision you receive feedback as to whether your decision was correct. We will study a class of algorithms that are called ***multiplicative-weights algorithms***. This class of algorithms has a wide variety of applications, including game playing in economics, approximately solving linear-programming and multicommodity-flow problems, and various applications in online machine learning. We emphasize the online nature of the problem here: you have to make a sequence of decisions, but some of the information needed to make the ith decision appears only after you have already made the $(i-1)$st decision. In this section, we look at one particular problem, known as "learning from experts," and develop an example of a multiplicative-weights algorithm, called the weighted-majority algorithm.

Suppose that a series of events will occur, and you want to make predictions about these events. For example, over a series of days, you want to predict whether it is going to rain. Or perhaps you want to predict whether the price of a stock will increase or decrease. One way to approach this problem is to assemble a group of "experts" and use their collective wisdom in order to make good predictions. Let's denote the experts, n of them, by $E_1, E_2, \ldots, E_n$, and let's say that T events are going to take place. Each event has an outcome of either 0 or 1, with $o^{(t)}$ denoting the outcome of the tth event. Before event t, each expert $E^{(i)}$ makes a prediction $q_i^{(t)} \in \{0, 1\}$. You, as the "learner," then take the set of n expert predictions for event t and produce a single prediction $p^{(t)} \in \{0, 1\}$ of your own. You base your prediction only on the predictions of the experts and anything you have learned about the experts from their previous predictions. You do not use any additional information about the event. Only after making your prediction do you ascertain the outcome $o^{(t)}$ of event t. If your prediction $p^{(t)}$ matches $o^{(t)}$, then you were correct; otherwise, you made a mistake. The goal is to minimize the total number m of mistakes, where $m = \sum_{t=1}^{T} \left| p^{(t)} - o^{(t)} \right|$. You can also keep track of the number of mistakes each expert makes: expert E_i makes m_i mistakes, where $m_i = \sum_{t=1}^{T} \left| q_i^{(t)} - o^{(t)} \right|$.

For example, suppose that you are following the price of a stock, and each day you decide whether to invest in it for just that day by buying it at the beginning of the day and selling it at the end of the day. If, on some day, you buy the stock and it goes up, then you made the correct decision, but if the stock goes down, then you made a mistake. Similarly, if on some day, you do not buy the stock and it goes down, then you made the correct decision, but if the stock goes up, then you made a mistake. Since you would like to make as few mistakes as possible, you use the advice of the experts to make your decisions.

We'll assume nothing about the movement of the stock. We'll also assume nothing about the experts: the experts' predictions could be correlated, they could be chosen to deceive you, or perhaps some are not really experts after all. What algorithm would you use?

Before designing an algorithm for this problem, we need to consider what is a fair way to evaluate our algorithm. It is reasonable to expect that our algorithm performs better when the expert predictions are better, and that it performs worse when the expert predictions are worse. The goal of the algorithm is to limit the number of mistakes you make to be close to the number of mistakes that the best of the experts makes. At first, this goal might seem impossible, because you do not know until the end which expert is best. We'll see, however, that by taking the advice provided by all the experts into account, you can achieve this goal. More formally, we use the notion of "regret," which compares our algorithm to the performance of the best expert (in hindsight) over all. Letting $m^* = \min\{m_i : 1 \le i \le n\}$ denote the number of mistakes made by the best expert, the ***regret*** is $m - m^*$. The goal is to design an algorithm with low regret. (Regret can be negative, although it typically isn't, since it is rare that you do better than the best expert.)

As a warm-up, let's consider the case in which one of the experts makes a correct prediction each time. Even without knowing who that expert is, you can still achieve good results.

Lemma 33.3

Suppose that out of n experts, there is one who always makes the correct prediction for all T events. Then there is an algorithm that makes at most $\lceil \lg n \rceil$ mistakes.

Proof The algorithm maintains a set S consisting of experts who have not yet made a mistake. Initially, S contains all n experts. The algorithm's prediction is always the majority vote of the predictions of the experts remaining in set S. In case of a tie, the algorithm makes any prediction. After each outcome is learned, set S is updated to remove all the experts who made an incorrect prediction about that outcome.

We now analyze the algorithm. The expert who always makes the correct prediction will always be in set S. Every time the algorithm makes a mistake, at least half of the experts who were still in S also make a mistake, and these experts are removed from S. If S' is the set of experts remaining after removing those who made a mistake, we have that $|S'| \le |S|/2$. The size of S can be halved at most $\lceil \lg n \rceil$ times until $|S| = 1$. From this point on, we know that the algorithm never makes a mistake, since the set S consists only of the one expert who never makes a mistake. Therefore, overall the algorithm makes at most $\lceil \lg n \rceil$ mistakes. ■

Exercise 33.2-1 asks you to generalize this result to the case when there is no expert who makes perfect predictions and show that, for any set of experts, there is an algorithm that makes at most $m^* \lceil \lg n \rceil$ mistakes. The generalized algorithm begins in the same way. The set S might become empty at some point, however. If that ever happens, reset S to contain all the experts and continue the algorithm.

You can substantially improve your prediction ability by not just tracking which experts have not made any mistakes, or have not made any mistakes recently, to a more nuanced evaluation of the quality of each expert. The key idea is to use the feedback you receive to update your evaluation of how much trust to put in each expert. As the experts make predictions, you observe whether they were correct and decrease your confidence in the experts who make more mistakes. In this way, you can learn over time which experts are more reliable and which are less reliable, and weight their predictions accordlingly. The change in weights is accomplished via multiplication, hence the term "multiplicative weights."

The algorithm appears in the procedure WEIGHTED-MAJORITY on the following page, which takes a set $E = \{E_1, E_2, \ldots, E_n\}$ of experts, a number T of events, the number n of experts, and a parameter $0 < \gamma \le 1/2$ that controls how the weights change. The algorithm maintains weights $w_i^{(t)}$ for $i = 1, 2, \ldots, n$ and $t = 1, 2, \ldots, T$, where $0 < w_i^{(t)} \le 1$. The **for** loop of lines 1–2 sets the initial weights $w_i^{(1)}$ to 1, capturing the idea that with no knowledge, you trust each expert equally. Each iteration of the main **for** loop of lines 3–18 does the following for an event $t = 1, 2, \ldots, T$. Each expert E_i makes a prediction for event t in line 4. Lines 5–8 compute $upweight^{(t)}$, the sum of the weights of the experts who predict 1 for event t, and $downweight^{(t)}$, the sum of the weights of the experts who predict 0 for the event. Lines 9–11 decide the algorithm's prediction $p^{(t)}$ for event t based on whichever weighted sum is larger (breaking ties in favor of deciding 1). The outcome of event t is revealed in line 12. Finally, lines 14–17 decrease the weights of the experts who made an incorrect prediction for event t by multiplying their weights by $1 - \gamma$, leaving alone the weights of the experts who correctly predicted the event's outcome. Thus, the fewer mistakes each expert makes, the higher that expert's weight.

The WEIGHTED-MAJORITY procedure doesn't do much worse than any expert. In particular, it doesn't do much worse than the best expert. To quantify this claim, let $m^{(t)}$ be the number of mistakes made by the procedure through event t, and let $m_i^{(t)}$ be the number of mistakes made by expert E_i through event t. The following theorem is the key.

```
WEIGHTED-MAJORITY(E, T, n, γ)
1  for i = 1 to n
2      w_i^(1) = 1                          // trust each expert equally
3  for t = 1 to T
4      each expert E_i ∈ E makes a prediction q_i^(t)
5      U = {E_i : q_i^(t) = 1}              // experts who predicted 1
6      upweight^(t) = Σ_{i:E_i∈U} w_i^(t)   // sum of weights of who predicted 1
7      D = {E_i : q_i^(t) = 0}              // experts who predicted 0
8      downweight^(t) = Σ_{i:E_i∈D} w_i^(t) // sum of weights of who predicted 0
9      if upweight^(t) ≥ downweight^(t)
10         p^(t) = 1                        // algorithm predicts 1
11     else p^(t) = 0                       // algorithm predicts 0
12     outcome o^(t) is revealed
13     // If p^(t) ≠ o^(t), the algorithm made a mistake.
14     for i = 1 to n
15         if q_i^(t) ≠ o^(t)               // if expert E^(i) made a mistake ...
16             w_i^(t+1) = (1 − γ) w_i^(t)  // ... then decrease that expert's weight
17         else w_i^(t+1) = w_i^(t)
18     return p^(t)
```

Theorem 33.4
When running WEIGHTED-MAJORITY, we have, for every expert E_i and every event $T' \le T$,

$$m^{(T')} \le 2(1+\gamma)m_i^{(T')} + \frac{2\ln n}{\gamma} . \tag{33.5}$$

Proof Every time an expert E_i makes a mistake, its weight, which is initially 1, is multiplied by $1-\gamma$, and so we have

$$w_i^{(t)} = (1-\gamma)^{m_i^{(t)}} \tag{33.6}$$

for $t = 1, 2, \ldots, T$.

We use a potential function $W(t) = \sum_{i=1}^{n} w_i^{(t)}$, summing the weights for all n experts after iteration t of the **for** loop of lines 3–18. Initially, we have $W(0) = n$ since all n weights start out with the value 1. Because each expert belongs to either the set U or the set D (defined in lines 5 and 7 of WEIGHTED-MAJORITY), we always have $W(t) = upweight^{(t)} + downweight^{(t)}$ after each execution of line 8.

Consider an iteration t in which the algorithm makes a mistake in its prediction, which means that either the algorithm predicts 1 and the outcome is 0 or the al-

gorithm predicts 0 and the outcome is 1. Without loss of generality, assume that the algorithm predicts 1 and the outcome is 0. The algorithm predicted 1 because $upweight^{(t)} \geq downweight^{(t)}$ in line 9, which implies that

$$upweight^{(t)} \geq W(t)/2 \; . \tag{33.7}$$

Each expert in U then has its weight multiplied by $1-\gamma$, and each expert in D has its weight unchanged. Thus, we have

$$\begin{aligned} W(t+1) &= upweight^{(t)}(1-\gamma) + downweight^{(t)} \\ &= upweight^{(t)} + downweight^{(t)} - \gamma \cdot upweight^{(t)} \\ &= W(t) - \gamma \cdot upweight^{(t)} \\ &\leq W(t) - \gamma \, \frac{W(t)}{2} \qquad \text{(by inequality (33.7))} \\ &= W(t)(1-\gamma/2) . \end{aligned}$$

Therefore, for every iteration t in which the algorithm makes a mistake, we have

$$W(t+1) \leq (1-\gamma/2)W(t) . \tag{33.8}$$

In an iteration where the algorithm does not make a mistake, some of the weights decrease and some remain unchanged, so that we have

$$W(t+1) \leq W(t) \; . \tag{33.9}$$

Since there are $m^{(T')}$ mistakes made through iteration T', and $W(1) = n$, we can repeatedly apply inequality (33.8) to iterations where the algorithm makes a mistake and inequality (33.9) to iterations where the algorithm does not make a mistake, obtaining

$$W(T') \leq n(1-\gamma/2)^{m^{(T')}} \; . \tag{33.10}$$

Because the function W is the sum of the weights and all weights are positive, its value exceeds any single weight. Therefore, using equation (33.6) we have, for any expert E_i and for any iteration $T' \leq T$,

$$W(T') \geq w_i^{(T')} = (1-\gamma)^{m_i^{(T')}} \; . \tag{33.11}$$

Combining inequalities (33.10) and (33.11) gives

$$(1-\gamma)^{m_i^{(T')}} \leq n(1-\gamma/2)^{m^{(T')}} \; .$$

Taking the natural logarithm of both sides yields

$$m_i^{(T')} \ln(1-\gamma) \leq m^{(T')} \ln(1-\gamma/2) + \ln n \; . \tag{33.12}$$

We now use the Taylor series expansion to derive upper and lower bounds on the logarithmic factors in inequality (33.12). The Taylor series for $\ln(1+x)$ is given in equation (3.22) on page 67. Substituting $-x$ for x, we have that for $0 < x \le 1/2$,

$$\ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \cdots . \tag{33.13}$$

Since each term on the right-hand side is negative, we can drop all terms except the first and obtain an upper bound of $\ln(1-x) \le -x$. Since $0 < \gamma \le 1/2$, we have

$$\ln(1-\gamma/2) \le -\gamma/2 \tag{33.14}$$

For the lower bound, Exercise 33.2-2 asks you to show that $\ln(1-x) \ge -x - x^2$ when $0 < x \le 1/2$, so that

$$-\gamma - \gamma^2 \le \ln(1-\gamma) . \tag{33.15}$$

Thus, we have

$$\begin{aligned} m_i^{(T')}(-\gamma-\gamma^2) &\le m_i^{(T')} \ln(1-\gamma) && \text{(by inequality (33.15))} \\ &\le m^{(T')} \ln(1-\gamma/2) + \ln n && \text{(by inequality (33.12)} \\ &\le m^{(T')}(-\gamma/2) + \ln n && \text{(by inequality (33.14))} , \end{aligned}$$

so that

$$m_i^{(T')}(-\gamma-\gamma^2) \le m^{(T')}(-\gamma/2) + \ln n . \tag{33.16}$$

Subtracting $\ln n$ from both sides of inequality (33.16) and then multiplying both sides by $-2/\gamma$ yields $m^{(T')} \le 2(1+\gamma)m_i^{(T')} + (2\ln n)/\gamma$, thus proving the theorem. ■

Theorem 33.4 applies to any expert and any event $T' \le T$. In particular, we can compare against the best expert after all events have occurred, producing the following corollary.

Corollary 33.5
At the end of procedure WEIGHTED-MAJORITY, we have

$$m^{(T)} \le 2(1+\gamma)m^* + \frac{2\ln n}{\gamma} . \tag{33.17}$$

■

Let's explore this bound. Assuming that $\sqrt{\ln n/m^*} \le 1/2$, we can choose $\gamma = \sqrt{\ln n/m^*}$ and plug into inequality (33.17) to obtain

$$
\begin{aligned}
m^{(T)} &\le 2\left(1+\sqrt{\frac{\ln n}{m^*}}\right)m^* + \frac{2\ln n}{\sqrt{\ln n/m^*}} \\
&= 2m^* + 2\sqrt{m^*\ln n} + 2\sqrt{m^*\ln n} \\
&= 2m^* + 4\sqrt{m^*\ln n}\ ,
\end{aligned}
$$

and so the number of errors is at most twice the number of errors made by the best expert plus a term that is often slower growing than m^*. Exercise 33.2-4 shows that you can decrease the bound on the number of errors by a factor of 2 by using randomization, which leads to much stronger bounds. In particular, the upper bound on regret $(m - m^*)$ is reduced from $(1+2\gamma)m^* + (2\ln n)/\gamma$ to an expected value of $\epsilon m^* + (\ln n)/\epsilon$, where both γ and ϵ are at most $1/2$. Numerically, we can see that if $\gamma = 1/2$, WEIGHTED-MAJORITY makes at most 3 times the number of errors as the best expert, plus $4\ln n$ errors. As another example, suppose that $T = 1000$ predictions are being made by $n = 20$ experts, and the best expert is correct 95% of the time, making 50 errors. Then WEIGHTED-MAJORITY makes at most $100(1+\gamma)+2\ln 20/\gamma$ errors. By choosing $\gamma = 1/4$, WEIGHTED-MAJORITY makes at most 149 errors, or a success rate of at least 85%.

Multiplicative weights methods typically refer to a broader class of algorithms that includes WEIGHTED-MAJORITY. The outcomes and predictions need not be only 0 or 1, but can be real numbers, and there can be a loss associated with a particular outcome and prediction. The weights can be updated by a multiplicative factor that depends on the loss, and the algorithm can, given a set of weights, treat them as a distribution on experts and use them to choose an expert to follow in each event. Even in these more general settings, bounds similar to Theorem 33.4 hold.

Exercises

33.2-1

The proof of Lemma 33.3 assumes that some expert never makes a mistake. It is possible to generalize the algorithm and analysis to remove this assumption. The new algorithm begins in the same way. The set S might become empty at some point, however. If that ever happens, reset S to contain all the experts and continue the algorithm. Show that the number of mistakes that this algorithm makes is at most $m^* \lceil \lg n \rceil$.

33.2-2

Show that $\ln(1-x) \ge -x - x^2$ when $0 < x \le 1/2$. (*Hint:* Start with equation (33.13), group all the terms after the first three, and use equation (A.7) on page 1142.)

33.2-3

Consider a randomized variant of the algorithm given in the proof of Lemma 33.3, in which some expert never makes a mistake. At each step, choose an expert E_i uniformly at random from the set S and then make the same predication as E_i. Show that the expected number of mistakes made by this algorithm is $\lceil \lg n \rceil$.

33.2-4

Consider a randomized version of WEIGHTED-MAJORITY. The algorithm is the same, except for the prediction step, which interprets the weights as a probability distribution over the experts and chooses an expert E_i according to that distribution. It then chooses its prediction to be the same as the prediction made by expert E_i. Show that, for any $0 < \epsilon < 1/2$, the expected number of mistakes made by this algorithm is at most $(1 + \epsilon)m^* + (\ln n)/\epsilon$.

33.3 Gradient descent

Suppose that you have a set $\{p_1, p_2, \ldots, p_n\}$ of points and you want to find the line that best fits these points. For any line ℓ, there is a distance d_i between each point p_i and the line. You want to find the line that minimizes some function $f(d_1, \ldots, d_n)$. There are many possible choices for the definition of distance and for the function f. For example, the distance can be the projection distance to the line and the function can be the sum of the squares of the distances. This type of problem is common in data science and machine learning—the line is the hypothesis that best describes the data—where the particular definition of best is determined by the definition of distance and the objective f. If the definition of distance and the function f are linear, then we have a linear-programming problem, as discussed in Chapter 29. Although the linear-programming framework captures several important problems, many other problems, including various machine-learning problems, have objectives and constraints that are not necessarily linear. We need frameworks and algorithms to solve such problems.

In this section, we consider the problem of optimizing a continuous function and discuss one of the most popular methods to do so: gradient descent. Gradient descent is a general method for finding a local minimum of a function $f : \mathbb{R}^n \to \mathbb{R}$, where informally, a local minimum of a function f is a point $\mathbf{x}$ for which $f(\mathbf{x}) \le f(\mathbf{x}')$ for all $\mathbf{x}'$ that are "near" $\mathbf{x}$. When the function is convex, it can find a point near the ***global minimizer*** of f: an n-vector argument $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ such that $f(\mathbf{x})$ is minimum. For the intuitive idea behind gradient descent, imagine being in a landscape of hills and valleys, and wanting to get to a low point as quickly as possible. You survey the terrain and choose to

move in the direction that takes you downhill the fastest from your current position. You move in that direction, but only for a short while, because as you proceed, the terrain changes and you might need to choose a different direction. So you stop, reevaluate the possible directions and move another short distance in the steepest downhill direction, which might differ from the direction of your previous movement. You continue this process until you reach a point from which all directions lead up. Such a point is a local minimum.

In order to make this informal procedure more formal, we need to define the gradient of a function, which in the analogy above is a measure of the steepness of the various directions. Given a function $f : \mathbb{R}^n \to \mathbb{R}$, its ***gradient*** ∇f is a function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ comprising n partial derivatives: $(\nabla f)(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$. Analogous to the derivative of a function of a single variable, the gradient can be viewed as a direction in which the function value locally increases the fastest, and the rate of that increase. This view is informal; in order to make it formal we would have to define what local means and place certain conditions, such as continuity or existence of derivatives, on the function. Nevertheless, this view motivates the key step of gradient descent—move in the direction opposite to the gradient, by a distance influenced by the magnitude of the gradient.

The general procedure of gradient descent proceeds in steps. You start at some initial point $\mathbf{x}^{(0)}$, which is an n-vector. At each step t, you compute the value of the gradient of f at point $\mathbf{x}^{(t)}$, that is, $(\nabla f)(\mathbf{x}^{(t)})$, which is also an n-vector. You then move in the direction opposite to the gradient in each dimension at $\mathbf{x}^{(t)}$ to arrive at the next point $\mathbf{x}^{(t+1)}$, which again is an n-vector. Because you moved in a monotonically decreasing direction in each dimension, you should have that $f(\mathbf{x}^{(t+1)}) \le f(\mathbf{x}^{(t)})$. Several details are needed to turn this idea into an actual algorithm. The two main details are that you need an initial point and that you need to decide how far to move in the direction of the negative gradient. You also need to understand when to stop and what you can conclude about the quality of the solution found. We will explore these issues further in this section, for both constrained minimization, where there are additional constraints on the points, and unconstrained minimization, where there are none.

Unconstrained gradient descent

In order to gain intuition, let's consider unconstrained gradient descent in just one dimension, that is, when f is a function of a scalar x, so that $f : \mathbb{R} \to \mathbb{R}$. In this case, the gradient ∇f of f is just $f'(x)$, the derivative of f with respect to x. Consider the function f shown in blue in Figure 33.3, with minimizer x^* and starting point $x^{(0)}$. The gradient (derivative) $f'(x^{(0)})$, shown in orange, has a negative slope, so that a small step from $x^{(0)}$ in the direction of increasing x results in a point x' for which $f(x') < f(x^{(0)})$. Too large a step, however, results in a

Figure 33.3 A function $f : \mathbb{R} \to \mathbb{R}$, shown in blue. Its gradient at point $x^{(0)}$, in orange, has a negative slope, and so a small increase in x from $x^{(0)}$ to x' results in $f(x') < f(x^{(0)})$. Small increases in x from $x^{(0)}$ head toward $\hat{x}$, which gives a local minimum. Too large an increase in x can end up at x'', where $f(x'') > f(x^{(0)})$. Small steps starting from $x^{(0)}$ and going only in the direction of decreasing values of f cannot end up at the global minimizer x^*.

point x'' for which $f(x'') > f(x^{(0)})$, so this is a bad idea. Restricting ourselves to small steps, where each one has $f(x') < f(x)$, eventually results in getting close to point $\hat{x}$, which gives a local minimum. By taking only small downhill steps, however, gradient descent has no chance to get to the global minimizer x^*, given the starting point $x^{(0)}$.

We draw two observations from this simple example. First, gradient descent converges toward a local minimum, and not necessarily a global minimum. Second, the speed at which it converges and how it behaves are related to properties of the function, to the initial point, and to the step size of the algorithm.

The procedure GRADIENT-DESCENT on the facing page takes as input a function f, an initial point $\mathbf{x}^{(0)} \in \mathbb{R}^n$, a fixed step-size multiplier $\gamma > 0$, and a number $T > 0$ of steps to take. Each iteration of the **for** loop of lines 2–4 performs a step by computing the n-dimensional gradient at point $\mathbf{x}^{(t)}$ and then moving distance γ in the opposite direction in the n-dimensional space. The complexity of computing the gradient depends on the function f and can sometimes be expensive. Line 3 sums the points visited. After the loop terminates, line 6 returns **x-avg**, the average of all the points visited except for the last one, $\mathbf{x}^{(T)}$. It might seem more natural to return $\mathbf{x}^{(T)}$, and in fact, in many circumstances, you might prefer to have the function return $\mathbf{x}^{(T)}$. For the version we will analyze, however, we use **x-avg**.

```
GRADIENT-DESCENT(f, x^(0), γ, T)
sum = 0                                  // n-dimensional vector, initially all 0
for t = 0 to T − 1
    sum = sum + x^(t)                    // add each of n dimensions into sum
    x^(t+1) = x^(t) − γ · (∇f)(x^(t))    // (∇f)(x^(t)), x^(t+1) are n-dimensional
x-avg = sum/T                            // divide each of n dimensions by T
return x-avg
```

Figure 33.4 depicts how gradient descent ideally runs on a convex 1-dimensional function.[1] We'll define convexity more formally below, but the figure shows that each iteration moves in the direction opposite to the gradient, with the distance moved being proportional to the magnitude of the gradient. As the iterations proceed, the magnitude of the gradient decreases, and thus the distance moved along the horizontal axis decreases. After each iteration, the distance to the optimal point $\mathbf{x}^*$ decreases. This ideal behavior is not guaranteed to occur in general, but the analysis in the remainder of this section formalizes when this behavior occurs and quantifies the number of iterations needed. Gradient descent does not always work, however. We have already seen that if the function is not convex, gradient descent can converge to a local, rather than global, minimum. We have also seen that if the step size is too large, GRADIENT-DESCENT can overshoot the minimum and wind up farther away. (It is also possible to overshoot the minimum and wind up closer to the optimum.)

Analysis of unconstrained gradient descent for convex functions

Our analysis of gradient descent focuses on convex functions. Inequality (C.29) on page 1194 defines a convex function of one variable, as shown in Figure 33.5. We can extend that definition to a function $f : \mathbb{R}^n \to \mathbb{R}$ and say that f is ***convex*** if for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and for all $0 \le \lambda \le 1$, we have

$$f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \le \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y}) . \tag{33.18}$$

(Inequalities (33.18) and (C.29) are the same, except for the dimensions of $\mathbf{x}$ and $\mathbf{y}$.) We also assume that our convex functions are closed[2] and differentiable.

---

[1] Although the curve in Figure 33.4 looks concave, according to the definition of convexity that we'll see below, the function f in the figure is convex.

[2] A function $f : \mathbb{R}^n \to \mathbb{R}$ is closed if, for each $\alpha \in \mathbb{R}$, the set $\{\mathbf{x} \in \mathrm{dom}(f) : f(\mathbf{x}) \le \alpha\}$ is closed, where $\mathrm{dom}(f)$ is the domain of f.

Figure 33.4 An example of running gradient descent on a convex function $f : \mathbb{R} \to \mathbb{R}$, shown in blue. Beginning at point $\mathbf{x}^{(0)}$, each iteration moves in the direction opposite to the gradient, and the distance moved is proportional to the magnitude of the gradient. Orange lines represent the negative of the gradient at each point, scaled by the step size γ. As the iterations proceed, the magnitude of the gradient decreases, and the distance moved decreases correspondingly. After each iteration, the distance to the optimal point $\mathbf{x}^*$ decreases.

Figure 33.5 A convex function $f : \mathbb{R} \to \mathbb{R}$, shown in blue, with local and global minimizer $\mathbf{x}^*$. Because f is convex, $f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \le \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y})$ for any two values $\mathbf{x}$ and $\mathbf{y}$ and all $0 \le \lambda \le 1$, shown for a particular value of λ. Here, the orange line segment represents all values $\lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y})$ for $0 \le \lambda \le 1$, and it is above the blue line.

A convex function has the property that any local minimum is also a global minimum. To verify this property, consider inequality (33.18), and suppose for the purpose of contradiction that $\mathbf{x}$ is a local minimum but not a global minimum and $\mathbf{y} \ne \mathbf{x}$ is a global minimum, so $f(\mathbf{y}) < f(\mathbf{x})$. Then we have

$$\begin{aligned} f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) &\le \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y}) \quad \text{(by inequality (33.18))} \\ &< \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{x}) \\ &= f(\mathbf{x}) \,. \end{aligned}$$

Thus, letting λ approach 1, we see that there is another point near $\mathbf{x}$, say $\mathbf{x}'$, such that $f(\mathbf{x}') < f(\mathbf{x})$, so $\mathbf{x}$ is not a local minimum.

Convex functions have several useful properties. The first property, whose proof we leave as Exercise 33.3-1, says that a convex function always lies above its tangent hyperplane. In the context of gradient descent, angle brackets denote the notation for inner product defined on page 1219 rather than denoting a sequence.

Lemma 33.6
For any convex differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ and for all $x, y \in \mathbb{R}^n$, we have $f(\mathbf{x}) \le f(\mathbf{y}) + \langle (\nabla f)(\mathbf{x}), \mathbf{x} - \mathbf{y} \rangle$. ■

The second property, which Exercise 33.3-2 asks you to prove, is a repeated application of the definition of convexity in inequality (33.18).

Lemma 33.7
For any convex function $f : \mathbb{R}^n \to \mathbb{R}$, for any integer $T \ge 1$, and for all $\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(T-1)} \in \mathbb{R}^n$, we have

$$f\left(\frac{\mathbf{x}^{(0)} + \cdots + \mathbf{x}^{(T-1)}}{T}\right) \le \frac{f(\mathbf{x}^{(0)}) + \cdots + f(\mathbf{x}^{(T-1)})}{T} . \tag{33.19}$$

■

The left-hand side of inequality (33.19) is the value of f at the vector **x-avg** that GRADIENT-DESCENT returns.

We now proceed to analyze GRADIENT-DESCENT. It might not return the exact global minimizer $\mathbf{x}^*$. We use an error bound ϵ, and we want to choose T so that $f(\textbf{x-avg}) - f(\mathbf{x}^*) \le \epsilon$ at termination. The value of ϵ depends on the number T of iterations and two additional values. First, since you expect it to be better to start close to the global minimizer, ϵ is a function of

$$R = \|\mathbf{x}^{(0)} - \mathbf{x}^*\| , \tag{33.20}$$

the euclidean norm (or distance, defined on page 1219) of the difference between $\mathbf{x}^{(0)}$ and $\mathbf{x}^*$. The error bound ϵ is also a function of a quantity we call L, which is an upper bound on the magnitude $\|(\nabla f)(\mathbf{x})\|$ of the gradient, so that

$$\|(\nabla f)(\mathbf{x})\| \le L , \tag{33.21}$$

where $\mathbf{x}$ ranges over all the points $\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(T-1)}$ whose gradients are computed by GRADIENT-DESCENT. Of course, we don't know the values of L and R, but for now let's assume that we do. We'll discuss later how to remove these assumptions. The analysis of GRADIENT-DESCENT is summarized in the following theorem.

Theorem 33.8
Let $\mathbf{x}^* \in \mathbb{R}^n$ be the minimizer of a convex function f, and suppose that an execution of GRADIENT-DESCENT$(f, \mathbf{x}^{(0)}, \gamma, T)$ returns **x-avg**, where $\gamma = R/(L\sqrt{T})$ and R and L are defined in equations (33.20) and (33.21). Let $\epsilon = RL/\sqrt{T}$. Then we have $f(\textbf{x-avg}) - f(\mathbf{x}^*) \le \epsilon$. ■

We now prove this theorem. We do not give an absolute bound on how much progress each iteration makes. Instead, we use a potential function, as in Section 16.3. Here, we define a potential $\Phi(t)$ after computing $\mathbf{x}^{(t)}$, such that $\Phi(t) \ge 0$ for $t = 0, \ldots, T$. We define the ***amortized progress*** in the iteration that computes $\mathbf{x}^{(t)}$ as

$$p(t) = f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*) + \Phi(t+1) - \Phi(t) . \tag{33.22}$$

Along with including the change in potential $(\Phi(t+1) - \Phi(t))$, equation (33.22) also subtracts the minimum value $f(\mathbf{x}^*)$ because ultimately, you care not about the values $f(\mathbf{x}^{(t)})$ but about how close they are to $f(\mathbf{x}^*)$. Suppose that we can show that $p(t) \le B$ for some value B and $t = 0, \ldots, T-1$. Then we can substitute for $p(t)$ using equation (33.22), giving

$$f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*) \le B - \Phi(t+1) + \Phi(t) . \tag{33.23}$$

Summing inequality (33.23) over $t = 0, \ldots, T-1$ yields

$$\sum_{t=0}^{T-1} (f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)) \le \sum_{t=0}^{T-1} (B - \Phi(t+1) + \Phi(t)) .$$

Observing that we have a telescoping series on the right and regrouping terms, we have that

$$\left(\sum_{t=0}^{T-1} f(\mathbf{x}^{(t)}) \right) - T \cdot f(\mathbf{x}^*) \le TB - \Phi(T) + \Phi(0) .$$

Dividing by T and dropping the positive term $\Phi(T)$ gives

$$\frac{\sum_{t=0}^{T-1} f(\mathbf{x}^{(t)})}{T} - f(\mathbf{x}^*) \le B + \frac{\Phi(0)}{T} , \tag{33.24}$$

and thus we have

$$\begin{aligned} f(\textbf{x-avg}) - f(\mathbf{x}^*) &= f\left(\frac{\sum_{t=0}^{T-1} \mathbf{x}^{(t)}}{T} \right) - f(\mathbf{x}^*) && \text{(by the definition of } \textbf{x-avg}\text{)} \\ &\le \frac{\sum_{t=0}^{T-1} f(\mathbf{x}^{(t)})}{T} - f(\mathbf{x}^*) && \text{(by Lemma 33.7)} \\ &\le B + \frac{\Phi(0)}{T} && \text{(by inequality (33.24))} . \end{aligned} \tag{33.25}$$

In other words, if we can show that $p(t) \le B$ for some value B and choose a potential function where $\Phi(0)$ is not too large, then inequality (33.25) tells us how close the function value $f(\textbf{x-avg})$ is to the function value $f(\mathbf{x}^*)$ after T iterations. That is, we can set the error bound ϵ to $B + \Phi(0)/T$.

In order to bound the amortized progress, we need to come up with a concrete potential function. Define the potential function $\Phi(t)$ by

$$\Phi(t) = \frac{\left\|\mathbf{x}^{(t)} - \mathbf{x}^*\right\|^2}{2\gamma} , \tag{33.26}$$

that is, the potential function is proportional to the square of the distance between the current point and the minimizer $\mathbf{x}^*$. With this potential function in hand, the next lemma provides a bound on the amortized progress made in any iteration of GRADIENT-DESCENT.

Lemma 33.9
Let $\mathbf{x}^* \in \mathbb{R}^n$ be the minimizer of a convex function f, and consider an execution of GRADIENT-DESCENT$(f, \mathbf{x}^{(0)}, \gamma, T)$. Then for each point $\mathbf{x}^{(t)}$ computed by the procedure, we have that

$$p(t) = f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*) + \Phi(t+1) - \Phi(t) \le \frac{\gamma L^2}{2} .$$

Proof We first bound the potential change $\Phi(t+1) - \Phi(t)$. Using the definition of $\Phi(t)$ from equation (33.26), we have

$$\Phi(t+1) - \Phi(t) = \frac{1}{2\gamma}\left\|\mathbf{x}^{(t+1)} - \mathbf{x}^*\right\|^2 - \frac{1}{2\gamma}\left\|\mathbf{x}^{(t)} - \mathbf{x}^*\right\|^2 . \tag{33.27}$$

From line 4 in GRADIENT-DESCENT, we know that

$$\mathbf{x}^{(t+1)} - \mathbf{x}^{(t)} = -\gamma \cdot (\nabla f)(\mathbf{x}^{(t)}) , \tag{33.28}$$

and so we would like to rewrite equation (33.27) to have $\mathbf{x}^{(t+1)} - \mathbf{x}^{(t)}$ terms. As Exercise 33.3-3 asks you to prove, for any two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, we have

$$\|\mathbf{a} + \mathbf{b}\|^2 - \|\mathbf{a}\|^2 = 2\langle \mathbf{b}, \mathbf{a}\rangle + \|\mathbf{b}\|^2 . \tag{33.29}$$

Letting $\mathbf{a} = \mathbf{x}^{(t)} - \mathbf{x}^*$ and $\mathbf{b} = \mathbf{x}^{(t+1)} - \mathbf{x}^{(t)}$, we can write the right-hand side of equation (33.27) as $\frac{1}{2\gamma}\left(\|\mathbf{a} + \mathbf{b}\|^2 - \|\mathbf{a}\|^2\right)$. Then we can express the potential change as

$$
\begin{aligned}
&\Phi(t+1) - \Phi(t) \\
&= \frac{1}{2\gamma}\left\|\mathbf{x}^{(t+1)} - \mathbf{x}^*\right\|^2 - \frac{1}{2\gamma}\left\|\mathbf{x}^{(t)} - \mathbf{x}^*\right\|^2 && \text{(by equation (33.27))} \\
&= \frac{1}{2\gamma}\left(2\langle \mathbf{x}^{(t+1)} - \mathbf{x}^{(t)}, \mathbf{x}^{(t)} - \mathbf{x}^*\rangle + \left\|\mathbf{x}^{(t+1)} - \mathbf{x}^{(t)}\right\|^2\right) && \text{(by equation (33.29))} \\
&= \frac{1}{2\gamma}\left(2\langle -\gamma \cdot (\nabla f)(\mathbf{x}^{(t)}), \mathbf{x}^{(t)} - \mathbf{x}^*\rangle + \left\|-\gamma \cdot (\nabla f)(\mathbf{x}^{(t)})\right\|^2\right) \\
& && \text{(by equation (33.28))} \\
&= -\langle (\nabla f)(\mathbf{x}^{(t)}), \mathbf{x}^{(t)} - \mathbf{x}^*\rangle + \frac{\gamma}{2}\left\|(\nabla f)(\mathbf{x}^{(t)})\right\|^2 && (33.30) \\
& && \text{(by equation (D.3) on page 1219)} \\
&\le -(f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)) + \frac{\gamma}{2}\left\|(\nabla f)(\mathbf{x}^{(t)})\right\|^2 && \text{(by Lemma 33.6)} ,
\end{aligned}
$$

and thus we have

$$
\Phi(t+1) - \Phi(t) \le -(f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)) + \frac{\gamma}{2}\left\|(\nabla f)(\mathbf{x}^{(t)})\right\|^2 \qquad (33.31)
$$

We can now proceed to bound $p(t)$. By the bound on the potential change from inequality (33.31), and using the definition of L (inequality (33.21)), we have

$$
\begin{aligned}
p(t) &= f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*) + \Phi(t+1) - \Phi(t) && \text{(by equation (33.22))} \\
&\le f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*) - (f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)) + \frac{\gamma}{2}\left\|(\nabla f)(\mathbf{x}^{(t)})\right\|^2 \\
& && \text{(by inequality (33.31))} \\
&= \frac{\gamma}{2}\left\|(\nabla f)(\mathbf{x}^{(t)})\right\|^2 \\
&\le \frac{\gamma L^2}{2} && \text{(by inequality (33.21))} . \quad \blacksquare
\end{aligned}
$$

Having bounded the amortized progress in one step, we now analyze the entire GRADIENT-DESCENT procedure, completing the proof of Theorem 33.8.

Proof of Theorem 33.8 Inequality (33.25) tells us that if we have an upper bound of B for $p(t)$, then we also have the bound $f(\mathbf{x\text{-}avg}) - f(\mathbf{x}^*) \le B + \Phi(0)/T$. By equations (33.20) and (33.26), we have that $\Phi(0) = R^2/(2\gamma)$. Lemma 33.9 gives us the upper bound of $B = \gamma L^2/2$, and so we have

$$
\begin{aligned}
f(\mathbf{x\text{-}avg}) - f(\mathbf{x}^*) &\le B + \frac{\Phi(0)}{T} && \text{(by inequality (33.25))} \\
&= \frac{\gamma L^2}{2} + \frac{R^2}{2\gamma T} .
\end{aligned}
$$

Our choice of $\gamma = R/(L\sqrt{T})$ in the statement of Theorem 33.8 balances the two terms, and we obtain

$$\begin{aligned}\frac{\gamma L^2}{2} + \frac{R^2}{2\gamma T} &= \frac{R}{L\sqrt{T}} \cdot \frac{L^2}{2} + \frac{R^2}{2T} \cdot \frac{L\sqrt{T}}{R} \\ &= \frac{RL}{2\sqrt{T}} + \frac{RL}{2\sqrt{T}} \\ &= \frac{RL}{\sqrt{T}} .\end{aligned}$$

Since we chose $\epsilon = RL/\sqrt{T}$ in the theorem statement, the proof is complete. ■

Continuing under the assumption that we know R (from equation (33.20)) and L (from inequality (33.21)), we can think of the analysis in a slightly different way. We can presume that we have a target accuracy ϵ and then compute the number of iterations needed. That is, we can solve $\epsilon = RL/\sqrt{T}$ for T, obtaining $T = R^2L^2/\epsilon^2$. The number of iterations thus depends on the square of R and L and, most importantly, on $1/\epsilon^2$. (The definition of L from inequality (33.21) depends on T, but we may know an upper bound on L that doesn't depend on the particular value of T.) Thus, if you want to halve your error bound, you need to run four times as many iterations.

It is quite possible that we don't really know R and L, since you'd need to know $\mathbf{x}^*$ in order to know R (since $R = \|\mathbf{x}^{(0)} - \mathbf{x}^*\|$), and you might not have an explicit upper bound on the gradient, which would provide L. You can, however, interpret the analysis of gradient descent as a proof that there is some step size for which the procedure makes progress toward the minimum. You can then compute a step size γ for which $f(\mathbf{x}^{(t)}) - f(\mathbf{x}^{(t+1)})$ is large enough. In fact, not having a fixed step size multiplier can actually help in practice, as you are free to use any step size s that achieves sufficient decrease in the value of f. You can search for a step size that achieves a large decrease via a binary-search-like routine, which is often called ***line search***. For a given function f and step size s, define the function $g(\mathbf{x}^{(t)}, s) = f(\mathbf{x}^{(t)}) - s(\nabla f)(\mathbf{x}^{(t)})$. Start with a small step size s for which $g(\mathbf{x}^{(t)}, s) \le f(\mathbf{x}^{(t)})$. Then repeatedly double s until $g(\mathbf{x}^{(t)}, 2s) \ge g(\mathbf{x}^{(t)}, s)$, and then perform a binary search in the interval $[s, 2s]$. This procedure can produce a step size that achieves a significant decrease in the objective function. In other circumstances, however, you may know good upper bounds on R and L, typically from problem-specific information, which can suffice.

The dominant computational step in each iteration of the **for** loop of lines 2–4 is computing the gradient. The complexity of computing and evaluating a gradient varies widely, depending on the application at hand. We'll discuss several applications later.

Constrained gradient descent

We can adapt gradient descent for constrained minimization to minimize a closed convex function $f(\mathbf{x})$, subject to the additional requirement that $\mathbf{x} \in K$, where K is a closed convex body. A ***body*** $K \subseteq \mathbb{R}^n$ is ***convex*** if for all $\mathbf{x}, \mathbf{y} \in K$, the convex combination $\lambda\mathbf{x} + (1-\lambda)\mathbf{y} \in K$ for all $0 \le \lambda \le 1$. A ***closed*** convex body contains its limit points. Somewhat surprisingly, restricting to the constrained problem does not significantly increase the number of iterations of gradient descent. The idea is that you run the same algorithm, but in each iteration, check whether the current point $\mathbf{x}^{(t)}$ is still within the convex body K. If it is not, just move to the closest point in K. Moving to the closest point is known as ***projection***. We formally define the projection $\Pi_K(\mathbf{x})$ of a point $\mathbf{x}$ in n dimensions onto a convex body K as the point $\mathbf{y} \in K$ such that $\|\mathbf{x} - \mathbf{y}\| = \min\{\|\mathbf{x} - \mathbf{z}\| : z \in K\}$. If we have $\mathbf{x} \in K$, then $\Pi_K(\mathbf{x}) = \mathbf{x}$.

This one change yields the procedure GRADIENT-DESCENT-CONSTRAINED, in which line 4 of GRADIENT-DESCENT is replaced by two lines. It assumes that $\mathbf{x}^{(0)} \in K$. Line 4 of GRADIENT-DESCENT-CONSTRAINED moves in the direction of the negative gradient, and line 5 projects back onto K. The lemma that follows helps to show that when $\mathbf{x}^* \in K$, if the projection step in line 5 moves from a point outside of K to a point in K, it cannot be moving away from $\mathbf{x}^*$.

```
GRADIENT-DESCENT-CONSTRAINED(f, x^(0), γ, T, K)
1  sum = 0                                  // n-dimensional vector, initially all 0
2  for t = 0 to T − 1
3      sum = sum + x^(t)                    // add each of n dimensions into sum
4      x'^(t+1) = x^(t) − γ · (∇f)(x^(t))   // (∇f)(x^(t)), x'^(t+1) are n-dimensional
5      x^(t+1) = Π_K(x'^(t+1))              // project onto K
6  x-avg = sum/T                            // divide each of n dimensions by T
7  return x-avg
```

Lemma 33.10
Consider a convex body $K \subseteq \mathbb{R}^n$ and points $\mathbf{a} \in K$ and $\mathbf{b}' \in \mathbb{R}^n$. Let $\mathbf{b} = \Pi_K(\mathbf{b}')$. Then $\|\mathbf{b} - \mathbf{a}\|^2 \le \|\mathbf{b}' - \mathbf{a}\|^2$.

Proof If $\mathbf{b}' \in K$, then $\mathbf{b} = \mathbf{b}'$ and the claim is true. Otherwise, $\mathbf{b}' \ne \mathbf{b}$, and as Figure 33.6 shows, we can extend the line segment between $\mathbf{b}$ and $\mathbf{b}'$ to a line ℓ. Let $\mathbf{c}$ be the projection of $\mathbf{a}$ onto ℓ. Point $\mathbf{c}$ may or may not be in K, and if $\mathbf{a}$ is on the boundary of K, then $\mathbf{c}$ could coincide with $\mathbf{b}$. If $\mathbf{c}$ coincides with $\mathbf{b}$ (part (c) of the figure), then $\mathbf{abb}'$ is a right triangle, and so $\|\mathbf{b} - \mathbf{a}\|^2 \le \|\mathbf{b}' - \mathbf{a}\|^2$.

Figure 33.6 Projecting a point $\mathbf{b}'$ outside the convex body K to the closest point $\mathbf{b} = \Pi_K(\mathbf{b}')$ in K. Line ℓ is the line containing $\mathbf{b}$ and $\mathbf{b}'$, and point $\mathbf{c}$ is the projection of $\mathbf{a}$ onto ℓ. **(a)** When $\mathbf{c}$ is in K. **(b)** When $\mathbf{c}$ is not in K. **(c)** When $\mathbf{a}$ is on the boundary of K and $\mathbf{c}$ coincides with $\mathbf{b}$.

If $\mathbf{c}$ does not coincide with $\mathbf{b}$ (parts (a) and (b) of the figure), then because of convexity, the angle $\angle\mathbf{abb}'$ must be obtuse. Because angle $\angle\mathbf{abb}'$ is obtuse, $\mathbf{b}$ lies between $\mathbf{c}$ and $\mathbf{b}'$ on ℓ. Furthermore, because $\mathbf{c}$ is the projection of $\mathbf{a}$ onto line ℓ, $\mathbf{acb}$ and $\mathbf{acb}'$ must be right triangles. By the Pythagorean theorem, we have that $\|\mathbf{b}' - \mathbf{a}\|^2 = \|\mathbf{a} - \mathbf{c}\|^2 + \|\mathbf{c} - \mathbf{b}'\|^2$ and $\|\mathbf{b} - \mathbf{a}\|^2 = \|\mathbf{a} - \mathbf{c}\|^2 + \|\mathbf{c} - \mathbf{b}\|^2$. Subtracting these two equations gives $\|\mathbf{b}' - \mathbf{a}\|^2 - \|\mathbf{b} - \mathbf{a}\|^2 = \|\mathbf{c} - \mathbf{b}'\|^2 - \|\mathbf{c} - \mathbf{b}\|^2$. Because $\mathbf{b}$ is between $\mathbf{c}$ and $\mathbf{b}'$, we must have $\|\mathbf{c} - \mathbf{b}'\|^2 \geq \|\mathbf{c} - \mathbf{b}\|^2$, and thus $\|\mathbf{b}' - \mathbf{a}\|^2 - \|\mathbf{b} - \mathbf{a}\|^2 \geq 0$. The lemma follows. ■

We can now repeat the entire proof for the unconstrained case and obtain the same bounds. Lemma 33.10 with $\mathbf{a} = \mathbf{x}^*$, $\mathbf{b} = \mathbf{x}^{(t+1)}$, and $\mathbf{b}' = \mathbf{x}'^{(t+1)}$ tells us that $\|\mathbf{x}^{(t+1)} - \mathbf{x}^*\|^2 \leq \|\mathbf{x}'^{(t+1)} - \mathbf{x}^*\|^2$. We can therefore derive an upper bound that matches inequality (33.31). We continue to define $\Phi(t)$ as in equation (33.26), but noting that $\mathbf{x}^{(t+1)}$, computed in line 5 of GRADIENT-DESCENT-CONSTRAINED, has a different meaning here from in inequality (33.31):

$$
\begin{aligned}
&\Phi(t+1) - \Phi(t) \\
&\quad = \frac{1}{2\gamma}\left\|\mathbf{x}^{(t+1)} - \mathbf{x}^*\right\|^2 - \frac{1}{2\gamma}\left\|\mathbf{x}^{(t)} - \mathbf{x}^*\right\|^2 && \text{(by equation (33.27))} \\
&\quad \leq \frac{1}{2\gamma}\left\|\mathbf{x}'^{(t+1)} - \mathbf{x}^*\right\|^2 - \frac{1}{2\gamma}\left\|\mathbf{x}^{(t)} - \mathbf{x}^*\right\|^2 && \text{(by Lemma 33.10)} \\
&\quad = \frac{1}{2\gamma}\left(2\langle \mathbf{x}'^{(t+1)} - \mathbf{x}^{(t)}, \mathbf{x}^{(t)} - \mathbf{x}^*\rangle + \left\|\mathbf{x}'^{(t+1)} - \mathbf{x}^*\right\|^2\right) \\
&\qquad\qquad && \text{(by equation (33.29))} \\
&\quad = \frac{1}{2\gamma}\left(2\langle -\gamma\cdot(\nabla f)(\mathbf{x}^{(t)}), \mathbf{x}^{(t)} - \mathbf{x}^*\rangle + \left\|-\gamma\cdot(\nabla f)(\mathbf{x}^{(t)})\right\|^2\right) \\
&\qquad\qquad && \text{(by line 4 of GRADIENT-DESCENT-CONSTRAINED)} \\
&\quad = -\langle (\nabla f)(\mathbf{x}^{(t)}), \mathbf{x}^{(t)} - \mathbf{x}^*\rangle + \frac{\gamma}{2}\left\|(\nabla f)(\mathbf{x}^{(t)})\right\|^2 .
\end{aligned}
$$

With the same upper bound on the change in the potential function as in equation (33.30), the entire proof of Lemma 33.9 can proceed as before. We can therefore conclude that the procedure GRADIENT-DESCENT-CONSTRAINED has the same asymptotic complexity as GRADIENT-DESCENT. We summarize this result in the following theorem.

Theorem 33.11
Let $K \subseteq \mathbb{R}^n$ be a convex body, $\mathbf{x}^* \in \mathbb{R}^n$ be the minimizer of a convex function f over K, and $\gamma = R/(L\sqrt{T})$, where R and L are defined in equations (33.20) and (33.21). Suppose that the vector **x-avg** is returned by an execution of GRADIENT-DESCENT-CONSTRAINED$(f, \mathbf{x}^{(0)}, \gamma, T, K)$. Let $\epsilon = RL/\sqrt{T}$. Then we have $f(\textbf{x-avg}) - f(\mathbf{x}^*) \le \epsilon$. ■

Applications of gradient descent

Gradient descent has many applications to minimizing functions and is widely used in optimization and machine learning. Here we sketch how it can be used to solve linear systems. Then we discuss an application to machine learning: prediction using linear regression.

In Chapter 28, we saw how to use Gaussian elimination to solve a system of linear equations $A\mathbf{x} = \mathbf{b}$, thereby computing $\mathbf{x} = A^{-1}\mathbf{b}$. If A is an $n \times n$ matrix and $\mathbf{b}$ is a length-n vector, then the running time of Gaussian elimination is $\Theta(n^3)$, which for large matrices might be prohibitively expensive. If an approximate solution is acceptable, however, you can use gradient descent.

First, let's see how to use gradient descent as a roundabout—and admittedly inefficient—way to solve for x in the scalar equation $ax = b$, where $a, x, b \in \mathbb{R}$. This equation is equivalent to $ax - b = 0$. If $ax - b$ is the derivative of a convex function $f(x)$, then $ax - b = 0$ for the value of x that minimizes $f(x)$. Given $f(x)$, gradient descent can then determine this minimizer. Of course, $f(x)$ is just the integral of $ax - b$, that is, $f(x) = \frac{1}{2}ax^2 - bx$, which is convex if $a \ge 0$. Therefore, one way to solve $ax = b$ for $a \ge 0$ is to find the minimizer for $\frac{1}{2}ax^2 - bx$ via gradient descent.

We now generalize this idea to higher dimensions, where using gradient descent may actually lead to a faster algorithm. One n-dimensional analog is the function $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}}A\mathbf{x} - \mathbf{b}^{\mathrm{T}}\mathbf{x}$, where A is an $n \times n$ matrix. The gradient of f with respect to $\mathbf{x}$ is the function $A\mathbf{x} - \mathbf{b}$. To find the value of $\mathbf{x}$ that minimizes f, we set the gradient of f to 0 and solve for $\mathbf{x}$. Solving $A\mathbf{x} - \mathbf{b} = 0$ for $\mathbf{x}$, we obtain $\mathbf{x} = A^{-1}\mathbf{b}$, Thus, minimizing $f(\mathbf{x})$ is equivalent to solving $A\mathbf{x} = \mathbf{b}$. If $f(\mathbf{x})$ is convex, then gradient descent can approximately compute this minimum.

A 1-dimensional function is convex when its second derivative is positive. The equivalent definition for a multidimensional function is that it is convex when its

Hessian matrix is positive-semidefinite (see page 1222 for a definition), where the ***Hessian matrix*** $(\nabla^2 f)(\mathbf{x})$ of a function $f(\mathbf{x})$ is the matrix in which entry (i, j) is the partial derivative of f with respect to i and j:

$$(\nabla^2 f)(\mathbf{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{pmatrix} .$$

Analogous to the 1-dimensional case, the Hessian of f is just A, and so if A is a positive-semidefinite matrix, then we can use gradient descent to find a point $\mathbf{x}$ where $A\mathbf{x} \approx \mathbf{b}$. If R and L are not too large, then this method is faster than using Gaussian elimination.

Gradient descent in machine learning

As a concrete example of supervised learning for prediction, suppose that you want to predict whether a patient will develop heart disease. For each of m patients, you have n different attributes. For example, you might have $n = 4$ and the four pieces of data are age, height, blood pressure, and number of close family members with heart disease. Denote the data for patient i as a vector $\mathbf{x}^{(i)} \in \mathbb{R}^n$, with $x_j^{(i)}$ giving the jth entry in vector $\mathbf{x}^{(i)}$. The ***label*** of patient i is denoted by a scalar $y^{(i)} \in \mathbb{R}$, signifying the severity of the patient's heart disease. The hypothesis should capture a relationship between the $\mathbf{x}^{(i)}$ values and $y^{(i)}$. For this example, we make the modeling assumption that the relationship is linear, and therefore the goal is to compute the "best" linear relationship between the $\mathbf{x}^{(i)}$ values and $y^{(i)}$: a linear function $f : \mathbb{R}^n \to \mathbb{R}$ such that $f(\mathbf{x}^{(i)}) \approx y^{(i)}$ for each patient i. Of course, no such function may exist, but you would like one that comes as close as possible. A linear function f can be defined by a vector of weights $\mathbf{w} = (w_0, w_1, \ldots, w_n)$, with

$$f(\mathbf{x}) = w_0 + \sum_{j=1}^{n} w_j x_j \ . \tag{33.32}$$

When evaluating a machine-learning model, you need to measure how close each value $f(\mathbf{x}^{(i)})$ is to its corresponding label $y^{(i)}$. In this example, we define the error $e^{(i)} \in \mathbb{R}$ associated with patient i as $e^{(i)} = f(\mathbf{x}^{(i)}) - y^{(i)}$. The objective function we choose is to minimize the sum of squares of the errors, which is

$$\begin{aligned}\sum_{i=1}^{m}\left(e^{(i)}\right)^2 &= \sum_{i=1}^{m}\left(f(\mathbf{x}^{(i)}) - y^{(i)}\right)^2 \\ &= \sum_{i=1}^{m}\left(w_0 + \sum_{j=1}^{n} w_j x_j^{(i)} - y^{(i)}\right)^2 .\end{aligned} \tag{33.33}$$

The objective function is typically called the ***loss function***, and the ***least-squares error*** given by equation (33.33) is just one example of many possible loss functions. The goal is then, given the $\mathbf{x}^{(i)}$ and $y^{(i)}$ values, to compute the weights $w_0, w_1, \ldots, w_n$ so as to minimize the loss function in equation (33.33). The variables here are the weights $w_0, w_1, \ldots, w_n$ and not the $\mathbf{x}^{(i)}$ or $y^{(i)}$ values.

This particular objective is sometimes known as a ***least-squares fit***, and the problem of finding a linear function to fit data and minimize the least-squares error is called ***linear regression***. Finding a least-squares fit is also addressed in Section 28.3.

When the function f is linear, the loss function defined in equation (33.33) is convex, because it is the sum of squares of linear functions, which are themselves convex. Therefore, we can apply gradient descent to compute a set of weights to approximately minimize the least-squares error. The concrete goal of learning is to be able to make predictions on new data. Informally, if the features are all reported in the same units and are from the same range (perhaps from being normalized), then the weights tend to have a natural interpretation because the features of the data that are better predictors of the label have a larger associated weight. For example, you would expect that, after normalization, the weight associated with the number of family members with heart disease would be larger than the weight associated with height.

The computed weights form a model of the data. Once you have a model, you can make predictions, so that given new data, you can predict its label. In our example, given a new patient $\mathbf{x}'$ who is not part of the original training data set, you would still hope to predict the chance that the new patient develops heart disease. You can do so by computing the label $f(\mathbf{x}')$, incorporating the weights computed by gradient descent.

For this linear-regression problem, the objective is to minimize the expression in equation (33.33), which is a quadratic in each of the $n+1$ weights w_j. Thus, entry j in the gradient is linear in w_j. Exercise 33.3-5 asks you to explicitly compute the gradient and see that it can be computed in $O(nm)$ time, which is linear in the input size. Compared with the exact method of solving equation (33.33) in Chapter 28, which needs to invert a matrix, gradient descent is typically much faster.

Section 33.1 briefly discussed regularization—the idea that a complicated hypothesis should be penalized in order to avoid overfitting the training data. Regularization often involves adding a term to the objective function, but it can also

be achieved by adding a constraint. One way to regularize this example would be to explicitly limit the norm of the weights, adding a constraint that $\|\mathbf{w}\| \leq B$ for some bound $B > 0$. (Recall again that the components of the vector $\mathbf{w}$ are the variables in the present application.) Adding this constraint controls the complexity of the model, as the number of values w_j that can have large absolute value is now limited.

In order to run GRADIENT-DESCENT-CONSTRAINED for any problem, you need to implement the projection step, as well as to compute bounds on R and L. We conclude this section by describing these calculations for gradient descent with the constraint $\|\mathbf{w}\| \leq B$. First, consider the projection step in line 5. Suppose that the update in line 4 results in a vector $\mathbf{w}'$. The projection is implemented by computing $\Pi_k(\mathbf{w}')$ where K is defined by $\|\mathbf{w}\| \leq B$. This particular projection can be accomplished by simply scaling $\mathbf{w}'$, since we know that closest point in K to $\mathbf{w}'$ must be the point along the vector whose norm is exactly B. The amount z by which we need to scale $\mathbf{w}'$ to hit the boundary of K is the solution to the equation $z\,\|\mathbf{w}'\| = B$, which is solved by $z = B/\|\mathbf{w}'\|$. Hence line 5 is implemented by computing $\mathbf{w} = \mathbf{w}'B/\|\mathbf{w}'\|$. Because we always have $\|\mathbf{w}\| \leq B$, Exercise 33.3-6 asks you to show that the upper bound on the magnitude L of the gradient is $O(B)$. We also get a bound on R, as follows. By the constraint $\|\mathbf{w}\| \leq B$, we know that both $\|\mathbf{w}^{(0)}\| \leq B$ and $\|\mathbf{w}^*\| \leq B$, and thus $\|\mathbf{w}^{(0)} - \mathbf{w}^*\| \leq 2B$. Using the definition of R in equation (33.20), we have $R = O(B)$. The bound $RL/\sqrt{T}$ on the accuracy of the solution after T iterations in Theorem 33.11 becomes $O(B)L/\sqrt{T} = O(B^2/\sqrt{T})$.

Exercises

33.3-1

Prove Lemma 33.6. Start from the definition of a convex function given in equation (33.18). (*Hint:* You can prove the statement when $n = 1$ first. The proof for general values of n is similar.)

33.3-2

Prove Lemma 33.7.

33.3-3

Prove equation (33.29). (*Hint:* The proof for $n = 1$ dimension is straightforward. The proof for general values of n dimensions follows along similar lines.)

33.3-4

Show that the function f in equation (33.32) is a convex function of the variables $w_0, w_1, \ldots, w_n$.

33.3-5
Compute the gradient of expression (33.33) and explain how to evaluate the gradient in $O(nm)$ time.

33.3-6
Consider the function f defined in equation (33.32), and suppose that you have a bound $\|\mathbf{w}\| \le B$, as is considered in the discussion on regularization. Show that $L = O(B)$ in this case.

33.3-7
Equation (33.2) on page 1009 gives a function that, when minimized, gives an optimal solution to the k-means problem. Explain how to use gradient descent to solve the k-means problem.

Problems

33-1 Newton's method
Gradient descent iteratively moves closer to a desired value (the minimum) of a function. Another algorithm in this spirit is known as ***Newton's method***, which is an iterative algorithm that finds the root of a function. Here, we consider Newton's method which, given a function $f : \mathbb{R} \to \mathbb{R}$, finds a value x^* such that $f(x^*) = 0$. The algorithm moves through a series of points $x^{(0)}, x^{(1)}, \ldots$. If the algorithm is currently at a point $x^{(t)}$, then to find point $x^{(t+1)}$, it first takes the equation of the line tangent to the curve at $x = x^{(t)}$,

$$y = f'(x^{(t)})(x - x^{(t)}) + f(x^{(t)}) .$$

It then uses the x-intercept of this line as the next point $x^{(t+1)}$.

a. Show that the algorithm described above can be summarized by the update rule

$$x^{(t+1)} = x^{(t)} - \frac{f(x^{(t)})}{f'(x^{(t)})} .$$

We restrict our attention to some domain I and assume that $f'(x) \ne 0$ for all $x \in I$ and that $f''(x)$ is continuous. We also assume that the starting point $x^{(0)}$ is sufficiently close to x^*, where "sufficiently close" means that we can use only the first two terms of the Taylor expansion of $f(x^*)$ about $x^{(0)}$, namely

$$f(x^*) = f(x^{(0)}) + f'(x^{(0)})(x^* - x^{(0)}) + \frac{1}{2} f''(\gamma^{(0)})(x^* - x^{(0)})^2 , \qquad (33.34)$$

where $\gamma^{(0)}$ is some value between $x^{(0)}$ and x^*. If the approximation in equation (33.34) holds for $x^{(0)}$, it also holds for any point closer to x^*.

b. Assume that the function f has exactly one point x^* for which $f(x^*) = 0$. Let $\epsilon^{(t)} = \left|x^{(t)} - x^*\right|$. Using the Taylor expansion in equation (33.34), show that

$$\epsilon^{(t+1)} = \frac{\left|f''(\gamma^{(t)})\right|}{2\,|f'(\gamma^{(t)})|}\epsilon^{(t)} ,$$

where $\gamma^{(t)}$ is some value between $x^{(t)}$ and x^*.

c. If

$$\frac{\left|f''(\gamma^{(t)})\right|}{2\,|f'(\gamma^{(t)})|} \le c$$

for some constant c and $\epsilon^{(0)} < 1$, then we say that the function f has ***quadratic convergence***, since the error decreases quadratically. Assuming that f has quadratic convergence, how many iterations are needed to find a root of $f(x)$ to an accuracy of δ? Your answer should include δ.

d. Suppose you wish to find a root of the function $f(x) = (x-3)^2$, which is also the minimizer, and you start at $x^{(0)} = 3.5$. Compare the number of iterations needed by gradient descent to find the minimizer and Newton's method to find the root.

33-2 Hedge

Another variant in the multiplicative-weights framework is known as HEDGE. It differs from WEIGHTED MAJORITY in two ways. First, HEDGE makes the prediction randomly—in iteration t, it assigns a probability $p_i^{(t)} = w_i^{(t)}/Z^{(t)}$ to expert E_i, where $Z^{(t)} = \sum_{i=1}^{n} w_i^{(t)}$. It then chooses an expert $E_{i'}$ according to this probability distribution and predicts according to $E_{i'}$. Second, the update rule is different. If an expert makes a mistake, line 16 updates that expert's weight by the rule $w_i^{(t+1)} = w_i^{(t)} e^{-\epsilon}$, for some $0 < \epsilon < 1$. Show that the expected number of mistakes made by HEDGE, running for T rounds, is at most $m^* + (\ln n)/\epsilon + \epsilon T$.

33-3 Nonoptimality of Lloyd's procedure in one dimension

Give an example to show that even in one dimension, Lloyd's procedure for finding clusters does not always return an optimum result. That is, Lloyd's procedure may terminate and return as a result a set C of clusters that does not minimize $f(S, C)$, even when S is a set of points on a line.

33-4 Stochastic gradient descent

Consider the problem described in Section 33.3 of fitting a line $f(x) = ax + b$ to a given set of point/value pairs $S = \{(x_1, y_1), ..., (x_T, y_T)\}$ by optimizing the choice of the parameters a and b using gradient descent to find a best least-squares fit. Here we consider the case where x is a real-valued variable, rather than a vector.

Suppose that you are not given the point/value pairs in S all at once, but only one at a time in an online manner. Furthermore, the points are given in random order. That is, you know that there are n points, but in iteration t you are given only (x_i, y_i) where i is independently and randomly chosen from $\{1, \ldots, T\}$.

You can use gradient descent to compute an estimate to the function. As each point (x_i, y_i) is considered, you can update the current values of a and b by taking the derivative with respect to a and b of the term of the objective function depending on (x_i, y_i). Doing so gives you a stochastic estimate of the gradient, and you can then take a small step in the opposite direction.

Give pseudcode to implement this variant of gradient descent. What would the expected value of the error be as a function of T, L, and R? (*Hint:* Replicate the analysis of GRADIENT-DESCENT in Section 33.3 for this variant.)

This procedure and its variants are known as ***stochastic gradient descent***.

Chapter notes

For a general introduction to artificial intelligence, we recommend Russell and Norvig [391]. For a general introduction to machine learning, we recommend Murphy [340].

Lloyd's procedure for the k-means problem was first proposed by Lloyd [304] and also later by Forgy [151]. It is sometimes called "Lloyd's algorithm" or the "Lloyd-Forgy algorithm." Although Mahajan et al. [310] showed that finding an optimal clustering is NP-hard, even in the plane, Kanungo et al. [241] have shown that there is an approximation algorithm for the k-means problem with approximation ratio $9 + \epsilon$, for any $\epsilon > 0$.

The multiplicative-weights method is surveyed by Arora, Hazan, and Kale [25]. The main idea of updating weights based on feedback has been rediscovered many times. One early use is in game theory, where Brown defined "Fictitious Play" [74] and conjectured its convergence to the value of a zero-sum game. The convergence properties were established by Robinson [382].

In machine learning, the first use of multiplicative weights was by Littlestone in the Winnow algorithm [300], which was later extended by Littlestone and Warmuth to the weighted-majority algorithm described in Section 33.2 [301]. This work is closely connected to the boosting algorithm, originally due to Freund and Shapire

[159]. The multiplicative-weights idea is also closely related to several more general optimization algorithms, including the perceptron algorithm [328] and algorithms for optimization problems such as packing linear programs [177, 359].

The treatment of gradient descent in this chapter draws heavily on the unpublished manuscript of Bansal and Gupta [35]. They emphasize the idea of using a potential function and using ideas from amortized analysis to explain gradient descent. Other presentations and analyses of gradient descent include works by Bubeck [75], Boyd and Vanderberghe [69], and Nesterov [343].

Gradient descent is known to converge faster when functions obey stronger properties than general convexity. For example, a function f is ***α-strongly convex*** if $f(\mathbf{y}) \geq f(\mathbf{x}) + \langle (\nabla f)(\mathbf{x}), (\mathbf{y} - \mathbf{x}) \rangle + \alpha \|\mathbf{y} - \mathbf{x}\|$ for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$. In this case, GRADIENT-DESCENT can use a variable step size and return $\mathbf{x}^{(T)}$. The step size at step t becomes $\gamma_t = 1/(\alpha(t+1))$, and the procedure returns a point such that $f(\mathbf{x}\text{-avg}) - f(\mathbf{x}^*) \leq L^2/(\alpha(T+1))$. This convergence is better than that of Theorem 33.8 because the number of iterations needed is linear, rather than quadratic, in the desired error parameter ϵ, and because the performance is independent of the initial point.

Another case in which gradient descent can be shown to perform better than the analysis in Section 33.3 suggests is for smooth convex functions. We say that a function is ***β-smooth*** if $f(\mathbf{y}) \leq f(\mathbf{x}) + \langle (\nabla f)(\mathbf{x}), (\mathbf{y} - \mathbf{x}) \rangle + \frac{\beta}{2} \|\mathbf{y} - \mathbf{x}\|^2$. This inequality goes in the opposite direction from the one for α-strong convexity. Better bounds on gradient descent are possible here as well.

34 NP-Completeness

Almost all the algorithms we have studied thus far have been ***polynomial-time algorithms***: on inputs of size n, their worst-case running time is $O(n^k)$ for some constant k. You might wonder whether *all* problems can be solved in polynomial time. The answer is no. For example, there are problems, such as Turing's famous "Halting Problem," that cannot be solved by any computer, no matter how long you're willing to wait for an answer.[1] There are also problems that can be solved, but not in $O(n^k)$ time for any constant k. Generally, we think of problems that are solvable by polynomial-time algorithms as being tractable, or "easy," and problems that require superpolynomial time as being intractable, or "hard."

The subject of this chapter, however, is an interesting class of problems, called the "NP-complete" problems, whose status is unknown. No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them. This so-called P $\neq$ NP question has been one of the deepest, most perplexing open research problems in theoretical computer science since it was first posed in 1971.

Several NP-complete problems are particularly tantalizing because they seem on the surface to be similar to problems that we know how to solve in polynomial time. In each of the following pairs of problems, one is solvable in polynomial time and the other is NP-complete, but the difference between the problems appears to be slight:

Shortest versus longest simple paths: In Chapter 22, we saw that even with negative edge weights, we can find *shortest* paths from a single source in a directed

[1] For the Halting Problem and other unsolvable problems, there are proofs that no algorithm can exist that, for every input, eventually produces the correct answer. A procedure attempting to solve an unsolvable problem might always produce an answer but is sometimes incorrect, or all the answers it produces might be correct but for some inputs it never produces an answer.

graph $G = (V, E)$ in $O(VE)$ time. Finding a *longest* simple path between two vertices is difficult, however. Merely determining whether a graph contains a simple path with at least a given number of edges is NP-complete.

Euler tour versus hamiltonian cycle: An ***Euler tour*** of a strongly connected, directed graph $G = (V, E)$ is a cycle that traverses each *edge* of G exactly once, although it is allowed to visit each vertex more than once. Problem 20-3 on page 583 asks you to show how to determine whether a strongly connected, directed graph has an Euler tour and, if it does, the order of the edges in the Euler tour, all in $O(E)$ time. A ***hamiltonian cycle*** of a directed graph $G = (V, E)$ is a simple cycle that contains each *vertex* in V. Determining whether a directed graph has a hamiltonian cycle is NP-complete. (Later in this chapter, we'll prove that determining whether an *undirected* graph has a hamiltonian cycle is NP-complete.)

2-CNF satisfiability versus 3-CNF satisfiability: Boolean formulas contain binary variables whose values are 0 or 1; boolean connectives such as $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT); and parentheses. A boolean formula is ***satisfiable*** if there exists some assignment of the values 0 and 1 to its variables that causes it to evaluate to 1. We'll define terms more formally later in this chapter, but informally, a boolean formula is in ***k-conjunctive normal form***, or k-CNF if it is the AND of clauses of ORs of exactly k variables or their negations. For example, the boolean formula $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ is in 2-CNF (with satisfying assignment $x_1 = 1$, $x_2 = 0$, and $x_3 = 1$). Although there is a polynomial-time algorithm to determine whether a 2-CNF formula is satisfiable, we'll see later in this chapter that determining whether a 3-CNF formula is satisfiable is NP-complete.

NP-completeness and the classes P and NP

Throughout this chapter, we refer to three classes of problems: P, NP, and NPC, the latter class being the NP-complete problems. We describe them informally here, with formal definitions to appear later on.

The class P consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in $O(n^k)$ time for some constant k, where n is the size of the input to the problem. Most of the problems examined in previous chapters belong to P.

The class NP consists of those problems that are "verifiable" in polynomial time. What do we mean by a problem being verifiable? If you were somehow given a "certificate" of a solution, then you could verify that the certificate is correct in time polynomial in the size of the input to the problem. For example, in the hamiltonian-cycle problem, given a directed graph $G = (V, E)$, a certificate would

be a sequence $\langle v_1, v_2, v_3, \ldots, v_{|V|} \rangle$ of $|V|$ vertices. You could check in polynomial time that the sequence contains each of the $|V|$ vertices exactly once, that $(v_i, v_{i+1}) \in E$ for $i = 1, 2, 3, \ldots, |V| - 1$, and that $(v_{|V|}, v_1) \in E$. As another example, for 3-CNF satisfiability, a certificate could be an assignment of values to variables. You could check in polynomial time that this assignment satisfies the boolean formula.

Any problem in P also belongs to NP, since if a problem belongs to P then it is solvable in polynomial time without even being supplied a certificate. We'll formalize this notion later in this chapter, but for now you can believe that $\text{P} \subseteq \text{NP}$. The famous open question is whether P is a proper subset of NP.

Informally, a problem belongs to the class NPC—and we call it ***NP-complete***—if it belongs to NP and is as "hard" as any problem in NP. We'll formally define what it means to be as hard as any problem in NP later in this chapter. In the meantime, we state without proof that if *any* NP-complete problem can be solved in polynomial time, then *every* problem in NP has a polynomial-time algorithm. Most theoretical computer scientists believe that the NP-complete problems are intractable, since given the wide range of NP-complete problems that have been studied to date—without anyone having discovered a polynomial-time solution to any of them—it would be truly astounding if all of them could be solved in polynomial time. Yet, given the effort devoted thus far to proving that NP-complete problems are intractable—without a conclusive outcome—we cannot rule out the possibility that the NP-complete problems could turn out to be solvable in polynomial time.

To become a good algorithm designer, you must understand the rudiments of the theory of NP-completeness. If you can establish a problem as NP-complete, you provide good evidence for its intractability. As an engineer, you would then do better to spend your time developing an approximation algorithm (see Chapter 35) or solving a tractable special case, rather than searching for a fast algorithm that solves the problem exactly. Moreover, many natural and interesting problems that on the surface seem no harder than sorting, graph searching, or network flow are in fact NP-complete. Therefore, you should become familiar with this remarkable class of problems.

Overview of showing problems to be NP-complete

The techniques used to show that a particular problem is NP-complete differ fundamentally from the techniques used throughout most of this book to design and analyze algorithms. If you can demonstrate that a problem is NP-complete, you are making a statement about how hard it is (or at least how hard we think it is), rather than about how easy it is. If you prove a problem NP-complete, you are saying that searching for efficient algorithm is likely to be a fruitless endeavor. In this

way, NP-completeness proofs bear some similarity to the proof in Section 8.1 of an $\Omega(n \lg n)$-time lower bound for any comparison sort algorithm, although the specific techniques used for showing NP-completeness differ from the decision-tree method used in Section 8.1.

We rely on three key concepts in showing a problem to be NP-complete:

Decision problems versus optimization problems

Many problems of interest are ***optimization problems***, in which each feasible (i.e., "legal") solution has an associated value, and the goal is to find a feasible solution with the best value. For example, in a problem that we call SHORTEST-PATH, the input is an undirected graph G and vertices u and v, and the goal is to find a path from u to v that uses the fewest edges. In other words, SHORTEST-PATH is the single-pair shortest-path problem in an unweighted, undirected graph. NP-completeness applies directly not to optimization problems, however, but to ***decision problems***, in which the answer is simply "yes" or "no" (or, more formally, "1" or "0").

Although NP-complete problems are confined to the realm of decision problems, there is usually a way to cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized. For example, a decision problem related to SHORTEST-PATH is PATH: given an undirected graph G, vertices u and v, and an integer k, does a path exist from u to v consisting of at most k edges?

The relationship between an optimization problem and its related decision problem works in your favor when you try to show that the optimization problem is "hard." That is because the decision problem is in a sense "easier," or at least "no harder." As a specific example, you can solve PATH by solving SHORTEST-PATH and then comparing the number of edges in the shortest path found to the value of the decision-problem parameter k. In other words, if an optimization problem is easy, its related decision problem is easy as well. Stated in a way that has more relevance to NP-completeness, if you can provide evidence that a decision problem is hard, you also provide evidence that its related optimization problem is hard. Thus, even though it restricts attention to decision problems, the theory of NP-completeness often has implications for optimization problems as well.

Reductions

The above notion of showing that one problem is no harder or no easier than another applies even when both problems are decision problems. Almost every NP-completeness proof takes advantage of this idea, as follows. Consider a decision problem A, which you would like to solve in polynomial time. We call the input to a particular problem an ***instance*** of that problem. For example, in PATH, an

Figure 34.1 How to use a polynomial-time reduction algorithm to solve a decision problem A in polynomial time, given a polynomial-time decision algorithm for another problem B. In polynomial time, transform an instance α of A into an instance β of B, solve B in polynomial time, and use the answer for β as the answer for α.

instance is a particular graph G, particular vertices u and v of G, and a particular integer k. Now suppose that you already know how to solve a different decision problem B in polynomial time. Finally, suppose that you have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:

- The transformation takes polynomial time.
- The answers are the same. That is, the answer for α is "yes" if and only if the answer for β is also "yes."

We call such a procedure a polynomial-time ***reduction algorithm*** and, as Figure 34.1 shows, it provides us a way to solve problem A in polynomial time:

1. Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.
2. Run the polynomial-time decision algorithm for B on the instance β.
3. Use the answer for β as the answer for α.

As long as each of these steps takes polynomial time, all three together do also, and so you have a way to decide on α in polynomial time. In other words, by "reducing" solving problem A to solving problem B, you use the "easiness" of B to prove the "easiness" of A.

Recalling that NP-completeness is about showing how hard a problem is rather than how easy it is, you use polynomial-time reductions in the opposite way to show that a problem is NP-complete. Let's take the idea a step further and show how you can use polynomial-time reductions to show that no polynomial-time algorithm can exist for a particular problem B. Suppose that you have a decision problem A for which you already know that no polynomial-time algorithm can exist. (Ignore for the moment how to find such a problem A.) Suppose further that you have a polynomial-time reduction transforming instances of A to instances of B. Now you can use a simple proof by contradiction to show that no polynomial-time algorithm can exist for B. Suppose otherwise, that is, suppose that B has a

polynomial-time algorithm. Then, using the method shown in Figure 34.1, you would have a way to solve problem A in polynomial time, which contradicts the assumption that there is no polynomial-time algorithm for A.

To prove that a problem B is NP-complete, the methodology is similar. Although you cannot assume that there is absolutely no polynomial-time algorithm for problem A, you prove that problem B is NP-complete on the assumption that problem A is also NP-complete.

A first NP-complete problem

Because the technique of reduction relies on having a problem already known to be NP-complete in order to prove a different problem NP-complete, there must be some "first" NP-complete problem. We'll use the circuit-satisfiability problem, in which the input is a boolean combinational circuit composed of AND, OR, and NOT gates, and the question is whether there exists some set of boolean inputs to this circuit that causes its output to be 1. Section 34.3 will prove that this first problem is NP-complete.

Chapter outline

This chapter studies the aspects of NP-completeness that bear most directly on the analysis of algorithms. Section 34.1 formalizes the notion of "problem" and defines the complexity class P of polynomial-time solvable decision problems. We'll also see how these notions fit into the framework of formal-language theory. Section 34.2 defines the class NP of decision problems whose solutions are verifiable in polynomial time. It also formally poses the $\mathrm{P} \neq \mathrm{NP}$ question.

Section 34.3 shows how to relate problems via polynomial-time "reductions." It defines NP-completeness and sketches a proof that the circuit-satisfiability problem is NP-complete. With one problem proven NP-complete, Section 34.4 demonstrates how to prove other problems to be NP-complete much more simply by the methodology of reductions. To illustrate this methodology, the section shows that two formula-satisfiability problems are NP-complete. Section 34.5 proves a variety of other problems to be NP-complete by using reductions. You will probably find several of these reductions to be quite creative, because they convert a problem in one domain to a problem in a completely different domain.

34.1 Polynomial time

Since NP-completeness relies on notions of solving a problem and verifying a certificate in polynomial time, let's first examine what it means for a problem to be solvable in polynomial time.

Recall that we generally regard problems that have polynomial-time solutions as tractable. Here are three reasons why:

1. Although no reasonable person considers a problem that requires $\Theta(n^{100})$ time to be tractable, few practical problems require time on the order of such a high-degree polynomial. The polynomial-time computable problems encountered in practice typically require much less time. Experience has shown that once the first polynomial-time algorithm for a problem has been discovered, more efficient algorithms often follow. Even if the current best algorithm for a problem has a running time of $\Theta(n^{100})$, an algorithm with a much better running time will likely soon be discovered.
2. For many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another. For example, the class of problems solvable in polynomial time by the serial random-access machine used throughout most of this book is the same as the class of problems solvable in polynomial time on abstract Turing machines.[2] It is also the same as the class of problems solvable in polynomial time on a parallel computer when the number of processors grows polynomially with the input size.
3. The class of polynomial-time solvable problems has nice closure properties, since polynomials are closed under addition, multiplication, and composition. For example, if the output of one polynomial-time algorithm is fed into the input of another, the composite algorithm is polynomial. Exercise 34.1-5 asks you to show that if an algorithm makes a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then the running time of the composite algorithm is polynomial.

Abstract problems

To understand the class of polynomial-time solvable problems, you must first have a formal notion of what a "problem" is. We define an ***abstract problem*** Q to be a

[2] See the books by Hopcroft and Ullman [228], Lewis and Papadimitriou [299], or Sipser [413] for a thorough treatment of the Turing-machine model.

binary relation on a set I of problem ***instances*** and a set S of problem ***solutions***. For example, an instance for SHORTEST-PATH is a triple consisting of a graph and two vertices. A solution is a sequence of vertices in the graph, with perhaps the empty sequence denoting that no path exists. The problem SHORTEST-PATH itself is the relation that associates each instance of a graph and two vertices with a shortest path in the graph that connects the two vertices. Since shortest paths are not necessarily unique, a given problem instance may have more than one solution.

This formulation of an abstract problem is more general than necessary for our purposes. As we saw above, the theory of NP-completeness restricts attention to ***decision problems***: those having a yes/no solution. In this case, we can view an abstract decision problem as a function that maps the instance set I to the solution set $\{0, 1\}$. For example, a decision problem related to SHORTEST-PATH is the problem PATH that we saw earlier. If $i = \langle G, u, v, k \rangle$ is an instance of PATH, then $\text{PATH}(i) = 1$ (yes) if G contains a path from u to v with at most k edges, and $\text{PATH}(i) = 0$ (no) otherwise. Many abstract problems are not decision problems, but rather ***optimization problems***, which require some value to be minimized or maximized. As we saw above, however, you can usually recast an optimization problem as a decision problem that is no harder.

Encodings

In order for a computer program to solve an abstract problem, its problem instances must appear in a way that the program understands. An ***encoding*** of a set S of abstract objects is a mapping e from S to the set of binary strings.[3] For example, we are all familiar with encoding the natural numbers $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$ as the strings $\{0, 1, 10, 11, 100, \ldots\}$. Using this encoding, $e(17) = 10001$. If you have looked at computer representations of keyboard characters, you probably have seen the ASCII code, where, for example, the encoding of `A` is 01000001. You can encode a compound object as a binary string by combining the representations of its constituent parts. Polygons, graphs, functions, ordered pairs, programs—all can be encoded as binary strings.

Thus, a computer algorithm that "solves" some abstract decision problem actually takes an encoding of a problem instance as input. The ***size*** of an instance i is just the length of its string, which we denote by $|i|$. We call a problem whose instance set is the set of binary strings a ***concrete problem***. We say that an algorithm ***solves*** a concrete problem in $O(T(n))$ time if, when it is provided a problem instance i of length $n = |i|$, the algorithm can produce the solution in $O(T(n))$

[3] The codomain of e need not be *binary* strings: any set of strings over a finite alphabet having at least two symbols will do.

time.[4] A concrete problem is ***polynomial-time solvable***, therefore, if there exists an algorithm to solve it in $O(n^k)$ time for some constant k.

We can now formally define the ***complexity class* P** as the set of concrete decision problems that are polynomial-time solvable.

Encodings map abstract problems to concrete problems. Given an abstract decision problem Q mapping an instance set I to $\{0, 1\}$, an encoding $e : I \to \{0, 1\}^*$ can induce a related concrete decision problem, which we denote by $e(Q)$.[5] If the solution to an abstract-problem instance $i \in I$ is $Q(i) \in \{0, 1\}$, then the solution to the concrete-problem instance $e(i) \in \{0, 1\}^*$ is also $Q(i)$. As a technicality, some binary strings might represent no meaningful abstract-problem instance. For convenience, assume that any such string maps arbitrarily to 0. Thus, the concrete problem produces the same solutions as the abstract problem on binary-string instances that represent the encodings of abstract-problem instances.

We would like to extend the definition of polynomial-time solvability from concrete problems to abstract problems by using encodings as the bridge, ideally with the definition independent of any particular encoding. That is, the efficiency of solving a problem should not depend on how the problem is encoded. Unfortunately, it depends quite heavily on the encoding. For example, suppose that the sole input to an algorithm is an integer k, and suppose that the running time of the algorithm is $\Theta(k)$. If the integer k is provided in ***unary***—a string of k 1s—then the running time of the algorithm is $O(n)$ on length-n inputs, which is polynomial time. If the input k is provided using the more natural binary representation, however, then the input length is $n = \lfloor \lg k \rfloor + 1$, so the size of the unary encoding is exponential in the size of the binary encoding. With the binary representation, the running time of the algorithm is $\Theta(k) = \Theta(2^n)$, which is exponential in the size of the input. Thus, depending on the encoding, the algorithm runs in either polynomial or superpolynomial time.

The encoding of an abstract problem matters quite a bit to how we understand polynomial time. We cannot really talk about solving an abstract problem without first specifying an encoding. Nevertheless, in practice, if we rule out "expensive" encodings such as unary ones, the actual encoding of a problem makes little difference to whether the problem can be solved in polynomial time. For example, representing integers in base 3 instead of binary has no effect on whether a problem is solvable in polynomial time, since we can convert an integer represented in base 3 to an integer represented in base 2 in polynomial time.

---

[4] We assume that the algorithm's output is separate from its input. Because it takes at least one time step to produce each bit of the output and the algorithm takes $O(T(n))$ time steps, the size of the output is $O(T(n))$.

[5] The notation $\{0, 1\}^*$ denotes the set of all strings composed of symbols from the set $\{0, 1\}$.

We say that a function $f : \{0,1\}^* \to \{0,1\}^*$ is ***polynomial-time computable*** if there exists a polynomial-time algorithm A that, given any input $x \in \{0,1\}^*$, produces as output $f(x)$. For some set I of problem instances, we say that two encodings e_1 and e_2 are ***polynomially related*** if there exist two polynomial-time computable functions f_{12} and f_{21} such that for any $i \in I$, we have $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$.[6] That is, a polynomial-time algorithm can compute the encoding $e_2(i)$ from the encoding $e_1(i)$, and vice versa. If two encodings e_1 and e_2 of an abstract problem are polynomially related, whether the problem is polynomial-time solvable or not is independent of which encoding we use, as the following lemma shows.

Lemma 34.1
Let Q be an abstract decision problem on an instance set I, and let e_1 and e_2 be polynomially related encodings on I. Then, $e_1(Q) \in \mathrm{P}$ if and only if $e_2(Q) \in \mathrm{P}$.

Proof We need only prove the forward direction, since the backward direction is symmetric. Suppose, therefore, that $e_1(Q)$ can be solved in $O(n^k)$ time for some constant k. Furthermore, suppose that for any problem instance i, the encoding $e_1(i)$ can be computed from the encoding $e_2(i)$ in $O(n^c)$ time for some constant c, where $n = |e_2(i)|$. To solve problem $e_2(Q)$ on input $e_2(i)$, first compute $e_1(i)$ and then run the algorithm for $e_1(Q)$ on $e_1(i)$. How long does this procedure take? Converting encodings takes $O(n^c)$ time, and therefore $|e_1(i)| = O(n^c)$, since the output of a serial computer cannot be longer than its running time. Solving the problem on $e_1(i)$ takes $O(|e_1(i)|^k) = O(n^{ck})$ time, which is polynomial since both c and k are constants. ■

Thus, whether an abstract problem has its instances encoded in binary or base 3 does not affect its "complexity," that is, whether it is polynomial-time solvable or not. If instances are encoded in unary, however, its complexity may change. In order to be able to converse in an encoding-independent fashion, we generally assume that problem instances are encoded in any reasonable, concise fashion, unless we specifically say otherwise. To be precise, we assume that the encoding of an integer is polynomially related to its binary representation, and that the encoding of a finite set is polynomially related to its encoding as a list of its elements, enclosed in braces and separated by commas. (ASCII is one such encoding scheme.) With

---

[6] Technically, we also require the functions f_{12} and f_{21} to "map noninstances to noninstances." A ***noninstance*** of an encoding e is a string $x \in \{0,1\}^*$ such that there is no instance i for which $e(i) = x$. We require that $f_{12}(x) = y$ for every noninstance x of encoding e_1, where y is some noninstance of e_2, and that $f_{21}(x') = y'$ for every noninstance x' of e_2, where y' is some noninstance of e_1.

such a "standard" encoding in hand, we can derive reasonable encodings of other mathematical objects, such as tuples, graphs, and formulas. To denote the standard encoding of an object, we enclose the object in angle brackets. Thus, $\langle G \rangle$ denotes the standard encoding of a graph G.

As long as the encoding implicitly used is polynomially related to this standard encoding, we can talk directly about abstract problems without reference to any particular encoding, knowing that the choice of encoding has no effect on whether the abstract problem is polynomial-time solvable. From now on, we will generally assume that all problem instances are binary strings encoded using the standard encoding, unless we explicitly specify the contrary. We'll also typically neglect the distinction between abstract and concrete problems. You should watch out for problems that arise in practice, however, in which a standard encoding is not obvious and the encoding does make a difference.

A formal-language framework

By focusing on decision problems, we can take advantage of the machinery of formal-language theory. Let's review some definitions from that theory. An ***alphabet*** Σ is a finite set of symbols. A ***language*** L over Σ is any set of strings made up of symbols from Σ. For example, if $\Sigma = \{0, 1\}$, the set $L = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}$ is the language of binary representations of prime numbers. We denote the ***empty string*** by ε, the ***empty language*** by $\emptyset$, and the language of all strings over Σ by Σ^*. For example, if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ is the set of all binary strings. Every language L over Σ is a subset of Σ^*.

Languages support a variety of operations. Set-theoretic operations, such as ***union*** and ***intersection***, follow directly from the set-theoretic definitions. We define the ***complement*** of a language L by $\overline{L} = \Sigma^* - L$. The ***concatenation*** $L_1 L_2$ of two languages L_1 and L_2 is the language

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\} \ .$$

The ***closure*** or ***Kleene star*** of a language L is the language

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots \ ,$$

where L^k is the language obtained by concatenating L to itself k times.

From the point of view of language theory, the set of instances for any decision problem Q is simply the set Σ^*, where $\Sigma = \{0, 1\}$. Since Q is entirely characterized by those problem instances that produce a 1 (yes) answer, we can view Q as a language L over $\Sigma = \{0, 1\}$, where

$$L = \{x \in \Sigma^* : Q(x) = 1\} \ .$$

For example, the decision problem PATH has the corresponding language

$$\begin{aligned}\text{PATH} = \{\langle G, u, v, k\rangle : \; & G = (V, E) \text{ is an undirected graph,} \\ & u, v \in V, \\ & k \geq 0 \text{ is an integer, and} \\ & G \text{ contains a path from } u \text{ to } v \text{ with at most } k \text{ edges}\} .\end{aligned}$$

(Where convenient, we'll sometimes use the same name—PATH in this case—to refer to both a decision problem and its corresponding language.)

The formal-language framework allows us to express concisely the relation between decision problems and algorithms that solve them. We say that an algorithm A ***accepts*** a string $x \in \{0, 1\}^*$ if, given input x, the algorithm's output $A(x)$ is 1. The language ***accepted*** by an algorithm A is the set of strings $L = \{x \in \{0, 1\}^* : A(x) = 1\}$, that is, the set of strings that the algorithm accepts. An algorithm A ***rejects*** a string x if $A(x) = 0$.

Even if language L is accepted by an algorithm A, the algorithm does not necessarily reject a string $x \notin L$ provided as input to it. For example, the algorithm might loop forever. A language L is ***decided*** by an algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A. A language L is ***accepted in polynomial time*** by an algorithm A if it is accepted by A and if in addition there exists a constant k such that for any length-n string $x \in L$, algorithm A accepts x in $O(n^k)$ time. A language L is ***decided in polynomial time*** by an algorithm A if there exists a constant k such that for any length-n string $x \in \{0, 1\}^*$, the algorithm correctly decides whether $x \in L$ in $O(n^k)$ time. Thus, to accept a language, an algorithm need only produce an answer when provided a string in L, but to decide a language, it must correctly accept or reject every string in $\{0, 1\}^*$.

As an example, the language PATH can be accepted in polynomial time. One polynomial-time accepting algorithm verifies that G encodes an undirected graph, verifies that u and v are vertices in G, uses breadth-first search to compute a path from u to v in G with the fewest edges, and then compares the number of edges on the path obtained with k. If G encodes an undirected graph and the path found from u to v has at most k edges, the algorithm outputs 1 and halts. Otherwise, the algorithm runs forever. This algorithm does not decide PATH, however, since it does not explicitly output 0 for instances in which a shortest path has more than k edges. A decision algorithm for PATH must explicitly reject binary strings that do not belong to PATH. For a decision problem such as PATH, such a decision algorithm is straightforward to design: instead of running forever when there is not a path from u to v with at most k edges, it outputs 0 and halts. (It must also output 0 and halt if the input encoding is faulty.) For other problems, such as Turing's Halting Problem, there exists an accepting algorithm, but no decision algorithm exists.

We can informally define a ***complexity class*** as a set of languages, membership in which is determined by a ***complexity measure***, such as running time, of an algorithm that determines whether a given string x belongs to language L. The actual definition of a complexity class is somewhat more technical.[7]

Using this language-theoretic framework, we can provide an alternative definition of the complexity class P:

$$\mathrm{P} = \{L \subseteq \{0,1\}^* : \text{there exists an algorithm } A \text{ that decides } L \text{ in polynomial time}\} .$$

In fact, as the following theorem shows, P is also the class of languages that can be accepted in polynomial time.

Theorem 34.2
$\mathrm{P} = \{L : L \text{ is accepted by a polynomial-time algorithm}\}$.

Proof Because the class of languages decided by polynomial-time algorithms is a subset of the class of languages accepted by polynomial-time algorithms, we need only show that if L is accepted by a polynomial-time algorithm, it is decided by a polynomial-time algorithm. Let L be the language accepted by some polynomial-time algorithm A. We use a classic "simulation" argument to construct another polynomial-time algorithm A' that decides L. Because A accepts L in $O(n^k)$ time for some constant k, there also exists a constant c such that A accepts L in at most cn^k steps. For any input string x, the algorithm A' simulates cn^k steps of A. After simulating cn^k steps, algorithm A' inspects the behavior of A. If A has accepted x, then A' accepts x by outputting a 1. If A has not accepted x, then A' rejects x by outputting a 0. The overhead of A' simulating A does not increase the running time by more than a polynomial factor, and thus A' is a polynomial-time algorithm that decides L. ■

The proof of Theorem 34.2 is nonconstructive. For a given language $L \in \mathrm{P}$, we may not actually know a bound on the running time for the algorithm A that accepts L. Nevertheless, we know that such a bound exists, and therefore, that an algorithm A' exists that can check the bound, even though we may not be able to find the algorithm A' easily.

---

[7] For more on complexity classes, see the seminal paper by Hartmanis and Stearns [210].

Exercises

34.1-1

Define the optimization problem LONGEST-PATH-LENGTH as the relation that associates each instance of an undirected graph and two vertices with the number of edges in a longest simple path between the two vertices. Define the decision problem LONGEST-PATH $= \{\langle G, u, v, k\rangle : G = (V, E)$ is an undirected graph, $u, v \in V$, $k \geq 0$ is an integer, and there exists a simple path from u to v in G consisting of at least k edges$\}$. Show that the optimization problem LONGEST-PATH-LENGTH can be solved in polynomial time if and only if LONGEST-PATH $\in$ P.

34.1-2

Give a formal definition for the problem of finding the longest simple cycle in an undirected graph. Give a related decision problem. Give the language corresponding to the decision problem.

34.1-3

Give a formal encoding of directed graphs as binary strings using an adjacency-matrix representation. Do the same using an adjacency-list representation. Argue that the two representations are polynomially related.

34.1-4

Is the dynamic-programming algorithm for the 0-1 knapsack problem that is asked for in Exercise 15.2-2 a polynomial-time algorithm? Explain your answer.

34.1-5

Show that if an algorithm makes at most a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then it runs in polynomial time. Also show that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

34.1-6

Show that the class P, viewed as a set of languages, is closed under union, intersection, concatenation, complement, and Kleene star. That is, if $L_1, L_2 \in$ P, then $L_1 \cup L_2 \in$ P, $L_1 \cap L_2 \in$ P, $L_1 L_2 \in$ P, $\overline{L}_1 \in$ P, and $L_1^* \in$ P.

34.2 Polynomial-time verification

Now, let's look at algorithms that verify membership in languages. For example, suppose that for a given instance $\langle G, u, v, k \rangle$ of the decision problem PATH, you are also given a path p from u to v. You can check whether p is a path in G and whether the length of p is at most k, and if so, you can view p as a "certificate" that the instance indeed belongs to PATH. For the decision problem PATH, this certificate doesn't seem to buy much. After all, PATH belongs to P—in fact, you can solve PATH in linear time—and so verifying membership from a given certificate takes as long as solving the problem from scratch. Instead, let's examine a problem for which we know of no polynomial-time decision algorithm and yet, given a certificate, verification is easy.

Hamiltonian cycles

The problem of finding a hamiltonian cycle in an undirected graph has been studied for over a hundred years. Formally, a ***hamiltonian cycle*** of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V. A graph that contains a hamiltonian cycle is said to be ***hamiltonian***, and otherwise, it is ***nonhamiltonian***. The name honors W. R. Hamilton, who described a mathematical game on the dodecahedron (Figure 34.2(a)) in which one player sticks five pins in any five consecutive vertices and the other player must complete the path to form a cycle containing all the vertices.[8] The dodecahedron is hamiltonian, and Figure 34.2(a) shows one hamiltonian cycle. Not all graphs are hamiltonian, however. For example, Figure 34.2(b) shows a bipartite graph with an odd number of vertices. Exercise 34.2-2 asks you to show that all such graphs are nonhamiltonian.

Here is how to define the ***hamiltonian-cycle problem***, "Does a graph G have a hamiltonian cycle?" as a formal language:

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ is a hamiltonian graph}\} .$$

How might an algorithm decide the language HAM-CYCLE? Given a problem instance $\langle G \rangle$, one possible decision algorithm lists all permutations of the vertices of G and then checks each permutation to see whether it is a hamiltonian cycle.

[8] In a letter dated 17 October 1856 to his friend John T. Graves, Hamilton [206, p. 624] wrote, "I have found that some young persons have been much amused by trying a new mathematical game which the Icosion furnishes, one person sticking five pins in any five consecutive points ... and the other player then aiming to insert, which by the theory in this letter can always be done, fifteen other pins, in cyclical succession, so as to cover all the other points, and to end in immediate proximity to the pin wherewith his antagonist had begun."

Figure 34.2 **(a)** A graph representing the vertices, edges, and faces of a dodecahedron, with a hamiltonian cycle shown by edges highlighted in blue. **(b)** A bipartite graph with an odd number of vertices. Any such graph is nonhamiltonian.

What is the running time of this algorithm? It depends on the encoding of the graph G. Let's say that G is encoded as its adjacency matrix. If the adjacency matrix contains n entries, so that the length of the encoding of G equals n, then the number m of vertices in the graph is $\Omega(\sqrt{n})$. There are $m!$ possible permutations of the vertices, and therefore the running time is $\Omega(m!) = \Omega(\sqrt{n}\,!) = \Omega(2^{\sqrt{n}})$, which is not $O(n^k)$ for any constant k. Thus, this naive algorithm does not run in polynomial time. In fact, the hamiltonian-cycle problem is NP-complete, as we'll prove in Section 34.5.

Verification algorithms

Consider a slightly easier problem. Suppose that a friend tells you that a given graph G is hamiltonian, and then the friend offers to prove it by giving you the vertices in order along the hamiltonian cycle. It would certainly be easy enough to verify the proof: simply verify that the provided cycle is hamiltonian by checking whether it is a permutation of the vertices of V and whether each of the consecutive edges along the cycle actually exists in the graph. You could certainly implement this verification algorithm to run in $O(n^2)$ time, where n is the length of the encoding of G. Thus, a proof that a hamiltonian cycle exists in a graph can be verified in polynomial time.

We define a ***verification algorithm*** as being a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a ***certificate***. A two-argument algorithm A ***verifies*** an input string x if there exists a certificate y such that $A(x, y) = 1$. The ***language verified*** by a verification algorithm A is

$$L = \{x \in \{0,1\}^* : \text{ there exists } y \in \{0,1\}^* \text{ such that } A(x, y) = 1\} .$$

Think of an algorithm A as verifying a language L if, for any string $x \in L$, there exists a certificate y that A can use to prove that $x \in L$. Moreover, for any string $x \notin L$, there must be no certificate proving that $x \in L$. For example, in the hamiltonian-cycle problem, the certificate is the list of vertices in some hamiltonian cycle. If a graph is hamiltonian, the hamiltonian cycle itself offers enough information to verify that the graph is indeed hamiltonian. Conversely, if a graph is not hamiltonian, there can be no list of vertices that fools the verification algorithm into believing that the graph is hamiltonian, since the verification algorithm carefully checks the so-called cycle to be sure.

The complexity class NP

The ***complexity class* NP** is the class of languages that can be verified by a polynomial-time algorithm.[9] More precisely, a language L belongs to NP if and only if there exist a two-input polynomial-time algorithm A and a constant c such that

$$L = \{x \in \{0,1\}^* : \text{ there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1\} .$$

We say that algorithm A ***verifies*** language L ***in polynomial time***.

From our earlier discussion about the hamiltonian-cycle problem, you can see that HAM-CYCLE $\in$ NP. (It is always nice to know that an important set is nonempty.) Moreover, if $L \in$ P, then $L \in$ NP, since if there is a polynomial-time algorithm to decide L, the algorithm can be converted to a two-argument verification algorithm that simply ignores any certificate and accepts exactly those input strings it determines to belong to L. Thus, P $\subseteq$ NP.

That leaves the question of whether P $=$ NP. A definitive answer is unknown, but most researchers believe that P and NP are not the same class. Think of the class P as consisting of problems that can be solved quickly and the class NP as

[9] The name "NP" stands for "nondeterministic polynomial time." The class NP was originally studied in the context of nondeterminism, but this book uses the somewhat simpler yet equivalent notion of verification. Hopcroft and Ullman [228] give a good presentation of NP-completeness in terms of nondeterministic models of computation.

Figure 34.3 Four possibilities for relationships among complexity classes. In each diagram, one region enclosing another indicates a proper-subset relation. **(a)** P = NP = co-NP. Most researchers regard this possibility as the most unlikely. **(b)** If NP is closed under complement, then NP = co-NP, but it need not be the case that P = NP. **(c)** P = NP ∩ co-NP, but NP is not closed under complement. **(d)** NP ≠ co-NP and P ≠ NP ∩ co-NP. Most researchers regard this possibility as the most likely.

consisting of problems for which a solution can be verified quickly. You may have learned from experience that it is often more difficult to solve a problem from scratch than to verify a clearly presented solution, especially when working under time constraints. Theoretical computer scientists generally believe that this analogy extends to the classes P and NP, and thus that NP includes languages that do not belong to P.

There is more compelling, though not conclusive, evidence that P ≠ NP—the existence of languages that are "NP-complete." Section 34.3 will study this class.

Many other fundamental questions beyond the P ≠ NP question remain unresolved. Figure 34.3 shows some possible scenarios. Despite much work by many researchers, no one even knows whether the class NP is closed under complement. That is, does $L \in$ NP imply $\overline{L} \in$ NP? We define the ***complexity class* co-NP** as the set of languages L such that $\overline{L} \in$ NP, so that the question of whether NP is closed under complement is also whether NP = co-NP. Since P is closed under complement (Exercise 34.1-6), it follows from Exercise 34.2-9 (P ⊆ co-NP) that P ⊆ NP ∩ co-NP. Once again, however, no one knows whether P = NP ∩ co-NP or whether there is some language in (NP ∩ co-NP) − P.

Thus our understanding of the precise relationship between P and NP is woefully incomplete. Nevertheless, even though we might not be able to prove that a particular problem is intractable, if we can prove that it is NP-complete, then we have gained valuable information about it.

Exercises

34.2-1
Consider the language GRAPH-ISOMORPHISM $= \{\langle G_1, G_2\rangle : G_1$ and G_2 are isomorphic graphs$\}$. Prove that GRAPH-ISOMORPHISM $\in$ NP by describing a polynomial-time algorithm to verify the language.

34.2-2
Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is nonhamiltonian.

34.2-3
Show that if HAM-CYCLE $\in$ P, then the problem of listing the vertices of a hamiltonian cycle, in order, is polynomial-time solvable.

34.2-4
Prove that the class NP of languages is closed under union, intersection, concatenation, and Kleene star. Discuss the closure of NP under complement.

34.2-5
Show that any language in NP can be decided by an algorithm with a running time of $2^{O(n^k)}$ for some constant k.

34.2-6
A ***hamiltonian path*** in a graph is a simple path that visits every vertex exactly once. Show that the language HAM-PATH $= \{\langle G, u, v\rangle :$ there is a hamiltonian path from u to v in graph $G\}$ belongs to NP.

34.2-7
Show that the hamiltonian-path problem from Exercise 34.2-6 can be solved in polynomial time on directed acyclic graphs. Give an efficient algorithm for the problem.

34.2-8
Let ϕ be a boolean formula constructed from the boolean input variables $x_1, x_2, \ldots, x_k$, negations ($\neg$), ANDs ($\wedge$), ORs ($\vee$), and parentheses. The formula ϕ is a ***tautology*** if it evaluates to 1 for every assignment of 1 and 0 to the input variables. Define TAUTOLOGY as the language of boolean formulas that are tautologies. Show that TAUTOLOGY $\in$ co-NP.

34.2-9
Prove that P $\subseteq$ co-NP.

34.2-10
Prove that if NP $\neq$ co-NP, then P $\neq$ NP.

34.2-11
Let G be a connected, undirected graph with at least three vertices, and let G^3 be the graph obtained by connecting all pairs of vertices that are connected by a path in G of length at most 3. Prove that G^3 is hamiltonian. (*Hint:* Construct a spanning tree for G, and use an inductive argument.)

34.3 NP-completeness and reducibility

Perhaps the most compelling reason why theoretical computer scientists believe that P $\neq$ NP comes from the existence of the class of NP-complete problems. This class has the intriguing property that if *any* NP-complete problem can be solved in polynomial time, then *every* problem in NP has a polynomial-time solution, that is, P = NP. Despite decades of study, though, no polynomial-time algorithm has ever been discovered for any NP-complete problem.

The language HAM-CYCLE is one NP-complete problem. If there were an algorithm to decide HAM-CYCLE in polynomial time, then every problem in NP could be solved in polynomial time. The NP-complete languages are, in a sense, the "hardest" languages in NP. In fact, if NP $-$ P turns out to be nonempty, we will be able to say with certainty that HAM-CYCLE $\in$ NP $-$ P.

This section starts by showing how to compare the relative "hardness" of languages using a precise notion called "polynomial-time reducibility." It then formally defines the NP-complete languages, finishing by sketching a proof that one such language, called CIRCUIT-SAT, is NP-complete. Sections 34.4 and 34.5 will use the notion of reducibility to show that many other problems are NP-complete.

Reducibility

One way that sometimes works for solving a problem is to recast it as a different problem. We call that strategy "reducing" one problem to another. Think of a problem Q as being reducible to another problem Q' if any instance of Q can be recast as an instance of Q', and the solution to the instance of Q' provides a solution to the instance of Q. For example, the problem of solving linear equations in an indeterminate x reduces to the problem of solving quadratic equations. Given a linear-equation instance $ax + b = 0$ (with solution $x = -b/a$), you can transform it to the quadratic equation $ax^2 + bx + 0 = 0$. This quadratic equation has the solutions $x = (-b \pm \sqrt{b^2 - 4ac})/2a$, where $c = 0$, so that $\sqrt{b^2 - 4ac} = b$. The

Figure 34.4 A function f that reduces language L_1 to language L_2. For any input $x \in \{0,1\}^*$, the question of whether $x \in L_1$ has the same answer as the question of whether $f(x) \in L_2$.

solutions are then $x = (-b+b)/2a = 0$ and $x = (-b-b)/2a = -b/a$, thereby providing a solution to $ax + b = 0$. Thus, if a problem Q reduces to another problem Q', then Q is, in a sense, "no harder to solve" than Q'.

Returning to our formal-language framework for decision problems, we say that a language L_1 is ***polynomial-time reducible*** to a language L_2, written $L_1 \le_{\mathrm{P}} L_2$, if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$$x \in L_1 \text{ if and only if } f(x) \in L_2 . \tag{34.1}$$

We call the function f the ***reduction function***, and a polynomial-time algorithm F that computes f is a ***reduction algorithm***.

Figure 34.4 illustrates the idea of a reduction from a language L_1 to another language L_2. Each language is a subset of $\{0,1\}^*$. The reduction function f provides a mapping such that if $x \in L_1$, then $f(x) \in L_2$. Moreover, if $x \notin L_1$, then $f(x) \notin L_2$. Thus, the reduction function maps any instance x of the decision problem represented by the language L_1 to an instance $f(x)$ of the problem represented by L_2. Providing an answer to whether $f(x) \in L_2$ directly provides the answer to whether $x \in L_1$. If, in addition, f can be computed in polynomial time, it is a polynomial-time reduction function.

Polynomial-time reductions give us a powerful tool for proving that various languages belong to P.

Lemma 34.3

If $L_1, L_2 \subseteq \{0,1\}^*$ are languages such that $L_1 \le_{\mathrm{P}} L_2$, then $L_2 \in \mathrm{P}$ implies $L_1 \in \mathrm{P}$.

Figure 34.5 The proof of Lemma 34.3. The algorithm F is a reduction algorithm that computes the reduction function f from L_1 to L_2 in polynomial time, and A_2 is a polynomial-time algorithm that decides L_2. Algorithm A_1 decides whether $x \in L_1$ by using F to transform any input x into $f(x)$ and then using A_2 to decide whether $f(x) \in L_2$.

Proof Let A_2 be a polynomial-time algorithm that decides L_2, and let F be a polynomial-time reduction algorithm that computes the reduction function f. We show how to construct a polynomial-time algorithm A_1 that decides L_1.

Figure 34.5 illustrates how we construct A_1. For a given input $x \in \{0,1\}^*$, algorithm A_1 uses F to transform x into $f(x)$, and then it uses A_2 to test whether $f(x) \in L_2$. Algorithm A_1 takes the output from algorithm A_2 and produces that answer as its own output.

The correctness of A_1 follows from condition (34.1). The algorithm runs in polynomial time, since both F and A_2 run in polynomial time (see Exercise 34.1-5). ■

NP-completeness

Polynomial-time reductions allow us to formally show that one problem is at least as hard as another, to within a polynomial-time factor. That is, if $L_1 \le_{\mathrm{P}} L_2$, then L_1 is not more than a polynomial factor harder than L_2, which is why the "less than or equal to" notation for reduction is mnemonic. We can now define the set of NP-complete languages, which are the hardest problems in NP.

A language $L \subseteq \{0,1\}^*$ is ***NP-complete*** if

1. $L \in \mathrm{NP}$, and
2. $L' \le_{\mathrm{P}} L$ for every $L' \in \mathrm{NP}$.

If a language L satisfies property 2, but not necessarily property 1, we say that L is ***NP-hard***. We also define NPC to be the class of NP-complete languages.

As the following theorem shows, NP-completeness is at the crux of deciding whether P is in fact equal to NP.

Theorem 34.4
If any NP-complete problem is polynomial-time solvable, then $\mathrm{P} = \mathrm{NP}$. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.

Figure 34.6 How most theoretical computer scientists view the relationships among P, NP, and NPC. Both P and NPC are wholly contained within NP, and $\text{P} \cap \text{NPC} = \emptyset$.

Proof Suppose that $L \in \text{P}$ and also that $L \in \text{NPC}$. For any $L' \in \text{NP}$, we have $L' \leq_{\text{P}} L$ by property 2 of the definition of NP-completeness. Thus, by Lemma 34.3, we also have that $L' \in \text{P}$, which proves the first statement of the theorem.

To prove the second statement, consider the contrapositive of the first statement: if $\text{P} \neq \text{NP}$, then there does not exist an NP-complete problem that is polynomial-time solvable. But $\text{P} \neq \text{NP}$ means that there is some problem in NP that is not polynomial-time solvable, and hence the second statement is the contrapositive of the first statement. ■

It is for this reason that research into the $\text{P} \neq \text{NP}$ question centers around the NP-complete problems. Most theoretical computer scientists believe that $\text{P} \neq \text{NP}$, which leads to the relationships among P, NP, and NPC shown in Figure 34.6. For all we know, however, someone may yet come up with a polynomial-time algorithm for an NP-complete problem, thus proving that $\text{P} = \text{NP}$. Nevertheless, since no polynomial-time algorithm for any NP-complete problem has yet been discovered, a proof that a problem is NP-complete provides excellent evidence that it is intractable.

Circuit satisfiability

We have defined the notion of an NP-complete problem, but up to this point, we have not actually proved that any problem is NP-complete. Once we prove that at least one problem is NP-complete, polynomial-time reducibility becomes a tool to prove other problems to be NP-complete. Thus, we now focus on demonstrating the existence of an NP-complete problem: the circuit-satisfiability problem.

Unfortunately, the formal proof that the circuit-satisfiability problem is NP-complete requires technical detail beyond the scope of this text. Instead, we'll informally describe a proof that relies on a basic understanding of boolean combinational circuits.

x	$\neg x$
0	1
1	0

(a)

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

(b)

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

(c)

Figure 34.7 Three basic logic gates, with binary inputs and outputs. Under each gate is the truth table that describes the gate's operation. **(a)** The NOT gate. **(b)** The AND gate. **(c)** The OR gate.

Boolean combinational circuits are built from boolean combinational elements that are interconnected by wires. A ***boolean combinational element*** is any circuit element that has a constant number of boolean inputs and outputs and that performs a well-defined function. Boolean values are drawn from the set $\{0, 1\}$, where 0 represents FALSE and 1 represents TRUE.

The boolean combinational elements appearing in the circuit-satisfiability problem compute simple boolean functions, and they are known as ***logic gates***. Figure 34.7 shows the three basic logic gates used in the circuit-satisfiability problem: the ***NOT gate*** (or ***inverter***), the ***AND gate***, and the ***OR gate***. The NOT gate takes a single binary ***input*** x, whose value is either 0 or 1, and produces a binary ***output*** z whose value is opposite that of the input value. Each of the other two gates takes two binary inputs x and y and produces a single binary output z.

The operation of each gate, or of any boolean combinational element, is defined by a ***truth table***, shown under each gate in Figure 34.7. A truth table gives the outputs of the combinational element for each possible setting of the inputs. For example, the truth table for the OR gate says that when the inputs are $x = 0$ and $y = 1$, the output value is $z = 1$. The symbol $\neg$ denotes the NOT function, $\wedge$ denotes the AND function, and $\vee$ denotes the OR function. Thus, for example, $0 \vee 1 = 1$.

AND and OR gates are not limited to just two inputs. An AND gate's output is 1 if all of its inputs are 1, and its output is 0 otherwise. An OR gate's output is 1 if any of its inputs are 1, and its output is 0 otherwise.

A ***boolean combinational circuit*** consists of one or more boolean combinational elements interconnected by ***wires***. A wire can connect the output of one element to the input of another, so that the output value of the first element becomes an input value of the second. Figure 34.8 shows two similar boolean combinational circuits, differing in only one gate. Part (a) of the figure also shows the values on

Figure 34.8 Two instances of the circuit-satisfiability problem. **(a)** The assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. **(b)** No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

the individual wires, given the input $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$. Although a single wire may have no more than one combinational-element output connected to it, it can feed several element inputs. The number of element inputs fed by a wire is called the ***fan-out*** of the wire. If no element output is connected to a wire, the wire is a ***circuit input***, accepting input values from an external source. If no element input is connected to a wire, the wire is a ***circuit output***, providing the results of the circuit's computation to the outside world. (An internal wire can also fan out to a circuit output.) For the purpose of defining the circuit-satisfiability problem, we limit the number of circuit outputs to 1, though in actual hardware design, a boolean combinational circuit may have multiple outputs.

Boolean combinational circuits contain no cycles. In other words, for a given combinational circuit, imagine a directed graph $G = (V, E)$ with one vertex for each combinational element and with k directed edges for each wire whose fan-out is k, where the graph contains a directed edge (u, v) if a wire connects the output of element u to an input of element v. Then G must be acyclic.

A ***truth assignment*** for a boolean combinational circuit is a set of boolean input values. We say that a 1-output boolean combinational circuit is ***satisfiable*** if it has a ***satisfying assignment***: a truth assignment that causes the output of the circuit to be 1. For example, the circuit in Figure 34.8(a) has the satisfying assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$, and so it is satisfiable. As Exercise 34.3-1 asks you to show, no assignment of values to x_1, x_2, and x_3 causes the circuit in Figure 34.8(b) to produce a 1 output. Since it always produces 0, it is unsatisfiable.

The ***circuit-satisfiability problem*** is, "Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?" In order to pose this

question formally, however, we must agree on a standard encoding for circuits. The ***size*** of a boolean combinational circuit is the number of boolean combinational elements plus the number of wires in the circuit. We could devise a graph-like encoding that maps any given circuit C into a binary string $\langle C \rangle$ whose length is polynomial in the size of the circuit itself. As a formal language, we can therefore define

$$\text{CIRCUIT-SAT} = \{\langle C \rangle : C \text{ is a satisfiable boolean combinational circuit}\} .$$

The circuit-satisfiability problem arises in the area of computer-aided hardware optimization. If a subcircuit always produces 0, that subcircuit is unnecessary: the designer can replace it by a simpler subcircuit that omits all logic gates and provides the constant 0 value as its output. You can see the value in having a polynomial-time algorithm for this problem.

Given a circuit C, you can determine whether it is satisfiable by simply checking all possible assignments to the inputs. Unfortunately, if the circuit has k inputs, then you would have to check up to 2^k possible assignments. When the size of C is polynomial in k, checking all possible assignments to the inputs takes $\Omega(2^k)$ time, which is superpolynomial in the size of the circuit.[10] In fact, as we have claimed, there is strong evidence that no polynomial-time algorithm exists that solves the circuit-satisfiability problem because circuit satisfiability is NP-complete. We break the proof of this fact into two parts, based on the two parts of the definition of NP-completeness.

Lemma 34.5
The circuit-satisfiability problem belongs to the class NP.

Proof We provide a two-input, polynomial-time algorithm A that can verify CIRCUIT-SAT. One of the inputs to A is (a standard encoding of) a boolean combinational circuit C. The other input is a certificate corresponding to an assignment of a boolean value to each of the wires in C. (See Exercise 34.3-4 for a smaller certificate.)

The algorithm A works as follows. For each logic gate in the circuit, it checks that the value provided by the certificate on the output wire is correctly computed as a function of the values on the input wires. Then, if the output of the entire circuit is 1, algorithm A outputs 1, since the values assigned to the inputs of C provide a satisfying assignment. Otherwise, A outputs 0.

---

[10] On the other hand, if the size of the circuit C is $\Theta(2^k)$, then an algorithm whose running time is $O(2^k)$ has a running time that is polynomial in the circuit size. Even if P $\neq$ NP, this situation would not contradict the NP-completeness of the problem. The existence of a polynomial-time algorithm for a special case does not imply that there is a polynomial-time algorithm for all cases.

Whenever a satisfiable circuit C is input to algorithm A, there exists a certificate whose length is polynomial in the size of C and that causes A to output a 1. Whenever an unsatisfiable circuit is input, no certificate can fool A into believing that the circuit is satisfiable. Algorithm A runs in polynomial time, and with a good implementation, linear time suffices. Thus, CIRCUIT-SAT is verifiable in polynomial time, and CIRCUIT-SAT $\in$ NP. ■

The second part of proving that CIRCUIT-SAT is NP-complete is to show that the language is NP-hard: that *every* language in NP is polynomial-time reducible to CIRCUIT-SAT. The actual proof of this fact is full of technical intricacies, and so instead we'll sketch the proof based on some understanding of the workings of computer hardware.

A computer program is stored in the computer's memory as a sequence of instructions. A typical instruction encodes an operation to be performed, addresses of operands in memory, and an address where the result is to be stored. A special memory location, called the ***program counter***, keeps track of which instruction is to be executed next. The program counter automatically increments when each instruction is fetched, thereby causing the computer to execute instructions sequentially. Certain instructions can cause a value to be written to the program counter, however, which alters the normal sequential execution and allows the computer to loop and perform conditional branches.

At any point while a program executes, the computer's memory holds the entire state of the computation. (Consider the memory to include the program itself, the program counter, working storage, and any of the various bits of state that a computer maintains for bookkeeping.) We call any particular state of computer memory a ***configuration***. When an instruction executes, it transforms the configuration. Think of an instruction as mapping one configuration to another. The computer hardware that accomplishes this mapping can be implemented as a boolean combinational circuit, which we denote by M in the proof of the following lemma.

Lemma 34.6
The circuit-satisfiability problem is NP-hard.

Proof Let L be any language in NP. We'll describe a polynomial-time algorithm F computing a reduction function f that maps every binary string x to a circuit $C = f(x)$ such that $x \in L$ if and only if $C \in$ CIRCUIT-SAT.

Since $L \in$ NP, there must exist an algorithm A that verifies L in polynomial time. The algorithm F that we construct uses the two-input algorithm A to compute the reduction function f.

Let $T(n)$ denote the worst-case running time of algorithm A on length-n input strings, and let $k \geq 1$ be a constant such that $T(n) = O(n^k)$ and the length of the

Figure 34.9 The sequence of configurations produced by an algorithm A running on an input x and certificate y. Each configuration represents the state of the computer for one step of the computation and, besides A, x, and y, includes the program counter (PC), auxiliary machine state, and working storage. Except for the certificate y, the initial configuration c_0 is constant. A boolean combinational circuit M maps each configuration to the next configuration. The output is a distinguished bit in the working storage.

certificate is $O(n^k)$. (The running time of A is actually a polynomial in the total input size, which includes both an input string and a certificate, but since the length of the certificate is polynomial in the length n of the input string, the running time is polynomial in n.)

The basic idea of the proof is to represent the computation of A as a sequence of configurations. As Figure 34.9 illustrates, consider each configuration as com-

prising a few parts: the program for A, the program counter and auxiliary machine state, the input x, the certificate y, and working storage. The combinational circuit M, which implements the computer hardware, maps each configuration c_i to the next configuration c_{i+1}, starting from the initial configuration c_0. Algorithm A writes its output—0 or 1—to some designated location by the time it finishes executing. After A halts, the output value never changes. Thus, if the algorithm runs for at most $T(n)$ steps, the output appears as one of the bits in $c_{T(n)}$.

The reduction algorithm F constructs a single combinational circuit that computes all configurations produced by a given initial configuration. The idea is to paste together $T(n)$ copies of the circuit M. The output of the ith circuit, which produces configuration c_i, feeds directly into the input of the $(i+1)$st circuit. Thus, the configurations, rather than being stored in the computer's memory, simply reside as values on the wires connecting copies of M.

Recall what the polynomial-time reduction algorithm F must do. Given an input x, it must compute a circuit $C = f(x)$ that is satisfiable if and only if there exists a certificate y such that $A(x, y) = 1$. When F obtains an input x, it first computes $n = |x|$ and constructs a combinational circuit C' consisting of $T(n)$ copies of M. The input to C' is an initial configuration corresponding to a computation on $A(x, y)$, and the output is the configuration $c_{T(n)}$.

Algorithm F modifies circuit C' slightly to construct the circuit $C = f(x)$. First, it wires the inputs to C' corresponding to the program for A, the initial program counter, the input x, and the initial state of memory directly to these known values. Thus, the only remaining inputs to the circuit correspond to the certificate y. Second, it ignores all outputs from C', except for the one bit of $c_{T(n)}$ corresponding to the output of A. This circuit C, so constructed, computes $C(y) = A(x, y)$ for any input y of length $O(n^k)$. The reduction algorithm F, when provided an input string x, computes such a circuit C and outputs it.

We need to prove two properties. First, we must show that F correctly computes a reduction function f. That is, we must show that C is satisfiable if and only if there exists a certificate y such that $A(x, y) = 1$. Second, we must show that F runs in polynomial time.

To show that F correctly computes a reduction function, suppose that there exists a certificate y of length $O(n^k)$ such that $A(x, y) = 1$. Then, upon applying the bits of y to the inputs of C, the output of C is $C(y) = A(x, y) = 1$. Thus, if a certificate exists, then C is satisfiable. For the other direction, suppose that C is satisfiable. Hence, there exists an input y to C such that $C(y) = 1$, from which we conclude that $A(x, y) = 1$. Thus, F correctly computes a reduction function.

To complete the proof sketch, we need to show that F runs in time polynomial in $n = |x|$. First, the number of bits required to represent a configuration is polynomial in n. Why? The program for A itself has constant size, independent of the length of its input x. The length of the input x is n, and the length of the certifi-

cate y is $O(n^k)$. Since the algorithm runs for at most $O(n^k)$ steps, the amount of working storage required by A is polynomial in n as well. (We implicitly assume that this memory is contiguous. Exercise 34.3-5 asks you to extend the argument to the situation in which the locations accessed by A are scattered across a much larger region of memory and the particular pattern of scattering can differ for each input x.)

The combinational circuit M implementing the computer hardware has size polynomial in the length of a configuration, which is $O(n^k)$, and hence, the size of M is polynomial in n. (Most of this circuitry implements the logic of the memory system.) The circuit C consists of $O(n^k)$ copies of M, and hence it has size polynomial in n. The reduction algorithm F can construct C from x in polynomial time, since each step of the construction takes polynomial time. ■

The language CIRCUIT-SAT is therefore at least as hard as any language in NP, and since it belongs to NP, it is NP-complete.

Theorem 34.7
The circuit-satisfiability problem is NP-complete.

Proof Immediate from Lemmas 34.5 and 34.6 and from the definition of NP-completeness. ■

Exercises

34.3-1
Verify that the circuit in Figure 34.8(b) is unsatisfiable.

34.3-2
Show that the $\leq_P$ relation is a transitive relation on languages. That is, show that if $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.

34.3-3
Prove that $L \leq_P \overline{L}$ if and only if $\overline{L} \leq_P L$.

34.3-4
Show that an alternative proof of Lemma 34.5 can use a satisfying assignment as a certificate. Which certificate makes for an easier proof?

34.3-5
The proof of Lemma 34.6 assumes that the working storage for algorithm A occupies a contiguous region of polynomial size. Where does the proof exploit this assumption? Argue that this assumption does not involve any loss of generality.

34.3-6

A language L is ***complete*** for a language class C with respect to polynomial-time reductions if $L \in C$ and $L' \leq_{\mathrm{P}} L$ for all $L' \in C$. Show that $\emptyset$ and $\{0, 1\}^*$ are the only languages in P that are not complete for P with respect to polynomial-time reductions.

34.3-7

Show that, with respect to polynomial-time reductions (see Exercise 34.3-6), L is complete for NP if and only if $\overline{L}$ is complete for co-NP.

34.3-8

The reduction algorithm F in the proof of Lemma 34.6 constructs the circuit $C = f(x)$ based on knowledge of x, A, and k. Professor Sartre observes that the string x is input to F, but only the existence of A, k, and the constant factor implicit in the $O(n^k)$ running time is known to F (since the language L belongs to NP), not their actual values. Thus, the professor concludes that F cannot possibly construct the circuit C and that the language CIRCUIT-SAT is not necessarily NP-hard. Explain the flaw in the professor's reasoning.

34.4 NP-completeness proofs

The proof that the circuit-satisfiability problem is NP-complete showed directly that $L \leq_{\mathrm{P}}$ CIRCUIT-SAT for every language $L \in$ NP. This section shows how to prove that languages are NP-complete without directly reducing *every* language in NP to the given language. We'll explore examples of this methodology by proving that various formula-satisfiability problems are NP-complete. Section 34.5 provides many more examples.

The following lemma provides a foundation for showing that a given language is NP-complete.

Lemma 34.8

If L is a language such that $L' \leq_{\mathrm{P}} L$ for some $L' \in$ NPC, then L is NP-hard. If, in addition, we have $L \in$ NP, then $L \in$ NPC.

Proof Since L' is NP-complete, for all $L'' \in$ NP, we have $L'' \leq_{\mathrm{P}} L'$. By supposition, we have $L' \leq_{\mathrm{P}} L$, and thus by transitivity (Exercise 34.3-2), we have $L'' \leq_{\mathrm{P}} L$, which shows that L is NP-hard. If $L \in$ NP, we also have $L \in$ NPC. ■

In other words, by reducing a known NP-complete language L' to L, we implicitly reduce every language in NP to L. Thus, Lemma 34.8 provides a method for proving that a language L is NP-complete:

1. Prove $L \in$ NP.
2. Prove that L is NP-hard:
   a. Select a known NP-complete language L'.
   b. Describe an algorithm that computes a function f mapping every instance $x \in \{0, 1\}^*$ of L' to an instance $f(x)$ of L.
   c. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0, 1\}^*$.
   d. Prove that the algorithm computing f runs in polynomial time.

This methodology of reducing from a single known NP-complete language is far simpler than the more complicated process of showing directly how to reduce from every language in NP. Proving CIRCUIT-SAT $\in$ NPC furnishes a starting point. Knowing that the circuit-satisfiability problem is NP-complete makes it much easier to prove that other problems are NP-complete. Moreover, as the catalog of known NP-complete problems grows, so will the choices for languages from which to reduce.

Formula satisfiability

To illustrate the reduction methodology, let's see an NP-completeness proof for the problem of determining whether a boolean *formula*, not a *circuit*, is satisfiable. This problem has the historical honor of being the first problem ever shown to be NP-complete.

We formulate the ***(formula) satisfiability*** problem in terms of the language SAT as follows. An instance of SAT is a boolean formula ϕ composed of

1. n boolean variables: $x_1, x_2, \ldots, x_n$;
2. m boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (implication), $\leftrightarrow$ (if and only if); and
3. parentheses. (Without loss of generality, assume that there are no redundant parentheses, i.e., a formula contains at most one pair of parentheses per boolean connective.)

We can encode a boolean formula ϕ in a length that is polynomial in $n + m$. As in boolean combinational circuits, a ***truth assignment*** for a boolean formula ϕ

is a set of values for the variables of ϕ, and a ***satisfying assignment*** is a truth assignment that causes it to evaluate to 1. A formula with a satisfying assignment is a ***satisfiable*** formula. The satisfiability problem asks whether a given boolean formula is satisfiable, which we can express in formal-language terms as

$$\text{SAT} = \{\langle\phi\rangle : \phi \text{ is a satisfiable boolean formula}\} .$$

As an example, the formula

$$\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

has the satisfying assignment $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$, since

$$\begin{aligned} \phi &= ((0 \to 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1 , \end{aligned} \tag{34.2}$$

and thus this formula ϕ belongs to SAT.

The naive algorithm to determine whether an arbitrary boolean formula is satisfiable does not run in polynomial time. A formula with n variables has 2^n possible assignments. If the length of $\langle\phi\rangle$ is polynomial in n, then checking every assignment requires $\Omega(2^n)$ time, which is superpolynomial in the length of $\langle\phi\rangle$. As the following theorem shows, a polynomial-time algorithm is unlikely to exist.

Theorem 34.9
Satisfiability of boolean formulas is NP-complete.

Proof We start by arguing that SAT $\in$ NP. Then we prove that SAT is NP-hard by showing that CIRCUIT-SAT $\le_{\text{P}}$ SAT, which by Lemma 34.8 will prove the theorem.

To show that SAT belongs to NP, we show that a certificate consisting of a satisfying assignment for an input formula ϕ can be verified in polynomial time. The verifying algorithm simply replaces each variable in the formula with its corresponding value and then evaluates the expression, much as we did in equation (34.2) above. This task can be done in polynomial time. If the expression evaluates to 1, then the algorithm has verified that the formula is satisfiable. Thus, SAT belongs to NP.

To prove that SAT is NP-hard, we show that CIRCUIT-SAT $\le_{\text{P}}$ SAT. In other words, we need to show how to reduce any instance of circuit satisfiability to an instance of formula satisfiability in polynomial time. We can use induction to express any boolean combinational circuit as a boolean formula. We simply look at the gate that produces the circuit output and inductively express each of the

Figure 34.10 Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit and a clause for each logic gate.

gate's inputs as formulas. We then obtain the formula for the circuit by writing an expression that applies the gate's function to its inputs' formulas.

Unfortunately, this straightforward method does not amount to a polynomial-time reduction. As Exercise 34.4-1 asks you to show, shared subformulas—which arise from gates whose output wires have fan-out of 2 or more—can cause the size of the generated formula to grow exponentially. Thus, the reduction algorithm must be somewhat more clever.

Figure 34.10 illustrates how to overcome this problem, using as an example the circuit from Figure 34.8(a). For each wire x_i in the circuit C, the formula ϕ has a variable x_i. To express how each gate operates, construct a small formula involving the variables of its incident wires. The formula has the form of an "if and only if" ($\leftrightarrow$), with the variable for the gate's output on the left and on the right a logical expression encapsulating the gate's function on its inputs. For example, the operation of the output AND gate (the rightmost gate in the figure) is $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$. We call each of these small formulas a ***clause***.

The formula ϕ produced by the reduction algorithm is the AND of the circuit-output variable with the conjunction of clauses describing the operation of each gate. For the circuit in the figure, the formula is

$$\begin{aligned}
\phi \;=\; x_{10} \;&\wedge\; (x_4 \leftrightarrow \neg x_3) \\
&\wedge\; (x_5 \leftrightarrow (x_1 \vee x_2)) \\
&\wedge\; (x_6 \leftrightarrow \neg x_4) \\
&\wedge\; (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
&\wedge\; (x_8 \leftrightarrow (x_5 \vee x_6)) \\
&\wedge\; (x_9 \leftrightarrow (x_6 \vee x_7)) \\
&\wedge\; (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))\ .
\end{aligned}$$

Given a circuit C, it is straightforward to produce such a formula ϕ in polynomial time.

Why is the circuit C satisfiable exactly when the formula ϕ is satisfiable? If C has a satisfying assignment, then each wire of the circuit has a well-defined value, and the output of the circuit is 1. Therefore, when wire values are assigned to variables in ϕ, each clause of ϕ evaluates to 1, and thus the conjunction of all evaluates to 1. Conversely, if some assignment causes ϕ to evaluate to 1, the circuit C is satisfiable by an analogous argument. Thus, we have shown that CIRCUIT-SAT $\leq_P$ SAT, which completes the proof. ■

3-CNF satisfiability

Reducing from formula satisfiability gives us an avenue to prove many problems NP-complete. The reduction algorithm must handle any input formula, though, and this requirement can lead to a huge number of cases to consider. Instead, it is usually simpler to reduce from a restricted language of boolean formulas. Of course, the restricted language must not be polynomial-time solvable. One convenient language is 3-CNF satisfiability, or 3-CNF-SAT.

In order to define 3-CNF satisfiability, we first need to define a few terms. A ***literal*** in a boolean formula is an occurrence of a variable (such as x_1) or its negation ($\neg x_1$). A ***clause*** is the OR of one or more literals, such as $x_1 \vee \neg x_2 \vee \neg x_3$. A boolean formula is in ***conjunctive normal form***, or ***CNF***, if it is expressed as an AND of clauses, and it's in ***3-conjunctive normal form***, or ***3-CNF***, if each clause contains exactly three distinct literals.

For example, the boolean formula

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

is in 3-CNF. The first of its three clauses is $(x_1 \vee \neg x_1 \vee \neg x_2)$, which contains the three literals x_1, $\neg x_1$, and $\neg x_2$.

The language 3-CNF-SAT consists of encodings of boolean formulas in 3-CNF that are satisfiable. The following theorem shows that a polynomial-time algorithm that can determine the satisfiability of boolean formulas is unlikely to exist, even when they are expressed in this simple normal form.

Theorem 34.10
Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.

Proof The argument from the proof of Theorem 34.9 to show that SAT $\in$ NP applies equally well here to show that 3-CNF-SAT $\in$ NP. By Lemma 34.8, therefore, we need only show that SAT $\leq_P$ 3-CNF-SAT.

Figure 34.11 The tree corresponding to the formula $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

We break the reduction algorithm into three basic steps. Each step progressively transforms the input formula ϕ closer to the desired 3-conjunctive normal form.

The first step is similar to the one used to prove CIRCUIT-SAT $\leq_{\mathrm{P}}$ SAT in Theorem 34.9. First, construct a binary "parse" tree for the input formula ϕ, with literals as leaves and connectives as internal nodes. Figure 34.11 shows such a parse tree for the formula

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 \ . \tag{34.3}$$

If the input formula contains a clause such as the OR of several literals, use associativity to parenthesize the expression fully so that every internal node in the resulting tree has just one or two children. The binary parse tree is like a circuit for computing the function.

Mimicking the reduction in the proof of Theorem 34.9, introduce a variable y_i for the output of each internal node. Then rewrite the original formula ϕ as the AND of the variable at the root of the parse tree and a conjunction of clauses describing the operation of each node. For the formula (34.3), the resulting expression is

$$\begin{aligned}
\phi' \ = \ & y_1 \ \wedge \ (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
& \wedge \ (y_2 \leftrightarrow (y_3 \vee y_4)) \\
& \wedge \ (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
& \wedge \ (y_4 \leftrightarrow \neg y_5) \\
& \wedge \ (y_5 \leftrightarrow (y_6 \vee x_4)) \\
& \wedge \ (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) \ .
\end{aligned}$$

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Figure 34.12 The truth table for the clause $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$.

The formula ϕ' thus obtained is a conjunction of clauses ϕ'_i, each of which has at most three literals. These clauses are not yet ORs of three literals.

The second step of the reduction converts each clause ϕ'_i into conjunctive normal form. Construct a truth table for ϕ'_i by evaluating all possible assignments to its variables. Each row of the truth table consists of a possible assignment of the variables of the clause, together with the value of the clause under that assignment. Using the truth-table entries that evaluate to 0, build a formula in ***disjunctive normal form*** (or ***DNF***)—an OR of ANDs—that is equivalent to $\neg\phi'_i$. Then negate this formula and convert it into a CNF formula ϕ''_i by using ***DeMorgan's laws*** for propositional logic,

$$\neg(a \wedge b) = \neg a \vee \neg b ,$$
$$\neg(a \vee b) = \neg a \wedge \neg b ,$$

to complement all literals, change ORs into ANDs, and change ANDs into ORs.

In our example, the clause $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ converts into CNF as follows. The truth table for ϕ'_1 appears in Figure 34.12. The DNF formula equivalent to $\neg\phi'_1$ is

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2) .$$

Negating and applying DeMorgan's laws yields the CNF formula

$$\begin{aligned}\phi''_1 = {} & (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \\ & \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2) ,\end{aligned}$$

which is equivalent to the original clause ϕ'_1.

At this point, each clause ϕ'_i of the formula ϕ' has been converted into a CNF formula ϕ''_i, and thus ϕ' is equivalent to the CNF formula ϕ'' consisting of the conjunction of the ϕ''_i. Moreover, each clause of ϕ'' has at most three literals.

The third and final step of the reduction further transforms the formula so that each clause has *exactly* three distinct literals. From the clauses of the CNF formula ϕ'', construct the final 3-CNF formula ϕ'''. This formula also uses two auxiliary variables, p and q. For each clause C_i of ϕ'', include the following clauses in ϕ''':

- If C_i contains three distinct literals, then simply include C_i as a clause of ϕ'''.
- If C_i contains exactly two distinct literals, that is, if $C_i = (l_1 \vee l_2)$, where l_1 and l_2 are literals, then include $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ as clauses of ϕ'''. The literals p and $\neg p$ merely fulfill the syntactic requirement that each clause of ϕ''' contain exactly three distinct literals. Whether $p = 0$ or $p = 1$, one of the clauses is equivalent to $l_1 \vee l_2$, and the other evaluates to 1, which is the identity for AND.
- If C_i contains just one distinct literal l, then include $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ as clauses of ϕ'''. Regardless of the values of p and q, one of the four clauses is equivalent to l, and the other three evaluate to 1.

We can see that the 3-CNF formula ϕ''' is satisfiable if and only if ϕ is satisfiable by inspecting each of the three steps. Like the reduction from CIRCUIT-SAT to SAT, the construction of ϕ' from ϕ in the first step preserves satisfiability. The second step produces a CNF formula ϕ'' that is algebraically equivalent to ϕ'. Then the third step produces a 3-CNF formula ϕ''' that is effectively equivalent to ϕ'', since any assignment to the variables p and q produces a formula that is algebraically equivalent to ϕ''.

We must also show that the reduction can be computed in polynomial time. Constructing ϕ' from ϕ introduces at most one variable and one clause per connective in ϕ. Constructing ϕ'' from ϕ' can introduce at most eight clauses into ϕ'' for each clause from ϕ', since each clause of ϕ' contains at most three variables, and the truth table for each clause has at most $2^3 = 8$ rows. The construction of ϕ''' from ϕ'' introduces at most four clauses into ϕ''' for each clause of ϕ''. Thus the size of the resulting formula ϕ''' is polynomial in the length of the original formula. Each of the constructions can be accomplished in polynomial time. ■

Exercises

34.4-1

Consider the straightforward (nonpolynomial-time) reduction in the proof of Theorem 34.9. Describe a circuit of size n that, when converted to a formula by this method, yields a formula whose size is exponential in n.

34.4-2
Show the 3-CNF formula that results upon using the method of Theorem 34.10 on the formula (34.3).

34.4-3
Professor Jagger proposes to show that SAT $\leq_P$ 3-CNF-SAT by using only the truth-table technique in the proof of Theorem 34.10, and not the other steps. That is, the professor proposes to take the boolean formula ϕ, form a truth table for its variables, derive from the truth table a formula in 3-DNF that is equivalent to $\neg\phi$, and then negate and apply DeMorgan's laws to produce a 3-CNF formula equivalent to ϕ. Show that this strategy does not yield a polynomial-time reduction.

34.4-4
Show that the problem of determining whether a boolean formula is a tautology is complete for co-NP. (*Hint:* See Exercise 34.3-7.)

34.4-5
Show that the problem of determining the satisfiability of boolean formulas in disjunctive normal form is polynomial-time solvable.

34.4-6
Someone gives you a polynomial-time algorithm to decide formula satisfiability. Describe how to use this algorithm to find satisfying assignments in polynomial time.

34.4-7
Let 2-CNF-SAT be the set of satisfiable boolean formulas in CNF with exactly two literals per clause. Show that 2-CNF-SAT $\in$ P. Make your algorithm as efficient as possible. (*Hint:* Observe that $x \vee y$ is equivalent to $\neg x \rightarrow y$. Reduce 2-CNF-SAT to an efficiently solvable problem on a directed graph.)

34.5 NP-complete problems

NP-complete problems arise in diverse domains: boolean logic, graphs, arithmetic, network design, sets and partitions, storage and retrieval, sequencing and scheduling, mathematical programming, algebra and number theory, games and puzzles, automata and language theory, program optimization, biology, chemistry, physics, and more. This section uses the reduction methodology to provide NP-completeness proofs for a variety of problems drawn from graph theory and set partitioning.

Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT.

Figure 34.13 outlines the structure of the NP-completeness proofs in this section and Section 34.4. We prove each language in the figure to be NP-complete by reduction from the language that points to it. At the root is CIRCUIT-SAT, which we proved NP-complete in Theorem 34.7. This section concludes with a recap of reduction strategies.

34.5.1 The clique problem

A ***clique*** in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. In other words, a clique is a complete subgraph of G. The ***size*** of a clique is the number of vertices it contains. The ***clique problem*** is the optimization problem of finding a clique of maximum size in a graph. The corresponding decision problem asks simply whether a clique of a given size k exists in the graph. The formal definition is

$$\text{CLIQUE} = \{\langle G, k\rangle : G \text{ is a graph containing a clique of size } k\} .$$

A naive algorithm for determining whether a graph $G = (V, E)$ with $|V|$ vertices contains a clique of size k lists all k-subsets of V and checks each one to see whether it forms a clique. The running time of this algorithm is $\Omega(k^2\binom{|V|}{k})$, which is polynomial if k is a constant. In general, however, k could be near $|V|/2$, in which case the algorithm runs in superpolynomial time. Indeed, an efficient algorithm for the clique problem is unlikely to exist.

Theorem 34.11
The clique problem is NP-complete.

Proof First, we show that CLIQUE $\in$ NP. For a given graph $G = (V, E)$, use the set $V' \subseteq V$ of vertices in the clique as a certificate for G. To check whether V' is a clique in polynomial time, check whether, for each pair $u, v \in V'$, the edge (u, v) belongs to E.

We next prove that 3-CNF-SAT $\leq_{\mathrm{P}}$ CLIQUE, which shows that the clique problem is NP-hard. You might be surprised that the proof reduces an instance of 3-CNF-SAT to an instance of CLIQUE, since on the surface logical formulas seem to have little to do with graphs.

The reduction algorithm begins with an instance of 3-CNF-SAT. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be a boolean formula in 3-CNF with k clauses. For $r = 1, 2, \ldots, k$, each clause C_r contains exactly three distinct literals: l_1^r, l_2^r, and l_3^r. We will construct a graph G such that ϕ is satisfiable if and only if G contains a clique of size k.

We construct the undirected graph $G = (V, E)$ as follows. For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in ϕ, place a triple of vertices v_1^r, v_2^r, and v_3^r into V. Add edge (v_i^r, v_j^s) into E if both of the following hold:

- v_i^r and v_j^s are in different triples, that is, $r \neq s$, and
- their corresponding literals are ***consistent***, that is, l_i^r is not the negation of l_j^s.

We can build this graph from ϕ in polynomial time. As an example of this construction, if

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) ,$$

then G is the graph shown in Figure 34.14.

We must show that this transformation of ϕ into G is a reduction. First, suppose that ϕ has a satisfying assignment. Then each clause C_r contains at least one literal l_i^r that is assigned 1, and each such literal corresponds to a vertex v_i^r. Picking one such "true" literal from each clause yields a set V' of k vertices. We claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$, where $r \neq s$, both corresponding literals l_i^r and l_j^s map to 1 by the given satisfying assignment, and thus the literals cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Conversely, suppose that G contains a clique V' of size k. No edges in G connect vertices in the same triple, and so V' contains exactly one vertex per triple. If $v_i^r \in V'$, then assign 1 to the corresponding literal l_i^r. Since G contains no edges between inconsistent literals, no literal and its complement are both assigned 1. Each clause is satisfied, and so ϕ is satisfied. (Any variables that do not correspond to a vertex in the clique may be set arbitrarily.) ■

Figure 34.14 The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 set to either 0 or 1. This assignment satisfies C_1 with $\neg x_2$, and it satisfies C_2 and C_3 with x_3, corresponding to the clique with blue vertices.

In the example of Figure 34.14, a satisfying assignment of ϕ has $x_2 = 0$ and $x_3 = 1$. A corresponding clique of size $k = 3$ consists of the vertices corresponding to $\neg x_2$ from the first clause, x_3 from the second clause, and x_3 from the third clause. Because the clique contains no vertices corresponding to either x_1 or $\neg x_1$, this satisfying assignment can set x_1 to either 0 or 1.

The proof of Theorem 34.11 reduced an arbitrary instance of 3-CNF-SAT to an instance of CLIQUE with a particular structure. You might think that we have shown only that CLIQUE is NP-hard in graphs in which the vertices are restricted to occur in triples and in which there are no edges between vertices in the same triple. Indeed, we have shown that CLIQUE is NP-hard only in this restricted case, but this proof suffices to show that CLIQUE is NP-hard in general graphs. Why? If there were a polynomial-time algorithm that solves CLIQUE on general graphs, it would also solve CLIQUE on restricted graphs.

The opposite approach—reducing instances of 3-CNF-SAT with a special structure to general instances of CLIQUE—does not suffice, however. Why not? Perhaps the instances of 3-CNF-SAT that we choose to reduce from are "easy," and so we would not have reduced an NP-hard problem to CLIQUE.

Moreover, the reduction uses the instance of 3-CNF-SAT, but not the solution. We would have erred if the polynomial-time reduction had relied on knowing

Figure 34.15 Reducing CLIQUE to VERTEX-COVER. **(a)** An undirected graph $G = (V, E)$ with clique $V' = \{u, v, x, y\}$, shown in blue. **(b)** The graph $\overline{G}$ produced by the reduction algorithm that has vertex cover $V - V' = \{w, z\}$, in blue.

whether the formula ϕ is satisfiable, since we do not know how to decide whether ϕ is satisfiable in polynomial time.

34.5.2 The vertex-cover problem

A ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). That is, each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E. The ***size*** of a vertex cover is the number of vertices in it. For example, the graph in Figure 34.15(b) has a vertex cover $\{w, z\}$ of size 2.

The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given graph. For this optimization problem, the corresponding decision problem asks whether a graph has a vertex cover of a given size k. As a language, we define

$$\text{VERTEX-COVER} = \{\langle G, k\rangle : \text{graph } G \text{ has a vertex cover of size } k\} .$$

The following theorem shows that this problem is NP-complete.

Theorem 34.12
The vertex-cover problem is NP-complete.

Proof We first show that VERTEX-COVER $\in$ NP. Given a graph $G = (V, E)$ and an integer k, the certificate is the vertex cover $V' \subseteq V$ itself. The verification algorithm affirms that $|V'| = k$, and then it checks, for each edge $(u, v) \in E$, that $u \in V'$ or $v \in V'$. It is easy to verify the certificate in polynomial time.

To prove that the vertex-cover problem is NP-hard, we reduce from the clique problem, showing that CLIQUE $\leq_{\text{P}}$ VERTEX-COVER. This reduction relies

on the notion of the complement of a graph. Given an undirected graph $G = (V, E)$, we define the ***complement*** of G as a graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) : u, v \in V, u \neq v, \text{ and } (u, v) \notin E\}$. In other words, $\overline{G}$ is the graph containing exactly those edges that are not in G. Figure 34.15 shows a graph and its complement and illustrates the reduction from CLIQUE to VERTEX-COVER.

The reduction algorithm takes as input an instance $\langle G, k \rangle$ of the clique problem and computes the complement $\overline{G}$ in polynomial time. The output of the reduction algorithm is the instance $\langle \overline{G}, |V| - k \rangle$ of the vertex-cover problem. To complete the proof, we show that this transformation is indeed a reduction: the graph G contains a clique of size k if and only if the graph $\overline{G}$ has a vertex cover of size $|V| - k$.

Suppose that G contains a clique $V' \subseteq V$ with $|V'| = k$. We claim that $V - V'$ is a vertex cover in $\overline{G}$. Let (u, v) be any edge in $\overline{E}$. Then, $(u, v) \notin E$, which implies that at least one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge of E. Equivalently, at least one of u or v belongs to $V - V'$, which means that edge (u, v) is covered by $V - V'$. Since (u, v) was chosen arbitrarily from $\overline{E}$, every edge of $\overline{E}$ is covered by a vertex in $V - V'$. Hence the set $V - V'$, which has size $|V| - k$, forms a vertex cover for $\overline{G}$.

Conversely, suppose that $\overline{G}$ has a vertex cover $V' \subseteq V$, where $|V'| = |V| - k$. Then for all $u, v \in V$, if $(u, v) \in \overline{E}$, then $u \in V'$ or $v \in V'$ or both. The contrapositive of this implication is that for all $u, v \in V$, if $u \notin V'$ and $v \notin V'$, then $(u, v) \in E$. In other words, $V - V'$ is a clique, and it has size $|V| - |V'| = k$. ■

Since VERTEX-COVER is NP-complete, we don't expect to find a polynomial-time algorithm for finding a minimum-size vertex cover. Section 35.1 presents a polynomial-time "approximation algorithm," however, which produces "approximate" solutions for the vertex-cover problem. The size of a vertex cover produced by the algorithm is at most twice the minimum size of a vertex cover.

Thus, you shouldn't give up hope just because a problem is NP-complete. You might be able to design a polynomial-time approximation algorithm that obtains near-optimal solutions, even though finding an optimal solution is NP-complete. Chapter 35 gives several approximation algorithms for NP-complete problems.

34.5.3 The hamiltonian-cycle problem

We now return to the hamiltonian-cycle problem defined in Section 34.2.

Theorem 34.13
The hamiltonian cycle problem is NP-complete.

Figure 34.16 The gadget used in reducing the vertex-cover problem to the hamiltonian-cycle problem. An edge (u, v) of graph G corresponds to gadget Γ_{uv} in the graph G' created in the reduction. **(a)** The gadget, with individual vertices labeled. **(b)–(d)** The paths highlighted in blue are the only possible ones through the gadget that include all vertices, assuming that the only connections from the gadget to the remainder of G' are through vertices $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$, and $[v, u, 6]$.

Proof We first show that HAM-CYCLE $\in$ NP. Given an undirected graph $G = (V, E)$, the certificate is the sequence of $|V|$ vertices that makes up the hamiltonian cycle. The verification algorithm checks that this sequence contains each vertex in V exactly once and that with the first vertex repeated at the end, it forms a cycle in G. That is, it checks that there is an edge between each pair of consecutive vertices and between the first and last vertices. This certificate can be verified in polynomial time.

We now prove that VERTEX-COVER $\leq_{\mathrm{P}}$ HAM-CYCLE, which shows that HAM-CYCLE is NP-complete. Given an undirected graph $G = (V, E)$ and an integer k, we construct an undirected graph $G' = (V', E')$ that has a hamiltonian cycle if and only if G has a vertex cover of size k. We assume without loss of generality that G contains no isolated vertices (that is, every vertex in V has at least one incident edge) and that $k \leq |V|$. (If an isolated vertex belongs to a vertex cover of size k, then there also exists a vertex cover of size $k-1$, and for any graph, the entire set V is always a vertex cover.)

Our construction uses a ***gadget***, which is a piece of a graph that enforces certain properties. Figure 34.16(a) shows the gadget we use. For each edge $(u, v) \in E$, the constructed graph G' contains one copy of this gadget, which we denote by Γ_{uv}. We denote each vertex in Γ_{uv} by $[u, v, i]$ or $[v, u, i]$, where $1 \leq i \leq 6$, so that each gadget Γ_{uv} contains 12 vertices. Gadget Γ_{uv} also contains the 14 edges shown in Figure 34.16(a).

Along with the internal structure of the gadget, we enforce the properties we want by limiting the connections between the gadget and the remainder of the graph G' that we construct. In particular, only vertices $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$, and $[v, u, 6]$ will have edges incident from outside Γ_{uv}. Any hamiltonian cycle

of G' must traverse the edges of Γ_{uv} in one of the three ways shown in Figures 34.16(b)–(d). If the cycle enters through vertex $[u, v, 1]$, it must exit through vertex $[u, v, 6]$, and it either visits all 12 of the gadget's vertices (Figure 34.16(b)) or the six vertices $[u, v, 1]$ through $[u, v, 6]$ (Figure 34.16(c)). In the latter case, the cycle will have to reenter the gadget to visit vertices $[v, u, 1]$ through $[v, u, 6]$. Similarly, if the cycle enters through vertex $[v, u, 1]$, it must exit through vertex $[v, u, 6]$, and either it visits all 12 of the gadget's vertices (Figure 34.16(d)) or it visits the six vertices $[v, u, 1]$ through $[v, u, 6]$ and reenters to visit $[u, v, 1]$ through $[u, v, 6]$ (Figure 34.16(c)). No other paths through the gadget that visit all 12 vertices are possible. In particular, it is impossible to construct two vertex-disjoint paths, one of which connects $[u, v, 1]$ to $[v, u, 6]$ and the other of which connects $[v, u, 1]$ to $[u, v, 6]$, such that the union of the two paths contains all of the gadget's vertices.

The only other vertices in V' other than those of gadgets are ***selector vertices*** $s_1, s_2, \ldots, s_k$. We'll use edges incident on selector vertices in G' to select the k vertices of the cover in G.

In addition to the edges in gadgets, E' contains two other types of edges, which Figure 34.17 shows. First, for each vertex $u \in V$, edges join pairs of gadgets in order to form a path containing all gadgets corresponding to edges incident on u in G. We arbitrarily order the vertices adjacent to each vertex $u \in V$ as $u^{(1)}, u^{(2)}, \ldots, u^{(\text{degree}(u))}$, where $\text{degree}(u)$ is the number of vertices adjacent to u. To create a path in G' through all the gadgets corresponding to edges incident on u, E' contains the edges $\{([u, u^{(i)}, 6], [u, u^{(i+1)}, 1]) : 1 \le i \le \text{degree}(u) - 1\}$. In Figure 34.17, for example, we order the vertices adjacent to w as $\langle x, y, z\rangle$, and so graph G' in part (b) of the figure includes the edges $([w, x, 6], [w, y, 1])$ and $([w, y, 6], [w, z, 1])$. The vertices adjacent to x are ordered as $\langle w, y\rangle$, so that G' includes the edge $([x, w, 6], [x, y, 1])$. For each vertex $u \in V$, these edges in G' fill in a path containing all gadgets corresponding to edges incident on u in G.

The intuition behind these edges is that if vertex $u \in V$ belongs to the vertex cover of G, then G' contains a path from $[u, u^{(1)}, 1]$ to $[u, u^{(\text{degree}(u))}, 6]$ that "covers" all gadgets corresponding to edges incident on u. That is, for each of these gadgets, say $\Gamma_{u,u^{(i)}}$, the path either includes all 12 vertices (if u belongs to the vertex cover but $u^{(i)}$ does not) or just the six vertices $[u, u^{(i)}, 1]$ through $[u, u^{(i)}, 6]$ (if both u and $u^{(i)}$ belong to the vertex cover).

The final type of edge in E' joins the first vertex $[u, u^{(1)}, 1]$ and the last vertex $[u, u^{(\text{degree}(u))}, 6]$ of each of these paths to each of the selector vertices. That is, E' includes the edges

$$\begin{aligned}&\{(s_j, [u, u^{(1)}, 1]) : u \in V \text{ and } 1 \le j \le k\}\\ &\qquad \cup \{(s_j, [u, u^{(\text{degree}(u))}, 6]) : u \in V \text{ and } 1 \le j \le k\}\ .\end{aligned}$$

Figure 34.17 Reducing an instance of the vertex-cover problem to an instance of the hamiltonian-cycle problem. **(a)** An undirected graph G with a vertex cover of size 2, consisting of the blue vertices w and y. **(b)** The undirected graph G' produced by the reduction, with the hamiltonian cycle corresponding to the vertex cover highlighted in blue. The vertex cover $\{w, y\}$ corresponds to edges $(s_1, [w, x, 1])$ and $(s_2, [y, x, 1])$ appearing in the hamiltonian cycle.

Next we show that the size of G' is polynomial in the size of G, and hence it takes time polynomial in the size of G to construct G'. The vertices of G' are those in the gadgets, plus the selector vertices. With 12 vertices per gadget, plus $k \le |V|$ selector vertices, G' contains a total of

$$\begin{aligned} |V'| &= 12\,|E| + k \\ &\le 12\,|E| + |V| \end{aligned}$$

vertices. The edges of G' are those in the gadgets, those that go between gadgets, and those connecting selector vertices to gadgets. Each gadget contains 14 edges, totaling $14\,|E|$ in all gadgets. For each vertex $u \in V$, graph G' has degree$(u) - 1$ edges going between gadgets, so that summed over all vertices in V,

$$\sum_{u \in V} (\text{degree}(u) - 1) = 2\,|E| - |V|$$

edges go between gadgets. Finally, G' has two edges for each pair consisting of a selector vertex and a vertex of V, totaling $2k\,|V|$ such edges. The total number of edges of G' is therefore

$$\begin{aligned} |E'| &= (14\,|E|) + (2\,|E| - |V|) + (2k\,|V|) \\ &= 16\,|E| + (2k-1)\,|V| \\ &\le 16\,|E| + (2\,|V| - 1)\,|V| \ . \end{aligned}$$

Now we show that the transformation from graph G to G' is a reduction. That is, we must show that G has a vertex cover of size k if and only if G' has a hamiltonian cycle.

Suppose that $G = (V, E)$ has a vertex cover $V^* \subseteq V$, where $|V^*| = k$. Let $V^* = \{u_1, u_2, \ldots, u_k\}$. As Figure 34.17 shows, we can construct a hamiltonian cycle in G' by including the following edges[11] for each vertex $u_j \in V^*$. Start by including edges $\left\{([u_j, u_j^{(i)}, 6], [u_j, u_j^{(i+1)}, 1]) : 1 \le i \le \text{degree}(u_j) - 1\right\}$, which connect all gadgets corresponding to edges incident on u_j. Also include the edges within these gadgets as Figures 34.16(b)–(d) show, depending on whether the edge is covered by one or two vertices in V^*. The hamiltonian cycle also includes the edges

$$\begin{aligned} &\{(s_j, [u_j, u_j^{(1)}, 1]) : 1 \le j \le k\} \\ &\qquad \cup \{(s_{j+1}, [u_j, u_j^{(\text{degree}(u_j))}, 6]) : 1 \le j \le k-1\} \\ &\qquad \cup \{(s_1, [u_k, u_k^{(\text{degree}(u_k))}, 6])\} \ . \end{aligned}$$

By inspecting Figure 34.17, you can verify that these edges form a cycle, where $u_1 = w$ and $u_2 = y$. The cycle starts at s_1, visits all gadgets corresponding to edges incident on u_1, then visits s_2, visits all gadgets corresponding to edges incident on u_2, and so on, until it returns to s_1. The cycle visits each gadget either once or twice, depending on whether one or two vertices of V^* cover its corresponding edge. Because V^* is a vertex cover for G, each edge in E is incident on some vertex in V^*, and so the cycle visits each vertex in each gadget of G'. Because the cycle also visits every selector vertex, it is hamiltonian.

Conversely, suppose that $G' = (V', E')$ contains a hamiltonian cycle $C \subseteq E'$. We claim that the set

$$V^* = \{u \in V : (s_j, [u, u^{(1)}, 1]) \in C \text{ for some } 1 \le j \le k\} \tag{34.4}$$

[11] Technically, a cycle is defined as a sequence of vertices rather than edges (see Section B.4). In the interest of clarity, we abuse notation here and define the hamiltonian cycle by its edges.

is a vertex cover for G.

We first argue that the set V^* is well defined, that is, for each selector vertex s_j, exactly one of the incident edges in the hamiltonian cycle C is of the form $(s_j, [u, u^{(1)}, 1])$ for some vertex $u \in V$. To see why, partition the hamiltonian cycle C into maximal paths that start at some selector vertex s_i, visit one or more gadgets, and end at some selector vertex s_j without passing through any other selector vertex. Let's call each of these maximal paths a "cover path." Let P be one such cover path, and orient it going from s_i to s_j. If P contains the edge $(s_i, [u, u^{(1)}, 1])$ for some vertex $u \in V$, then we have shown that one edge incident on s_i has the required form. Assume, then, that P contains the edge $(s_i, [v, v^{(\text{degree}(v))}, 6])$ for some vertex $v \in V$. This path enters a gadget from the bottom, as drawn in Figures 34.16 and 34.17, and it leaves from the top. It might go through several gadgets, but it always enters from the bottom of a gadget and leaves from the top. The only edges incident on vertices at the top of a gadget either go to the bottoms of other gadgets or to selector vertices. Therefore, after the last gadget in the series of gadgets visited by P, the edge taken must go to a selector vertex s_j, so that P contains an edge of the form $(s_j, [u, u^{(1)}, 1])$, where $[u, u^{(1)}, 1]$ is a vertex at the top of some gadget. To see that not both edges incident on s_j have this form, simply reverse the direction of traversing P in the above argument.

Having established that the set V^* is well defined, let's see why it is a vertex cover for G. We have already established that each cover path starts at some s_i, takes the edge $(s_i, [u, u^{(1)}, 1])$ for some vertex $u \in V$, passes through all the gadgets corresponding to edges in E incident on u, and then ends at some selector vertex s_j. (This orientation is the reverse of the orientation in the paragraph above.) Let's call this cover path P_u, and by equation (34.4), the vertex cover V^* includes u. Each gadget visited by P_u must be Γ_{uv} or Γ_{vu} for some $v \in V$. For each gadget visited by P_u, its vertices are visited by either one or two cover paths. If they are visited by one cover path, then edge $(u, v) \in E$ is covered in G by vertex u. If two cover paths visit the gadget, then the other cover path must be P_v, which implies that $v \in V^*$, and edge $(u, v) \in E$ is covered by both u and v. Because each vertex in each gadget is visited by some cover path, we see that each edge in E is covered by some vertex in V^*. ■

34.5.4 The traveling-salesperson problem

In the ***traveling-salesperson problem***, which is closely related to the hamiltonian-cycle problem, a salesperson must visit n cities. Let's model the problem as a complete graph with n vertices, so that the salesperson wishes to make a ***tour***, or hamiltonian cycle, visiting each city exactly once and finishing at the starting city. The salesperson incurs a nonnegative integer cost $c(i, j)$ to travel from city i

Figure 34.18 An instance of the traveling-salesperson problem. Edges highlighted in blue represent a minimum-cost tour, with cost 7.

to city j. In the optimization version of the problem, the salesperson wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour. For example, in Figure 34.18, a minimum-cost tour is $\langle u, w, v, x, u \rangle$, with cost 7. The formal language for the corresponding decision problem is

$$\begin{aligned} \text{TSP} = \{\langle G, c, k \rangle : \; & G = (V, E) \text{ is a complete graph,} \\ & c \text{ is a function from } V \times V \to \mathbb{N}, \\ & k \in \mathbb{N}, \text{ and} \\ & G \text{ has a traveling-salesperson tour with cost at most } k\} \; . \end{aligned}$$

The following theorem shows that a fast algorithm for the traveling-salesperson problem is unlikely to exist.

Theorem 34.14
The traveling-salesperson problem is NP-complete.

Proof We first show that TSP $\in$ NP. Given an instance of the problem, the certificate is the sequence of n vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs, and checks that the sum is at most k. This process can certainly be done in polynomial time.

To prove that TSP is NP-hard, we show that HAM-CYCLE $\leq_{\text{P}}$ TSP. Given an instance $G = (V, E)$ of HAM-CYCLE, construct an instance of TSP by forming the complete graph $G' = (V, E')$, where $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$, with the cost function c defined as

$$c(i, j) = \begin{cases} 0 & \text{if } (i, j) \in E \; , \\ 1 & \text{if } (i, j) \notin E \; . \end{cases}$$

(Because G is undirected, it contains no self-loops, and so $c(v, v) = 1$ for all vertices $v \in V$.) The instance of TSP is then $\langle G', c, 0 \rangle$, which can be created in polynomial time.

We now show that graph G has a hamiltonian cycle if and only if graph G' has a tour of cost at most 0. Suppose that graph G has a hamiltonian cycle H. Each edge in H belongs to E and thus has cost 0 in G'. Thus, H is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour H' of cost at most 0. Since the costs of the edges in E' are 0 and 1, the cost of tour H' is exactly 0 and each edge on the tour must have cost 0. Therefore, H' contains only edges in E. We conclude that H' is a hamiltonian cycle in graph G. ■

34.5.5 The subset-sum problem

We next consider an arithmetic NP-complete problem. The ***subset-sum problem*** takes as inputs a finite set S of positive integers and an integer ***target*** $t > 0$. It asks whether there exists a subset $S' \subseteq S$ whose elements sum to exactly t. For example, if $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$ and $t = 138457$, then the subset $S' = \{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$ is a solution.

As usual, we express the problem as a language:

$$\text{SUBSET-SUM} = \left\{\langle S, t\rangle : \text{there exists a subset } S' \subseteq S \text{ such that } t = \textstyle\sum_{s \in S'} s\right\} .$$

As with any arithmetic problem, it is important to recall that our standard encoding assumes that the input integers are coded in binary. With this assumption in mind, we can show that the subset-sum problem is unlikely to have a fast algorithm.

Theorem 34.15
The subset-sum problem is NP-complete.

Proof To show that SUBSET-SUM $\in$ NP, for an instance $\langle S, t\rangle$ of the problem, let the subset S' be the certificate. A verification algorithm can check whether $t = \sum_{s \in S'} s$ in polynomial time.

We now show that 3-CNF-SAT $\leq_{\mathrm{P}}$ SUBSET-SUM. Given a 3-CNF formula ϕ over variables $x_1, x_2, \ldots, x_n$ with clauses $C_1, C_2, \ldots, C_k$, each containing exactly three distinct literals, the reduction algorithm constructs an instance $\langle S, t\rangle$ of the subset-sum problem such that ϕ is satisfiable if and only if there exists a subset of S whose sum is exactly t. Without loss of generality, we make two simplifying assumptions about the formula ϕ. First, no clause contains both a variable and its negation, for such a clause is automatically satisfied by any assignment of values to the variables. Second, each variable appears in at least one clause, because it does not matter what value is assigned to a variable that appears in no clauses.

The reduction creates two numbers in set S for each variable x_i and two numbers in S for each clause C_j. The numbers will be represented in base 10, with each number containing $n + k$ digits and each digit corresponding to either one variable

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v'_3	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s'_1	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s'_2	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s'_3	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s'_4	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

Figure 34.19 The reduction of 3-CNF-SAT to SUBSET-SUM. The formula in 3-CNF is $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$, $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$, and $C_4 = (x_1 \vee x_2 \vee x_3)$. A satisfying assignment of ϕ is $\langle x_1 = 0, x_2 = 0, x_3 = 1 \rangle$. The set S produced by the reduction consists of the base-10 numbers shown: reading from top to bottom, $S = \{1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$. The target t is 1114444. The subset $S' \subseteq S$ is shaded blue, and it contains v'_1, v'_2, and v_3, corresponding to the satisfying assignment. Subset S' also contains slack variables $s_1, s'_1, s'_2, s_3, s_4$, and s'_4 to achieve the target value of 4 in the digits labeled by C_1 through C_4.

or one clause. Base 10 (and other bases, as we shall see) has the property we need of preventing carries from lower digits to higher digits.

As Figure 34.19 shows, we construct set S and target t as follows. Label each digit position by either a variable or a clause. The least significant k digits are labeled by the clauses, and the most significant n digits are labeled by variables.

- The target t has a 1 in each digit labeled by a variable and a 4 in each digit labeled by a clause.
- For each variable x_i, set S contains two integers v_i and v'_i. Each of v_i and v'_i has a 1 in the digit labeled by x_i and 0s in the other variable digits. If literal x_i appears in clause C_j, then the digit labeled by C_j in v_i contains a 1. If literal $\neg x_i$ appears in clause C_j, then the digit labeled by C_j in v'_i contains a 1. All other digits labeled by clauses in v_i and v'_i are 0.

All v_i and v'_i values in set S are unique. Why? For $\ell \neq i$, no v_ℓ or v'_ℓ values can equal v_i and v'_i in the most significant n digits. Furthermore, by our simplifying assumptions above, no v_i and v'_i can be equal in all k least significant digits. If v_i and v'_i were equal, then x_i and $\neg x_i$ would have to appear in exactly the same set of clauses. But we assume that no clause contains both x_i and $\neg x_i$ and that either x_i or $\neg x_i$ appears in some clause, and so there must be some clause C_j for which v_i and v'_i differ.

- For each clause C_j, set S contains two integers s_j and s'_j. Each of s_j and s'_j has 0s in all digits other than the one labeled by C_j. For s_j, there is a 1 in the C_j digit, and s'_j has a 2 in this digit. These integers are "slack variables," which we use to get each clause-labeled digit position to add to the target value of 4.

  Simple inspection of Figure 34.19 demonstrates that all s_j and s'_j values in S are unique in set S.

The greatest sum of digits in any one digit position is 6, which occurs in the digits labeled by clauses (three 1s from the v_i and v'_i values, plus 1 and 2 from the s_j and s'_j values). Interpreting these numbers in base 10, therefore, no carries can occur from lower digits to higher digits.[12]

The reduction can be performed in polynomial time. The set S consists of $2n + 2k$ values, each of which has $n + k$ digits, and the time to produce each digit is polynomial in $n + k$. The target t has $n + k$ digits, and the reduction produces each in constant time.

Let's now show that the 3-CNF formula ϕ is satisfiable if and only if there exists a subset $S' \subseteq S$ whose sum is t. First, suppose that ϕ has a satisfying assignment. For $i = 1, 2, \ldots, n$, if $x_i = 1$ in this assignment, then include v_i in S'. Otherwise, include v'_i. In other words, S' includes exactly the v_i and v'_i values that correspond to literals with the value 1 in the satisfying assignment. Having included either v_i or v'_i, but not both, for all i, and having put 0 in the digits labeled by variables in all s_j and s'_j, we see that for each variable-labeled digit, the sum of the values of S' must be 1, which matches those digits of the target t. Because each clause is satisfied, the clause contains some literal with the value 1. Therefore, each digit labeled by a clause has at least one 1 contributed to its sum by a v_i or v'_i value in S'. In fact, one, two, or three literals may be 1 in each clause, and so each clause-labeled digit has a sum of 1, 2, or 3 from the v_i and v'_i values in S'. In Figure 34.19 for example, literals $\neg x_1$, $\neg x_2$, and x_3 have the value 1 in a satisfying assignment. Each of clauses C_1 and C_4 contains exactly one of these literals, and so together v'_1, v'_2, and v_3 contribute 1 to the sum in the digits for C_1 and C_4.

---

[12] In fact, any base $b \geq 7$ works. The instance at the beginning of this subsection is the set S and target t in Figure 34.19 interpreted in base 7, with S listed in sorted order.

Clause C_2 contains two of these literals, and v'_1, v'_2, and v_3 contribute 2 to the sum in the digit for C_2. Clause C_3 contains all three of these literals, and v'_1, v'_2, and v_3 contribute 3 to the sum in the digit for C_3. To achieve the target of 4 in each digit labeled by clause C_j, include in S' the appropriate nonempty subset of slack variables $\{s_j, s'_j\}$. In Figure 34.19, S' includes s_1, s'_1, s'_2, s_3, s_4, and s'_4. Since S' matches the target in all digits of the sum, and no carries can occur, the values of S' sum to t.

Now suppose that some subset $S' \subseteq S$ sums to t. The subset S' must include exactly one of v_i and v'_i for each $i = 1, 2, \ldots, n$, for otherwise the digits labeled by variables would not sum to 1. If $v_i \in S'$, then set $x_i = 1$. Otherwise, $v'_i \in S'$, and set $x_i = 0$. We claim that every clause C_j, for $j = 1, 2, \ldots, k$, is satisfied by this assignment. To prove this claim, note that to achieve a sum of 4 in the digit labeled by C_j, the subset S' must include at least one v_i or v'_i value that has a 1 in the digit labeled by C_j, since the contributions of the slack variables s_j and s'_j together sum to at most 3. If S' includes a v_i that has a 1 in C_j's position, then the literal x_i appears in clause C_j. Since $x_i = 1$ when $v_i \in S'$, clause C_j is satisfied. If S' includes a v'_i that has a 1 in that position, then the literal $\neg x_i$ appears in C_j. Since $x_i = 0$ when $v'_i \in S'$, clause C_j is again satisfied. Thus, all clauses of ϕ are satisfied, which completes the proof. ■

34.5.6 Reduction strategies

From the reductions in this section, you can see that no single strategy applies to all NP-complete problems. Some reductions are straightforward, such as reducing the hamiltonian-cycle problem to the traveling-salesperson problem. Others are considerably more complicated. Here are a few things to keep in mind and some strategies that you can often bring to bear.

Pitfalls

Make sure that you don't get the reduction backward. That is, in trying to show that problem Y is NP-complete, you might take a known NP-complete problem X and give a polynomial-time reduction from Y to X. That is the wrong direction. The reduction should be from X to Y, so that a solution to Y gives a solution to X.

Remember also that reducing a known NP-complete problem X to a problem Y does not in itself prove that Y is NP-complete. It proves that Y is NP-hard. In order to show that Y is NP-complete, you additionally need to prove that it's in NP by showing how to verify a certificate for Y in polynomial time.

Go from general to specific

When reducing problem X to problem Y, you always have to start with an arbitrary input to problem X. But you are allowed to restrict the input to problem Y as much as you like. For example, when reducing 3-CNF satisfiability to the subset-sum problem, the reduction had to be able to handle *any* 3-CNF formula as its input, but the input to the subset-sum problem that it produced had a particular structure: $2n + 2k$ integers in the set, and each integer was formed in a particular way. The reduction did not need to produce *every* possible input to the subset-sum problem. The point is that one way to solve the 3-CNF satisfiability problem transforms the input into an input to the subset-sum problem and then uses the answer to the subset-sum problem as the answer to the 3-CNF satisfiability problem.

Take advantage of structure in the problem you are reducing from

When you are choosing a problem to reduce from, you might consider two problems in the same domain, but one problem has more structure than the other. For example, it's almost always much easier to reduce from 3-CNF satisfiability than to reduce from formula satisfiability. Boolean formulas can be arbitrarily complicated, but you can exploit the structure of 3-CNF formulas when reducing.

Likewise, it is usually more straightforward to reduce from the hamiltonian-cycle problem than from the traveling-salesperson problem, even though they are so similar. That's because you can view the hamiltonian-cycle problem as taking a complete graph but with edge weights of just 0 or 1, as they would appear in the adjacency matrix. In that sense, the hamiltonian-cycle problem has more structure than the traveling-salesperson problem, in which edge weights are unrestricted.

Look for special cases

Several NP-complete problems are just special cases of other NP-complete problems. For example, consider the decision version of the 0-1 knapsack problem: given a set of n items, each with a weight and a value, does there exist a subset of items whose total weight is at most a given weight W and whose total value is at least a given value V? You can view the set-partition problem in Exercise 34.5-5 as a special case of the 0-1 knapsack problem: let the value of each item equal its weight, and set both W and V to half the total weight. If problem X is NP-hard and it is a special case of problem Y, then problem Y must be NP-hard as well. That is because a polynomial-time solution for problem Y automatically gives a polynomial-time solution for problem X. More intuitively, problem Y, being more general than problem X, is at least as hard.

Select an appropriate problem to reduce from

It's often a good strategy to reduce from a problem in a domain that is the same as, or at least related to, the domain of the problem that you're trying to prove NP-complete. For example, we saw that the vertex-cover problem—a graph problem—was NP-hard by reducing from the clique problem—also a graph problem. From the vertex-cover problem, we reduced to the hamiltonian-cycle problem, and from the hamiltonian-cycle problem, we reduced to the traveling-salesperson problem. All of these problems take undirected graphs as inputs.

Sometimes, however, you will find that is it better to cross over from one domain to another, such as when we reduced from 3-CNF satisfiability to the clique problem or to the subset-sum problem. 3-CNF satisfiability often turns out to be a good choice as a problem to reduce from when crossing domains.

Within graph problems, if you need to select a portion of the graph, without regard to ordering, then the vertex-cover problem is often a good place to start. If ordering matters, then consider starting from the hamiltonian-cycle or hamiltonian-path problem (see Exercise 34.5-6).

Make big rewards and big penalties

The strategy for reducing the hamiltonian-cycle problem with a graph G to the traveling-salesperson problem encouraged using edges present in G when choosing edges for the traveling-salesperson tour. The reduction did so by giving these edges a low weight: 0. In other words, we gave a big reward for using these edges.

Alternatively, the reduction could have given the edges in G a finite weight and given edges not in G infinite weight, thereby exacting a hefty penalty for using edges not in G. With this approach, if each edge in G has weight W, then the target weight of the traveling-salesperson tour becomes $W \cdot |V|$. You can sometimes think of the penalties as a way to enforce requirements. For example, if the traveling-salesperson tour includes an edge with infinite weight, then it violates the requirement that the tour should include only edges belonging to G.

Design gadgets

The reduction from the vertex-cover problem to the hamiltonian-cycle problem uses the gadget shown in Figure 34.16. This gadget is a subgraph that is connected to other parts of the constructed graph in order to restrict the ways that a cycle can visit each vertex in the gadget once. More generally, a gadget is a component that enforces certain properties. Gadgets can be complicated, as in the reduction to the hamiltonian-cycle problem. Or they can be simple: in the reduction of 3-CNF satisfiability to the subset-sum problem, you can view the slack variables s_j and s'_j

as gadgets enabling each clause-labeled digit position to achieve the target value of 4.

Exercises

34.5-1

The ***subgraph-isomorphism problem*** takes two undirected graphs G_1 and G_2, and asks whether G_1 is isomorphic to a subgraph of G_2. Show that the subgraph-isomorphism problem is NP-complete.

34.5-2

Given an integer $m \times n$ matrix A and an integer m-vector b, the ***0-1 integer-programming problem*** asks whether there exists an integer n-vector x with elements in the set $\{0, 1\}$ such that $Ax \le b$. Prove that 0-1 integer programming is NP-complete. (*Hint:* Reduce from 3-CNF-SAT.)

34.5-3

The ***integer linear-programming problem*** is like the 0-1 integer-programming problem given in Exercise 34.5-2, except that the values of the vector x may be any integers rather than just 0 or 1. Assuming that the 0-1 integer-programming problem is NP-hard, show that the integer linear-programming problem is NP-complete.

34.5-4

Show how to solve the subset-sum problem in polynomial time if the target value t is expressed in unary.

34.5-5

The ***set-partition problem*** takes as input a set S of numbers. The question is whether the numbers can be partitioned into two sets A and $\overline{A} = S - A$ such that $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$. Show that the set-partition problem is NP-complete.

34.5-6

Show that the hamiltonian-path problem is NP-complete.

34.5-7

The ***longest-simple-cycle problem*** is the problem of determining a simple cycle (no repeated vertices) of maximum length in a graph. Formulate a related decision problem, and show that the decision problem is NP-complete.

34.5-8

In the ***half 3-CNF satisfiability*** problem, the input is a 3-CNF formula ϕ with n variables and m clauses, where m is even. The question is whether there exists a truth assignment to the variables of ϕ such that exactly half the clauses evaluate to 0 and exactly half the clauses evaluate to 1. Prove that the half 3-CNF satisfiability problem is NP-complete.

34.5-9

The proof that VERTEX-COVER $\leq_{\mathrm{P}}$ HAM-CYCLE assumes that the graph G given as input to the vertex-cover problem has no isolated vertices. Show how the reduction in the proof can break down if G has an isolated vertex.

Problems

34-1 Independent set

An ***independent set*** of a graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices such that each edge in E is incident on at most one vertex in V'. The ***independent-set problem*** is to find a maximum-size independent set in G.

a. Formulate a related decision problem for the independent-set problem, and prove that it is NP-complete. (*Hint:* Reduce from the clique problem.)

b. You are given a "black-box" subroutine to solve the decision problem you defined in part (a). Give an algorithm to find an independent set of maximum size. The running time of your algorithm should be polynomial in $|V|$ and $|E|$, counting queries to the black box as a single step.

Although the independent-set decision problem is NP-complete, certain special cases are polynomial-time solvable.

c. Give an efficient algorithm to solve the independent-set problem when each vertex in G has degree 2. Analyze the running time, and prove that your algorithm works correctly.

d. Give an efficient algorithm to solve the independent-set problem when G is bipartite. Analyze the running time, and prove that your algorithm works correctly. (*Hint:* First prove that in a bipartite graph, the size of the maximimum independent set plus the size of the maximum matching is equal to $|V|$. Then use a maximum-matching algorithm (see Section 25.1) as a first step in an algorithm to find an independent set.)

34-2 Bonnie and Clyde

Bonnie and Clyde have just robbed a bank. They have a bag of money and want to divide it up. For each of the following scenarios, either give a polynomial-time algorithm to divide the money or prove that the problem of dividing the money in the manner described is NP-complete. The input in each case is a list of the n items in the bag, along with the value of each.

a. The bag contains n coins, but only two different denominations: some coins are worth x dollars, and some are worth y dollars. Bonnie and Clyde wish to divide the money exactly evenly.

b. The bag contains n coins, with an arbitrary number of different denominations, but each denomination is a nonnegative exact power of 2, so that the possible denominations are 1 dollar, 2 dollars, 4 dollars, etc. Bonnie and Clyde wish to divide the money exactly evenly.

c. The bag contains n checks, which are, in an amazing coincidence, made out to "Bonnie or Clyde." They wish to divide the checks so that they each get the exact same amount of money.

d. The bag contains n checks as in part (c), but this time Bonnie and Clyde are willing to accept a split in which the difference is no larger than 100 dollars.

34-3 Graph coloring

Mapmakers try to use as few colors as possible when coloring countries on a map, subject to the restriction that if two countries share a border, they must have different colors. You can model this problem with an undirected graph $G = (V, E)$ in which each vertex represents a country and vertices whose respective countries share a border are adjacent. Then, a ***k-coloring*** is a function $c : V \to \{1, 2, \ldots, k\}$ such that $c(u) \neq c(v)$ for every edge $(u, v) \in E$. In other words, the numbers $1, 2, \ldots, k$ represent the k colors, and adjacent vertices must have different colors. The ***graph-coloring problem*** is to determine the minimum number of colors needed to color a given graph.

a. Give an efficient algorithm to determine a 2-coloring of a graph, if one exists.

b. Cast the graph-coloring problem as a decision problem. Show that your decision problem is solvable in polynomial time if and only if the graph-coloring problem is solvable in polynomial time.

c. Let the language 3-COLOR be the set of graphs that can be 3-colored. Show that if 3-COLOR is NP-complete, then your decision problem from part (b) is NP-complete.

Figure 34.20 The subgraph of G in Problem 34-3 formed by the literal edges. The special vertices TRUE, FALSE, and RED form a triangle, and for each variable x_i, the vertices x_i, $\neg x_i$, and RED form a triangle.

Figure 34.21 The gadget corresponding to a clause $(x \vee y \vee z)$, used in Problem 34-3.

To prove that 3-COLOR is NP-complete, you can reduce from 3-CNF-SAT. Given a formula ϕ of m clauses on n variables $x_1, x_2, \ldots, x_n$, construct a graph $G = (V, E)$ as follows. The set V consists of a vertex for each variable, a vertex for the negation of each variable, five vertices for each clause, and three special vertices: TRUE, FALSE, and RED. The edges of the graph are of two types: "literal" edges that are independent of the clauses and "clause" edges that depend on the clauses. As Figure 34.20 shows, the literal edges form a triangle on the three special vertices TRUE, FALSE, and RED, and they also form a triangle on x_i, $\neg x_i$, and RED for $i = 1, 2, \ldots, n$.

d. Consider a graph containing the literal edges. Argue that in any 3-coloring c of such a graph, exactly one of a variable and its negation is colored $c(\text{TRUE})$ and the other is colored $c(\text{FALSE})$. Then argue that for any truth assignment for ϕ, there exists a 3-coloring of the graph containing just the literal edges.

The gadget shown in Figure 34.21 helps to enforce the condition corresponding to a clause $(x \vee y \vee z)$, where x, y, and z are literals. Each clause requires a unique copy of the five blue vertices in the figure. They connect as shown to the literals of the clause and the special vertex TRUE.

e. Argue that if each of x, y, and z is colored $c(\text{TRUE})$ or $c(\text{FALSE})$, then the gadget is 3-colorable if and only if at least one of x, y, or z is colored $c(\text{TRUE})$.

f. Complete the proof that 3-COLOR is NP-complete.

34-4 Scheduling with profits and deadlines

You have one computer and a set of n tasks $\{a_1, a_2, \ldots, a_n\}$ requiring time on the computer. Each task a_j requires t_j time units on the computer (its processing time), yields a profit of p_j, and has a deadline d_j. The computer can process only one task at a time, and task a_j must run without interruption for t_j consecutive time units. If task a_j completes by its deadline d_j, you receive a profit p_j. If instead task a_j completes after its deadline, you receive no profit. As an optimization problem, given the processing times, profits, and deadlines for a set of n tasks, you wish to find a schedule that completes all the tasks and returns the greatest amount of profit. The processing times, profits, and deadlines are all nonnegative numbers.

a. State this problem as a decision problem.

b. Show that the decision problem is NP-complete.

c. Give a polynomial-time algorithm for the decision problem, assuming that all processing times are integers from 1 to n. (*Hint:* Use dynamic programming.)

d. Give a polynomial-time algorithm for the optimization problem, assuming that all processing times are integers from 1 to n.

Chapter notes

The book by Garey and Johnson [176] provides a wonderful guide to NP-completeness, discussing the theory at length and providing a catalogue of many problems that were known to be NP-complete in 1979. The proof of Theorem 34.13 is adapted from their book, and the list of NP-complete problem domains at the beginning of Section 34.5 is drawn from their table of contents. Johnson wrote a series of 23 columns in the *Journal of Algorithms* between 1981 and 1992 reporting new developments in NP-completeness. Fortnow's book [152] gives a history of NP-completeness, along with societal implications. Hopcroft, Motwani, and Ullman [225], Lewis and Papadimitriou [299], Papadimitriou [352], and Sipser [413] have good treatments of NP-completeness in the context of complexity theory. NP-completeness and several reductions also appear in books by Aho, Hopcroft, and Ullman [5], Dasgupta, Papadimitriou, and Vazirani [107], Johnsonbaugh and

Schaefer [239], and Kleinberg and Tardos [257]. The book by Hromkovič [229] studies various methods for solving hard problems.

The class P was introduced in 1964 by Cobham [96] and, independently, in 1965 by Edmonds [130], who also introduced the class NP and conjectured that $P \neq NP$. The notion of NP-completeness was proposed in 1971 by Cook [100], who gave the first NP-completeness proofs for formula satisfiability and 3-CNF satisfiability. Levin [297] independently discovered the notion, giving an NP-completeness proof for a tiling problem. Karp [248] introduced the methodology of reductions in 1972 and demonstrated the rich variety of NP-complete problems. Karp's paper included the original NP-completeness proofs of the clique, vertex-cover, and hamiltonian-cycle problems. Since then, thousands of problems have been proven to be NP-complete by many researchers.

Work in complexity theory has shed light on the complexity of computing approximate solutions. This work gives a new definition of NP using "probabilistically checkable proofs." This new definition implies that for problems such as clique, vertex cover, the traveling-salesperson problem with the triangle inequality, and many others, computing good approximate solutions (see Chapter 35) is NP-hard and hence no easier than computing optimal solutions. An introduction to this area can be found in Arora's thesis [21], a chapter by Arora and Lund in Hochbaum [221], a survey article by Arora [22], a book edited by Mayr, Prömel, and Steger [319], a survey article by Johnson [237], and a chapter in the textbook by Arora and Barak [24].

35 Approximation Algorithms

Many problems of practical significance are NP-complete, yet they are too important to abandon merely because nobody knows how to find an optimal solution in polynomial time. Even if a problem is NP-complete, there may be hope. You have at least three options to get around NP-completeness. First, if the actual inputs are small, an algorithm with exponential running time might be fast enough. Second, you might be able to isolate important special cases that you can solve in polynomial time. Third, you can try to devise an approach to find a *near-optimal* solution in polynomial time (either in the worst case or the expected case). In practice, near-optimality is often good enough. We call an algorithm that returns near-optimal solutions an ***approximation algorithm***. This chapter presents polynomial-time approximation algorithms for several NP-complete problems.

Performance ratios for approximation algorithms

Suppose that you are working on an optimization problem in which each potential solution has a positive cost, and you want to find a near-optimal solution. Depending on the problem, you could define an optimal solution as one with maximum possible cost or as one with minimum possible cost, which is to say that the problem might be either a maximization or a minimization problem.

We say that an algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \le \rho(n). \tag{35.1}$$

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a ***$\rho(n)$-approximation algorithm***. The definitions of approximation ratio and $\rho(n)$-approximation algorithm apply to both minimization and maximization problems. For a maximization problem, $0 < C \le C^*$, and the ratio C^*/C gives the factor by which

the cost of an optimal solution is larger than the cost of the approximate solution. Similarly, for a minimization problem, $0 < C^* \leq C$, and the ratio C/C^* gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution. Because we assume that all solutions have positive cost, these ratios are always well defined. The approximation ratio of an approximation algorithm is never less than 1, since $C/C^* \leq 1$ implies $C^*/C \geq 1$. Therefore, a 1-approximation algorithm[1] produces an optimal solution, and an approximation algorithm with a large approximation ratio may return a solution that is much worse than optimal.

For many problems, we know of polynomial-time approximation algorithms with small constant approximation ratios, although for other problems, the best known polynomial-time approximation algorithms have approximation ratios that grow as functions of the input size n. An example of such a problem is the set-cover problem presented in Section 35.3.

Some polynomial-time approximation algorithms can achieve increasingly better approximation ratios by using more and more computation time. For such problems, you can trade computation time for the quality of the approximation. An example is the subset-sum problem studied in Section 35.5. This situation is important enough to deserve a name of its own.

An ***approximation scheme*** for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is a $(1 + \epsilon)$-approximation algorithm. We say that an approximation scheme is a ***polynomial-time approximation scheme*** if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.

The running time of a polynomial-time approximation scheme can increase very rapidly as ϵ decreases. For example, the running time of a polynomial-time approximation scheme might be $O(n^{2/\epsilon})$. Ideally, if ϵ decreases by a constant factor, the running time to achieve the desired approximation should not increase by more than a constant factor (though not necessarily the same constant factor by which ϵ decreased).

We say that an approximation scheme is a ***fully polynomial-time approximation scheme*** if it is an approximation scheme and its running time is polynomial in both $1/\epsilon$ and the size n of the input instance. For example, the scheme might have a running time of $O((1/\epsilon)^2 n^3)$. With such a scheme, any constant-factor decrease in ϵ comes with a corresponding constant-factor increase in the running time.

---

[1] When the approximation ratio is independent of n, we use the terms "approximation ratio of ρ" and "ρ-approximation algorithm," indicating no dependence on n.

Chapter outline

The first four sections of this chapter present some examples of polynomial-time approximation algorithms for NP-complete problems, and the fifth section gives a fully polynomial-time approximation scheme. We begin in Section 35.1 with a study of the vertex-cover problem, an NP-complete minimization problem that has an approximation algorithm with an approximation ratio of 2. Section 35.2 looks at a version of the traveling-salesperson problem in which the cost function satisfies the triangle inequality and presents an approximation algorithm with an approximation ratio of 2. The section also shows that without the triangle inequality, for any constant $\rho \geq 1$, a ρ-approximation algorithm cannot exist unless $\mathrm{P} = \mathrm{NP}$. Section 35.3 applies a greedy method as an effective approximation algorithm for the set-covering problem, obtaining a covering whose cost is at worst a logarithmic factor larger than the optimal cost. Section 35.4 uses randomization and linear programming to develop two more approximation algorithms. The section first defines the optimization version of 3-CNF satisfiability and gives a simple randomized algorithm that produces a solution with an expected approximation ratio of $8/7$. Then Section 35.4 examines a weighted variant of the vertex-cover problem and exhibits how to use linear programming to develop a 2-approximation algorithm. Finally, Section 35.5 presents a fully polynomial-time approximation scheme for the subset-sum problem.

35.1 The vertex-cover problem

Section 34.5.2 defined the vertex-cover problem and proved it NP-complete. Recall that a ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover is the number of vertices in it.

The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an ***optimal vertex cover***. This problem is the optimization version of an NP-complete decision problem.

Even though nobody knows how to find an optimal vertex cover in a graph G in polynomial time, there is an efficient algorithm to find a vertex cover that is near-optimal. The approximation algorithm APPROX-VERTEX-COVER on the facing page takes as input an undirected graph G and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.

Figure 35.1 illustrates how APPROX-VERTEX-COVER operates on an example graph. The variable C contains the vertex cover being constructed. Line 1 initializes C to the empty set. Line 2 sets E' to be a copy of the edge set $G.E$ of the graph. The **while** loop of lines 3–6 repeatedly picks an edge (u, v) from E', adds

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' edge (u, v) and every edge incident on either u or v
7  return C
```

its endpoints u and v into C, and deletes all edges in E' that u or v covers. Finally, line 7 returns the vertex cover C. The running time of this algorithm is $O(V + E)$, using adjacency lists to represent E'.

Theorem 35.1
APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof We have already shown that APPROX-VERTEX-COVER runs in polynomial time.

The set C of vertices that is returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in $G.E$ has been covered by some vertex in C.

To see that APPROX-VERTEX-COVER returns a vertex cover that is at most twice the size of an optimal cover, let A denote the set of edges that line 4 of APPROX-VERTEX-COVER picked. In order to cover the edges in A, any vertex cover—in particular, an optimal cover C^*—must include at least one endpoint of each edge in A. No two edges in A share an endpoint, since once an edge is picked in line 4, all other edges that are incident on its endpoints are deleted from E' in line 6. Thus, no two edges in A are covered by the same vertex from C^*, meaning that for every vertex in C^*, there is at most one edge in A, giving the lower bound

$$|C^*| \geq |A| \tag{35.2}$$

on the size of an optimal vertex cover. Each execution of line 4 picks an edge for which neither of its endpoints is already in C, yielding an upper bound (an exact upper bound, in fact) on the size of the vertex cover returned:

$$|C| = 2\,|A| \ . \tag{35.3}$$

Combining equations (35.2) and (35.3) yields

$$\begin{aligned} |C| &= 2\,|A| \\ &\leq 2\,|C^*| \ , \end{aligned}$$

thereby proving the theorem. ■

Figure 35.1 The operation of APPROX-VERTEX-COVER. **(a)** The input graph G, which has 7 vertices and 8 edges. **(b)** The highlighted edge (b, c) is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, in blue, are added to the set C containing the vertex cover being created. Dashed edges (a, b), (c, e), and (c, d) are removed since they are now covered by some vertex in C. **(c)** Edge (e, f) is chosen, and vertices e and f are added to C. **(d)** Edge (d, g) is chosen, and vertices d and g are added to C. **(e)** The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b, c, d, e, f, g. **(f)** The optimal vertex cover for this problem contains only three vertices: b, d, and e.

Let us reflect on this proof. At first, you might wonder how you can possibly prove that the size of the vertex cover returned by APPROX-VERTEX-COVER is at most twice the size of an optimal vertex cover, when you don't even know the size of an optimal vertex cover. Instead of requiring that you know the exact size of an optimal vertex cover, you find a lower bound on the size. As Exercise 35.1-2 asks you to show, the set A of edges that line 4 of APPROX-VERTEX-COVER selects is actually a maximal matching in the graph G. (A ***maximal matching*** is a matching to which no edges can be added and still have a matching.) The size of a maximal matching is, as we argued in the proof of Theorem 35.1, a lower bound on the size of an optimal vertex cover. The algorithm returns a vertex cover whose size is at most twice the size of the maximal matching A. The approximation ratio comes from relating the size of the solution returned to the lower bound. We will use this methodology in later sections as well.

Exercises

35.1-1
Give an example of a graph for which APPROX-VERTEX-COVER always yields a suboptimal solution.

35.1-2
Prove that the set of edges picked in line 4 of APPROX-VERTEX-COVER forms a maximal matching in the graph G.

★ ***35.1-3***
Consider the following heuristic to solve the vertex-cover problem. Repeatedly select a vertex of highest degree, and remove all of its incident edges. Give an example to show that this heuristic does not provide an approximation ratio of 2. (*Hint:* Try a bipartite graph with vertices of uniform degree on the left and vertices of varying degree on the right.)

35.1-4
Give an efficient greedy algorithm that finds an optimal vertex cover for a tree in linear time.

35.1-5
The proof of Theorem 34.12 on page 1084 illustrates that the vertex-cover problem and the NP-complete clique problem are complementary in the sense that an optimal vertex cover is the complement of a maximum-size clique in the complement graph. Does this relationship imply that there is a polynomial-time approximation algorithm with a constant approximation ratio for the clique problem? Justify your answer.

35.2 The traveling-salesperson problem

The input to the traveling-salesperson problem, introduced in Section 34.5.4, is a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$. The goal is to find a hamiltonian cycle (a tour) of G with minimum cost. As an extension of our notation, let $c(A)$ denote the total cost of the edges in the subset $A \subseteq E$:

$$c(A) = \sum_{(u,v) \in A} c(u, v) .$$

In many practical situations, the least costly way to go from a place u to a place w is to go directly, with no intermediate steps. Put another way, cutting out an intermediate stop never increases the cost. Such a cost function c satisfies the ***triangle inequality***: for all vertices $u, v, w \in V$,

$$c(u, w) \le c(u, v) + c(v, w) .$$

The triangle inequality seems as though it should naturally hold, and it is automatically satisfied in several applications. For example, if the vertices of the graph are points in the plane and the cost of traveling between two vertices is the ordinary euclidean distance between them, then the triangle inequality is satisfied. Furthermore, many cost functions other than euclidean distance satisfy the triangle inequality.

As Exercise 35.2-2 shows, the traveling-salesperson problem is NP-complete even if you require the cost function to satisfy the triangle inequality. Thus, you should not expect to find a polynomial-time algorithm for solving this problem exactly. Your time would be better spent looking for good approximation algorithms.

In Section 35.2.1, we examine a 2-approximation algorithm for the traveling-salesperson problem with the triangle inequality. In Section 35.2.2, we show that without the triangle inequality, a polynomial-time approximation algorithm with a constant approximation ratio does not exist unless P = NP.

35.2.1 The traveling-salesperson problem with the triangle inequality

Applying the methodology of the previous section, start by computing a structure —a minimum spanning tree—whose weight gives a lower bound on the length of an optimal traveling-salesperson tour. Then use the minimum spanning tree to create a tour whose cost is no more than twice that of the minimum spanning tree's weight, as long as the cost function satisfies the triangle inequality. The procedure APPROX-TSP-TOUR on the next page implements this approach, calling the minimum-spanning-tree algorithm MST-PRIM on page 596 as a subroutine. The parameter G is a complete undirected graph, and the cost function c satisfies the triangle inequality.

Recall from Section 12.1 that a preorder tree walk recursively visits every vertex in the tree, listing a vertex when it is first encountered, before visiting any of its children.

Figure 35.2 illustrates the operation of APPROX-TSP-TOUR. Part (a) of the figure shows a complete undirected graph, and part (b) shows the minimum spanning tree T grown from root vertex a by MST-PRIM. Part (c) shows how a preorder walk of T visits the vertices, and part (d) displays the corresponding tour, which is the tour returned by APPROX-TSP-TOUR. Part (e) displays an optimal tour, which is about 23% shorter.

Figure 35.2 The operation of APPROX-TSP-TOUR. **(a)** A complete undirected graph. Vertices lie on intersections of integer grid lines. For example, f is one unit to the right and two units up from h. The cost function between two points is the ordinary euclidean distance. **(b)** A minimum spanning tree T of the complete graph, as computed by MST-PRIM. Vertex a is the root vertex. Only edges in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order. **(c)** A walk of T, starting at a. A full walk of the tree visits the vertices in the order $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g. **(d)** A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour H returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074. **(e)** An optimal tour H^* for the original complete graph. Its total cost is approximately 14.715.

```
APPROX-TSP-TOUR(G, c)
select a vertex r ∈ G.V to be a "root" vertex
compute a minimum spanning tree T for G from root r
    using MST-PRIM(G, c, r)
let H be a list of vertices, ordered according to when they are first visited
    in a preorder tree walk of T
return the hamiltonian cycle H
```

By Exercise 21.2-2, even with a simple implementation of MST-PRIM, the running time of APPROX-TSP-TOUR is $\Theta(V^2)$. We now show that if the cost function for an instance of the traveling-salesperson problem satisfies the triangle inequality, then APPROX-TSP-TOUR returns a tour whose cost is at most twice the cost of an optimal tour.

Theorem 35.2
When the triangle inequality holds, APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesperson problem.

Proof We have already seen that APPROX-TSP-TOUR runs in polynomial time.

Let H^* denote an optimal tour for the given set of vertices. Deleting any edge from a tour yields a spanning tree, and each edge cost is nonnegative. Therefore, the weight of the minimum spanning tree T computed in line 2 of APPROX-TSP-TOUR provides a lower bound on the cost of an optimal tour:

$$c(T) \le c(H^*) . \tag{35.4}$$

A ***full walk*** of T lists the vertices when they are first visited and also whenever they are returned to after a visit to a subtree. Let's call this full walk W. The full walk of our example gives the order

$$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a .$$

Since the full walk traverses every edge of T exactly twice, by extending the definition of the cost c in the natural manner to handle multisets of edges, we have

$$c(W) = 2c(T) . \tag{35.5}$$

Inequality (35.4) and equation (35.5) imply that

$$c(W) \le 2c(H^*) , \tag{35.6}$$

and so the cost of W is within a factor of 2 of the cost of an optimal tour.

Of course, the full walk W is not a tour, since it visits some vertices more than once. By the triangle inequality, however, deleting a visit to any vertex from W does not increase the cost. (When a vertex v is deleted from W between visits to u and w, the resulting ordering specifies going directly from u to w.) Repeatedly apply this operation on each visit to a vertex after the first time it's visited in W, so that W is left with only the first visit to each vertex. In our example, this process leaves the ordering

$$a, b, c, h, d, e, f, g .$$

This ordering is the same as that obtained by a preorder walk of the tree T. Let H be the cycle corresponding to this preorder walk. It is a hamiltonian cycle, since ev-

ery vertex is visited exactly once, and in fact it is the cycle computed by APPROX-TSP-TOUR. Since H is obtained by deleting vertices from the full walk W, we have

$$c(H) \le c(W) . \tag{35.7}$$

Combining inequalities (35.6) and (35.7) gives $c(H) \le 2c(H^*)$, which completes the proof. ■

Despite the small approximation ratio provided by Theorem 35.2, APPROX-TSP-TOUR is usually not the best practical choice for this problem. There are other approximation algorithms that typically perform much better in practice. (See the references at the end of this chapter.)

35.2.2 The general traveling-salesperson problem

When the cost function c does not satisfy the triangle inequality, there is no way to find good approximate tours in polynomial time unless P = NP.

Theorem 35.3
If P $\neq$ NP, then for any constant $\rho \ge 1$, there is no polynomial-time approximation algorithm with approximation ratio ρ for the general traveling-salesperson problem.

Proof The proof is by contradiction. Suppose to the contrary that for some number $\rho \ge 1$, there is a polynomial-time approximation algorithm A with approximation ratio ρ. Without loss of generality, assume that ρ is an integer, by rounding it up if necessary. We will show how to use A to solve instances of the hamiltonian-cycle problem (defined in Section 34.2) in polynomial time. Since Theorem 34.13 on page 1085 says that the hamiltonian-cycle problem is NP-complete, Theorem 34.4 on page 1063 implies that if it has a polynomial-time algorithm, then P = NP.

Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem. We will show how to determine efficiently whether G contains a hamiltonian cycle by making use of the hypothesized approximation algorithm A. Convert G into an instance of the traveling-salesperson problem as follows. Let $G' = (V, E')$ be the complete graph on V, that is,

$$E' = \{(u, v) : u, v \in V \text{ and } u \neq v\} .$$

Assign an integer cost to each edge in E' as follows:

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E , \\ \rho |V| + 1 & \text{otherwise} . \end{cases}$$

Given a representation of G, it takes time polynomial in $|V|$ and $|E|$ to create representations of G' and c.

Now consider the traveling-salesperson problem (G', c). If the original graph G has a hamiltonian cycle H, then the cost function c assigns to each edge of H a cost of 1, and so (G', c) contains a tour of cost $|V|$. On the other hand, if G does not contain a hamiltonian cycle, then any tour of G' must use some edge not in E. But any tour that uses an edge not in E has a cost of at least

$$\begin{aligned}(\rho|V| + 1) + (|V| - 1) &= \rho|V| + |V| \\ &> \rho|V| \ .\end{aligned}$$

Because edges not in G are so costly, there is a gap of at least $\rho|V|$ between the cost of a tour that is a hamiltonian cycle in G (cost $|V|$) and the cost of any other tour (cost at least $\rho|V| + |V|$). Therefore, the cost of a tour that is not a hamiltonian cycle in G is at least a factor of $\rho + 1$ greater than the cost of a tour that is a hamiltonian cycle in G.

What happens upon applying the approximation algorithm A to the traveling-salesperson problem (G', c)? Because A is guaranteed to return a tour of cost no more than ρ times the cost of an optimal tour, if G contains a hamiltonian cycle, then A must return it. If G has no hamiltonian cycle, then A returns a tour of cost more than $\rho|V|$. Therefore, using A solves the hamiltonian-cycle problem in polynomial time. ■

The proof of Theorem 35.3 serves as an example of a general technique to prove that no good approximation algorithm exists for a particular problem. Given an NP-hard decision problem X, produce in polynomial time a minimization problem Y such that "yes" instances of X correspond to instances of Y with value at most k (for some k), but that "no" instances of X correspond to instances of Y with value greater than ρk. This technique shows that, unless P = NP, there is no polynomial-time ρ-approximation algorithm for problem Y.

Exercises

35.2-1

Let $G = (V, E)$ be a complete undirected graph containing at least 3 vertices, and let c be a cost function that satisfies the triangle inequality. Prove that $c(u, v) \geq 0$ for all $u, v \in V$.

35.2-2

Show how in polynomial time to transform one instance of the traveling-salesperson problem into another instance whose cost function satisfies the triangle inequality. The two instances must have the same set of optimal tours. Explain why

such a polynomial-time transformation does not contradict Theorem 35.3, assuming that P $\neq$ NP.

35.2-3

Consider the following ***closest-point heuristic*** for building an approximate traveling-salesperson tour whose cost function satisfies the triangle inequality. Begin with a trivial cycle consisting of a single arbitrarily chosen vertex. At each step, identify the vertex u that is not on the cycle but whose distance to any vertex on the cycle is minimum. Suppose that the vertex on the cycle that is nearest u is vertex v. Extend the cycle to include u by inserting u just after v. Repeat until all vertices are on the cycle. Prove that this heuristic returns a tour whose total cost is not more than twice the cost of an optimal tour.

35.2-4

A solution to the ***bottleneck traveling-salesperson problem*** is the hamiltonian cycle that minimizes the cost of the most costly edge in the cycle. Assuming that the cost function satisfies the triangle inequality, show that there exists a polynomial-time approximation algorithm with approximation ratio 3 for this problem. (*Hint:* Show recursively how to visit all the nodes in a bottleneck spanning tree, as discussed in Problem 21-4 on page 601, exactly once by taking a full walk of the tree and skipping nodes, but without skipping more than two consecutive intermediate nodes. Show that the costliest edge in a bottleneck spanning tree has a cost bounded from above by the cost of the costliest edge in a bottleneck hamiltonian cycle.)

35.2-5

Suppose that the vertices for an instance of the traveling-salesperson problem are points in the plane and that the cost $c(u, v)$ is the euclidean distance between points u and v. Show that an optimal tour never crosses itself.

35.2-6

Adapt the proof of Theorem 35.3 to show that for any constant $c \geq 0$, there is no polynomial-time approximation algorithm with approximation ratio $|V|^c$ for the general traveling-salesperson problem.

35.3 The set-covering problem

The set-covering problem is an optimization problem that models many problems that require resources to be allocated. Its corresponding decision problem generalizes the NP-complete vertex-cover problem and is therefore also NP-hard. The

approximation algorithm developed to handle the vertex-cover problem doesn't apply here, however. Instead, this section investigates a simple greedy heuristic with a logarithmic approximation ratio. That is, as the size of the instance gets larger, the size of the approximate solution may grow, relative to the size of an optimal solution. Because the logarithm function grows rather slowly, however, this approximation algorithm may nonetheless give useful results.

An instance $(X, \mathcal{F})$ of the ***set-covering problem*** consists of a finite set X and a family $\mathcal{F}$ of subsets of X, such that every element of X belongs to at least one subset in $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S .$$

We say that a subfamily $\mathcal{C} \subseteq \mathcal{F}$ ***covers*** a set of elements U if

$$U \subseteq \bigcup_{S \in \mathcal{C}} S .$$

The problem is to find a minimum-size subfamily $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of X:

$$X = \bigcup_{S \in \mathcal{C}} S .$$

Figure 35.3 illustrates the set-covering problem. The size of $\mathcal{C}$ is the number of sets it contains, rather than the number of individual elements in these sets, since every subfamily $\mathcal{C}$ that covers X must contain all $|X|$ individual elements. In Figure 35.3, the minimum set cover has size 3.

The set-covering problem abstracts many commonly arising combinatorial problems. As a simple example, suppose that X represents a set of skills that are needed to solve a problem and that you have a given set of people available to work on the problem. You wish to form a committee, containing as few people as possible, such that for every requisite skill in X, at least one member of the committee has that skill. The decision version of the set-covering problem asks whether a covering exists with size at most k, where k is an additional parameter specified in the problem instance. The decision version of the problem is NP-complete, as Exercise 35.3-2 asks you to show.

A greedy approximation algorithm

The greedy method in the procedure GREEDY-SET-COVER on the facing page works by picking, at each stage, the set S that covers the greatest number of remaining elements that are uncovered. In the example of Figure 35.3, GREEDY-SET-COVER adds to $\mathcal{C}$, in order, the sets S_1, S_4, and S_5, followed by either S_3 or S_6.

Figure 35.3 An instance $(X, \mathcal{F})$ of the set-covering problem, where X consists of the 12 tan points and $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. Each set $S_i \in \mathcal{F}$ is outlined in blue. A minimum-size set cover is $\mathcal{C} = \{S_3, S_4, S_5\}$, with size 3. The greedy algorithm produces a cover of size 4 by selecting either the sets S_1, S_4, S_5, and S_3 or the sets S_1, S_4, S_5, and S_6, in order.

```
GREEDY-SET-COVER(X, F)
1  U_0 = X
2  C = ∅
3  i = 0
4  while U_i ≠ ∅
5      select S ∈ F that maximizes |S ∩ U_i|
6      U_{i+1} = U_i − S
7      C = C ∪ {S}
8      i = i + 1
9  return C
```

The greedy algorithm works as follows. At the start of each iteration, U_i is a subset of X containing the remaining uncovered elements, with the initial subset U_0 containing all the elements in X. The set $\mathcal{C}$ contains the subfamily being constructed. Line 5 is the greedy decision-making step, choosing a subset S that covers as many uncovered elements as possible (breaking ties arbitrarily). After S is selected, line 6 updates the set of remaining uncovered elements, denoting it by U_{i+1}, and line 7 places S into $\mathcal{C}$. When the algorithm terminates, $\mathcal{C}$ is a subfamily of $\mathcal{F}$ that covers X.

Analysis

We now show that the greedy algorithm returns a set cover that is not too much larger than an optimal set cover.

Theorem 35.4
The procedure GREEDY-SET-COVER run on a set X and family of subsets $\mathcal{F}$ is a polynomial-time $O(\lg X)$-approximation algorithm.

Proof Let's first show that the algorithm runs in time that is polynomial in $|X|$ and $|\mathcal{F}|$. The number of iterations of the loop in lines 4–7 is bounded above by $\min\{|X|, |\mathcal{F}|\} = O(|X| + |\mathcal{F}|)$. The loop body can be implemented to run in $O(|X| \cdot |\mathcal{F}|)$ time. Thus the algorithm runs in $O(|X| \cdot |\mathcal{F}| \cdot (|X| + |\mathcal{F}|))$ time, which is polynomial in the input size. (Exercise 35.3-3 asks for a linear-time algorithm.)

To prove the approximation bound, let $\mathcal{C}^*$ be an optimal set cover for the original instance $(X, \mathcal{F})$, and let $k = |\mathcal{C}^*|$. Since $\mathcal{C}^*$ is also a set cover of each subset U_i of X constructed by the algorithm, we know that any subset U_i constructed by the algorithm can be covered by k sets. Therefore, if $(U_i, \mathcal{F})$ is an instance of the set-covering problem, its optimal set cover has size at most k.

If an optimal set cover for an instance $(U_i, \mathcal{F})$ has size at most k, at least one of the sets in $\mathcal{C}$ covers at least $|U_i|/k$ new elements. Thus, line 5 of GREEDY-SET-COVER, which chooses a set with the maximum number of uncovered elements, must choose a set in which the number of newly covered elements is at least $|U_i|/k$. These elements are removed when constructing U_{i+1}, giving

$$\begin{aligned} |U_{i+1}| &\le |U_i| - |U_i|/k \\ &= |U_i|(1 - 1/k) . \end{aligned} \tag{35.8}$$

Iterating inequality (35.8) gives

$$\begin{aligned} |U_0| &= |X| , \\ |U_1| &\le |U_0|(1 - 1/k) , \\ |U_2| &\le |U_1|(1 - 1/k) = |U|(1 - 1/k)^2 , \end{aligned}$$

and in general

$$|U_i| \le |U_0|(1 - 1/k)^i = |X|(1 - 1/k)^i . \tag{35.9}$$

The algorithm stops when $U_i = \emptyset$, which means that $|U_i| < 1$. Thus an upper bound on the number of iterations of the algorithm is the smallest value of i for which $|U_i| < 1$.

Since $1 + x \le e^x$ for all real x (see inequality (3.14) on page 66), by letting $x = -1/k$, we have $1 - 1/k \le e^{-1/k}$, so that $(1 - 1/k)^k \le (e^{-1/k})^k = 1/e$. Denoting the number i of iterations by ck for some nonnegative integer c, we want c such that

$$|X|(1 - 1/k)^{ck} \le |X| e^{-c} < 1 . \tag{35.10}$$

Multiplying both sides by e^c and then taking the natural logarithm of both sides gives $c \ge \ln |X|$, so we can choose for c any integer that is at least $\ln |X|$. We

choose $c = \lceil \ln |X| \rceil$. Since $i = ck$ is an upper bound on the number of iterations, which equals the size of $\mathcal{C}$, and $k = |\mathcal{C}^*|$, we have $|\mathcal{C}| \le i = ck = c\,|\mathcal{C}^*| = |\mathcal{C}^*|\,\lceil \ln |X| \rceil$, and the theorem follows. ■

Exercises

35.3-1
Consider each of the following words as a set of letters: {`arid`, `dash`, `drain`, `heard`, `lost`, `nose`, `shun`, `slate`, `snare`, `thread`}. Show which set cover GREEDY-SET-COVER produces when you break ties in favor of the word that appears first in the dictionary.

35.3-2
Show that the decision version of the set-covering problem is NP-complete by reducing the vertex-cover problem to it.

35.3-3
Show how to implement GREEDY-SET-COVER to run in $O\left(\sum_{S \in \mathcal{F}} |S|\right)$ time.

35.3-4
The proof of Theorem 35.4 says that when GREEDY-SET-COVER, run on the instance $(X, \mathcal{F})$, returns the subfamily $\mathcal{C}$, then $|\mathcal{C}| \le |\mathcal{C}^*|\,\lceil \ln X \rceil$. Show that the following weaker bound is trivially true:

$$|\mathcal{C}| \le |\mathcal{C}^*| \max\{|S| : S \in \mathcal{F}\} .$$

35.3-5
GREEDY-SET-COVER can return a number of different solutions, depending on how it breaks ties in line 5. Give a procedure BAD-SET-COVER-INSTANCE(n) that returns an n-element instance of the set-covering problem for which, depending on how line 5 breaks ties, GREEDY-SET-COVER can return a number of different solutions that is exponential in n.

35.4 Randomization and linear programming

This section studies two useful techniques for designing approximation algorithms: randomization and linear programming. It starts with a simple randomized algorithm for an optimization version of 3-CNF satisfiability, and then it shows how to design an approximation algorithm for a weighted version of the vertex-cover problem based on linear programming. This section only scratches the surface of

these two powerful techniques. The chapter notes give references for further study of these areas.

A randomized approximation algorithm for MAX-3-CNF satisfiability

Just as some randomized algorithms compute exact solutions, some randomized algorithms compute approximate solutions. We say that a randomized algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the *expected* cost C of the solution produced by the randomized algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \le \rho(n). \tag{35.11}$$

We call a randomized algorithm that achieves an approximation ratio of $\rho(n)$ a ***randomized $\rho(n)$-approximation algorithm.*** In other words, a randomized approximation algorithm is like a deterministic approximation algorithm, except that the approximation ratio is for an expected cost.

A particular instance of 3-CNF satisfiability, as defined in Section 34.4, may or may not be satisfiable. In order to be satisfiable, there must exist an assignment of the variables so that every clause evaluates to 1. If an instance is not satisfiable, you might instead want to know how "close" to satisfiable it is, that is, find an assignment of the variables that satisfies as many clauses as possible. We call the resulting maximization problem ***MAX-3-CNF satisfiability***. The input to MAX-3-CNF satisfiability is the same as for 3-CNF satisfiability, and the goal is to return an assignment of the variables that maximizes the number of clauses evaluating to 1. You might be surprised that randomly setting each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ yields a randomized $8/7$-approximation algorithm, but we're about to see why. Recall that the definition of 3-CNF satisfiability from Section 34.4 requires each clause to consist of exactly three distinct literals. We now further assume that no clause contains both a variable and its negation. Exercise 35.4-1 asks you to remove this last assumption.

Theorem 35.5
Given an instance of MAX-3-CNF satisfiability with n variables $x_1, x_2, \ldots, x_n$ and m clauses, the randomized algorithm that independently sets each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ is a randomized $8/7$-approximation algorithm.

Proof Suppose that each variable is independently set to 1 with probability $1/2$ and to 0 with probability $1/2$. Define, for $i = 1, 2, \ldots, m$, the indicator random variable

$$Y_i = \mathrm{I}\{\text{clause } i \text{ is satisfied}\} ,$$

so that $Y_i = 1$ as long as at least one of the literals in the ith clause is set to 1. Since no literal appears more than once in the same clause, and since we assume that no variable and its negation appear in the same clause, the settings of the three literals in each clause are independent. A clause is not satisfied only if all three of its literals are set to 0, and so $\Pr\{\text{clause } i \text{ is not satisfied}\} = (1/2)^3 = 1/8$. Thus, we have $\Pr\{\text{clause } i \text{ is satisfied}\} = 1 - 1/8 = 7/8$, and Lemma 5.1 on page 130 gives $\mathrm{E}[Y_i] = 7/8$. Let Y be the number of satisfied clauses overall, so that $Y = Y_1 + Y_2 + \cdots + Y_m$. Then, we have

$$\begin{aligned}
\mathrm{E}[Y] &= \mathrm{E}\left[\sum_{i=1}^{m} Y_i\right] \\
&= \sum_{i=1}^{m} \mathrm{E}[Y_i] \qquad \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{m} 7/8 \\
&= 7m/8 .
\end{aligned}$$

Since m is an upper bound on the number of satisfied clauses, the approximation ratio is at most $m/(7m/8) = 8/7$. ■

Approximating weighted vertex cover using linear programming

The ***minimum-weight vertex-cover problem*** takes as input an undirected graph $G = (V, E)$ in which each vertex $v \in V$ has an associated positive weight $w(v)$. The weight $w(V')$ of a vertex cover $V' \subseteq V$ is the sum of the weights of its vertices: $w(V') = \sum_{v \in V'} w(v)$. The goal is to find a vertex cover of minimum weight.

The approximation algorithm for unweighted vertex cover from Section 35.1 won't work here, because the solution it returns could be far from optimal for the weighted problem. Instead, we'll first compute a lower bound on the weight of the minimum-weight vertex cover, by using a linear program. Then we'll "round" this solution and use it to obtain a vertex cover.

Start by associating a variable $x(v)$ with each vertex $v \in V$, and require that $x(v)$ equals either 0 or 1 for each $v \in V$. The vertex cover includes v if and only if $x(v) = 1$. Then the constraint that for any edge (u, v), at least one of u and v must belong to the vertex cover can be expressed as $x(u) + x(v) \geq 1$. This view gives rise to the following ***0-1 integer program*** for finding a minimum-weight vertex cover:

$$\text{minimize} \quad \sum_{v \in V} w(v)\, x(v) \tag{35.12}$$

subject to

$$x(u) + x(v) \;\geq\; 1 \qquad \text{for each } (u, v) \in E \tag{35.13}$$

$$x(v) \;\in\; \{0, 1\} \quad \text{for each } v \in V\ . \tag{35.14}$$

In the special case in which all the weights $w(v)$ equal 1, this formulation is the optimization version of the NP-hard vertex-cover problem. Let's remove the constraint that $x(v) \in \{0, 1\}$ and replace it by $0 \leq x(v) \leq 1$, resulting in the following linear program:

$$\text{minimize} \quad \sum_{v \in V} w(v)\, x(v) \tag{35.15}$$

subject to

$$x(u) + x(v) \;\geq\; 1 \quad \text{for each } (u, v) \in E \tag{35.16}$$

$$x(v) \;\leq\; 1 \quad \text{for each } v \in V \tag{35.17}$$

$$x(v) \;\geq\; 0 \quad \text{for each } v \in V\ . \tag{35.18}$$

We refer to this linear program as the ***linear-programming relaxation***. Any feasible solution to the 0-1 integer program in lines (35.12)–(35.14) is also a feasible solution to its linear-programming relaxation in lines (35.15)–(35.18). Therefore, the value of an optimal solution to the linear-programming relaxation provides a lower bound on the value of an optimal solution to the 0-1 integer program, and hence a lower bound on the optimal weight in the minimum-weight vertex-cover problem.

The procedure APPROX-MIN-WEIGHT-VC on the facing page starts with a solution to the linear-programming relaxation and uses it to construct an approximate solution to the minimum-weight vertex-cover problem. The procedure works as follows. Line 1 initializes the vertex cover to be empty. Line 2 formulates the linear-programming relaxation in lines (35.15)–(35.18) and then solves this linear program. An optimal solution gives each vertex v an associated value $\bar{x}(v)$, where $0 \leq \bar{x}(v) \leq 1$. The procedure uses this value to guide the choice of which vertices to add to the vertex cover C in lines 3–5: the vertex cover C includes vertex v if and only if $\bar{x}(v) \geq 1/2$. In effect, the procedure "rounds" each fractional variable in the solution to the linear-programming relaxation to either 0 or 1 in order to obtain a solution to the 0-1 integer program in lines (35.12)–(35.14). Finally, line 6 returns the vertex cover C.

Theorem 35.6

Algorithm APPROX-MIN-WEIGHT-VC is a polynomial-time2-approximation algorithm for the minimum-weight vertex-cover problem.

```
APPROX-MIN-WEIGHT-VC(G, w)
1  C = ∅
2  compute x̄, an optimal solution to the linear-programming relaxation
       in lines (35.15)–(35.18)
3  for each vertex v ∈ V
4      if x̄(v) ≥ 1/2
5          C = C ∪ {v}
6  return C
```

Proof Because there is a polynomial-time algorithm to solve the linear program in line 2, and because the **for** loop of lines 3–5 runs in polynomial time, APPROX-MIN-WEIGHT-VC is a polynomial-time algorithm.

It remains to show that APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm. Let C^* be an optimal solution to the minimum-weight vertex-cover problem, and let z^* be the value of an optimal solution to the linear-programming relaxation in lines (35.15)–(35.18). Since an optimal vertex cover is a feasible solution to the linear-programming relaxation, z^* must be a lower bound on $w(C^*)$, that is,

$$z^* \le w(C^*) . \tag{35.19}$$

Next, we claim that rounding the fractional values of the variables $\bar{x}(v)$ in lines 3–5 produces a set C that is a vertex cover and satisfies $w(C) \le 2z^*$. To see that C is a vertex cover, consider any edge $(u, v) \in E$. By constraint (35.16), we know that $x(u) + x(v) \ge 1$, which implies that at least one of $\bar{x}(u)$ and $\bar{x}(v)$ is at least $1/2$. Therefore, at least one of u and v is included in the vertex cover, and so every edge is covered.

Now we consider the weight of the cover. We have

$$\begin{aligned}
z^* &= \sum_{v \in V} w(v)\,\bar{x}(v) \\
&\ge \sum_{v \in V : \bar{x}(v) \ge 1/2} w(v)\,\bar{x}(v) \\
&\ge \sum_{v \in V : \bar{x}(v) \ge 1/2} w(v) \cdot \frac{1}{2} \\
&= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\
&= \frac{1}{2} \sum_{v \in C} w(v) \\
&= \frac{1}{2} w(C) .
\end{aligned} \tag{35.20}$$

Combining inequalities (35.19) and (35.20) gives

$$w(C) \le 2z^* \le 2w(C^*) ,$$

and hence APPROX-MIN-WEIGHT-VC is a2-approximation algorithm. ■

Exercises

35.4-1
Show that even if a clause is allowed to contain both a variable and its negation, randomly setting each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ still yields a randomized $8/7$-approximation algorithm.

35.4-2
The ***MAX-CNF satisfiability problem*** is like the MAX-3-CNF satisfiability problem, except that it does not restrict each clause to have exactly three literals. Give a randomized 2-approximation algorithm for the MAX-CNF satisfiability problem.

35.4-3
In the MAX-CUT problem, the input is an unweighted undirected graph $G = (V, E)$. We define a cut $(S, V - S)$ as in Chapter 21 and the ***weight*** of a cut as the number of edges crossing the cut. The goal is to find a cut of maximum weight. Suppose that each vertex v is randomly and independently placed into S with probability $1/2$ and into $V - S$ with probability $1/2$. Show that this algorithm is a randomized 2-approximation algorithm.

35.4-4
Show that the constraints in line (35.17) are redundant in the sense that removing them from the linear-programming relaxation in lines (35.15)–(35.18) yields a linear program for which any optimal solution x must satisfy $x(v) \le 1$ for each $v \in V$.

35.5 The subset-sum problem

Recall from Section 34.5.5 that an instance of the subset-sum problem is given by a pair (S, t), where S is a set $\{x_1, x_2, \ldots, x_n\}$ of positive integers and t is a positive integer. This decision problem asks whether there exists a subset of S that adds up exactly to the target value t. As we saw in Section 34.5.5, this problem is NP-complete.

The optimization problem associated with this decision problem arises in practical applications. The optimization problem seeks a subset of $\{x_1, x_2, \ldots, x_n\}$

whose sum is as large as possible but not larger than t. For example, consider a truck that can carry no more than t pounds, which is to be loaded with up to n different boxes, the ith of which weighs x_i pounds. How heavy a load can the truck take without exceeding the t-pound weight limit?

We start this section with an exponential-time algorithm to compute the optimal value for this optimization problem. Then we show how to modify the algorithm so that it becomes a fully polynomial-time approximation scheme. (Recall that a fully polynomial-time approximation scheme has a running time that is polynomial in $1/\epsilon$ as well as in the size of the input.)

An exponential-time exact algorithm

Suppose that you compute, for each subset S' of S, the sum of the elements in S', and then you select, among the subsets whose sum does not exceed t, the one whose sum is closest to t. This algorithm returns the optimal solution, but it might take exponential time. To implement this algorithm, you can use an iterative procedure that, in iteration i, computes the sums of all subsets of $\{x_1, x_2, \ldots, x_i\}$, using as a starting point the sums of all subsets of $\{x_1, x_2, \ldots, x_{i-1}\}$. In doing so, you would realize that once a particular subset S' has a sum exceeding t, there is no reason to maintain it, since no superset of S' can be an optimal solution. Let's see how to implement this strategy.

The procedure EXACT-SUBSET-SUM takes an input set $S = \{x_1, x_2, \ldots, x_n\}$, the size $n = |S|$, and a target value t. This procedure iteratively computes L_i, the list of sums of all subsets of $\{x_1, \ldots, x_i\}$ that do not exceed t, and then it returns the maximum value in L_n.

If L is a list of positive integers and x is another positive integer, then let $L + x$ denote the list of integers derived from L by increasing each element of L by x. For example, if $L = \langle 1, 2, 3, 5, 9 \rangle$, then $L + 2 = \langle 3, 4, 5, 7, 11 \rangle$. This notation extends to sets, so that

$$S + x = \{s + x : s \in S\} .$$

```
EXACT-SUBSET-SUM(S, n, t)
L_0 = ⟨0⟩
for i = 1 to n
    L_i = MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
    remove from L_i every element that is greater than t
return the largest element in L_n
```

EXACT-SUBSET-SUM invokes an auxiliary procedure MERGE-LISTS(L, L'), which returns the sorted list that is the merge of its two sorted input lists L and L',

with duplicate values removed. Like the MERGE procedure we used in merge sort on page 36, MERGE-LISTS runs in $O(|L| + |L'|)$ time. We omit the pseudocode for MERGE-LISTS.

To see how EXACT-SUBSET-SUM works, let P_i denote the set of values obtained by selecting each (possibly empty) subset of $\{x_1, x_2, \ldots, x_i\}$ and summing its members. For example, if $S = \{1, 4, 5\}$, then

$$\begin{aligned} P_1 &= \{0, 1\} \ , \\ P_2 &= \{0, 1, 4, 5\} \ , \\ P_3 &= \{0, 1, 4, 5, 6, 9, 10\} \ . \end{aligned}$$

Given the identity

$$P_i = P_{i-1} \cup (P_{i-1} + x_i) \ , \tag{35.21}$$

you can prove by induction on i (see Exercise 35.5-1) that the list L_i is a sorted list containing every element of P_i whose value is not more than t. Since the length of L_i can be as much as 2^i, EXACT-SUBSET-SUM is an exponential-time algorithm in general, although it is a polynomial-time algorithm in the special cases in which t is polynomial in $|S|$ or all the numbers in S are bounded by a polynomial in $|S|$.

A fully polynomial-time approximation scheme

The key to devising a fully polynomial-time approximation scheme for the subset-sum problem is to "trim" each list L_i after it is created. Here's the idea behind trimming: if two values in L are close to each other, then since the goal is just an approximate solution, there is no need to maintain both of them explicitly. More precisely, use a trimming parameter δ such that $0 < \delta < 1$. When ***trimming*** a list L by δ, remove as many elements from L as possible, in such a way that if L' is the result of trimming L, then for every element y that was removed from L, some element z still in L' approximates y. For z to approximate y, it must be no greater than y and also within a factor of $1 + \delta$ of y, so that

$$\frac{y}{1 + \delta} \le z \le y \ . \tag{35.22}$$

You can think of such a z as "representing" y in the new list L'. Each removed element y is represented by a remaining element z satisfying inequality (35.22). For example, suppose that $\delta = 0.1$ and

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \ .$$

Then trimming L results in

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle \ ,$$

where the deleted value 11 is represented by 10, the deleted values 21 and 22 are represented by 20, and the deleted value 24 is represented by 23. Because every element of the trimmed version of the list is also an element of the original version of the list, trimming can dramatically decrease the number of elements kept while keeping a close (and slightly smaller) representative value in the list for each deleted element.

The procedure TRIM trims list $L = \langle y_1, y_2, \ldots, y_m \rangle$ in $\Theta(m)$ time, given L and the trimming parameter δ. It assumes that L is sorted into monotonically increasing order. The output of the procedure is a trimmed, sorted list. The procedure scans the elements of L in monotonically increasing order. A number is appended onto the returned list L' only if it is the first element of L or if it cannot be represented by the most recent number placed into L'.

```
TRIM(L, δ)
let m be the length of L
L' = ⟨y_1⟩
last = y_1
for i = 2 to m
    if y_i > last · (1 + δ)        // y_i ≥ last because L is sorted
        append y_i onto the end of L'
        last = y_i
return L'
```

Given the procedure TRIM, the procedure APPROX-SUBSET-SUM on the following page implements the approximation scheme. This procedure takes as input a set $S = \{x_1, x_2, \ldots, x_n\}$ of n integers (in arbitrary order), the size $n = |S|$, the target integer t, and an approximation parameter ϵ, where

$$0 < \epsilon < 1 . \tag{35.23}$$

It returns a value z^* whose value is within a factor of $1 + \epsilon$ of the optimal solution.

The APPROX-SUBSET-SUM procedure works as follows. Line 1 initializes the list L_0 to be the list containing just the element 0. The **for** loop in lines 2–5 computes L_i as a sorted list containing a suitably trimmed version of the set P_i, with all elements larger than t removed. Since the procedure creates L_i from L_{i-1}, it must ensure that the repeated trimming doesn't introduce too much compounded inaccuracy. That's why instead of the trimming parameter being ϵ in the call to TRIM, it has the smaller value $\epsilon/2n$. We'll soon see that APPROX-SUBSET-SUM returns a correct approximation if one exists.

```
APPROX-SUBSET-SUM(S, n, t, ε)
1  L_0 = ⟨0⟩
2  for i = 1 to n
3      L_i = MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
4      L_i = TRIM(L_i, ε/2n)
5      remove from L_i every element that is greater than t
6  let z* be the largest value in L_n
7  return z*
```

As an example, suppose that APPROX-SUBSET-SUM is given

$$S = \langle 104, 102, 201, 101 \rangle$$

with $t = 308$ and $\epsilon = 0.40$. The trimming parameter δ is $\epsilon/2n = 0.40/8 = 0.05$. The procedure computes the following values on the indicated lines:

line 1: $L_0 = \langle 0 \rangle$,

line 3: $L_1 = \langle 0, 104 \rangle$,
line 4: $L_1 = \langle 0, 104 \rangle$,
line 5: $L_1 = \langle 0, 104 \rangle$,

line 3: $L_2 = \langle 0, 102, 104, 206 \rangle$,
line 4: $L_2 = \langle 0, 102, 206 \rangle$,
line 5: $L_2 = \langle 0, 102, 206 \rangle$,

line 3: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$,
line 4: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$,
line 5: $L_3 = \langle 0, 102, 201, 303 \rangle$,

line 3: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$,
line 4: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$,
line 5: $L_4 = \langle 0, 101, 201, 302 \rangle$.

The procedure returns $z^* = 302$ as its answer, which is well within $\epsilon = 40\%$ of the optimal answer $307 = 104 + 102 + 101$. In fact, it is within 2%.

Theorem 35.7
APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme for the subset-sum problem.

Proof The operations of trimming L_i in line 4 and removing from L_i every element that is greater than t maintain the property that every element of L_i is also a member of P_i. Therefore, the value z^* returned in line 7 is indeed the sum of some subset of S, that is, $z^* \in P_n$. Let $y^* \in P_n$ denote an optimal solution to the subset-sum problem, so that it is the greatest value in P_n that is less than or equal to t. Because line 5 ensures that $z^* \le t$, we know that $z^* \le y^*$. By inequality (35.1), we need to show that $y^*/z^* \le 1 + \epsilon$. We must also show that the running time of this algorithm is polynomial in both $1/\epsilon$ and the size of the input.

As Exercise 35.5-2 asks you to show, for every element y in P_i that is at most t, there exists an element $z \in L_i$ such that

$$\frac{y}{(1+\epsilon/2n)^i} \le z \le y \; . \tag{35.24}$$

Inequality (35.24) must hold for $y^* \in P_n$, and therefore there exists an element $z \in L_n$ such that

$$\frac{y^*}{(1+\epsilon/2n)^n} \le z \le y^* \; ,$$

and thus

$$\frac{y^*}{z} \le \left(1 + \frac{\epsilon}{2n}\right)^n \; . \tag{35.25}$$

Since there exists an element $z \in L_n$ fulfilling inequality (35.25), the inequality must hold for z^*, which is the largest value in L_n, which is to say

$$\frac{y^*}{z^*} \le \left(1 + \frac{\epsilon}{2n}\right)^n \; . \tag{35.26}$$

Now we show that $y^*/z^* \le 1+\epsilon$. We do so by showing that $(1+\epsilon/2n)^n \le 1+\epsilon$. First, inequality (35.23), $0 < \epsilon < 1$, implies that

$$(\epsilon/2)^2 \le \epsilon/2 \tag{35.27}$$

Next, from equation (3.16) on page 66, we have $\lim_{n\to\infty}(1 + \epsilon/2n)^n = e^{\epsilon/2}$. Exercise 35.5-3 asks you to show that

$$\frac{d}{dn}\left(1 + \frac{\epsilon}{2n}\right)^n > 0 \; . \tag{35.28}$$

Therefore, the function $(1 + \epsilon/2n)^n$ increases with n as it approaches its limit of $e^{\epsilon/2}$, and we have

$$\begin{aligned}
\left(1 + \frac{\epsilon}{2n}\right)^n &\le e^{\epsilon/2} \\
&\le 1 + \epsilon/2 + (\epsilon/2)^2 && \text{(by inequality (3.15) on page 66)} \\
&\le 1 + \epsilon && \text{(by inequality (35.27))} \; .
\end{aligned} \tag{35.29}$$

Combining inequalities (35.26) and (35.29) completes the analysis of the approximation ratio.

To show that APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme, we derive a bound on the length of L_i. After trimming, successive elements z and z' of L_i must have the relationship $z'/z > 1+\epsilon/2n$. That is, they must differ by a factor of at least $1 + \epsilon/2n$. Each list, therefore, contains the value 0, possibly the value 1, and up to $\lfloor \log_{1+\epsilon/2n} t \rfloor$ additional values. The number of elements in each list L_i is at most

$$\begin{aligned}
\log_{1+\epsilon/2n} t + 2 &= \frac{\ln t}{\ln(1+\epsilon/2n)} + 2 \\
&\le \frac{2n(1+\epsilon/2n)\ln t}{\epsilon} + 2 \quad \text{(by inequality (3.23) on page 67)} \\
&< \frac{3n \ln t}{\epsilon} + 2 \quad \text{(by inequality (35.23), } 0 < \epsilon < 1) .
\end{aligned}$$

This bound is polynomial in the size of the input—which is the number of bits $\lg t$ needed to represent t plus the number of bits needed to represent the set S, which in turn is polynomial in n—and in $1/\epsilon$. Since the running time of APPROX-SUBSET-SUM is polynomial in the lengths of the lists L_i, we conclude that APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme. ■

Exercises

35.5-1

Prove equation (35.21). Then show that after executing line 4 of EXACT-SUBSET-SUM, L_i is a sorted list containing every element of P_i whose value is not more than t.

35.5-2

Using induction on i, prove inequality (35.24).

35.5-3

Prove inequality (35.28).

35.5-4

How can you modify the approximation scheme presented in this section to find a good approximation to the smallest value not less than t that is a sum of some subset of the given input list?

35.5-5

Modify the APPROX-SUBSET-SUM procedure to also return the subset of S that sums to the value z^*.

Problems

35-1 Bin packing

You are given a set of n objects, where the size s_i of the ith object satisfies $0 < s_i < 1$. Your goal is to pack all the objects into the minimum number of unit-size bins. Each bin can hold any subset of the objects whose total size does not exceed 1.

a. Prove that the problem of determining the minimum number of bins required is NP-hard. (*Hint:* Reduce from the subset-sum problem.)

The ***first-fit*** heuristic takes each object in turn and places it into the first bin that can accommodate it, as follows. It maintains an ordered list of bins. Let b denote the number of bins in the list, where b increases over the course of the algorithm, and let $\langle B_1, \ldots, B_b \rangle$ be the list of bins. Initially $b = 0$ and the list is empty. The algorithm takes each object i in turn and places it in the lowest-numbered bin that can still accommodate it. If no bin can accommodate object i, then b is incremented and a new bin B_b is opened, containing object i. Let $S = \sum_{i=1}^{n} s_i$.

b. Argue that the optimal number of bins required is at least $\lceil S \rceil$.

c. Argue that the first-fit heuristic leaves at most one bin at most half full.

d. Prove that the number of bins used by the first-fit heuristic never exceeds $\lceil 2S \rceil$.

e. Prove an approximation ratio of 2 for the first-fit heuristic.

f. Give an efficient implementation of the first-fit heuristic, and analyze its running time.

35-2 Approximating the size of a maximum clique

Let $G = (V, E)$ be an undirected graph. For any $k \geq 1$, define $G^{(k)}$ to be the undirected graph $(V^{(k)}, E^{(k)})$, where $V^{(k)}$ is the set of all ordered k-tuples of vertices from V and $E^{(k)}$ is defined so that $(v_1, v_2, \ldots, v_k)$ is adjacent to $(w_1, w_2, \ldots, w_k)$ if and only if for $i = 1, 2, \ldots, k$, either vertex v_i is adjacent to w_i in G, or else $v_i = w_i$.

a. Prove that the size of the maximum clique in $G^{(k)}$ is equal to the kth power of the size of the maximum clique in G.

b. Argue that if there is an approximation algorithm that has a constant approximation ratio for finding a maximum-size clique, then there is a polynomial-time approximation scheme for the problem.

35-3 Weighted set-covering problem

Suppose that sets have weights in the set-covering problem, so that each set S_i in the family $\mathcal{F}$ has an associated weight w_i. The weight of a cover $\mathcal{C}$ is $\sum_{S_i \in \mathcal{C}} w_i$. The goal is wish to determine a minimum-weight cover. (Section 35.3 handles the case in which $w_i = 1$ for all i.)

Show how to generalize the greedy set-covering heuristic in a natural manner to provide an approximate solution for any instance of the weighted set-covering problem. Letting d be the maximum size of any set S_i, show that your heuristic has an approximation ratio of $H(d) = \sum_{i=1}^{d} 1/i$.

35-4 Maximum matching

Recall that for an undirected graph G, a matching is a set of edges such that no two edges in the set are incident on the same vertex. Section 25.1 showed how to find a maximum matching in a bipartite graph, that is, a matching such that no other matching in G contains more edges. This problem examines matchings in undirected graphs that are not required to be bipartite.

a. Show that a maximal matching need not be a maximum matching by exhibiting an undirected graph G and a maximal matching M in G that is not a maximum matching. (*Hint:* You can find such a graph with only four vertices.)

b. Consider a connected, undirected graph $G = (V, E)$. Give an $O(E)$-time greedy algorithm to find a maximal matching in G.

This problem concentrates on a polynomial-time approximation algorithm for maximum matching. Whereas the fastest known algorithm for maximum matching takes superlinear (but polynomial) time, the approximation algorithm here will run in linear time. You will show that the linear-time greedy algorithm for maximal matching in part (b) is a 2-approximation algorithm for maximum matching.

c. Show that the size of a maximum matching in G is a lower bound on the size of any vertex cover for G.

d. Consider a maximal matching M in $G = (V, E)$. Let $T = \{v \in V : \text{some edge in } M \text{ is incident on } v\}$. What can you say about the subgraph of G induced by the vertices of G that are not in T?

e. Conclude from part (d) that $2|M|$ is the size of a vertex cover for G.

f. Using parts (c) and (e), prove that the greedy algorithm in part (b) is a 2-approximation algorithm for maximum matching.

35-5 Parallel machine scheduling

In the ***parallel-machine-scheduling problem***, the input has two parts: n jobs, $J_1, J_2, \ldots, J_n$, where each job J_k has an associated nonnegative processing time of p_k, and m identical machines, $M_1, M_2, \ldots, M_m$. Any job can run on any machine. A ***schedule*** specifies, for each job J_k, the machine on which it runs and the time period during which it runs. Each job J_k must run on some machine M_i for p_k consecutive time units, and during that time period no other job may run on M_i. Let C_k denote the ***completion time*** of job J_k, that is, the time at which job J_k completes processing. Given a schedule, define $C_{\max} = \max\{C_j : 1 \le j \le n\}$ to be the ***makespan*** of the schedule. The goal is to find a schedule whose makespan is minimum.

For example, consider an input with two machines M_1 and M_2, and four jobs J_1, J_2, J_3, and J_4 with $p_1 = 2$, $p_2 = 12$, $p_3 = 4$, and $p_4 = 5$. Then one possible schedule runs, on machine M_1, job J_1 followed by job J_2, and on machine M_2, job J_4 followed by job J_3. For this schedule, $C_1 = 2$, $C_2 = 14$, $C_3 = 9$, $C_4 = 5$, and $C_{\max} = 14$. An optimal schedule runs job J_2 on machine M_1 and jobs J_1, J_3, and J_4 on machine M_2. For this schedule, we have $C_1 = 2$, $C_2 = 12$, $C_3 = 6$, and $C_4 = 11$, and so $C_{\max} = 12$.

Given the input to a parallel-machine-scheduling problem, let $C^*_{\max}$ denote the makespan of an optimal schedule.

a. Show that the optimal makespan is at least as large as the greatest processing time, that is,

$$C^*_{\max} \ge \max\{p_k : 1 \le k \le n\} .$$

b. Show that the optimal makespan is at least as large as the average machine load, that is,

$$C^*_{\max} \ge \frac{1}{m}\sum_{k=1}^{n} p_k .$$

Consider the following greedy algorithm for parallel machine scheduling: whenever a machine is idle, schedule any job that has not yet been scheduled.

c. Write pseudocode to implement this greedy algorithm. What is the running time of your algorithm?

d. For the schedule returned by the greedy algorithm, show that

$$C_{\max} \le \frac{1}{m}\sum_{k=1}^{n} p_k + \max\{p_k : 1 \le k \le n\} .$$

Conclude that this algorithm is a polynomial-time 2-approximation algorithm.

35-6 Approximating a maximum spanning tree

Let $G = (V, E)$ be an undirected graph with distinct edge weights $w(u, v)$ on each edge $(u, v) \in E$. For each vertex $v \in V$, denote by $\max(v)$ the maximum-weight edge incident on that vertex. Let $S_G = \{\max(v) : v \in V\}$ be the set of maximum-weight edges incident on each vertex, and let T_G be the maximum-weight spanning tree of G, that is, the spanning tree of maximum total weight. For any subset of edges $E' \subseteq E$, define $w(E') = \sum_{(u,v)\in E'} w(u, v)$.

a. Give an example of a graph with at least 4 vertices for which $S_G = T_G$.

b. Give an example of a graph with at least 4 vertices for which $S_G \neq T_G$.

c. Prove that $S_G \subseteq T_G$ for any graph G.

d. Prove that $w(S_G) \geq w(T_G)/2$ for any graph G.

e. Give an $O(V + E)$-time algorithm to compute a 2-approximation to the maximum spanning tree.

35-7 An approximation algorithm for the 0-1 knapsack problem

Recall the knapsack problem from Section 15.2. The input includes n items, where the ith item is worth v_i dollars and weighs w_i pounds. The input also includes the capacity of a knapsack, which is W pounds. Here, we add the further assumptions that each weight w_i is at most W and that the items are indexed in monotonically decreasing order of their values: $v_1 \geq v_2 \geq \cdots \geq v_n$.

In the 0-1 knapsack problem, the goal is to find a subset of the items whose total weight is at most W and whose total value is maximum. The fractional knapsack problem is like the 0-1 knapsack problem, except that a fraction of each item may be put into the knapsack, rather than either all or none of each item. If a fraction x_i of item i goes into the knapsack, where $0 \leq x_i \leq 1$, it contributes $x_i w_i$ to the weight of the knapsack and adds value $x_i v_i$. The goal of this problem is to develop a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

In order to design a polynomial-time algorithm, let's consider restricted instances of the 0-1 knapsack problem. Given an instance I of the knapsack problem, form restricted instances I_j, for $j = 1, 2, \ldots, n$, by removing items $1, 2, \ldots, j-1$ and requiring the solution to include item j (all of item j in both the fractional and 0-1 knapsack problems). No items are removed in instance I_1. For instance I_j, let P_j denote an optimal solution to the 0-1 problem and Q_j denote an optimal solution to the fractional problem.

a. Argue that an optimal solution to instance I of the 0-1 knapsack problem is one of $\{P_1, P_2, \ldots, P_n\}$.

b. Prove that to find an optimal solution Q_j to the fractional problem for instance I_j, you can include item j and then use the greedy algorithm in which each step takes as much as possible of the unchosen item with the maximum value per pound v_i/w_i in the set $\{j+1, j+2, \ldots, n\}$.

c. Prove that there is always an optimal solution Q_j to the fractional problem for instance I_j that includes at most one item fractionally. That is, for all items except possibly one, either all of the item or none of the item goes into the knapsack.

d. Given an optimal solution Q_j to the fractional problem for instance I_j, form solution R_j from Q_j by deleting any fractional items from Q_j. Let $v(S)$ denote the total value of items taken in a solution S. Prove that $v(R_j) \geq v(Q_j)/2 \geq v(P_j)/2$.

e. Give a polynomial-time algorithm that returns a maximum-value solution from the set $\{R_1, R_2, \ldots, R_n\}$, and prove that your algorithm is a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

Chapter notes

Although methods that do not necessarily compute exact solutions have been known for thousands of years (for example, methods to approximate the value of π), the notion of an approximation algorithm is much more recent. Hochbaum [221] credits Garey, Graham, and Ullman [175] and Johnson [236] with formalizing the concept of a polynomial-time approximation algorithm. The first such algorithm is often credited to Graham [197].

Since this early work, thousands of approximation algorithms have been designed for a wide range of problems, and there is a wealth of literature on this field. Texts by Ausiello et al. [29], Hochbaum [221], Vazirani [446], and Williamson and Shmoys [459] deal exclusively with approximation algorithms, as do surveys by Shmoys [409] and Klein and Young [256]. Several other texts, such as Garey and Johnson [176] and Papadimitriou and Steiglitz [353], have significant coverage of approximation algorithms as well. Books edited by Lawler, Lenstra, Rinnooy Kan, and Shmoys [277] and by Gutin and Punnen [204] provide extensive treatments of approximation algorithms and heuristics for the traveling-salesperson problem.

Papadimitriou and Steiglitz attribute the algorithm APPROX-VERTEX-COVER to F. Gavril and M. Yannakakis. The vertex-cover problem has been studied extensively (Hochbaum [221] lists 16 different approximation algorithms for this problem), but all the approximation ratios are at least $2 - o(1)$.

The algorithm APPROX-TSP-TOUR appears in a paper by Rosenkrantz, Stearns, and Lewis [384]. Christofides improved on this algorithm and gave a 3/2-approximation algorithm for the traveling-salesperson problem with the triangle inequality. Arora [23] and Mitchell [330] have shown that if the points lie in the euclidean plane, there is a polynomial-time approximation scheme. Theorem 35.3 is due to Sahni and Gonzalez [392].

The algorithm APPROX-SUBSET-SUM and its analysis are loosely modeled after related approximation algorithms for the knapsack and subset-sum problems by Ibarra and Kim [234].

Problem 35-7 is a combinatorial version of a more general result on approximating knapsack-type integer programs by Bienstock and McClosky [55].

The randomized algorithm for MAX-3-CNF satisfiability is implicit in the work of Johnson [236]. The weighted vertex-cover algorithm is by Hochbaum [220]. Section 35.4 only touches on the power of randomization and linear programming in the design of approximation algorithms. A combination of these two ideas yields a technique called "randomized rounding," which formulates a problem as an integer linear program, solves the linear-programming relaxation, and interprets the variables in the solution as probabilities. These probabilities then help guide the solution of the original problem. This technique was first used by Raghavan and Thompson [374], and it has had many subsequent uses. (See Motwani, Naor, and Raghavan [335] for a survey.) Several other notable ideas in the field of approximation algorithms include the primal-dual method (see Goemans and Williamson [184] for a survey), finding sparse cuts for use in divide-and-conquer algorithms [288], and the use of semidefinite programming [183].

As mentioned in the chapter notes for Chapter 34, results in probabilistically checkable proofs have led to lower bounds on the approximability of many problems, including several in this chapter. In addition to the references there, the chapter by Arora and Lund [26] contains a good description of the relationship between probabilistically checkable proofs and the hardness of approximating various problems.

Part VIII Appendix: Mathematical Background

Introduction

When you analyze algorithms, you often need to draw upon a body of mathematical tools. Some of these tools are as simple as high-school algebra, but others may be new to you. In Part I, we saw how to manipulate asymptotic notations and solve recurrences. This appendix comprises a compendium of several other concepts and methods used in analyzing algorithms. As noted in the introduction to Part I, you may have seen much of the material in this appendix before having read this book, although some of the specific notational conventions appearing here might differ from those you have seen elsewhere. Hence, you should treat this appendix as reference material. As in the rest of this book, however, we have included exercises and problems, in order for you to improve your skills in these areas.

Appendix A offers methods for evaluating and bounding summations, which occur frequently in the analysis of algorithms. Many of the formulas here appear in any calculus text, but you will find it convenient to have these methods compiled in one place.

Appendix B contains basic definitions and notations for sets, relations, functions, graphs, and trees. It also gives some basic properties of these mathematical objects.

Appendix C begins with elementary principles of counting: permutations, combinations, and the like. The remainder contains definitions and properties of basic probability. Most of the algorithms in this book require no probability for their analysis, and thus you can easily omit the latter sections of the chapter on a first reading, even without skimming them. Later, when you encounter a probabilistic analysis that you want to understand better, you will find Appendix C well organized for reference purposes.

Appendix D defines matrices, their operations, and some of their basic properties. You have probably seen most of this material already if you have taken a course in linear algebra. But you might find it helpful to have one place to look for notations and definitions.

A Summations

When an algorithm contains an iterative control construct such as a **while** or **for** loop, you can express its running time as the sum of the times spent on each execution of the body of the loop. For example, Section 2.2 argued that the ith iteration of insertion sort took time proportional to i in the worst case. Adding up the time spent on each iteration produced the summation (or series) $\sum_{i=2}^{n} i$. Evaluating this summation resulted in a bound of $\Theta(n^2)$ on the worst-case running time of the algorithm. This example illustrates why you should know how to manipulate and bound summations.

Section A.1 lists several basic formulas involving summations. Section A.2 offers useful techniques for bounding summations. The formulas in Section A.1 appear without proof, though proofs for some of them appear in Section A.2 to illustrate the methods of that section. You can find most of the other proofs in any calculus text.

A.1 Summation formulas and properties

Given a sequence $a_1, a_2, \ldots, a_n$ of numbers, where n is a nonnegative integer, the finite sum $a_1 + a_2 + \cdots + a_n$ can be expressed as $\sum_{k=1}^{n} a_k$. If $n = 0$, the value of the summation is defined to be 0. The value of a finite series is always well defined, and the order in which its terms are added does not matter.

Given an infinite sequence $a_1, a_2, \ldots$ of numbers, we can write their infinite sum $a_1 + a_2 + \cdots$ as $\sum_{k=1}^{\infty} a_k$, which means $\lim_{n\to\infty} \sum_{k=1}^{n} a_k$. If the limit does not exist, the series ***diverges***, and otherwise, it ***converges***. The terms of a convergent series cannot always be added in any order. You can, however, rearrange the terms of an ***absolutely convergent series***, that is, a series $\sum_{k=1}^{\infty} a_k$ for which the series $\sum_{k=1}^{\infty} |a_k|$ also converges.

Linearity

For any real number c and any finite sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$,

$$\sum_{k=1}^{n}(ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k \ .$$

The linearity property also applies to infinite convergent series.

The linearity property applies to summations incorporating asymptotic notation. For example,

$$\sum_{k=1}^{n} \Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right) \ .$$

In this equation, the Θ-notation on the left-hand side applies to the variable k, but on the right-hand side, it applies to n. Such manipulations also apply to infinite convergent series.

Arithmetic series

The summation

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n \ ,$$

is an ***arithmetic series*** and has the value

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \tag{A.1}$$

$$= \Theta(n^2) \ . \tag{A.2}$$

A ***general arithmetic series*** includes an additive constant $a \geq 0$ and a constant coefficient $b > 0$ in each term, but has the same total asymptotically:

$$\sum_{k=1}^{n}(a + bk) = \Theta(n^2) \ . \tag{A.3}$$

Sums of squares and cubes

The following formulas apply to summations of squares and cubes:

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \ , \tag{A.4}$$

$$\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4} \ . \tag{A.5}$$

Geometric series

For real $x \neq 1$, the summation

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n$$

is a ***geometric series*** and has the value

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} . \tag{A.6}$$

The infinite decreasing geometric series occurs when the summation is infinite and $|x| < 1$:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} . \tag{A.7}$$

If we assume that $0^0 = 1$, these formulas apply even when $x = 0$.

Harmonic series

For positive integers n, the nth ***harmonic number*** is

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \\ &= \sum_{k=1}^{n} \frac{1}{k} \qquad (A.8) \\ &= \ln n + O(1) . \qquad (A.9) \end{aligned}$$

Inequalities (A.20) and (A.21) on page 1150 provide the stronger bounds

$$\ln(n + 1) \leq H_n \leq \ln n + 1 . \tag{A.10}$$

Integrating and differentiating series

Integrating or differentiating the formulas above yields additional formulas. For example, differentiating both sides of the infinite geometric series (A.7) and multiplying by x gives

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1 - x)^2} \tag{A.11}$$

for $|x| < 1$.

Telescoping series

For any sequence $a_0, a_1, \ldots, a_n$,

$$\sum_{k=1}^{n}(a_k - a_{k-1}) = a_n - a_0 \ , \tag{A.12}$$

since each of the terms $a_1, a_2, \ldots, a_{n-1}$ is added in exactly once and subtracted out exactly once. We say that the sum ***telescopes***. Similarly,

$$\sum_{k=0}^{n-1}(a_k - a_{k+1}) = a_0 - a_n \ .$$

As an example of a telescoping sum, consider the series

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} \ .$$

Rewriting each term as

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1} \ ,$$

gives

$$\begin{aligned}\sum_{k=1}^{n-1} \frac{1}{k(k+1)} &= \sum_{k=1}^{n-1}\left(\frac{1}{k} - \frac{1}{k+1}\right) \\ &= 1 - \frac{1}{n} \ .\end{aligned}$$

Reindexing summations

A series can sometimes be simplified by changing its index, often reversing the order of summation. Consider the series $\sum_{k=0}^{n} a_{n-k}$. Because the terms in this summation are $a_n, a_{n-1}, \ldots, a_0$, we can reverse the order of indices by letting $j = n - k$ and rewrite this summation as

$$\sum_{k=0}^{n} a_{n-k} = \sum_{j=0}^{n} a_j \ . \tag{A.13}$$

Generally, if the summation index appears in the body of the sum with a minus sign, it's worth thinking about reindexing.

As an example, consider the summation

$$\sum_{k=1}^{n} \frac{1}{n-k+1} .$$

The index k appears with a negative sign in $1/(n-k+1)$. And indeed, we can simplify this summation, this time setting $j = n-k+1$, yielding

$$\sum_{k=1}^{n} \frac{1}{n-k+1} = \sum_{j=1}^{n} \frac{1}{j} , \tag{A.14}$$

which is just the harmonic series (A.8).

Products

The finite product $a_1 a_2 \cdots a_n$ can be expressed as

$$\prod_{k=1}^{n} a_k .$$

If $n = 0$, the value of the product is defined to be 1. You can convert a formula with a product to a formula with a summation by using the identity

$$\lg\left(\prod_{k=1}^{n} a_k\right) = \sum_{k=1}^{n} \lg a_k .$$

Exercises

A.1-1
Prove that $\sum_{k=1}^{n} O(f_k(i)) = O\left(\sum_{k=1}^{n} f_k(i)\right)$ by using the linearity property of summations.

A.1-2
Find a simple formula for $\sum_{k=1}^{n}(2k-1)$.

A.1-3
Interpret the decimal number 111,111,111 in light of equation (A.6).

A.1-4
Evaluate the infinite series $1 - \frac{1}{2} + \frac{1}{4} - \frac{1}{8} + \frac{1}{16} - \cdots$.

A.1-5
Let $c \geq 0$ be a constant. Show that $\sum_{k=1}^{n} k^c = \Theta(n^{c+1})$.

A.1-6
Show that $\sum_{k=0}^{\infty} k^2 x^k = x(1+x)/(1-x)^3$ for $|x| < 1$.

A.1-7
Prove that $\sum_{k=1}^{n} \sqrt{k \lg k} = \Theta(n^{3/2} \lg^{1/2} n)$. (*Hint:* Show the asymptotic upper and lower bounds separately.)

★ ***A.1-8***
Show that $\sum_{k=1}^{n} 1/(2k-1) = \ln(\sqrt{n}) + O(1)$ by manipulating the harmonic series.

★ ***A.1-9***
Show that $\sum_{k=0}^{\infty} (k-1)/2^k = 0$.

★ ***A.1-10***
Evaluate the sum $\sum_{k=1}^{\infty} (2k+1)x^{2k}$ for $|x| < 1$.

★ ***A.1-11***
Evaluate the product $\prod_{k=2}^{n} (1 - 1/k^2)$.

A.2 Bounding summations

You can choose from several techniques to bound the summations that describe the running times of algorithms. Here are some of the most frequently used methods.

Mathematical induction

The most basic way to evaluate a series is to use mathematical induction. As an example, let's prove that the arithmetic series $\sum_{k=1}^{n} k$ evaluates to $n(n+1)/2$. For $n = 1$, we have that $n(n+1)/2 = 1 \cdot 2/2 = 1$, which equals $\sum_{k=1}^{1} k$. With the inductive assumption that it holds for n, we prove that it holds for $n+1$. We have

$$\begin{aligned}
\sum_{k=1}^{n+1} k &= \sum_{k=1}^{n} k + (n+1) \\
&= \frac{n(n+1)}{2} + (n+1) \\
&= \frac{n^2 + n + 2n + 2}{2} \\
&= \frac{(n+1)(n+2)}{2} .
\end{aligned}$$

You don't always need to guess the exact value of a summation in order to use mathematical induction. Instead, you can use induction to prove an upper or lower

bound on a summation. As an example, let's prove the asymptotic upper bound $\sum_{k=0}^{n} 3^k = O(3^n)$. More specifically, we'll prove that $\sum_{k=0}^{n} 3^k \le c3^n$ for some constant c. For the initial condition $n = 0$, we have $\sum_{k=0}^{0} 3^k = 1 \le c \cdot 1$ as long as $c \ge 1$. Assuming that the bound holds for n, we prove that it holds for $n + 1$. We have

$$\begin{aligned}
\sum_{k=0}^{n+1} 3^k &= \sum_{k=0}^{n} 3^k + 3^{n+1} \\
&\le c3^n + 3^{n+1} \qquad \text{(by the inductive hypothesis)} \\
&= \left(\frac{1}{3} + \frac{1}{c}\right) c3^{n+1} \\
&\le c3^{n+1}
\end{aligned}$$

as long as $(1/3 + 1/c) \le 1$ or, equivalently, $c \ge 3/2$. Thus, $\sum_{k=0}^{n} 3^k = O(3^n)$, as we wished to show.

You need to take care when using asymptotic notation to prove bounds by induction. Consider the following fallacious proof that $\sum_{k=1}^{n} k = O(n)$. Certainly, $\sum_{k=1}^{1} k = O(1)$. Assuming that the bound holds for n, we now prove it for $n + 1$:

$$\begin{aligned}
\sum_{k=1}^{n+1} k &= \sum_{k=1}^{n} k + (n + 1) \\
&= O(n) + (n + 1) \qquad \Longleftarrow \textit{wrong!} \\
&= O(n + 1) .
\end{aligned}$$

The bug in the argument is that the "constant" hidden by the "big-oh" grows with n and thus is not constant. We have not shown that the same constant works for *all* n.

Bounding the terms

You can sometimes obtain a good upper bound on a series by bounding each term of the series, and it often suffices to use the largest term to bound the others. For example, a quick upper bound on the arithmetic series (A.1) is

$$\begin{aligned}
\sum_{k=1}^{n} k &\le \sum_{k=1}^{n} n \\
&= n^2 .
\end{aligned}$$

In general, for a series $\sum_{k=1}^{n} a_k$, if we let $a_{\max} = \max\{a_k : 1 \le k \le n\}$, then

$$\sum_{k=1}^{n} a_k \le n \cdot a_{\max} .$$

The technique of bounding each term in a series by the largest term is a weak method when the series can in fact be bounded by a geometric series. Given the series $\sum_{k=0}^{n} a_k$, suppose that $a_{k+1}/a_k \le r$ for all $k \ge 0$, where $0 < r < 1$ is a constant. You can bound the sum by an infinite decreasing geometric series, since $a_k \le a_0 r^k$, and thus

$$\begin{aligned}\sum_{k=0}^{n} a_k &\le \sum_{k=0}^{\infty} a_0 r^k \\ &= a_0 \sum_{k=0}^{\infty} r^k \qquad \text{(A.15)} \\ &= a_0 \frac{1}{1-r} \,. \qquad \text{(A.16)}\end{aligned}$$

You can apply this method to bound the summation $\sum_{k=1}^{\infty}(k/3^k)$. In order to start the summation at $k = 0$, rewrite it as $\sum_{k=0}^{\infty}((k+1)/3^{k+1})$. The first term ($a_0$) is $1/3$, and the ratio (r) of consecutive terms is

$$\begin{aligned}\frac{(k+2)/3^{k+2}}{(k+1)/3^{k+1}} &= \frac{1}{3} \cdot \frac{k+2}{k+1} \\ &\le \frac{2}{3}\end{aligned}$$

for all $k \ge 0$. Thus, we have

$$\begin{aligned}\sum_{k=1}^{\infty} \frac{k}{3^k} &= \sum_{k=0}^{\infty} \frac{k+1}{3^{k+1}} \\ &\le \frac{1}{3} \cdot \frac{1}{1-2/3} \\ &= 1 \,.\end{aligned}$$

A common bug in applying this method is to show that the ratio of consecutive terms is less than 1 and then to assume that the summation is bounded by a geometric series. An example is the infinite harmonic series, which diverges since

$$\begin{aligned}\sum_{k=1}^{\infty} \frac{1}{k} &= \lim_{n\to\infty} \sum_{k=1}^{n} \frac{1}{k} \\ &= \lim_{n\to\infty} \Theta(\lg n) \\ &= \infty \,.\end{aligned}$$

The ratio of the $(k+1)$st and kth terms in this series is $k/(k+1) < 1$, but the series is not bounded by a decreasing geometric series. To bound a series by a geometric

series, you need to show that there is an $r < 1$, which is a *constant*, such that the ratio of all pairs of consecutive terms never exceeds r. In the harmonic series, no such r exists because the ratio becomes arbitrarily close to 1.

Splitting summations

One way to obtain bounds on a difficult summation is to express the series as the sum of two or more series by partitioning the range of the index and then to bound each of the resulting series. For example, let's find a lower bound on the arithmetic series $\sum_{k=1}^{n} k$, which we have already seen has an upper bound of n^2. You might attempt to bound each term in the summation by the smallest term, but since that term is 1, you would get a lower bound of n for the summation—far off from the upper bound of n^2.

You can obtain a better lower bound by first splitting the summation. Assume for convenience that n is even, so that

$$\begin{aligned}
\sum_{k=1}^{n} k &= \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k \\
&\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} \frac{n}{2} \\
&= \left(\frac{n}{2}\right)^2 \\
&= \Omega(n^2) ,
\end{aligned}$$

which is an asymptotically tight bound, since $\sum_{k=1}^{n} k = O(n^2)$.

For a summation arising from the analysis of an algorithm, you can sometimes split the summation and ignore a constant number of the initial terms. Generally, this technique applies when each term a_k in a summation $\sum_{k=0}^{n} a_k$ is independent of n. Then for any constant $k_0 > 0$, you can write

$$\begin{aligned}
\sum_{k=0}^{n} a_k &= \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^{n} a_k \\
&= \Theta(1) + \sum_{k=k_0}^{n} a_k ,
\end{aligned}$$

since the initial terms of the summation are all constant and there are a constant number of them. You can then use other methods to bound $\sum_{k=k_0}^{n} a_k$. This technique applies to infinite summations as well. For example, let's find an asymptotic upper bound on $\sum_{k=0}^{\infty} k^2/2^k$. The ratio of consecutive terms is

$$\begin{aligned}\frac{(k+1)^2/2^{k+1}}{k^2/2^k} &= \frac{(k+1)^2}{2k^2} \\ &\le 8/9\end{aligned}$$

if $k \ge 3$. Thus, you can split the summation into

$$\begin{aligned}\sum_{k=0}^{\infty} \frac{k^2}{2^k} &= \sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \\ &= \sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=0}^{\infty} \frac{(k+3)^2}{2^{k+3}} && \text{(by reindexing)} \\ &\le \sum_{k=0}^{2} \frac{k^2}{2^k} + \frac{9}{8} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k && \text{(by inequality (A.15))} \\ &= (0 + 1/2 + 1) + \frac{9/8}{1 - 8/9} && \text{(by equation (A.16))} \\ &= O(1) .\end{aligned}$$

The technique of splitting summations can help determine asymptotic bounds in much more difficult situations. For example, here is one way to obtain a bound of $O(\lg n)$ on the harmonic series (A.9):

$$H_n = \sum_{k=1}^{n} \frac{1}{k} .$$

The idea is to split the range 1 to n into $\lfloor \lg n \rfloor + 1$ pieces and upper-bound the contribution of each piece by 1. For $i = 0, 1, \ldots, \lfloor \lg n \rfloor$, the ith piece consists of the terms starting at $1/2^i$ and going up to but not including $1/2^{i+1}$. The last piece might contain terms not in the original harmonic series, giving

$$\begin{aligned}\sum_{k=1}^{n} \frac{1}{k} &\le \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i - 1} \frac{1}{2^i + j} \\ &\le \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i - 1} \frac{1}{2^i} \\ &= \sum_{i=0}^{\lfloor \lg n \rfloor} \left(2^i \cdot \frac{1}{2^i}\right) \\ &= \sum_{i=0}^{\lfloor \lg n \rfloor} 1 \\ &\le \lg n + 1 . \qquad \text{(A.17)}\end{aligned}$$

Approximation by integrals

When a summation has the form $\sum_{k=m}^{n} f(k)$, where $f(k)$ is a monotonically increasing function, you can approximate it by integrals:

$$\int_{m-1}^{n} f(x)\,dx \le \sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)\,dx \,. \tag{A.18}$$

Figure A.1 justifies this approximation. The summation is represented as the area of the rectangles in the figure, and the integral is the blue region under the curve. When $f(k)$ is a monotonically decreasing function, you can use a similar method to provide the bounds

$$\int_{m}^{n+1} f(x)\,dx \le \sum_{k=m}^{n} f(k) \le \int_{m-1}^{n} f(x)\,dx \,. \tag{A.19}$$

The integral approximation (A.19) can be used to prove the tight bounds in inequality (A.10) for the nth harmonic number. The lower bound is

$$\begin{aligned} \sum_{k=1}^{n} \frac{1}{k} &\ge \int_{1}^{n+1} \frac{dx}{x} \\ &= \ln(n+1) \,, \end{aligned} \tag{A.20}$$

For the upper bound, the integral approximation gives

$$\begin{aligned} \sum_{k=1}^{n} \frac{1}{k} &= \sum_{k=2}^{n} \frac{1}{k} + 1 \\ &\le \int_{1}^{n} \frac{dx}{x} + 1 \\ &= \ln n + 1 \,. \end{aligned} \tag{A.21}$$

Exercises

A.2-1
Show that $\sum_{k=1}^{n} 1/k^2$ is bounded above by a constant.

A.2-2
Find an asymptotic upper bound on the summation

$$\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil \,.$$

Figure A.1 Approximation of $\sum_{k=m}^{n} f(k)$ by integrals. The area of each rectangle is shown within the rectangle, and the total rectangle area represents the value of the summation. The integral is represented by the blue area under the curve. Comparing areas in **(a)** gives the lower bound $\int_{m-1}^{n} f(x)\,dx \le \sum_{k=m}^{n} f(k)$. Shifting the rectangles one unit to the right gives the upper bound $\sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)\,dx$ in **(b)**.

A.2-3
Show that the nth harmonic number is $\Omega(\lg n)$ by splitting the summation.

A.2-4
Approximate $\sum_{k=1}^{n} k^3$ with an integral.

A.2-5
Why can't you use the integral approximation (A.19) directly on $\sum_{k=1}^{n} 1/k$ to obtain an upper bound on the nth harmonic number?

Problems

A-1 Bounding summations
Give asymptotically tight bounds on the following summations. Assume that $r \geq 0$ and $s \geq 0$ are constants.

a. $\displaystyle\sum_{k=1}^{n} k^r$.

b. $\displaystyle\sum_{k=1}^{n} \lg^s k$.

c. $\displaystyle\sum_{k=1}^{n} k^r \lg^s k$.

Appendix notes

Knuth [259] provides an excellent reference for the material presented here. You can find basic properties of series in any good calculus book, such as Apostol [19] or Thomas et al. [433].

B Sets, Etc.

Many chapters of this book touch on the elements of discrete mathematics. This appendix reviews the notations, definitions, and elementary properties of sets, relations, functions, graphs, and trees. If you are already well versed in this material, you can probably just skim this chapter.

B.1 Sets

A ***set*** is a collection of distinguishable objects, called its ***members*** or ***elements***. If an object x is a member of a set S, we write $x \in S$ (read "x is a member of S" or, more briefly, "x belongs to S"). If x is not a member of S, we write $x \notin S$. To describe a set explicitly, write its members as a list inside braces. For example, to define a set S to contain precisely the numbers 1, 2, and 3, write $S = \{1, 2, 3\}$. Since 2 belongs to the set S, we can write $2 \in S$, and since 4 is not a member, we can write $4 \notin S$. A set cannot contain the same object more than once,[1] and its elements are not ordered. Two sets A and B are ***equal***, written $A = B$, if they contain the same elements. For example, $\{1, 2, 3, 1\} = \{1, 2, 3\} = \{3, 2, 1\}$.

We adopt special notations for frequently encountered sets:

- $\emptyset$ denotes the ***empty set***, that is, the set containing no members.
- $\mathbb{Z}$ denotes the set of ***integers***, that is, the set $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.
- $\mathbb{R}$ denotes the set of ***real numbers***.
- $\mathbb{N}$ denotes the set of ***natural numbers***, that is, the set $\{0, 1, 2, \ldots\}$.[2]

[1] A variation of a set, which can contain the same object more than once, is called a ***multiset***.

[2] Some authors start the natural numbers with 1 instead of 0. The modern trend seems to be to start with 0.

If all the elements of a set A are contained in a set B, that is, if $x \in A$ implies $x \in B$, then we write $A \subseteq B$ and say that A is a ***subset*** of B. A set A is a ***proper subset*** of set B, written $A \subset B$, if $A \subseteq B$ but $A \neq B$. (Some authors use the symbol "$\subset$" to denote the ordinary subset relation, rather than the proper-subset relation.) Every set is a subset of itself: $A \subseteq A$ for any set A. For two sets A and B, we have $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$. The subset relation is transitive (see page 1159): for any three sets A, B, and C, if $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$. The proper-subset relation is transitive as well. The empty set is a subset of all sets: for any set A, we have $\emptyset \subseteq A$.

Sets can be specified in terms of other sets. Given a set A, a set $B \subseteq A$ can be defined by stating a property that distinguishes the elements of B. For example, one way to define the set of even integers is $\{x : x \in \mathbb{Z} \text{ and } x/2 \text{ is an integer}\}$. The colon in this notation is read "such that." (Some authors use a vertical bar in place of the colon.)

Given two sets A and B, ***set operations*** define new sets:

- The ***intersection*** of sets A and B is the set

  $$A \cap B = \{x : x \in A \text{ and } x \in B\} .$$

- The ***union*** of sets A and B is the set

  $$A \cup B = \{x : x \in A \text{ or } x \in B\} .$$

- The ***difference*** between two sets A and B is the set

  $$A - B = \{x : x \in A \text{ and } x \notin B\} .$$

Set operations obey the following laws:

Empty set laws:

$$A \cap \emptyset = \emptyset ,$$
$$A \cup \emptyset = A .$$

Idempotency laws:

$$A \cap A = A ,$$
$$A \cup A = A .$$

Commutative laws:

$$A \cap B = B \cap A ,$$
$$A \cup B = B \cup A .$$

$$A \quad - \quad (B \cap C) \quad = \quad A - (B \cap C) \quad = \quad (A - B) \quad \cup \quad (A - C)$$

Figure B.1 A Venn diagram illustrating the first of DeMorgan's laws (B.2). Each of the sets A, B, and C is represented as a circle.

Associative laws:

$$\begin{aligned} A \cap (B \cap C) &= (A \cap B) \cap C \ , \\ A \cup (B \cup C) &= (A \cup B) \cup C \ . \end{aligned}$$

Distributive laws:

$$\begin{aligned} A \cap (B \cup C) &= (A \cap B) \cup (A \cap C) \ , \\ A \cup (B \cap C) &= (A \cup B) \cap (A \cup C) \ . \end{aligned} \tag{B.1}$$

Absorption laws:

$$\begin{aligned} A \cap (A \cup B) &= A \ , \\ A \cup (A \cap B) &= A \ . \end{aligned}$$

DeMorgan's laws:

$$\begin{aligned} A - (B \cap C) &= (A - B) \cup (A - C) \ , \\ A - (B \cup C) &= (A - B) \cap (A - C) \ . \end{aligned} \tag{B.2}$$

Figure B.1 illustrates the first of DeMorgan's laws, using a ***Venn diagram***: a graphical picture in which sets are represented as regions of the plane.

Often, all the sets under consideration are subsets of some larger set U called the ***universe***. For example, when considering various sets made up only of integers, the set $\mathbb{Z}$ of integers is an appropriate universe. Given a universe U, we define the ***complement*** of a set A as $\overline{A} = U - A = \{x : x \in U \text{ and } x \notin A\}$. For any set $A \subseteq U$, we have the following laws:

$$\begin{aligned} \overline{\overline{A}} &= A \ , \\ A \cap \overline{A} &= \emptyset \ , \\ A \cup \overline{A} &= U \ . \end{aligned}$$

An equivalent way to express DeMorgan's laws (B.2) uses set complements. For any two sets $B, C \subseteq U$, we have

$$\overline{B \cap C} = \overline{B} \cup \overline{C} ,$$
$$\overline{B \cup C} = \overline{B} \cap \overline{C} .$$

Two sets A and B are ***disjoint*** if they have no elements in common, that is, if $A \cap B = \emptyset$. A ***collection*** of sets $S_1, S_2, \ldots$, either finite or infinite, is a set of sets, in which each member is a set S_i. A collection $\mathcal{S} = \{S_i\}$ of nonempty sets forms a ***partition*** of a set S if

- the sets are ***pairwise disjoint***, that is, $S_i, S_j \in \mathcal{S}$ and $i \neq j$ imply $S_i \cap S_j = \emptyset$, and
- their union is S, that is,

$$S = \bigcup_{S_i \in \mathcal{S}} S_i .$$

In other words, $\mathcal{S}$ forms a partition of S if each element of S appears in exactly one set $S_i \in \mathcal{S}$.

The number of elements in a set is the ***cardinality*** (or ***size***) of the set, denoted $|S|$. Two sets have the same cardinality if their elements can be put into a one-to-one correspondence. The cardinality of the empty set is $|\emptyset| = 0$. If the cardinality of a set is a natural number, the set is ***finite***, and otherwise, it is ***infinite***. An infinite set that can be put into a one-to-one correspondence with the natural numbers $\mathbb{N}$ is ***countably infinite***, and otherwise, it is ***uncountable***. For example, the integers $\mathbb{Z}$ are countable, but the reals $\mathbb{R}$ are uncountable.

For any two finite sets A and B, we have the identity

$$|A \cup B| = |A| + |B| - |A \cap B| , \tag{B.3}$$

from which we can conclude that

$$|A \cup B| \leq |A| + |B| .$$

If A and B are disjoint, then $|A \cap B| = 0$ and thus $|A \cup B| = |A| + |B|$. If $A \subseteq B$, then $|A| \leq |B|$.

A finite set of n elements is sometimes called an ***n-set***. A 1-set is called a ***singleton***. A subset of k elements of a set is sometimes called a ***k-subset***.

We denote the set of all subsets of a set S, including the empty set and S itself, by 2^S, called the ***power set*** of S. For example, $2^{\{a,b\}} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. The power set of a finite set S has cardinality $2^{|S|}$ (see Exercise B.1-5).

We sometimes care about setlike structures in which the elements are ordered. An ***ordered pair*** of two elements a and b is denoted (a, b) and is defined formally

as the set $(a, b) = \{a, \{a, b\}\}$. Thus, the ordered pair (a, b) is *not* the same as the ordered pair (b, a).

The ***Cartesian product*** of two sets A and B, denoted $A \times B$, is the set of all ordered pairs such that the first element of the pair is an element of A and the second is an element of B. More formally,

$$A \times B = \{(a, b) : a \in A \text{ and } b \in B\} .$$

For example, $\{a, b\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)\}$. When A and B are finite sets, the cardinality of their Cartesian product is

$$|A \times B| = |A| \cdot |B| . \tag{B.4}$$

The Cartesian product of n sets $A_1, A_2, \ldots, A_n$ is the set of ***n-tuples***

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) : a_i \in A_i \text{ for } i = 1, 2, \ldots, n\} ,$$

whose cardinality is

$$|A_1 \times A_2 \times \cdots \times A_n| = |A_1| \cdot |A_2| \cdots |A_n|$$

if all sets A_i are finite. We denote an n-fold Cartesian product over a single set A by the set

$$A^n = \underbrace{A \times A \times \cdots \times A}_{n \text{ times}} ,$$

whose cardinality is $|A^n| = |A|^n$ if A is finite. We can also view an n-tuple as a finite sequence of length n (see page 1162).

Intervals are continuous sets of real numbers. We denote them with parentheses and/or brackets. Given real numbers a and b, the ***closed interval*** $[a, b]$ is the set $\{x \in \mathbb{R} : a \le x \le b\}$ of reals between a and b, including both a and b. (If $a > b$, this definition implies that $[a, b] = \emptyset$.) The ***open interval*** $(a, b) = \{x \in \mathbb{R} : a < x < b\}$ omits both of the endpoints from the set. There are two ***half-open intervals*** $[a, b) = \{x \in \mathbb{R} : a \le x < b\}$ and $(a, b] = \{x \in \mathbb{R} : a < x \le b\}$, each of which excludes one endpoint.

Intervals can also be defined on the integers by replacing $\mathbb{R}$ in the these definitions by $\mathbb{Z}$. Whether the interval is defined over the reals or integers can usually be inferred from context.

Exercises

B.1-1

Draw Venn diagrams that illustrate the first of the distributive laws (B.1).

B.1-2
Prove the generalization of DeMorgan's laws to any finite collection of sets:

$$\overline{A_1 \cap A_2 \cap \cdots \cap A_n} = \overline{A_1} \cup \overline{A_2} \cup \cdots \cup \overline{A_n} ,$$
$$\overline{A_1 \cup A_2 \cup \cdots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n} .$$

★ ***B.1-3***
Prove the generalization of equation (B.3), which is called the ***principle of inclusion and exclusion***:

$$\begin{aligned}
&|A_1 \cup A_2 \cup \cdots \cup A_n| = \\
&\quad |A_1| + |A_2| + \cdots + |A_n| \\
&\quad - |A_1 \cap A_2| - |A_1 \cap A_3| - \cdots && \text{(all pairs)} \\
&\quad + |A_1 \cap A_2 \cap A_3| + \cdots && \text{(all triples)} \\
&\qquad\qquad \vdots \\
&\quad + (-1)^{n-1} |A_1 \cap A_2 \cap \cdots \cap A_n| .
\end{aligned}$$

B.1-4
Show that the set of odd natural numbers is countable.

B.1-5
Show that for any finite set S, the power set 2^S has $2^{|S|}$ elements (that is, there are $2^{|S|}$ distinct subsets of S).

B.1-6
Give an inductive definition for an n-tuple by extending the set-theoretic definition for an ordered pair.

B.2 Relations

A ***binary relation*** R on two sets A and B is a subset of the Cartesian product $A \times B$. If $(a, b) \in R$, we sometimes write $a \; R \; b$. When we say that R is a binary relation on a set A, we mean that R is a subset of $A \times A$. For example, the "less than" relation on the natural numbers is the set $\{(a, b) : a, b \in \mathbb{N} \text{ and } a < b\}$. An n-ary relation on sets $A_1, A_2, \ldots, A_n$ is a subset of $A_1 \times A_2 \times \cdots \times A_n$.

A binary relation $R \subseteq A \times A$ is ***reflexive*** if

$$a \; R \; a$$

for all $a \in A$. For example, "$=$" and "$\leq$" are reflexive relations on $\mathbb{N}$, but "$<$" is not. The relation R is ***symmetric*** if

$a \; R \; b$ implies $b \; R \; a$

for all $a, b \in A$. For example, "$=$" is symmetric, but "$<$" and "$\leq$" are not. The relation R is ***transitive*** if

$a \; R \; b$ and $b \; R \; c$ imply $a \; R \; c$

for all $a, b, c \in A$. For example, the relations "$<$," "$\leq$," and "$=$" are transitive, but the relation $R = \{(a, b) : a, b \in \mathbb{N} \text{ and } a = b - 1\}$ is not, since $3 \; R \; 4$ and $4 \; R \; 5$ do not imply $3 \; R \; 5$.

A relation that is reflexive, symmetric, and transitive is an ***equivalence relation***. For example, "$=$" is an equivalence relation on the natural numbers, but "$<$" is not. If R is an equivalence relation on a set A, then for $a \in A$, the ***equivalence class*** of a is the set $[a] = \{b \in A : a \; R \; b\}$, that is, the set of all elements equivalent to a. For example, if we define $R = \{(a, b) : a, b \in \mathbb{N} \text{ and } a + b \text{ is an even number}\}$, then R is an equivalence relation, since $a + a$ is even (reflexive), $a + b$ is even implies $b + a$ is even (symmetric), and $a + b$ is even and $b + c$ is even imply $a + c$ is even (transitive). The equivalence class of 4 is $[4] = \{0, 2, 4, 6, \ldots\}$, and the equivalence class of 3 is $[3] = \{1, 3, 5, 7, \ldots\}$. A basic theorem of equivalence classes is the following.

Theorem B.1 (An equivalence relation is the same as a partition)
The equivalence classes of any equivalence relation R on a set A form a partition of A, and any partition of A determines an equivalence relation on A for which the sets in the partition are the equivalence classes.

Proof For the first part of the proof, we must show that the equivalence classes of R are nonempty, pairwise-disjoint sets whose union is A. Because R is reflexive, $a \in [a]$, and so the equivalence classes are nonempty. Moreover, since every element $a \in A$ belongs to the equivalence class $[a]$, the union of the equivalence classes is A. It remains to show that the equivalence classes are pairwise disjoint, that is, if two equivalence classes $[a]$ and $[b]$ have an element c in common, then they are in fact the same set. Suppose that $a \; R \; c$ and $b \; R \; c$. Symmetry gives that $c \; R \; b$ and, by transitivity, $a \; R \; b$. Thus, we have $x \; R \; a$ for any arbitrary element $x \in [a]$ and, by transitivity, $x \; R \; b$, and thus $[a] \subseteq [b]$. Similarly, $[b] \subseteq [a]$, and thus $[a] = [b]$.

For the second part of the proof, let $\mathcal{A} = \{A_i\}$ be a partition of A, and define $R = \{(a, b) : \text{there exists } i \text{ such that } a \in A_i \text{ and } b \in A_i\}$. We claim that R is an equivalence relation on A. Reflexivity holds, since $a \in A_i$ implies $a \; R \; a$. Symmetry holds, because if $a \; R \; b$, then a and b belong to the same set A_i, and

hence $b\ R\ a$. If $a\ R\ b$ and $b\ R\ c$, then all three elements are in the same set A_i, and thus $a\ R\ c$ and transitivity holds. To see that the sets in the partition are the equivalence classes of R, observe that if $a \in A_i$, then $x \in [a]$ implies $x \in A_i$, and $x \in A_i$ implies $x \in [a]$. ■

A binary relation R on a set A is ***antisymmetric*** if

$$a\ R\ b \text{ and } b\ R\ a \text{ imply } a = b\ .$$

For example, the "$\leq$" relation on the natural numbers is antisymmetric, since $a \leq b$ and $b \leq a$ imply $a = b$. A relation that is reflexive, antisymmetric, and transitive is a ***partial order***, and we call a set on which a partial order is defined a ***partially ordered set***. For example, the relation "is a descendant of" is a partial order on the set of all people (if we view individuals as being their own descendants).

In a partially ordered set A, there may be no single "maximum" element a such that $b\ R\ a$ for all $b \in A$. Instead, the set may contain several ***maximal*** elements a such that for no $b \in A$, where $b \neq a$, is it the case that $a\ R\ b$. For example, a collection of different-sized boxes may contain several maximal boxes that don't fit inside any other box, yet it has no single "maximum" box into which any other box will fit.[3]

A relation R on a set A is a ***total relation*** if for all $a, b \in A$, we have $a\ R\ b$ or $b\ R\ a$ (or both), that is, if every pairing of elements of A is related by R. A partial order that is also a total relation is a ***total order*** or ***linear order***. For example, the relation "$\leq$" is a total order on the natural numbers, but the "is a descendant of" relation is not a total order on the set of all people, since there are individuals neither of whom is descended from the other. A total relation that is transitive, but not necessarily either symmetric or antisymmetric, is a ***total preorder***.

Exercises

B.2-1

Prove that the subset relation "$\subseteq$" on all subsets of $\mathbb{Z}$ is a partial order but not a total order.

B.2-2

Show that for any positive integer n, the relation "equivalent modulo n" is an equivalence relation on the integers. (We say that $a = b \pmod{n}$ if there exists an integer q such that $a - b = qn$.) Into what equivalence classes does this relation partition the integers?

---

[3] To be precise, in order for the "fit inside" relation to be a partial order, we need to view a box as fitting inside itself.

B.2-3
Give examples of relations that are

a. reflexive and symmetric but not transitive,

b. reflexive and transitive but not symmetric,

c. symmetric and transitive but not reflexive.

B.2-4
Let S be a finite set, and let R be an equivalence relation on $S \times S$. Show that if in addition R is antisymmetric, then the equivalence classes of S with respect to R are singletons.

B.2-5
Professor Narcissus claims that if a relation R is symmetric and transitive, then it is also reflexive. He offers the following proof. By symmetry, $a \; R \; b$ implies $b \; R \; a$. Transitivity, therefore, implies $a \; R \; a$. Is the professor correct?

B.3 Functions

Given two sets A and B, a ***function*** f is a binary relation on A and B such that for all $a \in A$, there exists precisely one $b \in B$ such that $(a, b) \in f$. The set A is called the ***domain*** of f, and the set B is called the ***codomain*** of f. We sometimes write $f : A \to B$, and if $(a, b) \in f$, we write $b = f(a)$, since the choice of a uniquely determines b.

Intuitively, the function f assigns an element of B to each element of A. No element of A is assigned two different elements of B, but the same element of B can be assigned to two different elements of A. For example, the binary relation

$$f = \{(a, b) : a, b \in \mathbb{N} \text{ and } b = a \bmod 2\}$$

is a function $f : \mathbb{N} \to \{0, 1\}$, since for each natural number a, there is exactly one value b in $\{0, 1\}$ such that $b = a \bmod 2$. For this example, $0 = f(0)$, $1 = f(1)$, $0 = f(2)$, $1 = f(3)$, etc. In contrast, the binary relation

$$g = \{(a, b) : a, b \in \mathbb{N} \text{ and } a + b \text{ is even}\}$$

is not a function, since $(1, 3)$ and $(1, 5)$ are both in g, and thus for the choice $a = 1$, there is not precisely one b such that $(a, b) \in g$.

Given a function $f : A \to B$, if $b = f(a)$, we say that a is the ***argument*** of f and that b is the ***value*** of f at a. We can define a function by stating its value for

every element of its domain. For example, we might define $f(n) = 2n$ for $n \in \mathbb{N}$, which means $f = \{(n, 2n) : n \in \mathbb{N}\}$. Two functions f and g are ***equal*** if they have the same domain and codomain and if $f(a) = g(a)$ for all a in the domain.

A ***finite sequence*** of length n is a function f whose domain is the set of n integers $\{0, 1, \ldots, n-1\}$. We often denote a finite sequence by listing its values in angle brackets: $\langle f(0), f(1), \ldots, f(n-1)\rangle$. An ***infinite sequence*** is a function whose domain is the set $\mathbb{N}$ of natural numbers. For example, the Fibonacci sequence, defined by recurrence (3.31), is the infinite sequence $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\rangle$.

When the domain of a function f is a Cartesian product, we often omit the extra parentheses surrounding the argument of f. For example, if we have a function $f : A_1 \times A_2 \times \cdots \times A_n \to B$, we write $b = f(a_1, a_2, \ldots, a_n)$ instead of writing $b = f((a_1, a_2, \ldots, a_n))$. We also call each a_i an ***argument*** to the function f, though technically f has just a single argument, which is the n-tuple $(a_1, a_2, \ldots, a_n)$.

If $f : A \to B$ is a function and $b = f(a)$, then we sometimes say that b is the ***image*** of a under f. The image of a set $A' \subseteq A$ under f is defined by

$$f(A') = \{b \in B : b = f(a) \text{ for some } a \in A'\} .$$

The ***range*** of f is the image of its domain, that is, $f(A)$. For example, the range of the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(n) = 2n$ is $f(\mathbb{N}) = \{m : m = 2n \text{ for some } n \in \mathbb{N}\}$, in other words, the set of nonnegative even integers.

A function is a ***surjection*** if its range is its codomain. For example, the function $f(n) = \lfloor n/2 \rfloor$ is a surjective function from $\mathbb{N}$ to $\mathbb{N}$, since every element in $\mathbb{N}$ appears as the value of f for some argument. In contrast, the function $f(n) = 2n$ is not a surjective function from $\mathbb{N}$ to $\mathbb{N}$, since no argument to f can produce any odd natural number as a value. The function $f(n) = 2n$ is, however, a surjective function from the natural numbers to the even numbers. A surjection $f : A \to B$ is sometimes described as mapping A ***onto*** B. When we say that f is onto, we mean that it is surjective.

A function $f : A \to B$ is an ***injection*** if distinct arguments to f produce distinct values, that is, if $a \neq a'$ implies $f(a) \neq f(a')$. For example, the function $f(n) = 2n$ is an injective function from $\mathbb{N}$ to $\mathbb{N}$, since each even number b is the image under f of at most one element of the domain, namely $b/2$. The function $f(n) = \lfloor n/2 \rfloor$ is not injective, since the value 1 is produced by two arguments: $f(2) = 1$ and $f(3) = 1$. An injection is sometimes called a ***one-to-one*** function.

A function $f : A \to B$ is a ***bijection*** if it is injective and surjective. For example, the function $f(n) = (-1)^n \lceil n/2 \rceil$ is a bijection from $\mathbb{N}$ to $\mathbb{Z}$:

$$\begin{aligned} 0 &\rightarrow 0\,, \\ 1 &\rightarrow -1\,, \\ 2 &\rightarrow 1\,, \\ 3 &\rightarrow -2\,, \\ 4 &\rightarrow 2\,, \\ &\vdots \end{aligned}$$

The function is injective, since no element of $\mathbb{Z}$ is the image of more than one element of $\mathbb{N}$. It is surjective, since every element of $\mathbb{Z}$ appears as the image of some element of $\mathbb{N}$. Hence, the function is bijective. A bijection is sometimes called a ***one-to-one correspondence***, since it pairs elements in the domain and codomain. A bijection from a set A to itself is sometimes called a ***permutation***.

When a function f is bijective, we define its ***inverse*** f^{-1} as

$$f^{-1}(b) = a \text{ if and only if } f(a) = b\,.$$

For example, the inverse of the function $f(n) = (-1)^n \lceil n/2 \rceil$ is

$$f^{-1}(m) = \begin{cases} 2m & \text{if } m \geq 0\,, \\ -2m-1 & \text{if } m < 0\,. \end{cases}$$

Exercises

B.3-1
Let A and B be finite sets, and let $f : A \rightarrow B$ be a function. Show the following:

a. If f is injective, then $|A| \leq |B|$.

b. If f is surjective, then $|A| \geq |B|$.

B.3-2
Is the function $f(x) = x + 1$ bijective when the domain and the codomain are the set $\mathbb{N}$? Is it bijective when the domain and the codomain are the set $\mathbb{Z}$?

B.3-3
Give a natural definition for the inverse of a binary relation such that if a relation is in fact a bijective function, its relational inverse is its functional inverse.

★ ***B.3-4***
Give a bijection from $\mathbb{Z}$ to $\mathbb{Z} \times \mathbb{Z}$.

B.4 Graphs

This section presents two kinds of graphs: directed and undirected. Certain definitions in the literature differ from those given here, but for the most part, the differences are slight. Section 20.1 shows how to represent graphs in computer memory.

A ***directed graph*** (or ***digraph***) G is a pair (V, E), where V is a finite set and E is a binary relation on V. The set V is called the ***vertex set*** of G, and its elements are called ***vertices*** (singular: ***vertex***). The set E is called the ***edge set*** of G, and its elements are called ***edges***. Figure B.2(a) is a pictorial representation of a directed graph on the vertex set $\{1, 2, 3, 4, 5, 6\}$. Vertices are represented by circles in the figure, and edges are represented by arrows. ***Self-loops***—edges from a vertex to itself—are possible.

In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices, rather than ordered pairs. That is, an edge is a set $\{u, v\}$, where $u, v \in V$ and $u \neq v$. By convention, we use the notation (u, v) for an edge, rather than the set notation $\{u, v\}$, and we consider (u, v) and (v, u) to be the same edge. In an undirected graph, self-loops are forbidden, so that every edge consists of two distinct vertices. Figure B.2(b) shows an undirected graph on the vertex set $\{1, 2, 3, 4, 5, 6\}$.

Many definitions for directed and undirected graphs are the same, although certain terms have slightly different meanings in the two contexts. If (u, v) is an edge in a directed graph $G = (V, E)$, we say that (u, v) is ***incident from*** or ***leaves*** vertex u and is ***incident to*** or ***enters*** vertex v. For example, the edges leaving vertex 2

Figure B.2 Directed and undirected graphs. **(a)** A directed graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. The edge $(2, 2)$ is a self-loop. **(b)** An undirected graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$. The vertex 4 is isolated. **(c)** The subgraph of the graph in part (a) induced by the vertex set $\{1, 2, 3, 6\}$.

in Figure B.2(a) are $(2, 2)$, $(2, 4)$, and $(2, 5)$. The edges entering vertex 2 are $(1, 2)$ and $(2, 2)$. If (u, v) is an edge in an undirected graph $G = (V, E)$, we say that (u, v) is ***incident on*** vertices u and v. In Figure B.2(b), the edges incident on vertex 2 are $(1, 2)$ and $(2, 5)$.

If (u, v) is an edge in a graph $G = (V, E)$, we say that vertex v is ***adjacent*** to vertex u. When the graph is undirected, the adjacency relation is symmetric. When the graph is directed, the adjacency relation is not necessarily symmetric. If v is adjacent to u in a directed graph, we can write $u \to v$. In parts (a) and (b) of Figure B.2, vertex 2 is adjacent to vertex 1, since the edge $(1, 2)$ belongs to both graphs. Vertex 1 is *not* adjacent to vertex 2 in Figure B.2(a), since the edge $(2, 1)$ is absent.

The ***degree*** of a vertex in an undirected graph is the number of edges incident on it. For example, vertex 2 in Figure B.2(b) has degree 2. A vertex whose degree is 0, such as vertex 4 in Figure B.2(b), is ***isolated***. In a directed graph, the ***out-degree*** of a vertex is the number of edges leaving it, and the ***in-degree*** of a vertex is the number of edges entering it. The ***degree*** of a vertex in a directed graph is its in-degree plus its out-degree. Vertex 2 in Figure B.2(a) has in-degree 2, out-degree 3, and degree 5.

A ***path*** of ***length*** k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ of vertices such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. The length of the path is the number of edges in the path, which is 1 less than the number of vertices in the path. The path ***contains*** the vertices $v_0, v_1, \ldots, v_k$ and the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. (There is always a 0-length path from u to u.) If there is a path p from u to u', we say that u' is ***reachable*** from u via p, which we can write as $u \overset{p}{\rightsquigarrow} u'$. A path is ***simple***[4] if all vertices in the path are distinct. In Figure B.2(a), the path $\langle 1, 2, 5, 4 \rangle$ is a simple path of length 3. The path $\langle 2, 5, 4, 5 \rangle$ is not simple. A ***subpath*** of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a contiguous subsequence of its vertices. That is, for any $0 \le i \le j \le k$, the subsequence of vertices $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a subpath of p.

In a directed graph, a path $\langle v_0, v_1, \ldots, v_k \rangle$ forms a ***cycle*** if $v_0 = v_k$ and the path contains at least one edge. The cycle is ***simple*** if, in addition, $v_1, v_2, \ldots, v_k$ are distinct. A cycle consisting of k vertices has ***length*** k. A self-loop is a cycle of length 1. Two paths $\langle v_0, v_1, v_2, \ldots, v_{k-1}, v_0 \rangle$ and $\langle v'_0, v'_1, v'_2, \ldots, v'_{k-1}, v'_0 \rangle$ form the same cycle if there exists an integer j such that $v'_i = v_{(i+j) \bmod k}$ for $i = 0, 1, \ldots, k-1$. In Figure B.2(a), the path $\langle 1, 2, 4, 1 \rangle$ forms the same cycle as the paths $\langle 2, 4, 1, 2 \rangle$ and $\langle 4, 1, 2, 4 \rangle$. This cycle is simple, but the cycle $\langle 1, 2, 4, 5, 4, 1 \rangle$ is not. The cycle $\langle 2, 2 \rangle$ formed by the edge $(2, 2)$ is a self-loop. A directed graph

[4] Some authors refer to what we call a path as a "walk" and to what we call a simple path as just a "path."

with no self-loops is ***simple***. In an undirected graph, a path $\langle v_0, v_1, \ldots, v_k \rangle$ forms a ***cycle*** if $k > 0$, $v_0 = v_k$, and all edges on the path are distinct. The cycle is ***simple*** if $v_1, v_2, \ldots, v_k$ are distinct. For example, in Figure B.2(b), the path $\langle 1, 2, 5, 1 \rangle$ is a simple cycle. A graph with no simple cycles is ***acyclic***.

An undirected graph is ***connected*** if every vertex is reachable from all other vertices. The ***connected components*** of an undirected graph are the equivalence classes of vertices under the "is reachable from" relation. The graph shown in Figure B.2(b) has three connected components: $\{1, 2, 5\}$, $\{3, 6\}$, and $\{4\}$. Every vertex in the connected component $\{1, 2, 5\}$ is reachable from every other vertex in $\{1, 2, 5\}$. An undirected graph is connected if it has exactly one connected component. The edges of a connected component are those that are incident on only the vertices of the component. In other words, edge (u, v) is an edge of a connected component only if both u and v are vertices of the component.

A directed graph is ***strongly connected*** if every two vertices are reachable from each other. The ***strongly connected components*** of a directed graph are the equivalence classes of vertices under the "are mutually reachable" relation. A directed graph is strongly connected if it has only one strongly connected component. The graph in Figure B.2(a) has three strongly connected components: $\{1, 2, 4, 5\}$, $\{3\}$, and $\{6\}$. All pairs of vertices in $\{1, 2, 4, 5\}$ are mutually reachable. The vertices $\{3, 6\}$ do not form a strongly connected component, since vertex 6 cannot be reached from vertex 3.

Two graphs $G = (V, E)$ and $G' = (V', E')$ are ***isomorphic*** if there exists a bijection $f : V \to V'$ such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$. In other words, G and G' are isomorphic if the vertices of G can be relabeled to be vertices of G', maintaining the corresponding edges in G and G'. Figure B.3(a) shows a pair of isomorphic graphs G and G' with respective vertex sets $V = \{1, 2, 3, 4, 5, 6\}$ and $V' = \{u, v, w, x, y, z\}$. The mapping from V to V' given by $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$ provides the required bijective function. The graphs in Figure B.3(b) are not isomorphic. Although both graphs have 5 vertices and 7 edges, the top graph has a vertex of degree 4 and the bottom graph does not.

We say that a graph $G' = (V', E')$ is a ***subgraph*** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Given a set $V' \subseteq V$, the subgraph of G ***induced*** by V' is the graph $G' = (V', E')$, where

$$E' = \{(u, v) \in E : u, v \in V'\} .$$

The subgraph induced by the vertex set $\{1, 2, 3, 6\}$ in Figure B.2(a) appears in Figure B.2(c) and has the edge set $\{(1, 2), (2, 2), (6, 3)\}$.

Given an undirected graph $G = (V, E)$, the ***directed version*** of G is the directed graph $G' = (V, E')$, where $(u, v) \in E'$ if and only if $(u, v) \in E$. That is, each undirected edge (u, v) in G turns into two directed edges, (u, v) and (v, u), in the

Figure B.3 **(a)** A pair of isomorphic graphs. The vertices of the top graph are mapped to the vertices of the bottom graph by $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$. **(b)** Two graphs that are not isomorphic. The top graph has a vertex of degree 4, and the bottom graph does not.

directed version. Given a directed graph $G = (V, E)$, the ***undirected version*** of G is the undirected graph $G' = (V, E')$, where $(u, v) \in E'$ if and only if $u \neq v$ and E contains at least one of the edges (u, v) and (v, u). That is, the undirected version contains the edges of G "with their directions removed" and with self-loops eliminated. (Since (u, v) and (v, u) are the same edge in an undirected graph, the undirected version of a directed graph contains it only once, even if the directed graph contains both edges (u, v) and (v, u).) In a directed graph $G = (V, E)$, a ***neighbor*** of a vertex u is any vertex that is adjacent to u in the undirected version of G. That is, v is a neighbor of u if $u \neq v$ and either $(u, v) \in E$ or $(v, u) \in E$. In an undirected graph, u and v are neighbors if they are adjacent.

Several kinds of graphs have special names. A ***complete graph*** is an undirected graph in which every pair of vertices is adjacent. An undirected graph $G = (V, E)$ is ***bipartite*** if V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. That is, all edges go between the two sets V_1 and V_2. An acyclic, undirected graph is a ***forest***, and a connected, acyclic, undirected graph is a ***(free) tree*** (see Section B.5). We often take the first letters of "directed acyclic graph" and call such a graph a ***dag***.

There are two variants of graphs that you may occasionally encounter. A ***multigraph*** is like an undirected graph, but it can have both multiple edges between vertices (such as two distinct edges (u, v) and (u, v)) and self-loops. A ***hypergraph*** is like an undirected graph, but each ***hyperedge***, rather than connecting two vertices,

connects an arbitrary subset of vertices. Many algorithms written for ordinary directed and undirected graphs can be adapted to run on these graphlike structures.

The ***contraction*** of an undirected graph $G = (V, E)$ by an edge $e = (u, v)$ is a graph $G' = (V', E')$, where $V' = V - \{u, v\} \cup \{x\}$ and x is a new vertex. The set of edges E' is formed from E by deleting the edge (u, v) and, for each vertex w adjacent to u or v, deleting whichever of (u, w) and (v, w) belongs to E and adding the new edge (x, w). In effect, u and v are "contracted" into a single vertex.

Exercises

B.4-1

Attendees of a faculty party shake hands to greet each other, with every pair of professors shaking hands one time. Each professor remembers the number of times he or she shook hands. At the end of the party, the department head asks the professors for their totals and adds them all up. Show that the result is even by proving the ***handshaking lemma***: if $G = (V, E)$ is an undirected graph, then

$$\sum_{v \in V} \text{degree}(v) = 2\,|E| \ .$$

B.4-2

Show that if a directed or undirected graph contains a path between two vertices u and v, then it contains a simple path between u and v. Show that if a directed graph contains a cycle, then it contains a simple cycle.

B.4-3

Show that any connected, undirected graph $G = (V, E)$ satisfies $|E| \geq |V| - 1$.

B.4-4

Verify that in an undirected graph, the "is reachable from" relation is an equivalence relation on the vertices of the graph. Which of the three properties of an equivalence relation hold in general for the "is reachable from" relation on the vertices of a directed graph?

B.4-5

What is the undirected version of the directed graph in Figure B.2(a)? What is the directed version of the undirected graph in Figure B.2(b)?

B.4-6

Show how a bipartite graph can represent a hypergraph by letting incidence in the hypergraph correspond to adjacency in the bipartite graph. (*Hint:* Let one set of

vertices in the bipartite graph correspond to vertices of the hypergraph, and let the other set of vertices of the bipartite graph correspond to hyperedges.)

B.5 Trees

As with graphs, there are many related, but slightly different, notions of trees. This section presents definitions and mathematical properties of several kinds of trees. Sections 10.3 and 20.1 describe how to represent trees in computer memory.

B.5.1 Free trees

As defined in Section B.4, a ***free tree*** is a connected, acyclic, undirected graph. We often omit the adjective "free" when we say that a graph is a tree. If an undirected graph is acyclic but possibly disconnected, it is a ***forest***. Many algorithms that work for trees also work for forests. Figure B.4(a) shows a free tree, and Figure B.4(b) shows a forest. The forest in Figure B.4(b) is not a tree because it is not connected. The graph in Figure B.4(c) is connected but neither a tree nor a forest, because it contains a cycle.

The following theorem captures many important facts about free trees.

Theorem B.2 (Properties of free trees)
Let $G = (V, E)$ be an undirected graph. The following statements are equivalent.

1. G is a free tree.
2. Any two vertices in G are connected by a unique simple path.
3. G is connected, but if any edge is removed from E, the resulting graph is disconnected.

Figure B.4 **(a)** A free tree. **(b)** A forest. **(c)** A graph that contains a cycle and is therefore neither a tree nor a forest.

Figure B.5 A step in the proof of Theorem B.2: if (1) G is a free tree, then (2) any two vertices in G are connected by a unique simple path. Assume for the sake of contradiction that vertices u and v are connected by two distinct simple paths. These paths first diverge at vertex w, and they first reconverge at vertex z. The path p' concatenated with the reverse of the path p'' forms a cycle, which yields the contradiction.

4. G is connected, and $|E| = |V| - 1$.
5. G is acyclic, and $|E| = |V| - 1$.
6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.

Proof (1) $\Rightarrow$ (2): Since a tree is connected, any two vertices in G are connected by at least one simple path. Suppose for the sake of contradiction that vertices u and v are connected by two distinct simple paths as shown in Figure B.5. Let w be the vertex at which the paths first diverge. That is, if we call the paths p_1 and p_2, then w is the first vertex on both p_1 and p_2 whose successor on p_1 is x and whose successor on p_2 is y, where $x \neq y$. Let z be the first vertex at which the paths reconverge, that is, z is the first vertex following w on p_1 that is also on p_2. Let $p' = w \to x \rightsquigarrow z$ be the subpath of p_1 from w through x to z, so that $p_1 = u \rightsquigarrow w \overset{p'}{\rightsquigarrow} z \rightsquigarrow v$, and let $p'' = w \to y \rightsquigarrow z$ be the subpath of p_2 from w through y to z, so that $p_2 = u \rightsquigarrow w \overset{p''}{\rightsquigarrow} z \rightsquigarrow v$. Paths p' and p'' share no vertices except their endpoints. Then, as Figure B.5 shows, the path obtained by concatenating p' and the reverse of p'' is a cycle, which contradicts our assumption that G is a tree. Thus, if G is a tree, there can be at most one simple path between two vertices.

(2) $\Rightarrow$ (3): If any two vertices in G are connected by a unique simple path, then G is connected. Let (u, v) be any edge in E. This edge is a path from u to v, and so it must be the unique path from u to v. If (u, v) were to be removed from G, there would be no path from u to v, and G would be disconnected.

(3) $\Rightarrow$ (4): By assumption, the graph G is connected, so Exercise B.4-3 gives that $|E| \geq |V| - 1$. We prove $|E| \leq |V| - 1$ by induction on $|V|$. The base cases are when $|V| = 1$ or $|V| = 2$, and in either case, $|E| = |V| - 1$. For the inductive step, suppose that $|V| \geq 3$ for graph G and that any graph $G' = (V', E')$, where

$|V'| < |V|$, that satisfies (3) also satisfies $|E'| \leq |V'| - 1$. Removing an arbitrary edge from G separates the graph into $k \geq 2$ connected components (actually $k = 2$). Each component satisfies (3), or else G would not satisfy (3). Consider each connected component V_i, with edge set E_i, as a separate free tree. Then, because each connected component has fewer than $|V|$ vertices, the inductive hypothesis implies that $|E_i| \leq |V_i| - 1$. Thus, the number of edges in all k connected components combined is at most $|V| - k \leq |V| - 2$. Adding in the removed edge yields $|E| \leq |V| - 1$.

(4) $\Rightarrow$ (5): Suppose that G is connected and that $|E| = |V| - 1$. We must show that G is acyclic. Suppose that G has a cycle containing k vertices $v_1, v_2, \ldots, v_k$, and without loss of generality assume that this cycle is simple. Let $G_k = (V_k, E_k)$ be the subgraph of G consisting of the cycle, so that $|V_k| = |E_k| = k$. If $k < |V|$, then because G is connected, there must be a vertex $v_{k+1} \in V - V_k$ that is adjacent to some vertex $v_i \in V_k$. Define $G_{k+1} = (V_{k+1}, E_{k+1})$ to be the subgraph of G with $V_{k+1} = V_k \cup \{v_{k+1}\}$ and $E_{k+1} = E_k \cup \{(v_i, v_{k+1})\}$. Note that $|V_{k+1}| = |E_{k+1}| = k + 1$. If $k + 1 < |V|$, then continue, defining G_{k+2} in the same manner, and so forth, until we obtain $G_n = (V_n, E_n)$, where $n = |V|$, $V_n = V$, and $|E_n| = |V_n| = |V|$. Since G_n is a subgraph of G, we have $E_n \subseteq E$, and hence $|E| \geq |E_n| = |V_n| = |V|$, which contradicts the assumption that $|E| = |V| - 1$. Thus, G is acyclic.

(5) $\Rightarrow$ (6): Suppose that G is acyclic and that $|E| = |V| - 1$. Let k be the number of connected components of G. Each connected component is a free tree by definition, and since (1) implies (5), the sum of all edges in all connected components of G is $|V| - k$. Consequently, k must equal 1, and G is in fact a tree. Since (1) implies (2), any two vertices in G are connected by a unique simple path. Thus, adding any edge to G creates a cycle.

(6) $\Rightarrow$ (1): Suppose that G is acyclic but that adding any edge to E creates a cycle. We must show that G is connected. Let u and v be arbitrary vertices in G. If u and v are not already adjacent, adding the edge (u, v) creates a cycle in which all edges but (u, v) belong to G. Thus, the cycle minus edge (u, v) must contain a path from u to v, and since u and v were chosen arbitrarily, G is connected. ■

B.5.2 Rooted and ordered trees

A ***rooted tree*** is a free tree in which one of the vertices is distinguished from the others. We call the distinguished vertex the ***root*** of the tree. We often refer to a

Figure B.6 Rooted and ordered trees. **(a)** A rooted tree with height 4. The tree is drawn in a standard way: the root (node 7) is at the top, its children (nodes with depth 1) are beneath it, their children (nodes with depth 2) are beneath them, and so forth. If the tree is ordered, the relative left-to-right order of the children of a node matters; otherwise, it doesn't. **(b)** Another rooted tree. As a rooted tree, it is identical to the tree in (a), but as an ordered tree it is different, since the children of node 3 appear in a different order.

vertex of a rooted tree as a ***node***[5] of the tree. Figure B.6(a) shows a rooted tree on a set of 12 nodes with root 7.

Consider a node x in a rooted tree T with root r. We call any node y on the unique simple path from r to x an ***ancestor*** of x. If y is an ancestor of x, then x is a ***descendant*** of y. (Every node is both an ancestor and a descendant of itself.) If y is an ancestor of x and $x \neq y$, then y is a ***proper ancestor*** of x and x is a ***proper descendant*** of y. The ***subtree rooted at x*** is the tree induced by descendants of x, rooted at x. For example, the subtree rooted at node 8 in Figure B.6(a) contains nodes 8, 6, 5, and 9.

If the last edge on the simple path from the root r of a tree T to a node x is (y, x), then y is the ***parent*** of x, and x is a ***child*** of y. The root is the only node in T with no parent. If two nodes have the same parent, they are ***siblings***. A node with no children is a ***leaf*** or ***external node***. A nonleaf node is an ***internal node***.

---

[5] The term "node" is often used in the graph theory literature as a synonym for "vertex." We reserve the term "node" to mean a vertex of a rooted tree.

The number of children of a node x in a rooted tree T is the ***degree*** of x.[6] The length of the simple path from the root r to a node x is the ***depth*** of x in T. A ***level*** of a tree consists of all nodes at the same depth. The ***height*** of a node in a tree is the number of edges on the longest simple downward path from the node to a leaf, and the height of a tree is the height of its root. The height of a tree is also equal to the largest depth of any node in the tree.

An ***ordered tree*** is a rooted tree in which the children of each node are ordered. That is, if a node has k children, then there is a first child, a second child, and so on, up to and including a kth child. The two trees in Figure B.6 are different when considered to be ordered trees, but the same when considered to be just rooted trees.

B.5.3 Binary and positional trees

We define binary trees recursively. A ***binary tree*** T is a structure defined on a finite set of nodes that either

- contains no nodes, or
- is composed of three disjoint sets of nodes: a ***root*** node, a binary tree called its ***left subtree***, and a binary tree called its ***right subtree***.

The binary tree that contains no nodes is called the ***empty tree*** or ***null tree***, sometimes denoted NIL. If the left subtree is nonempty, its root is called the ***left child*** of the root of the entire tree. Likewise, the root of a nonnull right subtree is the ***right child*** of the root of the entire tree. If a subtree is the null tree NIL, we say that the child is ***absent*** or ***missing***. Figure B.7(a) shows a binary tree.

A binary tree is not simply an ordered tree in which each node has degree at most 2. For example, in a binary tree, if a node has just one child, the position of the child—whether it is the ***left child*** or the ***right child***—matters. In an ordered tree, there is no distinguishing a sole child as being either left or right. Figure B.7(b) shows a binary tree that differs from the tree in Figure B.7(a) because of the position of one node. Considered as ordered trees, however, the two trees are identical.

One way to represent the positioning information in a binary tree is by the internal nodes of an ordered tree, as shown in Figure B.7(c). The idea is to replace each missing child in the binary tree with a node having no children. These leaf nodes

---

[6] The degree of a node depends on whether we consider T to be a rooted tree or a free tree. The degree of a vertex in a free tree is, as in any undirected graph, the number of adjacent vertices. In a rooted tree, however, the degree is the number of children—the parent of a node does not count toward its degree.

Figure B.7 Binary trees. **(a)** A binary tree drawn in a standard way. The left child of a node is drawn beneath the node and to the left. The right child is drawn beneath and to the right. **(b)** A binary tree different from the one in (a). In (a), the left child of node 7 is 5 and the right child is absent. In (b), the left child of node 7 is absent and the right child is 5. As ordered trees, these trees are the same, but as binary trees, they are distinct. **(c)** The binary tree in (a) represented by the internal nodes of a full binary tree: an ordered tree in which each internal node has degree 2. The leaves in the tree are shown as squares.

are drawn as squares in the figure. The tree that results is a ***full binary tree***: each node is either a leaf or has degree exactly 2. No nodes have degree 1. Consequently, the order of the children of a node preserves the position information.

The positioning information that distinguishes binary trees from ordered trees extends to trees with more than two children per node. In a ***positional tree***, the children of a node are labeled with distinct positive integers. The ith child of a node is ***absent*** if no child is labeled with integer i. A ***k-ary*** tree is a positional tree in which for every node, all children with labels greater than k are missing. Thus, a binary tree is a k-ary tree with $k = 2$.

A ***complete k-ary tree*** is a k-ary tree in which all leaves have the same depth and all internal nodes have degree k. Figure B.8 shows a complete binary tree of height 3. How many leaves does a complete k-ary tree of height h have? The root has k children at depth 1, each of which has k children at depth 2, etc. Thus, the number of nodes at depth d is k^d. In a complete k-ary tree with height h, the leaves are at depth h, so that there are k^h leaves. Consequently, the height of a complete k-ary tree with n leaves is $\log_k n$. A complete k-ary tree of height h has

$$\begin{aligned} 1 + k + k^2 + \cdots + k^{h-1} &= \sum_{d=0}^{h-1} k^d \\ &= \frac{k^h - 1}{k - 1} \quad \text{(by equation (A.6) on page 1142)} \end{aligned}$$

internal nodes. Thus, a complete binary tree has $2^h - 1$ internal nodes.

Figure B.8 A complete binary tree of height 3 with 8 leaves and 7 internal nodes.

Exercises

B.5-1
Draw all the free trees composed of the three vertices x, y, and z. Draw all the rooted trees with nodes x, y, and z with x as the root. Draw all the ordered trees with nodes x, y, and z with x as the root. Draw all the binary trees with nodes x, y, and z with x as the root.

B.5-2
Let $G = (V, E)$ be a directed acyclic graph in which there is a vertex $v_0 \in V$ such that there exists a unique path from v_0 to every vertex $v \in V$. Prove that the undirected version of G forms a tree.

B.5-3
Show by induction that the number of degree-2 nodes in any nonempty binary tree is one less than the number of leaves. Conclude that the number of internal nodes in a full binary tree is one less than the number of leaves.

B.5-4
Prove that for any integer $k \geq 1$, there is a full binary tree with k leaves.

B.5-5
Use induction to show that a nonempty binary tree with n nodes has height at least $\lfloor \lg n \rfloor$.

★ ***B.5-6***
The ***internal path length*** of a full binary tree is the sum, taken over all internal nodes of the tree, of the depth of each node. Likewise, the ***external path length*** is the sum, taken over all leaves of the tree, of the depth of each leaf. Consider a full binary tree with n internal nodes, internal path length i, and external path length e. Prove that $e = i + 2n$.

★ ***B.5-7***
Associate a "weight" $w(x) = 2^{-d}$ with each leaf x of depth d in a binary tree T, and let L be the set of leaves of T. Prove the ***Kraft inequality***: $\sum_{x \in L} w(x) \le 1$.

★ ***B.5-8***
Show that if $L \ge 2$, then every binary tree with L leaves contains a subtree having between $L/3$ and $2L/3$ leaves, inclusive.

Problems

B-1 Graph coloring
A ***k-coloring*** of undirected graph $G = (V, E)$ is a function $c : V \to \{1, 2, \ldots, k\}$ such that $c(u) \ne c(v)$ for every edge $(u, v) \in E$. In other words, the numbers $1, 2, \ldots, k$ represent the k colors, and adjacent vertices must have different colors.

a. Show that any tree is 2-colorable.

b. Show that the following are equivalent:

1. G is bipartite.
2. G is 2-colorable.
3. G has no cycles of odd length.

c. Let d be the maximum degree of any vertex in a graph G. Prove that G can be colored with $d + 1$ colors.

d. Show that if G has $O(|V|)$ edges, then G can be colored with $O(\sqrt{|V|})$ colors.

B-2 Friendly graphs
Reword each of the following statements as a theorem about undirected graphs, and then prove it. Assume that friendship is symmetric but not reflexive.

a. Any group of at least two people contains at least two people with the same number of friends in the group.

b. Every group of six people contains either at least three mutual friends or at least three mutual strangers.

c. Any group of people can be partitioned into two subgroups such that at least half the friends of each person belong to the subgroup of which that person is *not* a member.

d. If everyone in a group is the friend of at least half the people in the group, then the group can be seated around a table in such a way that everyone is seated between two friends.

B-3 Bisecting trees

Many divide-and-conquer algorithms that operate on graphs require that the graph be bisected into two nearly equal-sized subgraphs, which are induced by a partition of the vertices. This problem investigates bisections of trees formed by removing a small number of edges. We require that whenever two vertices end up in the same subtree after removing edges, then they must belong to the same partition.

a. Show that the vertices of any n-vertex binary tree can be partitioned into two sets A and B, such that $|A| \leq 3n/4$ and $|B| \leq 3n/4$, by removing a single edge.

b. Show that the constant $3/4$ in part (a) is optimal in the worst case by giving an example of a simple binary tree whose most evenly balanced partition upon removal of a single edge has $|A| = 3n/4$.

c. Show that by removing at most $O(\lg n)$ edges, we can partition the vertices of any n-vertex binary tree into two sets A and B such that $|A| = \lfloor n/2 \rfloor$ and $|B| = \lceil n/2 \rceil$.

Appendix notes

G. Boole pioneered the development of symbolic logic, and he introduced many of the basic set notations in a book published in 1854. Modern set theory was created by G. Cantor during the period 1874–1895. Cantor focused primarily on sets of infinite cardinality. The term "function" is attributed to G. W. Leibniz, who used it to refer to several kinds of mathematical formulas. His limited definition has been generalized many times. Graph theory originated in 1736, when L. Euler proved that it was impossible to cross each of the seven bridges in the city of Königsberg exactly once and return to the starting point.

The book by Harary [208] provides a useful compendium of many definitions and results from graph theory.

C Counting and Probability

This appendix reviews elementary combinatorics and probability theory. If you have a good background in these areas, you may want to skim the beginning of this appendix lightly and concentrate on the later sections. Most of this book's chapters do not require probability, but for some chapters it is essential.

Section C.1 reviews elementary results in counting theory, including standard formulas for counting permutations and combinations. The axioms of probability and basic facts concerning probability distributions form Section C.2. Random variables are introduced in Section C.3, along with the properties of expectation and variance. Section C.4 investigates the geometric and binomial distributions that arise from studying Bernoulli trials. The study of the binomial distribution continues in Section C.5, an advanced discussion of the "tails" of the distribution.

C.1 Counting

Counting theory tries to answer the question "How many?" without actually enumerating all the choices. For example, you might ask, "How many different n-bit numbers are there?" or "How many orderings of n distinct elements are there?" This section reviews the elements of counting theory. Since some of the material assumes a basic understanding of sets, you might wish to start by reviewing the material in Section B.1.

Rules of sum and product

We can sometimes express a set of items that we wish to count as a union of disjoint sets or as a Cartesian product of sets.

The ***rule of sum*** says that the number of ways to choose one element from one of two *disjoint* sets is the sum of the cardinalities of the sets. That is, if A and B are two finite sets with no members in common, then $|A \cup B| = |A| + |B|$, which

follows from equation (B.3) on page 1156. For example, if each position on a car's license plate is a letter or a digit, then the number of possibilities for each position is $26 + 10 = 36$, since there are 26 choices if it is a letter and 10 choices if it is a digit.

The ***rule of product*** says that the number of ways to choose an ordered pair is the number of ways to choose the first element times the number of ways to choose the second element. That is, if A and B are two finite sets, then $|A \times B| = |A| \cdot |B|$, which is simply equation (B.4) on page 1157. For example, if an ice-cream parlor offers 28 flavors of ice cream and four toppings, the number of possible sundaes with one scoop of ice cream and one topping is $28 \cdot 4 = 112$.

Strings

A ***string*** over a finite set S is a sequence of elements of S. For example, there are eight binary strings of length 3:

$$000, 001, 010, 011, 100, 101, 110, 111 .$$

(Here we use the shorthand of omitting the angle brackets when denoting a sequence.) We sometimes call a string of length k a ***k-string***. A ***substring*** s' of a string s is an ordered sequence of consecutive elements of s. A ***k-substring*** of a string is a substring of length k. For example, 010 is a 3-substring of 01101001 (the 3-substring that begins in position 4), but 111 is not a substring of 01101001.

We can view a k-string over a set S as an element of the Cartesian product S^k of k-tuples, which means that there are $|S|^k$ strings of length k. For example, the number of binary k-strings is 2^k. Intuitively, to construct a k-string over an n-set, there are n ways to pick the first element; for each of these choices, there are n ways to pick the second element; and so forth k times. This construction leads to the k-fold product $\underbrace{n \cdot n \cdots n}_{n \text{ times}} = n^k$ as the number of k-strings.

Permutations

A ***permutation*** of a finite set S is an ordered sequence of all the elements of S, with each element appearing exactly once. For example, if $S = \{a, b, c\}$, then S has 6 permutations:

$$abc, acb, bac, bca, cab, cba .$$

(Again, we use the shorthand of omitting the angle brackets when denoting a sequence.) There are $n!$ permutations of a set of n elements, since there are n ways to choose the first element of the sequence, $n - 1$ ways for the second element, $n - 2$ ways for the third, and so on.

A ***k*-permutation** of S is an ordered sequence of k elements of S, with no element appearing more than once in the sequence. (Thus, an ordinary permutation is an n-permutation of an n-set.) Here are the 2-permutations of the set $\{a, b, c, d\}$:

$$ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc \ .$$

The number of k-permutations of an n-set is

$$n(n-1)(n-2)\cdots(n-k+1) = \frac{n!}{(n-k)!} \ , \tag{C.1}$$

since there are n ways to choose the first element, $n-1$ ways to choose the second element, and so on, until k elements are chosen, with the last element chosen from the remaining $n-k+1$ elements. For the above example, with $n = 4$ and $k = 2$, the formula (C.1) evaluates to $4!/2! = 12$, matching the number of 2-permutations listed.

Combinations

A ***k*-combination** of an n-set S is simply a k-subset of S. For example, the 4-set $\{a, b, c, d\}$ has six 2-combinations:

$$ab, ac, ad, bc, bd, cd \ .$$

(Here we use the shorthand of omitting the braces around each subset.) To construct a k-combination of an n-set, choose k distinct (different) elements from the n-set. The order of selecting the elements does not matter.

We can express the number of k-combinations of an n-set in terms of the number of k-permutations of an n-set. Every k-combination has exactly $k!$ permutations of its elements, each of which is a distinct k-permutation of the n-set. Thus the number of k-combinations of an n-set is the number of k-permutations divided by $k!$. From equation (C.1), this quantity is

$$\frac{n!}{k!\,(n-k)!} \ . \tag{C.2}$$

For $k = 0$, this formula tells us that the number of ways to choose 0 elements from an n-set is 1 (not 0), since $0! = 1$.

Binomial coefficients

The notation $\binom{n}{k}$ (read "n choose k") denotes the number of k-combinations of an n-set. Equation (C.2) gives

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \ .$$

This formula is symmetric in k and $n - k$:

$$\binom{n}{k} = \binom{n}{n-k} . \tag{C.3}$$

These numbers are also known as ***binomial coefficients***, due to their appearance in the ***binomial theorem***:

$$(x + y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} , \tag{C.4}$$

where $n \in \mathbb{N}$ and $x, y \in \mathbb{R}$. The right-hand side of equation (C.4) is called the ***binomial expansion*** of the left-hand side. A special case of the binomial theorem occurs when $x = y = 1$:

$$2^n = \sum_{k=0}^{n} \binom{n}{k} .$$

This formula corresponds to counting the 2^n binary n-strings by the number of 1s they contain: $\binom{n}{k}$ binary n-strings contain exactly k 1s, since there are $\binom{n}{k}$ ways to choose k out of the n positions in which to place the 1s.

Many identities involve binomial coefficients. The exercises at the end of this section give you the opportunity to prove a few.

Binomial bounds

You sometimes need to bound the size of a binomial coefficient. For $1 \le k \le n$, we have the lower bound

$$\begin{aligned}
\binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \\
&= \left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-k+1}{1}\right) \\
&\ge \left(\frac{n}{k}\right)^k .
\end{aligned} \tag{C.5}$$

Taking advantage of the inequality $k! \ge (k/e)^k$ derived from Stirling's approximation (3.25) on page 67, we obtain the upper bounds

$$\begin{aligned}
\binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \\
&\le \frac{n^k}{k!} \\
&\le \left(\frac{en}{k}\right)^k .
\end{aligned} \tag{C.6}$$

For all integers k such that $0 \le k \le n$, you can use induction (see Exercise C.1-12) to prove the bound

$$\binom{n}{k} \le \frac{n^n}{k^k (n-k)^{n-k}} , \tag{C.7}$$

where for convenience we assume that $0^0 = 1$. For $k = \lambda n$, where $0 \le \lambda \le 1$, we can rewrite this bound as

$$\begin{aligned}\binom{n}{\lambda n} &\le \frac{n^n}{(\lambda n)^{\lambda n}((1-\lambda)n)^{(1-\lambda)n}} \\ &= \left(\left(\frac{1}{\lambda}\right)^{\lambda}\left(\frac{1}{1-\lambda}\right)^{1-\lambda}\right)^{n} \\ &= 2^{n\,H(\lambda)} ,\end{aligned}$$

where

$$H(\lambda) = -\lambda \lg \lambda - (1-\lambda)\lg(1-\lambda) \tag{C.8}$$

is the ***(binary) entropy function*** and where, for convenience, we assume that $0 \lg 0 = 0$, so that $H(0) = H(1) = 0$.

Exercises

C.1-1
How many k-substrings does an n-string have? (Consider identical k-substrings at different positions to be different.) How many substrings does an n-string have in total?

C.1-2
An n-input, m-output ***boolean function*** is a function from $\{0,1\}^n$ to $\{0,1\}^m$. How many n-input, 1-output boolean functions are there? How many n-input, m-output boolean functions are there?

C.1-3
In how many ways can n professors sit around a circular conference table? Consider two seatings to be the same if one can be rotated to form the other.

C.1-4
In how many ways is it possible to choose three distinct numbers from the set $\{1, 2, \ldots, 99\}$ so that their sum is even?

C.1-5
Prove the identity

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \tag{C.9}$$

for $0 < k \le n$.

C.1-6
Prove the identity

$$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$$

for $0 \le k < n$.

C.1-7
To choose k objects from n, you can make one of the objects distinguished and consider whether the distinguished object is chosen. Use this approach to prove that

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} .$$

C.1-8
Using the result of Exercise C.1-7, make a table for $n = 0, 1, \ldots, 6$ and $0 \le k \le n$ of the binomial coefficients $\binom{n}{k}$ with $\binom{0}{0}$ at the top, $\binom{1}{0}$ and $\binom{1}{1}$ on the next line, then $\binom{2}{0}$, $\binom{2}{1}$, and $\binom{2}{2}$, and so forth. Such a table of binomial coefficients is called ***Pascal's triangle***.

C.1-9
Prove that

$$\sum_{i=1}^{n} i = \binom{n+1}{2} .$$

C.1-10
Show that for any integers $n \ge 0$ and $0 \le k \le n$, the expression $\binom{n}{k}$ achieves its maximum value when $k = \lfloor n/2 \rfloor$ or $k = \lceil n/2 \rceil$.

★ ***C.1-11***
Argue that for any integers $n \geq 0$, $j \geq 0$, $k \geq 0$, and $j + k \leq n$,

$$\binom{n}{j+k} \leq \binom{n}{j}\binom{n-j}{k} . \tag{C.10}$$

Provide both an algebraic proof and an argument based on a method for choosing $j + k$ items out of n. Give an example in which equality does not hold.

★ ***C.1-12***
Use induction on all integers k such that $0 \leq k \leq n/2$ to prove inequality (C.7), and use equation (C.3) to extend it to all integers k such that $0 \leq k \leq n$.

★ ***C.1-13***
Use Stirling's approximation to prove that

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}(1 + O(1/n)) . \tag{C.11}$$

★ ***C.1-14***
By differentiating the entropy function $H(\lambda)$, show that it achieves its maximum value at $\lambda = 1/2$. What is $H(1/2)$?

★ ***C.1-15***
Show that for any integer $n \geq 0$,

$$\sum_{k=0}^{n} \binom{n}{k} k = n\, 2^{n-1} . \tag{C.12}$$

★ ***C.1-16***
Inequality (C.5) provides a lower bound on the binomial coefficient $\binom{n}{k}$. For small values of k, a stronger bound holds. Prove that

$$\binom{n}{k} \geq \frac{n^k}{4k!} \tag{C.13}$$

for $k \leq \sqrt{n}$.

C.2 Probability

Probability is an essential tool for the design and analysis of probabilistic and randomized algorithms. This section reviews basic probability theory.

We define probability in terms of a ***sample space*** S, which is a set whose elements are called ***outcomes*** or ***elementary events***. Think of each outcome as a possible result of an experiment. For the experiment of flipping two distinguishable coins, with each individual flip resulting in a head (H) or a tail (T), you can view the sample space S as consisting of the set of all possible 2-strings over $\{\text{H}, \text{T}\}$:

$$S = \{\text{HH}, \text{HT}, \text{TH}, \text{TT}\} .$$

An ***event*** is a subset[1] of the sample space S. For example, in the experiment of flipping two coins, the event of obtaining one head and one tail is $\{\text{HT}, \text{TH}\}$. The event S is called the ***certain event***, and the event $\emptyset$ is called the ***null event***. We say that two events A and B are ***mutually exclusive*** if $A \cap B = \emptyset$. An outcome s also defines the event $\{s\}$, which we sometimes write as just s. By definition, all outcomes are mutually exclusive.

Axioms of probability

A ***probability distribution*** $\Pr\{\}$ on a sample space S is a mapping from events of S to real numbers satisfying the following ***probability axioms***:

1. $\Pr\{A\} \geq 0$ for any event A.
2. $\Pr\{S\} = 1$.
3. $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}$ for any two mutually exclusive events A and B. More generally, for any sequence of events $A_1, A_2, \ldots$ (finite or countably infinite) that are pairwise mutually exclusive,

$$\Pr\left\{\bigcup_i A_i\right\} = \sum_i \Pr\{A_i\} .$$

We call $\Pr\{A\}$ the ***probability*** of the event A. Axiom 2 is simply a normalization requirement: there is really nothing fundamental about choosing 1 as the probability of the certain event, except that it is natural and convenient.

Several results follow immediately from these axioms and basic set theory (see Section B.1). The null event $\emptyset$ has probability $\Pr\{\emptyset\} = 0$. If $A \subseteq B$, then

[1] For a general probability distribution, there may be some subsets of the sample space S that are not considered to be events. This situation usually arises when the sample space is uncountably infinite. The main requirement for what subsets are events is that the set of events of a sample space must be closed under the operations of taking the complement of an event, forming the union of a finite or countable number of events, and taking the intersection of a finite or countable number of events. Most of the probability distributions we see in this book are over finite or countable sample spaces, and we generally consider all subsets of a sample space to be events. A notable exception is the continuous uniform probability distribution, which we'll see shortly.

$\Pr\{A\} \le \Pr\{B\}$. Using $\overline{A}$ to denote the event $S - A$ (the ***complement*** of A), we have $\Pr\left\{\overline{A}\right\} = 1 - \Pr\{A\}$. For any two events A and B,

$$\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\} - \Pr\{A \cap B\} \tag{C.14}$$

$$\le \Pr\{A\} + \Pr\{B\} \ . \tag{C.15}$$

In our coin-flipping example, suppose that each of the four outcomes has probability $1/4$. Then the probability of getting at least one head is

$$\begin{aligned}\Pr\{\text{HH}, \text{HT}, \text{TH}\} &= \Pr\{\text{HH}\} + \Pr\{\text{HT}\} + \Pr\{\text{TH}\} \\ &= 3/4 \ .\end{aligned}$$

Another way to obtain the same result is to observe that since the probability of getting strictly less than one head is $\Pr\{\text{TT}\} = 1/4$, the probability of getting at least one head is $1 - 1/4 = 3/4$.

Discrete probability distributions

A probability distribution is ***discrete*** if it is defined over a finite or countably infinite sample space. Let S be the sample space. Then for any event A,

$$\Pr\{A\} = \sum_{s \in A} \Pr\{s\} \ ,$$

since outcomes, specifically those in A, are mutually exclusive. If S is finite and every outcome $s \in S$ has probability $\Pr\{s\} = 1/|S|$, then we have the ***uniform probability distribution*** on S. In such a case the experiment is often described as "picking an element of S at random."

As an example, consider the process of flipping a ***fair coin***, one for which the probability of obtaining a head is the same as the probability of obtaining a tail, that is, $1/2$. Flipping the coin n times gives the uniform probability distribution defined on the sample space $S = \{\text{H}, \text{T}\}^n$, a set of size 2^n. We can represent each outcome in S as a string of length n over $\{\text{H}, \text{T}\}$, with each string occurring with probability $1/2^n$. The event $A = \{\text{exactly } k \text{ heads and exactly } n - k \text{ tails occur}\}$ is a subset of S of size $|A| = \binom{n}{k}$, since $\binom{n}{k}$ strings of length n over $\{\text{H}, \text{T}\}$ contain exactly k H's. The probability of event A is thus $\Pr\{A\} = \binom{n}{k}/2^n$.

Continuous uniform probability distribution

The continuous uniform probability distribution is an example of a probability distribution in which not all subsets of the sample space are considered to be events. The continuous uniform probability distribution is defined over a closed interval $[a, b]$ of the reals, where $a < b$. The intuition is that each point in the interval $[a, b]$ should be "equally likely." Because there are an uncountable number

of points, however, if all points had the same finite, positive probability, axioms 2 and 3 would not be simultaneously satisfied. For this reason, we'd like to associate a probability only with *some* of the subsets of S in such a way that the axioms are satisfied for these events.

For any closed interval $[c, d]$, where $a \le c \le d \le b$, the ***continuous uniform probability distribution*** defines the probability of the event $[c, d]$ to be

$$\Pr\{[c, d]\} = \frac{d - c}{b - a} .$$

Letting $c = d$ gives that the probability of a single point is 0. Removing the endpoints $[c, c]$ and $[d, d]$ of an interval $[c, d]$ results in the open interval (c, d). Since $[c, d] = [c, c] \cup (c, d) \cup [d, d]$, axiom 3 gives $\Pr\{[c, d]\} = \Pr\{(c, d)\}$. Generally, the set of events for the continuous uniform probability distribution contains any subset of the sample space $[a, b]$ that can be obtained by a finite or countable union of open and closed intervals, as well as certain more complicated sets.

Conditional probability and independence

Sometimes you have some prior partial knowledge about the outcome of an experiment. For example, suppose that a friend has flipped two fair coins and has told you that at least one of the coins showed a head. What is the probability that both coins are heads? The information given eliminates the possibility of two tails. The three remaining outcomes are equally likely, and so you infer that each occurs with probability $1/3$. Since only one of these outcomes shows two heads, the answer is $1/3$.

Conditional probability formalizes the notion of having prior partial knowledge of the outcome of an experiment. The ***conditional probability*** of an event A given that another event B occurs is defined to be

$$\Pr\{A \mid B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}} \tag{C.16}$$

whenever $\Pr\{B\} \ne 0$. (Read "$\Pr\{A \mid B\}$" as "the probability of A given B.") The idea behind equation (C.16) is that since we are given that event B occurs, the event that A also occurs is $A \cap B$. That is, $A \cap B$ is the set of outcomes in which both A and B occur. Because the outcome is one of the elementary events in B, we normalize the probabilities of all the elementary events in B by dividing them by $\Pr\{B\}$, so that they sum to 1. The conditional probability of A given B is, therefore, the ratio of the probability of event $A \cap B$ to the probability of event B. In the example above, A is the event that both coins are heads, and B is the event that at least one coin is a head. Thus, $\Pr\{A \mid B\} = (1/4)/(3/4) = 1/3$.

Two events are ***independent*** if

$$\Pr\{A \cap B\} = \Pr\{A\}\Pr\{B\} , \tag{C.17}$$

which is equivalent, if $\Pr\{B\} \neq 0$, to the condition

$$\Pr\{A \mid B\} = \Pr\{A\} .$$

For example, suppose that you flip two fair coins and that the outcomes are independent. Then the probability of two heads is $(1/2)(1/2) = 1/4$. Now suppose that one event is that the first coin comes up heads and the other event is that the coins come up differently. Each of these events occurs with probability $1/2$, and the probability that both events occur is $1/4$. Thus, according to the definition of independence, the events are independent—even though you might think that both events depend on the first coin. Finally, suppose that the coins are welded together so that they both fall heads or both fall tails and that the two possibilities are equally likely. Then the probability that each coin comes up heads is $1/2$, but the probability that they both come up heads is $1/2 \neq (1/2)(1/2)$. Consequently, the event that one comes up heads and the event that the other comes up heads are not independent.

A collection $A_1, A_2, \ldots, A_n$ of events is said to be ***pairwise independent*** if

$$\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\}$$

for all $1 \leq i < j \leq n$. We say that the events of the collection are ***(mutually) independent*** if every k-subset $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$ of the collection, where $2 \leq k \leq n$ and $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, satisfies

$$\Pr\{A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}\} = \Pr\{A_{i_1}\}\Pr\{A_{i_2}\}\cdots\Pr\{A_{i_k}\} .$$

For example, suppose that you flip two fair coins. Let A_1 be the event that the first coin is heads, let A_2 be the event that the second coin is heads, and let A_3 be the event that the two coins are different. Then,

$$\begin{aligned}
\Pr\{A_1\} &= 1/2 , \\
\Pr\{A_2\} &= 1/2 , \\
\Pr\{A_3\} &= 1/2 , \\
\Pr\{A_1 \cap A_2\} &= 1/4 , \\
\Pr\{A_1 \cap A_3\} &= 1/4 , \\
\Pr\{A_2 \cap A_3\} &= 1/4 , \\
\Pr\{A_1 \cap A_2 \cap A_3\} &= 0 .
\end{aligned}$$

Since for $1 \leq i < j \leq 3$, we have $\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\} = 1/4$, the events A_1, A_2, and A_3 are pairwise independent. The events are not mutually independent, however, because $\Pr\{A_1 \cap A_2 \cap A_3\} = 0$ and $\Pr\{A_1\}\Pr\{A_2\}\Pr\{A_3\} = 1/8 \neq 0$.

Bayes's theorem

From the definition (C.16) of conditional probability and the commutative law $A \cap B = B \cap A$, it follows that for two events A and B, each with nonzero probability,

$$\begin{aligned} \Pr\{A \cap B\} &= \Pr\{B\}\Pr\{A \mid B\} \\ &= \Pr\{A\}\Pr\{B \mid A\} \ . \end{aligned} \tag{C.18}$$

Solving for $\Pr\{A \mid B\}$, we obtain

$$\Pr\{A \mid B\} = \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{B\}} \ , \tag{C.19}$$

which is known as ***Bayes's theorem***. The denominator $\Pr\{B\}$ is a normalizing constant, which we can reformulate as follows. Since $B = (B \cap A) \cup (B \cap \overline{A})$, and since $B \cap A$ and $B \cap \overline{A}$ are mutually exclusive events,

$$\begin{aligned} \Pr\{B\} &= \Pr\{B \cap A\} + \Pr\left\{B \cap \overline{A}\right\} \\ &= \Pr\{A\}\Pr\{B \mid A\} + \Pr\left\{\overline{A}\right\}\Pr\left\{B \mid \overline{A}\right\} \ . \end{aligned}$$

Substituting into equation (C.19) produces an equivalent form of Bayes's theorem:

$$\Pr\{A \mid B\} = \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{A\}\Pr\{B \mid A\} + \Pr\left\{\overline{A}\right\}\Pr\left\{B \mid \overline{A}\right\}} \ . \tag{C.20}$$

Bayes's theorem can simplify the computing of conditional probabilities. For example, suppose that you have a fair coin and a biased coin that always comes up heads. Run an experiment consisting of three independent events: choose one of the two coins at random, flip that coin once, and then flip it again. Suppose that the coin you have chosen comes up heads both times. What is the probability that it's the biased coin?

Bayes's theorem solves this problem. Let A be the event that you choose the biased coin, and let B be the event that the chosen coin comes up heads both times. We wish to determine $\Pr\{A \mid B\}$, knowing that $\Pr\{A\} = 1/2$, $\Pr\{B \mid A\} = 1$, $\Pr\left\{\overline{A}\right\} = 1/2$, and $\Pr\left\{B \mid \overline{A}\right\} = 1/4$. Thus we have

$$\begin{aligned} \Pr\{A \mid B\} &= \frac{(1/2) \cdot 1}{(1/2) \cdot 1 + (1/2) \cdot (1/4)} \\ &= 4/5 \ . \end{aligned}$$

Exercises

C.2-1
Professor Rosencrantz flips a fair coin twice. Professor Guildenstern flips a fair coin once. What is the probability that Professor Rosencrantz obtains strictly more heads than Professor Guildenstern?

C.2-2
Prove ***Boole's inequality***: For any finite or countably infinite sequence of events $A_1, A_2, \ldots,$

$$\Pr\{A_1 \cup A_2 \cup \cdots\} \le \Pr\{A_1\} + \Pr\{A_2\} + \cdots . \tag{C.21}$$

C.2-3
You shuffle a deck of 10 cards, each bearing a distinct number from 1 to 10, in order to mix the cards thoroughly. You then remove three cards, one at a time, from the deck. What is the probability that the three cards you select are in sorted (increasing) order?

C.2-4
Prove that

$$\Pr\{A \mid B\} + \Pr\left\{\overline{A} \mid B\right\} = 1 .$$

C.2-5
Prove that for any collection of events $A_1, A_2, \ldots, A_n$,

$$\begin{aligned}\Pr\{A_1 \cap A_2 \cap \cdots \cap A_n\} = {} & \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ & \Pr\{A_n \mid A_1 \cap A_2 \cap \cdots \cap A_{n-1}\} .\end{aligned} \tag{C.22}$$

★ ***C.2-6***
Show how to construct a set of n events that are pairwise independent but such that no subset of $k > 2$ of them is mutually independent.

★ ***C.2-7***
Two events A and B are ***conditionally independent***, given C, if

$$\Pr\{A \cap B \mid C\} = \Pr\{A \mid C\} \cdot \Pr\{B \mid C\} .$$

Give a simple but nontrivial example of two events that are not independent but are conditionally independent given a third event.

★ ***C.2-8***
Professor Gore teaches a music class on rhythm in which three students—Jeff, Tim, and Carmine—are in danger of failing. Professor Gore tells the three that one of

them will pass the course and the other two will fail. Carmine asks Professor Gore privately which of Jeff and Tim will fail, arguing that since he already knows at least one of them will fail, the professor won't be revealing any information about Carmine's outcome. In a breach of privacy law, Professor Gore tells Carmine that Jeff will fail. Carmine feels somewhat relieved now, figuring that either he or Tim will pass, so that his probability of passing is now $1/2$. Is Carmine correct, or is his chance of passing still $1/3$? Explain.

C.3 Discrete random variables

A ***(discrete) random variable*** X is a function from a finite or countably infinite sample space S to the real numbers. It associates a real number with each possible outcome of an experiment, which allows us to work with the probability distribution induced on the resulting set of numbers. Random variables can also be defined for uncountably infinite sample spaces, but they raise technical issues that are unnecessary to address for our purposes. Therefore we'll assume that random variables are discrete.

For a random variable X and a real number x, we define the event $X = x$ to be $\{s \in S : X(s) = x\}$, and thus

$$\Pr\{X = x\} = \sum_{s \in S : X(s) = x} \Pr\{s\} \ .$$

The function

$$f(x) = \Pr\{X = x\}$$

is the ***probability density function*** of the random variable X. From the probability axioms, $\Pr\{X = x\} \geq 0$ and $\sum_x \Pr\{X = x\} = 1$.

As an example, consider the experiment of rolling a pair of ordinary, 6-sided dice. There are 36 possible outcomes in the sample space. Assume that the probability distribution is uniform, so that each outcome $s \in S$ is equally likely: $\Pr\{s\} = 1/36$. Define the random variable X to be the *maximum* of the two values showing on the dice. We have $\Pr\{X = 3\} = 5/36$, since X assigns a value of 3 to 5 of the 36 possible outcomes, namely, $(1, 3)$, $(2, 3)$, $(3, 3)$, $(3, 2)$, and $(3, 1)$.

We can define several random variables on the same sample space. If X and Y are random variables, the function

$$f(x, y) = \Pr\{X = x \text{ and } Y = y\}$$

is the ***joint probability density function*** of X and Y. For a fixed value y,

$$\Pr\{Y = y\} = \sum_x \Pr\{X = x \text{ and } Y = y\} ,$$

and similarly, for a fixed value x,

$$\Pr\{X = x\} = \sum_y \Pr\{X = x \text{ and } Y = y\} .$$

Using the definition (C.16) of conditional probability on page 1187, we have

$$\Pr\{X = x \mid Y = y\} = \frac{\Pr\{X = x \text{ and } Y = y\}}{\Pr\{Y = y\}} .$$

We define two random variables X and Y to be ***independent*** if for all x and y, the events $X = x$ and $Y = y$ are independent or, equivalently, if for all x and y, we have $\Pr\{X = x \text{ and } Y = y\} = \Pr\{X = x\}\Pr\{Y = y\}$.

Given a set of random variables defined over the same sample space, we can define new random variables as sums, products, or other functions of the original variables.

Expected value of a random variable

The simplest, and often the most useful, summary of the distribution of a random variable is the "average" of the values it takes on. The ***expected value*** (or, synonymously, ***expectation*** or ***mean***) of a discrete random variable X is

$$\mathrm{E}[X] = \sum_x x \cdot \Pr\{X = x\} , \tag{C.23}$$

which is well defined if the sum is finite or converges absolutely. Sometimes the expectation of X is denoted by μ_X or, when the random variable is apparent from context, simply by μ.

Consider a game in which you flip two fair coins. You earn \$3 for each head but lose \$2 for each tail. The expected value of the random variable X representing your earnings is

$$\begin{aligned}\mathrm{E}[X] &= 6 \cdot \Pr\{2 \text{ H's}\} + 1 \cdot \Pr\{1 \text{ H}, 1 \text{ T}\} - 4 \cdot \Pr\{2 \text{ T's}\} \\ &= 6 \cdot (1/4) + 1 \cdot (1/2) - 4 \cdot (1/4) \\ &= 1 .\end{aligned}$$

Linearity of expectation says that the expectation of the sum of two random variables is the sum of their expectations, that is,

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y] , \tag{C.24}$$

whenever $\mathrm{E}[X]$ and $\mathrm{E}[Y]$ are defined. Linearity of expectation applies to a broad range of situations, holding even when X and Y are not independent. It also extends to finite and absolutely convergent summations of expectations. Linearity of expectation is the key property that enables us to perform probabilistic analyses by using indicator random variables (see Section 5.2).

If X is any random variable, any function $g(x)$ defines a new random variable $g(X)$. If the expectation of $g(X)$ is defined, then

$$\mathrm{E}[g(X)] = \sum_x g(x) \cdot \Pr\{X = x\} .$$

Letting $g(x) = ax$, we have for any constant a,

$$\mathrm{E}[aX] = a\mathrm{E}[X] . \tag{C.25}$$

Consequently, expectations are linear: for any two random variables X and Y and any constant a,

$$\mathrm{E}[aX + Y] = a\mathrm{E}[X] + \mathrm{E}[Y] . \tag{C.26}$$

When two random variables X and Y are independent and each has a defined expectation,

$$\begin{aligned}
\mathrm{E}[XY] &= \sum_x \sum_y xy \cdot \Pr\{X = x \text{ and } Y = y\} \\
&= \sum_x \sum_y xy \cdot \Pr\{X = x\}\Pr\{Y = y\} \quad \text{(by independence of } X \text{ and } Y) \\
&= \left(\sum_x x \cdot \Pr\{X = x\}\right)\left(\sum_y y \cdot \Pr\{Y = y\}\right) \\
&= \mathrm{E}[X]\,\mathrm{E}[Y] \qquad\qquad \text{(by equation (C.23))} .
\end{aligned}$$

In general, when n random variables $X_1, X_2, \ldots, X_n$ are mutually independent,

$$\mathrm{E}[X_1 X_2 \cdots X_n] = \mathrm{E}[X_1]\,\mathrm{E}[X_2] \cdots \mathrm{E}[X_n] . \tag{C.27}$$

When a random variable X takes on values from the set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$, we have a nice formula for its expectation:

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=0}^{\infty} i \cdot \Pr\{X = i\} \\
&= \sum_{i=0}^{\infty} i \cdot (\Pr\{X \ge i\} - \Pr\{X \ge i + 1\}) \\
&= \sum_{i=1}^{\infty} \Pr\{X \ge i\} ,
\end{aligned} \tag{C.28}$$

since each term $\Pr\{X \geq i\}$ is added in i times and subtracted out $i - 1$ times (except $\Pr\{X \geq 0\}$, which is added in 0 times and not subtracted out at all).

A function $f(x)$ is ***convex*** if

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y) \tag{C.29}$$

for all x and y and for all $0 \leq \lambda \leq 1$. ***Jensen's inequality*** says that when a convex function $f(x)$ is applied to a random variable X,

$$\mathrm{E}[f(X)] \geq f(\mathrm{E}[X]) , \tag{C.30}$$

provided that the expectations exist and are finite.

Variance and standard deviation

The expected value of a random variable does not express how "spread out" the variable's values are. For example, consider random variables X and Y for which $\Pr\{X = 1/4\} = \Pr\{X = 3/4\} = 1/2$ and $\Pr\{Y = 0\} = \Pr\{Y = 1\} = 1/2$. Then both $\mathrm{E}[X]$ and $\mathrm{E}[Y]$ are $1/2$, yet the actual values taken on by Y are further from the mean than the actual values taken on by X.

The notion of variance mathematically expresses how far from the mean a random variable's values are likely to be. The ***variance*** of a random variable X with mean $\mathrm{E}[X]$ is

$$\begin{aligned}
\mathrm{Var}[X] &= \mathrm{E}\left[(X - \mathrm{E}[X])^2\right] \\
&= \mathrm{E}\left[X^2 - 2X\mathrm{E}[X] + \mathrm{E}^2[X]\right] \\
&= \mathrm{E}[X^2] - 2\mathrm{E}[X\mathrm{E}[X]] + \mathrm{E}^2[X] \\
&= \mathrm{E}[X^2] - 2\mathrm{E}^2[X] + \mathrm{E}^2[X] \\
&= \mathrm{E}[X^2] - \mathrm{E}^2[X] .
\end{aligned} \tag{C.31}$$

To justify the equation $\mathrm{E}[\mathrm{E}^2[X]] = \mathrm{E}^2[X]$, note that because $\mathrm{E}[X]$ is a real number and not a random variable, so is $\mathrm{E}^2[X]$. The equation $\mathrm{E}[X\mathrm{E}[X]] = \mathrm{E}^2[X]$ follows from equation (C.25), with $a = \mathrm{E}[X]$. Rewriting equation (C.31) yields an expression for the expectation of the square of a random variable:

$$\mathrm{E}[X^2] = \mathrm{Var}[X] + \mathrm{E}^2[X] . \tag{C.32}$$

The variance of a random variable X and the variance of aX are related (see Exercise C.3-10):

$$\mathrm{Var}[aX] = a^2\mathrm{Var}[X] .$$

When X and Y are independent random variables,

$$\mathrm{Var}[X + Y] = \mathrm{Var}[X] + \mathrm{Var}[Y] .$$

In general, if n random variables $X_1, X_2, \ldots, X_n$ are pairwise independent, then

$$\mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{Var}\,[X_i] \ . \tag{C.33}$$

The ***standard deviation*** of a random variable X is the nonnegative square root of the variance of X. The standard deviation of a random variable X is sometimes denoted σ_X or simply σ when the random variable X is understood from context. With this notation, the variance of X is denoted σ^2.

Exercises

C.3-1
You roll two ordinary, 6-sided dice. What is the expectation of the sum of the two values showing? What is the expectation of the maximum of the two values showing?

C.3-2
An array $A[1:n]$ contains n distinct numbers that are randomly ordered, with each permutation of the n numbers being equally likely. What is the expectation of the index of the maximum element in the array? What is the expectation of the index of the minimum element in the array?

C.3-3
A carnival game consists of three dice in a cage. A player can bet a dollar on any of the numbers 1 through 6. The cage is shaken, and the payoff is as follows. If the player's number doesn't appear on any of the dice, the player loses the dollar. Otherwise, if the player's number appears on exactly k of the three dice, for $k = 1, 2, 3$, the player keeps the dollar and wins k more dollars. What is the expected gain from playing the carnival game once?

C.3-4
Argue that if X and Y are nonnegative random variables, then

$$\mathrm{E}\,[\max\{X, Y\}] \le \mathrm{E}\,[X] + \mathrm{E}\,[Y] \ .$$

★ ***C.3-5***
Let X and Y be independent random variables. Prove that $f(X)$ and $g(Y)$ are independent for any choice of functions f and g.

★ ***C.3-6***
Let X be a nonnegative random variable, and suppose that $\mathrm{E}[X]$ is well defined. Prove ***Markov's inequality***:

$$\Pr\{X \geq t\} \leq \mathrm{E}[X]/t \tag{C.34}$$

for all $t > 0$.

★ ***C.3-7***
Let S be a sample space, and let X and X' be random variables such that $X(s) \geq X'(s)$ for all $s \in S$. Prove that for any real constant t,

$$\Pr\{X \geq t\} \geq \Pr\{X' \geq t\} \ .$$

C.3-8
Which is larger: the expectation of the square of a random variable, or the square of its expectation?

C.3-9
Show that for any random variable X that takes on only the values 0 and 1, we have $\mathrm{Var}[X] = \mathrm{E}[X]\,\mathrm{E}[1 - X]$.

C.3-10
Prove that $\mathrm{Var}[aX] = a^2\mathrm{Var}[X]$ from the definition (C.31) of variance.

C.4 The geometric and binomial distributions

A ***Bernoulli trial*** is an experiment with only two possible outcomes: ***success***, which occurs with probability p, and ***failure***, which occurs with probability $q = 1 - p$. A coin flip serves as an example where, depending on your point of view, heads equates to success and tails to failure. When we speak of ***Bernoulli trials*** collectively, we mean that the trials are mutually independent and, unless we specifically say otherwise, that each has the same probability p for success. Two important distributions arise from Bernoulli trials: the geometric distribution and the binomial distribution.

The geometric distribution

Consider a sequence of Bernoulli trials, each with a probability p of success and a probability $q = 1 - p$ of failure. How many trials occur before a success? Define the random variable X to be the number of trials needed to obtain a success. Then X has values in the range $\{1, 2, \ldots\}$, and for $k \geq 1$,

Figure C.1 A geometric distribution with probability $p = 1/3$ of success and a probability $q = 1 - p$ of failure. The expectation of the distribution is $1/p = 3$.

$$\Pr\{X = k\} = q^{k-1} p \,, \tag{C.35}$$

since $k - 1$ failures occur before the first success. A probability distribution satisfying equation (C.35) is said to be a ***geometric distribution***. Figure C.1 illustrates such a distribution.

Assuming that $q < 1$, we can calculate the expectation of a geometric distribution:

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{k=1}^{\infty} k q^{k-1} p \\
&= \frac{p}{q} \sum_{k=0}^{\infty} k q^{k} \\
&= \frac{p}{q} \cdot \frac{q}{(1-q)^2} \quad \text{(by equation (A.11) on page 1142)} \\
&= \frac{p}{q} \cdot \frac{q}{p^2} \\
&= 1/p. \qquad \text{(C.36)}
\end{aligned}$$

Thus, on average, it takes $1/p$ trials before a success occurs, an intuitive result. As Exercise C.4-3 asks you to show, the variance is

$$\mathrm{Var}[X] = q/p^2 . \tag{C.37}$$

As an example, suppose that you repeatedly roll two dice until you obtain either a seven or an eleven. Of the 36 possible outcomes, 6 yield a seven and 2 yield an eleven. Thus, the probability of success is $p = 8/36 = 2/9$, and you'd have to roll $1/p = 9/2 = 4.5$ times on average to obtain a seven or eleven.

The binomial distribution

How many successes occur during n Bernoulli trials, where a success occurs with probability p and a failure with probability $q = 1 - p$? Define the random variable X to be the number of successes in n trials. Then X has values in the range $\{0, 1, \ldots, n\}$, and for $k = 0, 1, \ldots, n$,

$$\Pr\{X = k\} = \binom{n}{k} p^k q^{n-k} , \tag{C.38}$$

since there are $\binom{n}{k}$ ways to pick which k of the n trials are successes, and the probability that each occurs is $p^k q^{n-k}$. A probability distribution satisfying equation (C.38) is said to be a ***binomial distribution***. For convenience, we define the family of binomial distributions using the notation

$$b(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k} . \tag{C.39}$$

Figure C.2 illustrates a binomial distribution. The name "binomial" comes from the right-hand side of equation (C.38) being the kth term of the expansion of $(p+q)^n$. Consequently, since $p + q = 1$, equation (C.4) on page 1181 gives

$$\sum_{k=0}^{n} b(k; n, p) = 1 , \tag{C.40}$$

as axiom 2 of the probability axioms requires.

We can compute the expectation of a random variable having a binomial distribution from equations (C.9) and (C.40). Let X be a random variable that follows the binomial distribution $b(k; n, p)$, and let $q = 1 - p$. The definition of expectation gives

Figure C.2 The binomial distribution $b(k; 15, 1/3)$ resulting from $n = 15$ Bernoulli trials, each with probability $p = 1/3$ of success. The expectation of the distribution is $np = 5$.

$$
\begin{aligned}
\mathrm{E}[X] &= \sum_{k=0}^{n} k \cdot \Pr\{X = k\} \\
&= \sum_{k=0}^{n} k \cdot b(k; n, p) \\
&= \sum_{k=1}^{n} k \binom{n}{k} p^k q^{n-k} \\
&= np \sum_{k=1}^{n} \binom{n-1}{k-1} p^{k-1} q^{n-k} && \text{(by equation (C.9) on page 1183)} \\
&= np \sum_{k=0}^{n-1} \binom{n-1}{k} p^k q^{(n-1)-k} \\
&= np \sum_{k=0}^{n-1} b(k; n-1, p) \\
&= np && \text{(by equation (C.40))} . \qquad \text{(C.41)}
\end{aligned}
$$

Linearity of expectation produces the same result with substantially less algebra. Let X_i be the random variable describing the number of successes in the ith trial. Then $\mathrm{E}[X_i] = p \cdot 1 + q \cdot 0 = p$, and the expected number of successes for n trials is

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \qquad \text{(by equation (C.24) on page 1192)} \\
&= \sum_{i=1}^{n} p \\
&= np. \qquad \text{(C.42)}
\end{aligned}$$

We can use the same approach to calculate the variance of the distribution. By equation (C.31), $\mathrm{Var}[X_i] = \mathrm{E}[X_i^2] - \mathrm{E}^2[X_i]$. Since X_i takes on only the values 0 and 1, we have $X_i^2 = X_i$, which implies $\mathrm{E}[X_i^2] = \mathrm{E}[X_i] = p$. Hence,

$$\mathrm{Var}[X_i] = p - p^2 = p(1-p) = pq\ . \qquad \text{(C.43)}$$

To compute the variance of X, we take advantage of the independence of the n trials. By equation (C.33), we have

$$\begin{aligned}
\mathrm{Var}[X] &= \mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{Var}[X_i] \\
&= \sum_{i=1}^{n} pq \\
&= npq\ . \qquad \text{(C.44)}
\end{aligned}$$

As Figure C.2 shows, the binomial distribution $b(k; n, p)$ increases with k until it reaches the mean np, and then it decreases. To prove that the distribution always behaves in this manner, examine the ratio of successive terms:

$$\begin{aligned}
\frac{b(k; n, p)}{b(k-1; n, p)} &= \frac{\binom{n}{k} p^k q^{n-k}}{\binom{n}{k-1} p^{k-1} q^{n-k+1}} \\
&= \frac{n!\,(k-1)!\,(n-k+1)!\,p}{k!\,(n-k)!\,n!\,q} \\
&= \frac{(n-k+1)p}{kq} \qquad \text{(C.45)} \\
&= 1 + \frac{(n-k+1)p - kq}{kq} \\
&= 1 + \frac{(n-k+1)p - k(1-p)}{kq}
\end{aligned}$$

$$= 1 + \frac{(n+1)p - k}{kq} .$$

This ratio is greater than 1 precisely when $(n+1)p - k$ is positive. Consequently, $b(k; n, p) > b(k-1; n, p)$ for $k < (n+1)p$ (the distribution increases), and $b(k; n, p) < b(k-1; n, p)$ for $k > (n+1)p$ (the distribution decreases). If $(n+1)p$ is an integer, then for $k = (n+1)p$, the ratio $b(k; n, p)/b(k-1; n, p)$ equals 1, so that $b(k; n, p) = b(k-1; n, p)$. In this case, the distribution has two maxima: at $k = (n+1)p$ and at $k-1 = (n+1)p-1 = np-q$. Otherwise, it attains a maximum at the unique integer k that lies in the range $np - q < k < (n+1)p$.

The following lemma provides an upper bound on the binomial distribution.

Lemma C.1
Let $n \geq 0$, let $0 < p < 1$, let $q = 1 - p$, and let $0 \leq k \leq n$. Then

$$b(k; n, p) \leq \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k} .$$

Proof We have

$$\begin{aligned} b(k; n, p) &= \binom{n}{k} p^k q^{n-k} \\ &\leq \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k} \quad \text{(by inequality (C.7) on page 1182)} \\ &= \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k} . \end{aligned}$$

■

Exercises

C.4-1
Verify axiom 2 of the probability axioms for the geometric distribution.

C.4-2
How many times on average do you need to flip six fair coins before obtaining three heads and three tails?

C.4-3
Show that the variance of the geometric distribution is q/p^2. (*Hint:* Use Exercise A.1-6 on page 1144.)

C.4-4
Show that $b(k; n, p) = b(n-k; n, q)$, where $q = 1 - p$.

C.4-5

Show that the value of the maximum of the binomial distribution $b(k; n, p)$ is approximately $1/\sqrt{2\pi npq}$, where $q = 1 - p$.

★ ***C.4-6***

Show that the probability of no successes in n Bernoulli trials, each with probability $p = 1/n$ of success, is approximately $1/e$. Show that the probability of exactly one success is also approximately $1/e$.

★ ***C.4-7***

Professor Rosencrantz flips a fair coin n times, and so does Professor Guildenstern. Show that the probability that they get the same number of heads is $\binom{2n}{n}/4^n$. (*Hint:* For Professor Rosencrantz, call a head a success, and for Professor Guildenstern, call a tail a success.) Use your argument to verify the identity

$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n} .$$

★ ***C.4-8***

Show that for $0 \le k \le n$,

$$b(k; n, 1/2) \le 2^{n\,H(k/n)-n} ,$$

where $H(x)$ is the entropy function (C.8) on page 1182.

★ ***C.4-9***

Consider n Bernoulli trials, where for $i = 1, 2, \ldots, n$, the ith trial has probability p_i of success, and let X be the random variable denoting the total number of successes. Let $p \ge p_i$ for all $i = 1, 2, \ldots, n$. Prove that for $1 \le k \le n$,

$$\Pr\{X < k\} \ge \sum_{i=0}^{k-1} b(i; n, p) .$$

★ ***C.4-10***

Let X be the random variable for the total number of successes in a set A of n Bernoulli trials, where the ith trial has a probability p_i of success, and let X' be the random variable for the total number of successes in a second set A' of n Bernoulli trials, where the ith trial has a probability $p'_i \ge p_i$ of success. Prove that for $0 \le k \le n$,

$$\Pr\{X' \ge k\} \ge \Pr\{X \ge k\} .$$

(*Hint:* Show how to obtain the Bernoulli trials in A' by an experiment involving the trials of A, and use the result of Exercise C.3-7.)

★ C.5 The tails of the binomial distribution

The probability of having at least, or at most, k successes in n Bernoulli trials, each with probability p of success, is often of more interest than the probability of having exactly k successes. In this section, we investigate the ***tails*** of the binomial distribution: the two regions of the distribution $b(k; n, p)$ that are far from the mean np. We'll prove several important bounds on (the sum of all terms in) a tail.

We first provide a bound on the right tail of the distribution $b(k; n, p)$. To determine bounds on the left tail, simply invert the roles of successes and failures.

Theorem C.2
Consider a sequence of n Bernoulli trials, where success occurs with probability p. Let X be the random variable denoting the total number of successes. Then for $0 \le k \le n$, the probability of at least k successes is

$$\begin{aligned}\Pr\{X \ge k\} &= \sum_{i=k}^{n} b(i; n, p) \\ &\le \binom{n}{k} p^k .\end{aligned}$$

Proof For $S \subseteq \{1, 2, \ldots, n\}$, let A_S denote the event that the ith trial is a success for every $i \in S$. Since $\Pr\{A_S\} = p^k$, where $|S| = k$, we have

$$\begin{aligned}\Pr\{X \ge k\} &= \Pr\{\text{there exists } S \subseteq \{1, 2, \ldots, n\} : |S| = k \text{ and } A_S\} \\ &= \Pr\left\{\bigcup_{S \subseteq \{1,2,\ldots,n\} : |S| = k} A_S\right\} \\ &\le \sum_{S \subseteq \{1,2,\ldots,n\} : |S| = k} \Pr\{A_S\} \qquad \text{(by inequality (C.21) on page 1190)} \\ &= \binom{n}{k} p^k .\end{aligned}$$

■

The following corollary restates the theorem for the left tail of the binomial distribution. In general, we'll leave it to you to adapt the proofs from one tail to the other.

Corollary C.3
Consider a sequence of n Bernoulli trials, where success occurs with probability p. If X is the random variable denoting the total number of successes, then for $0 \le k \le n$, the probability of at most k successes is

$$\begin{aligned}
\Pr\{X \le k\} &= \sum_{i=0}^{k} b(i; n, p) \\
&\le \binom{n}{n-k}(1-p)^{n-k} \\
&= \binom{n}{k}(1-p)^{n-k} .
\end{aligned}$$

■

Our next bound concerns the left tail of the binomial distribution. Its corollary shows that, far from the mean, the left tail diminishes exponentially.

Theorem C.4
Consider a sequence of n Bernoulli trials, where success occurs with probability p and failure with probability $q = 1 - p$. Let X be the random variable denoting the total number of successes. Then for $0 < k < np$, the probability of fewer than k successes is

$$\begin{aligned}
\Pr\{X < k\} &= \sum_{i=0}^{k-1} b(i; n, p) \\
&< \frac{kq}{np-k}\, b(k; n, p) .
\end{aligned}$$

Proof We bound the series $\sum_{i=0}^{k-1} b(i; n, p)$ by a geometric series using the technique from Section A.2, page 1147. For $i = 1, 2, \ldots, k$, equation (C.45) gives

$$\begin{aligned}
\frac{b(i-1; n, p)}{b(i; n, p)} &= \frac{iq}{(n-i+1)p} \\
&< \frac{iq}{(n-i)p} \\
&\le \frac{kq}{(n-k)p} .
\end{aligned}$$

If we let

$$\begin{aligned}
x &= \frac{kq}{(n-k)p} \\
&< \frac{kq}{(n-np)p} \\
&= \frac{kq}{nqp}
\end{aligned}$$

$$= \frac{k}{np}$$
$$< 1 ,$$

it follows that

$$b(i-1;n,p) < x\, b(i;n,p)$$

for $0 < i \le k$. Iteratively applying this inequality $k - i$ times gives

$$b(i;n,p) < x^{k-i}\, b(k;n,p)$$

for $0 \le i < k$, and hence

$$\begin{aligned}
\sum_{i=0}^{k-1} b(i;n,p) &< \sum_{i=0}^{k-1} x^{k-i} b(k;n,p) \\
&< b(k\ ;n,p) \sum_{i=1}^{\infty} x^i \\
&= \frac{x}{1-x}\, b(k;n,p) \\
&= \frac{kq/((n-k)p)}{((n-k)p-kq)/((n-k)p)}\, b(k;n,p) \\
&= \frac{kq}{np-kp-kq}\, b(k;n,p) \\
&= \frac{kq}{np-k}\, b(k;n,p) .
\end{aligned}$$

■

Corollary C.5
Consider a sequence of n Bernoulli trials, where success occurs with probability p and failure with probability $q = 1 - p$. Then for $0 < k \le np/2$, the probability of fewer than k successes is less than half the probability of fewer than $k + 1$ successes.

Proof Because $k \le np/2$, we have

$$\begin{aligned}
\frac{kq}{np-k} &\le \frac{(np/2)q}{np-(np/2)} \\
&= \frac{(np/2)q}{np/2} \\
&\le 1 ,
\end{aligned} \tag{C.46}$$

since $q \le 1$. Letting X be the random variable denoting the number of successes, Theorem C.4 and inequality (C.46) imply that the probability of fewer than k successes is

$$\Pr\{X < k\} = \sum_{i=0}^{k-1} b(i; n, p) < b(k; n, p) .$$

Thus we have

$$\begin{aligned}\frac{\Pr\{X < k\}}{\Pr\{X < k+1\}} &= \frac{\sum_{i=0}^{k-1} b(i; n, p)}{\sum_{i=0}^{k} b(i; n, p)} \\ &= \frac{\sum_{i=0}^{k-1} b(i; n, p)}{\sum_{i=0}^{k-1} b(i; n, p) + b(k; n, p)} \\ &< 1/2,\end{aligned}$$

since $\sum_{i=0}^{k-1} b(i; n, p) < b(k; n, p)$. ■

Bounds on the right tail follow similarly. Exercise C.5-2 asks you to prove them.

Corollary C.6

Consider a sequence of n Bernoulli trials, where success occurs with probability p. Let X be the random variable denoting the total number of successes. Then for $np < k < n$, the probability of more than k successes is

$$\begin{aligned}\Pr\{X > k\} &= \sum_{i=k+1}^{n} b(i; n, p) \\ &< \frac{(n-k)p}{k-np} b(k; n, p) .\end{aligned}$$ ■

Corollary C.7

Consider a sequence of n Bernoulli trials, where success occurs with probability p and failure with probability $q = 1 - p$. Then for $(np + n)/2 < k < n$, the probability of more than k successes is less than half the probability of more than $k - 1$ successes. ■

The next theorem considers n Bernoulli trials, each with a probability p_i of success, for $i = 1, 2, \ldots, n$. As the subsequent corollary shows, we can use the theorem to provide a bound on the right tail of the binomial distribution by setting $p_i = p$ for each trial.

Theorem C.8

Consider a sequence of n Bernoulli trials, where in the ith trial, for $i = 1, 2, \ldots, n$, success occurs with probability p_i and failure occurs with probability $q_i = 1 - p_i$. Let X be the random variable describing the total number of successes, and let $\mu = \mathrm{E}[X]$. Then for $r > \mu$,

$$\Pr\{X - \mu \geq r\} \leq \left(\frac{\mu e}{r}\right)^r .$$

Proof Since for any $\alpha > 0$, the function $e^{\alpha x}$ strictly increases in x,

$$\Pr\{X - \mu \geq r\} = \Pr\left\{e^{\alpha(X-\mu)} \geq e^{\alpha r}\right\} , \tag{C.47}$$

where we will determine α later. Using Markov's inequality (C.34), we obtain

$$\Pr\left\{e^{\alpha(X-\mu)} \geq e^{\alpha r}\right\} \leq \mathrm{E}\left[e^{\alpha(X-\mu)}\right] e^{-\alpha r} . \tag{C.48}$$

The bulk of the proof consists of bounding $\mathrm{E}\left[e^{\alpha(X-\mu)}\right]$ and substituting a suitable value for α in inequality (C.48). First, we evaluate $\mathrm{E}\left[e^{\alpha(X-\mu)}\right]$. Using the technique of indicator random variables (see Section 5.2), let $X_i = \mathrm{I}\{\text{the } i\text{th Bernoulli trial is a success}\}$ for $i = 1, 2, \ldots, n$. That is, X_i is the random variable that is 1 if the ith Bernoulli trial is a success and 0 if it is a failure. Thus, we have

$$X = \sum_{i=1}^{n} X_i ,$$

and by linearity of expectation,

$$\mu = \mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} p_i ,$$

which implies

$$X - \mu = \sum_{i=1}^{n} (X_i - p_i) .$$

To evaluate $\mathrm{E}\left[e^{\alpha(X-\mu)}\right]$, we substitute for $X - \mu$, obtaining

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha(X-\mu)}\right] &= \mathrm{E}\left[e^{\alpha \sum_{i=1}^{n}(X_i - p_i)}\right] \\
&= \mathrm{E}\left[\prod_{i=1}^{n} e^{\alpha(X_i - p_i)}\right] \\
&= \prod_{i=1}^{n} \mathrm{E}\left[e^{\alpha(X_i - p_i)}\right] ,
\end{aligned}$$

which follows from equation (C.27), since the mutual independence of the random variables X_i implies the mutual independence of the random variables $e^{\alpha(X_i - p_i)}$ (see Exercise C.3-5). By the definition of expectation,

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha(X_i - p_i)}\right] &= e^{\alpha(1-p_i)} p_i + e^{\alpha(0-p_i)} q_i \\
&= p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \\
&\le p_i e^{\alpha} + 1 \\
&\le \exp(p_i e^{\alpha}) ,
\end{aligned} \tag{C.49}$$

where $\exp(x)$ denotes the exponential function: $\exp(x) = e^x$. (Inequality (C.49) follows from the inequalities $\alpha > 0$, $q_i \le 1$, $e^{\alpha q_i} \le e^{\alpha}$, and $e^{-\alpha p_i} \le 1$. The last line follows from inequality (3.14) on page 66.) Consequently,

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha(X-\mu)}\right] &= \prod_{i=1}^{n} \mathrm{E}\left[e^{\alpha(X_i - p_i)}\right] \\
&\le \prod_{i=1}^{n} \exp(p_i e^{\alpha}) \\
&= \exp\left(\sum_{i=1}^{n} p_i e^{\alpha}\right) \\
&= \exp(\mu e^{\alpha}) ,
\end{aligned} \tag{C.50}$$

since $\mu = \sum_{i=1}^{n} p_i$. Therefore, from equation (C.47) and inequalities (C.48) and (C.50), it follows that

$$\Pr\{X - \mu \ge r\} \le \exp(\mu e^{\alpha} - \alpha r) . \tag{C.51}$$

Choosing $\alpha = \ln(r/\mu)$ (see Exercise C.5-7), we obtain

$$\begin{aligned}
\Pr\{X - \mu \ge r\} &\le \exp(\mu e^{\ln(r/\mu)} - r \ln(r/\mu)) \\
&= \exp(r - r \ln(r/\mu)) \\
&= \frac{e^r}{(r/\mu)^r} \\
&= \left(\frac{\mu e}{r}\right)^r .
\end{aligned}$$

■

When applied to Bernoulli trials in which each trial has the same probability of success, Theorem C.8 yields the following corollary bounding the right tail of a binomial distribution.

Corollary C.9

Consider a sequence of n Bernoulli trials, where in each trial success occurs with probability p and failure occurs with probability $q = 1 - p$. Then for $r > np$,

$$\begin{aligned}\Pr\{X - np \geq r\} &= \sum_{k=\lceil np+r \rceil}^{n} b(k; n, p) \\ &\leq \left(\frac{npe}{r}\right)^r .\end{aligned}$$

Proof By equation (C.41), we have $\mu = \mathrm{E}[X] = np$. ■

Exercises

★ ***C.5-1***
Which is more likely: getting exactly n heads in $2n$ flips of a fair coin, or n heads in n flips of a fair coin?

★ ***C.5-2***
Prove Corollaries C.6 and C.7.

★ ***C.5-3***
Show that

$$\sum_{i=0}^{k-1} \binom{n}{i} a^i < (a+1)^n \frac{k}{na - k(a+1)} b(k; n, a/(a+1))$$

for all $a > 0$ and all k such that $0 < k < na/(a+1)$.

★ ***C.5-4***
Prove that if $0 < k < np$, where $0 < p < 1$ and $q = 1 - p$, then

$$\sum_{i=0}^{k-1} p^i q^{n-i} < \frac{kq}{np - k} \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k} .$$

★ ***C.5-5***
Use Theorem C.8 to show that

$$\Pr\{\mu - X \geq r\} \leq \left(\frac{(n-\mu)e}{r}\right)^r$$

for $r > n - \mu$. Similarly, use Corollary C.9 to show that

$$\Pr\{np - X \geq r\} \leq \left(\frac{nqe}{r}\right)^r$$

for $r > n - np$.

★ ***C.5-6***
Consider a sequence of n Bernoulli trials, where in the ith trial, for $i = 1, 2, \ldots, n$, success occurs with probability p_i and failure occurs with probability $q_i = 1 - p_i$. Let X be the random variable describing the total number of successes, and let $\mu = \mathrm{E}[X]$. Show that for $r \geq 0$,

$$\Pr\{X - \mu \geq r\} \leq e^{-r^2/2n} .$$

(*Hint:* Prove that $p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leq e^{\alpha^2/2}$. Then follow the outline of the proof of Theorem C.8, using this inequality in place of inequality (C.49).)

★ ***C.5-7***
Show that choosing $\alpha = \ln(r/\mu)$ minimizes the right-hand side of inequality (C.51).

Problems

C-1 The Monty Hall problem
Imagine that you are a contestant in the 1960s game show *Let's Make a Deal*, hosted by emcee Monty Hall. A valuable prize is hidden behind one of three doors and comparatively worthless prizes behind the other two doors. You will win the valuable prize, typically an automobile or other expensive product, if you select the correct door. After you have picked one door, but before the door has been opened, Monty, who knows which door hides the automobile, directs his assistant Carol Merrill to open one of the other doors, revealing a goat (not a valuable prize). He asks whether you would like to stick with your current choice or to switch to the other closed door. What should you do to maximize your chances of winning the automobile and not the other goat?

The answer to this question—stick or switch?—has been heavily debated, in part because the problem setup is ambiguous. We'll explore different subtle assumptions.

a. Suppose that your first pick is random, with probability $1/3$ of choosing the right door. Moreover, you know that Monty always gives every contestant (and will give you) the opportunity to switch. Prove that it is better to switch than stick. What is your probability of winning the automobile?

This answer is the one typically given, even though the original statement of the problem rarely mentions the assumption that Monty *always* offers the contestant the opportunity to switch. But, as the remainder of this problem will elucidate, your best strategy may be different if this unstated assumption does not hold. In

fact, in the real game show, after a contestant picked a door, Monty sometimes simply asked Carol to open the door that the contestant had chosen.

Let's model the interactions between you and Monty as a probabilistic experiment, where you both employ randomized strategies. Specifically, after you pick a door, Monty offers you the opportunity to switch with probability p_{right} if you picked the right door and with probability p_{wrong} if you picked the wrong door. Given the opportunity to switch, you randomly choose to switch with probability p_{switch}. For example, if Monty always offers you the opportunity to switch, then his strategy is given by $p_{\text{right}} = p_{\text{wrong}} = 1$. If you always switch, then your strategy is given by $p_{\text{switch}} = 1$.

The game can now be viewed as an experiment consisting of five steps:

1. You pick a door at random, choosing the automobile (right) with probability $1/3$ or a goat (wrong) with probability $2/3$.
2. Carol opens one of the two closed doors, revealing a goat.
3. Monty offers you the opportunity to switch with probability p_{right} if your choice is right and with probability p_{wrong} if your choice is wrong.
4. If Monty makes you an offer in step 3, you switch with probability p_{switch}.
5. Carol opens the door you've chosen, revealing either an automobile (you win) or a goat (you lose).

Let's now analyze this game and understand how the choices of p_{right}, p_{wrong}, and p_{switch} influence the probability of winning.

b. What are the six outcomes in the sample space for this game? Which outcomes correspond to you winning the automobile? What are the probabilities in terms of p_{right}, p_{wrong}, and p_{switch} of each outcome? Organize your answers into a table.

c. Use the results of your table (or other means) to prove that the probability of winning the automobile is

$$\frac{1}{3}(2p_{\text{wrong}}p_{\text{switch}} - p_{\text{right}}p_{\text{switch}} + 1) .$$

Suppose that Monty knows the probability p_{switch} that you switch, and his goal is to minimize your chance of winning.

d. If $p_{\text{switch}} > 0$ (you switch with a positive probability), what is Monty's best strategy, that is, his best choice for p_{right} and p_{wrong}?

e. If $p_{\text{switch}} = 0$ (you always stick), argue that all of Monty's possible strategies are optimal for him.

Suppose that now Monty's strategy is fixed, with particular values for p_{right} and p_{wrong}.

f. If you know p_{right} and p_{wrong}, what is your best strategy for choosing your probability p_{switch} of switching as a function of p_{right} and p_{wrong}?

g. If you don't know p_{right} and p_{wrong}, what choice of p_{switch} maximizes the minimum probability of winning over all the choices of p_{right} and p_{wrong}?

Let's return to the original problem as stated, where Monty has given you the option of switching, but you have no knowledge of Monty's possible motivations or strategies.

h. Argue that the conditional probability of winning the automobile given that Monty offers you the opportunity to switch is

$$\frac{p_{\text{right}} - p_{\text{right}}\, p_{\text{switch}} + 2p_{\text{wrong}}\, p_{\text{switch}}}{p_{\text{right}} + 2p_{\text{wrong}}} . \tag{C.52}$$

Explain why $p_{\text{right}} + 2p_{\text{wrong}} \neq 0$.

i. What is the value of expression (C.52) when $p_{\text{switch}} = 1/2$? Show that choosing $p_{\text{switch}} < 1/2$ or $p_{\text{switch}} > 1/2$ allows Monty to select values for p_{right} and p_{wrong} that yield a lower value for expression (C.52) than choosing $p_{\text{switch}} = 1/2$.

j. Suppose that you don't know Monty's strategy. Explain why choosing to switch with probability $1/2$ is a good strategy for the original problem as stated. Summarize what you have learned overall from this problem.

C-2 Balls and bins

This problem investigates the effect of various assumptions on the number of ways of placing n balls into b distinct bins.

a. Suppose that the n balls are distinct and that their order within a bin does not matter. Argue that the number of ways of placing the balls in the bins is b^n.

b. Suppose that the balls are distinct and that the balls in each bin are ordered. Prove that there are exactly $(b + n - 1)!/(b - 1)!$ ways to place the balls in the bins. (*Hint:* Consider the number of ways of arranging n distinct balls and $b - 1$ indistinguishable sticks in a row.)

c. Suppose that the balls are identical, and hence their order within a bin does not matter. Show that the number of ways of placing the balls in the bins is $\binom{b+n-1}{n}$. (*Hint:* Of the arrangements in part (b), how many are repeated if the balls are made identical?)

d. Suppose that the balls are identical and that no bin may contain more than one ball, so that $n \leq b$. Show that the number of ways of placing the balls is $\binom{b}{n}$.

e. Suppose that the balls are identical and that no bin may be left empty. Assuming that $n \geq b$, show that the number of ways of placing the balls is $\binom{n-1}{b-1}$.

Appendix notes

The first general methods for solving probability problems were discussed in a famous correspondence between B. Pascal and P. de Fermat, which began in 1654, and in a book by C. Huygens in 1657. Rigorous probability theory began with the work of J. Bernoulli in 1713 and A. De Moivre in 1730. Further developments of the theory were provided by P.-S. Laplace, S.-D. Poisson, and C. F. Gauss.

Sums of random variables were originally studied by P. L. Chebyshev and A. A. Markov. A. N. Kolmogorov axiomatized probability theory in 1933. Chernoff [91] and Hoeffding [222] provided bounds on the tails of distributions. Seminal work in random combinatorial structures was done by P. Erdős.

Knuth [259] and Liu [302] are good references for elementary combinatorics and counting. Standard textbooks such as Billingsley [56], Chung [93], Drake [125], Feller [139], and Rozanov [390] offer comprehensive introductions to probability.

D Matrices

Matrices arise in numerous applications, including, but by no means limited to, scientific computing. If you have seen matrices before, much of the material in this appendix will be familiar to you, but some of it might be new. Section D.1 covers basic matrix definitions and operations, and Section D.2 presents some basic matrix properties.

D.1 Matrices and matrix operations

This section reviews some basic concepts of matrix theory and some fundamental properties of matrices.

Matrices and vectors

A ***matrix*** is a rectangular array of numbers. For example,

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \end{aligned} \tag{D.1}$$

is a 2×3 matrix $A = (a_{ij})$, where for $i = 1, 2$ and $j = 1, 2, 3$, the element of the matrix in row i and column j is denoted by a_{ij}. By convention, uppercase letters denote matrices and corresponding subscripted lowercase letters denote their elements. We denote the set of all $m \times n$ matrices with real-valued entries by $\mathbb{R}^{m \times n}$ and, in general, the set of $m \times n$ matrices with entries drawn from a set S by $S^{m \times n}$.

The ***transpose*** of a matrix A is the matrix A^{T} obtained by exchanging the rows and columns of A. For the matrix A of equation (D.1),

$$A^{\mathrm{T}} = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} .$$

A ***vector*** is a one-dimensional array of numbers. For example,

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix}$$

is a vector of size 3. We sometimes call a vector of length n an ***n-vector***. By convention, lowercase letters denote vectors, and the ith element of a size-n vector x is denoted by x_i, for $i = 1, 2, \ldots, n$. We take the standard form of a vector to be as a ***column vector*** equivalent to an $n \times 1$ matrix, whereas the corresponding ***row vector*** is obtained by taking the transpose:

$$x^{\mathrm{T}} = \begin{pmatrix} 2 & 3 & 5 \end{pmatrix} .$$

The ***unit vector*** e_i is the vector whose ith element is 1 and all of whose other elements are 0. Usually, the context makes the size of a unit vector clear.

A ***zero matrix*** is a matrix all of whose entries are 0. Such a matrix is often denoted 0, since the ambiguity between the number 0 and a matrix of 0s can usually be resolved from context. If a matrix of 0s is intended, then the size of the matrix also needs to be derived from the context.

Square matrices

Square $n \times n$ matrices arise frequently. Several special cases of square matrices are of particular interest:

1. A ***diagonal matrix*** has $a_{ij} = 0$ whenever $i \neq j$. Because all of the off-diagonal elements are 0, a succinct way to specify the matrix lists only the elements along the diagonal:

$$\mathrm{diag}(a_{11}, a_{22}, \ldots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \ldots & 0 \\ 0 & a_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & a_{nn} \end{pmatrix} .$$

2. The $n \times n$ ***identity matrix*** I_n is a diagonal matrix with 1s along the diagonal:

$$\begin{aligned} I_n &= \mathrm{diag}(1, 1, \ldots, 1) \\ &= \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{pmatrix} . \end{aligned}$$

When I appears without a subscript, its size derives from the context. The ith column of an identity matrix is the unit vector e_i.

3. A ***tridiagonal matrix*** T is one for which $t_{ij} = 0$ if $|i - j| > 1$. Nonzero entries appear only on the main diagonal, immediately above the main diagonal ($t_{i,i+1}$ for $i = 1, 2, \ldots, n-1$), or immediately below the main diagonal ($t_{i+1,i}$ for $i = 1, 2, \ldots, n-1$):

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \ldots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \ldots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \ldots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \ldots & 0 & t_{n,n-1} & t_{nn} \end{pmatrix} .$$

4. An ***upper-triangular matrix*** U is one for which $u_{ij} = 0$ if $i > j$. All entries below the diagonal are 0:

$$U = \begin{pmatrix} u_{11} & u_{12} & \ldots & u_{1n} \\ 0 & u_{22} & \ldots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & u_{nn} \end{pmatrix} .$$

An upper-triangular matrix is ***unit upper-triangular*** if it has all 1s along the diagonal.

5. A ***lower-triangular matrix*** L is one for which $l_{ij} = 0$ if $i < j$. All entries above the diagonal are 0:

$$L = \begin{pmatrix} l_{11} & 0 & \ldots & 0 \\ l_{21} & l_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \ldots & l_{nn} \end{pmatrix} .$$

A lower-triangular matrix is ***unit lower-triangular*** if it has all 1s along the diagonal.

6. A ***permutation matrix*** P has exactly one 1 in each row or column, and 0s elsewhere. An example of a permutation matrix is

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} .$$

   Such a matrix is called a permutation matrix because multiplying a vector x by a permutation matrix has the effect of permuting (rearranging) the elements of x. Exercise D.1-4 explores additional properties of permutation matrices.

7. A ***symmetric matrix*** A satisfies the condition $A = A^{\mathrm{T}}$. For example,

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

   is a symmetric matrix.

Basic matrix operations

The elements of a matrix or vector are ***scalar numbers*** from a number system, such as the real numbers, the complex numbers, or integers modulo a prime. The number system defines how to add and multiply scalars. These definitions extend to encompass addition and multiplication of matrices.

We define ***matrix addition*** as follows. If $A = (a_{ij})$ and $B = (b_{ij})$ are $m \times n$ matrices, then their matrix sum $C = (c_{ij}) = A + B$ is the $m \times n$ matrix defined by

$$c_{ij} = a_{ij} + b_{ij}$$

for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$. That is, matrix addition is performed componentwise. A zero matrix is the identity for matrix addition:

$$A + 0 = A = 0 + A .$$

If λ is a scalar number and $A = (a_{ij})$ is a matrix, then $\lambda A = (\lambda a_{ij})$ is the ***scalar multiple*** of A obtained by multiplying each of its elements by λ. As a special case, we define the ***negative*** of a matrix $A = (a_{ij})$ to be $-1 \cdot A = -A$, so that the ijth entry of $-A$ is $-a_{ij}$. Thus,

$$A + (-A) = 0 = (-A) + A .$$

The negative of a matrix defines ***matrix subtraction***: $A - B = A + (-B)$.

We define ***matrix multiplication*** as follows. Start with two matrices A and B that are ***compatible*** in the sense that the number of columns of A equals the number of rows of B. (In general, an expression containing a matrix product AB is always assumed to imply that matrices A and B are compatible.) If $A = (a_{ik})$ is a $p \times q$ matrix and $B = (b_{kj})$ is a $q \times r$ matrix, then their matrix product $C = AB$ is the $p \times r$ matrix $C = (c_{ij})$, where

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj} \tag{D.2}$$

for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, p$. The procedure RECTANGULAR-MATRIX-MULTIPLY on page 374 implements matrix multiplication in the straightforward manner based on equation (D.2), assuming that C is initialized to 0, using pqr multiplications and $p(q-1)r$ additions for a running time of $\Theta(pqr)$. If the matrices are $n \times n$ square matrices, so that $n = p = q = r$, the pseudocode reduces to MATRIX-MULTIPLY on page 81, whose running time is $\Theta(n^3)$. (Section 4.2 describes an asymptotically faster $\Theta(n^{\lg 7})$-time algorithm due to V. Strassen.)

Matrices have many (but not all) of the algebraic properties typical of numbers. Identity matrices are identities for matrix multiplication:

$$I_m A = A I_n = A$$

for any $m \times n$ matrix A. Multiplying by a zero matrix gives a zero matrix:

$$A \cdot 0 = 0 .$$

Matrix multiplication is associative:

$$A(BC) = (AB)C$$

for compatible matrices A, B, and C. Matrix multiplication distributes over addition:

$$\begin{aligned} A(B + C) &= AB + AC, \\ (B + C)D &= BD + CD . \end{aligned}$$

For $n > 1$, multiplication of $n \times n$ matrices is not commutative. For example, if $A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ and $B = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$, then $AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$ and $BA = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$.

We define matrix-vector products or vector-vector products as if the vector were the equivalent $n \times 1$ matrix (or a $1 \times n$ matrix, in the case of a row vector). Thus, if A is an $m \times n$ matrix and x is an n-vector, then Ax is an m-vector. If x and y are n-vectors, then

$$x^{\mathrm{T}} y = \sum_{i=1}^{n} x_i y_i$$

is a scalar number (actually a 1×1 matrix) called the ***inner product*** of x and y. We also use the notation $\langle x, y \rangle$ to denote $x^{\mathrm{T}} y$. The inner-product operator is commutative: $\langle x, y \rangle = \langle y, x \rangle$. The matrix $x y^{\mathrm{T}}$ is an $n \times n$ matrix Z called the ***outer product*** of x and y, where $z_{ij} = x_i y_j$. The ***(euclidean) norm*** $\|x\|$ of an n-vector x is defined by

$$\begin{aligned} \|x\| &= (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2} \\ &= (x^{\mathrm{T}} x)^{1/2} . \end{aligned}$$

Thus, the norm of x is its length in n-dimensional euclidean space. A useful fact, which follows from the equality

$$\left((a x_1)^2 + (a x_2)^2 + \cdots + (a x_n)^2\right)^{1/2} = |a| \, (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2}$$

is that for any real number a and n-vector x,

$$\|a x\| = |a| \, \|x\| \ . \tag{D.3}$$

Exercises

D.1-1
Show that if A and B are symmetric $n \times n$ matrices, then so are $A + B$ and $A - B$.

D.1-2
Prove that $(AB)^{\mathrm{T}} = B^{\mathrm{T}} A^{\mathrm{T}}$ and that $A^{\mathrm{T}} A$ is always a symmetric matrix.

D.1-3
Prove that the product of two lower-triangular matrices is lower-triangular.

D.1-4
Prove that if P is an $n \times n$ permutation matrix and A is an $n \times n$ matrix, then the matrix product PA is A with its rows permuted, and the matrix product AP is A with its columns permuted. Prove that the product of two permutation matrices is a permutation matrix.

D.2 Basic matrix properties

We now define some basic properties pertaining to matrices: inverses, linear dependence and independence, rank, and determinants. We also define the class of positive-definite matrices.

Matrix inverses, ranks, and determinants

The ***inverse*** of an $n \times n$ matrix A is the $n \times n$ matrix, denoted A^{-1} (if it exists), such that $AA^{-1} = I_n = A^{-1}A$. For example,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} .$$

Many nonzero $n \times n$ matrices do not have inverses. A matrix without an inverse is called ***noninvertible***, or ***singular***. An example of a nonzero singular matrix is

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} .$$

If a matrix has an inverse, it is called ***invertible***, or ***nonsingular***. Matrix inverses, when they exist, are unique. (See Exercise D.2-1.) If A and B are nonsingular $n \times n$ matrices, then

$$(BA)^{-1} = A^{-1}B^{-1} .$$

The inverse operation commutes with the transpose operation:

$$(A^{-1})^{\mathrm{T}} = (A^{\mathrm{T}})^{-1} .$$

The vectors $x_1, x_2, \ldots, x_n$ are ***linearly dependent*** if there exist coefficients $c_1, c_2, \ldots, c_n$, not all of which are 0, such that $c_1x_1 + c_2x_2 + \cdots + c_nx_n = 0$. The row vectors $x_1 = (\ 1\ \ 2\ \ 3\ )$, $x_2 = (\ 2\ \ 6\ \ 4\ )$, and $x_3 = (\ 4\ \ 11\ \ 9\ )$ are linearly dependent, for example, since $2x_1 + 3x_2 - 2x_3 = 0$. If vectors are not linearly dependent, they are ***linearly independent***. For example, the columns of an identity matrix are linearly independent.

The ***column rank*** of a nonzero $m \times n$ matrix A is the size of the largest set of linearly independent columns of A. Similarly, the ***row rank*** of A is the size of the largest set of linearly independent rows of A. A fundamental property of any matrix A is that its row rank always equals its column rank, so that we can simply refer to the ***rank*** of A. The rank of an $m \times n$ matrix is an integer between 0 and $\min\{m, n\}$, inclusive. (The rank of a zero matrix is 0, and the rank of an $n \times n$ identity matrix is n.) An alternate, but equivalent and often more useful, definition is that the rank of a nonzero $m \times n$ matrix A is the smallest number r such that there exist matrices B and C of respective sizes $m \times r$ and $r \times n$ such that $A = BC$. A square $n \times n$ matrix has ***full rank*** if its rank is n. An $m \times n$ matrix has ***full column rank*** if its rank is n. The following theorem gives a fundamental property of ranks.

Theorem D.1
A square matrix has full rank if and only if it is nonsingular. ■

A ***null vector*** for a matrix A is a nonzero vector x such that $Ax = 0$. The following theorem (whose proof is left as Exercise D.2-7) and its corollary relate the notions of column rank and singularity to null vectors.

Theorem D.2
A matrix has full column rank if and only if it does not have a null vector. ■

Corollary D.3
A square matrix is singular if and only if it has a null vector. ■

The ijth ***minor*** of an $n \times n$ matrix A, for $n > 1$, is the $(n-1) \times (n-1)$ matrix $A_{[ij]}$ obtained by deleting the ith row and jth column of A. The ***determinant*** of an $n \times n$ matrix A is defined recursively in terms of its minors by

$$\det(A) = \begin{cases} a_{11} & \text{if } n = 1 , \\ \displaystyle\sum_{j=1}^{n} (-1)^{1+j} a_{1j} \det(A_{[1j]}) & \text{if } n > 1 . \end{cases}$$

The term $(-1)^{i+j} \det(A_{[ij]})$ is known as the ***cofactor*** of the element a_{ij}.

The following theorems, whose proofs are omitted, express fundamental properties of the determinant.

Theorem D.4 (Determinant properties)
The determinant of a square matrix A has the following properties:

- If any row or any column of A is zero, then $\det(A) = 0$.
- The determinant of A is multiplied by λ if the entries of any one row (or any one column) of A are all multiplied by λ.
- The determinant of A is unchanged if the entries in one row (respectively, column) are added to those in another row (respectively, column).
- The determinant of A equals the determinant of A^{T}.
- The determinant of A is multiplied by -1 if any two rows (or any two columns) are exchanged.

Also, for any square matrices A and B, we have $\det(AB) = \det(A) \det(B)$. ■

Theorem D.5
An $n \times n$ matrix A is singular if and only if $\det(A) = 0$. ■

Positive-definite matrices

Positive-definite matrices play an important role in many applications. An $n \times n$ matrix A is ***positive-definite*** if $x^{\mathrm{T}}Ax > 0$ for all n-vectors $x \neq 0$. For example, the identity matrix is positive-definite, since if $x = (\, x_1 \;\; x_2 \;\; \cdots \;\; x_n \,)^{\mathrm{T}}$ is a nonzero vector, then

$$\begin{aligned} x^{\mathrm{T}} I_n x &= x^{\mathrm{T}} x \\ &= \sum_{i=1}^{n} x_i^2 \\ &> 0 \,. \end{aligned}$$

Matrices that arise in applications are often positive-definite due to the following theorem.

Theorem D.6
For any matrix A with full column rank, the matrix $A^{\mathrm{T}}A$ is positive-definite.

Proof We must show that $x^{\mathrm{T}}(A^{\mathrm{T}}A)x > 0$ for any nonzero vector x. For any vector x,

$$\begin{aligned} x^{\mathrm{T}}(A^{\mathrm{T}}A)x &= (Ax)^{\mathrm{T}}(Ax) \quad \text{(by Exercise D.1-2)} \\ &= \|Ax\|^2 \,. \end{aligned}$$

The value $\|Ax\|^2$ is just the sum of the squares of the elements of the vector Ax. Therefore, $\|Ax\|^2 \geq 0$. We'll show by contradiction that $\|Ax\|^2 > 0$. Suppose that $\|Ax\|^2 = 0$. Then, every element of Ax is 0, which is to say $Ax = 0$. Since A has full column rank, Theorem D.2 says that $x = 0$, which contradicts the requirement that x is nonzero. Hence, $A^{\mathrm{T}}A$ is positive-definite. ■

Section 28.3 explores other properties of positive-definite matrices. Section 33.3 uses a similar condition, known as positive-semidefinite. An $n \times n$ matrix A is ***positive-semidefinite*** if $x^{\mathrm{T}}Ax \geq 0$ for all n-vectors $x \neq 0$.

Exercises

D.2-1
Prove that matrix inverses are unique, that is, if B and C are inverses of A, then $B = C$.

D.2-2
Prove that the determinant of a lower-triangular or upper-triangular matrix is equal to the product of its diagonal elements. Prove that the inverse of a lower-triangular matrix, if it exists, is lower-triangular.

D.2-3
Prove that if P is a permutation matrix, then P is invertible, its inverse is P^{T}, and P^{T} is a permutation matrix.

D.2-4
Let A and B be $n \times n$ matrices such that $AB = I$. Prove that if A' is obtained from A by adding row j into row i, where $i \neq j$, then subtracting column i from column j of B yields the inverse B' of A'.

D.2-5
Let A be a nonsingular $n \times n$ matrix with complex entries. Show that every entry of A^{-1} is real if and only if every entry of A is real.

D.2-6
Show that if A is a nonsingular, symmetric, $n \times n$ matrix, then A^{-1} is symmetric. Show that if B is an arbitrary $m \times n$ matrix, then the $m \times m$ matrix given by the product BAB^{T} is symmetric.

D.2-7
Prove Theorem D.2. That is, show that a matrix A has full column rank if and only if $Ax = 0$ implies $x = 0$. (*Hint:* Express the linear dependence of one column on the others as a matrix-vector equation.)

D.2-8
Prove that for any two compatible matrices A and B,

$$\operatorname{rank}(AB) \leq \min\{\operatorname{rank}(A), \operatorname{rank}(B)\} ,$$

where equality holds if either A or B is a nonsingular square matrix. (*Hint:* Use the alternate definition of the rank of a matrix.)

Problems

D-1 Vandermonde matrix
Given numbers $x_0, x_1, \ldots, x_{n-1}$, prove that the determinant of the ***Vandermonde matrix***

$$V(x_0, x_1, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}$$

is

$$\det(V(x_0, x_1, \ldots, x_{n-1})) = \prod_{0 \le j < k \le n-1} (x_k - x_j) .$$

(*Hint:* Multiply column i by $-x_0$ and add it to column $i + 1$ for $i = n - 1$, $n - 2, \ldots, 1$, and then use induction.)

D-2 Permutations defined by matrix-vector multiplication over GF(2)

One class of permutations of the integers in the set $S_n = \{0, 1, 2, \ldots, 2^n - 1\}$ is defined by matrix multiplication over $GF(2)$, the Galois field of two elements. For each integer $x \in S_n$, we view its binary representation as an n-bit vector

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{pmatrix} ,$$

where $x = \sum_{i=0}^{n-1} x_i 2^i$. If A is an $n \times n$ matrix in which each entry is either 0 or 1, then we can define a permutation mapping each value $x \in S_n$ to the number whose binary representation is the matrix-vector product Ax. All this arithmetic is performed over ***GF*(2)**: all values are either 0 or 1, and with one exception, the usual rules of addition and multiplication apply. The exception is that $1 + 1 = 0$. You can think of arithmetic over $GF(2)$ as being just like regular integer arithmetic, except that you use only the least-significant bit.

As an example, for $S_2 = \{0, 1, 2, 3\}$, the matrix

$$A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

defines the following permutation π_A: $\pi_A(0) = 0$, $\pi_A(1) = 3$, $\pi_A(2) = 2$, $\pi_A(3) = 1$. To see why $\pi_A(3) = 1$, observe that, working in $GF(2)$,

$$\begin{aligned}\pi_A(3) &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 \cdot 1 + 0 \cdot 1 \\ 1 \cdot 1 + 1 \cdot 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 \\ 0 \end{pmatrix},\end{aligned}$$

which is the binary representation of 1.

For the remainder of this problem, we'll work over $GF(2)$, and all matrix and vector entries will be 0 or 1. Define the ***rank*** of a 0-1 matrix (a matrix for which each entry is either 0 or 1) over $GF(2)$ the same as for a regular matrix, but with all arithmetic that determines linear independence performed over $GF(2)$. We define the ***range*** of an $n \times n$ 0-1 matrix A by

$$R(A) = \{y : y = Ax \text{ for some } x \in S_n\} ,$$

so that $R(A)$ is the set of numbers in S_n that are produced by multiplying each value $x \in S_n$ by A.

a. If r is the rank of matrix A, prove that $|R(A)| = 2^r$. Conclude that A defines a permutation on S_n only if A has full rank.

For a given $n \times n$ matrix A and a given value $y \in R(A)$, we define the ***preimage*** of y by

$$P(A, y) = \{x : Ax = y\} ,$$

so that $P(A, y)$ is the set of values in S_n that map to y when multiplied by A.

b. If r is the rank of $n \times n$ matrix A and $y \in R(A)$, prove that $|P(A, y)| = 2^{n-r}$.

Let $0 \le m \le n$, and suppose that we partition the set S_n into blocks of consecutive numbers, where the ith block consists of the 2^m numbers $i2^m, i2^m + 1, i2^m + 2, \ldots, (i+1)2^m - 1$. For any subset $S \subseteq S_n$, define $B(S, m)$ to be the set of size-2^m blocks of S_n containing some element of S. As an example, when $n = 3$, $m = 1$, and $S = \{1, 4, 5\}$, then $B(S, m)$ consists of blocks 0 (since 1 is in the 0th block) and 2 (since both 4 and 5 belong to block 2).

c. Let r be the rank of the lower left $(n - m) \times m$ submatrix of A, that is, the matrix formed by taking the intersection of the bottom $n - m$ rows and the leftmost m columns of A. Let S be any size-2^m block of S_n, and let $S' = \{y : y = Ax \text{ for some } x \in S\}$. Prove that $|B(S', m)| = 2^r$ and that for each block in $B(S', m)$, exactly 2^{m-r} numbers in S map to that block.

Because multiplying the zero vector by any matrix yields a zero vector, the set of permutations of S_n defined by multiplying by $n \times n$ 0-1 matrices with full rank over $GF(2)$ cannot include all permutations of S_n. Let's extend the class of permutations defined by matrix-vector multiplication to include an additive term, so that $x \in S_n$ maps to $Ax + c$, where c is an n-bit vector and addition is performed over $GF(2)$. For example, when

$$A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

and

$$c = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

we get the following permutation $\pi_{A,c}$: $\pi_{A,c}(0) = 2$, $\pi_{A,c}(1) = 1$, $\pi_{A,c}(2) = 0$, $\pi_{A,c}(3) = 3$. We call any permutation that maps $x \in S_n$ to $Ax + c$, for some $n \times n$ 0-1 matrix A with full rank and some n-bit vector c, a ***linear permutation***.

d. Use a counting argument to show that the number of linear permutations of S_n is much less than the number of permutations of S_n.

e. Give an example of a value of n and a permutation of S_n that cannot be achieved by any linear permutation. (*Hint:* For a given permutation, think about how multiplying a matrix by a unit vector relates to the columns of the matrix.)

Appendix notes

Linear-algebra textbooks provide plenty of background information on matrices. The books by Strang [422, 423] are particularly good.

Bibliography

[1] Milton Abramowitz and Irene A. Stegun, editors. *Handbook of Mathematical Functions*. Dover, 1965.

[2] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3(5):1259–1263, 1962.

[3] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[4] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[6] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[7] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[8] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, 1990.

[9] Ravindra K. Ahuja and James B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5):748–759, 1989.

[10] Ravindra K. Ahuja, James B. Orlin, and Robert E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18(5):939–954, 1989.

[11] Miklós Ajtai, Nimrod Megiddo, and Orli Waarts. Improved algorithms and analysis for secretary problems and generalizations. *SIAM Journal on Discrete Mathematics*, 14(1):1–27, 2001.

[12] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice Hall, 1989.

[13] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.

[14] Susanne Albers. Online algorithms: A survey. *Mathematical Programming*, 97(1-2):3–26, 2003.

[15] Noga Alon. Generating pseudo-random permutations and maximum flow algorithms. *Information Processing Letters*, 35:201–204, 1990.

[16] Arne Andersson. Balanced search trees made simple. In *Proceedings of the Third Workshop on Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1993.

[17] Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 135–141, 1996.

[18] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57:74–93, 1998.

[19] Tom M. Apostol. *Calculus*, volume 1. Blaisdell Publishing Company, second edition, 1967.

[20] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

[21] Sanjeev Arora. *Probabilistic checking of proofs and the hardness of approximation problems*. PhD thesis, University of California, Berkeley, 1994.

[22] Sanjeev Arora. The approximability of NP-hard problems. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 337–348, 1998.

[23] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5):753–782, 1998.

[24] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[25] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: A meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[26] Sanjeev Arora and Carsten Lund. Hardness of approximations. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 399–446. PWS Publishing Company, 1997.

[27] Mikhail J. Atallah and Marina Blanton, editors. *Algorithms and Theory of Computation Handbook*, volume 1. Chapman & Hall/CRC Press, second edition, 2009.

[28] Mikhail J. Atallah and Marina Blanton, editors. *Algorithms and Theory of Computation Handbook*, volume 2. Chapman & Hall/CRC Press, second edition, 2009.

[29] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 1999.

[30] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. *ACM Transactions on Graphics*, 26(3), article 10, 2007.

[31] László Babai, Eugene M. Luks, and Ákos Seress. Fast management of permutation groups I. *SIAM Journal on Computing*, 26(5):1310–1342, 1997.

[32] Eric Bach. Private communication, 1989.

[33] Eric Bach. Number-theoretic algorithms. In *Annual Review of Computer Science*, volume 4, pages 119–172. Annual Reviews, Inc., 1990.

[34] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory—Volume I: Efficient Algorithms*. The MIT Press, 1996.

[35] Nikhil Bansal and Anupam Gupta. Potential-function proofs for first-order methods. *CoRR*, abs/1712.04581, 2017.

[36] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. In *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.

[37] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. *Journal of Algorithms*, 62(2):74–92, 2007.

[38] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[39] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[40] Pierre Beauchemin, Gilles Brassard, Claude Crépeau, Claude Goutier, and Carl Pomerance. The generation of random numbers that are probably prime. *Journal of Cryptology*, 1(1):53–64, 1988.

[41] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[42] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, 2000.

[43] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[44] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[45] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[46] Michael Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 80–86, 1983.

[47] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[48] Samuel W. Bent and John W. John. Finding the median requires $2n$ comparisons. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 213–216, 1985.

[49] Jon L. Bentley. *Writing Efficient Programs*. Prentice Hall, 1982.

[50] Jon L. Bentley. *More Programming Pearls: Confessions of a Coder*. Addison-Wesley, 1988.

[51] Jon L. Bentley. *Programming Pearls*. Addison-Wesley, second edition, 1999.

[52] Jon L. Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, 1980.

[53] Claude Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences*, 43(9):842–844, 1957.

[54] Aditya Y. Bhargava. *Grokking Algorithms: An Illustrated Guide For Programmers and Other Curious People*. Manning Publications, 2016.

[55] Daniel Bienstock and Benjamin McClosky. Tightening simplex mixed-integer sets with guaranteed bounds. *Optimization Online*, 2008.

[56] Patrick Billingsley. *Probability and Measure*. John Wiley & Sons, second edition, 1986.

[57] Guy E. Blelloch. *Scan Primitives and Parallel Vector Models*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1989. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-463.

[58] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[59] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 181–192, 2012.

[60] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *Proceedings of the 32nd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 89–102, 2020.

[61] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[62] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[63] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[64] Robert L Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism (HotPar)*, 2009.

[65] Béla Bollobás. *Random Graphs*. Academic Press, 1985.

[66] Leonardo Bonacci. *Liber Abaci*, 1202.

[67] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. American Elsevier, 1976.

[68] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[69] Stephen P. Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[70] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996.

[71] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[72] Gerth Stølting Brodal. A survey on priority queues. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 150–163. Springer, 2013.

[73] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. Strict Fibonacci heaps. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing*, pages 1177–1184, 2012.

[74] George W. Brown. Some notes on computation of games solutions. *RAND Corporation Report*, P-78, 1949.

[75] Sébastien Bubeck. Convex optimization: Algorithms and complexity. *Foundations and Trends in Machine Learning*, 8(3-4):231–357, 2015.

[76] Niv Buchbinder and Joseph Naor. The design of competitive online algorithms via a primal-dual approach. *Foundations and Trends in Theoretical Computer Science*, 3(2–3):93–263, 2009.

[77] J. P. Buhler, H. W. Lenstra, Jr., and Carl Pomerance. Factoring integers with the number field sieve. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 50–94. Springer, 1993.

[78] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. SRC Research Report 124, Digital Equipment Corporation Systems Research Center, May 1994.

[79] Neville Campbell. Recurrences. Unpublished treatise available at https://nevillecampbell.com/Recurrences.pdf, 2020.

[80] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[81] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[82] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 519–538, 2005.

[83] Bernard Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[84] Ke Chen and Adrian Dumitrescu. Selection algorithms with small groups. *International Journal of Foundations of Computer Science*, 31(3):355–369, 2020.

[85] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 298–309, 1998.

[86] Joseph Cheriyan and Torben Hagerup. A randomized maximum-flow algorithm. *SIAM Journal on Computing*, 24(2):203–226, 1995.

[87] Joseph Cheriyan and S. N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing*, 18(6):1057–1086, 1989.

[88] Boris V. Cherkassky and Andrew V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[89] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996.

[90] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, heaps, lists and monotone priority queues. *SIAM Journal on Computing*, 28(4):1326–1346, 1999.

[91] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23(4):493–507, 1952.

[92] Brian Christian and Tom Griffiths. *Algorithms to Live By: The Computer Science of Human Decisions*. Picador, 2017.

[93] Kai Lai Chung. *Elementary Probability Theory with Stochastic Processes*. Springer, 1974.

[94] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.

[95] V. Chvátal, D. A. Klarner, and D. E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72-292, Computer Science Department, Stanford University, 1972.

[96] Alan Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30. North-Holland, 1964.

[97] H. Cohen and H. W. Lenstra, Jr. Primality testing and Jacobi sums. *Mathematics of Computation*, 42(165):297–330, 1984.

[98] Michael B. Cohen, Aleksander Madry, Piotr Sankowski, and Adrian Vladu. Negative-weight shortest paths and unit capacity minimum cost flow in $\widetilde{O}(m^{10/7} \log w)$ time (extended abstract). In *Proceedings of the 28th ACM-SIAM Symposium on Discrete Algorithms*, pages 752–771, 2017.

[99] Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[100] Stephen Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[101] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[102] Don Coppersmith. Modifications to the number field sieve. *Journal of Cryptology*, 6(3):169–180, 1993.

[103] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[104] Thomas H. Cormen. *Algorithms Unlocked*. The MIT Press, 2013.

[105] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal on Computing*, 28(1):105–136, 1998.

[106] Don Dailey and Charles E. Leiserson. Using Cilk to write multiprocessor chess programs. In H. J. van den Herik and B. Monien, editors, *Advances in Computer Games*, volume 9, pages 25–52. University of Maastricht, Netherlands, 2001.

[107] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill, 2008.

[108] Abraham de Moivre. De fractionibus algebraicis radicalitate immunibus ad fractiones simpliciores reducendis, deque summandis terminis quarundam serierum aequali intervallo a se distantibus. *Philosophical Transactions*, 32(373):162–168, 1722.

[109] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic optimality—almost. *SIAM Journal on Computing*, 37(1):240–251, 2007.

[110] Camil Demetrescu, David Eppstein, Zvi Galik, and Giuseppe F. Italiano. Dynamic graph algorithms. In Mikhail J. Attalah and Marina Blanton, editors, *Algorithms and Theory of Computation Handbook*, chapter 9, pages 9-1–9-28. Chapman & Hall/CRC, second edition, 2009.

[111] Camil Demetrescu and Giuseppe F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. *Journal of Computer and System Sciences*, 72(5):813–837, 2006.

[112] Eric V. Denardo and Bennett L. Fox. Shortest-route methods: 1. Reaching, pruning, and buckets. *Operations Research*, 27(1):161–186, 1979.

[113] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[114] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[115] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[116] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[117] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[118] Dimitar Dimitrov, Martin Vechev, and Vivek Sarkar. Race detection in two dimensions. *ACM Transactions on Parallel Computing*, 4(4):1–22, 2018.

[119] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics Doklady*, 11(5):1277–1280, 1970.

[120] Brandon Dixon, Monika Rauch, and Robert E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[121] John D. Dixon. Factorization and primality tests. *The American Mathematical Monthly*, 91(6):333–352, 1984.

[122] Dorit Dor, Johan Håstad, Staffan Ulfberg, and Uri Zwick. On lower bounds for selecting the median. *SIAM Journal on Discrete Mathematics*, 14(3):299–311, 2001.

[123] Dorit Dor and Uri Zwick. Selecting the median. *SIAM Journal on Computing*, 28(5):1722–1758, 1999.

[124] Dorit Dor and Uri Zwick. Median selection requires $(2 + \epsilon)n$ comparisons. *SIAM Journal on Discrete Mathematics*, 14(3):312–325, 2001.

[125] Alvin W. Drake. *Fundamentals of Applied Probability Theory*. McGraw-Hill, 1967.

[126] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[127] Ran Duan, Seth Pettie, and Hsin-Hao Su. Scaling algorithms for weighted matching in general graphs. *ACM Transactions on Algorithms*, 14(1):8:1–8:35, 2018.

[128] Richard Durstenfeld. Algorithm 235 (RANDOM PERMUTATION). *Communications of the ACM*, 7(7):420, 1964.

[129] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[130] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[131] Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1(1):127–136, 1971.

[132] Jack Edmonds and Richard M. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[133] Jeff Edmonds. *How To Think About Algorithms*. Cambridge University Press, 2008.

[134] Mourad Elloumi and Albert Y. Zomaya, editors. *Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications*. John Wiley & Sons, 2011.

[135] Jeff Erickson. *Algorithms*. https://archive.org/details/Algorithms-Jeff-Erickson, 2019.

[136] Martin Erwig. *Once Upon an Algorithm: How Stories Explain Computing*. The MIT Press, 2017.

[137] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

[138] Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, 1981.

[139] William Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, third edition, 1968.

[140] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–11, 1997.

[141] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel Dominic Sleator, and Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[142] Amos Fiat and Gerhard J. Woeginger, editors. *Online Algorithms, The State of the Art*, volume 1442 of *Lecture Notes in Computer Science*. Springer, 1998.

[143] Sir Ronald A. Fisher and Frank Yates. *Statistical Tables for Biological, Agricultural and Medical Research*. Hafner Publishing Company, fifth edition, 1957.

[144] Robert W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.

[145] Robert W. Floyd. Algorithm 245 (TREESORT). *Communications of the ACM*, 7(12):701, 1964.

[146] Robert W. Floyd. Permuting information in idealized two-level storage. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 105–109. Plenum Press, 1972.

[147] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, 1975.

[148] L. R. Ford. *Network Flow Theory.* RAND Corporation, Santa Monica, CA, 1956.

[149] Lestor R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[150] Lestor R. Ford, Jr. and Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66(5):387–389, 1959.

[151] E. W. Forgy. Cluster analysis of multivariate efficiency versus interpretatbility of classifications. *Biometrics*, 21(3):768–769, 1965.

[152] Lance Fortnow. *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton University Press, 2013.

[153] Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976.

[154] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[155] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 345–354, 1989.

[156] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[157] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[158] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.

[159] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[160] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 79–90, 2009.

[161] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[162] Hannah Fry. *Hello World: Being Human in the Age of Algorithms*. W. W. Norton & Company, 2018.

[163] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3–4):107–114, 2000.

[164] Harold N. Gabow. The weighted matching approach to maximum cardinality matching. *Fundamenta Informaticae*, 154(1-4):109–130, 2017.

[165] Harold N. Gabow, Z. Galil, T. Spencer, and Robert E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[166] Harold N. Gabow and Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[167] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.

[168] Harold N. Gabow and Robert Endre Tarjan. Faster scaling algorithms for general graph-matching problems. *Journal of the ACM*, 38(4):815–853, 1991.

[169] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69(1):9–15, 1962.

[170] Zvi Galil and Oded Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134(2):103–139, 1997.

[171] Zvi Galil and Oded Margalit. All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences*, 54(2):243–254, 1997.

[172] Zvi Galil and Kunsoo Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92(1):49–76, 1992.

[173] Zvi Galil and Joel Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26(3):280–294, 1983.

[174] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174, 1993.

[175] Michael R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analyis of memory allocation algorithms. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, pages 143–150, 1972.

[176] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[177] Naveen Garg and Jochen Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM Journal on Computing*, 37(2):630–652, 2007.

[178] Saul Gass. *Linear Programming: Methods and Applications*. International Thomson Publishing, fourth edition, 1975.

[179] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

[180] Alan George and Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, 1981.

[181] E. N. Gilbert and E. F. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38(4):933–967, 1959.

[182] Ashish Goel, Sanjeev Khanna, Daniel H. Larkin, and Rober E. Tarjan. Disjoint set union with randomized linking. In *Proceedings of the 25th ACM-SIAM Symposium on Discrete Algorithms*, pages 1005–1017, 2014.

[183] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995.

[184] Michel X. Goemans and David P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 144–191. PWS Publishing Company, 1997.

[185] Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.

[186] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, 1995.

[187] Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, 1998.

[188] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[189] D. Goldfarb and M. J. Todd. Linear programming. In G. L. Nemhauser, A. H. G. Rinnooy-Kan, and M. J. Todd, editors, *Handbooks in Operations Research and Management Science, Vol. 1, Optimization*, pages 73–170. Elsevier Science Publishers, 1989.

[190] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[191] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[192] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[193] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures in Pascal and C*. Addison-Wesley, second edition, 1991.

[194] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992.

[195] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, 2001.

[196] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, sixth edition, 2014.

[197] Ronald L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[198] Ronald L. Graham and Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[199] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, second edition, 1994.

[200] David Gries. *The Science of Programming*. Springer, 1981.

[201] M. Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer, 1988.

[202] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[203] Dan Gusfield and Robert W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. The MIT Press, 1989.

[204] Gregory Gutin and Abraham P. Punnen, editors. *The Traveling Salesman Problem and Its Variations*. Kluwer Academic Publishers, 2002.

[205] Torben Hagerup. Improved shortest paths on the word RAM. In *Procedings of 27th International Colloquium on Automata, Languages and Programming, ICALP 2000*, volume 1853 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2000.

[206] H. Halberstam and R. E. Ingram, editors. *The Mathematical Papers of Sir William Rowan Hamilton*, volume III (Algebra). Cambridge University Press, 1967.

[207] Yijie Han. Improved fast integer sorting in linear space. *Information and Computation*, 170(1):81–94, 2001.

[208] Frank Harary. *Graph Theory*. Addison-Wesley, 1969.

[209] Gregory C. Harfst and Edward M. Reingold. A potential-based amortized analysis of the union-find data structure. *SIGACT News*, 31(3):86–95, 2000.

[210] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[211] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the Fast Fourier Transform. *IEEE ASSP Magazine*, 1(4):14–21, 1984.

[212] Monika R. Henzinger and Valerie King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 664–672, 1995.

[213] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[214] Monika R. Henzinger, Satish Rao, and Harold N. Gabow. Computing vertex connectivity: New bounds from old techniques. *Journal of Algorithms*, 34(2):222–250, 2000.

[215] Nicholas J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Transactions on Mathematical Software*, 16(4):352–368, 1990.

[216] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, second edition, 2002.

[217] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[218] C. A. R. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of the ACM*, 4(7):321–322, 1961.

[219] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–15, 1962.

[220] Dorit S. Hochbaum. Efficient bounds for the stable set, vertex cover and set packing problems. *Discrete Applied Mathematics*, 6(3):243–254, 1983.

[221] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997.

[222] W. Hoeffding. On the distribution of the number of successes in independent trials. *Annals of Mathematical Statistics*, 27(3):713–721, 1956.

[223] Micha Hofri. *Probabilistic Analysis of Algorithms*. Springer, 1987.

[224] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[225] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, third edition, 2006.

[226] John E. Hopcroft and Robert E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[227] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973.

[228] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[229] Juraj Hromkovič. *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*. Springer-Verlag, 2001.

[230] T. C. Hu and M. T. Shing. Computation of matrix chain products. Part I. *SIAM Journal on Computing*, 11(2):362–373, 1982.

[231] T. C. Hu and M. T. Shing. Computation of matrix chain products. Part II. *SIAM Journal on Computing*, 13(2):228–251, 1984.

[232] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetic codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

[233] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[234] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, 1975.

[235] E. J. Isaac and R. C. Singleton. Sorting by address calculation. *Journal of the ACM*, 3(3):169–174, 1956.

[236] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, 1974.

[237] David S. Johnson. The NP-completeness column: An ongoing guide—The tale of the second prover. *Journal of Algorithms*, 13(3):502–524, 1992.

[238] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[239] Richard Johnsonbaugh and Marcus Schaefer. *Algorithms*. Pearson Prentice Hall, 2004.

[240] Neil C. Jones and Pavel Pevzner. *An Introduction to Bioinformatics Algorithms*. The MIT Press, 2004.

[241] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. A local search approximation algorithm for k-means clustering. *Computational Geometry*, 28:89–112, 2004.

[242] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics—Doklady*, 7(7):595–596, 1963. Translation of an article in *Doklady Akademii Nauk SSSR*, 145(2), 1962.

[243] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[244] David R. Karger, Daphne Koller, and Steven J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6):1199–1217, 1993.

[245] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.

[246] Howard Karloff. *Linear Programming*. Birkhäuser, 1991.

[247] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[248] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[249] Richard M. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34(1–3):165–201, 1991.

[250] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[251] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15(2):434–437, 1974.

[252] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, volume 2089, pages 181–192. Springer-Verlag, 2001.

[253] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, second edition, 2015.

[254] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997.

[255] Valerie King, Satish Rao, and Robert E. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447–474, 1994.

[256] Philip N. Klein and Neal E. Young. Approximation algorithms for NP-hard optimization problems. In *CRC Handbook on Algorithms*, pages 34-1–34-19. CRC Press, 1999.

[257] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2006.

[258] Robert D. Kleinberg. A multiple-choice secretary algorithm with applications to online auctions. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 630–631, 2005.

[259] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[260] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[261] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1998.

[262] Donald E. Knuth. *Combinatorial Algorithms*, volume 4A of *The Art of Computer Programming*. Addison-Wesley, 2011.

[263] Donald E. Knuth. *Satisfiability*, volume 4, fascicle 6 of *The Art of Computer Programming*. Addison-Wesley, 2015.

[264] Donald E. Knuth. Optimum binary search trees. *Acta Informatica*, 1(1):14–25, 1971.

[265] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–23, 1976.

[266] Donald E. Knuth. *Stable Marriage and Its Relation to Other Combinatorial Problems: An Introduction to the Mathematical Analysis of Algorithms*, volume 10 of *CRM Proceedings and Lecture Notes*. American Mathematical Society, 1997.

[267] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[268] Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. The MIT Press, 2019.

[269] J. Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.

[270] Dexter C. Kozen. *The Design and Analysis of Algorithms*. Springer, 1992.

[271] David W. Krumme, George Cybenko, and K. N. Venkataraman. Gossiping in minimal time. *SIAM Journal on Computing*, 21(1):111–139, 1992.

[272] Joseph B. Kruskal, Jr. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[273] Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[274] William Kuszmaul and Charles E. Leiserson. Floors and ceilings in divide-and-conquer recurrences. In *Proceedings of the 3rd SIAM Symposium on Simplicity in Algorithms*, pages 133–141, 2021.

[275] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[276] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, 1976.

[277] Eugene L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

[278] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 2014 International Symposium on Symbolic and Algebraic Computation, (ISSAC)*, pages 296–303, 2014.

[279] Doug Lea. A Java fork/join framework. In *ACM 2000 Conference on Java Grande*, pages 36–43, 2000.

[280] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961.

[281] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(3):33–42, 2006.

[282] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing*, 2(3):17:1–17:42, 2015.

[283] I-Ting Angelina Lee and Tao B. Schardl. Efficient race detection for reducer hyperobjects. *ACM Transactions on Parallel Computing*, 4(4):1–40, 2018.

[284] Mun-Kyu Lee, Pierre Michaud, Jeong Seop Sim, and Daehun Nyang. A simple proof of optimality for the MIN cache replacement policy. *Information Processing Letters*, 116(2):168–170, 2016.

[285] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in $\widetilde{O}(\sqrt{rank})$ iterations and faster algorithms for maximum flow. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*, pages 424–433, 2014.

[286] Tom Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, 1985.

[287] Tom Leighton. Notes on better master theorems for divide-and-conquer recurrences. Class notes. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.1636, 1996.

[288] Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999.

[289] Daan Leijen and Judd Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, 2007.

[290] Charles E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, March 2010.

[291] Charles E. Leiserson. Cilk. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 273–288. Springer, 2011.

[292] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proceddings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 193–204, 2012.

[293] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science*, 368(6495), 2020.

[294] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, 1987.

[295] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard. The number field sieve. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 11–42. Springer, 1993.

[296] H. W. Lenstra, Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3):649–673, 1987.

[297] L. A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973. Translated from the original Russian article in *Problemy Peredachi Informatsii* 9(3): 115–116, 1973.

[298] Anany Levitin. *Introduction to the Design & Analysis of Algorithms*. Addison-Wesley, third edition, 2011.

[299] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, second edition, 1998.

[300] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1988.

[301] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108(2):212–261, 1994.

[302] C. L. Liu. *Introduction to Combinatorial Mathematics*. McGraw-Hill, 1968.

[303] Yang P. Liu and Aaron Sidford. Faster energy maximization for faster maximum flow. In *Proceedings of the 52nd Annual ACM Symposium on Theory of Computing*, pages 803–814, 2020.

[304] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[305] Panos Louridas. *Real-World Algorithms: A Beginner's Guide*. The MIT Press, 2017.

[306] László Lovász and Michael D. Plummer. *Matching Theory*, volume 121 of *Annals of Discrete Mathematics*. North Holland, 1986.

[307] John MacCormick. *9 Algorithms That Changed the Future: The Ingenious Ideas That Drive Today's Computers*. Princeton University Press, 2012.

[308] Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *Proceedings of the 54th Annual Symposium on Foundations of Computer Science*, pages 253–262, 2013.

[309] Bruce M. Maggs and Serge A. Plotkin. Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters*, 26(6):291–293, 1988.

[310] M. Mahajan, P. Nimbhorkar, and K. Varadarajan. The planar k-means problem is NP-hard. In S. Das and R. Uehara, editors, *WALCOM 2009: Algorithms and Computation*, volume 5431 of *Lecture Notes in Computer Science*, pages 274–285. Springer, 2009.

[311] Michael Main. *Data Structures and Other Objects Using Java*. Addison-Wesley, 1999.

[312] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[313] David F. Manlove. *Algorithmics of Matching Under Preferences*, volume 2 of *Series on Theoretical Computer Science*. World Scientific, 2013.

[314] Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[315] Mario Andrea Marchisio, editor. *Computational Methods in Synthetic Biology*. Humana Press, 2015.

[316] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[317] Yu. V. Matiyasevich. Real-time recognition of the inclusion relation. *Journal of Soviet Mathematics*, 1(1):64–70, 1973. Translated from the original Russian article in *Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Institute im. V. A. Steklova Akademii Nauk SSSR* 20: 104–114, 1971.

[318] H. A. Maurer, Th. Ottmann, and H.-W. Six. Implementing dictionaries using binary trees of very small height. *Information Processing Letters*, 5(1):11–14, 1976.

[319] Ernst W. Mayr, Hans Jürgen Prömel, and Angelika Steger, editors. *Lectures on Proof Verification and Approximation Algorithms*, volume 1367 of *Lecture Notes in Computer Science*. Springer, 1998.

[320] Catherine C. McGeoch. All pairs shortest paths and the essential subgraph. *Algorithmica*, 13(5):426–441, 1995.

[321] Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012.

[322] Andrew McGregor. Graph stream algorithms: A survey. *SIGMOD Record*, 43(1):9–20, 2014.

[323] M. D. McIlroy. A killer adversary for quicksort. *Software—Practice and Experience*, 29(4):341–344, 1999.

[324] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[325] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[326] Dinesh P. Mehta and Sartaj Sahni. *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, second edition, 2018.

[327] Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[328] Marvin Minsky and Seymore A. Pappert. *Perceptrons*. The MIT Press, 1969.

[329] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[330] Joseph S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. *SIAM Journal on Computing*, 28(4):1298–1309, 1999.

[331] Michael Mitzenmacher and Eli Upfal. *Probability and Computing*. Cambridge University Press, second edition, 2017.

[332] Louis Monier. *Algorithmes de Factorisation D'Entiers*. PhD thesis, L'Université Paris-Sud, 1980.

[333] Louis Monier. Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theoretical Computer Science*, 12(1):97–108, 1980.

[334] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[335] Rajeev Motwani, Joseph (Seffi) Naor, and Prabhakar Raghavan. Randomized approximation algorithms in combinatorial optimization. In Dorit Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 11, pages 447–481. PWS Publishing Company, 1997.

[336] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[337] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[338] J. I. Munro and V. Raman. Fast stable in-place sorting with $O(n)$ data moves. *Algorithmica*, 16(2):151–160, 1996.

[339] Yoichi Muraoka and David J. Kuck. On the time required for a sequence of matrix products. *Communications of the ACM*, 16(1):22–26, 1973.

[340] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[341] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[342] Richard Neapolitan. *Foundations of Algorithms*. Jones & Bartlett Learning, fifth edition, 2014.

[343] Yurii Nesterov. *Introductory Lectures on Convex Optimization: A Basic Course*, volume 87 of *Applied Optimization*. Springer, 2004.

[344] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[345] Ivan Niven and Herbert S. Zuckerman. *An Introduction to the Theory of Numbers*. John Wiley & Sons, fourth edition, 1980.

[346] National Institute of Standards and Technology. Hash functions. https://csrc.nist.gov/projects/hash-functions, 2019.

[347] Alan V. Oppenheim and Ronald W. Schafer, with John R. Buck. *Discrete-Time Signal Processing*. Prentice Hall, second edition, 1998.

[348] Alan V. Oppenheim and Alan S. Willsky, with S. Hamid Nawab. *Signals and Systems*. Prentice Hall, second edition, 1997.

[349] James B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78(1):109–129, 1997.

[350] James B. Orlin. Max flows in $O(nm)$ time, or better. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing*, pages 765–774, 2013.

[351] Anna Pagh, Rasmus Pagh, and Milan Ruzic. Linear probing with constant independence. https://arxiv.org/abs/cs/0612055, 2006.

[352] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[353] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[354] Michael S. Paterson. Progress in selection. In *Proceedings of the Fifth Scandinavian Workshop on Algorithm Theory*, pages 368–379, 1996.

[355] Seth Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47–74, 2004.

[356] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, 2002.

[357] Seth Pettie and Vijaya Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM Journal on Computing*, 34(6):1398–1431, 2005.

[358] Steven Phillips and Jeffery Westbrook. Online load balancing and network flow. *Algorithmica*, 21(3):245–261, 1998.

[359] Serge A. Plotkin, David. B. Shmoys, and Éva Tardos. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research*, 20:257–301, 1995.

[360] J. M. Pollard. Factoring with cubic integers. In A. K. Lenstra and H. W. Lenstra, Jr., editors, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 4–10. Springer, 1993.

[361] Carl Pomerance. On the distribution of pseudoprimes. *Mathematics of Computation*, 37(156):587–593, 1981.

[362] Carl Pomerance, editor. *Proceedings of the AMS Symposia in Applied Mathematics: Computational Number Theory and Cryptography*. American Mathematical Society, 1990.

[363] William K. Pratt. *Digital Image Processing*. John Wiley & Sons, fourth edition, 2007.

[364] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer, 1985.

[365] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, second edition, 2002.

[366] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, third edition, 2007.

[367] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[368] Robert L. Probert. On the additive complexity of matrix multiplication. *SIAM Journal on Computing*, 5(2):187–203, 1976.

[369] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[370] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2), 2007.

[371] Paul W. Purdom, Jr. and Cynthia A. Brown. *The Analysis of Algorithms*. Holt, Rinehart, and Winston, 1985.

[372] Michael O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39. Academic Press, 1976.

[373] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[374] P. Raghavan and C. D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

[375] Rajeev Raman. Recent results on the single-source shortest paths problem. *SIGACT News*, 28(2):81–87, 1997.

[376] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[377] Edward M. Reingold, Kenneth J. Urban, and David Gries. K-M-P string matching revisited. *Information Processing Letters*, 64(5):217–223, 1997.

[378] Hans Riesel. *Prime Numbers and Computer Methods for Factorization*, volume 126 of *Progress in Mathematics*. Birkhäuser, second edition, 1994.

[379] Ronald L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6 block cipher. In *First Advanced Encryption Standard (AES) Conference*, 1998.

[380] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. See also U.S. Patent 4,405,829.

[381] Herbert Robbins. A remark on Stirling's formula. *American Mathematical Monthly*, 62(1):26–29, 1955.

[382] Julia Robinson. An iterative method of solving a game. *The Annals of Mathematics*, 54(2):296–301, 1951.

[383] Arch D. Robison and Charles E. Leiserson. Cilk Plus. In Pavan Balaji, editor, *Programming Models for Parallel Computing*, chapter 13, pages 323–352. The MIT Press, 2015.

[384] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563–581, 1977.

[385] Tim Roughgarden. *Algorithms Illuminated, Part 1: The Basics*. Soundlikeyourself Publishing, 2017.

[386] Tim Roughgarden. *Algorithms Illuminated, Part 2: Graph Algorithms and Data Structures*. Soundlikeyourself Publishing, 2018.

[387] Tim Roughgarden. *Algorithms Illuminated, Part 3: Greedy Algorithms and Dynamic Programming*. Soundlikeyourself Publishing, 2019.

[388] Tim Roughgarden. *Algorithms Illuminated, Part 4: Algorithms for NP-Hard Problems*. Soundlikeyourself Publishing, 2020.

[389] Salvador Roura. Improved master theorems for divide-and-conquer recurrences. *Journal of the ACM*, 48(2):170–205, 2001.

[390] Y. A. Rozanov. *Probability Theory: A Concise Course*. Dover, 1969.

[391] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, fourth edition, 2020.

[392] S. Sahni and T. Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23(3):555–565, 1976.

[393] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures: The Basic Toolkit*. Springer, 2019.

[394] Piotr Sankowski. Shortest paths in matrix multiplication time. In *Proceedings of the 13th Annual European Symposium on Algorithms*, pages 770–778, 2005.

[395] Russel Schaffer and Robert Sedgewick. The analysis of heapsort. *Journal of Algorithms*, 15(1):76–100, 1993.

[396] Tao B. Schardl, I-Ting Angelina Lee, and Charles E. Leiserson. Brief announcement: Open Cilk. In *Proceedings of the 30th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 351–353, 2018.

[397] A. Schönhage, M. Paterson, and N. Pippenger. Finding the median. *Journal of Computer and System Sciences*, 13(2):184–199, 1976.

[398] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[399] Alexander Schrijver. Paths and flows—A historical survey. *CWI Quarterly*, 6(3):169–183, 1993.

[400] Alexander Schrijver. On the history of the shortest paths problem. *Documenta Mathematica*, 17(1):155–167, 2012.

[401] Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978.

[402] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, fourth edition, 2011.

[403] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

[404] Raimund Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4–5):464–497, 1996.

[405] João Setubal and João Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[406] Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*. Prentice Hall, second edition, 2001.

[407] Jeffrey Shallit. Origins of the analysis of the Euclidean algorithm. *Historia Mathematica*, 21(4):401–419, 1994.

[408] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

[409] David B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems. In William Cook, László Lovász, and Paul Seymour, editors, *Combinatorial Optimization*, volume 20 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1995.

[410] Avi Shoshan and Uri Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 605–614, 1999.

[411] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, second edition, 2009.

[412] Julian Shun. *Shared-Memory Parallelism Can Be Simple, Fast, and Scalable*. Association for Computing Machinery and Morgan & Claypool, 2017.

[413] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, third edition, 2013.

[414] Steven S. Skiena. *The Algorithm Design Manual*. Springer, second edition, corrected printing, 2012.

[415] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[416] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update rules. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, pages 488–492, 1984.

[417] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[418] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[419] Michael Soltys-Kulinicz. *An Introduction to the Analysis of Algorithms*. World Scientific, third edition, 2018.

[420] Joel Spencer. *Ten Lectures on the Probabilistic Method*, volume 64 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, 1993.

[421] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3):385–463, 2004.

[422] Gilbert Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986.

[423] Gilbert Strang. *Linear Algebra and Its Applications*. Thomson Brooks/Cole, fourth edition, 2006.

[424] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[425] T. G. Szymanski. A special case of the maximal common subsequence problem. Technical Report TR-170, Computer Science Laboratory, Princeton University, 1975.

[426] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[427] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[428] Robert E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979.

[429] Robert E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[430] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[431] Robert E. Tarjan. Class notes: Disjoint set union. COS 423, Princeton University, 1999. Available at https://www.cs.princeton.edu/courses/archive/spr00/cs423/handout3.pdf.

[432] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

[433] George B. Thomas, Jr., Maurice D. Weir, Joel Hass, and Frank R. Giordano. *Thomas' Calculus*. Addison-Wesley, eleventh edition, 2005.

[434] Mikkel Thorup. Faster deterministic sorting and priority queues in linear space. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 550–555, 1998.

[435] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.

[436] Mikkel Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000.

[437] Mikkel Thorup. High speed hashing for integers and strings. http://arxiv.org/abs/1504.06804, 2015.

[438] Mikkel Thorup. Linear probing with 5-independent hashing. http://arxiv.org/abs/1509.04549, 2015.

[439] Richard Tolimieri, Myoung An, and Chao Lu. *Mathematics of Multidimensional Fourier Transform Algorithms*. Springer, second edition, 1997.

[440] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[441] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(1):99–127, 1976.

[442] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992.

[443] Benjamin Van Roy. A short proof of optimality for the MIN cache replacement algorithm. *Information Processing Letters*, 102(2–3):72–73, 2007.

[444] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer Academic Publishers, 1996.

[445] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing*, pages 887–898, 2012.

[446] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[447] Rakesh M. Verma. General techniques for analyzing recursive algorithms with applications. *SIAM Journal on Computing*, 26(2):568–581, 1997.

[448] Berthold Vöcking, Helmut Alt, Martin Dietzfelbinger, Rüdiger Reischuk, Christian Scheideler, Heribert Vollmer, and Dorothea Wager, editors. *Algorithms Unplugged*. Springer, 2011.

[449] Antony F. Ware. Fast approximate Fourier transforms for irregularly spaced data. *SIAM Review*, 40(4):838–856, 1998.

[450] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[451] Mark Allen Weiss. *Data Structures and Problem Solving Using C++*. Addison-Wesley, second edition, 2000.

[452] Mark Allen Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley, third edition, 2006.

[453] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, third edition, 2007.

[454] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison-Wesley, second edition, 2007.

[455] Herbert S. Wilf. *Algorithms and Complexity*. A K Peters, second edition, 2002.

[456] J. W. J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7(6):347–348, 1964.

[457] Ryan Williams. Faster all-pairs shortest paths via circuit complexity. *SIAM Journal on Computing*, 47(5):1965–1985, 2018.

[458] David P. Williamson. *Network Flow Algorithms*. Cambridge University Press, 2019.

[459] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

[460] Shmuel Winograd. On the algebraic complexity of functions. In *Actes du Congrès International des Mathématiciens*, volume 3, pages 283–288, 1970.

[461] Yifan Xu, I-Ting Angelina Lee, and Kunal Agrawal. Efficient parallel determinacy race detection for two-dimensional dags. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 368–380, 2018.

[462] Chee Yap. A real elementary approach to the master recurrence and generalizations. In M. Ogihara and J. Tarui, editors, *Theory and Applications of Models of Computation. TAMC 2011*, volume 6648 of *Lecture Notes in Computer Science*, pages 14–26. Springer, 2011.

[463] Yinyu Ye. *Interior Point Algorithms: Theory and Analysis*. John Wiley & Sons, 1997.

[464] Neal E. Young. Online paging and caching. In *Encyclopedia of Algorithms*, pages 1457–1461. Springer, 2016.

[465] Raphael Yuster and Uri Zwick. Answering distance queries in directed graphs using fast matrix multiplication. In *Proceedings of the 46th Annual Symposium on Foundations of Computer Science*, pages 389–396, 2005.

[466] Jisheng Zhao and Vivek Sarkar. The design and implementation of the Habanero-Java parallel programming language. In *Symposium on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 185–186, 2011.

[467] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, 2002.

[468] Daniel Zwillinger, editor. *CRC Standard Mathematical Tables and Formulae*. Chapman & Hall/CRC Press, 31st edition, 2003.

Index

This index uses the following conventions. Numbers are alphabetized as if spelled out; for example, "2-3-4 tree" is indexed as if it were "two-three-four tree." When an entry refers to a place other than the main text, the page number is followed by a tag: ex. for exercise, pr. for problem, fig. for figure, and n. for footnote. A tagged page number often indicates the first page of an exercise or problem, which is not necessarily the page on which the reference actually appears.

$\alpha(n)$, 533
α-strongly convex function, 1041
β-smooth function, 1041
δ
  (shortest-path distance), 558
  (shortest-path weight), 604
ϕ (golden ratio), 69
$\hat{\phi}$ (conjugate of the golden ratio), 69
$\phi(n)$ (Euler's phi function), 920
π
  (predecessor in a breadth-first tree), 555
  (predecessor in a shortest-paths tree), 608
$\rho(n)$-approximation algorithm, 1104, 1120
o-notation, 60
O-notation, 50, 54–55
O'-notation, 73 pr.
$\widetilde{O}$-notation, 73 pr.
ω-notation, 61
Ω-notation, 51, 54 fig., 55–56
$\overset{\infty}{\Omega}$-notation, 73 pr.
$\widetilde{\Omega}$-notation, 73 pr.
Θ-notation, 33, 51, 54 fig., 56
$\widetilde{\Theta}$-notation, 73 pr.
{ } (set), 1153
$\in$ (set member), 1153
$\notin$ (not a set member), 1153
$\emptyset$
  (empty language), 1052
  (empty set), 1153
$\subseteq$ (subset), 1154
$\subset$ (proper subset), 1154
: (such that), 54 n., 1154
$\cap$ (set intersection), 1154
$\cup$ (set union), 1154
$-$ (set difference), 1154
| |
  (flow value), 672
  (length of a string), 959
  (set cardinality), 1156
$\times$ (Cartesian product), 1157
$\langle\ \rangle$
  (sequence), 1162
  (standard encoding), 1052
: (subarray), 19, 23
$[a, b]$ (closed interval), 1157
(a, b) (open interval), 1157
$[a, b)$ or $(a, b]$ (half-open interval), 1157
$\binom{n}{k}$ (choose), 1180
$\|\ \|$ (euclidean norm), 1219
! (factorial), 67–68
$\lceil\ \rceil$ (ceiling), 63
$\lfloor\ \rfloor$ (floor), 63

∂ (partial derivative), 1023
$\sum$ (sum), 1140
$\prod$ (product), 1144
$\rightarrow$ (adjacency relation), 1165
$\rightsquigarrow$ (reachability relation), 1165
$\wedge$ (AND), 659, 1065
$\|$ (concatenation), 291
$\neg$ (NOT), 1065
$\vee$ (OR), 659, 1065
$\lll$ (left shift), 305
$\ggg$ (logical right shift), 285
$\oplus$
  (group operator), 917
  (semiring operator), 651 n.
  (symmetric difference), 706
$\otimes$
  (convolution operator), 880
  (semiring operator), 651 n.
* (closure operator), 1052
| (divides relation), 904
$\nmid$ (does-not-divide relation), 904
$= \pmod{n}$ (equivalent, modulo n), 64
$\not\equiv \pmod{n}$ (not equivalent, modulo n), 64
$[a]_n$ (equivalence class modulo n), 905
$+_n$ (addition modulo n), 917
$\cdot_n$ (multiplication modulo n), 917
$\left(\frac{a}{p}\right)$ (Legendre symbol), 954 pr.
ε (empty string), 959, 1052
$\sqsubset$ (prefix relation), 959
$\sqsupset$ (suffix relation), 959
// (comment symbol), 22
$\gg$ (much-greater-than relation), 533
$\ll$ (much-less-than relation), 761
$\leq_P$ (polynomial-time reducibility relation), 1062, 1071 ex.

AA-tree, 358
abelian group, 917
absent child, 1173
absolutely convergent series, 1140
absorption laws for sets, 1155
abstract problem, 1048
abuse of asymptotic notation, 55, 59–60
acceptable pair of integers, 950
acceptance
  by an algorithm, 1053
  by a finite automaton, 968
accepting state, 967
accounting method, 453–456
  for binary counters, 455
  for dynamic tables, 463
  for stack operations, 454–455
Ackermann's function, 544
activity-selection problem, 418–425
acyclic graph, 1166
ADD-BINARY-INTEGERS, 25 ex.
add instruction, 26
addition
  of matrices, 1217
  modulo n ($+_n$), 917
  of polynomials, 877
additive group modulo n, 918
addressing, open, *see* open-address hash table
ADD-SUBARRAY, 783 pr.
adjacency-list representation, 550–551
  replaced by a hash table, 553 ex.
adjacency-matrix representation, 551–552
adjacency relation ($\rightarrow$), 1165
adjacent vertices, 1165
Advanced Encryption Standard (AES), 291
adversary, 204, 286, 805, 807, 941
AES, 291
aggregate analysis, 449–453
  for binary counters, 451–453
  for breadth-first search, 558
  for depth-first search, 566–567
  for Dijkstra's algorithm, 623–624
  for disjoint-set data structures, 525–526, 527 ex.
  for dynamic tables, 462–463
  for the Knuth-Morris-Pratt algorithm, 977–978
  for Prim's algorithm, 597
  for rod cutting, 370
  for shortest paths in a dag, 617
  for stack operations, 449–451
aggregate flow, 864
Akra-Bazzi recurrence, 115–119
  solving by Akra-Bazzi method, 117–118
algorithm, 1–1226
  analysis of, 25–34
  approximation, 1104–1136
  compare-exchange, 222 pr.
  correctness of, 6
  decision, 1053

algorithm, *continued*
deterministic, 135
lookahead, 815 ex.
nondeterministic, 765
oblivious, 222 pr.
offline, 791
online, *see* online algorithm
origin of word, 48
parallel, *see* parallel algorithm
push-relabel, 702
randomized, *see* randomized algorithm
recursive, 34
reduction, 1046, 1062
running time of, 29
scaling, 641 pr., 699 pr.
streaming, 818
as a technology, 13
verification, 1058
algorithmic recurrence, 77–78
ALLOCATE-NODE, 506
all-pairs shortest paths, 605, 646–669
in dynamic graphs, 669
in ϵ-dense graphs, 668 pr.
Floyd-Warshall algorithm for, 655–659
Johnson's algorithm for, 662–667
by matrix multiplication, 648–655, 668–669
by repeated squaring, 652–653
α-balanced, 472 pr.
$\alpha(n)$, 533
α-strongly convex function, 1041
alphabet, 967, 1052
alternating path, 705
amortized analysis, 448–475
by accounting method, 453–456
by aggregate analysis, 370, 449–453
for breadth-first search, 558
for depth-first search, 566–567
for Dijkstra's algorithm, 623–624
for disjoint-set data structures, 525–526, 527 ex., 531 ex., 534–540, 541 ex.
for dynamic tables, 460–471
for the Knuth-Morris-Pratt algorithm, 977–978
for making binary search dynamic, 472 pr.
by potential method, 456–460
for Prim's algorithm, 597
for restructuring red-black trees, 473 pr.
for shortest paths in a dag, 617
for stacks on secondary storage, 517 pr.
for weight-balanced trees, 472 pr.
amortized cost
in the accounting method, 453
in aggregate analysis, 449
in the potential method, 456
amortized progress, 1028
analysis of algorithms, 25–34
see also amortized analysis, competitive analysis, probabilistic analysis
ancestor, 1172
lowest common, 543 pr.
AND function ($\wedge$), 659, 1065
AND gate, 1065
and, in pseudocode, 24
antiparallel edges, 673–674
antisymmetric relation, 1160
approximation
by least squares, 841–845
of summation by integrals, 1150
approximation algorithm, 1103–1136
for bin packing, 1131 pr.
for MAX-CNF satisfiability, 1124 ex.
for maximum clique, 1131 pr.
for maximum matching, 1132 pr.
for maximum spanning tree, 1134 pr.
for maximum-weight cut, 1124 ex.
for MAX-3-CNF satisfiability, 1120–1121
for parallel machine scheduling, 1133 pr.
randomized, 1120
for set cover, 1115–1119
for subset sum, 1124–1130
for traveling-salesperson problem, 1109–1115
for vertex cover, 1106–1109, 1121–1124
for weighted set cover, 1132 pr.
for 0-1 knapsack problem, 1134 pr.
approximation error, 842
approximation ratio, 1104, 1120
approximation scheme, 1105
APPROX-MIN-WEIGHT-VC, 1123
APPROX-SUBSET-SUM, 1128
APPROX-TSP-TOUR, 1111
APPROX-VERTEX-COVER, 1107
arbitrage, 641 pr.
arc, *see* edge
argument of a function, 1161–1162
arithmetic instructions, 26

arithmetic, modular, 64, 916–923
arithmetic series, 1141
arithmetic with infinities, 611
arm in a disk drive, 498
array
  indexing into, 22–23, 26 n., 252
  inversion in, 47 pr.
  Monge, 123 pr.
  passing as a parameter, 24
  in pseudocode, 22–23
  storage of, 26 n., 252
articulation point, 582 pr.
assignment
  optimal, 723–739
  satisfying, 1066, 1074
  truth, 1066, 1073
assignment problem, 723–739
associative laws for sets, 1155
associative operation, 917
asymptotically larger, 62
asymptotically nonnegative, 54
asymptotically positive, 54
asymptotically smaller, 62
asymptotically tight bound, 56
asymptotic lower bound, 55
asymptotic notation, 53–63, 72 pr.
  and graph algorithms, 548
  and linearity of summations, 1141
asymptotic running time, 49
asymptotic upper bound, 54
attribute
  in clustering, 1006
  in a graph, 552
  of an object, 23
augmentation of a flow, 678
augmented primal linear program, 870
augmenting data structures, 480–496
augmenting path, 681–682, 705
  widest, 700 pr.
authentication, 309 pr., 938–939, 942
automaton, 967–974
auxiliary hash function, 295
average-case running time, 32, 128
AVL tree, 357 pr., 358

back edge, 569, 573
back substitution, 823
balanced search tree
  AA-trees, 358
  AVL trees, 357 pr., 358
  B-trees, 497–519
  k-neighbor trees, 358
  left-leaning red-black binary search trees, 358
  red-black trees, 331–359
  scapegoat trees, 358
  splay trees, 359, 478
  treaps, 358
  2-3-4 trees, 502, 518 pr.
  2-3 trees, 358, 519
  weight-balanced trees, 358, 472 pr.
balls and bins, 143–144, 1212 pr.
base-a pseudoprime, 944
base case
  of a divide-and-conquer algorithm, 34, 76
  of a recurrence, 41, 77–78
base, in DNA, 393
basis function, 841
Bayes's theorem, 1189
BELLMAN-FORD, 612
Bellman-Ford algorithm, 612–616
  for all-pairs shortest paths, 647
  in Johnson's algorithm, 664–666
  and objective functions, 632 ex.
  to solve systems of difference constraints, 630–631
  Yen's improvement to, 640 pr.
Bernoulli trial, 1196
  and balls and bins, 143–144
  in bucket sort analysis, 217
  in finding prime numbers, 943
  in randomized selection analysis, 232
  and streaks, 144–150
best-case running time, 34 ex.
β-smooth function, 1041
BFS, 556
  see also breadth-first search
BIASED-RANDOM, 129 ex.
biconnected component, 582 pr.
big-oh notation (O), 50, 54–55
big-omega notation (Ω), 51, 54 fig., 55–56
bijective function, 1162
binary character code, 431

binary counter
  analyzed by accounting method, 455
  analyzed by aggregate analysis, 451–453
  analyzed by potential method, 458–459
binary entropy function, 1182
binary gcd algorithm, 953 pr.
binary heap, *see* heap
binary logarithm (lg), 66
binary reflected Gray code, 471 pr.
binary relation, 1158
binary search, 44 ex.
  with fast insertion, 472 pr.
  in insertion sort, 45 ex.
  in parallel merging, 777–778
  in searching B-trees, 512 ex.
binary search tree, 312–330
  AA-trees, 358
  AVL trees, 357 pr., 358
  deletion from, 322–325, 326 ex.
  with equal keys, 327 pr.
  insertion into, 321–322
  k-neighbor trees, 358
  left-leaning red-black binary search trees, 358
  maximum key of, 317–318
  minimum key of, 317–318
  optimal, 400–407
  persistent, 355 pr.
  predecessor in, 318–319
  querying, 316–320
  randomly built, 328 pr.
  red-black trees, 331–359
  right-converting of, 337 ex.
  scapegoat trees, 358
  searching, 316–317
  for sorting, 326 ex.
  splay trees, 359
  successor in, 318–319
  weight-balanced trees, 358
  see also red-black tree
binary-search-tree property, 313–314
  vs. min-heap property, 315 ex.
binary tree, 1173
  full, 433, 1174
  number of different ones, 329 pr.
  representation of, 265
  see also binary search tree
binomial coefficient, 1181–1182
binomial distribution, 1198–1201
  and balls and bins, 143
  in bucket sort analysis, 217
  maximum value of, 1202 ex.
  tails of, 1203–1210
binomial expansion, 1181
binomial theorem, 1181
bin packing, 1131 pr.
bipartite graph, 1167
  complete, 716
  corresponding flow network of, 694
  d-regular, 716 ex., 740 pr.
  matching in, 693–697, 704–743
bipartite matching, 693–697, 704–743
birthday paradox, 140–143
bisection of a tree, 1177 pr.
bitonic euclidean traveling-salesperson problem, 407 pr.
bitonic sequence, 644 pr.
bitonic tour, 407 pr.
bit operation, 904
  in Euclid's algorithm, 954 pr.
bit-reversal permutation, 897
bit vector, 274 ex.
black-height, 332
black vertex, 554, 564
block
  in a cache, 440, 802
  on a disk, 499, 512 ex., 517 pr.
blocking flow, 702
blocking pair, 716
block representation of matrices, 254
block structure in pseudocode, 21–22
body, 1032
Boole's inequality, 1190 ex.
boolean combinational circuit, 1065
boolean combinational element, 1065
boolean connective, 1073
boolean data type, 26
boolean formula, 1043, 1060 ex., 1073–1074
boolean function, 1182 ex.
boolean operators, 24
Borůvka's algorithm, 603
bottleneck spanning tree, 601 pr.
bottleneck traveling-salesperson problem, 1115 ex.
bottoming out, 76
bottom of a stack, 254

BOTTOM-UP-CUT-ROD, 369
bottom-up method, for dynamic programming, 368
bound
  asymptotically tight, 56
  asymptotic lower, 55
  asymptotic upper, 54
  on binomial coefficients, 1181–1182
  on binomial distributions, 1201
  polylogarithmic, 67
  on the tails of a binomial distribution, 1203–1210
  see also lower bounds
bounding a summation, 1145–1152
box, nesting, 640 pr.
B^+-tree, 501
branch instructions, 26
breadth-first forest, 728
breadth-first search, 554–563
  in the Hopcroft-Karp algorithm, 711
  in the Hungarian algorithm, 727–728
  in maximum flow, 689–691
  and shortest paths, 558–561, 605
  similarity to Dijkstra's algorithm, 624, 625 ex.
breadth-first tree, 555, 561
bridge, 582 pr.
B^*-tree, 502 n.
B-tree, 497–519
  compared with red-black trees, 497, 503
  creating, 505–506
  deletion from, 513–516
  full node in, 502
  height of, 502–504
  insertion into, 506–511
  minimum degree of, 502
  properties of, 501–504
  searching, 504–505
  splitting a node in, 506–508
  2-3-4 trees, 502
B-TREE-CREATE, 506
B-TREE-DELETE, 513
B-TREE-INSERT, 508
B-TREE-INSERT-NONFULL, 511
B-TREE-SEARCH, 505, 512 ex.
B-TREE-SPLIT-CHILD, 507
B-TREE-SPLIT-ROOT, 509
BUBBLESORT, 46 pr.
bucket, 215
bucket sort, 215–219
BUCKET-SORT, 216
BUILD-MAX-HEAP, 167
BUILD-MAX-HEAP′, 179 pr.
BUILD-MIN-HEAP, 169
Burrows-Wheeler transform (BWT), 1000 pr.
butterfly operation, 894
BWT (Burrows-Wheeler transform), 1000 pr.
by, in pseudocode, 22

cache, 27, 301, 440, 802
cache block, 301, 440, 802
cache hit, 440, 803
cache line, *see* cache block
cache miss, 440, 803
cache obliviousness, 519
caching
  offline, 440–446
  online, 802–815
call
  in a parallel computation, 753
  of a subroutine, 26, 29 n.
  by value, 23
cancellation lemma, 886
cancellation of flow, 679
capacity
  of a cut, 682
  of an edge, 671
  residual, 677, 681
  of a vertex, 676 ex.
capacity constraint, 672
cardinality of a set (| |), 1156
Carmichael number, 945, 953 ex.
Cartesian product (×), 1157
Cartesian sum, 885 ex.
Catalan numbers, 329 pr., 375
CBC-MAC, 291, 306
c-competitive, 793
ceiling function (⌈ ⌉), 63
  in recurrences, 116–117
ceiling instruction, 26
center of a cluster, 1008
centralized scheduler, 759
centroid of a cluster, 1009
certain event, 1185

certificate
  in a cryptosystem, 942
  for verification algorithms, 1058
CHAINED-HASH-DELETE, 278
CHAINED-HASH-INSERT, 278
CHAINED-HASH-SEARCH, 278
chaining, 277–281, 308 pr.
changing variables, to solve a recurrence, 120 pr.
character code, 431
character data type, 26
chess-playing program, 768–769
child
  in a binary tree, 1173
  in a parallel computation, 753
  in a rooted tree, 1172
Chinese remainder theorem, 928–931
chirp transform, 893 ex.
choose $\binom{n}{k}$, 1180
chord, 486 ex.
Cilk, 750, 790
ciphertext, 938
circuit
  boolean combinational, 1065
  depth of, 894
  for fast Fourier transform, 894–897
CIRCUIT-SAT, 1067
circuit satisfiability, 1064–1071
circular, doubly linked list with a sentinel, 262
circular linked list, 259
class
  complexity, 1054
  equivalence, 1159
classification of edges
  in breadth-first search, 581 pr.
  in depth-first search, 569–570, 571 ex.
clause, 1075–1076
clean area, 222 pr.
climate change, 845
clique, 1081
CLIQUE, 1081
clique problem
  approximation algorithm for, 1131 pr.
  NP-completeness of, 1081–1084
closed convex body, 1032
closed interval ($[a, b]$), 1157
closed semiring, 669
closest-point heuristic, 1115 ex.
closure
  group property, 917
  of a language (*), 1052
  transitive, *see* transitive closure
cluster, 1008
  for parallel computing, 748
clustering, 1005–1013
  Lloyd's procedure for, 1011–1013
  primary, 303
CNF (conjunctive normal form), 1043, 1076
CNF satisfiability, 1124 ex.
coarsening leaves of recursion
  in merge sort, 45 pr.
  in quicksort, 198 ex.
  when recursively spawning, 764
code, 431–432
  Huffman, 431–439
codeword, 432
codomain, 1161
coefficient
  binomial, 1181
  of a polynomial, 65, 877
coefficient representation, 879
  and fast multiplication, 882–884
cofactor, 1221
coin changing, 446 pr.
coin flipping, 131–132
collection of sets, 1156
collision, 275
  resolution by chaining, 277–281
  resolution by open addressing, 293–301
collision-resistant hash function, 941
coloring, 425 ex., 1100 pr., 1176 pr.
color, of a red-black-tree node, 331
column-major order, 222 pr., 253
column rank, 1220
columnsort, 222 pr.
column vector, 1215
combination, 1180
combinational circuit, 1065
combinational element, 1065
combine step, in divide-and-conquer, 34, 76
comment, in pseudocode (//), 22
commodity, 864
common divisor, 906
  greatest, *see* greatest common divisor
common multiple, 916 ex.
common subexpression, 894

common subsequence, 394
longest, 393–399
commutative laws for sets, 1154
commutative operation, 917
compact list, 269 pr.
COMPACT-LIST-SEARCH, 269 pr.
COMPACT-LIST-SEARCH′, 270 pr.
COMPARE-EXCHANGE, 222 pr.
COMPARE-EXCHANGE-INSERTION-SORT, 223 pr.
compare-exchange operation, 222 pr.
comparison sort, 205
and binary search trees, 315 ex.
randomized, 219 pr.
and selection, 241
compatible activities, 418
compatible matrices, 1218
competitive analysis, 792
competitive ratio, 793
expected, 808
unbounded, 804
complement
of an event, 1186
of a graph, 1085
of a language, 1052
Schur, 825, 839
of a set, 1155
complementary slackness, 873 pr.
complete graph, 1167
bipartite, 716
complete k-ary tree, 1174
see also heap
completeness of a language, 1072 ex.
complete step, 759
completion time, 446 pr., 816 pr., 1133 pr.
COMPLETION-TIME-SCHEDULE, 817 pr.
complexity class, 1054
co-NP, 1059
NP, 1043, 1058, 1060 ex.
NPC, 1044, 1063
P, 1043, 1050, 1054, 1055 ex.
complexity measure, 1054
complex numbers
inverting matrices of, 838 ex.
multiplication of, 90 ex.
complex root of unity, 885
interpolation at, 891–892
component
biconnected, 582 pr.
connected, 1166
strongly connected, 1166
component graph, 576
composite number, 905
witness to, 946
composition
of logarithms, 66
of parallel traces, 762 fig.
compression
by Huffman code, 431–439
of images, 412 pr.
compulsory miss, 440
computational depth, *see* span
computational problem, 5–6
computation dag, 754 n.
COMPUTE-LCP, 993
COMPUTE-PREFIX-FUNCTION, 978
COMPUTE-SUFFIX-ARRAY, 988
COMPUTE-TRANSITION-FUNCTION, 974
concatenation
of languages, 1052
operator (‖), 291
of strings, 959
concrete problem, 1049
conditional branch instruction, 26
conditional independence, 1190 ex.
conditional probability, 1187, 1189
configuration, 1068
conjugate of the golden ratio ($\hat{\phi}$), 69, 70 ex.
conjugate transpose, 838 ex.
conjunctive normal form, 1043, 1076
connected component, 1166
identified using depth-first search, 572 ex.
identified using disjoint-set data structures, 521–523
CONNECTED-COMPONENTS, 522
connected graph, 1166
connective, 1073
co-NP (complexity class), 1059
conquer step, in divide-and-conquer, 34, 76
conservation of flow, 672
consistency
of literals, 1082
sequential, 756
constrained gradient descent, 1032–1034

constraint
difference, 627
equality, 632 ex.
linear, 851, 853–854
nonnegativity, 854
constraint graph, 628–630
contain, in a path, 1165
continuous master theorem, 112
proof of, 107–115
continuous uniform probability distribution, 1187
contraction
of a dynamic table, 465–470
of an undirected graph by an edge, 1168
contraction algorithm, 701 pr.
control instructions, 26
convergence property, 611, 634–635
convergent series, 1140
converting binary to decimal, 910 ex.
convex body, 1032
convex function, 1025–1027, 1194
α-strongly convex, 1041
convex set, 675 ex.
convolution ($\otimes$), 880
convolution theorem, 892
copy instruction, 26
correctness of an algorithm, 6
corresponding flow network for bipartite matching, 694
countably infinite set, 1156
counter, *see* binary counter
counting, 1178–1184
probabilistic, 153 pr.
counting sort, 208–211
in computing suffix arrays, 992
in radix sort, 213
COUNTING-SORT, 209
coupon collector's problem, 144
cover
path, 698 pr.
by a subfamily, 1116
vertex, 1084, 1106
credit, 453
critical edge, 690
critical path
in a dag, 619
of a PERT chart, 617
of a task-parallel trace, 757
cross a cut, 587, 701 pr.
cross edge, 569
cryptographic hash function, 291
cryptosystem, 936–942
cubic spline, 847 pr.
currency exchange, 641 pr.
curve fitting, 841–845
cut
capacity of, 682
of a flow network, 682–685
global, 701 pr.
minimum, 682
net flow across, 682
of an undirected graph, 587
weight of, 1124 ex.
CUT-ROD, 366
cycle of a graph, 1165–1166
hamiltonian, 1043, 1056, 1085–1090
minimum mean-weight, 642 pr.
negative-weight, *see* negative-weight cycle
and shortest paths, 607–608
cycle cover, 741 pr.
cyclic group, 932

dag, *see* directed acyclic graph
DAG-SHORTEST-PATHS, 617
d-ary heap, 179 pr.
in shortest-paths algorithms, 668 pr.
data-movement instructions, 26
data-parallel model, 789
data science, 14–15
data structure, 9, 249–359, 477–545
AA-trees, 358
augmentation of, 480–496
AVL trees, 357 pr., 358
binary search trees, 312–330
bit vectors, 274 ex.
B-trees, 497–519
deques, 258 ex.
dictionaries, 249
direct-address tables, 273–275
for disjoint sets, 520–545
for dynamic graphs, 479
dynamic sets, 249–251
dynamic trees, 478
exponential search trees, 226, 478
Fibonacci heaps, 478
fusion trees, 226, 478

data structure, *continued*
hash tables, 275–282
heaps, 161–181
interval trees, 489–495
k-neighbor trees, 358
left-leaning red-black binary search trees, 358
linked lists, 258–264
order-statistic trees, 480–486
persistent, 355 pr., 478
potential of, 456
priority queues, 172–178
queues, 254, 256–257
radix trees, 327 pr.
red-black trees, 331–359
rooted trees, 265–268
scapegoat trees, 358
on secondary storage, 498–501
skip lists, 359
splay trees, 359, 478
stacks, 254–255
treaps, 358
2-3-4 trees, 502, 518 pr.
2-3 trees, 358, 519
van Emde Boas trees, 478
weight-balanced trees, 358
data type, 26
decision by an algorithm, 1053
decision problem, 1045, 1049
and optimization problems, 1045
decision tree, 206–207, 219 pr.
decision variable, 851
DECREASE-KEY, 173
decrementing, 22
decryption, 936
default vertex labeling, 724
degree
minimum, of a B-tree, 502
of a node, 1173
of a polynomial, 65, 877
of a vertex, 1165
degree-bound, 877
DELETE, 250
DELETE-LARGER-HALF, 460 ex.
deletion
from binary search trees, 322–325, 326 ex.
from B-trees, 513–516
from chained hash tables, 278
from direct-address tables, 274
from dynamic tables, 465–470
from hash tables with linear probing, 302–303
from heaps, 178 ex.
from interval trees, 491
from linked lists, 261
from open-address hash tables, 294–295
from order-statistic trees, 484–485
from queues, 256
from red-black trees, 346–355
from stacks, 254
DeMorgan's laws
for propositional logic, 1078
for sets, 1155, 1158 ex.
dense graph, 549
ϵ-dense, 668 pr.
dense matrix, 81
density
of prime numbers, 943
of a rod, 372 ex.
dependence
and indicator random variables, 131
linear, 1220
see also independence
depth
average, of a node in a randomly built binary search tree, 328 pr.
of a circuit, 894
of a node in a rooted tree, 1173
of quicksort recursion tree, 191 ex.
of a stack, 202 pr.
depth-determination problem, 542 pr.
depth-first forest, 564
depth-first search, 563–572
in finding articulation points, bridges, and biconnected components, 582 pr.
in finding strongly connected components, 576–581
in the Hopcroft-Karp algorithm, 711
in topological sorting, 573–576
depth-first tree, 564
deque, 258 ex.
DEQUEUE, 257
derivative of a series, 1142
descendant, 1172
destination vertex, 605
det, 1221

determinacy race, 765–768
determinant, 1221
deterministic algorithm, 135
  parallel, 765
DETERMINISTIC-SEARCH, 154 pr.
DFS, 565
  see also depth-first search
DFS-VISIT, 565
DFT, 888
diagonal matrix, 1215
diameter
  of a network, 646
  of a tree, 563 ex.
dictionary, 249
difference
  of sets (−), 1154
  symmetric, 706
difference constraints, 626–632
differentiation of a series, 1142
digital signature, 938
digraph, *see* directed graph
DIJKSTRA, 620
Dijkstra's algorithm, 620–626
  for all-pairs shortest paths, 646, 666
  with edge weights in a range, 626 ex.
  implemented with a Fibonacci heap, 623–624
  implemented with a min-heap, 623
  with integer edge weights, 625–626 ex.
  in Johnson's algorithm, 664
  similarity to breadth-first search, 624, 625 ex.
  similarity to Prim's algorithm, 624
d-independent family of hash functions, 288
DIRECT-ADDRESS-DELETE, 274
direct addressing, 273–275
DIRECT-ADDRESS-INSERT, 274
DIRECT-ADDRESS-SEARCH, 274
direct-address table, 273–275
directed acyclic graph (dag), 1167
  and back edges, 573
  and component graphs, 578
  and hamiltonian paths, 1060 ex.
  longest simple path in, 407 pr.
  for representing a parallel computation, 754
  single-source shortest-paths algorithm for, 616–619
  topological sort of, 573–576
directed equality subgraph, 727
directed graph, 1164
  all-pairs shortest paths in, 646–669
  constraint graph, 628
  Euler tour of, 583 pr., 1043
  hamiltonian cycle of, 1043
  incidence matrix of, 553 ex.
  and longest paths, 1042
  path cover of, 698 pr.
  PERT chart, 617, 619 ex.
  semiconnected, 581 ex.
  shortest path in, 604
  single-source shortest paths in, 604–645
  singly connected, 572 ex.
  square of, 553 ex.
  transitive closure of, 659
  transpose of, 553 ex.
  universal sink in, 553 ex.
  see also directed acyclic graph, graph, network
directed version of an undirected graph, 1166
dirty area, 222 pr.
discovered vertex, 554, 564
discovery time, 565
discrete Fourier transform, 888
discrete logarithm, 933
discrete logarithm theorem, 933
discrete probability distribution, 1186
discrete random variable, 1191–1196
disjoint-set data structure, 520–545
  analysis of, 534–540
  in connected components, 521–523
  in depth determination, 542 pr.
  disjoint-set-forest implementation of, 527–531
  in Kruskal's algorithm, 593
  linear-time special case of, 545
  linked-list implementation of, 523–527
  lower bound for, 545
  in offline lowest common ancestors, 543 pr.
  in offline minimum, 541 pr.
disjoint-set forest, 527–531
  analysis of, 534–540
  rank properties of, 533–534, 540 ex.
  see also disjoint-set data structure
disjoint sets, 1156
disjunctive normal form, 1078

disk drive, 498–500
  see also secondary storage
DISK-READ, 500
DISK-WRITE, 500
dissimilarity, 1006
distance
  edit, 409 pr.
  Manhattan, 244 pr.
  of a shortest path (δ), 558
distributed memory, 748
distribution
  binomial, *see* binomial distribution
  continuous uniform, 1187
  discrete, 1186
  geometric, *see* geometric distribution
  of inputs, 128, 134
  of prime numbers, 943
  probability, 218 ex., 1185
  uniform, 1186
distributive laws for sets, 1155
divergent series, 1140
divide-and-conquer method, 34, 76
  analysis of, 39–41, 90–119
  for binary search, 44 ex.
  for conversion of binary to decimal, 910 ex.
  for fast Fourier transform, 888–891, 895
  for matrix inversion, 834–837
  for matrix multiplication, 81–90, 770–775, 783 pr.
  for merge sort, 34–44, 775–782
  for multiplication, 899 pr.
  for quicksort, 182–204
  relation to dynamic programming, 362
  for selection, 230–243
  solving recurrences for, 90–119
  for Strassen's algorithm, 85–90, 773–774
divide instruction, 26
divides relation (|), 904
divide step, in divide-and-conquer, 34, 76
division method, 284, 292 ex.
division theorem, 905
divisor, 904
  common, 906
  see also greatest common divisor
DNA, 6, 393–394, 409 pr.
DNF (disjunctive normal form), 1078
does-not-divide relation ($\nmid$), 904
Dog River, 717
dolphins, allowing to vote, 850
domain, 1161
double hashing, 295–297, 301 ex.
doubly linked list, 258–259, 264 ex.
  circular, with a sentinel, 262
downto, in pseudocode, 22
d-regular graph, 716 ex., 740 pr.
driving function, 101
duality, 734, 866–873, 874 pr.
  weak, 868–869, 874 pr.
dual linear program, 866
dummy key, 400
dynamic graph, 523
  all-pairs shortest paths algorithms for, 669
  data structures for, 479
  minimum-spanning-tree algorithm for, 599 ex.
  transitive closure of, 667 pr., 669
dynamic graph algorithm, 817
dynamic multiset, 460 ex.
dynamic order statistics, 480–486
dynamic-programming method, 362–416
  for activity selection, 424 ex.
  for all-pairs shortest paths, 648–659
  for bitonic euclidean traveling-salesperson problem, 407 pr.
  bottom-up, 368
  for breaking a string, 412 pr.
  compared with greedy method, 384–385, 393 ex., 421, 426–430
  for edit distance, 409 pr.
  elements of, 382–393
  for Floyd-Warshall algorithm, 655–659
  for inventory planning, 414 pr.
  for longest common subsequence, 393–399
  for longest palindrome subsequence, 407 pr.
  for longest simple path in a weighted directed acyclic graph, 407 pr.
  for matrix-chain multiplication, 373–382
  and memoization, 390–392
  for optimal binary search trees, 400–407
  optimal substructure in, 382–387
  overlapping subproblems in, 387–390
  for printing neatly, 408 pr.
  reconstructing an optimal solution in, 390
  relation to divide-and-conquer, 362
  for rod cutting, 363–373
  for seam carving, 412 pr.

dynamic-programming method, *continued*
for signing free agents, 414 pr.
top-down with memoization, 368
for transitive closure, 659–661
for Viterbi algorithm, 411 pr.
for the 0-1 knapsack problem, 430 ex.
dynamic set, 249–251
see also data structure
dynamic table, 460–471
analyzed by accounting method, 463
analyzed by aggregate analysis, 462–463
analyzed by potential method, 463–470
load factor of, 461
dynamic tree, 478

E [ ], *see* expected value
e (base of the natural logarithm), 65
edge, 1164
antiparallel, 673–674
attributes of, 552
back, 569
bridge, 582 pr.
capacity of, 671
classification in breadth-first search, 581 pr.
classification in depth-first search, 569–570, 571 ex.
critical, 690
cross, 569
forward, 569
light, 587
negative-weight, 606–607
residual, 678
safe, 587
tree, 561, 564, 569
weight of, 551
edge connectivity, 692 ex.
edge set, 1164
edit distance, 409 pr.
Edmonds-Karp algorithm, 689–691
elementary event, 1185
elementary insertion, 461
element of a set ($\in$), 1153
ellipsoid algorithm, 857
elliptic-curve factorization method, 956
elseif, in pseudocode, 22 n.
else, in pseudocode, 22
empty language ($\emptyset$), 1052
empty set ($\emptyset$), 1153
empty set laws, 1154
empty stack, 255
empty string (ε), 959, 1052
empty tree, 1173
encoding of problem instances, 1049–1052
encryption, 936
endpoint of an interval, 489
ENQUEUE, 257
entering a vertex, 1164
entropy function, 1182
epoch, 805
ϵ-dense graph, 668 pr.
ϵ-universal family of hash functions, 287, 292 ex.
equality
of functions, 1162
linear, 853
of sets, 1153
equality constraint, 632 ex.
equality subgraph, 724
directed, 727
equations and asymptotic notation, 58–59
equivalence class, 1159
modulo n ($[a]_n$), 905
equivalence, modular ($= (\text{mod } n)$), 64
equivalence relation, 1159
error bound, 1027
error, in pseudocode, 24
escape problem, 697 pr.
EUCLID, 912
Euclid's algorithm, 911–916, 954 pr.
euclidean norm ($\| \ \|$), 1219
Euler's constant, 921
Euler's phi function, 920
Euler's theorem, 932, 953 ex.
Euler tour, 583 pr., 740 pr.
and hamiltonian cycles, 1043
evaluation of a polynomial, 46 pr., 879, 884 ex.
derivatives of, 900 pr.
at multiple points, 900 pr.
event, 1185
event-driven simulation, 173, 181
EXACT-SUBSET-SUM, 1125
example, in clustering, 1006
exclusion and inclusion, 1158 ex.
execute a subroutine, 29 n.
expansion of a dynamic table, 461–465
expectation, *see* expected value

expected competitive ratio, 808
expected running time, 32, 129
expected value, 1192–1194
of a binomial distribution, 1198
of a geometric distribution, 1197
of an indicator random variable, 130
explored edge, 565
exponential function, 65–66
exponential search tree, 226, 478
exponentiation
of logarithms, 66
modular, 934–935
exponentiation instruction, 27
EXTENDED-BOTTOM-UP-CUT-ROD, 372
EXTENDED-EUCLID, 914
EXTEND-SHORTEST-PATHS, 650
external node, 1172
external path length, 1175 ex.
extracting the maximum key
from d-ary heaps, 179 pr.
from max-heaps, 174
extracting the minimum key
from Young tableaus, 179 pr.
EXTRACT-MAX, 173–174
EXTRACT-MIN, 173

factor, 904
twiddle, 891
factorial function (!), 67–68
factorization, 956
unique, 909
failure, in a Bernoulli trial, 1196
fair coin, 1186
family of hash functions, 286–288, 292 ex.
fan-out, 1066
Farkas's lemma, 869
FASTER-APSP, 653, 655 ex.
fast Fourier transform (FFT), 877–902
circuit for, 894–897
multidimensional, 899 pr.
recursive implementation of, 888–891
using modular arithmetic, 901 pr.
feasibility problem, 627, 873 pr.
feasible linear program, 854
feasible region, 854
feasible solution, 627, 854
feasible vertex labeling, 724, 742 pr.
feature vector, 1006
Fermat's theorem, 932
FFT, 890
see also fast Fourier transform
FFTW, 902
FIB, 751
Fibonacci heap, 478
in Dijkstra's algorithm, 623–624
in Johnson's algorithm, 666
in Prim's algorithm, 597
Fibonacci numbers, 69, 70 ex., 121 pr.
computation of, 750–753, 954 pr.
FIFO, *see* first-in, first-out; queue
final-state function, 968
FIND-AUGMENTING-PATH, 738
FIND-DEPTH, 542 pr.
find path, 528
FIND-POM, 496 pr.
FIND-SET, 521
disjoint-set-forest implementation of, 530, 544
linked-list implementation of, 523
FIND-SPLIT-POINT, 778
finished vertex, 564
finish time
in activity selection, 418
in depth-first search, 565
and strongly connected components, 578
finite automaton, 967–975
FINITE-AUTOMATON-MATCHER, 971
finite group, 917
finite sequence, 1162
finite set, 1156
finite sum, 1140
first-fit heuristic, 1131 pr.
first-in, first-out (FIFO), 254, 803–804, 814 ex.
implemented with a priority queue, 178 ex.
see also queue
fixed-length code, 432
floating-point data type, 26
floor function ($\lfloor\ \rfloor$), 63
in recurrences, 116–117
floor instruction, 26
flow, 671–676
aggregate, 864
augmentation of, 678
blocking, 702
cancellation of, 679
integer-valued, 695

flow, *continued*
net, across a cut, 682
value of, 672
flow conservation, 672
flow network, 671–676
corresponding to a bipartite graph, 694
cut of, 682–685
with multiple sources and sinks, 674
FLOYD-WARSHALL, 657
FLOYD-WARSHALL′, 661 ex.
Floyd-Warshall algorithm, 655–659, 661–662 ex.
flying cars, highways for, 850
FORD-FULKERSON, 686
Ford-Fulkerson method, 676–693
FORD-FULKERSON-METHOD, 676
FORESEE, 797
forest, 1167, 1169
breadth-first, 728
depth-first, 564
disjoint-set, 527–531
for, in pseudocode, 22
and loop invariants, 21 n.
fork-join parallelism, 749–770
see also parallel algorithm
fork-join scheduling, 759–761, 769 ex.
formal power series, 121 pr.
formula satisfiability, 1073–1076
forward edge, 569
forward substitution, 822–823
Fourier transform, *see* discrete Fourier transform, fast Fourier transform
fractional knapsack problem, 429
fractional matching, 741 pr.
free tree, 1167, 1169–1171
frequency count, 802 ex.
frequency domain, 877
full binary tree, 1174
relation to optimal code, 433
full node, 502
full rank, 1220
full walk of a tree, 1112
fully parenthesized matrix-chain product, 374
fully polynomial-time approximation scheme, 1105
for subset sum, 1124–1130
function, 1161–1163
Ackermann's, 544
α-strongly convex, 1041
basis, 841
β-smooth, 1041
boolean, 1182 ex.
convex, 1025–1027, 1194
driving, 101
final-state, 968
hash, *see* hash function
iterated, 68, 74 pr.
linear, 30, 853
objective, 626, 852, 854
potential, 456
prefix, 975–977
probability distribution, 218 ex.
quadratic, 31
reduction, 1062
suffix, 968
transition, 967, 973–974
watershed, 103
functional iteration, 68
fundamental theorem of linear programming, 872
furthest-in-future, 441
fusion tree, 226, 478
fuzzy sorting, 203 pr.

Gabow's scaling algorithm for single-source shortest paths, 641 pr.
gadget, 1086, 1097
GALE-SHAPLEY, 719
Gale-Shapley algorithm, 718–722
Galois field of two elements ($GF(2)$), 1224 pr.
gap character, 961 ex., 975 ex.
gate, 1065
Gaussian elimination, 825
gcd, *see* greatest common divisor
GCD recursion theorem, 911
general arithmetic series, 1141
general number-field sieve, 956
generating function, 121 pr.
generation of partitioned sets, 234
generator
of a subgroup, 922
of $\mathbb{Z}_n^*$, 932

GENERIC-MST, 587
geometric distribution, 1196–1198
  and balls and bins, 143–144
  in finding prime numbers, 943
  in randomized selection analysis, 232
geometric series, 1142
$GF(2)$ (Galois field of two elements), 1224 pr.
global cut, 701 pr.
global minimizer, 1022, 1024 fig., 1026 fig.
global variable, 22
golden ratio (ϕ), 69, 70 ex.
gossiping, 475
gradient descent, 1022–1038
  constrained, 1032–1034
  in machine learning, 1035–1037
  for solving systems of linear equations, 1034–1035
  stochastic, 1040 pr.
  unconstrained, 1023–1031
GRADIENT-DESCENT, 1025
GRADIENT-DESCENT-CONSTRAINED, 1032
gradient of a function, 1023
GRAFT, 542 pr.
grain size in a parallel algorithm, 783 pr.
graph, 1164–1169
  adjacency-list representation of, 550–551
  adjacency-matrix representation of, 551–552
  and asymptotic notation, 548
  attributes of, 548, 552
  breadth-first search of, 554–563
  coloring of, 1100 pr.
  complement of, 1085
  component, 576
  constraint, 628–630
  dense, 549
  depth-first search of, 563–572
  dynamic, 523, 817
  ϵ-dense, 668 pr.
  hamiltonian, 1056
  interval, 425 ex.
  matching in, 693–697, 704–743
  nonhamiltonian, 1056
  planar, 584 pr.
  regular, 716 ex., 740 pr.
  shortest path in, 558
  singly connected, 572 ex.
  sparse, 549
  static, 522
  subproblem, 370–371
  tour of, 1090
  weighted, 551
  see also directed acyclic graph, directed graph, flow network, undirected graph, tree
GRAPH-ISOMORPHISM, 1060 ex.
Gray code, 471 pr.
gray vertex, 554, 564
greatest common divisor (gcd), 906–907, 910 ex.
  binary gcd algorithm for, 953 pr.
  Euclid's algorithm for, 911–916, 954 pr.
  with more than two arguments, 916 ex.
  recursion theorem for, 911
GREEDY-ACTIVITY-SELECTOR, 424
GREEDY-BIPARTITE-MATCHING, 726
greedy-choice property, 427–428
  of activity selection, 420–421
  of Huffman codes, 436–437
  of offline caching, 442–445
greedy method, 417–447
  for activity selection, 418–425
  for coin changing, 446 pr.
  compared with dynamic programming, 384–385, 393 ex., 421, 426–430
  Dijkstra's algorithm, 620–626
  elements of, 426–431
  for the fractional knapsack problem, 429
  greedy-choice property in, 427–428
  for Huffman code, 431–439
  Kruskal's algorithm, 592–594
  for maximal bipartite matching, 726
  for minimum spanning tree, 591–599
  for offline caching, 440–446
  optimal substructure in, 428
  Prim's algorithm, 594–597
  for set cover, 1115–1119
  for task-parallel scheduling, 759–761, 769 ex.
  for task scheduling, 446 pr.
  for weighted set cover, 1132 pr.
greedy scheduler, 759
GREEDY-SET-COVER, 1117
grid, 697 pr.
group, 916–923
  cyclic, 932
  operator ($\oplus$), 917

growth step, 736
guessing the solution, in the substitution method, 92

Habanero-Java, 750
half 3-CNF satisfiability, 1099 ex.
half-open interval ($[a, b)$ or $(a, b]$), 1157
Hall's theorem, 715 ex.
halting, 6
halting problem, 1042
halving lemma, 887
HAM-CYCLE, 1056
hamiltonian cycle, 1043, 1056
  NP completeness of, 1085–1090
hamiltonian graph, 1056
hamiltonian path, 1060 ex., 1098 ex.
HAM-PATH, 1060 ex.
handle, 173
handshaking lemma, 1168 ex.
harmonic number, 1142, 1149
harmonic series, 1142, 1149
HASH-DELETE, 300 ex.
hash function, 275, 282–292
  auxiliary, 295
  collision-resistant, 941
  cryptographic, 291
  division method for, 284, 292 ex.
  for hierarchical memory, 304–307
  multiplication method for, 284–286
  multiply-shift method for, 285–286
  random, 286–290
  static, 282, 284–286
  universal, 286–290
  wee, 305–307
  see also family of hash functions
hashing, 272–311
  with chaining, 277–281, 308 pr.
  double, 295–297, 301 ex.
  independent uniform, 276
  with linear probing, 297, 302–304
  in memoization, 368, 391
  with open addressing, 293–301, 308 pr.
  perfect, 310
  random, 286–290
  to replace adjacency lists, 553 ex.
  of static sets, 308 pr.
  of strings, 290–291, 292 ex.
  uniform, 278
  universal, 286–290, 309 pr.
  of variable-length inputs, 290–291
  of vectors, 290–291
HASH-INSERT, 294, 300 ex.
HASH-SEARCH, 294, 300 ex.
hash table, 275–282
  dynamic, 470 ex.
  used within a priority queue, 174
  see also hashing
hash value, 275
hat-check problem, 134 ex.
head
  in a disk drive, 498
  of a linked list, 259
  of a queue, 256
heap, 161–181
  analyzed by potential method, 459 ex.
  building, 167–170, 178 pr.
  in constructing Huffman codes, 436
  d-ary, 179 pr., 668 pr.
  deletion from, 178 ex.
  in Dijkstra's algorithm, 623
  extracting the maximum key from, 174
  Fibonacci, 478
  height of, 163
  increasing a key in, 174–175
  insertion into, 175
  in Johnson's algorithm, 666
  max-heap, 162
  maximum key of, 174
  mergeable, 268 pr.
  min-heap, 163
  in Prim's algorithm, 597
  as a priority queue, 172–178
HEAP-DECREASE-KEY, 176 ex.
HEAP-EXTRACT-MIN, 176 ex.
HEAP-MINIMUM, 176 ex.
heap property, 162
  maintenance of, 164–167
  vs. binary-search-tree property, 315 ex.
heapsort, 161–181
  lower bound for, 207
HEAPSORT, 170
HEDGE, 1039 pr.
height
  black-, 332
  of a B-tree, 502–504
  of a d-ary heap, 179 pr.

height, *continued*
of a decision tree, 207
of a heap, 163
of a node in a heap, 163, 170 ex.
of a node in a tree, 1173
of a red-black tree, 332
of a tree, 1173
height-balanced tree, 357 pr.
helpful partitioning, 232
Hermitian matrix, 838 ex.
Hessian matrix, 1035
heuristic
first-fit for bin packing, 1131 pr.
path compression, 528
in the Rabin-Karp algorithm, 965
for the set-covering problem, 1116, 1132 pr.
table doubling, 461
for the traveling-salesperson problem, 1115 ex.
union by rank, 528
weighted union, 525
high endpoint of an interval, 489
high side of a partition, 183
HIRE-ASSISTANT, 127
hiring problem, 126–127, 135–136
online, 150–152
probabilistic analysis of, 132–133
hit, 965
HOARE-PARTITION, 199 pr.
HOPCROFT-KARP, 709
Hopcroft-Karp algorithm, 709–715
HORNER, 47 pr.
Horner's rule, 46 pr., 879, 963
HUFFMAN, 434
Huffman code, 431–439
Human Genome Project, 6
HUNGARIAN, 737
Hungarian algorithm, 723–739, 740 pr.
hybrid cryptosystem, 941
hyperedge, 1167
hypergraph, 1167
hypotheses, 1003

ideal parallel computer, 756
idempotency laws, 1154
identity, 917
identity matrix, 1215
identity permutation, 138 ex.
if, in pseudocode, 22
ill-defined recurrence, 77
image, 1162
image compression, 412 pr.
incidence, 1164–1165
incidence matrix
and difference constraints, 628
of a directed graph, 553 ex.
inclusion and exclusion, 1158 ex.
incomplete step, 759
INCREASE-KEY, 173
increasing a key, in a max-heap, 174–175
INCREMENT, 451
incremental design method, 34
incrementing, 21
in-degree, 1165
indentation in pseudocode, 21–22
independence
of events, 1188, 1190 ex.
of random variables, 1192
of subproblems in dynamic programming, 386–387
independent family of hash functions, 288
independent set, 1099 pr.
independent uniform hash function, 276
independent uniform hashing, 276, 278
independent uniform permutation hashing, 295
indexing into an array, 22–23, 26 n., 252
index of an element of $\mathbb{Z}_n^*$, 933
indicator random variable, 130–133
in approximation algorithm for MAX-3-CNF satisfiability, 1120–1121
in birthday paradox analysis, 142–143
in bounding the right tail of the binomial distribution, 1207–1208
in coin flipping analysis, 131–132
expected value of, 130
in hashing analysis, 279–281
in the hat-check problem, 134 ex.
in hiring-problem analysis, 132–133
and linearity of expectation, 131
in quicksort analysis, 197–198, 200 pr.
in randomized caching analysis, 812
in randomized-selection analysis, 245 pr.
in streak analysis, 148–150
induced subgraph, 1166
inequality, linear, 853
infeasible linear program, 854

infeasible solution, 854
inference, 1004
infinite sequence, 1162
infinite set, 1156
infinite sum, 1140
infinity, arithmetic with, 611
initialization of a loop invariant, 20
INITIALIZE-SINGLE-SOURCE, 609
injective function, 1162
inner product, 1219
inorder tree walk, 314, 320 ex.
INORDER-TREE-WALK, 314
in-place permuting, 136
in-place sorting, 158, 220 pr.
in play, 232
input
  to an algorithm, 5
  to a combinational circuit, 1066
  distribution of, 128, 134
  to a logic gate, 1065
  size of, 28
input alphabet, 967
INSERT, 173, 250, 460 ex.
insertion
  into binary search trees, 321–322
  into B-trees, 506–511
  into chained hash tables, 278
  into d-ary heaps, 179 pr.
  into direct-address tables, 274
  into dynamic tables, 461–465
  elementary, 461
  into heaps, 175
  into interval trees, 491
  into linked lists, 260
  into open-address hash tables, 293–294
  into order-statistic trees, 484
  into queues, 256
  into red-black trees, 338–346
  into stacks, 254
  into Young tableaus, 179 pr.
insertion sort, 17–21, 29–31, 51–53, 56–57
  in bucket sort, 216–218
  compared with merge sort, 12–13, 15 ex.
  compared with quicksort, 191 ex.
  decision tree for, 206 fig.
  lower bound for, 52–53
  in merge sort, 45 pr.
  in quicksort, 198 ex.
  using binary search, 45 ex.
INSERTION-SORT, 19, 30, 51
instance
  of an abstract problem, 1045, 1049
  of a problem, 6
instructions of the RAM model, 26
integer data type, 26
integer linear programming, 857, 874 pr., 1098 ex.
integers ($\mathbb{Z}$), 1153
integer-valued flow, 695
integrality theorem, 696
integral, to approximate summations, 1150
integration of a series, 1142
interior-point methods, 857
intermediate vertex, 655
internal node, 1172
internal path length, 1175 ex.
interpolation by a cubic spline, 847 pr.
interpolation by a polynomial, 880, 885 ex.
  at complex roots of unity, 891–892
intersection
  of chords, 486 ex.
  of languages, 1052
  of sets ($\cap$), 1154
interval, 489–490, 1157
  fuzzy sorting of, 203 pr.
INTERVAL-DELETE, 490, 496 pr.
interval graph, 425 ex.
INTERVAL-INSERT, 490, 496 pr.
INTERVAL-SEARCH, 490, 492
INTERVAL-SEARCH-EXACTLY, 495 ex.
interval tree, 489–495
interval trichotomy, 490
intractability, 1042
invalid shift, 957
inventory planning, 414 pr.
inverse
  of a bijective function, 1163
  in a group, 917
  of a matrix, 784 pr., 833–837, 1220
  multiplicative, modulo n, 927
inversion
  in an array, 47 pr.
  in linked lists, 798
  in a sequence, 134 ex., 486 ex.

inversion count, 798
inverter, 1065
invertible matrix, 1220
invocation tree, 756
isolated vertex, 1165
isomorphic graphs, 1166
iterated function, 68, 74 pr.
iterated logarithm function ($\lg^*$), 68
ITERATIVE-TREE-SEARCH, 316
iter function, 536

Java Fork-Join Framework, 750
Jensen's inequality, 1194
JOHNSON, 666
Johnson's algorithm, 662–667
joining
  of red-black trees, 356 pr.
  of 2-3-4 trees, 518 pr.
joint probability density function, 1191
Josephus permutation, 496 pr.

Karmarkar's algorithm, 876
Karp's minimum mean-weight cycle algorithm, 642 pr.
k-ary tree, 1174
k-clustering, 1008
k-CNF, 1043
k-coloring, 1100 pr., 1176 pr.
k-combination, 1180
k-conjunctive normal form, 1043
Keeling curve, 845 fig.
key, 17, 157, 173, 249, 283–284
  in a cryptosystem, 936, 939
  dummy, 400
  median, of a B-tree node, 506
keywords, in pseudocode, 21–22, 24
  parallel, 750, 752–754, 762–763
Kleene star (*), 1052
k-means problem, 1008
KMP algorithm, 975–985
KMP-MATCHER, 978
knapsack problem
  decision version, 1096
  fractional, 429
  0-1, 428, 430 ex., 1134 pr.
k-neighbor tree, 358
knot, of a spline, 847 pr.
Knuth-Morris-Pratt algorithm, 975–985
k-permutation, 136, 1180
Kraft inequality, 1176 ex.
Kruskal's algorithm, 592–594
  with integer edge weights, 598 ex.
k-sorted, 221 pr.
k-string, 1179
k-subset, 1156
k-substring, 1179
kth power, 910 ex.

label
  in machine learning, 1003, 1035
  of a vertex, 724, 742 pr.
Lagrange's formula, 881
Lagrange's theorem, 921
Lamé's theorem, 913
language, 1052
  completeness of, 1072 ex.
  proving NP-completeness of, 1072–1073
  verification of, 1058
lasers, sharks with, 850
last-in, first-out (LIFO), 254, 803–804
  implemented with a priority queue, 178 ex.
  see also stack
latency, 499
LCA, 544 pr.
lcm (least common multiple), 916 ex.
LCP, *see* longest common prefix array
LCS, 393–399
LCS-LENGTH, 397
leading submatrix, 839
leaf, 1172
least common multiple, 916 ex.
least frequently used (LFU), 803, 814 ex.
least recently used (LRU), 445 ex., 803–805
least-squares approximation, 841–845, 1035–1037
leaving a vertex, 1164
LEFT, 162
left child, 1173
left-child, right-sibling representation, 265, 268 ex.
left-leaning red-black binary search tree, 358
LEFT-ROTATE, 336, 495 ex.
left rotation, 335
left shift ($\lll$), 305
left subtree, 1173
Legendre symbol $\left(\frac{a}{p}\right)$, 954 pr.

length
  of a cycle, 1165
  of a path, 1165
  of a sequence, 1162
  of a string, 959, 1179
level
  of a function, 532
  of a node in a disjoint-set forest, 535
  of a tree, 1173
lexicographically less than, 327 pr.
lexicographic sorting, 327 pr., 986 n.
LFU (least frequently used), 803, 814 ex.
lg (binary logarithm), 66
$\lg^*$ (iterated logarithm function), 68
$\lg^k$ (exponentiation of logarithms), 66
lg lg (composition of logarithms), 66
LIFO, *see* last-in, first-out; stack
light edge, 587
linear constraint, 853–854
linear dependence, 1220
linear equality, 853
linear equations
  solving modular, 924–928
  solving systems of, 819–833, 1034–1035
  solving tridiagonal systems of, 847 pr.
linear function, 30, 853
linear independence, 1220
linear inequality, 853
linear-inequality feasibility problem, 873 pr.
linearity of expectation, 1192–1193
  and indicator random variables, 131
linearity of summations, 1141
linear order, 1160
linear permutation, 1224 pr.
linear probing, 297, 302–304
LINEAR-PROBING-HASH-DELETE, 303
linear programming, 850–876, 1121–1124
  applications of, 860–866
  duality in, 866–873
  ellipsoid algorithm for, 857
  fundamental theorem of, 872
  integer, 857, 874 pr.
  interior-point methods for, 857
  Karmarkar's algorithm for, 876
  and maximum flow, 862
  and minimum-cost circulation, 875 pr.
  and minimum-cost flow, 862–864
  and multicommodity flow, 864–865
  simplex algorithm for, 857
  and single-pair shortest path, 861
  and single-source shortest paths, 626–632
  standard form for, 854
  see also integer linear programming, 0-1 integer programming
linear-programming relaxation, 1122
linear regression, 1036
linear search, 25 ex.
linear speedup, 758
line search, 1031
LINK, 530
linked list, 258–264
  compact, 269 pr.
  deletion from, 261
  to implement disjoint sets, 523–527
  insertion into, 260
  maintained by an online algorithm, 795–802
  searching, 260, 292 ex.
linking of trees in a disjoint-set forest, 529
list, *see* linked list
LIST-DELETE, 261
LIST-DELETE′, 262
LIST-INSERT, 261
LIST-INSERT′, 263
LIST-PREPEND, 260
LIST-SEARCH, 260
LIST-SEARCH′, 263
literal, 1076
little-oh notation (o), 60
little-omega notation (ω), 61
Lloyd's procedure, 1011–1013, 1039 pr.
ln (natural logarithm), 66
load factor
  of a dynamic table, 461
  of a hash table, 278
load instruction, 26, 756
local minimizer, 1026 fig.
local variable, 22
logarithm function (log), 66–67
  discrete, 933
  iterated ($\lg^*$), 68
logical parallelism, 753
logical right shift ($\ggg$), 285
logic gate, 1065
longest common prefix (*LCP*) array, 986, 992–994
longest common subsequence, 393–399

longest common substring, 995 ex.
longest monotonically increasing subsequence, 399 ex.
longest palindrome subsequence, 407 pr.
LONGEST-PATH, 1055 ex.
LONGEST-PATH-LENGTH, 1055 ex.
longest repeated substring, 987
longest simple cycle, 1098 ex.
longest simple path, 1042
  in an unweighted graph, 385
  in a weighted directed acyclic graph, 407 pr.
lookahead algorithm, 815 ex.
LOOKUP-CHAIN, 391
loop, in pseudocode, 22
  parallel, 762–765
loop invariant, 19–20
  for breadth-first search, 555
  for building a heap, 167
  for counting sort, 211 ex.
  for determining the rank of an element in an order-statistic tree, 483
  and **for** loops, 21 n.
  for the generic minimum-spanning-tree method, 586
  for heapsort, 172 ex.
  for Horner's rule, 46 pr.
  for increasing a key in a heap, 177 ex.
  for insertion sort, 19–20
  for partitioning, 184
  for Prim's algorithm, 597
  for the Rabin-Karp algorithm, 965
  for randomly permuting an array, 137
  for red-black tree insertion, 340
  for string-matching automata, 970, 973
loss function, 1036
low endpoint of an interval, 489
lower bounds
  asymptotic, 55
  on binomial coefficients, 1181, 1184 ex.
  for comparing water jugs, 220 pr.
  for competitive ratios for online caching, 804–806
  for constructing binary search trees, 315 ex.
  for disjoint-set data structures, 545
  for finding the minimum, 228
  for insertion sort, 52–53
  for k-sorting, 221 pr.
  for median finding, 247
  for merging, 222 pr.
  and potential functions, 475
  for simultaneous minimum and maximum, 229 ex.
  for sorting, 205–208, 219 pr., 225
  for streaks, 147–148, 153 ex.
  on summations, 1148, 1150
  for task-parallel computations, 757
  for traveling-salesperson tour, 1110–1113
  for vertex cover, 1108, 1121–1123, 1132 pr.
lower median, 227
lower-triangular matrix, 1216
lowest common ancestor, 543 pr.
low side of a partition, 183
LRU (least recently used), 445 ex., 803–805
LU decomposition, 824–827
  parallel algorithm for, 784 pr.
LU-DECOMPOSITION, 827
LUP decomposition, 821
  computation of, 828–832
  in matrix inversion, 833–834
  and matrix multiplication, 838 ex.
  parallel algorithm for, 784 pr.
  use of, 821–824
LUP-DECOMPOSITION, 830
LUP-SOLVE, 824

machine learning, 14, 1003–1041
main memory, 498
maintenance of a loop invariant, 20
MAKE-RANKS, 988
MAKE-SET, 521
  disjoint-set-forest implementation of, 530
  linked-list implementation of, 523
makespan, 1133 pr.
MAKE-TREE, 542 pr.
Manhattan distance, 244 pr.
MARKING, 807, 815 ex.
Markov's inequality, 1196 ex.
master method for solving a recurrence, 101–107
master recurrence, 101
master theorem, 102
  continuous, 112
  proof of, 107–115
matched vertex, 693, 705

matching, 704–743
bipartite, 693–697, 704–743
fractional, 741 pr.
by Hopcroft-Karp algorithm, 709–715
maximal, 705, 1108, 1132 pr.
maximum, 704–716, 1132 pr.
and maximum flow, 693–697
perfect, 715 ex., 740 pr.
stable, 716
of strings, 957–1002
unstable, 717
matrix, 1214–1226
addition of, 1217
adjacency, 551–552
conjugate transpose of, 838 ex.
dense, 81
determinant of, 1221
diagonal, 1215
Hermitian, 838 ex.
Hessian, 1035
identity, 1215
incidence, 553 ex.
inverse of, 784 pr., 833–837, 1220
lower-triangular, 1216
minor of, 1221
multiplication of, *see* matrix multiplication
negative of, 1217
permutation, 1217
positive-definite, 1222
positive-semidefinite, 1222
predecessor, 647, 655 ex., 657–659, 661 ex.
product of, with a vector, 762–765, 767, 1218
pseudoinverse of, 843
representation of, 253–254
scalar multiple of, 1217
sparse, 81
subtraction of, 1218
symmetric, 1217
symmetric positive-definite, 838–841
transpose of, 1214
tridiagonal, 1216
unit lower-triangular, 1216
unit upper-triangular, 1216
upper-triangular, 1216
Vandermonde, 881, 1223 pr.
matrix-chain multiplication, 373–382
MATRIX-CHAIN-MULTIPLY, 381 ex.
MATRIX-CHAIN-ORDER, 378
matrix multiplication, 80–90, 1218
for all-pairs shortest paths, 648–655, 668–669
divide-and-conquer method for, 81–90, 770–775, 783 pr.
and LUP decomposition, 838 ex.
and matrix inversion, 834–837
Pan's method for, 89 ex.
parallel algorithm for, 770–775, 783 pr.
Strassen's algorithm for, 85–90, 124–125, 773–774
and transitive closure, 838 ex.
MATRIX-MULTIPLY, 81
MATRIX-MULTIPLY-RECURSIVE, 83
matrix-vector multiplication, 762–765, 767, 1218
MAX-CNF satisfiability, 1124 ex.
MAX-CUT problem, 1124 ex.
MAX-FLOW-BY-SCALING, 700 pr.
max-flow min-cut theorem, 684
max-heap, 162
building, 167–170
d-ary, 179 pr.
deletion from, 178 ex.
extracting the maximum key from, 174
in heapsort, 170–172
increasing a key in, 174–175
insertion into, 175
maximum key of, 174
as a max-priority queue, 172–178
mergeable, 268 n.
MAX-HEAP-DECREASE-KEY, 176 ex.
MAX-HEAP-DELETE, 178 ex.
MAX-HEAP-EXTRACT-MAX, 175
MAX-HEAPIFY, 165
MAX-HEAP-INCREASE-KEY, 176
MAX-HEAP-INSERT, 176
building a heap with, 178 pr.
MAX-HEAP-MAXIMUM, 175
max-heap property, 162
maintenance of, 164–167
maximal element, 1160
maximal matching, 705, 1108, 1132 pr.
greedy method for, 726
maximization linear program, 853

maximum, 227
in binary search trees, 317–318
of a binomial distribution, 1202 ex.
finding, 228–229
in heaps, 174
in red-black trees, 334
MAXIMUM, 173–174, 250
maximum bipartite matching, 693–697, 705–716
maximum flow, 670–703
Edmonds-Karp algorithm for, 689–691
Ford-Fulkerson method for, 676–693
as a linear program, 862
and maximum bipartite matching, 693–697
push-relabel algorithms for, 702
scaling algorithm for, 699 pr.
updating, 699 pr.
maximum matching, 693, 704, 1132 pr.
see also maximum bipartite matching
maximum spanning tree, 1134 pr.
max-priority queue, 173
MAX-3-CNF satisfiability, 1120–1121
MAYBE-MST-A, 602 pr.
MAYBE-MST-B, 602 pr.
MAYBE-MST-C, 602 pr.
mean
of a cluster, 1009
see also expected value
mean weight of a cycle, 642 pr.
median, 227–247
weighted, 244 pr.
median key, of a B-tree node, 506
median-of-3 method, 203 pr.
member of a set ($\in$), 1153
memoization, 368, 390–392
MEMOIZED-CUT-ROD, 369
MEMOIZED-CUT-ROD-AUX, 369
MEMOIZED-MATRIX-CHAIN, 391
memory hierarchy, 27, 301
hash functions for, 304–307
MERGE, 36
mergeable heap, 268 pr.
MERGE-LISTS, 1125
merge sort, 34–44, 57
compared with insertion sort, 12–13, 15 ex.
lower bound for, 207
parallel algorithm for, 775–782
use of insertion sort in, 45 pr.
MERGE-SORT, 39
merging
of k sorted lists, 178 ex.
lower bounds for, 222 pr.
parallel algorithm for, 776–780
of two sorted subarrays, 35–38
MILLER-RABIN, 946
Miller-Rabin primality test, 945–953
MIN-GAP, 495 ex.
min-heap, 163
analyzed by potential method, 459 ex.
building, 167–170
in constructing Huffman codes, 436
d-ary, 668 pr.
in Dijkstra's algorithm, 623
in Johnson's algorithm, 666
mergeable, 268 n.
as a min-priority queue, 176 ex.
in Prim's algorithm, 597
MIN-HEAPIFY, 166 ex.
MIN-HEAP-INSERT, 176 ex.
min-heap property, 163
maintenance of, 166 ex.
vs. binary-search-tree property, 315 ex.
minimization linear program, 853
minimizer of a function, 1022, 1024 fig., 1026 fig.
minimum, 227
in binary search trees, 317–318
finding, 228–229
offline, 541 pr.
in red-black trees, 334
MINIMUM, 173, 228, 250
minimum-cost circulation, 875 pr.
minimum-cost flow, 862–864
minimum-cost multicommodity flow, 866 ex.
minimum-cost spanning tree, *see* minimum spanning tree
minimum cut, 682
global, 701 pr.
minimum degree, of a B-tree, 502
minimum mean-weight cycle, 642 pr.
minimum path cover, 698 pr.
minimum spanning tree, 585–603
in approximation algorithm for traveling-salesperson problem, 1110
Borůvka's algorithm for, 603
on dynamic graphs, 599 ex.

minimum spanning tree, *continued*
generic method for, 586–591
Kruskal's algorithm for, 592–594
Prim's algorithm for, 594–597
second-best, 599 pr.
minimum-weight spanning tree, *see* minimum spanning tree
minimum-weight vertex cover, 1121–1124
minor of a matrix, 1221
min-priority queue, 173
in constructing Huffman codes, 434
in Dijkstra's algorithm, 623–624
in Prim's algorithm, 596–597
missing child, 1173
mod, 64, 905
modeling, 851
modifying operation, 250
modular arithmetic, 64, 901 pr., 916–923
modular equivalence ($= (\text{mod } n)$), 64, 905
modular exponentiation, 934–935
MODULAR-EXPONENTIATION, 935
modular linear equations, 924–928
MODULAR-LINEAR-EQUATION-SOLVER, 926
modulo, 64, 905
Monge array, 123 pr.
monotone sequence, 181
monotonically decreasing, 63
monotonically increasing, 63
Monty Hall problem, 1210 pr.
MOVE-TO-FRONT, 796–797
MST-KRUSKAL, 594
MST-PRIM, 596
MST-REDUCE, 601 pr.
much-greater-than ($\gg$), 533
much-less-than ($\ll$), 761
multicommodity flow, 864–865
multicore computer, 748
multidimensional fast Fourier transform, 899 pr.
multigraph, 1167
multiple, 904
of an element modulo n, 924–928
least common, 916 ex.
scalar, 1217
multiple sources and sinks, 674
multiplication
of complex numbers, 90 ex.
divide-and-conquer method for, 899 pr.
of matrices, *see* matrix multiplication
of a matrix chain, 373–382
matrix-vector, 762–765, 767, 1218
modulo n ($\cdot_n$), 917
of polynomials, 878
multiplication method, 284–286
multiplicative group modulo n, 919
multiplicative inverse, modulo n, 927
multiplicative weights, 1015–1022
multiply instruction, 26
multiply-shift method, 285–286
MULTIPOP, 450
multiset, 1153 n.
dynamic, 460 ex.
mutually exclusive events, 1185
mutually independent events, 1188
mutually noninterfering strands, 767

$\mathbb{N}$ (set of natural numbers), 1153
naive algorithm for string matching, 960–962
NAIVE-STRING-MATCHER, 960
National Resident Matching Program, 704, 722 ex.
natural cubic spline, 847 pr.
natural logarithm (ln), 66
natural numbers ($\mathbb{N}$), 1153
keys interpreted as, 283–284
nearest-center rule, 1008
negative of a matrix, 1217
negative-weight cycle
and difference constraints, 629
and relaxation, 639 ex.
and shortest paths, 606–607, 614–615, 655 ex., 662 ex.
negative-weight edges, 606–607
neighbor, 1167
neighborhood, 715 ex.
nesting boxes, 640 pr.
net flow across a cut, 682
network
flow, *see* flow network
residual, 677–681
for sorting, 789

new request, 810
Newton's method, 1038 pr.
NIL, 23
node, 1172
  see also vertex
nondeterministic algorithm, 765
nondeterministic polynomial time, 1058 n.
  see also NP
nonempty suffix, 997 pr.
nonhamiltonian graph, 1056
noninstance, 1051 n.
noninvertible matrix, 1220
nonnegativity constraint, 854
nonoblivious adversary, 807
nonoverlappable string pattern, 974 ex.
nonsample position, 997 pr.
nonsample suffix, 997 pr.
nonsingular matrix, 1220
nontrivial power, 910 ex.
nontrivial square root of 1, modulo n, 934
no-path property, 611, 634
normal equation, 843
norm of a vector ($\| \|$), 1219
NOT function ($\neg$), 1065
not a set member ($\notin$), 1153
not equivalent ($\not\equiv \pmod{n}$), 64
NOT gate, 1065
NP (complexity class), 1043, 1058, 1060 ex.
NPC (complexity class), 1044, 1063
NP-complete, 1044, 1063
NP-completeness, 9–10, 1042–1103
  of the circuit-satisfiability problem, 1064–1071
  of the clique problem, 1081–1084
  of the formula-satisfiability problem, 1073–1076
  of the graph-coloring problem, 1100 pr.
  of the half 3-CNF satisfiability problem, 1099 ex.
  of the hamiltonian-cycle problem, 1085–1090
  of the hamiltonian-path problem, 1098 ex.
  of the independent-set problem, 1099 pr.
  of integer linear programming, 1098 ex.
  of the longest-simple-cycle problem, 1098 ex.
  proving, of a language, 1072–1073
  reduction strategies for, 1095–1098
  of scheduling with profits and deadlines, 1102 pr.
  of the set-covering problem, 1119 ex.
  of the set-partition problem, 1098 ex.
  of the subgraph-isomorphism problem, 1098 ex.
  of the subset-sum problem, 1092–1095
  of the 3-CNF-satisfiability problem, 1076–1079
  of the traveling-salesperson problem, 1090–1092
  of the vertex-cover problem, 1084–1085
  of 0-1 integer programming, 1098 ex.
NP-hard, 1063
n-set, 1156
n-tuple, 1157
null event, 1185
null tree, 1173
null vector, 1221
number-field sieve, 956
numerical stability, 819, 821
n-vector, 1215

o-notation, 60
O-notation, 50, 54–55
O'-notation, 73 pr.
$\widetilde{O}$-notation, 73 pr.
object, 23
objective function, 626, 852, 854
objective value, 854
oblivious adversary, 807
oblivious compare-exchange algorithm, 222 pr.
occurrence of a pattern, 957
offline algorithm, 791
OFFLINE-MINIMUM, 542 pr.
offline problem
  caching, 440–446
  lowest common ancestors, 543 pr.
  minimum, 541 pr.
old request, 810
Omega-notation, 51, 54 fig., 55–56
1-approximation algorithm, 1105
one-pass method, 544
one-to-one correspondence, 1163
one-to-one function, 1162
online algorithm, 791–818
  for caching, 802–815
  for the cow-path problem, 815 pr.

online algorithm, *continued*
for hiring, 150–152
for maintaining a linked list, 795–802
for task scheduling, 816 pr.
for waiting for an elevator, 792–794
online learning, 1003
ONLINE-MAXIMUM, 150
online task-parallel scheduler, 759
onto function, 1162
open-address hash table, 293–301, 308 pr.
with double hashing, 295–297, 301 ex.
with linear probing, 297, 302–304
open interval ((a, b)), 1157
OpenMP, 750
optimal assignment, 723–739
optimal binary search tree, 400–407
OPTIMAL-BST, 405
optimal objective value, 854
optimal solution, 854
optimal substructure, 382–387
of activity selection, 419
of binary search trees, 402–403
of the fractional knapsack problem, 429
in greedy method, 428
of Huffman codes, 438
of longest common subsequences, 394–395
of matrix-chain multiplication, 376
of offline caching, 441–442
of rod cutting, 365
of shortest paths, 605–606, 649, 655–656
of unweighted shortest paths, 385
of the 0-1 knapsack problem, 429
optimal vertex cover, 1106
optimization problem, 362, 1045, 1049
approximation algorithms for, 1104–1136
and decision problems, 1045
OR function ($\vee$), 659, 1065
order
of a group, 922
of growth, 32
linear, 1160
partial, 1160
total, 1160
ordered pair, 1156
ordered tree, 1173
order statistics, 160, 227–247
dynamic, 480–486
order-statistic tree, 480–486
ord function, 987
OR gate, 1065
or, in pseudocode, 24
orthonormal, 849
OS-KEY-RANK, 485 ex.
OS-RANK, 483
OS-SELECT, 482
outcome, 1185
out-degree, 1165
outer product, 1219
output
of an algorithm, 5
of a combinational circuit, 1066
of a logic gate, 1065
overdetermined system of linear equations, 821
overflow
of a queue, 257
of a stack, 255
overflowing vertex, 703
overlapping intervals, 489
finding all, 495 ex.
point of maximum overlap, 496 pr.
overlapping rectangles, 495 ex.
overlapping subproblems, 387–390
overlapping-suffix lemma, 959

P (complexity class), 1043, 1050, 1054, 1055 ex.
page, in virtual memory, 440
pair
blocking, 716
ordered, 1156
pairwise disjoint sets, 1156
pairwise independence, 1188
pairwise relatively prime, 908
palindrome, 407 pr., 995 ex.
Pan's method for matrix multiplication, 89 ex.
parallel algorithm, 748–790
for computing Fibonacci numbers, 750–753
grain size in, 783 pr.
for LU decomposition, 784 pr.
for LUP decomposition, 784 pr.
for matrix inversion, 784 pr.
for matrix multiplication, 770–775, 783 pr.
for matrix-vector product, 762–765, 767
for merge sort, 775–782
for merging, 776–780
for prefix computation, 784 pr.

parallel algorithm, *continued*
for quicksort, 789 pr.
randomized, 789 pr.
for reduction, 784 pr.
for a simple stencil calculation, 787 pr.
for solving systems of linear equations, 784 pr.
Strassen's algorithm, 773–774
for well-formed parentheses, 786 pr.
parallel computer, 10, 748, 756
parallel for, in pseudocode, 762–763
parallelism
logical, 753
of a randomized parallel algorithm, 789 pr.
spawning, 753
syncing, 754
of a task-parallel computation, 758
parallel keywords, 750, 752, 762
parallel loop, 762–765, 783 pr.
parallel-machine-scheduling problem, 1133 pr.
parallel prefix, 784 pr.
parallel random-access machine, 789
parallel slackness, 758
rule of thumb, 761
parallel, strands logically in, 756
parallel trace, 754–756
series-parallel composition of, 762 fig.
parameter, 23
costs of passing, 120 pr.
parent
in a breadth-first tree (π), 555
in a parallel computation, 753
in a rooted tree, 1172
PARENT, 162
parenthesis theorem, 567
parenthesization of a matrix-chain product, 374
Pareto optimality, 722 ex.
parse tree, 1077
partial derivative (∂), 1023
partial order, 1160
PARTITION, 184
PARTITION′, 200 pr.
PARTITION-AROUND, 237
partition function, 363 n.
partitioning, 183–186
around median of 3 elements, 198 ex.
helpful, 232
Hoare's method for, 199 pr.
randomized, 192, 198 ex., 200 pr., 203 pr.
partition of a set, 1156, 1159
Pascal's triangle, 1183 ex.
path, 1165
alternating, 705
augmenting, 681–682, 705
critical, 619
find, 528
hamiltonian, 1060 ex., 1098 ex.
longest, 385, 1042
shortest, *see* shortest paths
simple, 1165
weight of, 407 pr., 604
PATH, 1045, 1053
path compression, 528
path cover, 698 pr.
path length, of a tree, 328 pr., 1175 ex.
path-relaxation property, 611, 635
pattern, 957
nonoverlappable, 974 ex.
pattern matching, *see* string matching
perfect hashing, 310
perfect linear speedup, 758
perfect matching, 715 ex., 740 pr.
permutation, 1163, 1179–1180
bit-reversal, 897
identity, 138 ex.
in place, 136
Josephus, 496 pr.
k-permutation, 136, 1180
linear, 1224 pr.
random, 136–138
uniform random, 128, 136
permutation matrix, 1217
PERMUTE-BY-CYCLE, 139 ex.
PERMUTE-WITH-ALL, 139 ex.
PERMUTE-WITHOUT-IDENTITY, 138 ex.
persistent data structure, 355 pr., 478
PERSISTENT-TREE-INSERT, 355 pr.
PERT chart, 617, 619 ex.
P-FIB, 753
phi function ($\phi(n)$), 920
pivot
in LU decomposition, 826
in quicksort, 183
in selection, 230

$P[:k]$ (prefix of a pattern), 959
planar graph, 584 pr.
platter in a disk drive, 498
P-MATRIX-MULTIPLY, 771
P-MATRIX-MULTIPLY-RECURSIVE, 772
P-MAT-VEC, 763
P-MAT-VEC-RECURSIVE, 763
P-MAT-VEC-WRONG, 768
P-MERGE, 779
P-MERGE-AUX, 779
P-MERGE-SORT, 775
P-NAIVE-MERGE-SORT, 775
pointer, 23
  trailing, 321
point, in clustering, 1006
point-value representation, 880
polylogarithmically bounded, 67
polynomial, 65, 877–885
  addition of, 877
  asymptotic behavior of, 71 pr.
  coefficient representation of, 879
  derivatives of, 900 pr.
  evaluation of, 46 pr., 879, 884 ex., 900 pr.
  interpolation by, 880, 885 ex.
  multiplication of, 878, 882–884, 899 pr.
  point-value representation of, 880
polynomial-growth condition, 116–117
polynomially bounded, 65
polynomially related, 1051
polynomial-time acceptance, 1053
polynomial-time algorithm, 904, 1042
polynomial-time approximation scheme, 1105
  for maximum clique, 1131 pr.
  for subset sum, 1124–1130
polynomial-time computability, 1051
polynomial-time decision, 1053
polynomial-time reducibility ($\leq_P$), 1062, 1071 ex.
polynomial-time solvability, 1050
polynomial-time verification, 1056–1061
POP, 255, 449
pop from a runtime stack, 202 pr.
positional tree, 1174
positive-definite matrix, 1222
positive-semidefinite matrix, 1222
post-office location problem, 244 pr.
postorder tree walk, 314
potential function, 456
  for lower bounds, 475
potential method, 456–460
  for binary counters, 458–459
  for disjoint-set data structures, 534–540, 541 ex.
  for dynamic tables, 463–470
  for maintaining a linked list, 799–801
  for min-heaps, 459 ex.
  for restructuring red-black trees, 473 pr.
  for stack operations, 457–458
potential of a data structure, 456
power
  of an element, modulo n, 932–936
  kth, 910 ex.
  nontrivial, 910 ex.
power series, 121 pr.
power set, 1156
Pr { } (probability distribution), 1185
PRAM, 789
predecessor
  in binary search trees, 318–319
  in breadth-first trees (π), 555
  in linked lists, 259
  in red-black trees, 334
  in shortest-paths trees (π), 608
PREDECESSOR, 250
predecessor matrix, 647, 655 ex., 657–659, 661 ex.
predecessor subgraph
  in all-pairs shortest paths, 647
  in breadth-first search, 561
  in depth-first search, 564
  in single-source shortest paths, 608
predecessor-subgraph property, 611, 637–638
prediction, 1004
prediction phase, 1003
preemption, 446 pr., 816 pr.
prefix
  of a sequence, 395
  of a string ($\sqsubset$), 959
prefix computation, 784 pr.
prefix-free code, 432
prefix function, 975–977
prefix-function iteration lemma, 980
preflow, 703
preimage of a matrix, 1224 pr.

preorder, total, 1160
preorder tree walk, 314
Prim's algorithm, 594–597
  with an adjacency matrix, 598 ex.
  in approximation algorithm for traveling-salesperson problem, 1110
  with integer edge weights, 598 ex.
  similarity to Dijkstra's algorithm, 624
  for sparse graphs, 599 pr.
primality testing, 942–953, 956
  Miller-Rabin test, 945–953
  pseudoprimality testing, 944–945
primal linear program, 866
  augmented, 870
primary clustering, 303
prime distribution function, 943
prime factorization of integers, 909
prime number, 905
  density of, 943
prime number theorem, 943
primitive root of $\mathbb{Z}_n^*$, 932
principal root of unity, 886
principle of inclusion and exclusion, 1158 ex.
PRINT-ALL-PAIRS-SHORTEST-PATH, 648
PRINT-CUT-ROD-SOLUTION, 372
PRINT-LCS, 397
PRINT-OPTIMAL-PARENS, 381
PRINT-PATH, 562
PRINT-SET, 531 ex.
priority queue, 172–178
  in constructing Huffman codes, 434
  in Dijkstra's algorithm, 623–624
  heap implementation of, 172–178
  max-priority queue, 173
  min-priority queue, 173, 176 ex.
  with monotone extractions, 181
  in Prim's algorithm, 596–597
  see also Fibonacci heap
probabilistically checkable proof, 1103, 1136
probabilistic analysis, 127–128, 140–153
  of approximation algorithm for MAX-3-CNF satisfiability, 1120–1121
  and average inputs, 32
  of average node depth in a randomly built binary search tree, 328 pr.
  of balls and bins, 143–144
  of birthday paradox, 140–143
  of bucket sort, 216–218, 218 ex.
  of collisions, 281 ex.
  of file comparison, 967 ex.
  of fuzzy sorting of intervals, 203 pr.
  of hashing with chaining, 278–281
  of hiring problem, 132–133, 150–152
  of insertion into a binary search tree with equal keys, 327 pr.
  of longest probe bound for hashing, 308 pr.
  of lower bound for sorting, 219 pr.
  of Miller-Rabin primality test, 948–953
  of online hiring problem, 150–152
  of open-address hashing, 297–300
  and parallel algorithms, 789 pr.
  of partitioning, 191 ex., 198 ex., 200 pr., 203 pr.
  of probabilistic counting, 153 pr.
  of quicksort, 194–198, 200 pr., 203 pr.
  of Rabin-Karp algorithm, 965–966
  and randomized algorithms, 134–136
  of randomized online caching, 809–814
  of randomized selection, 232–236, 245 pr.
  of randomized weighted majority, 1022 ex.
  of searching a sorted compact list, 269 pr.
  of slot-size bound for chaining, 308 pr.
  of sorting points by distance from origin, 218 ex.
  of streaks, 144–150
  of universal hashing, 286–290
probabilistic counting, 153 pr.
probability, 1184–1191
probability axioms, 1185
probability density function, 1191
probability distribution, 1185
probability distribution function, 218 ex.
probe sequence, 293
probing, 293
  see also linear probing, double hashing
problem
  abstract, 1048
  computational, 5–6
  concrete, 1049
  decision, 1045, 1049
  intractable, 1042
  optimization, 362, 1045, 1049
  solution to, 6, 1049
  tractable, 1042
procedure, 18
  calling, 23, 26, 29 n.

product ($\prod$), 1144
  Cartesian ($\times$), 1157
  inner, 1219
  of matrices, *see* matrix multiplication
  outer, 1219
  of polynomials, 878
  rule of, 1179
  scalar flow, 675 ex.
professional wrestler, 563 ex.
program counter, 1068
programming, *see* dynamic programming, linear programming
projection, 1032
proper ancestor, 1172
proper descendant, 1172
proper prefix, 959
proper subgroup, 921
proper subset ($\subset$), 1154
proper suffix, 959
P-SCAN-1, 785 pr.
P-SCAN-1-AUX, 785 pr.
P-SCAN-2, 786 pr.
P-SCAN-2-AUX, 786 pr.
P-SCAN-3, 787 pr.
P-SCAN-DOWN, 787 pr.
P-SCAN-UP, 787 pr.
pseudocode, 18, 21–24
pseudoinverse, 843
pseudoprime, 944–945
PSEUDOPRIME, 945
pseudorandom-number generator, 129
P-TRANSPOSE, 770 ex.
public key, 936, 939
public-key cryptosystem, 936–942
PUSH, 255, 449
push onto a runtime stack, 202 pr.
push-relabel algorithms, 702

quadratic convergence, 1039 pr.
quadratic function, 31
quadratic residue, 954 pr.
quantile, 242 ex.
query, 250
queue, 254, 256–257
  in breadth-first search, 554
  implemented by stacks, 258 ex., 460 ex.
  linked-list implementation of, 264 ex.
  priority, *see* priority queue
quicksort, 182–204
  analysis of, 187–191, 193–198
  average-case analysis of, 194–198
  compared with insertion sort, 191 ex.
  compared with radix sort, 214
  with equal element values, 200 pr.
  good worst-case implementation of, 241 ex.
  with median-of-3 method, 203 pr.
  parallel algorithm for, 789 pr.
  randomized version of, 191–193, 200 pr., 203 pr.
  stack depth of, 202 pr.
  and tail recursion, 202 pr.
  use of insertion sort in, 198 ex.
  worst-case analysis of, 193–194
QUICKSORT, 183
QUICKSORT′, 200 pr.
quotient, 905

$\mathbb{R}$ (set of real numbers), 1153
Rabin-Karp algorithm, 962–967
RABIN-KARP-MATCHER, 966
race condition, 765–768
RACE-EXAMPLE, 766
radix sort, 211–215
  compared with quicksort, 214
  in computing suffix arrays, 992
RADIX-SORT, 213
radix tree, 327 pr.
RAM, 26–27
RANDOM, 129
random-access machine, 26–27
  parallel, 789
random hashing, 286–290
randomized algorithm, 128–129, 134–140
  and average inputs, 32
  comparison sort, 219 pr.
  for fuzzy sorting of intervals, 203 pr.
  for hiring problem, 135–136
  for insertion into a binary search tree with equal keys, 327 pr.
  for MAX-3-CNF satisfiability, 1120–1121
  Miller-Rabin primality test, 945–953
  for online caching, 807–814
  parallel, 789 pr.
  for partitioning, 192, 198 ex., 200 pr., 203 pr.
  for permuting an array, 136–138
  and probabilistic analysis, 134–136

randomized algorithm, *continued*
quicksort, 191–193, 200 pr., 203 pr.
random hashing, 286–290
randomized rounding, 1136
for searching a sorted compact list, 269 pr.
for selection, 230–236, 245 pr.
universal hashing, 286–290
for weighted majority, 1022 ex.
RANDOMIZED-HIRE-ASSISTANT, 135
RANDOMIZED-MARKING, 808
RANDOMIZED-PARTITION, 192
RANDOMIZED-PARTITION′, 200 pr.
RANDOMIZED-QUICKSORT, 192
relation to randomly built binary search trees, 328 pr.
randomized rounding, 1136
RANDOMIZED-SELECT, 230
randomly built binary search tree, 328 pr.
RANDOMLY-PERMUTE, 136, 138 ex.
random-number generator, 129
random oracle, 276
random permutation, 136–138
uniform, 128, 136
RANDOM-SAMPLE, 139 ex.
RANDOM-SEARCH, 154 pr.
random variable, 1191–1196
indicator, *see* indicator random variable
range, 1162
of a matrix, 1224 pr.
rank
column, 1220
in computing suffix arrays, 987
full, 1220
of a matrix, 1220, 1224 pr.
of a node in a disjoint-set forest, 528, 533–534, 540 ex.
of a number in an ordered set, 480
in order-statistic trees, 482–484, 485–486 ex.
row, 1220
rate of growth, 32
RB-DELETE, 348
RB-DELETE-FIXUP, 351
RB-ENUMERATE, 355 ex.
RB-INSERT, 338
RB-INSERT-FIXUP, 339
RB-JOIN, 356 pr.
RB-TRANSPLANT, 347
RC6, 304
reachability in a graph ($\rightsquigarrow$), 1165
real numbers ($\mathbb{R}$), 1153
reconstructing an optimal solution, in dynamic programming, 390
record, 17, 157
rectangle, 495 ex.
RECTANGULAR-MATRIX-MULTIPLY, 374
recurrence, 39, 76–80, 90–125
Akra-Bazzi, 115–119
algorithmic, 77–78
inequalities in, 78
master, 101
solution by Akra-Bazzi method, 117–118
solution by master method, 101–107
solution by recursion-tree method, 95–101
solution by substitution method, 90–95
recursion, 34
recursion tree, 42, 95–101
in matrix-chain multiplication analysis, 388–390
in merge sort analysis, 42–44
in proof of continuous master theorem, 108–110
in quicksort analysis, 188–190
in rod cutting analysis, 366–367
and the substitution method, 98
RECURSIVE-ACTIVITY-SELECTOR, 422
recursive case, 34
of a divide-and-conquer algorithm, 76
of a recurrence, 77
RECURSIVE-MATRIX-CHAIN, 389
red-black properties, 331–332
red-black tree, 331–359
augmentation of, 487–489
compared with B-trees, 497, 503
deletion from, 346–355
for enumerating keys in a range, 355 ex.
height of, 332
insertion into, 338–346
in interval trees, 490–495
joining of, 356 pr.
left-leaning, 358
maximum key of, 334
minimum key of, 334
in order-statistic trees, 480–486
persistent, 355 pr.
predecessor in, 334
properties of, 331–335

red-black tree, *continued*
  relaxed, 334 ex.
  restructuring, 473 pr.
  rotation in, 335–338
  searching in, 334
  successor in, 334
  see also interval tree, order-statistic tree
REDUCE, 784 pr.
reducibility, 1061–1063
reduction algorithm, 1046, 1062
reduction function, 1062
reduction, of an array, 784 pr.
reduction strategies, 1095–1098
reference, 23
reflexive relation, 1158
reflexivity of asymptotic notation, 61
region, feasible, 854
register, 301, 756
regret, 1016
regular graph, 716 ex., 740 pr.
regularity condition, 103, 112, 114 ex.
regularization, 1012, 1036–1037
reindexing summations, 1143–1144
reinforcement learning, 1004
rejection
  by an algorithm, 1053
  by a finite automaton, 968
relation, 1158–1161
relatively prime, 908
RELAX, 610
relaxation
  of an edge, 609–611
  linear programming, 1122
relaxed red-black tree, 334 ex.
release time, 446 pr., 816 pr.
remainder, 64, 905
remainder instruction, 26
repeated squaring
  for all-pairs shortest paths, 652–653
  for raising a number to a power, 934
repeat, in pseudocode, 22
repetition factor, of a string, 996 pr.
REPETITION-MATCHER, 996 pr.
representative of a set, 520
RESET, 456 ex.
residual capacity, 677, 681
residual edge, 678
residual network, 677–681
residue, 64, 905, 954 pr.
respecting a set of edges, 587
return, in pseudocode, 24
return instruction, 26
reweighting
  in all-pairs shortest paths, 662–664
  in single-source shortest paths, 641 pr.
$\rho(n)$-approximation algorithm, 1104, 1120
RIGHT, 162
right child, 1173
right-conversion, 337 ex.
RIGHT-ROTATE, 336
right rotation, 335
right shift ($\ggg$), 285
right subtree, 1173
rod cutting, 363–373, 393 ex.
root
  of a tree, 1171
  of unity, 885–886
  of $\mathbb{Z}_n^*$, 932
rooted tree, 1171
  representation of, 265–268
rotation, 335–338
rounding, 1122
  randomized, 1136
row-major order, 253, 396
row rank, 1220
row vector, 1215
RSA public-key cryptosystem, 936–942
rule of product, 1179
rule of sum, 1178
running time, 29
  asymptotic, 49
  average-case, 32, 128
  best-case, 34 ex.
  expected, 32, 129
  of a graph algorithm, 548
  order of growth, 32
  parallel, 757–758
  and proper use of asymptotic notation, 56–57
  rate of growth, 32
  worst-case, 31

SA, *see* suffix array
sabermetrics, 415 n.
safe edge, 587
SAME-COMPONENT, 522

sample position, 997 pr.
sample space, 1185
sample suffix, 997 pr.
sampling, 139 ex.
SAT, 1074
satellite data, 17, 157, 249
satisfiability, 1066, 1073–1079, 1120–1121, 1124 ex.
satisfiable formula, 1043, 1074
satisfying assignment, 1066, 1074
scalar, 1217
scalar flow product, 675 ex.
scaling
  in maximum flow, 699 pr.
  in single-source shortest paths, 641 pr.
scan, 784 pr.
SCAN, 785 pr.
scapegoat tree, 358
schedule, 1133 pr.
scheduler for task-parallel computations, 753, 759–761, 769 ex., 789
scheduling, 446 pr., 816 pr., 1102 pr., 1133 pr.
Schur complement, 825, 839
Schur complement lemma, 840
SCRAMBLE-SEARCH, 154 pr.
seam carving, 412 pr.
SEARCH, 250
searching
  binary search, 44 ex., 777–778
  in binary search trees, 316–317
  in B-trees, 504–505
  in chained hash tables, 278
  in direct-address tables, 274
  for an exact interval, 495 ex.
  in interval trees, 492–494
  linear search, 25 ex.
  in linked lists, 260
  in open-address hash tables, 294
  in red-black trees, 334
  in sorted compact lists, 269 pr.
  of static sets, 308 pr.
  in an unsorted array, 154 pr.
search list, *see* linked list
search tree, *see* balanced search tree, binary search tree, B-tree, exponential search tree, interval tree, optimal binary search tree, order-statistic tree, red-black tree, splay tree, 2-3 tree, 2-3-4 tree
secondary storage
  search tree for, 497–519
  stacks on, 517 pr.
second-best minimum spanning tree, 599 pr.
secret key, 936, 939
SELECT, 237
  used in quicksort, 241 ex.
SELECT3, 247 pr.
selection, 227
  of activities, 418–425
  and comparison sorts, 241
  in order-statistic trees, 481–482
  randomized, 230–236, 245 pr.
  in worst-case linear time, 236–243
selection sort, 33 ex., 53 ex.
selector vertex, 1087
self-loop, 1164
semiconnected graph, 581 ex.
semiring, 651 n., 669
sentinel
  in linked lists, 261–264
  in red-black trees, 332
sequence ($\langle\ \rangle$), 1162
  bitonic, 644 pr.
  inversion in, 134 ex., 486 ex.
  probe, 293
sequential consistency, 756
serial algorithm versus parallel algorithm, 748
serial projection, 750, 753
series, 1141–1144
  strands logically in, 756
series-parallel composition of parallel traces, 762 fig.
set ($\{\ \}$), 1153–1158
  cardinality ($|\ |$), 1156
  collection of, 1156
  convex, 675 ex.
  difference ($-$), 1154
  independent, 1099 pr.
  intersection ($\cap$), 1154
  member ($\in$), 1153
  not a member ($\notin$), 1153
  partially ordered, 1160
  static, 308 pr.
  union ($\cup$), 1154
set-covering problem, 1115–1119
  weighted, 1132 pr.
set-partition problem, 1098 ex.

SHA-256, 291
shared memory, 748
sharks with lasers, 850
Shell's sort, 48
shift
  left ($\lll$), 305
  right ($\ggg$), 285
  in string matching, 957
shift instruction, 27
short-circuiting operator, 24
SHORTEST-PATH, 1045
shortest paths, 604–669
  all-pairs, 605, 646–669
  Bellman-Ford algorithm for, 612–616
  with bitonic shortest paths, 644 pr.
  and breadth-first search, 558–561, 605
  convergence property of, 611, 634–635
  and cycles, 607–608
  and difference constraints, 626–632
  Dijkstra's algorithm for, 620–626
  in a directed acyclic graph, 616–619
  distance in (δ), 558
  in ϵ-dense graphs, 668 pr.
  estimate of, 609
  Floyd-Warshall algorithm for, 655–659, 662 ex.
  Gabow's scaling algorithm for, 641 pr.
  Johnson's algorithm for, 662–667
  as a linear program, 861
  and longest paths, 1042
  by matrix multiplication, 648–655, 668–669
  and negative-weight cycles, 606–607, 614–615, 655 ex., 662 ex.
  with negative-weight edges, 606–607
  no-path property of, 611, 634
  optimal substructure of, 605–606, 649, 655–656
  path-relaxation property of, 611, 635
  predecessor in (π), 608
  predecessor-subgraph property of, 611, 637–638
  problem variants, 605
  and relaxation, 609–611
  by repeated squaring, 652–653
  single-destination, 605
  single-pair, 385, 605
  single-source, 604–645
  tree of, 608–609, 635–638
  triangle inequality of, 611, 633
  in an unweighted graph, 385, 558
  upper-bound property of, 611, 633–634
  in a weighted graph, 604
  weight in (δ), 604
shortest remaining processing time (SRPT), 816 pr.
sibling, 1172
signature, 938
simple cycle, 1165–1166
simple graph, 1166
simple path, 1165
  longest, 385, 1042
SIMPLER-RANDOMIZED-SELECT, 243 pr.
simplex, 857
simplex algorithm, 626, 857, 876
simulation, 173, 181
single-destination shortest paths, 605
single-pair shortest path, 385, 605
  as a linear program, 861
single-source shortest paths, 604–645
  Bellman-Ford algorithm for, 612–616
  with bitonic shortest paths, 644 pr.
  and difference constraints, 626–632
  Dijkstra's algorithm for, 620–626
  in a directed acyclic graph, 616–619
  in ϵ-dense graphs, 668 pr.
  Gabow's scaling algorithm for, 641 pr.
  and longest paths, 1042
singleton, 1156
singly connected graph, 572 ex.
singly linked list, 259
singular matrix, 1220
singular value decomposition, 849
sink vertex, 553 ex., 671, 674
size
  of an algorithm's input, 28, 903–904, 1049–1052
  of a boolean combinational circuit, 1067
  of a clique, 1081
  of a group, 917
  of a set, 1156
  of a vertex cover, 1084, 1106
skip list, 359

slackness
complementary, 873 pr.
parallel, 758
slot
of a direct-access table, 273
of a hash table, 275
SLOW-APSP, 652
smoothed analysis, 876
solution
to an abstract problem, 1049
to a computational problem, 6
to a concrete problem, 1049
feasible, 627, 854
infeasible, 854
optimal, 854
to a system of linear equations, 820
sorted linked list, 259
sorting, 5, 17–21, 34–44, 51–53, 56–57, 157–226, 775–782
bubblesort, 46 pr.
bucket sort, 215–219
columnsort, 222 pr.
comparison sort, 205
counting sort, 208–211
fuzzy, 203 pr.
heapsort, 161–181
in place, 158, 220 pr.
insertion sort, 12–13, 17–21, 51–53, 56–57
k-sorting, 221 pr.
lexicographic, 327 pr., 986 n.
in linear time, 208–219, 220 pr.
lower bounds for, 205–208, 225
merge sort, 12–13, 34–44, 57, 775–782
by an oblivious compare-exchange algorithm, 222 pr.
parallel merge sort, 775–782
parallel quicksort, 789 pr.
probabilistic lower bound for, 219 pr.
quicksort, 182–204
radix sort, 211–215
selection sort, 33 ex., 53 ex.
Shell's sort, 48
stable, 210
table of running times, 159
topological, 573–576
using a binary search tree, 326 ex.
with variable-length items, 220 pr.
0-1 sorting lemma, 222 pr.
sorting network, 789
source vertex, 554, 605, 671, 674
span, 757
span law, 758
spanning tree, 585
bottleneck, 601 pr.
maximum, 1134 pr.
verification of, 603
see also minimum spanning tree
sparse graph, 549
all-pairs shortest paths for, 662–667
and Prim's algorithm, 599 pr.
sparse matrix, 81
spawn, in pseudocode, 752–754
spawning, 753
speedup, 758
of a randomized parallel algorithm, 789 pr.
spindle in a disk drive, 498
spine of a string-matching automaton, 970
splay tree, 359, 478
splicing
in a binary search tree, 324–325
in a linked list, 260–261
spline, 847 pr.
splitting
of B-tree nodes, 506–508
of 2-3-4 trees, 518 pr.
splitting summations, 1148–1149
spurious hit, 965
square matrix, 1215
square of a directed graph, 553 ex.
square root, modulo a prime, 954 pr.
squaring, repeated
for all-pairs shortest paths, 652–653
for raising a number to a power, 934
SRPT (shortest remaining processing time), 816 pr.
stability
numerical, 819, 821
of sorting algorithms, 210
stable-marriage problem, 716–723
stable matching, 716
stable-roommates problem, 723 ex.
stack, 254–255
implemented by queues, 258 ex.
implemented with a priority queue, 178 ex.
linked-list implementation of, 264 ex.

stack, *continued*
operations analyzed by accounting method, 454–455
operations analyzed by aggregate analysis, 449–451
operations analyzed by potential method, 457–458
for procedure execution, 202 pr.
on secondary storage, 517 pr.
STACK-EMPTY, 255
standard deviation, 1195
standard encoding ($\langle\ \rangle$), 1052
standard form of a linear program, 854
start state, 967
start time, 418
state of a finite automaton, 967
static graph, 522
static hashing, 282, 284–286
static set, 308 pr.
stencil, 787 pr.
Stirling's approximation, 67
stochastic gradient descent, 1040 pr.
STOOGE-SORT, 202 pr.
store instruction, 26, 756
strand, 754
mutually noninterfering, 767
Strassen's algorithm, 85–90, 124–125
parallel algorithm for, 773–774
streaks, 144–150, 153 ex.
streaming algorithms, 818
strict Fibonacci heap, 478
strictly decreasing, 63
strictly increasing, 63
string, 957, 1179
interpreted as a key, 290–291, 292 ex.
string matching, 957–1002
based on repetition factors, 996 pr.
by finite automata, 967–975
with gap characters, 961 ex., 975 ex.
Knuth-Morris-Pratt algorithm for, 975–985
naive algorithm for, 960–962
Rabin-Karp algorithm for, 962–967
by suffix arrays, 985–996
string-matching automaton, 968–975
strongly connected component, 1166
decomposition into, 576–581
STRONGLY-CONNECTED-COMPONENTS, 577
strongly connected graph, 1166
subarray (:), 19, 23
subgraph, 1166
equality, 724
predecessor, *see* predecessor subgraph
subgraph-isomorphism problem, 1098 ex.
subgroup, 921–923
subpath, 1165
subproblem graph, 370–371
subroutine, 23, 26, 29 n.
subsequence, 394
subset ($\subseteq$), 1154, 1156
SUBSET-SUM, 1092
subset-sum problem
approximation algorithm for, 1124–1130
NP-completeness of, 1092–1095
with unary target, 1098 ex.
substitution method, 90–95
in quicksort analysis, 191 ex., 193–194
and recursion trees, 98
in selection analysis, 240–241
substring, 962, 1179
rank of, 987
subtracting a low-order term, in the substitution method, 92–93
subtract instruction, 26
subtraction of matrices, 1218
subtree, 1172
maintaining size of, in order-statistic trees, 484–485
success, in a Bernoulli trial, 1196
successor
in binary search trees, 318–319
finding ith, of a node in an order-statistic tree, 486 ex.
in linked lists, 259
in red-black trees, 334
SUCCESSOR, 250
such that (:), 1154
suffix ($\sqsupset$), 959
suffix array (SA), 985–996, 997 pr.
suffix function, 968
suffix-function inequality, 971
suffix-function recursion lemma, 972
sum ($\sum$), 1140
Cartesian, 885 ex.
of matrices, 1217

sum ($\sum$), *continued*
of polynomials, 877
rule of, 1178
telescoping, 1143
SUM-ARRAY, 25 ex.
SUM-ARRAYS, 783 pr.
SUM-ARRAYS′, 783 pr.
summation, 1140–1152
approximated by integrals, 1150
in asymptotic notation, 58, 1141
bounding, 1145–1152
formulas and properties of, 1140–1145
linearity of, 1141
lower bounds on, 1148, 1150
splitting, 1148–1149
summation lemma, 887
supercomputer, 748
superpolynomial time, 1042
supersink, 674
supersource, 674
supervised learning, 1004
surjection, 1162
SVD, 849
symbol table, 272
symmetric difference, 706
symmetric-key cryptosystem, 941
symmetric matrix, 1217
symmetric positive-definite matrix, 838–841
inverse of, 784 pr.
symmetric relation, 1159
symmetry of Θ-notation, 61
sync, in pseudocode, 752–754
system of difference constraints, 626–632
system of linear equations, 784 pr., 819–833, 847 pr., 1034–1035

TABLE-DELETE, 467
TABLE-INSERT, 462
tail
of a binomial distribution, 1203–1210
of a linked list, 259
of a queue, 256
tail recursion, 202 pr., 422
target, 1092
Tarjan's offline lowest-common-ancestors algorithm, 543 pr.
task parallelism, 749
see also parallel algorithm
Task Parallel Library, 750
task-parallel scheduling, 759–761, 769 ex.
task scheduling, 446 pr., 816 pr.
tautology, 1060 ex.
Taylor series, 329 pr.
telescoping series, 1143
telescoping sum, 1143
termination of a loop invariant, 20
testing
of primality, 942–953, 956
of pseudoprimality, 944–945
text, 957
Theta-notation (Θ), 33, 51, 54 fig., 56
thread, 748
Threading Building Blocks, 750
thread parallelism, 748
3-CNF, 1076
3-CNF-SAT, 1076
3-CNF satisfiability, 1076–1079
approximation algorithm for, 1120–1121
and 2-CNF satisfiability, 1043
3-COLOR, 1100 pr.
3-conjunctive normal form, 1076
threshold constant, 77
tight bounds, 56
time, *see* running time
time domain, 877
time-memory trade-off, 367
timestamp, 564, 571 ex.
$T[i:]$ (suffix of a text), 986
$T[:k]$ (prefix of a text), 959
to, in pseudocode, 22
top-down method, for dynamic programming, 368
top of a stack, 254
topological sort, 573–576
in computing single-source shortest paths in a dag, 616
TOPOLOGICAL-SORT, 573
total order, 1160
total path length, 328 pr.
total preorder, 1160
total relation, 1160
tour
bitonic, 407 pr.
Euler, 583 pr., 1043
of a graph, 1090

trace, 754–756
  series-parallel composition of, 762 fig.
track in a disk drive, 498
tractability, 1042
trailing pointer, 321
training data, 1003
training phase, 1003
transition function, 967, 973–974
transitive closure, 659–661
  and boolean matrix multiplication, 838 ex.
  of dynamic graphs, 667 pr., 669
TRANSITIVE-CLOSURE, 660
transitive relation, 1159
transitivity of asymptotic notation, 61
TRANSPLANT, 324, 346
transpose
  conjugate, 838 ex.
  of a directed graph, 553 ex.
  of a matrix, 1214
transpose symmetry of asymptotic notation, 62
traveling-salesperson problem
  approximation algorithm for, 1109–1115
  bitonic euclidean, 407 pr.
  bottleneck, 1115 ex.
  NP-completeness of, 1090–1092
  with the triangle inequality, 1110–1113
  without the triangle inequality, 1113–1114
traversal of a tree, 314, 320 ex., 1112
treap, 358
tree, 1169–1176
  AA-trees, 358
  AVL, 357 pr., 358
  binary, *see* binary tree
  bisection of, 1177 pr.
  breadth-first, 555, 561
  B-trees, 497–519
  decision, 206–207, 219 pr.
  depth-first, 564
  diameter of, 563 ex.
  dynamic, 478
  free, 1167, 1169–1171
  full walk of, 1112
  fusion, 226, 478
  heap, 161–181
  height-balanced, 357 pr.
  height of, 1173
  interval, 489–495
  k-neighbor, 358
  left-leaning red-black binary search trees, 358
  minimum spanning, *see* minimum spanning tree
  optimal binary search, 400–407
  order-statistic, 480–486
  parse, 1077
  recursion, 42, 95–101
  red-black, *see* red-black tree
  rooted, 265–268, 1171
  scapegoat, 358
  search, *see* search tree
  shortest-paths, 608–609, 635–638
  spanning, *see* minimum spanning tree, spanning tree
  splay, 359, 478
  treap, 358
  2-3, 358, 519
  2-3-4, 502, 518 pr.
  van Emde Boas, 478
  walk of, 314, 320 ex., 1112
  weight-balanced trees, 358
TREE-DELETE, 325, 326 ex., 346–347
tree edge, 561, 564, 569
TREE-INSERT, 321, 338
TREE-MAXIMUM, 318
TREE-MINIMUM, 318
TREE-PREDECESSOR, 319
TREE-SEARCH, 316
TREE-SUCCESSOR, 319
tree walk, 314, 320 ex., 1112
TRE-QUICKSORT, 202 pr.
trial division, 943
triangle inequality, 1110
  for shortest paths, 611, 633
triangular matrix, 1216
trichotomy, interval, 490
trichotomy property of real numbers, 62
tridiagonal linear systems, 847 pr.
tridiagonal matrix, 1216
trie (radix tree), 327 pr.
TRIM, 1127
trimming a list, 1126
trivial divisor, 904
tropical semiring, 651 n.
truth assignment, 1066, 1073
truth table, 1065
TSP, 1091

tuple, 1157
twiddle factor, 891
2-CNF-SAT, 1080 ex.
2-CNF satisfiability, 1080 ex.
  and 3-CNF satisfiability, 1043
two-pass method, 529
2-3-4 tree, 502, 518 pr.
2-3 tree, 358, 519

unary, 1050
unbounded competitive ratio, 804
unbounded linear program, 854
uncle, 340
unconditional branch instruction, 26
unconstrained gradient descent, 1023–1031
uncountable set, 1156
underdetermined system of linear equations, 820
underflow
  of a queue, 256
  of a stack, 255
undirected graph, 1164
  articulation point of, 582 pr.
  biconnected component of, 582 pr.
  bridge of, 582 pr.
  clique in, 1081
  coloring of, 1100 pr., 1176 pr.
  computing a minimum spanning tree in, 585–603
  d-regular, 716 ex., 740 pr.
  grid, 697 pr.
  hamiltonian, 1056
  independent set of, 1099 pr.
  matching in, 693–697, 704–743
  nonhamiltonian, 1056
  vertex cover of, 1084, 1106
  see also graph
undirected version of a directed graph, 1167
uniform family of hash functions, 287
uniform hash function, 278
uniform hashing, 295
uniform probability distribution, 1186–1187
uniform random permutation, 128, 136
union
  of languages, 1052
  of linked lists, 264 ex.
  of sets (∪), 1154
UNION, 264 ex., 521
  disjoint-set-forest implementation of, 530
  linked-list implementation of, 524–526
union by rank, 528
unit (1), 905
unit lower-triangular matrix, 1216
unit upper-triangular matrix, 1216
unit vector, 1215
universal family of hash functions, 286–287
universal hash function, 278
universal hashing, 286–290, 309 pr.
universal sink, 553 ex.
universe, 273, 1155
unmatched vertex, 693, 705
unsorted linked list, 259
unstable matching, 717
unsupervised learning, 1004
until, in pseudocode, 22
unweighted longest simple paths, 385
unweighted shortest paths, 385
upper bound, 54
upper-bound property, 611, 633–634
upper median, 227
upper-triangular matrix, 1216

valid shift, 957
value
  of a flow, 672
  of a function, 1161
  objective, 854
Vandermonde matrix, 881, 1223 pr.
van Emde Boas tree, 478
Var [ ], *see* variance
variable
  decision, 851
  in pseudocode, 22
  random, 1191–1196
  see also indicator random variable
variable-length code, 432
variable-length input
  interpreted as a key, 290–291
  to the wee hash function, 306
variance, 1194
  of a binomial distribution, 1200
  of a geometric distribution, 1198
vector, 1215, 1219–1221
  convolution of, 880

CONCRETE MATHEMATICS

A FOUNDATION FOR COMPUTER SCIENCE

GRAHAM ○ KNUTH ○ PATASHNIK

SECOND EDITION

CONCRETE MATHEMATICS

Second Edition

Dedicated to Leonhard Euler (1707–1783)

A Foundation for Computer Science

CONCRETE MATHEMATICS

Second Edition

Ronald L. Graham
AT&T Bell Laboratories

Donald E. Knuth
Stanford University

Oren Patashnik
Center for Communications Research

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan Milan Paris

Library of Congress Cataloging-in-Publication Data

Graham, Ronald Lewis, 1935-
Concrete mathematics : a foundation for computer science / Ronald L. Graham, Donald E. Knuth, Oren Patashnik. -- 2nd ed.
xiii,657 p. 24 cm.
Bibliography: p. 604
Includes index.
ISBN 0-201-55802-5
1. Mathematics. 2. Computer science--Mathematics. I. Knuth, Donald Ervin, 1938- . II. Patashnik, Oren, 1954- . III. Title.
QA39.2.G733 1994
510--dc20 93-40325
CIP

Reproduced by Addison-Wesley from camera-ready copy supplied by the authors.

Copyright © 1994, 1989 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

1 2 3 4 5 6 7 8 9 10–MA–9897969594

Preface

"Audience, level, and treatment — a description of such matters is what prefaces are supposed to be about."
— P. R. Halmos [173]

THIS BOOK IS BASED on a course of the same name that has been taught annually at Stanford University since 1970. About fifty students have taken it each year — juniors and seniors, but mostly graduate students — and alumni of these classes have begun to spawn similar courses elsewhere. Thus the time seems ripe to present the material to a wider audience (including sophomores).

It was a dark and stormy decade when Concrete Mathematics was born. Long-held values were constantly being questioned during those turbulent years; college campuses were hotbeds of controversy. The college curriculum itself was challenged, and mathematics did not escape scrutiny. John Hammersley had just written a thought-provoking article "On the enfeeblement of mathematical skills by 'Modern Mathematics' and by similar soft intellectual trash in schools and universities" [176]; other worried mathematicians [332] even asked, "Can mathematics be saved?" One of the present authors had embarked on a series of books called *The Art of Computer Programming*, and in writing the first volume he (DEK) had found that there were mathematical tools missing from his repertoire; the mathematics he needed for a thorough, well-grounded understanding of computer programs was quite different from what he'd learned as a mathematics major in college. So he introduced a new course, teaching what he wished somebody had taught him.

"People do acquire a little brief authority by equipping themselves with jargon: they can pontificate and air a superficial expertise. But what we should ask of educated mathematicians is not what they can speechify about, nor even what they know about the existing corpus of mathematical knowledge, but rather what can they now do with their learning and whether they can actually solve mathematical problems arising in practice. In short, we look for deeds not words."
— J. Hammersley [176]

The course title "Concrete Mathematics" was originally intended as an antidote to "Abstract Mathematics," since concrete classical results were rapidly being swept out of the modern mathematical curriculum by a new wave of abstract ideas popularly called the "New Math." Abstract mathematics is a wonderful subject, and there's nothing wrong with it: It's beautiful, general, and useful. But its adherents had become deluded that the rest of mathematics was inferior and no longer worthy of attention. The goal of generalization had become so fashionable that a generation of mathematicians had become unable to relish beauty in the particular, to enjoy the challenge of solving quantitative problems, or to appreciate the value of technique. Abstract mathematics was becoming inbred and losing touch with reality; mathematical education needed a concrete counterweight in order to restore a healthy balance.

When DEK taught Concrete Mathematics at Stanford for the first time, he explained the somewhat strange title by saying that it was his attempt

to teach a math course that was hard instead of soft. He announced that, contrary to the expectations of some of his colleagues, he was *not* going to teach the Theory of Aggregates, nor Stone's Embedding Theorem, nor even the Stone–Čech compactification. (Several students from the civil engineering department got up and quietly left the room.)

"The heart of mathematics consists of concrete examples and concrete problems."
—P. R. Halmos [172]

Although Concrete Mathematics began as a reaction against other trends, the main reasons for its existence were positive instead of negative. And as the course continued its popular place in the curriculum, its subject matter "solidified" and proved to be valuable in a variety of new applications. Meanwhile, independent confirmation for the appropriateness of the name came from another direction, when Z. A. Melzak published two volumes entitled *Companion to Concrete Mathematics* [267].

"It is downright sinful to teach the abstract before the concrete."
—Z. A. Melzak [267]

The material of concrete mathematics may seem at first to be a disparate bag of tricks, but practice makes it into a disciplined set of tools. Indeed, the techniques have an underlying unity and a strong appeal for many people. When another one of the authors (RLG) first taught the course in 1979, the students had such fun that they decided to hold a class reunion a year later.

But what exactly is Concrete Mathematics? It is a blend of CONtinuous and disCRETE mathematics. More concretely, it is the controlled manipulation of mathematical formulas, using a collection of techniques for solving problems. Once you, the reader, have learned the material in this book, all you will need is a cool head, a large sheet of paper, and fairly decent handwriting in order to evaluate horrendous-looking sums, to solve complex recurrence relations, and to discover subtle patterns in data. You will be so fluent in algebraic techniques that you will often find it easier to obtain exact results than to settle for approximate answers that are valid only in a limiting sense.

Concrete Mathematics is a bridge to abstract mathematics.

The major topics treated in this book include sums, recurrences, elementary number theory, binomial coefficients, generating functions, discrete probability, and asymptotic methods. The emphasis is on manipulative technique rather than on existence theorems or combinatorial reasoning; the goal is for each reader to become as familiar with discrete operations (like the greatest-integer function and finite summation) as a student of calculus is familiar with continuous operations (like the absolute-value function and infinite integration).

"The advanced reader who skips parts that appear too elementary may miss more than the less advanced reader who skips parts that appear too complex."
—G. Pólya [297]

Notice that this list of topics is quite different from what is usually taught nowadays in undergraduate courses entitled "Discrete Mathematics." Therefore the subject needs a distinctive name, and "Concrete Mathematics" has proved to be as suitable as any other.

(We're not bold enough to try Distinuous Mathematics.)

The original textbook for Stanford's course on concrete mathematics was the "Mathematical Preliminaries" section in *The Art of Computer Programming* [207]. But the presentation in those 110 pages is quite terse, so another author (OP) was inspired to draft a lengthy set of supplementary notes. The

present book is an outgrowth of those notes; it is an expansion of, and a more leisurely introduction to, the material of Mathematical Preliminaries. Some of the more advanced parts have been omitted; on the other hand, several topics not found there have been included here so that the story will be complete.

"... a concrete life preserver thrown to students sinking in a sea of abstraction."
—W. Gottschalk

The authors have enjoyed putting this book together because the subject began to jell and to take on a life of its own before our eyes; this book almost seemed to write itself. Moreover, the somewhat unconventional approaches we have adopted in several places have seemed to fit together so well, after these years of experience, that we can't help feeling that this book is a kind of manifesto about our favorite way to do mathematics. So we think the book has turned out to be a tale of mathematical beauty and surprise, and we hope that our readers will share at least ϵ of the pleasure we had while writing it.

Since this book was born in a university setting, we have tried to capture the spirit of a contemporary classroom by adopting an informal style. Some people think that mathematics is a serious business that must always be cold and dry; but we think mathematics is fun, and we aren't ashamed to admit the fact. Why should a strict boundary line be drawn between work and play? Concrete mathematics is full of appealing patterns; the manipulations are not always easy, but the answers can be astonishingly attractive. The joys and sorrows of mathematical work are reflected explicitly in this book because they are part of our lives.

Math graffiti:
Kilroy wasn't Haar.
Free the group.
Nuke the kernel.
Power to the n.
$N=1 \Rightarrow P=NP$.

Students always know better than their teachers, so we have asked the first students of this material to contribute their frank opinions, as "graffiti" in the margins. Some of these marginal markings are merely corny, some are profound; some of them warn about ambiguities or obscurities, others are typical comments made by wise guys in the back row; some are positive, some are negative, some are zero. But they all are real indications of feelings that should make the text material easier to assimilate. (The inspiration for such marginal notes comes from a student handbook entitled *Approaching Stanford*, where the official university line is counterbalanced by the remarks of outgoing students. For example, Stanford says, "There are a few things you cannot miss in this amorphous shape which is Stanford"; the margin says, "Amorphous ... what the h*** does that mean? Typical of the pseudo-intellectualism around here." Stanford: "There is no end to the potential of a group of students living together." Graffito: "Stanford dorms are like zoos without a keeper.")

I have only a marginal interest in this subject.

This was the most enjoyable course I've ever had. But it might be nice to summarize the material as you go along.

The margins also include direct quotations from famous mathematicians of past generations, giving the actual words in which they announced some of their fundamental discoveries. Somehow it seems appropriate to mix the words of Leibniz, Euler, Gauss, and others with those of the people who will be continuing the work. Mathematics is an ongoing endeavor for people everywhere; many strands are being woven into one rich fabric.

This book contains more than 500 exercises, divided into six categories:

I see: Concrete mathematics means drilling.

- ***Warmups*** are exercises that EVERY READER should try to do when first reading the material.
- ***Basics*** are exercises to develop facts that are best learned by trying one's own derivation rather than by reading somebody else's.

The homework was tough but I learned a lot. It was worth every hour.

- ***Homework exercises*** are problems intended to deepen an understanding of material in the current chapter.
- ***Exam problems*** typically involve ideas from two or more chapters simultaneously; they are generally intended for use in take-home exams (not for in-class exams under time pressure).

Take-home exams are vital — keep them.

- ***Bonus problems*** go beyond what an average student of concrete mathematics is expected to handle while taking a course based on this book; they extend the text in interesting ways.

Exams were harder than the homework led me to expect.

- ***Research problems*** may or may not be humanly solvable, but the ones presented here seem to be worth a try (without time pressure).

Answers to all the exercises appear in Appendix A, often with additional information about related results. (Of course, the "answers" to research problems are incomplete; but even in these cases, partial results or hints are given that might prove to be helpful.) Readers are encouraged to look at the answers, especially the answers to the warmup problems, but only AFTER making a serious attempt to solve the problem without peeking.

Cheaters may pass this course by just copying the answers, but they're only cheating themselves.

We have tried in Appendix C to give proper credit to the sources of each exercise, since a great deal of creativity and/or luck often goes into the design of an instructive problem. Mathematicians have unfortunately developed a tradition of borrowing exercises without any acknowledgment; we believe that the opposite tradition, practiced for example by books and magazines about chess (where names, dates, and locations of original chess problems are routinely specified) is far superior. However, we have not been able to pin down the sources of many problems that have become part of the folklore. If any reader knows the origin of an exercise for which our citation is missing or inaccurate, we would be glad to learn the details so that we can correct the omission in subsequent editions of this book.

Difficult exams don't take into account students who have other classes to prepare for.

The typeface used for mathematics throughout this book is a new design by Hermann Zapf [227], commissioned by the American Mathematical Society and developed with the help of a committee that included B. Beeton, R. P. Boas, L. K. Durst, D. E. Knuth, P. Murdock, R. S. Palais, P. Renz, E. Swanson, S. B. Whidden, and W. B. Woolf. The underlying philosophy of Zapf's design is to capture the flavor of mathematics as it might be written by a mathematician with excellent handwriting. A handwritten rather than mechanical style is appropriate because people generally create mathematics with pen, pencil,

I'm unaccustomed to this face.

or chalk. (For example, one of the trademarks of the new design is the symbol for zero, '0', which is slightly pointed at the top because a handwritten zero rarely closes together smoothly when the curve returns to its starting point.) The letters are upright, not italic, so that subscripts, superscripts, and accents are more easily fitted with ordinary symbols. This new type family has been named *AMS Euler*, after the great Swiss mathematician Leonhard Euler (1707–1783) who discovered so much of mathematics as we know it today. The alphabets include Euler Text (Aa Bb Cc through Xx Yy Zz), Euler Fraktur ($\mathfrak{Aa}\,\mathfrak{Bb}\,\mathfrak{Cc}$ through $\mathfrak{Xx}\,\mathfrak{Yy}\,\mathfrak{Zz}$), and Euler Script Capitals ($\mathcal{A}\,\mathcal{B}\,\mathcal{C}$ through $\mathcal{X}\,\mathcal{Y}\,\mathcal{Z}$), as well as Euler Greek ($A\alpha\,B\beta\,\Gamma\gamma$ through $X\chi\,\Psi\psi\,\Omega\omega$) and special symbols such as $\wp$ and $\aleph$. We are especially pleased to be able to inaugurate the Euler family of typefaces in this book, because Leonhard Euler's spirit truly lives on every page: Concrete mathematics is Eulerian mathematics.

Dear prof: Thanks for (1) the puns, (2) the subject matter.

The authors are extremely grateful to Andrei Broder, Ernst Mayr, Andrew Yao, and Frances Yao, who contributed greatly to this book during the years that they taught Concrete Mathematics at Stanford. Furthermore we offer 1024 thanks to the teaching assistants who creatively transcribed what took place in class each year and who helped to design the examination questions; their names are listed in Appendix C. This book, which is essentially a compendium of sixteen years' worth of lecture notes, would have been impossible without their first-rate work.

I don't see how what I've learned will ever help me.

Many other people have helped to make this book a reality. For example, we wish to commend the students at Brown, Columbia, CUNY, Princeton, Rice, and Stanford who contributed the choice graffiti and helped to debug our first drafts. Our contacts at Addison-Wesley were especially efficient and helpful; in particular, we wish to thank our publisher (Peter Gordon), production supervisor (Bette Aaronson), designer (Roy Brown), and copy editor (Lyn Dupré). The National Science Foundation and the Office of Naval Research have given invaluable support. Cheryl Graham was tremendously helpful as we prepared the index. And above all, we wish to thank our wives (Fan, Jill, and Amy) for their patience, support, encouragement, and ideas.

I had a lot of trouble in this class, but I know it sharpened my math skills and my thinking skills.

This second edition features a new Section 5.8, which describes some important ideas that Doron Zeilberger discovered shortly after the first edition went to press. Additional improvements to the first printing can also be found on almost every page.

We have tried to produce a perfect book, but we are imperfect authors. Therefore we solicit help in correcting any mistakes that we've made. A reward of \$2.56 will gratefully be paid to the first finder of any error, whether it is mathematical, historical, or typographical.

I would advise the casual student to stay away from this course.

Murray Hill, New Jersey — RLG
and Stanford, California DEK
May 1988 and October 1993 OP

A Note on Notation

SOME OF THE SYMBOLISM in this book has not (yet?) become standard. Here is a list of notations that might be unfamiliar to readers who have learned similar material from other books, together with the page numbers where these notations are explained. (See the general index, at the end of the book, for references to more standard notations.)

Notation	*Name*	*Page*
$\ln x$	natural logarithm: $\log_e x$	276
$\lg x$	binary logarithm: $\log_2 x$	70
$\log x$	common logarithm: $\log_{10} x$	449
$\lfloor x \rfloor$	floor: $\max\{\, n \mid n \leqslant x,\ \text{integer } n \,\}$	67
$\lceil x \rceil$	ceiling: $\min\{\, n \mid n \geqslant x,\ \text{integer } n \,\}$	67
$x \bmod y$	remainder: $x - y\lfloor x/y \rfloor$	82
$\{x\}$	fractional part: $x \bmod 1$	70
$\sum f(x)\,\delta x$	indefinite summation	48
$\sum_a^b f(x)\,\delta x$	definite summation	49
$x^{\underline{n}}$	falling factorial power: $x!/(x-n)!$	47, 211
$x^{\overline{n}}$	rising factorial power: $\Gamma(x+n)/\Gamma(x)$	48, 211
$n¡$	subfactorial: $n!/0! - n!/1! + \cdots + (-1)^n n!/n!$	194
$\Re z$	real part: x, if $z = x + iy$	64
$\Im z$	imaginary part: y, if $z = x + iy$	64
H_n	harmonic number: $1/1 + \cdots + 1/n$	29
$H_n^{(x)}$	generalized harmonic number: $1/1^x + \cdots + 1/n^x$	277

If you don't understand what the x denotes at the bottom of this page, try asking your Latin professor instead of your math professor.

$f^{(m)}(z)$	mth derivative of f at z	470
$\left[{n \atop m}\right]$	Stirling cycle number (the "first kind")	259
$\left\{{n \atop m}\right\}$	Stirling subset number (the "second kind")	258
$\left\langle{n \atop m}\right\rangle$	Eulerian number	267
$\left\langle\!\left\langle{n \atop m}\right\rangle\!\right\rangle$	Second-order Eulerian number	270
$(a_m \ldots a_0)_b$	radix notation for $\sum_{k=0}^{m} a_k b^k$	11
$K(a_1, \ldots, a_n)$	continuant polynomial	302
$F\left(\begin{matrix} a, b \\ c \end{matrix} \middle\vert z\right)$	hypergeometric function	205
$\#A$	cardinality: number of elements in the set A	39
$[z^n]\, f(z)$	coefficient of z^n in $f(z)$	197
$[\alpha \mathinner{..} \beta]$	closed interval: the set $\{x \mid \alpha \leqslant x \leqslant \beta\}$	73
$[m = n]$	1 if $m = n$, otherwise 0 *	24
$[m \backslash n]$	1 if m divides n, otherwise 0 *	102
$[m \backslash\backslash n]$	1 if m exactly divides n, otherwise 0 *	146
$[m \perp n]$	1 if m is relatively prime to n, otherwise 0 *	115

Prestressed concrete mathematics is concrete mathematics that's preceded by a bewildering list of notations.

*In general, if S is any statement that can be true or false, the bracketed notation $[S]$ stands for 1 if S is true, 0 otherwise.

Throughout this text, we use single-quote marks ('...') to delimit text as it is *written*, double-quote marks ("...") for a phrase as it is *spoken*. Thus, the string of letters 'string' is sometimes called a "string."

Also 'nonstring' is a string.

An expression of the form 'a/bc' means the same as '$a/(bc)$'. Moreover, $\log x / \log y = (\log x)/(\log y)$ and $2n! = 2(n!)$.

Contents

1 Recurrent Problems — **1**

1.1 The Tower of Hanoi 1
1.2 Lines in the Plane 4
1.3 The Josephus Problem 8
Exercises 17

2 Sums — **21**

2.1 Notation 21
2.2 Sums and Recurrences 25
2.3 Manipulation of Sums 30
2.4 Multiple Sums 34
2.5 General Methods 41
2.6 Finite and Infinite Calculus 47
2.7 Infinite Sums 56
Exercises 62

3 Integer Functions — **67**

3.1 Floors and Ceilings 67
3.2 Floor/Ceiling Applications 70
3.3 Floor/Ceiling Recurrences 78
3.4 'mod': The Binary Operation 81
3.5 Floor/Ceiling Sums 86
Exercises 95

4 Number Theory — **102**

4.1 Divisibility 102
4.2 Primes 105
4.3 Prime Examples 107
4.4 Factorial Factors 111
4.5 Relative Primality 115
4.6 'mod': The Congruence Relation 123
4.7 Independent Residues 126
4.8 Additional Applications 129
4.9 Phi and Mu 133
Exercises 144

5 Binomial Coefficients — **153**

5.1 Basic Identities 153
5.2 Basic Practice 172
5.3 Tricks of the Trade 186

5.4 Generating Functions 196
5.5 Hypergeometric Functions 204
5.6 Hypergeometric Transformations 216
5.7 Partial Hypergeometric Sums 223
5.8 Mechanical Summation 229
Exercises 242

6 Special Numbers **257**
6.1 Stirling Numbers 257
6.2 Eulerian Numbers 267
6.3 Harmonic Numbers 272
6.4 Harmonic Summation 279
6.5 Bernoulli Numbers 283
6.6 Fibonacci Numbers 290
6.7 Continuants 301
Exercises 309

7 Generating Functions **320**
7.1 Domino Theory and Change 320
7.2 Basic Maneuvers 331
7.3 Solving Recurrences 337
7.4 Special Generating Functions 350
7.5 Convolutions 353
7.6 Exponential Generating Functions 364
7.7 Dirichlet Generating Functions 370
Exercises 371

8 Discrete Probability **381**
8.1 Definitions 381
8.2 Mean and Variance 387
8.3 Probability Generating Functions 394
8.4 Flipping Coins 401
8.5 Hashing 411
Exercises 427

9 Asymptotics **439**
9.1 A Hierarchy 440
9.2 O Notation 443
9.3 O Manipulation 450
9.4 Two Asymptotic Tricks 463
9.5 Euler's Summation Formula 469
9.6 Final Summations 476
Exercises 489

A Answers to Exercises **497**

B Bibliography **604**

C Credits for Exercises **632**

Index **637**

List of Tables **657**

1

Recurrent Problems

THIS CHAPTER EXPLORES three sample problems that give a feel for what's to come. They have two traits in common: They've all been investigated repeatedly by mathematicians; and their solutions all use the idea of *recurrence*, in which the solution to each problem depends on the solutions to smaller instances of the same problem.

1.1 THE TOWER OF HANOI

Raise your hand if you've never seen this. OK, the rest of you can cut to equation (1.1).

Let's look first at a neat little puzzle called the Tower of Hanoi, invented by the French mathematician Edouard Lucas in 1883. We are given a tower of eight disks, initially stacked in decreasing size on one of three pegs:

The objective is to transfer the entire tower to one of the other pegs, moving only one disk at a time and never moving a larger one onto a smaller.

Gold — wow. Are our disks made of concrete?

Lucas [260] furnished his toy with a romantic legend about a much larger Tower of Brahma, which supposedly has 64 disks of pure gold resting on three diamond needles. At the beginning of time, he said, God placed these golden disks on the first needle and ordained that a group of priests should transfer them to the third, according to the rules above. The priests reportedly work day and night at their task. When they finish, the Tower will crumble and the world will end.

It's not immediately obvious that the puzzle has a solution, but a little thought (or having seen the problem before) convinces us that it does. Now the question arises: What's the best we can do? That is, how many moves are necessary and sufficient to perform the task?

The best way to tackle a question like this is to generalize it a bit. The Tower of Brahma has 64 disks and the Tower of Hanoi has 8; let's consider what happens if there are n disks.

One advantage of this generalization is that we can scale the problem down even more. In fact, we'll see repeatedly in this book that it's advantageous to LOOK AT SMALL CASES first. It's easy to see how to transfer a tower that contains only one or two disks. And a small amount of experimentation shows how to transfer a tower of three.

The next step in solving the problem is to introduce appropriate notation: NAME AND CONQUER. Let's say that T_n is the minimum number of moves that will transfer n disks from one peg to another under Lucas's rules. Then T_1 is obviously 1, and $T_2 = 3$.

We can also get another piece of data for free, by considering the smallest case of all: Clearly $T_0 = 0$, because no moves at all are needed to transfer a tower of $n = 0$ disks! Smart mathematicians are not ashamed to think small, because general patterns are easier to perceive when the extreme cases are well understood (even when they are trivial).

But now let's change our perspective and try to think big; how can we transfer a large tower? Experiments with three disks show that the winning idea is to transfer the top two disks to the middle peg, then move the third, then bring the other two onto it. This gives us a clue for transferring n disks in general: We first transfer the $n-1$ smallest to a different peg (requiring T_{n-1} moves), then move the largest (requiring one move), and finally transfer the $n-1$ smallest back onto the largest (requiring another T_{n-1} moves). Thus we can transfer n disks (for $n > 0$) in at most $2T_{n-1} + 1$ moves:

$$T_n \leqslant 2T_{n-1} + 1\,, \qquad \text{for } n > 0.$$

This formula uses '$\leqslant$' instead of '$=$' because our construction proves only that $2T_{n-1} + 1$ moves suffice; we haven't shown that $2T_{n-1} + 1$ moves are necessary. A clever person might be able to think of a shortcut.

But is there a better way? Actually no. At some point we must move the largest disk. When we do, the $n-1$ smallest must be on a single peg, and it has taken at least T_{n-1} moves to put them there. We might move the largest disk more than once, if we're not too alert. But after moving the largest disk for the last time, we must transfer the $n-1$ smallest disks (which must again be on a single peg) back onto the largest; this too requires T_{n-1} moves. Hence

Most of the published "solutions" to Lucas's problem, like the early one of Allardice and Fraser [7], fail to explain why T_n must be $\geqslant 2T_{n-1} + 1$.

$$T_n \geqslant 2T_{n-1} + 1\,, \qquad \text{for } n > 0.$$

These two inequalities, together with the trivial solution for $n = 0$, yield

$$\begin{aligned} T_0 &= 0\,; \\ T_n &= 2T_{n-1} + 1\,, \qquad \text{for } n > 0. \end{aligned} \tag{1.1}$$

(Notice that these formulas are consistent with the known values $T_1 = 1$ and $T_2 = 3$. Our experience with small cases has not only helped us to discover a general formula, it has also provided a convenient way to check that we haven't made a foolish error. Such checks will be especially valuable when we get into more complicated maneuvers in later chapters.)

Yeah, yeah . . . I seen that word before.

A set of equalities like (1.1) is called a *recurrence* (a.k.a. recurrence relation or recursion relation). It gives a boundary value and an equation for the general value in terms of earlier ones. Sometimes we refer to the general equation alone as a recurrence, although technically it needs a boundary value to be complete.

The recurrence allows us to compute T_n for any n we like. But nobody really likes to compute from a recurrence, when n is large; it takes too long. The recurrence only gives indirect, local information. A *solution to the recurrence* would make us much happier. That is, we'd like a nice, neat, "closed form" for T_n that lets us compute it quickly, even for large n. With a closed form, we can understand what T_n really is.

So how do we solve a recurrence? One way is to guess the correct solution, then to prove that our guess is correct. And our best hope for guessing the solution is to look (again) at small cases. So we compute, successively, $T_3 = 2\cdot 3 + 1 = 7$; $T_4 = 2\cdot 7 + 1 = 15$; $T_5 = 2\cdot 15 + 1 = 31$; $T_6 = 2\cdot 31 + 1 = 63$. Aha! It certainly looks as if

$$T_n \;=\; 2^n - 1\,, \qquad \text{for } n \geqslant 0. \tag{1.2}$$

At least this works for $n \leqslant 6$.

Mathematical induction is a general way to prove that some statement about the integer n is true for all $n \geqslant n_0$. First we prove the statement when n has its smallest value, n_0; this is called the *basis*. Then we prove the statement for $n > n_0$, assuming that it has already been proved for all values between n_0 and $n - 1$, inclusive; this is called the *induction*. Such a proof gives infinitely many results with only a finite amount of work.

Mathematical induction proves that we can climb as high as we like on a ladder, by proving that we can climb onto the bottom rung (the basis) and that from each rung we can climb up to the next one (the induction).

Recurrences are ideally set up for mathematical induction. In our case, for example, (1.2) follows easily from (1.1): The basis is trivial, since $T_0 = 2^0 - 1 = 0$. And the induction follows for $n > 0$ if we assume that (1.2) holds when n is replaced by $n - 1$:

$$T_n \;=\; 2T_{n-1} + 1 \;=\; 2(2^{n-1} - 1) + 1 \;=\; 2^n - 1\,.$$

Hence (1.2) holds for n as well. Good! Our quest for T_n has ended successfully.

Of course the priests' task hasn't ended; they're still dutifully moving disks, and will be for a while, because for $n = 64$ there are $2^{64}-1$ moves (about 18 quintillion). Even at the impossible rate of one move per microsecond, they will need more than 5000 centuries to transfer the Tower of Brahma. Lucas's original puzzle is a bit more practical. It requires $2^8 - 1 = 255$ moves, which takes about four minutes for the quick of hand.

The Tower of Hanoi recurrence is typical of many that arise in applications of all kinds. In finding a closed-form expression for some quantity of interest like T_n we go through three stages:

1 Look at small cases. This gives us insight into the problem and helps us in stages 2 and 3.

2 Find and prove a mathematical expression for the quantity of interest. For the Tower of Hanoi, this is the recurrence (1.1) that allows us, given the inclination, to compute T_n for any n.

What is a proof? "One half of one percent pure alcohol."

3 Find and prove a closed form for our mathematical expression. For the Tower of Hanoi, this is the recurrence solution (1.2).

The third stage is the one we will concentrate on throughout this book. In fact, we'll frequently skip stages 1 and 2 entirely, because a mathematical expression will be given to us as a starting point. But even then, we'll be getting into subproblems whose solutions will take us through all three stages.

Our analysis of the Tower of Hanoi led to the correct answer, but it required an "inductive leap"; we relied on a lucky guess about the answer. One of the main objectives of this book is to explain how a person can solve recurrences *without* being clairvoyant. For example, we'll see that recurrence (1.1) can be simplified by adding 1 to both sides of the equations:

$$\begin{aligned} &T_0 + 1 = 1\,; \\ &T_n + 1 = 2T_{n-1} + 2\,, \qquad \text{for } n > 0. \end{aligned}$$

Now if we let $U_n = T_n + 1$, we have

Interesting: We get rid of the $+1$ in (1.1) by adding, not by subtracting.

$$\begin{aligned} &U_0 = 1\,; \\ &U_n = 2U_{n-1}\,, \qquad \text{for } n > 0. \end{aligned} \tag{1.3}$$

It doesn't take genius to discover that the solution to *this* recurrence is just $U_n = 2^n$; hence $T_n = 2^n - 1$. Even a computer could discover this.

1.2 LINES IN THE PLANE

Our second sample problem has a more geometric flavor: How many slices of pizza can a person obtain by making n straight cuts with a pizza knife? Or, more academically: What is the maximum number L_n of regions

(A pizza with Swiss cheese?)

defined by n lines in the plane? This problem was first solved in 1826, by the Swiss mathematician Jacob Steiner [338].

Again we start by looking at small cases, remembering to begin with the smallest of all. The plane with no lines has one region; with one line it has two regions; and with two lines it has four regions:

$L_0 = 1$ $\qquad$ $L_1 = 2$ $\qquad$ $L_2 = 4$

(Each line extends infinitely in both directions.)

Sure, we think, $L_n = 2^n$; of course! Adding a new line simply doubles the number of regions. Unfortunately this is wrong. We could achieve the doubling if the nth line would split each old region in two; certainly it can split an old region in at most two pieces, since each old region is convex. (A straight line can split a convex region into at most two new regions, which will also be convex.) But when we add the third line — the thick one in the diagram below — we soon find that it can split at most three of the old regions, no matter how we've placed the first two lines:

A region is convex if it includes all line segments between any two of its points. (That's not what my dictionary says, but it's what mathematicians believe.)

Thus $L_3 = 4 + 3 = 7$ is the best we can do.

And after some thought we realize the appropriate generalization. The nth line (for $n > 0$) increases the number of regions by k if and only if it splits k of the old regions, and it splits k old regions if and only if it hits the previous lines in $k-1$ different places. Two lines can intersect in at most one point. Therefore the new line can intersect the $n-1$ old lines in at most $n-1$ different points, and we must have $k \leqslant n$. We have established the upper bound

$$L_n \leqslant L_{n-1} + n, \qquad \text{for } n > 0.$$

Furthermore it's easy to show by induction that we can achieve equality in this formula. We simply place the nth line in such a way that it's not parallel to any of the others (hence it intersects them all), and such that it doesn't go

through any of the existing intersection points (hence it intersects them all in different places). The recurrence is therefore

$$\begin{aligned} L_0 &= 1\,; \\ L_n &= L_{n-1} + n\,, \qquad \text{for } n > 0. \end{aligned} \tag{1.4}$$

The known values of L_1, L_2, and L_3 check perfectly here, so we'll buy this.

Now we need a closed-form solution. We could play the guessing game again, but 1, 2, 4, 7, 11, 16, ... doesn't look familiar; so let's try another tack. We can often understand a recurrence by "unfolding" or "unwinding" it all the way to the end, as follows:

Unfolding? I'd call this "plugging in."

$$\begin{aligned} L_n &= L_{n-1} + n \\ &= L_{n-2} + (n-1) + n \\ &= L_{n-3} + (n-2) + (n-1) + n \\ &\quad\ \vdots \\ &= L_0 + 1 + 2 + \cdots + (n-2) + (n-1) + n \\ &= 1 \ + \ S_n\,, \qquad \text{where } S_n = 1 + 2 + 3 + \cdots + (n-1) + n. \end{aligned}$$

In other words, L_n is one more than the sum S_n of the first n positive integers.

The quantity S_n pops up now and again, so it's worth making a table of small values. Then we might recognize such numbers more easily when we see them the next time:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S_n	1	3	6	10	15	21	28	36	45	55	66	78	91	105

These values are also called the *triangular numbers*, because S_n is the number of bowling pins in an n-row triangular array. For example, the usual four-row array has $S_4 = 10$ pins.

To evaluate S_n we can use a trick that Gauss reportedly came up with in 1786, when he was nine years old [88] (see also Euler [114, part 1, §415]):

It seems a lot of stuff is attributed to Gauss — either he was really smart or he had a great press agent.

$$\begin{array}{rcccccccccccc} S_n &=& 1 &+& 2 &+& 3 &+\cdots+& (n-1) &+& n \\ +\,S_n &=& n &+& (n-1) &+& (n-2) &+\cdots+& 2 &+& 1 \\ \hline 2S_n &=& (n+1) &+& (n+1) &+& (n+1) &+\cdots+& (n+1) &+& (n+1) \end{array}$$

We merely add S_n to its reversal, so that each of the n columns on the right sums to $n+1$. Simplifying,

Maybe he just had a magnetic personality.

$$S_n = \frac{n(n+1)}{2}\,, \qquad \text{for } n \geqslant 0. \tag{1.5}$$

Actually Gauss is often called the greatest mathematician of all time. So it's nice to be able to understand at least one of his discoveries.

OK, we have our solution:

$$L_n = \frac{n(n+1)}{2} + 1, \qquad \text{for } n \geqslant 0. \tag{1.6}$$

As experts, we might be satisfied with this derivation and consider it a proof, even though we waved our hands a bit when doing the unfolding and reflecting. But students of mathematics should be able to meet stricter standards; so it's a good idea to construct a rigorous proof by induction. The key induction step is

$$L_n = L_{n-1} + n = \left(\tfrac{1}{2}(n-1)n + 1\right) + n = \tfrac{1}{2}n(n+1) + 1.$$

Now there can be no doubt about the closed form (1.6).

When in doubt, look at the words. Why is it "closed," as opposed to "open"? What image does it bring to mind? Answer: The equation is "closed," not defined in terms of itself—not leading to recurrence. The case is "closed"—it won't happen again. Metaphors are the key.

Incidentally we've been talking about "closed forms" without explicitly saying what we mean. Usually it's pretty clear. Recurrences like (1.1) and (1.4) are not in closed form—they express a quantity in terms of itself; but solutions like (1.2) and (1.6) are. Sums like $1 + 2 + \cdots + n$ are not in closed form—they cheat by using '$\cdots$'; but expressions like $n(n+1)/2$ are. We could give a rough definition like this: An expression for a quantity $f(n)$ is in closed form if we can compute it using at most a fixed number of "well known" standard operations, independent of n. For example, $2^n - 1$ and $n(n+1)/2$ are closed forms because they involve only addition, subtraction, multiplication, division, and exponentiation, in explicit ways.

The total number of simple closed forms is limited, and there are recurrences that don't have simple closed forms. When such recurrences turn out to be important, because they arise repeatedly, we add new operations to our repertoire; this can greatly extend the range of problems solvable in "simple" closed form. For example, the product of the first n integers, $n!$, has proved to be so important that we now consider it a basic operation. The formula '$n!$' is therefore in closed form, although its equivalent '$1 \cdot 2 \cdot \ldots \cdot n$' is not.

Is "zig" a technical term?

And now, briefly, a variation of the lines-in-the-plane problem: Suppose that instead of straight lines we use bent lines, each containing one "zig." What is the maximum number Z_n of regions determined by n such bent lines in the plane? We might expect Z_n to be about twice as big as L_n, or maybe three times as big. Let's see:

$Z_1 = 2$

$Z_2 = 7$

From these small cases, and after a little thought, we realize that a bent line is like two straight lines except that regions merge when the "two" lines don't extend past their intersection point.

... and a little afterthought...

Regions 2, 3, and 4, which would be distinct with two lines, become a single region when there's a bent line; we lose two regions. However, if we arrange things properly — the zig point must lie "beyond" the intersections with the other lines — that's all we lose; that is, we lose only two regions per line. Thus

Exercise 18 has the details.

$$\begin{aligned} Z_n &= L_{2n} - 2n = 2n(2n+1)/2 + 1 - 2n \\ &= 2n^2 - n + 1\,, \qquad \text{for } n \geqslant 0. \end{aligned} \tag{1.7}$$

Comparing the closed forms (1.6) and (1.7), we find that for large n,

$$\begin{aligned} L_n &\sim \tfrac{1}{2}n^2\,, \\ Z_n &\sim 2n^2\,; \end{aligned}$$

so we get about four times as many regions with bent lines as with straight lines. (In later chapters we'll be discussing how to analyze the approximate behavior of integer functions when n is large. The '$\sim$' symbol is defined in Section 9.1.)

1.3 THE JOSEPHUS PROBLEM

Our final introductory example is a variant of an ancient problem named for Flavius Josephus, a famous historian of the first century. Legend has it that Josephus wouldn't have lived to become famous without his mathematical talents. During the Jewish–Roman war, he was among a band of 41 Jewish rebels trapped in a cave by the Romans. Preferring suicide to capture, the rebels decided to form a circle and, proceeding around it, to kill every third remaining person until no one was left. But Josephus, along with an unindicted co-conspirator, wanted none of this suicide nonsense; so he quickly calculated where he and his friend should stand in the vicious circle.

(Ahrens [5, vol. 2] and Herstein and Kaplansky [187] discuss the interesting history of this problem. Josephus himself [197] is a bit vague.)

... thereby saving his tale for us to hear.

In our variation, we start with n people numbered 1 to n around a circle, and we eliminate every *second* remaining person until only one survives. For

example, here's the starting configuration for $n = 10$:

The elimination order is 2, 4, 6, 8, 10, 3, 7, 1, 9, so 5 survives. The problem: Determine the survivor's number, $J(n)$.

Here's a case where $n = 0$ makes no sense.

We just saw that $J(10) = 5$. We might conjecture that $J(n) = n/2$ when n is even; and the case $n = 2$ supports the conjecture: $J(2) = 1$. But a few other small cases dissuade us—the conjecture fails for $n = 4$ and $n = 6$.

n	1	2	3	4	5	6
$J(n)$	1	1	3	1	3	5

Even so, a bad guess isn't a waste of time, because it gets us involved in the problem.

It's back to the drawing board; let's try to make a better guess. Hmmm ... $J(n)$ always seems to be odd. And in fact, there's a good reason for this: The first trip around the circle eliminates all the even numbers. Furthermore, if n itself is an even number, we arrive at a situation similar to what we began with, except that there are only half as many people, and their numbers have changed.

So let's suppose that we have $2n$ people originally. After the first go-round, we're left with

and 3 will be the next to go. This is just like starting out with n people, except that each person's number has been doubled and decreased by 1. That is,

This is the tricky part: We have $J(2n) = \mathit{newnumber}(J(n))$, where $\mathit{newnumber}(k) = 2k - 1$.

$$J(2n) \;=\; 2J(n) - 1\,, \qquad \text{for } n \geqslant 1.$$

We can now go quickly to large n. For example, we know that $J(10) = 5$, so

$$J(20) \;=\; 2J(10) - 1 \;=\; 2{\cdot}5 - 1 \;=\; 9\,.$$

Similarly $J(40) = 17$, and we can deduce that $J(5{\cdot}2^m) = 2^{m+1} + 1$.

But what about the odd case? With $2n+1$ people, it turns out that person number 1 is wiped out just after person number $2n$, and we're left with

Odd case? Hey, leave my brother out of it.

Again we almost have the original situation with n people, but this time their numbers are doubled and *increased* by 1. Thus

$$J(2n+1) \;=\; 2J(n)+1\,, \qquad \text{for } n \geqslant 1.$$

Combining these equations with $J(1)=1$ gives us a recurrence that defines J in all cases:

$$\begin{aligned} J(1) &= 1\,; \\ J(2n) &= 2J(n)-1\,, \qquad \text{for } n \geqslant 1; \\ J(2n+1) &= 2J(n)+1\,, \qquad \text{for } n \geqslant 1. \end{aligned} \tag{1.8}$$

Instead of getting $J(n)$ from $J(n-1)$, this recurrence is much more "efficient," because it reduces n by a factor of 2 or more each time it's applied. We could compute $J(1000000)$, say, with only 19 applications of (1.8). But still, we seek a closed form, because that will be even quicker and more informative. After all, this is a matter of life or death.

Our recurrence makes it possible to build a table of small values very quickly. Perhaps we'll be able to spot a pattern and guess the answer.

n	1	2 3	4 5 6 7	8 9 10 11 12 13 14 15	16
$J(n)$	1	1 3	1 3 5 7	1 3 5 7 9 11 13 15	1

Voilà! It seems we can group by powers of 2 (marked by vertical lines in the table); $J(n)$ is always 1 at the beginning of a group and it increases by 2 within a group. So if we write n in the form $n = 2^m + l$, where 2^m is the largest power of 2 not exceeding n and where l is what's left, the solution to our recurrence seems to be

$$J(2^m + l) \;=\; 2l+1\,, \qquad \text{for } m \geqslant 0 \text{ and } 0 \leqslant l < 2^m. \tag{1.9}$$

(Notice that if $2^m \leqslant n < 2^{m+1}$, the remainder $l = n - 2^m$ satisfies $0 \leqslant l < 2^{m+1} - 2^m = 2^m$.)

We must now prove (1.9). As in the past we use induction, but this time the induction is on m. When $m=0$ we must have $l=0$; thus the basis of

But there's a simpler way! The key fact is that $J(2^m) = 1$ *for all* m*, and this follows immediately from our first equation,* $J(2n) = 2J(n) - 1$. *Hence we know that the first person will survive whenever* n *is a power of* 2. *And in the general case, when* $n = 2^m + l$, *the number of people is reduced to a power of* 2 *after there have been* l *executions. The first remaining person at this point, the survivor, is number* $2l + 1$.

(1.9) reduces to $J(1) = 1$, which is true. The induction step has two parts, depending on whether l is even or odd. If $m > 0$ and $2^m + l = 2n$, then l is even and

$$J(2^m + l) \;=\; 2J(2^{m-1} + l/2) - 1 \;=\; 2(2l/2 + 1) - 1 \;=\; 2l + 1\,,$$

by (1.8) and the induction hypothesis; this is exactly what we want. A similar proof works in the odd case, when $2^m + l = 2n + 1$. We might also note that (1.8) implies the relation

$$J(2n + 1) - J(2n) \;=\; 2.$$

Either way, the induction is complete and (1.9) is established.

To illustrate solution (1.9), let's compute $J(100)$. In this case we have $100 = 2^6 + 36$, so $J(100) = 2 \cdot 36 + 1 = 73$.

Now that we've done the hard stuff (solved the problem) we seek the soft: Every solution to a problem can be generalized so that it applies to a wider class of problems. Once we've learned a technique, it's instructive to look at it closely and see how far we can go with it. Hence, for the rest of this section, we will examine the solution (1.9) and explore some generalizations of the recurrence (1.8). These explorations will uncover the structure that underlies all such problems.

Powers of 2 played an important role in our finding the solution, so it's natural to look at the radix 2 representations of n and $J(n)$. Suppose n's binary expansion is

$$n \;=\; (b_m\, b_{m-1} \ldots b_1\, b_0)_2\,;$$

that is,

$$n \;=\; b_m 2^m \;+\; b_{m-1} 2^{m-1} \;+\; \cdots \;+\; b_1 2 \;+\; b_0\,,$$

where each b_i is either 0 or 1 and where the leading bit b_m is 1. Recalling that $n = 2^m + l$, we have, successively,

$$\begin{aligned}
n &= (1\, b_{m-1}\, b_{m-2} \ldots b_1\, b_0)_2\,,\\
l &= (0\, b_{m-1}\, b_{m-2} \ldots b_1\, b_0)_2\,,\\
2l &= (b_{m-1}\, b_{m-2} \ldots b_1\, b_0\, 0)_2\,,\\
2l + 1 &= (b_{m-1}\, b_{m-2} \ldots b_1\, b_0\, 1)_2\,,\\
J(n) &= (b_{m-1}\, b_{m-2} \ldots b_1\, b_0\, b_m)_2\,.
\end{aligned}$$

(The last step follows because $J(n) = 2l + 1$ and because $b_m = 1$.) We have proved that

$$J\big((b_m\, b_{m-1} \ldots b_1\, b_0)_2\big) \;=\; (b_{m-1} \ldots b_1\, b_0\, b_m)_2\,; \tag{1.10}$$

that is, in the lingo of computer programming, we get $J(n)$ from n by doing a one-bit cyclic shift left! Magic. For example, if $n = 100 = (1100100)_2$ then $J(n) = J\big((1100100)_2\big) = (1001001)_2$, which is $64 + 8 + 1 = 73$. If we had been working all along in binary notation, we probably would have spotted this pattern immediately.

If we start with n and iterate the J function $m + 1$ times, we're doing $m + 1$ one-bit cyclic shifts; so, since n is an $(m+1)$-bit number, we might expect to end up with n again. But this doesn't quite work. For instance if $n = 13$ we have $J\big((1101)_2\big) = (1011)_2$, but then $J\big((1011)_2\big) = (111)_2$ and the process breaks down; the 0 disappears when it becomes the leading bit. In fact, $J(n)$ must always be $\leqslant n$ by definition, since $J(n)$ is the survivor's number; hence if $J(n) < n$ we can never get back up to n by continuing to iterate.

("Iteration" here means applying a function to itself.)

Repeated application of J produces a sequence of decreasing values that eventually reach a "fixed point," where $J(n) = n$. The cyclic shift property makes it easy to see what that fixed point will be: Iterating the function enough times will always produce a pattern of all 1's whose value is $2^{\nu(n)} - 1$, where $\nu(n)$ is the number of 1 bits in the binary representation of n. Thus, since $\nu(13) = 3$, we have

$$\overbrace{J(J(\ldots J}^{\text{2 or more J's}}(13)\ldots)) \;=\; 2^3 - 1 \;=\; 7\,;$$

similarly

$$\overbrace{J(J(\ldots J}^{\text{8 or more}}((101101101101011)_2)\ldots)) \;=\; 2^{10} - 1 \;=\; 1023\,.$$

Curious, but true.

Curiously enough, if M is a compact C^∞ n-manifold ($n > 1$), there exists a differentiable immersion of M into $\mathbf{R}^{2n-\nu(n)}$ but not necessarily into $\mathbf{R}^{2n-\nu(n)-1}$. I wonder if Josephus was secretly a topologist?

Let's return briefly to our first guess, that $J(n) = n/2$ when n is even. This is obviously not true in general, but we can now determine exactly when it *is* true:

$$\begin{aligned} J(n) &= n/2\,, \\ 2l + 1 &= (2^m + l)/2\,, \\ l &= \tfrac{1}{3}(2^m - 2)\,. \end{aligned}$$

If this number $l = \frac{1}{3}(2^m - 2)$ is an integer, then $n = 2^m + l$ will be a solution, because l will be less than 2^m. It's not hard to verify that $2^m - 2$ is a multiple of 3 when m is odd, but not when m is even. (We will study such things in Chapter 4.) Therefore there are infinitely many solutions to the equation

$J(n) = n/2$, beginning as follows:

m	l	$n = 2^m + l$	$J(n) = 2l + 1 = n/2$	n (binary)
1	0	2	1	10
3	2	10	5	1010
5	10	42	21	101010
7	42	170	85	10101010

Notice the pattern in the rightmost column. These are the binary numbers for which cyclic-shifting one place left produces the same result as ordinary-shifting one place right (halving).

OK, we understand the J function pretty well; the next step is to generalize it. What would have happened if our problem had produced a recurrence that was something like (1.8), but with different constants? Then we might not have been lucky enough to guess the solution, because the solution might have been really weird. Let's investigate this by introducing constants α, β, and γ and trying to find a closed form for the more general recurrence

Looks like Greek to me.

$$\begin{aligned} f(1) &= \alpha\,; \\ f(2n) &= 2f(n) + \beta\,, \qquad \text{for } n \geqslant 1; \\ f(2n+1) &= 2f(n) + \gamma\,, \qquad \text{for } n \geqslant 1. \end{aligned} \tag{1.11}$$

(Our original recurrence had $\alpha = 1$, $\beta = -1$, and $\gamma = 1$.) Starting with $f(1) = \alpha$ and working our way up, we can construct the following general table for small values of n:

n	$f(n)$
1	α
2	$2\alpha + \beta$
3	$2\alpha + \gamma$
4	$4\alpha + 3\beta$
5	$4\alpha + 2\beta + \gamma$
6	$4\alpha + \beta + 2\gamma$
7	$4\alpha + 3\gamma$
8	$8\alpha + 7\beta$
9	$8\alpha + 6\beta + \gamma$

(1.12)

It seems that α's coefficient is n's largest power of 2. Furthermore, between powers of 2, β's coefficient decreases by 1 down to 0 and γ's increases by 1 up from 0. Therefore if we express $f(n)$ in the form

$$f(n) \;=\; A(n)\,\alpha \;+\; B(n)\,\beta \;+\; C(n)\,\gamma\,, \tag{1.13}$$

by separating out its dependence on α, β, and γ, it seems that

$$\begin{aligned} A(n) &= 2^m\,; \\ B(n) &= 2^m - 1 - l\,; \\ C(n) &= l\,. \end{aligned} \tag{1.14}$$

Here, as usual, $n = 2^m + l$ and $0 \leqslant l < 2^m$, for $n \geqslant 1$.

It's not terribly hard to prove (1.13) and (1.14) by induction, but the calculations are messy and uninformative. Fortunately there's a better way to proceed, by choosing particular values and then combining them. Let's illustrate this by considering the special case $\alpha = 1$, $\beta = \gamma = 0$, when $f(n)$ is supposed to be equal to $A(n)$: Recurrence (1.11) becomes

Hold onto your hats, this next part is new stuff.

$$\begin{aligned} A(1) &= 1\,; \\ A(2n) &= 2A(n)\,, \qquad \text{for } n \geqslant 1; \\ A(2n+1) &= 2A(n)\,, \qquad \text{for } n \geqslant 1. \end{aligned}$$

Sure enough, it's true (by induction on m) that $A(2^m + l) = 2^m$.

Next, let's use recurrence (1.11) and solution (1.13) *in reverse*, by starting with a simple function $f(n)$ and seeing if there are any constants (α, β, γ) that will define it. Plugging the constant function $f(n) = 1$ into (1.11) says that

A neat idea!

$$\begin{aligned} 1 &= \alpha; \\ 1 &= 2\cdot 1 + \beta; \\ 1 &= 2\cdot 1 + \gamma; \end{aligned}$$

hence the values $(\alpha, \beta, \gamma) = (1, -1, -1)$ satisfying these equations will yield $A(n) - B(n) - C(n) = f(n) = 1$. Similarly, we can plug in $f(n) = n$:

$$\begin{aligned} 1 &= \alpha; \\ 2n &= 2\cdot n + \beta; \\ 2n+1 &= 2\cdot n + \gamma; \end{aligned}$$

These equations hold for all n when $\alpha = 1$, $\beta = 0$, and $\gamma = 1$, so we don't need to prove by induction that these parameters will yield $f(n) = n$. We already *know* that $f(n) = n$ will be the solution in such a case, because the recurrence (1.11) uniquely defines $f(n)$ for every value of n.

And now we're essentially done! We have shown that the functions $A(n)$, $B(n)$, and $C(n)$ of (1.13), which solve (1.11) in general, satisfy the equations

$$\begin{aligned} A(n) &= 2^m\,, \qquad \text{where } n = 2^m + l \text{ and } 0 \leqslant l < 2^m; \\ A(n) - B(n) - C(n) &= 1\,; \\ A(n) + C(n) &= n\,. \end{aligned}$$

Our conjectures in (1.14) follow immediately, since we can solve these equations to get $C(n) = n - A(n) = l$ and $B(n) = A(n) - 1 - C(n) = 2^m - 1 - l$.

Beware: The authors are expecting us to figure out the idea of the repertoire method from seat-of-the-pants examples, instead of giving us a top-down presentation. The method works best with recurrences that are "linear," in the sense that the solutions can be expressed as a sum of arbitrary parameters multiplied by functions of n, as in (1.13). Equation (1.13) is the key.

This approach illustrates a surprisingly useful *repertoire method* for solving recurrences. First we find settings of general parameters for which we know the solution; this gives us a repertoire of special cases that we can solve. Then we obtain the general case by combining the special cases. We need as many independent special solutions as there are independent parameters (in this case three, for α, β, and γ). Exercises 16 and 20 provide further examples of the repertoire approach.

We know that the original J-recurrence has a magical solution, in binary:

$$J\big((b_m\, b_{m-1} \ldots b_1\, b_0)_2\big) \;=\; (b_{m-1} \ldots b_1\, b_0\, b_m)_2\,, \qquad \text{where } b_m = 1.$$

Does the generalized Josephus recurrence admit of such magic?

Sure, why not? We can rewrite the generalized recurrence (1.11) as

$$\begin{aligned} f(1) &= \alpha\,; \\ f(2n+j) &= 2f(n) + \beta_j\,, \qquad \text{for } j = 0, 1 \quad \text{and} \quad n \geqslant 1, \end{aligned} \tag{1.15}$$

if we let $\beta_0 = \beta$ and $\beta_1 = \gamma$. And this recurrence unfolds, binary-wise:

$$\begin{aligned} f\big((b_m\, b_{m-1} \ldots b_1\, b_0)_2\big) &= 2f\big((b_m\, b_{m-1} \ldots b_1)_2\big) + \beta_{b_0} \\ &= 4f\big((b_m\, b_{m-1} \ldots b_2)_2\big) + 2\beta_{b_1} + \beta_{b_0} \\ &\;\;\vdots \\ &= 2^m f\big((b_m)_2\big) + 2^{m-1}\beta_{b_{m-1}} + \cdots + 2\beta_{b_1} + \beta_{b_0} \\ &= 2^m\alpha + 2^{m-1}\beta_{b_{m-1}} + \cdots + 2\beta_{b_1} + \beta_{b_0}\,. \end{aligned}$$

('relax' = 'destroy')

Suppose we now relax the radix 2 notation to allow arbitrary digits instead of just 0 and 1. The derivation above tells us that

$$f\big((b_m\, b_{m-1} \ldots b_1\, b_0)_2\big) \;=\; (\alpha\, \beta_{b_{m-1}}\, \beta_{b_{m-2}} \ldots \beta_{b_1}\, \beta_{b_0})_2\,. \tag{1.16}$$

Nice. We would have seen this pattern earlier if we had written (1.12) in another way:

n	$f(n)$
1	α
2	$2\alpha + \beta$
3	$2\alpha + \gamma$
4	$4\alpha + 2\beta + \beta$
5	$4\alpha + 2\beta + \gamma$
6	$4\alpha + 2\gamma + \beta$
7	$4\alpha + 2\gamma + \gamma$

I think I get it: The binary representations of $A(n)$, $B(n)$, and $C(n)$ have 1's in different positions.

For example, when $n = 100 = (1100100)_2$, our original Josephus values $\alpha = 1$, $\beta = -1$, and $\gamma = 1$ yield

$$\begin{array}{rcccccccccc} n = & (1 & 1 & 0 & 0 & 1 & 0 & 0)_2 & = & 100 \\ \hline f(n) = & (1 & 1 & -1 & -1 & 1 & -1 & -1)_2 & & \\ = & +64 & +32 & -16 & -8 & +4 & -2 & -1 & = & 73 \end{array}$$

as before. The cyclic-shift property follows because each block of binary digits $(1\,0\ldots0\,0)_2$ in the representation of n is transformed into

$$(1\,{-1}\ldots{-1}\,{-1})_2 \;=\; (0\,0\ldots0\,1)_2\,.$$

So our change of notation has given us the compact solution (1.16) to the general recurrence (1.15). If we're really uninhibited we can now generalize even more. The recurrence

"There are two kinds of generalizations. One is cheap and the other is valuable. It is easy to generalize by diluting a little idea with a big terminology. It is much more difficult to prepare a refined and condensed extract from several good ingredients."
— G. Pólya [297]

$$\begin{aligned} f(j) &= \alpha_j\,, && \text{for } 1 \leqslant j < d; \\ f(dn+j) &= cf(n) + \beta_j\,, && \text{for } 0 \leqslant j < d \quad \text{and} \quad n \geqslant 1, \end{aligned} \tag{1.17}$$

is the same as the previous one except that we start with numbers in radix d and produce values in radix c. That is, it has the radix-changing solution

$$f\big((b_m\,b_{m-1}\ldots b_1\,b_0)_d\big) \;=\; (\alpha_{b_m}\,\beta_{b_{m-1}}\,\beta_{b_{m-2}}\ldots\beta_{b_1}\,\beta_{b_0})_c\,. \tag{1.18}$$

For example, suppose that by some stroke of luck we're given the recurrence

$$\begin{aligned} f(1) &= 34\,, \\ f(2) &= 5\,, \\ f(3n) &= 10f(n) + 76\,, && \text{for } n \geqslant 1, \\ f(3n+1) &= 10f(n) - 2\,, && \text{for } n \geqslant 1, \\ f(3n+2) &= 10f(n) + 8\,, && \text{for } n \geqslant 1, \end{aligned}$$

and suppose we want to compute $f(19)$. Here we have $d = 3$ and $c = 10$. Now $19 = (201)_3$, and the radix-changing solution tells us to perform a digit-by-digit replacement from radix 3 to radix 10. So the leading 2 becomes a 5, and the 0 and 1 become 76 and -2, giving

Perhaps this was a stroke of <u>bad</u> luck.

$$f(19) \;=\; f\big((201)_3\big) \;=\; (5\;76\;{-2})_{10} \;=\; 1258\,,$$

which is our answer.

Thus Josephus and the Jewish–Roman war have led us to some interesting general recurrences.

But in general I'm against recurrences of war.

Exercises

Warmups

Please do all the warmups in all the chapters!
— The Mgm't

1 All horses are the same color; we can prove this by induction on the number of horses in a given set. Here's how: "If there's just one horse then it's the same color as itself, so the basis is trivial. For the induction step, assume that there are n horses numbered 1 to n. By the induction hypothesis, horses 1 through $n-1$ are the same color, and similarly horses 2 through n are the same color. But the middle horses, 2 through $n-1$, can't change color when they're in different groups; these are horses, not chameleons. So horses 1 and n must be the same color as well, by transitivity. Thus all n horses are the same color; QED." What, if anything, is wrong with this reasoning?

2 Find the shortest sequence of moves that transfers a tower of n disks from the left peg A to the right peg B, if direct moves between A and B are disallowed. (Each move must be to or from the middle peg. As usual, a larger disk must never appear above a smaller one.)

3 Show that, in the process of transferring a tower under the restrictions of the preceding exercise, we will actually encounter every properly stacked arrangement of n disks on three pegs.

4 Are there any starting and ending configurations of n disks on three pegs that are more than $2^n - 1$ moves apart, under Lucas's original rules?

5 A "Venn diagram" with three overlapping circles is often used to illustrate the eight possible subsets associated with three given sets:

Can the sixteen possibilities that arise with four given sets be illustrated by four overlapping circles?

6 Some of the regions defined by n lines in the plane are infinite, while others are bounded. What's the maximum possible number of bounded regions?

7 Let $H(n) = J(n+1) - J(n)$. Equation (1.8) tells us that $H(2n) = 2$, and $H(2n+1) = J(2n+2) - J(2n+1) = \big(2J(n+1)-1\big) - \big(2J(n)+1\big) = 2H(n)-2$, for all $n \geqslant 1$. Therefore it seems possible to prove that $H(n) = 2$ for all n, by induction on n. What's wrong here?

Homework exercises

8 Solve the recurrence

$$\begin{aligned} Q_0 &= \alpha; \qquad Q_1 = \beta; \\ Q_n &= (1 + Q_{n-1})/Q_{n-2}, \qquad \text{for } n > 1. \end{aligned}$$

Assume that $Q_n \neq 0$ for all $n \geqslant 0$. *Hint:* $Q_4 = (1 + \alpha)/\beta$.

9 Sometimes it's possible to use induction backwards, proving things from n to $n - 1$ instead of vice versa! For example, consider the statement

... now that's a horse of a different color.

$$P(n) : \quad x_1 \ldots x_n \leqslant \left(\frac{x_1 + \cdots + x_n}{n}\right)^n, \quad \text{if } x_1, \ldots, x_n \geqslant 0.$$

This is true when $n = 2$, since $(x_1 + x_2)^2 - 4x_1x_2 = (x_1 - x_2)^2 \geqslant 0$.

a By setting $x_n = (x_1 + \cdots + x_{n-1})/(n - 1)$, prove that $P(n)$ implies $P(n - 1)$ whenever $n > 1$.

b Show that $P(n)$ and $P(2)$ imply $P(2n)$.

c Explain why this implies the truth of $P(n)$ for all n.

10 Let Q_n be the minimum number of moves needed to transfer a tower of n disks from A to B if all moves must be *clockwise*—that is, from A to B, or from B to the other peg, or from the other peg to A. Also let R_n be the minimum number of moves needed to go from B back to A under this restriction. Prove that

$$Q_n = \begin{cases} 0, & \text{if } n = 0; \\ 2R_{n-1} + 1, & \text{if } n > 0; \end{cases} \qquad R_n = \begin{cases} 0, & \text{if } n = 0; \\ Q_n + Q_{n-1} + 1, & \text{if } n > 0. \end{cases}$$

(You need not solve these recurrences; we'll see how to do that in Chapter 7.)

11 A Double Tower of Hanoi contains $2n$ disks of n different sizes, two of each size. As usual, we're required to move only one disk at a time, without putting a larger one over a smaller one.

a How many moves does it take to transfer a double tower from one peg to another, if disks of equal size are indistinguishable from each other?

b What if we are required to reproduce the original top-to-bottom order of all the equal-size disks in the final arrangement? [*Hint:* This is difficult—it's really a "bonus problem."]

12 Let's generalize exercise 11a even further, by assuming that there are n different sizes of disks and exactly m_k disks of size k. Determine $A(m_1, \ldots, m_n)$, the minimum number of moves needed to transfer a tower when equal-size disks are considered to be indistinguishable.

13 What's the maximum number of regions definable by n zig-zag lines,

$ZZ_2 = 12$

each of which consists of two parallel infinite half-lines joined by a straight segment?

Good luck keeping the cheese in position.

14 How many pieces of cheese can you obtain from a single thick piece by making five straight slices? (The cheese must stay in its original position while you do all the cutting, and each slice must correspond to a plane in 3D.) Find a recurrence relation for P_n, the maximum number of three-dimensional regions that can be defined by n different planes.

15 Josephus had a friend who was saved by getting into the next-to-last position. What is $I(n)$, the number of the penultimate survivor when every second person is executed?

16 Use the repertoire method to solve the general four-parameter recurrence

$$\begin{aligned} g(1) &= \alpha; \\ g(2n+j) &= 3g(n) + \gamma n + \beta_j, \qquad \text{for } j = 0, 1 \quad \text{and} \quad n \geqslant 1. \end{aligned}$$

Hint: Try the function $g(n) = n$.

Exam problems

17 If W_n is the minimum number of moves needed to transfer a tower of n disks from one peg to another when there are four pegs instead of three, show that

$$W_{n(n+1)/2} \leqslant 2W_{n(n-1)/2} + T_n, \qquad \text{for } n > 0.$$

(Here $T_n = 2^n - 1$ is the ordinary three-peg number.) Use this to find a closed form $f(n)$ such that $W_{n(n+1)/2} \leqslant f(n)$ for all $n \geqslant 0$.

18 Show that the following set of n bent lines defines Z_n regions, where Z_n is defined in (1.7): The jth bent line, for $1 \leqslant j \leqslant n$, has its zig at $(n^{2j}, 0)$ and goes up through the points $(n^{2j} - n^j, 1)$ and $(n^{2j} - n^j - n^{-n}, 1)$.

19 Is it possible to obtain Z_n regions with n bent lines when the angle at each zig is $30°$?

Is this like a five-star general recurrence?

20 Use the repertoire method to solve the general five-parameter recurrence

$$\begin{aligned} h(1) &= \alpha; \\ h(2n+j) &= 4h(n) + \gamma_j n + \beta_j, \qquad \text{for } j = 0, 1 \quad \text{and} \quad n \geqslant 1. \end{aligned}$$

Hint: Try the functions $h(n) = n$ and $h(n) = n^2$.

21 Suppose there are $2n$ people in a circle; the first n are "good guys" and the last n are "bad guys." Show that there is always an integer m (depending on n) such that, if we go around the circle executing every mth person, all the bad guys are first to go. (For example, when $n = 3$ we can take $m = 5$; when $n = 4$ we can take $m = 30$.)

Bonus problems

22 Show that it's possible to construct a Venn diagram for all 2^n possible subsets of n given sets, using n convex polygons that are congruent to each other and rotated about a common center.

23 Suppose that Josephus finds himself in a given position j, but he has a chance to name the elimination parameter q such that every qth person is executed. Can he always save himself?

Research problems

24 Find all recurrence relations of the form

$$X_n = \frac{1 + a_1 X_{n-1} + \cdots + a_k X_{n-k}}{b_1 X_{n-1} + \cdots + b_k X_{n-k}}$$

whose solution is periodic.

25 Solve infinitely many cases of the four-peg Tower of Hanoi problem by proving that equality holds in the relation of exercise 17.

26 Generalizing exercise 23, let's say that a *Josephus subset* of $\{1, 2, \ldots, n\}$ is a set of k numbers such that, for some q, the people with the other $n-k$ numbers will be eliminated first. (These are the k positions of the "good guys" Josephus wants to save.) It turns out that when $n = 9$, three of the 2^9 possible subsets are non-Josephus, namely $\{1, 2, 5, 8, 9\}$, $\{2, 3, 4, 5, 8\}$, and $\{2, 5, 6, 7, 8\}$. There are 13 non-Josephus sets when $n = 12$, none for any other values of $n \leqslant 12$. Are non-Josephus subsets rare for large n? *Yes, and well done if you find them.*

2

Sums

SUMS ARE EVERYWHERE in mathematics, so we need basic tools to handle them. This chapter develops the notation and general techniques that make summation user-friendly.

2.1 NOTATION

In Chapter 1 we encountered the sum of the first n integers, which we wrote out as $1+2+3+\cdots+(n-1)+n$. The '$\cdots$' in such formulas tells us to complete the pattern established by the surrounding terms. Of course we have to watch out for sums like $1+7+\cdots+41.7$, which are meaningless without a mitigating context. On the other hand, the inclusion of terms like 3 and $(n-1)$ was a bit of overkill; the pattern would presumably have been clear if we had written simply $1+2+\cdots+n$. Sometimes we might even be so bold as to write just $1+\cdots+n$.

We'll be working with sums of the general form

$$a_1 + a_2 + \cdots + a_n\,, \tag{2.1}$$

where each a_k is a number that has been defined somehow. This notation has the advantage that we can "see" the whole sum, almost as if it were written out in full, if we have a good enough imagination.

A term is how long this course lasts.

Each element a_k of a sum is called a *term*. The terms are often specified implicitly as formulas that follow a readily perceived pattern, and in such cases we must sometimes write them in an expanded form so that the meaning is clear. For example, if

$$1+2+\cdots+2^{n-1}$$

is supposed to denote a sum of n terms, not of 2^{n-1}, we should write it more explicitly as

$$2^0+2^1+\cdots+2^{n-1}.$$

The three-dots notation has many uses, but it can be ambiguous and a bit long-winded. Other alternatives are available, notably the delimited form

$$\sum_{k=1}^{n} a_k\,, \tag{2.2}$$

which is called Sigma-notation because it uses the Greek letter $\sum$ (uppercase sigma). This notation tells us to include in the sum precisely those terms a_k whose index k is an integer that lies between the lower and upper limits 1 and n, inclusive. In words, we "sum over k, from 1 to n." Joseph Fourier introduced this delimited $\sum$-notation in 1820, and it soon took the mathematical world by storm.

"Le signe $\sum_{i=1}^{i=\infty}$ indique que l'on doit donner au nombre entier i toutes ses valeurs 1, 2, 3, ..., et prendre la somme des termes."
—J. Fourier [127]

Incidentally, the quantity after $\sum$ (here a_k) is called the *summand*.

The index variable k is said to be *bound* to the $\sum$ sign in (2.2), because the k in a_k is unrelated to appearances of k outside the Sigma-notation. Any other letter could be substituted for k here without changing the meaning of (2.2). The letter i is often used (perhaps because it stands for "index"), but we'll generally sum on k since it's wise to keep i for $\sqrt{-1}$.

Well, I wouldn't want to use a or n as the index variable instead of k in (2.2); those letters are "free variables" that do have meaning outside the $\sum$ here.

It turns out that a generalized Sigma-notation is even more useful than the delimited form: We simply write one or more conditions under the $\sum$, to specify the set of indices over which summation should take place. For example, the sums in (2.1) and (2.2) can also be written as

$$\sum_{1\leqslant k\leqslant n} a_k\,. \tag{2.3}$$

In this particular example there isn't much difference between the new form and (2.2), but the general form allows us to take sums over index sets that aren't restricted to consecutive integers. For example, we can express the sum of the squares of all odd positive integers below 100 as follows:

$$\sum_{\substack{1\leqslant k<100\\ k \text{ odd}}} k^2\,.$$

The delimited equivalent of this sum,

$$\sum_{k=0}^{49} (2k+1)^2\,,$$

is more cumbersome and less clear. Similarly, the sum of reciprocals of all prime numbers between 1 and N is

$$\sum_{\substack{p\leqslant N\\ p \text{ prime}}} \frac{1}{p}\,;$$

the delimited form would require us to write

$$\sum_{k=1}^{\pi(N)} \frac{1}{p_k},$$

where p_k denotes the kth prime and $\pi(N)$ is the number of primes $\leqslant N$. (Incidentally, this sum gives the approximate average number of distinct prime factors of a random integer near N, since about $1/p$ of those integers are divisible by p. Its value for large N is approximately $\ln\ln N + M$, where $M \approx 0.261497212847642783755426838608695859051 5666$ is Mertens's constant [271]; $\ln x$ stands for the natural logarithm of x, and $\ln\ln x$ stands for $\ln(\ln x)$.)

The summation symbol looks like a distorted pacman.

The biggest advantage of general Sigma-notation is that we can manipulate it more easily than the delimited form. For example, suppose we want to change the index variable k to $k+1$. With the general form, we have

$$\sum_{1\leqslant k\leqslant n} a_k = \sum_{1\leqslant k+1\leqslant n} a_{k+1};$$

it's easy to see what's going on, and we can do the substitution almost without thinking. But with the delimited form, we have

$$\sum_{k=1}^{n} a_k = \sum_{k=0}^{n-1} a_{k+1};$$

it's harder to see what's happened, and we're more likely to make a mistake.

A tidy sum.

On the other hand, the delimited form isn't completely useless. It's nice and tidy, and we can write it quickly because (2.2) has seven symbols compared with (2.3)'s eight. Therefore we'll often use $\sum$ with upper and lower delimiters when we state a problem or present a result, but we'll prefer to work with relations-under-$\sum$ when we're manipulating a sum whose index variables need to be transformed.

That's nothing. You should see how many times Σ appears in The Iliad.

The $\sum$ sign occurs more than 1000 times in this book, so we should be sure that we know exactly what it means. Formally, we write

$$\sum_{P(k)} a_k \tag{2.4}$$

as an abbreviation for the sum of all terms a_k such that k is an integer satisfying a given property $P(k)$. (A "property $P(k)$" is any statement about k that can be either true or false.) For the time being, we'll assume that only finitely many integers k satisfying $P(k)$ have $a_k \neq 0$; otherwise infinitely many nonzero numbers are being added together, and things can get a bit

tricky. At the other extreme, if $P(k)$ is false for all integers k, we have an "empty" sum; the value of an empty sum is defined to be zero.

A slightly modified form of (2.4) is used when a sum appears within the text of a paragraph rather than in a displayed equation: We write '$\sum_{P(k)} a_k$', attaching property $P(k)$ as a subscript of $\sum$, so that the formula won't stick out too much. Similarly, '$\sum_{k=1}^{n} a_k$' is a convenient alternative to (2.2) when we want to confine the notation to a single line.

People are often tempted to write

$$\sum_{k=2}^{n-1} k(k-1)(n-k) \qquad \text{instead of} \qquad \sum_{k=0}^{n} k(k-1)(n-k)$$

because the terms for $k = 0$, 1, and n in this sum are zero. Somehow it seems more efficient to add up $n-2$ terms instead of $n+1$ terms. But such temptations should be resisted; efficiency of computation is not the same as efficiency of understanding! We will find it advantageous to keep upper and lower bounds on an index of summation as simple as possible, because sums can be manipulated much more easily when the bounds are simple. Indeed, the form $\sum_{k=2}^{n-1}$ can even be dangerously ambiguous, because its meaning is not at all clear when $n = 0$ or $n = 1$ (see exercise 1). Zero-valued terms cause no harm, and they often save a lot of trouble.

So far the notations we've been discussing are quite standard, but now we are about to make a radical departure from tradition. Kenneth E. Iverson introduced a wonderful idea in his programming language APL [191, page 11; see also 220], and we'll see that it greatly simplifies many of the things we want to do in this book. The idea is simply to enclose a true-or-false statement in brackets, and to say that the result is 1 if the statement is true, 0 if the statement is false. For example,

Hey: The "Kronecker delta" that I've seen in other books (I mean δ_{kn}, which is 1 if $k = n$, 0 otherwise) is just a special case of Iverson's convention: We can write $[k = n]$ instead.

$$[p \text{ prime}] = \begin{cases} 1, & \text{if } p \text{ is a prime number;} \\ 0, & \text{if } p \text{ is not a prime number.} \end{cases}$$

Iverson's convention allows us to express sums with no constraints whatever on the index of summation, because we can rewrite (2.4) in the form

$$\sum_k a_k \big[P(k)\big] . \tag{2.5}$$

If $P(k)$ is false, the term $a_k\big[P(k)\big]$ is zero, so we can safely include it among the terms being summed. This makes it easy to manipulate the index of summation, because we don't have to fuss with boundary conditions.

"I am often surprised by new, important applications [of this notation]."
—B. de Finetti [123]

A slight technicality needs to be mentioned: Sometimes a_k isn't defined for all integers k. We get around this difficulty by assuming that $\big[P(k)\big]$ is "very strongly zero" when $P(k)$ is false; it's so much zero, it makes $a_k\big[P(k)\big]$

equal to zero even when a_k is undefined. For example, if we use Iverson's convention to write the sum of reciprocal primes $\leqslant N$ as

$$\sum_p [p \text{ prime}][p \leqslant N]/p\,,$$

there's no problem of division by zero when $p = 0$, because our convention tells us that $[0 \text{ prime}][0 \leqslant N]/0 = 0$.

Let's sum up what we've discussed so far about sums. There are two good ways to express a sum of terms: One way uses '$\cdots$', the other uses '$\sum$'. The three-dots form often suggests useful manipulations, particularly the combination of adjacent terms, since we might be able to spot a simplifying pattern if we let the whole sum hang out before our eyes. But too much detail can also be overwhelming. Sigma-notation is compact, impressive to family and friends, and often suggestive of manipulations that are not obvious in three-dots form. When we work with Sigma-notation, zero terms are not generally harmful; in fact, zeros often make $\sum$-manipulation easier.

... and it's less likely to lose points on an exam for "lack of rigor."

2.2 SUMS AND RECURRENCES

OK, we understand now how to express sums with fancy notation. But how does a person actually go about finding the value of a sum? One way is to observe that there's an intimate relation between sums and recurrences. The sum

$$S_n = \sum_{k=0}^{n} a_k$$

(Think of S_n as not just a single number, but as a sequence defined for all $n \geqslant 0$.)

is equivalent to the recurrence

$$\begin{aligned} S_0 &= a_0\,; \\ S_n &= S_{n-1} + a_n\,, \qquad \text{for } n > 0. \end{aligned} \tag{2.6}$$

Therefore we can evaluate sums in closed form by using the methods we learned in Chapter 1 to solve recurrences in closed form.

For example, if a_n is equal to a constant plus a multiple of n, the sum-recurrence (2.6) takes the following general form:

$$\begin{aligned} R_0 &= \alpha\,; \\ R_n &= R_{n-1} + \beta + \gamma n\,, \qquad \text{for } n > 0. \end{aligned} \tag{2.7}$$

Proceeding as in Chapter 1, we find $R_1 = \alpha + \beta + \gamma$, $R_2 = \alpha + 2\beta + 3\gamma$, and so on; in general the solution can be written in the form

$$R_n = A(n)\alpha + B(n)\beta + C(n)\gamma\,, \tag{2.8}$$

where $A(n)$, $B(n)$, and $C(n)$ are the coefficients of dependence on the general parameters α, β, and γ.

The repertoire method tells us to try plugging in simple functions of n for R_n, hoping to find constant parameters α, β, and γ where the solution is especially simple. Setting $R_n = 1$ implies $\alpha = 1$, $\beta = 0$, $\gamma = 0$; hence

$$A(n) \;=\; 1\,.$$

Setting $R_n = n$ implies $\alpha = 0$, $\beta = 1$, $\gamma = 0$; hence

$$B(n) \;=\; n\,.$$

Setting $R_n = n^2$ implies $\alpha = 0$, $\beta = -1$, $\gamma = 2$; hence

$$2C(n) - B(n) \;=\; n^2$$

and we have $C(n) = (n^2 + n)/2$. Easy as pie.

Actually easier; $\pi = \sum_{n\geqslant 0} \frac{8}{(4n+1)(4n+3)}$.

Therefore if we wish to evaluate

$$\sum_{k=0}^{n} (a + bk)\,,$$

the sum-recurrence (2.6) boils down to (2.7) with $\alpha = \beta = a$, $\gamma = b$, and the answer is $aA(n) + aB(n) + bC(n) = a(n+1) + b(n+1)n/2$.

Conversely, many recurrences can be reduced to sums; therefore the special methods for evaluating sums that we'll be learning later in this chapter will help us solve recurrences that might otherwise be difficult. The Tower of Hanoi recurrence is a case in point:

$$\begin{aligned} T_0 &= 0\,; \\ T_n &= 2T_{n-1} + 1\,, \qquad \text{for } n > 0. \end{aligned}$$

It can be put into the special form (2.6) if we divide both sides by 2^n:

$$\begin{aligned} T_0/2^0 &= 0; \\ T_n/2^n &= T_{n-1}/2^{n-1} + 1/2^n\,, \qquad \text{for } n > 0. \end{aligned}$$

Now we can set $S_n = T_n/2^n$, and we have

$$\begin{aligned} S_0 &= 0; \\ S_n &= S_{n-1} + 2^{-n}\,, \qquad \text{for } n > 0. \end{aligned}$$

It follows that

$$S_n \;=\; \sum_{k=1}^{n} 2^{-k}.$$

(Notice that we've left the term for $k = 0$ out of this sum.) The sum of the geometric series $2^{-1}+2^{-2}+\cdots+2^{-n} = (\frac{1}{2})^1+(\frac{1}{2})^2+\cdots+(\frac{1}{2})^n$ will be derived later in this chapter; it turns out to be $1-(\frac{1}{2})^n$. Hence $T_n = 2^n S_n = 2^n - 1$.

We have converted T_n to S_n in this derivation by noticing that the recurrence could be divided by 2^n. This trick is a special case of a general technique that can reduce virtually any recurrence of the form

$$a_n T_n \;=\; b_n T_{n-1} + c_n \tag{2.9}$$

to a sum. The idea is to multiply both sides by a *summation factor*, s_n:

$$s_n a_n T_n \;=\; s_n b_n T_{n-1} + s_n c_n\,.$$

This factor s_n is cleverly chosen to make

$$s_n b_n \;=\; s_{n-1} a_{n-1}\,.$$

Then if we write $S_n = s_n a_n T_n$ we have a sum-recurrence,

$$S_n \;=\; S_{n-1} + s_n c_n\,.$$

Hence

$$S_n \;=\; s_0 a_0 T_0 + \sum_{k=1}^{n} s_k c_k \;=\; s_1 b_1 T_0 + \sum_{k=1}^{n} s_k c_k\,,$$

and the solution to the original recurrence (2.9) is

$$T_n \;=\; \frac{1}{s_n a_n}\Bigl(s_1 b_1 T_0 + \sum_{k=1}^{n} s_k c_k\Bigr)\,. \tag{2.10}$$

(The value of s_1 cancels out, so it can be anything but zero.)

For example, when $n = 1$ we get $T_1 = (s_1 b_1 T_0 + s_1 c_1)/s_1 a_1 = (b_1 T_0 + c_1)/a_1$.

But how can we be clever enough to find the right s_n? No problem: The relation $s_n = s_{n-1} a_{n-1}/b_n$ can be unfolded to tell us that the fraction

$$s_n \;=\; \frac{a_{n-1} a_{n-2} \ldots a_1}{b_n b_{n-1} \ldots b_2}\,, \tag{2.11}$$

or any convenient constant multiple of this value, will be a suitable summation factor. For example, the Tower of Hanoi recurrence has $a_n = 1$ and $b_n = 2$; the general method we've just derived says that $s_n = 2^{-n}$ is a good thing to multiply by, if we want to reduce the recurrence to a sum. We don't need a brilliant flash of inspiration to discover this multiplier.

We must be careful, as always, not to divide by zero. The summation-factor method works whenever all the a's and all the b's are nonzero.

Let's apply these ideas to a recurrence that arises in the study of "quicksort," one of the most important methods for sorting data inside a computer. The average number of comparison steps made by quicksort when it is applied to n items in random order satisfies the recurrence

(Quicksort was invented by Hoare in 1962 [189].)

$$\begin{aligned} C_0 &= 0\,; \\ C_n &= n+1+\frac{2}{n}\sum_{k=0}^{n-1} C_k\,, \qquad \text{for } n>0. \end{aligned} \tag{2.12}$$

Hmmm. This looks much scarier than the recurrences we've seen before; it includes a sum over all previous values, and a division by n. Trying small cases gives us some data ($C_1 = 2$, $C_2 = 5$, $C_3 = \frac{26}{3}$) but doesn't do anything to quell our fears.

We can, however, reduce the complexity of (2.12) systematically, by first getting rid of the division and then getting rid of the $\sum$ sign. The idea is to multiply both sides by n, obtaining the relation

$$nC_n \;=\; n^2+n+2\sum_{k=0}^{n-1} C_k\,, \qquad \text{for } n>0;$$

hence, if we replace n by $n-1$,

$$(n-1)C_{n-1} \;=\; (n-1)^2+(n-1)+2\sum_{k=0}^{n-2} C_k\,, \qquad \text{for } n-1>0.$$

We can now subtract the second equation from the first, and the $\sum$ sign disappears:

$$nC_n-(n-1)C_{n-1} \;=\; 2n+2C_{n-1}\,, \qquad \text{for } n>1.$$

It turns out that this relation also holds when $n = 1$, because $C_1 = 2$. Therefore the original recurrence for C_n reduces to a much simpler one:

$$\begin{aligned} C_0 &= 0\,; \\ nC_n &= (n+1)C_{n-1}+2n\,, \qquad \text{for } n>0. \end{aligned}$$

Progress. We're now in a position to apply a summation factor, since this recurrence has the form of (2.9) with $a_n = n$, $b_n = n+1$, and $c_n = 2n$. The general method described on the preceding page tells us to multiply the recurrence through by some multiple of

$$s_n \;=\; \frac{a_{n-1}a_{n-2}\ldots a_1}{b_n b_{n-1}\ldots b_2} \;=\; \frac{(n-1)\cdot(n-2)\cdot\ldots\cdot 1}{(n+1)\cdot n\cdot\ldots\cdot 3} \;=\; \frac{2}{(n+1)n}\,.$$

We started with a $\sum$ in the recurrence, and worked hard to get rid of it. But then after applying a summation factor, we came up with another $\sum$. Are sums good, or bad, or what?

The solution, according to (2.10), is therefore

$$C_n = 2(n+1)\sum_{k=1}^{n}\frac{1}{k+1}.$$

The sum that remains is very similar to a quantity that arises frequently in applications. It arises so often, in fact, that we give it a special name and a special notation:

$$H_n = 1+\frac{1}{2}+\cdots+\frac{1}{n} = \sum_{k=1}^{n}\frac{1}{k}. \tag{2.13}$$

The letter H stands for "harmonic"; H_n is a *harmonic number*, so called because the kth harmonic produced by a violin string is the fundamental tone produced by a string that is $1/k$ times as long.

We can complete our study of the quicksort recurrence (2.12) by putting C_n into closed form; this will be possible if we can express C_n in terms of H_n. The sum in our formula for C_n is

$$\sum_{k=1}^{n}\frac{1}{k+1} = \sum_{1\leqslant k\leqslant n}\frac{1}{k+1}.$$

We can relate this to H_n without much difficulty by changing k to $k-1$ and revising the boundary conditions:

$$\begin{aligned}\sum_{1\leqslant k\leqslant n}\frac{1}{k+1} &= \sum_{1\leqslant k-1\leqslant n}\frac{1}{k}\\ &= \sum_{2\leqslant k\leqslant n+1}\frac{1}{k}\\ &= \left(\sum_{1\leqslant k\leqslant n}\frac{1}{k}\right)-\frac{1}{1}+\frac{1}{n+1} = H_n-\frac{n}{n+1}.\end{aligned}$$

But your spelling is alwrong.

Alright! We have found the sum needed to complete the solution to (2.12): The average number of comparisons made by quicksort when it is applied to n randomly ordered items of data is

$$C_n = 2(n+1)H_n - 2n. \tag{2.14}$$

As usual, we check that small cases are correct: $C_0 = 0$, $C_1 = 2$, $C_2 = 5$.

2.3 MANIPULATION OF SUMS

Not to be confused with finance.

The key to success with sums is an ability to change one $\sum$ into another that is simpler or closer to some goal. And it's easy to do this by learning a few basic rules of transformation and by practicing their use.

Let K be any finite set of integers. Sums over the elements of K can be transformed by using three simple rules:

$$\sum_{k\in K} ca_k = c\sum_{k\in K} a_k\,; \qquad \text{(distributive law)} \tag{2.15}$$

$$\sum_{k\in K}(a_k+b_k) = \sum_{k\in K} a_k + \sum_{k\in K} b_k\,; \qquad \text{(associative law)} \tag{2.16}$$

$$\sum_{k\in K} a_k = \sum_{p(k)\in K} a_{p(k)}\,. \qquad \text{(commutative law)} \tag{2.17}$$

The distributive law allows us to move constants in and out of a $\sum$. The associative law allows us to break a $\sum$ into two parts, or to combine two $\sum$'s into one. The commutative law says that we can reorder the terms in any way we please; here $p(k)$ is any permutation of the set of all integers. For example, if $K=\{-1,0,+1\}$ and if $p(k)=-k$, these three laws tell us respectively that

Why not call it permutative instead of commutative?

$$\begin{aligned}
&ca_{-1}+ca_0+ca_1 = c(a_{-1}+a_0+a_1)\,; && \text{(distributive law)}\\
&(a_{-1}+b_{-1})+(a_0+b_0)+(a_1+b_1)\\
&\quad = (a_{-1}+a_0+a_1)+(b_{-1}+b_0+b_1)\,; && \text{(associative law)}\\
&a_{-1}+a_0+a_1 = a_1+a_0+a_{-1}\,. && \text{(commutative law)}
\end{aligned}$$

Gauss's trick in Chapter 1 can be viewed as an application of these three basic laws. Suppose we want to compute the general sum of an *arithmetic progression*,

$$S = \sum_{0\leqslant k\leqslant n}(a+bk)\,.$$

By the commutative law we can replace k by $n-k$, obtaining

This is something like changing variables inside an integral, but easier.

$$S = \sum_{0\leqslant n-k\leqslant n}\big(a+b(n-k)\big) = \sum_{0\leqslant k\leqslant n}(a+bn-bk)\,.$$

These two equations can be added by using the associative law:

$$2S = \sum_{0\leqslant k\leqslant n}\big((a+bk)+(a+bn-bk)\big) = \sum_{0\leqslant k\leqslant n}(2a+bn)\,.$$

"What's one and one and one and one and one and one and one and one and one and one?" "I don't know," said Alice. "I lost count." "She can't do Addition." —Lewis Carroll [50]

And we can now apply the distributive law and evaluate a trivial sum:

$$2S = (2a+bn)\sum_{0\leqslant k\leqslant n} 1 = (2a+bn)(n+1)\,.$$

Dividing by 2, we have proved that

$$\sum_{k=0}^{n}(a+bk) = (a+\tfrac{1}{2}bn)(n+1)\,. \tag{2.18}$$

The right-hand side can be remembered as the average of the first and last terms, namely $\frac{1}{2}\big(a+(a+bn)\big)$, times the number of terms, namely $(n+1)$.

It's important to bear in mind that the function $p(k)$ in the general commutative law (2.17) is supposed to be a permutation of all the integers. In other words, for every integer n there should be exactly one integer k such that $p(k)=n$. Otherwise the commutative law might fail; exercise 3 illustrates this with a vengeance. Transformations like $p(k)=k+c$ or $p(k)=c-k$, where c is an integer constant, are always permutations, so they always work.

On the other hand, we can relax the permutation restriction a little bit: We need to require only that there be exactly one integer k with $p(k)=n$ when n is an element of the index set K. If $n\notin K$ (that is, if n is not in K), it doesn't matter how often $p(k)=n$ occurs, because such k don't take part in the sum. Thus, for example, we can argue that

$$\sum_{\substack{k\in K\\ k\text{ even}}} a_k = \sum_{\substack{n\in K\\ n\text{ even}}} a_n = \sum_{\substack{2k\in K\\ 2k\text{ even}}} a_{2k} = \sum_{2k\in K} a_{2k}\,, \tag{2.19}$$

since there's exactly one k such that $2k=n$ when $n\in K$ and n is even.

Additional, eh?

Iverson's convention, which allows us to obtain the values 0 or 1 from logical statements in the middle of a formula, can be used together with the distributive, associative, and commutative laws to deduce additional properties of sums. For example, here is an important rule for combining different sets of indices: If K and K' are any sets of integers, then

$$\sum_{k\in K} a_k + \sum_{k\in K'} a_k = \sum_{k\in K\cap K'} a_k + \sum_{k\in K\cup K'} a_k\,. \tag{2.20}$$

This follows from the general formulas

$$\sum_{k\in K} a_k = \sum_{k} a_k\,[k\in K] \tag{2.21}$$

and

$$[k\in K]+[k\in K'] = [k\in K\cap K']+[k\in K\cup K']\,. \tag{2.22}$$

Typically we use rule (2.20) either to combine two almost-disjoint index sets, as in

$$\sum_{k=1}^{m} a_k \;+\; \sum_{k=m}^{n} a_k \;=\; a_m \;+\; \sum_{k=1}^{n} a_k\,, \qquad \text{for } 1 \leqslant m \leqslant n;$$

or to split off a single term from a sum, as in

$$\sum_{0\leqslant k\leqslant n} a_k \;=\; a_0 \;+\; \sum_{1\leqslant k\leqslant n} a_k\,, \qquad \text{for } n \geqslant 0. \tag{2.23}$$

(The two sides of (2.20) have been switched here.)

This operation of splitting off a term is the basis of a *perturbation method* that often allows us to evaluate a sum in closed form. The idea is to start with an unknown sum and call it S_n:

$$S_n \;=\; \sum_{0\leqslant k\leqslant n} a_k\,.$$

(Name and conquer.) Then we rewrite S_{n+1} in two ways, by splitting off both its last term and its first term:

$$\begin{aligned} S_n + a_{n+1} \;=\; \sum_{0\leqslant k\leqslant n+1} a_k \;&=\; a_0 + \sum_{1\leqslant k\leqslant n+1} a_k \\ &=\; a_0 + \sum_{1\leqslant k+1\leqslant n+1} a_{k+1} \\ &=\; a_0 + \sum_{0\leqslant k\leqslant n} a_{k+1}\,. \end{aligned} \tag{2.24}$$

Now we can work on this last sum and try to express it in terms of S_n. If we succeed, we obtain an equation whose solution is the sum we seek.

For example, let's use this approach to find the sum of a general *geometric progression*,

If it's geometric, there should be a geometric proof.

$$S_n \;=\; \sum_{0\leqslant k\leqslant n} ax^k\,.$$

The general perturbation scheme in (2.24) tells us that

$$S_n + ax^{n+1} \;=\; ax^0 + \sum_{0\leqslant k\leqslant n} ax^{k+1}\,,$$

and the sum on the right is $x\sum_{0\leqslant k\leqslant n} ax^k = xS_n$ by the distributive law. Therefore $S_n + ax^{n+1} = a + xS_n$, and we can solve for S_n to obtain

$$\sum_{k=0}^{n} ax^k \;=\; \frac{a - ax^{n+1}}{1-x}, \qquad \text{for } x \neq 1. \tag{2.25}$$

Ah yes, this formula was drilled into me in high school.

(When $x = 1$, the sum is of course simply $(n+1)a$.) The right-hand side can be remembered as the first term included in the sum minus the first term excluded (the term after the last), divided by 1 minus the term ratio.

That was almost too easy. Let's try the perturbation technique on a slightly more difficult sum,

$$S_n = \sum_{0 \leqslant k \leqslant n} k\,2^k .$$

In this case we have $S_0 = 0$, $S_1 = 2$, $S_2 = 10$, $S_3 = 34$, $S_4 = 98$; what is the general formula? According to (2.24) we have

$$S_n + (n+1)2^{n+1} = \sum_{0 \leqslant k \leqslant n} (k+1)2^{k+1} ;$$

so we want to express the right-hand sum in terms of S_n. Well, we can break it into two sums with the help of the associative law,

$$\sum_{0 \leqslant k \leqslant n} k\,2^{k+1} + \sum_{0 \leqslant k \leqslant n} 2^{k+1} ,$$

and the first of the remaining sums is $2S_n$. The other sum is a geometric progression, which equals $(2-2^{n+2})/(1-2) = 2^{n+2}-2$ by (2.25). Therefore we have $S_n + (n+1)2^{n+1} = 2S_n + 2^{n+2} - 2$, and algebra yields

$$\sum_{0 \leqslant k \leqslant n} k\,2^k = (n-1)2^{n+1} + 2 .$$

Now we understand why $S_3 = 34$: It's $32 + 2$, not $2 \cdot 17$.

A similar derivation with x in place of 2 would have given us the equation $S_n + (n+1)x^{n+1} = xS_n + (x - x^{n+2})/(1-x)$; hence we can deduce that

$$\sum_{k=0}^{n} kx^k = \frac{x - (n+1)x^{n+1} + nx^{n+2}}{(1-x)^2} , \qquad \text{for } x \neq 1. \tag{2.26}$$

It's interesting to note that we could have derived this closed form in a completely different way, by using elementary techniques of differential calculus. If we start with the equation

$$\sum_{k=0}^{n} x^k = \frac{1 - x^{n+1}}{1-x}$$

and take the derivative of both sides with respect to x, we get

$$\sum_{k=0}^{n} kx^{k-1} = \frac{(1-x)\big(-(n+1)x^n\big) + 1 - x^{n+1}}{(1-x)^2} = \frac{1 - (n+1)x^n + nx^{n+1}}{(1-x)^2} ,$$

because the derivative of a sum is the sum of the derivatives of its terms. We will see many more connections between calculus and discrete mathematics in later chapters.

2.4 MULTIPLE SUMS

The terms of a sum might be specified by two or more indices, not just by one. For example, here's a double sum of nine terms, governed by two indices j and k:

Oh no, a nine-term governor.

$$\begin{aligned}\sum_{1\leqslant j,k\leqslant 3} a_j b_k \;=\; & a_1b_1 + a_1b_2 + a_1b_3 \\ & + a_2b_1 + a_2b_2 + a_2b_3 \\ & + a_3b_1 + a_3b_2 + a_3b_3\,.\end{aligned}$$

Notice that this doesn't mean to sum over all $j \geqslant 1$ and all $k \leqslant 3$.

We use the same notations and methods for such sums as we do for sums with a single index. Thus, if $P(j,k)$ is a property of j and k, the sum of all terms $a_{j,k}$ such that $P(j,k)$ is true can be written in two ways, one of which uses Iverson's convention and sums over *all* pairs of integers j and k:

$$\sum_{P(j,k)} a_{j,k} \;=\; \sum_{j,k} a_{j,k}\,\big[P(j,k)\big]\,.$$

Only one $\sum$ sign is needed, although there is more than one index of summation; $\sum$ denotes a sum over all combinations of indices that apply.

We also have occasion to use two $\sum$'s, when we're talking about a sum of sums. For example,

$$\sum_j \sum_k a_{j,k}\,\big[P(j,k)\big]$$

is an abbreviation for

$$\sum_j \Bigg(\sum_k a_{j,k}\,\big[P(j,k)\big]\Bigg)\,,$$

which is the sum, over all integers j, of $\sum_k a_{j,k}\,\big[P(j,k)\big]$, the latter being the sum over all integers k of all terms $a_{j,k}$ for which $P(j,k)$ is true. In such cases we say that the double sum is "summed first on k." A sum that depends on more than one index can be summed first on any one of its indices.

Multiple Σ's are evaluated right to left (inside-out).

In this regard we have a basic law called *interchanging the order of summation*, which generalizes the associative law (2.16) we saw earlier:

$$\sum_j \sum_k a_{j,k}\big[P(j,k)\big] \;=\; \sum_{P(j,k)} a_{j,k} \;=\; \sum_k \sum_j a_{j,k}\big[P(j,k)\big]\,. \tag{2.27}$$

The middle term of this law is a sum over two indices. On the left, $\sum_j \sum_k$ stands for summing first on k, then on j. On the right, $\sum_k \sum_j$ stands for summing first on j, then on k. In practice when we want to evaluate a double sum in closed form, it's usually easier to sum it first on one index rather than on the other; we get to choose whichever is more convenient.

Who's panicking? I think this rule is fairly obvious compared to some of the stuff in Chapter 1.

Sums of sums are no reason to panic, but they can appear confusing to a beginner, so let's do some more examples. The nine-term sum we began with provides a good illustration of the manipulation of double sums, because that sum can actually be simplified, and the simplification process is typical of what we can do with $\sum\sum$'s:

$$\begin{aligned}
\sum_{1\leqslant j,k\leqslant 3} a_j b_k &= \sum_{j,k} a_j b_k [1\leqslant j,k\leqslant 3] = \sum_{j,k} a_j b_k [1\leqslant j\leqslant 3][1\leqslant k\leqslant 3] \\
&= \sum_j \sum_k a_j b_k [1\leqslant j\leqslant 3][1\leqslant k\leqslant 3] \\
&= \sum_j a_j [1\leqslant j\leqslant 3] \sum_k b_k [1\leqslant k\leqslant 3] \\
&= \sum_j a_j [1\leqslant j\leqslant 3] \Bigg(\sum_k b_k [1\leqslant k\leqslant 3]\Bigg) \\
&= \Bigg(\sum_j a_j [1\leqslant j\leqslant 3]\Bigg)\Bigg(\sum_k b_k [1\leqslant k\leqslant 3]\Bigg) \\
&= \Bigg(\sum_{j=1}^{3} a_j\Bigg)\Bigg(\sum_{k=1}^{3} b_k\Bigg).
\end{aligned}$$

The first line here denotes a sum of nine terms in no particular order. The second line groups them in threes, $(a_1b_1 + a_1b_2 + a_1b_3) + (a_2b_1 + a_2b_2 + a_2b_3) + (a_3b_1 + a_3b_2 + a_3b_3)$. The third line uses the distributive law to factor out the a's, since a_j and $[1\leqslant j\leqslant 3]$ do not depend on k; this gives $a_1(b_1 + b_2 + b_3) + a_2(b_1 + b_2 + b_3) + a_3(b_1 + b_2 + b_3)$. The fourth line is the same as the third, but with a redundant pair of parentheses thrown in so that the fifth line won't look so mysterious. The fifth line factors out the $(b_1 + b_2 + b_3)$ that occurs for each value of j: $(a_1 + a_2 + a_3)(b_1 + b_2 + b_3)$. The last line is just another way to write the previous line. This method of derivation can be used to prove a *general distributive law*,

$$\sum_{\substack{j\in J\\ k\in K}} a_j b_k = \Bigg(\sum_{j\in J} a_j\Bigg)\Bigg(\sum_{k\in K} b_k\Bigg), \tag{2.28}$$

valid for all sets of indices J and K.

The basic law (2.27) for interchanging the order of summation has many variations, which arise when we want to restrict the ranges of the indices

instead of summing over all integers j and k. These variations come in two flavors, vanilla and rocky road. First, the vanilla version:

$$\sum_{j\in J}\sum_{k\in K} a_{j,k} \;=\; \sum_{\substack{j\in J\\ k\in K}} a_{j,k} \;=\; \sum_{k\in K}\sum_{j\in J} a_{j,k}\,. \tag{2.29}$$

This is just another way to write (2.27), since the Iversonian $[j\in J, k\in K]$ factors into $[j\in J][k\in K]$. The vanilla-flavored law applies whenever the ranges of j and k are independent of each other.

The rocky-road formula for interchange is a little trickier. It applies when the range of an inner sum depends on the index variable of the outer sum:

$$\sum_{j\in J}\sum_{k\in K(j)} a_{j,k} \;=\; \sum_{k\in K'}\sum_{j\in J'(k)} a_{j,k}\,. \tag{2.30}$$

Here the sets J, $K(j)$, K', and $J'(k)$ must be related in such a way that

$$[j\in J]\big[k\in K(j)\big] \;=\; [k\in K']\big[j\in J'(k)\big]\,.$$

A factorization like this is always possible in principle, because we can let $J = K'$ be the set of all integers and $K(j) = J'(k)$ be the basic property $P(j,k)$ that governs a double sum. But there are important special cases where the sets J, $K(j)$, K', and $J'(k)$ have a simple form. These arise frequently in applications. For example, here's a particularly useful factorization:

$$[1\leqslant j\leqslant n][j\leqslant k\leqslant n] \;=\; [1\leqslant j\leqslant k\leqslant n] \;=\; [1\leqslant k\leqslant n][1\leqslant j\leqslant k]\,. \tag{2.31}$$

This Iversonian equation allows us to write

$$\sum_{j=1}^{n}\sum_{k=j}^{n} a_{j,k} \;=\; \sum_{1\leqslant j\leqslant k\leqslant n} a_{j,k} \;=\; \sum_{k=1}^{n}\sum_{j=1}^{k} a_{j,k}\,. \tag{2.32}$$

One of these two sums of sums is usually easier to evaluate than the other; we can use (2.32) to switch from the hard one to the easy one.

(Now is a good time to do warmup exercises 4 and 6.)

(Or to check out the Snickers bar languishing in the freezer.)

Let's apply these ideas to a useful example. Consider the array

$$\begin{bmatrix} a_1a_1 & a_1a_2 & a_1a_3 & \dots & a_1a_n \\ a_2a_1 & a_2a_2 & a_2a_3 & \dots & a_2a_n \\ a_3a_1 & a_3a_2 & a_3a_3 & \dots & a_3a_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_na_1 & a_na_2 & a_na_3 & \dots & a_na_n \end{bmatrix}$$

of n^2 products a_ja_k. Our goal will be to find a simple formula for

$$S_{\triangledown} \;=\; \sum_{1\leqslant j\leqslant k\leqslant n} a_ja_k\,,$$

the sum of all elements on or above the main diagonal of this array. Because $a_j a_k = a_k a_j$, the array is symmetrical about its main diagonal; therefore $S_{\triangledown}$ will be approximately half the sum of *all* the elements (except for a fudge factor that takes account of the main diagonal).

Does rocky road have fudge in it?

Such considerations motivate the following manipulations. We have

$$S_{\triangledown} = \sum_{1\leqslant j\leqslant k\leqslant n} a_j a_k = \sum_{1\leqslant k\leqslant j\leqslant n} a_k a_j = \sum_{1\leqslant k\leqslant j\leqslant n} a_j a_k = S_{\triangle},$$

because we can rename (j,k) as (k,j). Furthermore, since

$$[1\leqslant j\leqslant k\leqslant n]+[1\leqslant k\leqslant j\leqslant n] = [1\leqslant j,k\leqslant n]+[1\leqslant j=k\leqslant n],$$

we have

$$2S_{\triangledown} = S_{\triangledown}+S_{\triangle} = \sum_{1\leqslant j,k\leqslant n} a_j a_k + \sum_{1\leqslant j=k\leqslant n} a_j a_k .$$

The first sum is $\left(\sum_{j=1}^n a_j\right)\left(\sum_{k=1}^n a_k\right) = \left(\sum_{k=1}^n a_k\right)^2$, by the general distributive law (2.28). The second sum is $\sum_{k=1}^n a_k^2$. Therefore we have

$$S_{\triangledown} = \sum_{1\leqslant j\leqslant k\leqslant n} a_j a_k = \frac{1}{2}\left(\left(\sum_{k=1}^n a_k\right)^2 + \sum_{k=1}^n a_k^2\right), \tag{2.33}$$

an expression for the upper triangular sum in terms of simpler single sums.

Encouraged by such success, let's look at another double sum:

$$S = \sum_{1\leqslant j<k\leqslant n} (a_k-a_j)(b_k-b_j).$$

Again we have symmetry when j and k are interchanged:

$$S = \sum_{1\leqslant k<j\leqslant n} (a_j-a_k)(b_j-b_k) = \sum_{1\leqslant k<j\leqslant n} (a_k-a_j)(b_k-b_j).$$

So we can add S to itself, making use of the identity

$$[1\leqslant j<k\leqslant n]+[1\leqslant k<j\leqslant n] = [1\leqslant j,k\leqslant n]-[1\leqslant j=k\leqslant n]$$

to conclude that

$$2S = \sum_{1\leqslant j,k\leqslant n} (a_j-a_k)(b_j-b_k) - \sum_{1\leqslant j=k\leqslant n} (a_j-a_k)(b_j-b_k).$$

The second sum here is zero; what about the first? It expands into four separate sums, each of which is vanilla flavored:

$$\begin{aligned}
&\sum_{1\leqslant j,k\leqslant n} a_j b_j \;-\; \sum_{1\leqslant j,k\leqslant n} a_j b_k \;-\; \sum_{1\leqslant j,k\leqslant n} a_k b_j \;+\; \sum_{1\leqslant j,k\leqslant n} a_k b_k \\
&\quad = 2\sum_{1\leqslant j,k\leqslant n} a_k b_k \;-\; 2\sum_{1\leqslant j,k\leqslant n} a_j b_k \\
&\quad = 2n\sum_{1\leqslant k\leqslant n} a_k b_k \;-\; 2\Bigl(\sum_{k=1}^{n} a_k\Bigr)\Bigl(\sum_{k=1}^{n} b_k\Bigr).
\end{aligned}$$

In the last step both sums have been simplified according to the general distributive law (2.28). If the manipulation of the first sum seems mysterious, here it is again in slow motion:

$$\begin{aligned}
2\sum_{1\leqslant j,k\leqslant n} a_k b_k &= 2\sum_{1\leqslant k\leqslant n}\sum_{1\leqslant j\leqslant n} a_k b_k \\
&= 2\sum_{1\leqslant k\leqslant n} a_k b_k \sum_{1\leqslant j\leqslant n} 1 \\
&= 2\sum_{1\leqslant k\leqslant n} a_k b_k n \;=\; 2n\sum_{1\leqslant k\leqslant n} a_k b_k\,.
\end{aligned}$$

An index variable that doesn't appear in the summand (here j) can simply be eliminated if we multiply what's left by the size of that variable's index set (here n).

Returning to where we left off, we can now divide everything by 2 and rearrange things to obtain an interesting formula:

$$\Bigl(\sum_{k=1}^{n} a_k\Bigr)\Bigl(\sum_{k=1}^{n} b_k\Bigr) \;=\; n\sum_{k=1}^{n} a_k b_k - \sum_{1\leqslant j<k\leqslant n}(a_k-a_j)(b_k-b_j)\,. \qquad (2.34)$$

This identity yields *Chebyshev's monotonic inequalities* as a special case:

(Chebyshev [58] actually proved the analogous result for integrals instead of sums, $(\int_a^b f(x)\,dx)\cdot(\int_a^b g(x)\,dx) \leqslant (b-a)\cdot(\int_a^b f(x)g(x)\,dx)$, *if* $f(x)$ *and* $g(x)$ *are monotone nondecreasing functions.)*

$$\begin{aligned}
\Bigl(\sum_{k=1}^{n} a_k\Bigr)\Bigl(\sum_{k=1}^{n} b_k\Bigr) &\leqslant n\sum_{k=1}^{n} a_k b_k\,, \quad \text{if } a_1\leqslant\cdots\leqslant a_n \text{ and } b_1\leqslant\cdots\leqslant b_n; \\
\Bigl(\sum_{k=1}^{n} a_k\Bigr)\Bigl(\sum_{k=1}^{n} b_k\Bigr) &\geqslant n\sum_{k=1}^{n} a_k b_k\,, \quad \text{if } a_1\leqslant\cdots\leqslant a_n \text{ and } b_1\geqslant\cdots\geqslant b_n.
\end{aligned}$$

(In general, if $a_1\leqslant\cdots\leqslant a_n$ and if p is a permutation of $\{1,\ldots,n\}$, it's not difficult to prove that the largest value of $\sum_{k=1}^{n} a_k b_{p(k)}$ occurs when $b_{p(1)}\leqslant\cdots\leqslant b_{p(n)}$, and the smallest value occurs when $b_{p(1)}\geqslant\cdots\geqslant b_{p(n)}$.)

Multiple summation has an interesting connection with the general operation of changing the index of summation in *single* sums. We know by the commutative law that

$$\sum_{k\in K} a_k = \sum_{p(k)\in K} a_{p(k)}\,,$$

if $p(k)$ is any permutation of the integers. But what happens when we replace k by $f(j)$, where f is an arbitrary function

$$f\colon\ J \to K$$

that takes an integer $j \in J$ into an integer $f(j) \in K$? The general formula for index replacement is

$$\sum_{j\in J} a_{f(j)} = \sum_{k\in K} a_k\, \#f^{-}(k)\,, \tag{2.35}$$

where $\#f^{-}(k)$ stands for the number of elements in the set

$$f^{-}(k) = \bigl\{\, j \bigm| f(j) = k \,\bigr\}\,,$$

that is, the number of values of $j \in J$ such that $f(j)$ equals k.

It's easy to prove (2.35) by interchanging the order of summation,

$$\sum_{j\in J} a_{f(j)} = \sum_{\substack{j\in J\\ k\in K}} a_k \bigl[f(j)=k\bigr] = \sum_{k\in K} a_k \sum_{j\in J} \bigl[f(j)=k\bigr]\,,$$

since $\sum_{j\in J}\bigl[f(j)=k\bigr] = \#f^{-}(k)$. In the special case that f is a one-to-one correspondence between J and K, we have $\#f^{-}(k) = 1$ for all k, and the general formula (2.35) reduces to

My other math teacher calls this a "bijection"; maybe I'll learn to love that word some day.

And then again . . .

$$\sum_{j\in J} a_{f(j)} = \sum_{f(j)\in K} a_{f(j)} = \sum_{k\in K} a_k\,.$$

This is the commutative law (2.17) we had before, slightly disguised.

Our examples of multiple sums so far have all involved general terms like a_k or b_k. But this book is supposed to be concrete, so let's take a look at a multiple sum that involves actual numbers:

Watch out—the authors seem to think that j, k, and n are "actual numbers."

$$S_n = \sum_{1\leqslant j<k\leqslant n} \frac{1}{k-j}\,.$$

For example, $S_1 = 0$; $S_2 = 1$; $S_3 = \frac{1}{2-1} + \frac{1}{3-1} + \frac{1}{3-2} = \frac{5}{2}$.

The normal way to evaluate a double sum is to sum first on j or first on k, so let's explore both options.

$$
\begin{aligned}
S_n &= \sum_{1\leqslant k\leqslant n}\ \sum_{1\leqslant j<k} \frac{1}{k-j} && \text{summing first on } j\\
&= \sum_{1\leqslant k\leqslant n}\ \sum_{1\leqslant k-j<k} \frac{1}{j} && \text{replacing } j \text{ by } k-j\\
&= \sum_{1\leqslant k\leqslant n}\ \sum_{0<j\leqslant k-1} \frac{1}{j} && \text{simplifying the bounds on } j\\
&= \sum_{1\leqslant k\leqslant n} H_{k-1} && \text{by (2.13), the definition of } H_{k-1}\\
&= \sum_{1\leqslant k+1\leqslant n} H_k && \text{replacing } k \text{ by } k+1\\
&= \sum_{0\leqslant k<n} H_k\,. && \text{simplifying the bounds on } k
\end{aligned}
$$

Alas! We don't know how to get a sum of harmonic numbers into closed form. *Get out the whip.*

If we try summing first the other way, we get

$$
\begin{aligned}
S_n &= \sum_{1\leqslant j\leqslant n}\ \sum_{j<k\leqslant n} \frac{1}{k-j} && \text{summing first on } k\\
&= \sum_{1\leqslant j\leqslant n}\ \sum_{j<k+j\leqslant n} \frac{1}{k} && \text{replacing } k \text{ by } k+j\\
&= \sum_{1\leqslant j\leqslant n}\ \sum_{0<k\leqslant n-j} \frac{1}{k} && \text{simplifying the bounds on } k\\
&= \sum_{1\leqslant j\leqslant n} H_{n-j} && \text{by (2.13), the definition of } H_{n-j}\\
&= \sum_{1\leqslant n-j\leqslant n} H_j && \text{replacing } j \text{ by } n-j\\
&= \sum_{0\leqslant j<n} H_j\,. && \text{simplifying the bounds on } j
\end{aligned}
$$

We're back at the same impasse.

But there's *another* way to proceed, if we replace k by $k+j$ *before* deciding to reduce S_n to a sum of sums:

$$
\begin{aligned}
S_n &= \sum_{1\leqslant j<k\leqslant n} \frac{1}{k-j} && \text{recopying the given sum}\\
&= \sum_{1\leqslant j<k+j\leqslant n} \frac{1}{k} && \text{replacing } k \text{ by } k+j
\end{aligned}
$$

It's smart to say $k \leqslant n$ instead of $k \leqslant n-1$ here. Simple bounds save energy.

$$
\begin{aligned}
&= \sum_{1\leqslant k\leqslant n} \sum_{1\leqslant j\leqslant n-k} \frac{1}{k} && \text{summing first on } j \\
&= \sum_{1\leqslant k\leqslant n} \frac{n-k}{k} && \text{the sum on } j \text{ is trivial} \\
&= \sum_{1\leqslant k\leqslant n} \frac{n}{k} - \sum_{1\leqslant k\leqslant n} 1 && \text{by the associative law} \\
&= n\Bigg(\sum_{1\leqslant k\leqslant n} \frac{1}{k}\Bigg) - n && \text{by gosh} \\
&= nH_n - n\,. && \text{by (2.13), the definition of } H_n
\end{aligned}
$$

Aha! We've found S_n. Combining this with the false starts we made gives us a further identity as a bonus:

$$\sum_{0\leqslant k<n} H_k \;=\; nH_n - n\,. \tag{2.36}$$

We can understand the trick that worked here in two ways, one algebraic and one geometric. (1) Algebraically, if we have a double sum whose terms involve $k+f(j)$, where f is an arbitrary function, this example indicates that it's a good idea to try replacing k by $k-f(j)$ and summing on j. (2) Geometrically, we can look at this particular sum S_n as follows, in the case $n=4$:

$$
\begin{array}{lccccccc}
 & k=1 & & k=2 & & k=3 & & k=4 \\
j=1 & & & \frac{1}{1} & + & \frac{1}{2} & + & \frac{1}{3} \\
j=2 & & & & & \frac{1}{1} & + & \frac{1}{2} \\
j=3 & & & & & & & \frac{1}{1} \\
j=4 & & & & & & &
\end{array}
$$

Our first attempts, summing first on j (by columns) or on k (by rows), gave us $H_1 + H_2 + H_3 = H_3 + H_2 + H_1$. The winning idea was essentially to sum by diagonals, getting $\frac{3}{1} + \frac{2}{2} + \frac{1}{3}$.

2.5 GENERAL METHODS

Now let's consolidate what we've learned, by looking at a single example from several different angles. On the next few pages we're going to try to find a closed form for the sum of the first n squares, which we'll call $\square_n$:

$$\square_n \;=\; \sum_{0\leqslant k\leqslant n} k^2\,, \qquad \text{for } n \geqslant 0. \tag{2.37}$$

We'll see that there are at least seven different ways to solve this problem, and in the process we'll learn useful strategies for attacking sums in general.

First, as usual, we look at some small cases.

n	0	1	2	3	4	5	6	7	8	9	10	11	12
n^2	0	1	4	9	16	25	36	49	64	81	100	121	144
$\square_n$	0	1	5	14	30	55	91	140	204	285	385	506	650

No closed form for $\square_n$ is immediately evident; but when we do find one, we can use these values as a check.

Method 0: You could look it up.

A problem like the sum of the first n squares has probably been solved before, so we can most likely find the solution in a handy reference book. Sure enough, page 36 of the *CRC Standard Mathematical Tables* [28] has the answer:

$$\square_n = \frac{n(n+1)(2n+1)}{6}, \qquad \text{for } n \geqslant 0. \tag{2.38}$$

Just to make sure we haven't misread it, we check that this formula correctly gives $\square_5 = 5 \cdot 6 \cdot 11/6 = 55$. Incidentally, page 36 of the *CRC Tables* has further information about the sums of cubes, ..., tenth powers.

The definitive reference for mathematical formulas is the *Handbook of Mathematical Functions*, edited by Abramowitz and Stegun [2]. Pages 813–814 of that book list the values of $\square_n$ for $n \leqslant 100$; and pages 804 and 809 exhibit formulas equivalent to (2.38), together with the analogous formulas for sums of cubes, ..., fifteenth powers, with or without alternating signs.

(Harder sums can be found in Hansen's comprehensive table [178].)

But the best source for answers to questions about sequences is an amazing little book called the *Handbook of Integer Sequences*, by Sloane [330], which lists thousands of sequences by their numerical values. If you come up with a recurrence that you suspect has already been studied, all you have to do is compute enough terms to distinguish your recurrence from other famous ones; then chances are you'll find a pointer to the relevant literature in Sloane's *Handbook*. For example, 1, 5, 14, 30, ... turns out to be Sloane's sequence number 1574, and it's called the sequence of "square pyramidal numbers" (because there are $\square_n$ balls in a pyramid that has a square base of n^2 balls). Sloane gives three references, one of which is to the handbook of Abramowitz and Stegun that we've already mentioned.

Still another way to probe the world's store of accumulated mathematical wisdom is to use a computer program (such as Axiom, MACSYMA, Maple, or Mathematica) that provides tools for symbolic manipulation. Such programs are indispensable, especially for people who need to deal with large formulas.

It's good to be familiar with standard sources of information, because they can be extremely helpful. But Method 0 isn't really consistent with the spirit of this book, because we want to know how to figure out the answers

Or, at least to problems having the same answers as problems that other people have decided to consider.

by ourselves. The look-up method is limited to problems that other people have decided are worth considering; a new problem won't be there.

Method 1: Guess the answer, prove it by induction.

Perhaps a little bird has told us the answer to a problem, or we have arrived at a closed form by some other less-than-rigorous means. Then we merely have to prove that it is correct.

We might, for example, have noticed that the values of $\square_n$ have rather small prime factors, so we may have come up with formula (2.38) as something that works for all small values of n. We might also have conjectured the equivalent formula

$$\square_n = \frac{n(n+\frac{1}{2})(n+1)}{3}, \qquad \text{for } n \geqslant 0, \tag{2.39}$$

which is nicer because it's easier to remember. The preponderance of the evidence supports (2.39), but we must prove our conjectures beyond all reasonable doubt. Mathematical induction was invented for this purpose.

"Well, Your Honor, we know that $\square_0 = 0 = 0(0+\frac{1}{2})(0+1)/3$, so the basis is easy. For the induction, suppose that $n > 0$, and assume that (2.39) holds when n is replaced by $n-1$. Since

$$\square_n = \square_{n-1} + n^2,$$

we have

$$\begin{aligned} 3\square_n &= (n-1)(n-\tfrac{1}{2})(n) \;+\; 3n^2 \\ &= (n^3 - \tfrac{3}{2}n^2 + \tfrac{1}{2}n) \;+\; 3n^2 \\ &= (n^3 + \tfrac{3}{2}n^2 + \tfrac{1}{2}n) \\ &= n(n+\tfrac{1}{2})(n+1)\,. \end{aligned}$$

Therefore (2.39) indeed holds, beyond a reasonable doubt, for all $n \geqslant 0$." Judge Wapner, in his infinite wisdom, agrees.

Induction has its place, and it is somewhat more defensible than trying to look up the answer. But it's still not really what we're seeking. All of the other sums we have evaluated so far in this chapter have been conquered without induction; we should likewise be able to determine a sum like $\square_n$ from scratch. Flashes of inspiration should not be necessary. We should be able to do sums even on our less creative days.

Method 2: Perturb the sum.

So let's go back to the perturbation method that worked so well for the geometric progression (2.25). We extract the first and last terms of $\square_{n+1}$ in

order to get an equation for $\square_n$:

$$\begin{aligned}\square_n + (n+1)^2 \;=\; \sum_{0\leqslant k\leqslant n}(k+1)^2 \;&=\; \sum_{0\leqslant k\leqslant n}(k^2+2k+1)\\ &=\; \sum_{0\leqslant k\leqslant n}k^2 + 2\sum_{0\leqslant k\leqslant n}k + \sum_{0\leqslant k\leqslant n}1\\ &=\; \square_n \;+\; 2\sum_{0\leqslant k\leqslant n}k \;+\; (n+1)\,.\end{aligned}$$

Oops — the $\square_n$'s cancel each other. Occasionally, despite our best efforts, the perturbation method produces something like $\square_n = \square_n$, so we lose.

Seems more like a draw.

On the other hand, this derivation is not a total loss; it does reveal a way to sum the first n integers in closed form,

$$2\sum_{0\leqslant k\leqslant n}k \;=\; (n+1)^2 - (n+1)\,,$$

even though we'd hoped to discover the sum of first integers squared. Could it be that if we start with the sum of the integers cubed, which we might call $\boxed{\square}_n$, we will get an expression for the integers squared? Let's try it.

$$\begin{aligned}\boxed{\square}_n + (n+1)^3 \;=\; \sum_{0\leqslant k\leqslant n}(k+1)^3 \;&=\; \sum_{0\leqslant k\leqslant n}(k^3+3k^2+3k+1)\\ &=\; \boxed{\square}_n + 3\square_n + 3\frac{(n+1)n}{2} + (n+1)\,.\end{aligned}$$

Sure enough, the $\boxed{\square}_n$'s cancel, and we have enough information to determine $\square_n$ without relying on induction:

Method 2′: Perturb your TA.

$$\begin{aligned}3\square_n \;&=\; (n+1)^3 - 3(n+1)n/2 - (n+1)\\ &=\; (n+1)(n^2+2n+1-\tfrac{3}{2}n-1) \;=\; (n+1)(n+\tfrac{1}{2})n\,.\end{aligned}$$

Method 3: Build a repertoire.

A slight generalization of the recurrence (2.7) will also suffice for summands involving n^2. The solution to

$$\begin{aligned}&R_0 = \alpha\,;\\ &R_n = R_{n-1} + \beta + \gamma n + \delta n^2\,, \qquad \text{for } n > 0,\end{aligned} \tag{2.40}$$

will be of the general form

$$R_n \;=\; A(n)\alpha + B(n)\beta + C(n)\gamma + D(n)\delta\,; \tag{2.41}$$

and we have already determined $A(n)$, $B(n)$, and $C(n)$, because (2.40) is the same as (2.7) when $\delta = 0$. If we now plug in $R_n = n^3$, we find that n^3 is the

solution when $\alpha = 0$, $\beta = 1$, $\gamma = -3$, $\delta = 3$. Hence

$$3D(n) - 3C(n) + B(n) \;=\; n^3\,;$$

this determines $D(n)$.

We're interested in the sum $\square_n$, which equals $\square_{n-1} + n^2$; thus we get $\square_n = R_n$ if we set $\alpha = \beta = \gamma = 0$ and $\delta = 1$ in (2.41). Consequently $\square_n = D(n)$. We needn't do the algebra to compute $D(n)$ from $B(n)$ and $C(n)$, since we already know what the answer will be; but doubters among us should be reassured to find that

$$3D(n) \;=\; n^3 + 3C(n) - B(n) \;=\; n^3 + 3\frac{(n+1)n}{2} - n \;=\; n(n+\tfrac{1}{2})(n+1)\,.$$

Method 4: Replace sums by integrals.

People who have been raised on calculus instead of discrete mathematics tend to be more familiar with $\int$ than with $\sum$, so they find it natural to try changing $\sum$ to $\int$. One of our goals in this book is to become so comfortable with $\sum$ that we'll think $\int$ is more difficult than $\sum$ (at least for exact results). But still, it's a good idea to explore the relation between $\sum$ and $\int$, since summation and integration are based on very similar ideas.

In calculus, an integral can be regarded as the area under a curve, and we can approximate this area by adding up the areas of long, skinny rectangles that touch the curve. We can also go the other way if a collection of long, skinny rectangles is given: Since $\square_n$ is the sum of the areas of rectangles whose sizes are 1×1, 1×4, $\ldots$, $1 \times n^2$, it is approximately equal to the area under the curve $f(x) = x^2$ between 0 and n.

The horizontal scale here is ten times the vertical scale.

The area under this curve is $\int_0^n x^2\,dx = n^3/3$; therefore we know that $\square_n$ is approximately $\frac{1}{3}n^3$.

One way to use this fact is to examine the error in the approximation, $E_n = \square_n - \frac{1}{3}n^3$. Since $\square_n$ satisfies the recurrence $\square_n = \square_{n-1} + n^2$, we find that E_n satisfies the simpler recurrence

$$\begin{aligned} E_n \;=\; \square_n - \tfrac{1}{3}n^3 \;=\; \square_{n-1} + n^2 - \tfrac{1}{3}n^3 \;&=\; E_{n-1} + \tfrac{1}{3}(n-1)^3 + n^2 - \tfrac{1}{3}n^3 \\ &=\; E_{n-1} + n - \tfrac{1}{3}\,. \end{aligned}$$

Another way to pursue the integral approach is to find a formula for E_n by summing the areas of the wedge-shaped error terms. We have

$$\begin{aligned} \square_n - \int_0^n x^2\,dx \;&=\; \sum_{k=1}^{n}\left(k^2 - \int_{k-1}^{k} x^2\,dx\right) \\ &=\; \sum_{k=1}^{n}\left(k^2 - \frac{k^3-(k-1)^3}{3}\right) \;=\; \sum_{k=1}^{n}\left(k - \tfrac{1}{3}\right). \end{aligned}$$

This is for people addicted to calculus.

Either way, we could find E_n and then $\square_n$.

Method 5: Expand and contract.

Yet another way to discover a closed form for $\square_n$ is to replace the original sum by a seemingly more complicated double sum that can actually be simplified if we massage it properly:

$$\begin{aligned} \square_n \;&=\; \sum_{1\leqslant k\leqslant n} k^2 \;=\; \sum_{1\leqslant j\leqslant k\leqslant n} k \\ &=\; \sum_{1\leqslant j\leqslant n}\;\sum_{j\leqslant k\leqslant n} k \\ &=\; \sum_{1\leqslant j\leqslant n}\left(\frac{j+n}{2}\right)(n-j+1) \\ &=\; \tfrac{1}{2}\sum_{1\leqslant j\leqslant n}\left(n(n+1)+j-j^2\right) \\ &=\; \tfrac{1}{2}n^2(n+1) + \tfrac{1}{4}n(n+1) - \tfrac{1}{2}\square_n \;=\; \tfrac{1}{2}n(n+\tfrac{1}{2})(n+1) - \tfrac{1}{2}\square_n\,. \end{aligned}$$

(The last step here is something like the last step of the perturbation method, because we get an equation with the unknown quantity on both sides.)

Going from a single sum to a double sum may appear at first to be a backward step, but it's actually progress, because it produces sums that are easier to work with. We can't expect to solve every problem by continually simplifying, simplifying, and simplifying: You can't scale the highest mountain peaks by climbing only uphill.

Method 6: Use finite calculus.

Method 7: Use generating functions.

Stay tuned for still more exciting calculations of $\square_n = \sum_{k=0}^{n} k^2$, as we learn further techniques in the next section and in later chapters.

2.6 FINITE AND INFINITE CALCULUS

We've learned a variety of ways to deal with sums directly. Now it's time to acquire a broader perspective, by looking at the problem of summation from a higher level. Mathematicians have developed a "finite calculus," analogous to the more traditional infinite calculus, by which it's possible to approach summation in a nice, systematic fashion.

Infinite calculus is based on the properties of the *derivative* operator D, defined by

$$Df(x) \;=\; \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}\,.$$

Finite calculus is based on the properties of the *difference* operator Δ, defined by

$$\Delta f(x) \;=\; f(x+1)-f(x)\,. \tag{2.42}$$

This is the finite analog of the derivative in which we restrict ourselves to positive integer values of h. Thus, $h=1$ is the closest we can get to the "limit" as $h\to 0$, and $\Delta f(x)$ is the value of $\big(f(x+h)-f(x)\big)/h$ when $h=1$.

The symbols D and Δ are called *operators* because they operate on functions to give new functions; they are functions of functions that produce functions. If f is a suitably smooth function of real numbers to real numbers, then Df is also a function from reals to reals. And if f is *any* real-to-real function, so is Δf. The values of the functions Df and Δf at a point x are given by the definitions above.

As opposed to a cassette function.

Early on in calculus we learn how D operates on the powers $f(x)=x^m$. In such cases $Df(x)=mx^{m-1}$. We can write this informally with f omitted,

$$D(x^m) \;=\; mx^{m-1}\,.$$

It would be nice if the Δ operator would produce an equally elegant result; unfortunately it doesn't. We have, for example,

$$\Delta(x^3) \;=\; (x+1)^3-x^3 \;=\; 3x^2+3x+1\,.$$

Math power.

But there is a type of "mth power" that does transform nicely under Δ, and this is what makes finite calculus interesting. Such newfangled mth powers are defined by the rule

$$x^{\underline{m}} \;=\; \overbrace{x(x-1)\ldots(x-m+1)}^{m\text{ factors}}\,, \qquad \text{integer } m\geqslant 0. \tag{2.43}$$

Notice the little straight line under the m; this implies that the m factors are supposed to go down and down, stepwise. There's also a corresponding

definition where the factors go up and up:

$$x^{\overline{m}} = \overbrace{x(x+1)\ldots(x+m-1)}^{m \text{ factors}}, \qquad \text{integer } m \geqslant 0. \tag{2.44}$$

When $m = 0$, we have $x^{\underline{0}} = x^{\overline{0}} = 1$, because a product of no factors is conventionally taken to be 1 (just as a sum of no terms is conventionally 0).

The quantity $x^{\underline{m}}$ is called "x to the m falling," if we have to read it aloud; similarly, $x^{\overline{m}}$ is "x to the m rising." These functions are also called *falling factorial powers* and *rising factorial powers*, since they are closely related to the factorial function $n! = n(n-1)\ldots(1)$. In fact, $n! = n^{\underline{n}} = 1^{\overline{n}}$.

Several other notations for factorial powers appear in the mathematical literature, notably "Pochhammer's symbol" $(x)_m$ for $x^{\overline{m}}$ or $x^{\underline{m}}$; notations like $x^{(m)}$ or $x_{(m)}$ are also seen for $x^{\underline{m}}$. But the underline/overline convention is catching on, because it's easy to write, easy to remember, and free of redundant parentheses.

Mathematical terminology is sometimes crazy: Pochhammer [293] actually used the notation $(x)_m$ for the binomial coefficient $\binom{x}{m}$, not for factorial powers.

Falling powers $x^{\underline{m}}$ are especially nice with respect to Δ. We have

$$\begin{aligned}\Delta(x^{\underline{m}}) &= (x+1)^{\underline{m}} - x^{\underline{m}} \\ &= (x+1)x\ldots(x-m+2) \;-\; x\ldots(x-m+2)(x-m+1) \\ &= m\,x(x-1)\ldots(x-m+2)\,,\end{aligned}$$

hence the finite calculus has a handy law to match $D(x^m) = mx^{m-1}$:

$$\Delta(x^{\underline{m}}) = m x^{\underline{m-1}}. \tag{2.45}$$

This is the basic factorial fact.

The operator D of infinite calculus has an inverse, the anti-derivative (or integration) operator $\int$. The Fundamental Theorem of Calculus relates D to $\int$:

$$g(x) = Df(x) \qquad \text{if and only if} \qquad \int g(x)\,dx = f(x) + C\,.$$

Here $\int g(x)\,dx$, the indefinite integral of $g(x)$, is the class of functions whose derivative is $g(x)$. Analogously, Δ has as an inverse, the anti-difference (or summation) operator $\sum$; and there's another Fundamental Theorem:

"Quemadmodum ad differentiam denotandam usi sumus signo Δ, ita summam indicabimus signo Σ. ... ex quo æquatio $z = \Delta y$, si invertatur, dabit quoque $y = \Sigma z + C$."
—L. Euler [110]

$$g(x) = \Delta f(x) \qquad \text{if and only if} \qquad \sum g(x)\,\delta x = f(x) + C\,. \tag{2.46}$$

Here $\sum g(x)\,\delta x$, the *indefinite sum* of $g(x)$, is the class of functions whose *difference* is $g(x)$. (Notice that the lowercase δ relates to uppercase Δ as d relates to D.) The "C" for indefinite integrals is an arbitrary constant; the "C" for indefinite sums is any function $p(x)$ such that $p(x+1) = p(x)$. For

example, C might be the periodic function $a + b \sin 2\pi x$; such functions get washed out when we take differences, just as constants get washed out when we take derivatives. At integer values of x, the function C is constant.

Now we're almost ready for the punch line. Infinite calculus also has *definite* integrals: If $g(x) = Df(x)$, then

$$\int_a^b g(x)\,dx \;=\; f(x)\Big|_a^b \;=\; f(b) - f(a)\,.$$

Therefore finite calculus — ever mimicking its more famous cousin — has definite *sums*: If $g(x) = \Delta f(x)$, then

$$\sum\nolimits_a^b g(x)\,\delta x \;=\; f(x)\Big|_a^b \;=\; f(b) - f(a)\,. \tag{2.47}$$

This formula gives a meaning to the notation $\sum_a^b g(x)\,\delta x$, just as the previous formula defines $\int_a^b g(x)\,dx$.

But what does $\sum_a^b g(x)\,\delta x$ really mean, intuitively? We've defined it by analogy, not by necessity. We want the analogy to hold, so that we can easily remember the rules of finite calculus; but the notation will be useless if we don't understand its significance. Let's try to deduce its meaning by looking first at some special cases, assuming that $g(x) = \Delta f(x) = f(x+1) - f(x)$. If $b = a$, we have

$$\sum\nolimits_a^a g(x)\,\delta x \;=\; f(a) - f(a) \;=\; 0\,.$$

Next, if $b = a + 1$, the result is

$$\sum\nolimits_a^{a+1} g(x)\,\delta x \;=\; f(a+1) - f(a) \;=\; g(a)\,.$$

More generally, if b increases by 1, we have

$$\begin{aligned}\sum\nolimits_a^{b+1} g(x)\,\delta x \;-\; \sum\nolimits_a^b g(x)\,\delta x \;&=\; \big(f(b+1) - f(a)\big) - \big(f(b) - f(a)\big)\\ &=\; f(b+1) - f(b) \;=\; g(b)\,.\end{aligned}$$

These observations, and mathematical induction, allow us to deduce exactly what $\sum_a^b g(x)\,\delta x$ means in general, when a and b are integers with $b \geqslant a$:

$$\sum\nolimits_a^b g(x)\,\delta x \;=\; \sum_{k=a}^{b-1} g(k) \;=\; \sum_{a \leqslant k < b} g(k)\,, \qquad \text{for integers } b \geqslant a. \tag{2.48}$$

You call this a punch line?

In other words, the definite sum is the same as an ordinary sum with limits, but excluding the value at the upper limit.

Let's try to recap this in a slightly different way. Suppose we've been given an unknown sum that's supposed to be evaluated in closed form, and suppose we can write it in the form $\sum_{a\leqslant k<b} g(k) = \sum_a^b g(x)\,\delta x$. The theory of finite calculus tells us that we can express the answer as $f(b) - f(a)$, if we can only find an indefinite sum or anti-difference function f such that $g(x) = f(x+1) - f(x)$. One way to understand this principle is to write $\sum_{a\leqslant k<b} g(k)$ out in full, using the three-dots notation:

$$\begin{aligned}\sum_{a\leqslant k<b} \bigl(f(k+1) - f(k)\bigr) \;&=\; \bigl(f(a{+}1) - f(a)\bigr) + \bigl(f(a{+}2) - f(a{+}1)\bigr) + \cdots \\ &\quad + \bigl(f(b{-}1) - f(b{-}2)\bigr) + \bigl(f(b) - f(b{-}1)\bigr)\,.\end{aligned}$$

Everything on the right-hand side cancels, except $f(b) - f(a)$; so $f(b) - f(a)$ is the value of the sum. (Sums of the form $\sum_{a\leqslant k<b}\bigl(f(k+1) - f(k)\bigr)$ are often called *telescoping*, by analogy with a collapsed telescope, because the thickness of a collapsed telescope is determined solely by the outer radius of the outermost tube and the inner radius of the innermost tube.)

And all this time I thought it was telescoping because it collapsed from a very long expression to a very short one.

But rule (2.48) applies only when $b \geqslant a$; what happens if $b < a$? Well, (2.47) says that we must have

$$\begin{aligned}\sum_a^b g(x)\,\delta x \;&=\; f(b) - f(a) \\ &=\; -\bigl(f(a) - f(b)\bigr) \;=\; -\sum_b^a g(x)\,\delta x\,.\end{aligned}$$

This is analogous to the corresponding equation for definite integration. A similar argument proves $\sum_a^b + \sum_b^c = \sum_a^c$, the summation analog of the identity $\int_a^b + \int_b^c = \int_a^c$. In full garb,

$$\sum_a^b g(x)\,\delta x + \sum_b^c g(x)\,\delta x \;=\; \sum_a^c g(x)\,\delta x\,, \tag{2.49}$$

for all integers a, b, and c.

At this point a few of us are probably starting to wonder what all these parallels and analogies buy us. Well for one, definite summation gives us a simple way to compute sums of falling powers: The basic laws (2.45), (2.47), and (2.48) imply the general law

Others have been wondering this for some time now.

$$\sum_{0\leqslant k<n} k^{\underline{m}} \;=\; \left.\frac{k^{\underline{m+1}}}{m+1}\right|_0^n \;=\; \frac{n^{\underline{m+1}}}{m+1}\,, \qquad \text{for integers } m, n \geqslant 0. \tag{2.50}$$

This formula is easy to remember because it's so much like the familiar $\int_0^n x^m\,dx = n^{m+1}/(m+1)$.

In particular, when $m = 1$ we have $k^{\underline{1}} = k$, so the principles of finite calculus give us an easy way to remember the fact that

$$\sum_{0 \leqslant k < n} k \;=\; \frac{n^{\underline{2}}}{2} \;=\; n(n-1)/2\,.$$

The definite-sum method also gives us an inkling that sums over the range $0 \leqslant k < n$ often turn out to be simpler than sums over $1 \leqslant k \leqslant n$; the former are just $f(n) - f(0)$, while the latter must be evaluated as $f(n+1) - f(1)$.

Ordinary powers can also be summed in this new way, if we first express them in terms of falling powers. For example,

$$k^2 \;=\; k^{\underline{2}} + k^{\underline{1}}\,,$$

hence

$$\sum_{0 \leqslant k < n} k^2 \;=\; \frac{n^{\underline{3}}}{3} + \frac{n^{\underline{2}}}{2} \;=\; \tfrac{1}{3}n(n-1)(n-2+\tfrac{3}{2}) \;=\; \tfrac{1}{3}n(n-\tfrac{1}{2})(n-1)\,.$$

With friends like this...

Replacing n by $n+1$ gives us yet another way to compute the value of our old friend $\square_n = \sum_{0 \leqslant k \leqslant n} k^2$ in closed form.

Gee, that was pretty easy. In fact, it was easier than any of the umpteen other ways that beat this formula to death in the previous section. So let's try to go up a notch, from squares to *cubes*: A simple calculation shows that

$$k^3 \;=\; k^{\underline{3}} + 3k^{\underline{2}} + k^{\underline{1}}\,.$$

(It's always possible to convert between ordinary powers and factorial powers by using Stirling numbers, which we will study in Chapter 6.) Thus

$$\sum_{a \leqslant k < b} k^3 \;=\; \left.\frac{k^{\underline{4}}}{4} + k^{\underline{3}} + \frac{k^{\underline{2}}}{2}\right|_a^b\,.$$

Falling powers are therefore very nice for sums. But do they have any other redeeming features? Must we convert our old friendly ordinary powers to falling powers before summing, but then convert back before we can do anything else? Well, no, it's often possible to work directly with factorial powers, because they have additional properties. For example, just as we have $(x+y)^2 = x^2 + 2xy + y^2$, it turns out that $(x+y)^{\underline{2}} = x^{\underline{2}} + 2x^{\underline{1}}y^{\underline{1}} + y^{\underline{2}}$, and the same analogy holds between $(x+y)^m$ and $(x+y)^{\underline{m}}$. (This "factorial binomial theorem" is proved in exercise 5.37.)

So far we've considered only falling powers that have nonnegative exponents. To extend the analogies with ordinary powers to negative exponents,

we need an appropriate definition of $x^{\underline{m}}$ for $m < 0$. Looking at the sequence

$$\begin{aligned} x^{\underline{3}} &= x(x-1)(x-2)\,, \\ x^{\underline{2}} &= x(x-1)\,, \\ x^{\underline{1}} &= x\,, \\ x^{\underline{0}} &= 1\,, \end{aligned}$$

we notice that to get from $x^{\underline{3}}$ to $x^{\underline{2}}$ to $x^{\underline{1}}$ to $x^{\underline{0}}$ we divide by $x-2$, then by $x-1$, then by x. It seems reasonable (if not imperative) that we should divide by $x+1$ next, to get from $x^{\underline{0}}$ to $x^{\underline{-1}}$, thereby making $x^{\underline{-1}} = 1/(x+1)$. Continuing, the first few negative-exponent falling powers are

$$\begin{aligned} x^{\underline{-1}} &= \frac{1}{x+1}\,, \\ x^{\underline{-2}} &= \frac{1}{(x+1)(x+2)}\,, \\ x^{\underline{-3}} &= \frac{1}{(x+1)(x+2)(x+3)}\,, \end{aligned}$$

and our general definition for negative falling powers is

$$x^{\underline{-m}} = \frac{1}{(x+1)(x+2)\ldots(x+m)}\,, \qquad \text{for } m > 0. \tag{2.51}$$

(It's also possible to define falling powers for real or even complex m, but we will defer that until Chapter 5.)

How can a complex number be even?

With this definition, falling powers have additional nice properties. Perhaps the most important is a general law of exponents, analogous to the law

$$x^{m+n} = x^m x^n$$

for ordinary powers. The falling-power version is

$$x^{\underline{m+n}} = x^{\underline{m}}(x-m)^{\underline{n}}\,, \qquad \text{integers } m \text{ and } n. \tag{2.52}$$

For example, $x^{\underline{2+3}} = x^{\underline{2}}(x-2)^{\underline{3}}$; and with a negative n we have

$$x^{\underline{2-3}} = x^{\underline{2}}(x-2)^{\underline{-3}} = x(x-1)\frac{1}{(x-1)x(x+1)} = \frac{1}{x+1} = x^{\underline{-1}}.$$

If we had chosen to define $x^{\underline{-1}}$ as $1/x$ instead of as $1/(x+1)$, the law of exponents (2.52) would have failed in cases like $m = -1$ and $n = 1$. In fact, we could have used (2.52) to tell us exactly how falling powers ought to be defined in the case of negative exponents, by setting $m = -n$. When an existing notation is being extended to cover more cases, it's always best to formulate definitions in such a way that general laws continue to hold.

Laws have their exponents and their detractors.

Now let's make sure that the crucial difference property holds for our newly defined falling powers. Does $\Delta x^{\underline{m}} = m x^{\underline{m-1}}$ when $m < 0$? If $m = -2$, for example, the difference is

$$\begin{aligned}
\Delta x^{\underline{-2}} &= \frac{1}{(x+2)(x+3)} - \frac{1}{(x+1)(x+2)} \\
&= \frac{(x+1)-(x+3)}{(x+1)(x+2)(x+3)} \\
&= -2x^{\underline{-3}}.
\end{aligned}$$

Yes — it works! A similar argument applies for all $m < 0$.

Therefore the summation property (2.50) holds for negative falling powers as well as positive ones, as long as no division by zero occurs:

$$\sum\nolimits_a^b x^{\underline{m}}\,\delta x = \left.\frac{x^{\underline{m+1}}}{m+1}\right|_a^b, \qquad \text{for } m \neq -1.$$

But what about when $m = -1$? Recall that for integration we use

$$\int_a^b x^{-1}\,dx = \ln x \Big|_a^b$$

when $m = -1$. We'd like to have a finite analog of $\ln x$; in other words, we seek a function $f(x)$ such that

$$x^{\underline{-1}} = \frac{1}{x+1} = \Delta f(x) = f(x+1) - f(x).$$

It's not too hard to see that

$$f(x) = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{x}$$

is such a function, when x is an integer, and this quantity is just the harmonic number H_x of (2.13). Thus H_x is the discrete analog of the continuous $\ln x$. (We will define H_x for noninteger x in Chapter 6, but integer values are good enough for present purposes. We'll also see in Chapter 9 that, for large x, the value of $H_x - \ln x$ is approximately $0.577 + 1/(2x)$. Hence H_x and $\ln x$ are not only analogous, their values usually differ by less than 1.)

0.577 exactly? Maybe they mean $1/\sqrt{3}$. Then again, maybe not.

We can now give a complete description of the sums of falling powers:

$$\sum\nolimits_a^b x^{\underline{m}}\,\delta x = \begin{cases} \left.\dfrac{x^{\underline{m+1}}}{m+1}\right|_a^b, & \text{if } m \neq -1; \\ H_x\Big|_a^b, & \text{if } m = -1. \end{cases} \tag{2.53}$$

This formula indicates why harmonic numbers tend to pop up in the solutions to discrete problems like the analysis of quicksort, just as so-called natural logarithms arise naturally in the solutions to continuous problems.

Now that we've found an analog for $\ln x$, let's see if there's one for e^x. What function $f(x)$ has the property that $\Delta f(x) = f(x)$, corresponding to the identity $De^x = e^x$? Easy:

$$f(x+1) - f(x) \;=\; f(x) \qquad \Longleftrightarrow \qquad f(x+1) \;=\; 2f(x)\,;$$

so we're dealing with a simple recurrence, and we can take $f(x) = 2^x$ as the discrete exponential function.

The difference of c^x is also quite simple, for arbitrary c, namely

$$\Delta(c^x) \;=\; c^{x+1} - c^x \;=\; (c-1)c^x\,.$$

Hence the anti-difference of c^x is $c^x/(c-1)$, if $c \neq 1$. This fact, together with the fundamental laws (2.47) and (2.48), gives us a tidy way to understand the general formula for the sum of a geometric progression:

$$\sum_{a \leqslant k < b} c^k \;=\; \sum\nolimits_a^b c^x\,\delta x \;=\; \frac{c^x}{c-1}\bigg|_a^b \;=\; \frac{c^b - c^a}{c-1}\,, \qquad \text{for } c \neq 1.$$

Every time we encounter a function f that might be useful as a closed form, we can compute its difference $\Delta f = g$; then we have a function g whose indefinite sum $\sum g(x)\,\delta x$ is known. Table 55 is the beginning of a table of difference/anti-difference pairs useful for summation.

'Table 55' is on page 55. Get it?

Despite all the parallels between continuous and discrete math, some continuous notions have no discrete analog. For example, the chain rule of infinite calculus is a handy rule for the derivative of a function of a function; but there's no corresponding chain rule of finite calculus, because there's no nice form for $\Delta f\big(g(x)\big)$. Discrete change-of-variables is hard, except in certain cases like the replacement of x by $c \pm x$.

However, $\Delta\big(f(x)\,g(x)\big)$ *does* have a fairly nice form, and it provides us with a rule for *summation by parts*, the finite analog of what infinite calculus calls integration by parts. Let's recall that the formula

$$D(uv) \;=\; u\,Dv + v\,Du$$

of infinite calculus leads to the rule for integration by parts,

$$\int u\,Dv \;=\; uv - \int v\,Du\,,$$

Table 55 What's the difference?

$f = \Sigma g$	$\Delta f = g$	$f = \Sigma g$	$\Delta f = g$
$x^{\underline{0}} = 1$	0	2^x	2^x
$x^{\underline{1}} = x$	1	c^x	$(c-1)c^x$
$x^{\underline{2}} = x(x-1)$	$2x$	$c^x/(c-1)$	c^x
$x^{\underline{m}}$	$mx^{\underline{m-1}}$	cf	$c\Delta f$
$x^{\underline{m+1}}/(m+1)$	$x^{\underline{m}}$	$f+g$	$\Delta f + \Delta g$
H_x	$x^{\underline{-1}} = 1/(x+1)$	fg	$f\Delta g + Eg\Delta f$

after integration and rearranging terms; we can do a similar thing in finite calculus.

We start by applying the difference operator to the product of two functions $u(x)$ and $v(x)$:

$$\begin{aligned}\Delta\big(u(x)\,v(x)\big) &= u(x{+}1)\,v(x{+}1) - u(x)\,v(x)\\ &= u(x{+}1)\,v(x{+}1) - u(x)\,v(x{+}1)\\ &\qquad\qquad + u(x)\,v(x{+}1) - u(x)\,v(x)\\ &= u(x)\,\Delta v(x) \;+\; v(x{+}1)\,\Delta u(x)\,. \end{aligned} \tag{2.54}$$

This formula can be put into a convenient form using the *shift operator* E, defined by

$$Ef(x) \;=\; f(x+1)\,.$$

Substituting $Ev(x)$ for $v(x{+}1)$ yields a compact rule for the difference of a product:

$$\Delta(uv) \;=\; u\,\Delta v \;+\; Ev\,\Delta u\,. \tag{2.55}$$

Infinite calculus avoids E *here by letting* $1 \to 0$.

(The E is a bit of a nuisance, but it makes the equation correct.) Taking the indefinite sum on both sides of this equation, and rearranging its terms, yields the advertised rule for summation by parts:

$$\sum u\,\Delta v \;=\; uv - \sum Ev\,\Delta u\,. \tag{2.56}$$

As with infinite calculus, limits can be placed on all three terms, making the indefinite sums definite.

This rule is useful when the sum on the left is harder to evaluate than the one on the right. Let's look at an example. The function $\int xe^x\,dx$ is typically integrated by parts; its discrete analog is $\sum x2^x\,\delta x$, which we encountered earlier this chapter in the form $\sum_{k=0}^{n} k\,2^k$. To sum this by parts, we let

I guess $e^x = 2^x$, *for small values of* 1.

$u(x) = x$ and $\Delta v(x) = 2^x$; hence $\Delta u(x) = 1$, $v(x) = 2^x$, and $Ev(x) = 2^{x+1}$. Plugging into (2.56) gives

$$\sum x2^x\,\delta x \;=\; x2^x - \sum 2^{x+1}\,\delta x \;=\; x2^x - 2^{x+1} + C\,.$$

And we can use this to evaluate the sum we did before, by attaching limits:

$$\begin{aligned}\sum_{k=0}^{n} k2^k \;&=\; \sum\nolimits_0^{n+1} x2^x\,\delta x\\ &=\; x2^x - 2^{x+1}\Big|_0^{n+1}\\ &=\; \big((n+1)2^{n+1} - 2^{n+2}\big) - (0\cdot 2^0 - 2^1) \;=\; (n-1)2^{n+1} + 2\,.\end{aligned}$$

It's easier to find the sum this way than to use the perturbation method, because we don't have to think.

The ultimate goal of mathematics is to eliminate all need for intelligent thought.

We stumbled across a formula for $\sum_{0\leqslant k<n} H_k$ earlier in this chapter, and counted ourselves lucky. But we could have found our formula (2.36) systematically, if we had known about summation by parts. Let's demonstrate this assertion by tackling a sum that looks even harder, $\sum_{0\leqslant k<n} kH_k$. The solution is not difficult if we are guided by analogy with $\int x \ln x\,dx$: We take $u(x) = H_x$ and $\Delta v(x) = x = x^{\underline{1}}$, hence $\Delta u(x) = x^{\underline{-1}}$, $v(x) = x^{\underline{2}}/2$, $Ev(x) = (x+1)^{\underline{2}}/2$, and we have

$$\begin{aligned}\sum xH_x\,\delta x \;&=\; \frac{x^{\underline{2}}}{2}H_x - \sum \frac{(x+1)^{\underline{2}}}{2}\,x^{\underline{-1}}\,\delta x\\ &=\; \frac{x^{\underline{2}}}{2}H_x - \frac{1}{2}\sum x^{\underline{1}}\,\delta x\\ &=\; \frac{x^{\underline{2}}}{2}H_x - \frac{x^{\underline{2}}}{4} + C\,.\end{aligned}$$

(In going from the first line to the second, we've combined two falling powers $(x+1)^{\underline{2}}\,x^{\underline{-1}}$ by using the law of exponents (2.52) with $m = -1$ and $n = 2$.) Now we can attach limits and conclude that

$$\sum_{0\leqslant k<n} kH_k \;=\; \sum\nolimits_0^n xH_x\,\delta x \;=\; \frac{n^{\underline{2}}}{2}\Big(H_n - \tfrac{1}{2}\Big)\,. \tag{2.57}$$

2.7 INFINITE SUMS

When we defined $\sum$-notation at the beginning of this chapter, we finessed the question of infinite sums by saying, in essence, "Wait until later. For now, we can assume that all the sums we meet have only finitely many nonzero terms." But the time of reckoning has finally arrived; we must face

This is finesse?

the fact that sums can be infinite. And the truth is that infinite sums are bearers of both good news and bad news.

First, the bad news: It turns out that the methods we've used for manipulating $\sum$'s are *not* always valid when infinite sums are involved. But next, the good news: There is a large, easily understood class of infinite sums for which all the operations we've been performing are perfectly legitimate. The reasons underlying both these news items will be clear after we have looked more closely at the underlying meaning of summation.

Everybody knows what a finite sum is: We add up a bunch of terms, one by one, until they've all been added. But an infinite sum needs to be defined more carefully, lest we get into paradoxical situations.

For example, it seems natural to define things so that the infinite sum

$$S = 1 + \tfrac{1}{2} + \tfrac{1}{4} + \tfrac{1}{8} + \tfrac{1}{16} + \tfrac{1}{32} + \cdots$$

is equal to 2, because if we double it we get

$$2S = 2 + 1 + \tfrac{1}{2} + \tfrac{1}{4} + \tfrac{1}{8} + \tfrac{1}{16} + \cdots = 2 + S\,.$$

On the other hand, this same reasoning suggests that we ought to define

$$T = 1 + 2 + 4 + 8 + 16 + 32 + \cdots$$

Sure: $1 + 2 + 4 + 8 + \cdots$ *is the "infinite precision" representation of the number* -1*, in a binary computer with infinite word size.*

to be -1, for if we double it we get

$$2T = 2 + 4 + 8 + 16 + 32 + 64 + \cdots = T - 1.$$

Something funny is going on; how can we get a negative number by summing positive quantities? It seems better to leave T undefined; or perhaps we should say that $T = \infty$, since the terms being added in T become larger than any fixed, finite number. (Notice that ∞ is another "solution" to the equation $2T = T - 1$; it also "solves" the equation $2S = 2 + S$.)

Let's try to formulate a good definition for the value of a general sum $\sum_{k \in K} a_k$, where K might be infinite. For starters, let's assume that all the terms a_k are *nonnegative*. Then a suitable definition is not hard to find: If there's a bounding constant A such that

$$\sum_{k \in F} a_k \leqslant A$$

for all *finite* subsets $F \subset K$, then we define $\sum_{k \in K} a_k$ to be the *least* such A. (It follows from well-known properties of the real numbers that the set of all such A always contains a smallest element.) But if there's no bounding constant A, we say that $\sum_{k \in K} a_k = \infty$; this means that if A is any real number, there's a set of finitely many terms a_k whose sum exceeds A.

The definition in the previous paragraph has been formulated carefully so that it doesn't depend on any order that might exist in the index set K. Therefore the arguments we are about to make will apply to multiple sums with many indices $k_1, k_2, \ldots$, not just to sums over the set of integers.

The set K might even be uncountable. But only a countable number of terms can be nonzero, if a bounding constant A exists, because at most nA terms are $\geqslant 1/n$.

In the special case that K is the set of nonnegative integers, our definition for nonnegative terms a_k implies that

$$\sum_{k\geqslant 0} a_k \;=\; \lim_{n\to\infty} \sum_{k=0}^{n} a_k\,.$$

Here's why: Any nondecreasing sequence of real numbers has a limit (possibly ∞). If the limit is A, and if F is any finite set of nonnegative integers whose elements are all $\leqslant n$, we have $\sum_{k\in F} a_k \leqslant \sum_{k=0}^{n} a_k \leqslant A$; hence $A=\infty$ or A is a bounding constant. And if A' is any number less than the stated limit A, then there's an n such that $\sum_{k=0}^{n} a_k > A'$; hence the finite set $F=\{0,1,\ldots,n\}$ witnesses to the fact that A' is not a bounding constant.

We can now easily compute the value of certain infinite sums, according to the definition just given. For example, if $a_k = x^k$, we have

$$\sum_{k\geqslant 0} x^k \;=\; \lim_{n\to\infty} \frac{1-x^{n+1}}{1-x} \;=\; \begin{cases} 1/(1-x), & \text{if } 0\leqslant x<1;\\ \infty, & \text{if } x\geqslant 1.\end{cases}$$

In particular, the infinite sums S and T considered a minute ago have the respective values 2 and ∞, just as we suspected. Another interesting example is

$$\begin{aligned}\sum_{k\geqslant 0} \frac{1}{(k+1)(k+2)} \;&=\; \sum_{k\geqslant 0} k^{\underline{-2}}\\ &=\; \lim_{n\to\infty}\sum_{k=0}^{n} k^{\underline{-2}} \;=\; \lim_{n\to\infty} \frac{k^{\underline{-1}}}{-1}\bigg|_0^n \;=\; 1\,.\end{aligned}$$

Now let's consider the case that the sum might have negative terms as well as nonnegative ones. What, for example, should be the value of

$$\sum_{k\geqslant 0} (-1)^k \;=\; 1-1+1-1+1-1+\cdots?$$

"Aggregatum quantitatum $a-a+a-a+a-a$ etc. nunc est $=a$, nunc $=0$, adeoque continuata in infinitum serie ponendus $=a/2$, fateor acumen et veritatem animadversionis tuæ."
— G. Grandi [163]

If we group the terms in pairs, we get

$$(1-1)+(1-1)+(1-1)+\cdots \;=\; 0+0+0+\cdots\,,$$

so the sum comes out zero; but if we start the pairing one step later, we get

$$1-(1-1)-(1-1)-(1-1)-\cdots \;=\; 1-0-0-0-\cdots\,;$$

the sum is 1.

We might also try setting $x = -1$ in the formula $\sum_{k\geqslant 0} x^k = 1/(1-x)$, since we've proved that this formula holds when $0 \leqslant x < 1$; but then we are forced to conclude that the infinite sum is $\frac{1}{2}$, although it's a sum of integers!

Another interesting example is the doubly infinite $\sum_k a_k$ where $a_k = 1/(k+1)$ for $k \geqslant 0$ and $a_k = 1/(k-1)$ for $k < 0$. We can write this as

$$\cdots + (-\tfrac{1}{4}) + (-\tfrac{1}{3}) + (-\tfrac{1}{2}) + 1 + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \cdots . \tag{2.58}$$

If we evaluate this sum by starting at the "center" element and working outward,

$$\cdots + \Big(-\tfrac{1}{4} + \big(-\tfrac{1}{3} + (-\tfrac{1}{2} + (1) + \tfrac{1}{2}) + \tfrac{1}{3}\big) + \tfrac{1}{4}\Big) + \cdots ,$$

we get the value 1; and we obtain the same value 1 if we shift all the parentheses one step to the left,

$$\cdots + \Big(-\tfrac{1}{5} + \big(-\tfrac{1}{4} + (-\tfrac{1}{3} + (-\tfrac{1}{2}) + 1) + \tfrac{1}{2}\big) + \tfrac{1}{3}\Big) + \cdots ,$$

because the sum of all numbers inside the innermost n parentheses is

$$-\frac{1}{n+1} - \frac{1}{n} - \cdots - \frac{1}{2} + 1 + \frac{1}{2} + \cdots + \frac{1}{n-1} \;=\; 1 - \frac{1}{n} - \frac{1}{n+1} .$$

A similar argument shows that the value is 1 if these parentheses are shifted any fixed amount to the left or right; this encourages us to believe that the sum is indeed 1. On the other hand, if we group terms in the following way,

$$\cdots + \Big(-\tfrac{1}{4} + \big(-\tfrac{1}{3} + (-\tfrac{1}{2} + 1 + \tfrac{1}{2}) + \tfrac{1}{3} + \tfrac{1}{4}\big) + \tfrac{1}{5} + \tfrac{1}{6}\Big) + \cdots ,$$

the nth pair of parentheses from inside out contains the numbers

$$-\frac{1}{n+1} - \frac{1}{n} - \cdots - \frac{1}{2} + 1 + \frac{1}{2} + \cdots + \frac{1}{2n-1} + \frac{1}{2n} \;=\; 1 + H_{2n} - H_{n+1} .$$

We'll prove in Chapter 9 that $\lim_{n\to\infty}(H_{2n} - H_{n+1}) = \ln 2$; hence this grouping suggests that the doubly infinite sum should really be equal to $1 + \ln 2$.

There's something flaky about a sum that gives different values when its terms are added up in different ways. Advanced texts on analysis have a variety of definitions by which meaningful values can be assigned to such pathological sums; but if we adopt those definitions, we cannot operate with $\sum$-notation as freely as we have been doing. We don't need the delicate refinements of "conditional convergence" for the purposes of this book; therefore we'll stick to a definition of infinite sums that preserves the validity of all the operations we've been doing in this chapter.

Is this the first page with no graffiti?

In fact, our definition of infinite sums is quite simple. Let K be any set, and let a_k be a real-valued term defined for each $k \in K$. (Here 'k' might actually stand for several indices $k_1, k_2, \ldots$, and K might therefore be multidimensional.) Any real number x can be written as the difference of its positive and negative parts,

$$x = x^+ - x^-, \qquad \text{where } x^+ = x \cdot [x > 0] \text{ and } x^- = -x \cdot [x < 0].$$

(Either $x^+ = 0$ or $x^- = 0$.) We've already explained how to define values for the infinite sums $\sum_{k \in K} a_k^+$ and $\sum_{k \in K} a_k^-$, because a_k^+ and a_k^- are nonnegative. Therefore our general definition is

$$\sum_{k \in K} a_k = \sum_{k \in K} a_k^+ - \sum_{k \in K} a_k^-, \tag{2.59}$$

unless the right-hand sums are both equal to ∞. In the latter case, we leave $\sum_{k \in K} a_k$ undefined.

Let $A^+ = \sum_{k \in K} a_k^+$ and $A^- = \sum_{k \in K} a_k^-$. If A^+ and A^- are both finite, the sum $\sum_{k \in K} a_k$ is said to *converge absolutely* to the value $A = A^+ - A^-$. If $A^+ = \infty$ but A^- is finite, the sum $\sum_{k \in K} a_k$ is said to *diverge* to $+\infty$. Similarly, if $A^- = \infty$ but A^+ is finite, $\sum_{k \in K} a_k$ is said to diverge to $-\infty$. If $A^+ = A^- = \infty$, all bets are off.

In other words, absolute convergence means that the sum of absolute values converges.

We started with a definition that worked for nonnegative terms, then we extended it to real-valued terms. If the terms a_k are complex numbers, we can extend the definition once again, in the obvious way: The sum $\sum_{k \in K} a_k$ is defined to be $\sum_{k \in K} \Re a_k + i \sum_{k \in K} \Im a_k$, where $\Re a_k$ and $\Im a_k$ are the real and imaginary parts of a_k—provided that both of those sums are defined. Otherwise $\sum_{k \in K} a_k$ is undefined. (See exercise 18.)

The bad news, as stated earlier, is that some infinite sums must be left undefined, because the manipulations we've been doing can produce inconsistencies in all such cases. (See exercise 34.) The good news is that all of the manipulations of this chapter are perfectly valid whenever we're dealing with sums that converge absolutely, as just defined.

We can verify the good news by showing that each of our transformation rules preserves the value of all absolutely convergent sums. This means, more explicitly, that we must prove the distributive, associative, and commutative laws, plus the rule for summing first on one index variable; everything else we've done has been derived from those four basic operations on sums.

The distributive law (2.15) can be formulated more precisely as follows: If $\sum_{k \in K} a_k$ converges absolutely to A and if c is any complex number, then $\sum_{k \in K} c a_k$ converges absolutely to cA. We can prove this by breaking the sum into real and imaginary, positive and negative parts as above, and by proving the special case in which $c > 0$ and each term a_k is nonnegative. The

proof in this special case works because $\sum_{k\in F} ca_k = c\sum_{k\in F} a_k$ for all finite sets F; the latter fact follows by induction on the size of F.

The associative law (2.16) can be stated as follows: If $\sum_{k\in K} a_k$ and $\sum_{k\in K} b_k$ converge absolutely to A and B, respectively, then $\sum_{k\in K}(a_k + b_k)$ converges absolutely to $A + B$. This turns out to be a special case of a more general theorem that we will prove shortly.

The commutative law (2.17) doesn't really need to be proved, because we have shown in the discussion following (2.35) how to derive it as a special case of a general rule for interchanging the order of summation.

The main result we need to prove is the fundamental principle of multiple sums: *Absolutely convergent sums over two or more indices can always be summed first with respect to any one of those indices.* Formally, we shall prove that if J and the elements of $\{K_j \mid j \in J\}$ are any sets of indices such that

Best to skim this page the first time you get here.
— Your friendly TA

$$\sum_{\substack{j\in J\\ k\in K_j}} a_{j,k} \quad \text{converges absolutely to } A\,,$$

then there exist complex numbers A_j for each $j \in J$ such that

$$\sum_{k\in K_j} a_{j,k} \quad \text{converges absolutely to } A_j, \quad \text{and}$$
$$\sum_{j\in J} A_j \quad \text{converges absolutely to } A\,.$$

It suffices to prove this assertion when all terms are nonnegative, because we can prove the general case by breaking everything into real and imaginary, positive and negative parts as before. Let's assume therefore that $a_{j,k} \geqslant 0$ for all pairs $(j,k) \in M$, where M is the master index set $\{(j,k) \mid j \in J, k \in K_j\}$.

We are given that $\sum_{(j,k)\in M} a_{j,k}$ is finite, namely that

$$\sum_{(j,k)\in F} a_{j,k} \leqslant A$$

for all finite subsets $F \subseteq M$, and that A is the least such upper bound. If j is any element of J, each sum of the form $\sum_{k\in F_j} a_{j,k}$ where F_j is a finite subset of K_j is bounded above by A. Hence these finite sums have a least upper bound $A_j \geqslant 0$, and $\sum_{k\in K_j} a_{j,k} = A_j$ by definition.

We still need to prove that A is the least upper bound of $\sum_{j\in G} A_j$, for all finite subsets $G \subseteq J$. Suppose that G is a finite subset of J with $\sum_{j\in G} A_j = A' > A$. We can find finite subsets $F_j \subseteq K_j$ such that $\sum_{k\in F_j} a_{j,k} > (A/A')A_j$ for each $j \in G$ with $A_j > 0$. There is at least one such j. But then $\sum_{j\in G, k\in F_j} a_{j,k} > (A/A')\sum_{j\in G} A_j = A$, contradicting the fact that we have

$\sum_{(j,k)\in F} a_{j,k} \leqslant A$ for all finite subsets $F \subseteq M$. Hence $\sum_{j\in G} A_j \leqslant A$, for all finite subsets $G \subseteq J$.

Finally, let A' be any real number less than A. Our proof will be complete if we can find a finite set $G \subseteq J$ such that $\sum_{j\in G} A_j > A'$. We know that there's a finite set $F \subseteq M$ such that $\sum_{(j,k)\in F} a_{j,k} > A'$; let G be the set of j's in this F, and let $F_j = \{k \mid (j,k) \in F\}$. Then $\sum_{j\in G} A_j \geqslant \sum_{j\in G} \sum_{k\in F_j} a_{j,k} = \sum_{(j,k)\in F} a_{j,k} > A'$; QED.

OK, we're now legitimate! Everything we've been doing with infinite sums is justified, as long as there's a finite bound on all finite sums of the absolute values of the terms. Since the doubly infinite sum (2.58) gave us two different answers when we evaluated it in two different ways, its positive terms $1 + \frac{1}{2} + \frac{1}{3} + \cdots$ must diverge to ∞; otherwise we would have gotten the same answer no matter how we grouped the terms.

So why have I been hearing a lot lately about "harmonic convergence"?

Exercises

Warmups

1 What does the notation

$$\sum_{k=4}^{0} q_k$$

mean?

2 Simplify the expression $x \cdot \big([x>0] - [x<0]\big)$.

3 Demonstrate your understanding of $\sum$-notation by writing out the sums

$$\sum_{0\leqslant k\leqslant 5} a_k \qquad \text{and} \qquad \sum_{0\leqslant k^2\leqslant 5} a_{k^2}$$

in full. (Watch out — the second sum is a bit tricky.)

4 Express the triple sum

$$\sum_{1\leqslant i<j<k\leqslant 4} a_{ijk}$$

as a three-fold summation (with three $\sum$'s),

a summing first on k, then j, then i;

b summing first on i, then j, then k.

Also write your triple sums out in full without the $\sum$-notation, using parentheses to show what is being added together first.

5 What's wrong with the following derivation?

$$\left(\sum_{j=1}^{n} a_j\right)\left(\sum_{k=1}^{n} \frac{1}{a_k}\right) = \sum_{j=1}^{n}\sum_{k=1}^{n} \frac{a_j}{a_k} = \sum_{k=1}^{n}\sum_{k=1}^{n} \frac{a_k}{a_k} = \sum_{k=1}^{n} n = n^2 .$$

6 What is the value of $\sum_k [1 \leqslant j \leqslant k \leqslant n]$, as a function of j and n?

Yield to the rising power.

7 Let $\nabla f(x) = f(x) - f(x-1)$. What is $\nabla(x^{\overline{m}})$?

8 What is the value of $0^{\underline{m}}$, when m is a given integer?

9 What is the law of exponents for rising factorial powers, analogous to (2.52)? Use this to define $x^{\overline{-n}}$.

10 The text derives the following formula for the difference of a product:

$$\Delta(uv) = u\,\Delta v + Ev\,\Delta u .$$

How can this formula be correct, when the left-hand side is symmetric with respect to u and v but the right-hand side is not?

Basics

11 The general rule (2.56) for summation by parts is equivalent to

$$\sum_{0\leqslant k<n} (a_{k+1} - a_k) b_k = a_n b_n - a_0 b_0 - \sum_{0\leqslant k<n} a_{k+1}(b_{k+1} - b_k), \quad \text{for } n \geqslant 0.$$

Prove this formula directly by using the distributive, associative, and commutative laws.

12 Show that the function $p(k) = k + (-1)^k c$ is a permutation of the set of all integers, whenever c is an integer.

13 Use the repertoire method to find a closed form for $\sum_{k=0}^{n} (-1)^k k^2$.

14 Evaluate $\sum_{k=1}^{n} k2^k$ by rewriting it as the multiple sum $\sum_{1\leqslant j\leqslant k\leqslant n} 2^k$.

15 Evaluate $\square_n = \sum_{k=1}^{n} k^3$ by the text's Method 5 as follows: First write $\square_n + \square_n = 2\sum_{1\leqslant j\leqslant k\leqslant n} jk$; then apply (2.33).

16 Prove that $x^{\underline{m}}/(x-n)^{\underline{m}} = x^{\underline{n}}/(x-m)^{\underline{n}}$, unless one of the denominators is zero.

17 Show that the following formulas can be used to convert between rising and falling factorial powers, for all integers m:

$$x^{\overline{m}} = (-1)^m (-x)^{\underline{m}} = (x+m-1)^{\underline{m}} = 1/(x-1)^{\underline{-m}};$$
$$x^{\underline{m}} = (-1)^m (-x)^{\overline{m}} = (x-m+1)^{\overline{m}} = 1/(x+1)^{\overline{-m}}.$$

(The answer to exercise 9 defines $x^{\overline{-m}}$.)

18 Let $\Re z$ and $\Im z$ be the real and imaginary parts of the complex number z. The absolute value $|z|$ is $\sqrt{(\Re z)^2 + (\Im z)^2}$. A sum $\sum_{k\in K} a_k$ of complex terms a_k is said to converge absolutely when the real-valued sums $\sum_{k\in K} \Re a_k$ and $\sum_{k\in K} \Im a_k$ both converge absolutely. Prove that $\sum_{k\in K} a_k$ converges absolutely if and only if there is a bounding constant B such that $\sum_{k\in F} |a_k| \leqslant B$ for all finite subsets $F \subseteq K$.

Homework exercises

19 Use a summation factor to solve the recurrence

$$\begin{aligned} T_0 &= 5\,; \\ 2T_n &= nT_{n-1} + 3\cdot n!\,, \qquad \text{for } n > 0. \end{aligned}$$

20 Try to evaluate $\sum_{k=0}^{n} kH_k$ by the perturbation method, but deduce the value of $\sum_{k=0}^{n} H_k$ instead.

21 Evaluate the sums $S_n = \sum_{k=0}^{n}(-1)^{n-k}$, $T_n = \sum_{k=0}^{n}(-1)^{n-k}k$, and $U_n = \sum_{k=0}^{n}(-1)^{n-k}k^2$ by the perturbation method, assuming that $n \geqslant 0$.

22 Prove *Lagrange's identity* (without using induction):

It's hard to prove the identity of somebody who's been dead for 175 years.

$$\sum_{1\leqslant j<k\leqslant n} (a_j b_k - a_k b_j)^2 = \left(\sum_{k=1}^{n} a_k^2\right)\left(\sum_{k=1}^{n} b_k^2\right) - \left(\sum_{k=1}^{n} a_k b_k\right)^2 .$$

Prove, in fact, an identity for the more general double sum

$$\sum_{1\leqslant j<k\leqslant n} (a_j b_k - a_k b_j)(A_j B_k - A_k B_j)\,.$$

23 Evaluate the sum $\sum_{k=1}^{n}(2k+1)/k(k+1)$ in two ways:

a Replace $1/k(k+1)$ by the "partial fractions" $1/k - 1/(k+1)$.

b Sum by parts.

24 What is $\sum_{0\leqslant k<n} H_k/(k+1)(k+2)$? *Hint:* Generalize the derivation of (2.57).

25 The notation $\prod_{k\in K} a_k$ means the product of the numbers a_k for all $k \in K$. Assume for simplicity that $a_k \neq 1$ for only finitely many k; hence infinite products need not be defined. What laws does this $\prod$-notation satisfy, analogous to the distributive, associative, and commutative laws that hold for $\sum$?

This notation was introduced by Jacobi in 1829 [192].

26 Express the double product $\prod_{1\leqslant j\leqslant k\leqslant n} a_j a_k$ in terms of the single product $\prod_{k=1}^{n} a_k$ by manipulating $\prod$-notation. (This exercise gives us a product analog of the upper-triangle identity (2.33).)

27 Compute $\Delta(c^{\underline{x}})$, and use it to deduce the value of $\sum_{k=1}^{n}(-2)^{\underline{k}}/k$.

28 At what point does the following derivation go astray?

$$\begin{aligned}
1 = \sum_{k\geqslant 1}\frac{1}{k(k+1)} &= \sum_{k\geqslant 1}\left(\frac{k}{k+1}-\frac{k-1}{k}\right)\\
&= \sum_{k\geqslant 1}\sum_{j\geqslant 1}\left(\frac{k}{j}[j=k+1]-\frac{j}{k}[j=k-1]\right)\\
&= \sum_{j\geqslant 1}\sum_{k\geqslant 1}\left(\frac{k}{j}[j=k+1]-\frac{j}{k}[j=k-1]\right)\\
&= \sum_{j\geqslant 1}\sum_{k\geqslant 1}\left(\frac{k}{j}[k=j-1]-\frac{j}{k}[k=j+1]\right)\\
&= \sum_{j\geqslant 1}\left(\frac{j-1}{j}-\frac{j}{j+1}\right) = \sum_{j\geqslant 1}\frac{-1}{j(j+1)} = -1\,.
\end{aligned}$$

Exam problems

29 Evaluate the sum $\sum_{k=1}^{n}(-1)^k k/(4k^2-1)$.

30 Cribbage players have long been aware that $15 = 7+8 = 4+5+6 = 1+2+3+4+5$. Find the number of ways to represent 1050 as a sum of consecutive positive integers. (The trivial representation '1050' by itself counts as one way; thus there are four, not three, ways to represent 15 as a sum of consecutive positive integers. Incidentally, a knowledge of cribbage rules is of no use in this problem.)

31 Riemann's zeta function $\zeta(k)$ is defined to be the infinite sum

$$1+\frac{1}{2^k}+\frac{1}{3^k}+\cdots = \sum_{j\geqslant 1}\frac{1}{j^k}\,.$$

Prove that $\sum_{k\geqslant 2}\bigl(\zeta(k)-1\bigr) = 1$. What is the value of $\sum_{k\geqslant 1}\bigl(\zeta(2k)-1\bigr)$?

32 Let $a \dot{-} b = \max(0, a-b)$. Prove that

$$\sum_{k\geqslant 0}\min(k, x\dot{-}k) = \sum_{k\geqslant 0}\bigl(x\dot{-}(2k+1)\bigr)$$

for all real $x \geqslant 0$, and evaluate the sums in closed form.

Bonus problems

33 Let $\bigwedge_{k\in K} a_k$ denote the minimum of the numbers a_k (or their greatest lower bound, if K is infinite), assuming that each a_k is either real or $\pm\infty$. What laws are valid for $\bigwedge$-notation, analogous to those that work for $\sum$ and $\prod$? (See exercise 25.)

The laws of the jungle.

34 Prove that if the sum $\sum_{k\in K} a_k$ is undefined according to (2.59), then it is extremely flaky in the following sense: If A^- and A^+ are any given real numbers, it's possible to find a sequence of finite subsets $F_1 \subset F_2 \subset F_3 \subset \cdots$ of K such that

$$\sum_{k\in F_n} a_k \leqslant A^-, \text{ when } n \text{ is odd}; \quad \sum_{k\in F_n} a_k \geqslant A^+, \text{ when } n \text{ is even}.$$

35 Prove Goldbach's theorem

$$1 = \frac{1}{3}+\frac{1}{7}+\frac{1}{8}+\frac{1}{15}+\frac{1}{24}+\frac{1}{26}+\frac{1}{31}+\frac{1}{35}+\cdots = \sum_{k\in P}\frac{1}{k-1},$$

where P is the set of "perfect powers" defined recursively as follows:

Perfect power corrupts perfectly.

$$P = \{\, m^n \mid m \geqslant 2,\ n \geqslant 2,\ m \notin P \,\}.$$

36 Solomon Golomb's "self-describing sequence" $\langle f(1), f(2), f(3), \ldots\rangle$ is the only nondecreasing sequence of positive integers with the property that it contains exactly $f(k)$ occurrences of k for each k. A few moments' thought reveals that the sequence must begin as follows:

n	1	2	3	4	5	6	7	8	9	10	11	12
$f(n)$	1	2	2	3	3	4	4	4	5	5	5	6

Let $g(n)$ be the largest integer m such that $f(m) = n$. Show that

a $g(n) = \sum_{k=1}^{n} f(k)$.

b $g\bigl(g(n)\bigr) = \sum_{k=1}^{n} kf(k)$.

c $g\bigl(g(g(n))\bigr) = \frac{1}{2}ng(n)\bigl(g(n)+1\bigr) - \frac{1}{2}\sum_{k=1}^{n-1} g(k)\bigl(g(k)+1\bigr)$.

Research problem

37 Will all the $1/k$ by $1/(k+1)$ rectangles, for $k \geqslant 1$, fit together inside a 1 by 1 square? (Recall that their areas sum to 1.)

3

Integer Functions

WHOLE NUMBERS constitute the backbone of discrete mathematics, and we often need to convert from fractions or arbitrary real numbers to integers. Our goal in this chapter is to gain familiarity and fluency with such conversions and to learn some of their remarkable properties.

3.1 FLOORS AND CEILINGS

We start by covering the floor (greatest integer) and ceiling (least integer) functions, which are defined for all real x as follows:

$$\begin{aligned} \lfloor x \rfloor &= \text{the greatest integer less than or equal to } x\,; \\ \lceil x \rceil &= \text{the least integer greater than or equal to } x\,. \end{aligned} \tag{3.1}$$

Kenneth E. Iverson introduced this notation, as well as the names "floor" and "ceiling," early in the 1960s [191, page 12]. He found that typesetters could handle the symbols by shaving the tops and bottoms off of '[' and ']'. His notation has become sufficiently popular that floor and ceiling brackets can now be used in a technical paper without an explanation of what they mean. Until recently, people had most often been writing '$[x]$' for the greatest integer $\leqslant x$, without a good equivalent for the least integer function. Some authors had even tried to use '$]x[$' — with a predictable lack of success.

)Ouch.(

Besides variations in notation, there are variations in the functions themselves. For example, some pocket calculators have an INT function, defined as $\lfloor x \rfloor$ when x is positive and $\lceil x \rceil$ when x is negative. The designers of these calculators probably wanted their INT function to satisfy the identity $\mathrm{INT}(-x) = -\mathrm{INT}(x)$. But we'll stick to our floor and ceiling functions, because they have even nicer properties than this.

One good way to become familiar with the floor and ceiling functions is to understand their graphs, which form staircase-like patterns above and

below the line $f(x) = x$:

We see from the graph that, for example,

$$\lfloor e \rfloor = 2\,, \qquad \lfloor -e \rfloor = -3\,,$$
$$\lceil e \rceil = 3\,, \qquad \lceil -e \rceil = -2\,,$$

since $e = 2.71828\ldots$.

By staring at this illustration we can observe several facts about floors and ceilings. First, since the floor function lies on or below the diagonal line $f(x) = x$, we have $\lfloor x \rfloor \leqslant x$; similarly $\lceil x \rceil \geqslant x$. (This, of course, is quite obvious from the definition.) The two functions are equal precisely at the integer points:

$$\lfloor x \rfloor = x \qquad \Longleftrightarrow \qquad x \text{ is an integer} \qquad \Longleftrightarrow \qquad \lceil x \rceil = x\,.$$

(We use the notation '$\Longleftrightarrow$' to mean "if and only if.") Furthermore, when they differ the ceiling is exactly 1 higher than the floor:

$$\lceil x \rceil - \lfloor x \rfloor = [x \text{ is not an integer}]\,. \tag{3.2}$$

Cute. By Iverson's bracket convention, this is a complete equation.

If we shift the diagonal line down one unit, it lies completely below the floor function, so $x - 1 < \lfloor x \rfloor$; similarly $x + 1 > \lceil x \rceil$. Combining these observations gives us

$$x - 1 < \lfloor x \rfloor \leqslant x \leqslant \lceil x \rceil < x + 1\,. \tag{3.3}$$

Finally, the functions are reflections of each other about both axes:

$$\lfloor -x \rfloor = -\lceil x \rceil\,; \qquad \lceil -x \rceil = -\lfloor x \rfloor\,. \tag{3.4}$$

Thus each is easily expressible in terms of the other. This fact helps to explain why the ceiling function once had no notation of its own. But we see ceilings often enough to warrant giving them special symbols, just as we have adopted special notations for rising powers as well as falling powers. Mathematicians have long had both sine and cosine, tangent and cotangent, secant and cosecant, max and min; now we also have both floor and ceiling.

Next week we're getting walls.

To actually prove properties about the floor and ceiling functions, rather than just to observe such facts graphically, the following four rules are especially useful:

$$\begin{aligned}
\lfloor x\rfloor = n \quad &\Longleftrightarrow \quad n \leqslant x < n+1\,, && \text{(a)}\\
\lfloor x\rfloor = n \quad &\Longleftrightarrow \quad x-1 < n \leqslant x\,, && \text{(b)}\\
\lceil x\rceil = n \quad &\Longleftrightarrow \quad n-1 < x \leqslant n\,, && \text{(c)}\\
\lceil x\rceil = n \quad &\Longleftrightarrow \quad x \leqslant n < x+1\,. && \text{(d)}
\end{aligned} \tag{3.5}$$

(We assume in all four cases that n is an integer and that x is real.) Rules (a) and (c) are immediate consequences of definition (3.1); rules (b) and (d) are the same but with the inequalities rearranged so that n is in the middle.

It's possible to move an integer term in or out of a floor (or ceiling):

$$\lfloor x+n\rfloor \;=\; \lfloor x\rfloor + n\,, \qquad \text{integer } n. \tag{3.6}$$

(Because rule (3.5(a)) says that this assertion is equivalent to the inequalities $\lfloor x\rfloor + n \leqslant x+n < \lfloor x\rfloor + n + 1$.) But similar operations, like moving out a constant factor, cannot be done in general. For example, we have $\lfloor nx\rfloor \neq n\lfloor x\rfloor$ when $n = 2$ and $x = 1/2$. This means that floor and ceiling brackets are comparatively inflexible. We are usually happy if we can get rid of them or if we can prove anything at all when they are present.

It turns out that there are many situations in which floor and ceiling brackets are redundant, so that we can insert or delete them at will. For example, any inequality between a real and an integer is equivalent to a floor or ceiling inequality between integers:

$$\begin{aligned}
x < n \quad &\Longleftrightarrow \quad \lfloor x\rfloor < n\,, && \text{(a)}\\
n < x \quad &\Longleftrightarrow \quad n < \lceil x\rceil\,, && \text{(b)}\\
x \leqslant n \quad &\Longleftrightarrow \quad \lceil x\rceil \leqslant n\,, && \text{(c)}\\
n \leqslant x \quad &\Longleftrightarrow \quad n \leqslant \lfloor x\rfloor\,. && \text{(d)}
\end{aligned} \tag{3.7}$$

These rules are easily proved. For example, if $x < n$ then surely $\lfloor x\rfloor < n$, since $\lfloor x\rfloor \leqslant x$. Conversely, if $\lfloor x\rfloor < n$ then we must have $x < n$, since $x < \lfloor x\rfloor + 1$ and $\lfloor x\rfloor + 1 \leqslant n$.

It would be nice if the four rules in (3.7) were as easy to remember as they are to prove. Each inequality without floor or ceiling corresponds to the

same inequality with floor or with ceiling; but we need to think twice before deciding which of the two is appropriate.

The difference between x and $\lfloor x \rfloor$ is called the *fractional part* of x, and it arises often enough in applications to deserve its own notation:

$$\{x\} = x - \lfloor x \rfloor . \tag{3.8}$$

Hmmm. We'd better not write $\{x\}$ for the fractional part when it could be confused with the set containing x as its only element.

We sometimes call $\lfloor x \rfloor$ the *integer part* of x, since $x = \lfloor x \rfloor + \{x\}$. If a real number x can be written in the form $x = n + \theta$, where n is an integer and $0 \leqslant \theta < 1$, we can conclude by (3.5(a)) that $n = \lfloor x \rfloor$ and $\theta = \{x\}$.

Identity (3.6) doesn't hold if n is an arbitrary real. But we can deduce that there are only two possibilities for $\lfloor x + y \rfloor$ in general: If we write $x = \lfloor x \rfloor + \{x\}$ and $y = \lfloor y \rfloor + \{y\}$, then we have $\lfloor x + y \rfloor = \lfloor x \rfloor + \lfloor y \rfloor + \lfloor \{x\} + \{y\} \rfloor$. And since $0 \leqslant \{x\} + \{y\} < 2$, we find that sometimes $\lfloor x + y \rfloor$ is $\lfloor x \rfloor + \lfloor y \rfloor$, otherwise it's $\lfloor x \rfloor + \lfloor y \rfloor + 1$.

The second case occurs if and only if there's a "carry" at the position of the decimal point, when the fractional parts $\{x\}$ and $\{y\}$ are added together.

3.2 FLOOR/CEILING APPLICATIONS

We've now seen the basic tools for handling floors and ceilings. Let's put them to use, starting with an easy problem: What's $\lceil \lg 35 \rceil$? (Following a suggestion of Edward M. Reingold, we use 'lg' to denote the base-2 logarithm.) Well, since $2^5 < 35 \leqslant 2^6$, we can take logs to get $5 < \lg 35 \leqslant 6$; so relation (3.5(c)) tells us that $\lceil \lg 35 \rceil = 6$.

Note that the number 35 is six bits long when written in radix 2 notation: $35 = (100011)_2$. Is it always true that $\lceil \lg n \rceil$ is the length of n written in binary? Not quite. We also need six bits to write $32 = (100000)_2$. So $\lceil \lg n \rceil$ is the wrong answer to the problem. (It fails only when n is a power of 2, but that's infinitely many failures.) We can find a correct answer by realizing that it takes m bits to write each number n such that $2^{m-1} \leqslant n < 2^m$; thus (3.5(a)) tells us that $m - 1 = \lfloor \lg n \rfloor$, so $m = \lfloor \lg n \rfloor + 1$. That is, we need $\lfloor \lg n \rfloor + 1$ bits to express n in binary, for all $n > 0$. Alternatively, a similar derivation yields the answer $\lceil \lg(n+1) \rceil$; this formula holds for $n = 0$ as well, if we're willing to say that it takes zero bits to write $n = 0$ in binary.

Let's look next at expressions with several floors or ceilings. What is $\lceil \lfloor x \rfloor \rceil$? Easy — since $\lfloor x \rfloor$ is an integer, $\lceil \lfloor x \rfloor \rceil$ is just $\lfloor x \rfloor$. So is any other expression with an innermost $\lfloor x \rfloor$ surrounded by any number of floors or ceilings.

Here's a tougher problem: Prove or disprove the assertion

$$\lfloor \sqrt{\lfloor x \rfloor} \rfloor = \lfloor \sqrt{x} \rfloor , \qquad \text{real } x \geqslant 0. \tag{3.9}$$

Equality obviously holds when x is an integer, because $x = \lfloor x \rfloor$. And there's equality in the special cases $\pi = 3.14159\ldots$, $e = 2.71828\ldots$, and $\phi = (1 + \sqrt{5})/2 = 1.61803\ldots$, because we get $1 = 1$. Our failure to find a counterexample suggests that equality holds in general, so let's try to prove it.

(Of course π, e, and ϕ are the obvious first real numbers to try, aren't they?)

Skepticism is healthy only to a limited extent. Being skeptical about proofs and programs (particularly your own) will probably keep your grades healthy and your job fairly secure. But applying that much skepticism will probably also keep you shut away working all the time, instead of letting you get out for exercise and relaxation.
Too much skepticism is an open invitation to the state of rigor mortis, where you become so worried about being correct and rigorous that you never get anything finished.
—A skeptic

Incidentally, when we're faced with a "prove or disprove," we're usually better off trying first to disprove with a counterexample, for two reasons: A disproof is potentially easier (we need just one counterexample); and nit-picking arouses our creative juices. Even if the given assertion is true, our search for a counterexample often leads us to a proof, as soon as we see why a counterexample is impossible. Besides, it's healthy to be skeptical.

If we try to prove that $\lfloor\sqrt{\lfloor x\rfloor}\rfloor = \lfloor\sqrt{x}\rfloor$ with the help of calculus, we might start by decomposing x into its integer and fractional parts $\lfloor x\rfloor + \{x\} = n + \theta$ and then expanding the square root using the binomial theorem: $(n+\theta)^{1/2} = n^{1/2} + n^{-1/2}\theta/2 - n^{-3/2}\theta^2/8 + \cdots$. But this approach gets pretty messy.

It's much easier to use the tools we've developed. Here's a possible strategy: Somehow strip off the outer floor and square root of $\lfloor\sqrt{\lfloor x\rfloor}\rfloor$, then remove the inner floor, then add back the outer stuff to get $\lfloor\sqrt{x}\rfloor$. OK. We let $m = \lfloor\sqrt{\lfloor x\rfloor}\rfloor$ and invoke (3.5(a)), giving $m \leqslant \sqrt{\lfloor x\rfloor} < m+1$. That removes the outer floor bracket without losing any information. Squaring, since all three expressions are nonnegative, we have $m^2 \leqslant \lfloor x\rfloor < (m+1)^2$. That gets rid of the square root. Next we remove the floor, using (3.7(d)) for the left inequality and (3.7(a)) for the right: $m^2 \leqslant x < (m+1)^2$. It's now a simple matter to retrace our steps, taking square roots to get $m \leqslant \sqrt{x} < m+1$ and invoking (3.5(a)) to get $m = \lfloor\sqrt{x}\rfloor$. Thus $\lfloor\sqrt{\lfloor x\rfloor}\rfloor = m = \lfloor\sqrt{x}\rfloor$; the assertion is true. Similarly, we can prove that

$$\lceil\sqrt{\lceil x\rceil}\rceil \;=\; \lceil\sqrt{x}\rceil\,, \qquad \text{real } x \geqslant 0.$$

The proof we just found doesn't rely heavily on the properties of square roots. A closer look shows that we can generalize the ideas and prove much more: Let $f(x)$ be any continuous, monotonically increasing function with the property that

$$f(x) \;=\; \text{integer} \qquad \Longrightarrow \qquad x \;=\; \text{integer}\,.$$

(The symbol '$\Longrightarrow$' means "implies.") Then we have

(This observation was made by R. J. McEliece when he was an undergrad.)

$$\lfloor f(x)\rfloor \;=\; \lfloor f(\lfloor x\rfloor)\rfloor \qquad \text{and} \qquad \lceil f(x)\rceil \;=\; \lceil f(\lceil x\rceil)\rceil, \tag{3.10}$$

whenever $f(x)$, $f(\lfloor x\rfloor)$, and $f(\lceil x\rceil)$ are defined. Let's prove this general property for ceilings, since we did floors earlier and since the proof for floors is almost the same. If $x = \lceil x\rceil$, there's nothing to prove. Otherwise $x < \lceil x\rceil$, and $f(x) < f(\lceil x\rceil)$ since f is increasing. Hence $\lceil f(x)\rceil \leqslant \lceil f(\lceil x\rceil)\rceil$, since $\lceil\,\rceil$ is nondecreasing. If $\lceil f(x)\rceil < \lceil f(\lceil x\rceil)\rceil$, there must be a number y such that $x \leqslant y < \lceil x\rceil$ and $f(y) = \lceil f(x)\rceil$, since f is continuous. This y is an integer, because of f's special property. But there cannot be an integer strictly between x and $\lceil x\rceil$. This contradiction implies that we must have $\lceil f(x)\rceil = \lceil f(\lceil x\rceil)\rceil$.

An important special case of this theorem is worth noting explicitly:

$$\left\lfloor \frac{x+m}{n} \right\rfloor = \left\lfloor \frac{\lfloor x \rfloor + m}{n} \right\rfloor \qquad \text{and} \qquad \left\lceil \frac{x+m}{n} \right\rceil = \left\lceil \frac{\lceil x \rceil + m}{n} \right\rceil, \quad (3.11)$$

if m and n are integers and the denominator n is positive. For example, let $m = 0$; we have $\lfloor \lfloor \lfloor x/10 \rfloor / 10 \rfloor / 10 \rfloor = \lfloor x/1000 \rfloor$. Dividing thrice by 10 and throwing off digits is the same as dividing by 1000 and tossing the remainder.

Let's try now to prove or disprove another statement:

$$\left\lceil \sqrt{\lfloor x \rfloor} \right\rceil \stackrel{?}{=} \lceil \sqrt{x} \rceil, \qquad \text{real } x \geqslant 0.$$

This works when $x = \pi$ and $x = e$, but it fails when $x = \phi$; so we know that it isn't true in general.

Before going any further, let's digress a minute to discuss different levels of problems that might appear in books about mathematics:

Level 1. Given an explicit object x and an explicit property $P(x)$, *prove* that $P(x)$ is true. For example, "Prove that $\lfloor \pi \rfloor = 3$." Here the problem involves finding a proof of some purported fact.

Level 2. Given an explicit set X and an explicit property $P(x)$, prove that $P(x)$ is true *for all* $x \in X$. For example, "Prove that $\lfloor x \rfloor \leqslant x$ for all real x." Again the problem involves finding a proof, but the proof this time must be general. We're doing algebra, not just arithmetic.

Level 3. Given an explicit set X and an explicit property $P(x)$, prove *or disprove* that $P(x)$ is true for all $x \in X$. For example, "Prove or disprove that $\left\lceil \sqrt{\lfloor x \rfloor} \right\rceil = \lceil \sqrt{x} \rceil$ for all real $x \geqslant 0$." Here there's an additional level of uncertainty; the outcome might go either way. This is closer to the real situation a mathematician constantly faces: Assertions that get into books tend to be true, but new things have to be looked at with a jaundiced eye. If the statement is false, our job is to find a counterexample. If the statement is true, we must find a proof as in level 2.

In my other texts "prove or disprove" seems to mean the same as "prove," about 99.44% of the time; but not in this book.

Level 4. Given an explicit set X and an explicit property $P(x)$, find a *necessary and sufficient condition* $Q(x)$ that $P(x)$ is true. For example, "Find a necessary and sufficient condition that $\lfloor x \rfloor \geqslant \lceil x \rceil$." The problem is to find Q such that $P(x) \iff Q(x)$. Of course, there's always a trivial answer; we can take $Q(x) = P(x)$. But the implied requirement is to find a condition that's as simple as possible. Creativity is required to discover a simple condition that will work. (For example, in this case, "$\lfloor x \rfloor \geqslant \lceil x \rceil \iff x$ is an integer.") The extra element of discovery needed to find $Q(x)$ makes this sort of problem more difficult, but it's more typical of what mathematicians must do in the "real world." Finally, of course, a proof must be given that $P(x)$ is true if and only if $Q(x)$ is true.

But no simpler.
—A. Einstein

Level 5. Given an explicit set X, find *an interesting property* $P(x)$ of its elements. Now we're in the scary domain of pure research, where students might think that total chaos reigns. This is real mathematics. Authors of textbooks rarely dare to pose level 5 problems.

End of digression. But let's convert the last question we looked at from level 3 to level 4: What is a necessary and sufficient condition that $\lceil\sqrt{\lfloor x\rfloor}\,\rceil = \lceil\sqrt{x}\,\rceil$? We have observed that equality holds when $x = 3.142$ but not when $x = 1.618$; further experimentation shows that it fails also when x is between 9 and 10. Oho. Yes. We see that bad cases occur whenever $m^2 < x < m^2+1$, since this gives m on the left and $m+1$ on the right. In all other cases where $\sqrt{x}$ is defined, namely when $x = 0$ or $m^2+1 \leqslant x \leqslant (m+1)^2$, we get equality. The following statement is therefore necessary and sufficient for equality: Either x is an integer or $\sqrt{\lfloor x\rfloor}$ isn't.

Home of the Toledo Mudhens.

For our next problem let's consider a handy new notation, suggested by C. A. R. Hoare and Lyle Ramshaw, for intervals of the real line: $[\alpha\,..\,\beta]$ denotes the set of real numbers x such that $\alpha \leqslant x \leqslant \beta$. This set is called a *closed interval* because it contains both endpoints α and β. The interval containing neither endpoint, denoted by $(\alpha\,..\,\beta)$, consists of all x such that $\alpha < x < \beta$; this is called an *open interval*. And the intervals $[\alpha\,..\,\beta)$ and $(\alpha\,..\,\beta]$, which contain just one endpoint, are defined similarly and called *half-open*.

(Or, by pessimists, half-closed.)

How many integers are contained in such intervals? The half-open intervals are easier, so we start with them. In fact half-open intervals are almost always nicer than open or closed intervals. For example, they're additive — we can combine the half-open intervals $[\alpha\,..\,\beta)$ and $[\beta\,..\,\gamma)$ to form the half-open interval $[\alpha\,..\,\gamma)$. This wouldn't work with open intervals because the point β would be excluded, and it could cause problems with closed intervals because β would be included twice.

Back to our problem. The answer is easy if α and β are integers: Then $[\alpha\,..\,\beta)$ contains the $\beta-\alpha$ integers $\alpha, \alpha+1, \ldots, \beta-1$, assuming that $\alpha \leqslant \beta$. Similarly $(\alpha\,..\,\beta]$ contains $\beta-\alpha$ integers in such a case. But our problem is harder, because α and β are arbitrary reals. We can convert it to the easier problem, though, since

$$\begin{aligned} \alpha \leqslant n < \beta \quad &\Longleftrightarrow \quad \lceil\alpha\rceil \leqslant n < \lceil\beta\rceil\,, \\ \alpha < n \leqslant \beta \quad &\Longleftrightarrow \quad \lfloor\alpha\rfloor < n \leqslant \lfloor\beta\rfloor\,, \end{aligned}$$

when n is an integer, according to (3.7). The intervals on the right have integer endpoints and contain the same number of integers as those on the left, which have real endpoints. So the interval $[\alpha\,..\,\beta)$ contains exactly $\lceil\beta\rceil - \lceil\alpha\rceil$ integers, and $(\alpha\,..\,\beta]$ contains $\lfloor\beta\rfloor - \lfloor\alpha\rfloor$. This is a case where we actually want to introduce floor or ceiling brackets, instead of getting rid of them.

By the way, there's a mnemonic for remembering which case uses floors and which uses ceilings: Half-open intervals that include the left endpoint but not the right (such as $0 \leqslant \theta < 1$) are slightly more common than those that include the right endpoint but not the left; and floors are slightly more common than ceilings. So by Murphy's Law, the correct rule is the opposite of what we'd expect — ceilings for $[\alpha\,..\,\beta)$ and floors for $(\alpha\,..\,\beta]$.

Just like we can remember the date of Columbus's departure by singing, "In fourteen hundred and ninety-three/ Columbus sailed the deep blue sea."

Similar analyses show that the closed interval $[\alpha\,..\,\beta]$ contains exactly $\lfloor\beta\rfloor - \lceil\alpha\rceil + 1$ integers and that the open interval $(\alpha\,..\,\beta)$ contains $\lceil\beta\rceil - \lfloor\alpha\rfloor - 1$; but we place the additional restriction $\alpha \neq \beta$ on the latter so that the formula won't ever embarrass us by claiming that an empty interval $(\alpha\,..\,\alpha)$ contains a total of -1 integers. To summarize, we've deduced the following facts:

interval	integers contained	restrictions
$[\alpha\,..\,\beta]$	$\lfloor\beta\rfloor - \lceil\alpha\rceil + 1$	$\alpha \leqslant \beta\,,$
$[\alpha\,..\,\beta)$	$\lceil\beta\rceil - \lceil\alpha\rceil$	$\alpha \leqslant \beta\,,$
$(\alpha\,..\,\beta]$	$\lfloor\beta\rfloor - \lfloor\alpha\rfloor$	$\alpha \leqslant \beta\,,$
$(\alpha\,..\,\beta)$	$\lceil\beta\rceil - \lfloor\alpha\rfloor - 1$	$\alpha < \beta\,.$

(3.12)

Now here's a problem we can't refuse. The Concrete Math Club has a casino (open only to purchasers of this book) in which there's a roulette wheel with one thousand slots, numbered 1 to 1000. If the number n that comes up on a spin is divisible by the floor of its cube root, that is, if

$$\lfloor\sqrt[3]{n}\rfloor \;\backslash\; n\,,$$

then it's a winner and the house pays us \$5; otherwise it's a loser and we must pay \$1. (The notation $a\backslash b$, read "a divides b," means that b is an exact multiple of a; Chapter 4 investigates this relation carefully.) Can we expect to make money if we play this game?

(A poll of the class at this point showed that 28 students thought it was a bad idea to play, 13 wanted to gamble, and the rest were too confused to answer.)

(So we hit them with the Concrete Math Club.)

We can compute the average winnings — that is, the amount we'll win (or lose) per play — by first counting the number W of winners and the number $L = 1000 - W$ of losers. If each number comes up once during 1000 plays, we win $5W$ dollars and lose L dollars, so the average winnings will be

$$\frac{5W - L}{1000} = \frac{5W - (1000 - W)}{1000} = \frac{6W - 1000}{1000}\,.$$

If there are 167 or more winners, we have the advantage; otherwise the advantage is with the house.

How can we count the number of winners among 1 through 1000? It's not hard to spot a pattern. The numbers from 1 through $2^3 - 1 = 7$ are all winners because $\lfloor\sqrt[3]{n}\rfloor = 1$ for each. Among the numbers $2^3 = 8$ through $3^3 - 1 = 26$, only the even numbers are winners. And among $3^3 = 27$ through $4^3 - 1 = 63$, only those divisible by 3 are. And so on.

The whole setup can be analyzed systematically if we use the summation techniques of Chapter 2, taking advantage of Iverson's convention about logical statements evaluating to 0 or 1:

$$\begin{aligned}
W &= \sum_{n=1}^{1000} [n \text{ is a winner}] \\
&= \sum_{1\leqslant n\leqslant 1000} \left[\lfloor\sqrt[3]{n}\rfloor \setminus n\right] \;=\; \sum_{k,n} \left[k=\lfloor\sqrt[3]{n}\rfloor\right][k\backslash n][1\leqslant n\leqslant 1000] \\
&= \sum_{k,m,n} \left[k^3\leqslant n<(k+1)^3\right][n=km][1\leqslant n\leqslant 1000] \\
&= 1+\sum_{k,m} \left[k^3\leqslant km<(k+1)^3\right][1\leqslant k<10] \\
&= 1+\sum_{k,m} \left[m\in\left[k^2\,..\,(k+1)^3/k\right)\right][1\leqslant k<10] \\
&= 1+\sum_{1\leqslant k<10} \left(\lceil k^2+3k+3+1/k\rceil-\lceil k^2\rceil\right) \\
&= 1+\sum_{1\leqslant k<10} (3k+4) \;=\; 1+\frac{7+31}{2}\cdot 9 \;=\; 172\,.
\end{aligned}$$

This derivation merits careful study. Notice that line 6 uses our formula (3.12) for the number of integers in a half-open interval. The only "difficult" maneuver is the decision made between lines 3 and 4 to treat $n = 1000$ as a special case. (The inequality $k^3 \leqslant n < (k+1)^3$ does not combine easily with $1 \leqslant n \leqslant 1000$ when $k = 10$.) In general, boundary conditions tend to be the most critical part of $\sum$-manipulations.

True.

The bottom line says that $W = 172$; hence our formula for average winnings per play reduces to $(6\cdot 172-1000)/1000$ dollars, which is 3.2 cents. We can expect to be about \$3.20 richer after making 100 bets of \$1 each. (Of course, the house may have made some numbers more equal than others.)

Where did you say this casino is?

The casino problem we just solved is a dressed-up version of the more mundane question, "How many integers n, where $1 \leqslant n \leqslant 1000$, satisfy the relation $\lfloor\sqrt[3]{n}\rfloor \setminus n$?" Mathematically the two questions are the same. But sometimes it's a good idea to dress up a problem. We get to use more vocabulary (like "winners" and "losers"), which helps us to understand what's going on.

Let's get general. Suppose we change 1000 to 1000000, or to an even larger number, N. (We assume that the casino has connections and can get a bigger wheel.) Now how many winners are there?

The same argument applies, but we need to deal more carefully with the largest value of k, which we can call K for convenience:

$$K \;=\; \lfloor\sqrt[3]{N}\rfloor\,.$$

(Previously K was 10.) The total number of winners for general N comes to

$$\begin{aligned} W &= \sum_{1\leqslant k<K}(3k+4)+\sum_m\left[K^3\leqslant Km\leqslant N\right] \\ &= \tfrac{1}{2}(7+3K+1)(K-1)+\sum_m\left[m\in[K^2\,..\,N/K]\right] \\ &= \tfrac{3}{2}K^2+\tfrac{5}{2}K-4+\sum_m\left[m\in[K^2\,..\,N/K]\right]. \end{aligned}$$

We know that the remaining sum is $\lfloor N/K\rfloor-\lceil K^2\rceil+1=\lfloor N/K\rfloor-K^2+1$; hence the formula

$$W = \lfloor N/K\rfloor+\tfrac{1}{2}K^2+\tfrac{5}{2}K-3\,,\qquad K=\lfloor\sqrt[3]{N}\rfloor \tag{3.13}$$

gives the general answer for a wheel of size N.

The first two terms of this formula are approximately $N^{2/3}+\frac{1}{2}N^{2/3}=\frac{3}{2}N^{2/3}$, and the other terms are much smaller in comparison, when N is large. In Chapter 9 we'll learn how to derive expressions like

$$W = \tfrac{3}{2}N^{2/3}+O(N^{1/3})\,,$$

where $O(N^{1/3})$ stands for a quantity that is no more than a constant times $N^{1/3}$. Whatever the constant is, we know that it's independent of N; so for large N the contribution of the O-term to W will be quite small compared with $\frac{3}{2}N^{2/3}$. For example, the following table shows how close $\frac{3}{2}N^{2/3}$ is to W:

N	$\frac{3}{2}N^{2/3}$	W	% error
1,000	150.0	172	12.791
10,000	696.2	746	6.670
100,000	3231.7	3343	3.331
1,000,000	15000.0	15247	1.620
10,000,000	69623.8	70158	0.761
100,000,000	323165.2	324322	0.357
1,000,000,000	1500000.0	1502496	0.166

It's a pretty good approximation.

Approximate formulas are useful because they're simpler than formulas with floors and ceilings. However, the exact truth is often important, too, especially for the smaller values of N that tend to occur in practice. For example, the casino owner may have falsely assumed that there are only $\frac{3}{2}N^{2/3}=150$ winners when $N=1000$ (in which case there would be a 10¢ advantage for the house).

Our last application in this section looks at so-called spectra. We define the *spectrum* of a real number α to be an infinite multiset of integers,

$$\mathrm{Spec}(\alpha) = \{\lfloor\alpha\rfloor, \lfloor 2\alpha\rfloor, \lfloor 3\alpha\rfloor, \ldots\}.$$

(A multiset is like a set but it can have repeated elements.) For example, the spectrum of $1/2$ starts out $\{0, 1, 1, 2, 2, 3, 3, \ldots\}$.

... without lots of generality...

It's easy to prove that no two spectra are equal—that $\alpha \neq \beta$ implies $\mathrm{Spec}(\alpha) \neq \mathrm{Spec}(\beta)$. For, assuming without loss of generality that $\alpha < \beta$, there's a positive integer m such that $m(\beta - \alpha) \geqslant 1$. (In fact, any $m \geqslant \lceil 1/(\beta - \alpha)\rceil$ will do; but we needn't show off our knowledge of floors and ceilings all the time.) Hence $m\beta - m\alpha \geqslant 1$, and $\lfloor m\beta\rfloor > \lfloor m\alpha\rfloor$. Thus $\mathrm{Spec}(\beta)$ has fewer than m elements $\leqslant \lfloor m\alpha\rfloor$, while $\mathrm{Spec}(\alpha)$ has at least m.

"If x be an incommensurable number less than unity, one of the series of quantities m/x, $m/(1-x)$, where m is a whole number, can be found which shall lie between any given consecutive integers, and but one such quantity can be found."
—Rayleigh [304]

Spectra have many beautiful properties. For example, consider the two multisets

$$\begin{aligned}\mathrm{Spec}(\sqrt{2}) &= \{1, 2, 4, 5, 7, 8, 9, 11, 12, 14, 15, 16, 18, 19, 21, 22, 24, \ldots\},\\ \mathrm{Spec}(2+\sqrt{2}) &= \{3, 6, 10, 13, 17, 20, 23, 27, 30, 34, 37, 40, 44, 47, 51, \ldots\}.\end{aligned}$$

It's easy to calculate $\mathrm{Spec}(\sqrt{2})$ with a pocket calculator, and the nth element of $\mathrm{Spec}(2+\sqrt{2})$ is just $2n$ more than the nth element of $\mathrm{Spec}(\sqrt{2})$, by (3.6). A closer look shows that these two spectra are also related in a much more surprising way: It seems that any number missing from one is in the other, but that no number is in both! And it's true: The positive integers are the disjoint union of $\mathrm{Spec}(\sqrt{2})$ and $\mathrm{Spec}(2+\sqrt{2})$. We say that these spectra form a *partition* of the positive integers.

To prove this assertion, we will count how many of the elements of $\mathrm{Spec}(\sqrt{2})$ are $\leqslant n$, and how many of the elements of $\mathrm{Spec}(2+\sqrt{2})$ are $\leqslant n$. If the total is n, for each n, these two spectra do indeed partition the integers.

Right, because exactly one of the counts must increase when n increases by 1.

Let α be positive. The number of elements in $\mathrm{Spec}(\alpha)$ that are $\leqslant n$ is

$$\begin{aligned}N(\alpha, n) &= \sum_{k>0} \big[\lfloor k\alpha\rfloor \leqslant n\big]\\ &= \sum_{k>0} \big[\lfloor k\alpha\rfloor < n+1\big]\\ &= \sum_{k>0} [k\alpha < n+1]\\ &= \sum_{k} \big[0 < k < (n+1)/\alpha\big]\\ &= \lceil (n+1)/\alpha\rceil - 1\,. \qquad (3.14)\end{aligned}$$

This derivation has two special points of interest. First, it uses the law

$$m \leqslant n \quad \Longleftrightarrow \quad m < n+1\,, \qquad \text{integers } m \text{ and } n \tag{3.15}$$

to change '$\leqslant$' to '$<$', so that the floor brackets can be removed by (3.7). Also — and this is more subtle — it sums over the range $k > 0$ instead of $k \geqslant 1$, because $(n+1)/\alpha$ might be less than 1 for certain n and α. If we had tried to apply (3.12) to determine the number of integers in $[1\,..\,(n+1)/\alpha)$, rather than the number of integers in $(0\,..\,(n+1)/\alpha)$, we would have gotten the right answer; but our derivation would have been faulty because the conditions of applicability wouldn't have been met.

Good, we have a formula for $N(\alpha, n)$. Now we can test whether or not $\mathrm{Spec}(\sqrt{2}\,)$ and $\mathrm{Spec}(2+\sqrt{2}\,)$ partition the positive integers, by testing whether or not $N(\sqrt{2}, n) + N(2+\sqrt{2}, n) = n$ for all integers $n > 0$, using (3.14):

$$\begin{aligned}
&\left\lceil \frac{n+1}{\sqrt{2}} \right\rceil - 1 + \left\lceil \frac{n+1}{2+\sqrt{2}} \right\rceil - 1 \;=\; n \\
\Longleftrightarrow\quad &\left\lfloor \frac{n+1}{\sqrt{2}} \right\rfloor + \left\lfloor \frac{n+1}{2+\sqrt{2}} \right\rfloor \;=\; n\,, && \text{by (3.2);} \\
\Longleftrightarrow\quad &\frac{n+1}{\sqrt{2}} - \left\{ \frac{n+1}{\sqrt{2}} \right\} + \frac{n+1}{2+\sqrt{2}} - \left\{ \frac{n+1}{2+\sqrt{2}} \right\} \;=\; n\,, && \text{by (3.8).}
\end{aligned}$$

Everything simplifies now because of the neat identity

$$\frac{1}{\sqrt{2}} + \frac{1}{2+\sqrt{2}} \;=\; 1\,;$$

our condition reduces to testing whether or not

$$\left\{ \frac{n+1}{\sqrt{2}} \right\} + \left\{ \frac{n+1}{2+\sqrt{2}} \right\} \;=\; 1\,,$$

for all $n > 0$. And we win, because these are the fractional parts of two noninteger numbers that add up to the integer $n+1$. A partition it is.

3.3 FLOOR/CEILING RECURRENCES

Floors and ceilings add an interesting new dimension to the study of recurrence relations. Let's look first at the recurrence

$$\begin{aligned}
K_0 &= 1\,; \\
K_{n+1} &= 1 + \min(2K_{\lfloor n/2 \rfloor}, 3K_{\lfloor n/3 \rfloor})\,, \qquad \text{for } n \geqslant 0.
\end{aligned} \tag{3.16}$$

Thus, for example, K_1 is $1 + \min(2K_0, 3K_0) = 3$; the sequence begins 1, 3, 3, 4, 7, 7, 7, 9, 9, 10, 13, ... One of the authors of this book has modestly decided to call these the Knuth numbers.

Exercise 25 asks for a proof or disproof that $K_n \geqslant n$, for all $n \geqslant 0$. The first few K's just listed do satisfy the inequality, so there's a good chance that it's true in general. Let's try an induction proof: The basis $n = 0$ comes directly from the defining recurrence. For the induction step, we assume that the inequality holds for all values up through some fixed nonnegative n, and we try to show that $K_{n+1} \geqslant n + 1$. From the recurrence we know that $K_{n+1} = 1 + \min(2K_{\lfloor n/2 \rfloor}, 3K_{\lfloor n/3 \rfloor})$. The induction hypothesis tells us that $2K_{\lfloor n/2 \rfloor} \geqslant 2\lfloor n/2 \rfloor$ and $3K_{\lfloor n/3 \rfloor} \geqslant 3\lfloor n/3 \rfloor$. However, $2\lfloor n/2 \rfloor$ can be as small as $n - 1$, and $3\lfloor n/3 \rfloor$ can be as small as $n - 2$. The most we can conclude from our induction hypothesis is that $K_{n+1} \geqslant 1 + (n - 2)$; this falls far short of $K_{n+1} \geqslant n + 1$.

We now have reason to worry about the truth of $K_n \geqslant n$, so let's try to disprove it. If we can find an n such that either $2K_{\lfloor n/2 \rfloor} < n$ or $3K_{\lfloor n/3 \rfloor} < n$, or in other words such that

$$K_{\lfloor n/2 \rfloor} < n/2 \qquad \text{or} \qquad K_{\lfloor n/3 \rfloor} < n/3\,,$$

we will have $K_{n+1} < n + 1$. Can this be possible? We'd better not give the answer away here, because that will spoil exercise 25.

Recurrence relations involving floors and/or ceilings arise often in computer science, because algorithms based on the important technique of "divide and conquer" often reduce a problem of size n to the solution of similar problems of integer sizes that are fractions of n. For example, one way to sort n records, if $n > 1$, is to divide them into two approximately equal parts, one of size $\lceil n/2 \rceil$ and the other of size $\lfloor n/2 \rfloor$. (Notice, incidentally, that

$$n = \lceil n/2 \rceil + \lfloor n/2 \rfloor\,; \tag{3.17}$$

this formula comes in handy rather often.) After each part has been sorted separately (by the same method, applied recursively), we can merge the records into their final order by doing at most $n - 1$ further comparisons. Therefore the total number of comparisons performed is at most $f(n)$, where

$$\begin{aligned} &f(1) = 0\,; \\ &f(n) = f(\lceil n/2 \rceil) + f(\lfloor n/2 \rfloor) + n - 1\,, \qquad \text{for } n > 1. \end{aligned} \tag{3.18}$$

A solution to this recurrence appears in exercise 34.

The Josephus problem of Chapter 1 has a similar recurrence, which can be cast in the form

$$\begin{aligned} J(1) &= 1\,; \\ J(n) &= 2J(\lfloor n/2 \rfloor) - (-1)^n\,, \qquad \text{for } n > 1. \end{aligned}$$

We've got more tools to work with than we had in Chapter 1, so let's consider the more authentic Josephus problem in which every third person is eliminated, instead of every second. If we apply the methods that worked in Chapter 1 to this more difficult problem, we wind up with a recurrence like

$$J_3(n) \;=\; \left\lceil \tfrac{3}{2} J_3\big(\lfloor \tfrac{2}{3} n \rfloor\big) + a_n \right\rceil \bmod n+1\,,$$

where 'mod' is a function that we will be studying shortly, and where we have $a_n = -2$, $+1$, or $-\frac{1}{2}$ according as $n \bmod 3 = 0$, 1, or 2. But this recurrence is too horrible to pursue.

There's another approach to the Josephus problem that gives a much better setup. Whenever a person is passed over, we can assign a new number. Thus, 1 and 2 become $n+1$ and $n+2$, then 3 is executed; 4 and 5 become $n+3$ and $n+4$, then 6 is executed; ...; $3k+1$ and $3k+2$ become $n+2k+1$ and $n+2k+2$, then $3k+3$ is executed; ... then $3n$ is executed (or left to survive). For example, when $n = 10$ the numbers are

1	2	3	4	5	6	7	8	9	10
11	12		13	14		15	16		17
18			19	20			21		22
			23	24					25
			26						27
			28						
			29						
			30						

The kth person eliminated ends up with number $3k$. So we can figure out who the survivor is if we can figure out the original number of person number $3n$.

If $N > n$, person number N must have had a previous number, and we can find it as follows: We have $N = n + 2k + 1$ or $N = n + 2k + 2$, hence $k = \lfloor (N - n - 1)/2 \rfloor$; the previous number was $3k+1$ or $3k+2$, respectively. That is, it was $3k + (N - n - 2k) = k + N - n$. Hence we can calculate the survivor's number $J_3(n)$ as follows:

$$\begin{aligned}
&N \;:=\; 3n\,;\\
&\textbf{while}\;\; N > n \;\;\textbf{do}\;\; N := \left\lfloor \frac{N-n-1}{2} \right\rfloor + N - n\,;\\
&J_3(n) \;:=\; N\,.
\end{aligned}$$

This is not a closed form for $J_3(n)$; it's not even a recurrence. But at least it tells us how to calculate the answer reasonably fast, if n is large.

"Not too slow, not too fast."
—L. Armstrong

Fortunately there's a way to simplify this algorithm if we use the variable $D = 3n + 1 - N$ in place of N. (This change in notation corresponds to assigning numbers from $3n$ down to 1, instead of from 1 up to $3n$; it's sort of like a countdown.) Then the complicated assignment to N becomes

$$\begin{aligned} D &:= 3n+1-\left(\left\lfloor\frac{(3n+1-D)-n-1}{2}\right\rfloor+(3n+1-D)-n\right) \\ &= n+D-\left\lfloor\frac{2n-D}{2}\right\rfloor = D-\left\lfloor\frac{-D}{2}\right\rfloor = D+\left\lceil\frac{D}{2}\right\rceil = \left\lceil\tfrac{3}{2}D\right\rceil, \end{aligned}$$

and we can rewrite the algorithm as follows:

$$\begin{aligned} &D := 1; \\ &\textbf{while } D \leqslant 2n \textbf{ do } D := \left\lceil\tfrac{3}{2}D\right\rceil; \\ &J_3(n) := 3n+1-D. \end{aligned}$$

Aha! This looks much nicer, because n enters the calculation in a very simple way. In fact, we can show by the same reasoning that the survivor $J_q(n)$ when every qth person is eliminated can be calculated as follows:

$$\begin{aligned} &D := 1; \\ &\textbf{while } D \leqslant (q-1)n \textbf{ do } D := \left\lceil\tfrac{q}{q-1}D\right\rceil; \\ &J_q(n) := qn+1-D. \end{aligned} \tag{3.19}$$

In the case $q = 2$ that we know so well, this makes D grow to 2^{m+1} when $n = 2^m + l$; hence $J_2(n) = 2(2^m + l) + 1 - 2^{m+1} = 2l + 1$. Good.

The recipe in (3.19) computes a sequence of integers that can be defined by the following recurrence:

$$\begin{aligned} &D_0^{(q)} = 1; \\ &D_n^{(q)} = \left\lceil\frac{q}{q-1}D_{n-1}^{(q)}\right\rceil \qquad \text{for } n > 0. \end{aligned} \tag{3.20}$$

These numbers don't seem to relate to any familiar functions in a simple way, except when $q = 2$; hence they probably don't have a nice closed form. But if we're willing to accept the sequence $D_n^{(q)}$ as "known," then it's easy to describe the solution to the generalized Josephus problem: The survivor $J_q(n)$ is $qn + 1 - D_k^{(q)}$, where k is as small as possible such that $D_k^{(q)} > (q-1)n$.

"Known" like, say, harmonic numbers. A. M. Odlyzko and H. S. Wilf have shown [283] that $D_n^{(3)} = \lfloor(\frac{3}{2})^n C\rfloor$, *where* $C \approx 1.622270503$.

3.4 'MOD': THE BINARY OPERATION

The quotient of n divided by m is $\lfloor n/m\rfloor$, when m and n are positive integers. It's handy to have a simple notation also for the remainder of this

division, and we call it '$n \bmod m$'. The basic formula

$$n = m\underbrace{\lfloor n/m \rfloor}_{\text{quotient}} + \underbrace{n \bmod m}_{\text{remainder}}$$

tells us that we can express $n \bmod m$ as $n - m\lfloor n/m \rfloor$. We can generalize this to negative integers, and in fact to arbitrary real numbers:

$$x \bmod y = x - y\lfloor x/y \rfloor, \qquad \text{for } y \neq 0. \tag{3.21}$$

This defines 'mod' as a binary operation, just as addition and subtraction are binary operations. Mathematicians have used mod this way informally for a long time, taking various quantities mod 10, mod 2π, and so on, but only in the last twenty years has it caught on formally. Old notion, new notation.

Why do they call it 'mod': The Binary Operation? Stay tuned to find out in the next, exciting, chapter!

We can easily grasp the intuitive meaning of $x \bmod y$, when x and y are positive real numbers, if we imagine a circle of circumference y whose points have been assigned real numbers in the interval $[0\,..\,y)$. If we travel a distance x around the circle, starting at 0, we end up at $x \bmod y$. (And the number of times we encounter 0 as we go is $\lfloor x/y \rfloor$.)

When x or y is negative, we need to look at the definition carefully in order to see exactly what it means. Here are some integer-valued examples:

Beware of computer languages that use another definition.

$$\begin{aligned}
5 \bmod 3 &= 5 - 3\lfloor 5/3 \rfloor &&= 2\,;\\
5 \bmod -3 &= 5 - (-3)\lfloor 5/(-3) \rfloor &&= -1\,;\\
-5 \bmod 3 &= -5 - 3\lfloor -5/3 \rfloor &&= 1\,;\\
-5 \bmod -3 &= -5 - (-3)\lfloor -5/(-3) \rfloor &&= -2\,.
\end{aligned}$$

The number after 'mod' is called the *modulus*; nobody has yet decided what to call the number before 'mod'. In applications, the modulus is usually positive, but the definition makes perfect sense when the modulus is negative. In both cases the value of $x \bmod y$ is between 0 and the modulus:

How about calling the other number the modumor?

$$\begin{aligned}
0 \leqslant x \bmod y &< y\,, \qquad \text{for } y > 0;\\
0 \geqslant x \bmod y &> y\,, \qquad \text{for } y < 0.
\end{aligned}$$

What about $y = 0$? Definition (3.21) leaves this case undefined, in order to avoid division by zero, but to be complete we can define

$$x \bmod 0 = x\,. \tag{3.22}$$

This convention preserves the property that $x \bmod y$ always differs from x by a multiple of y. (It might seem more natural to make the function continuous at 0, by defining $x \bmod 0 = \lim_{y\to 0} x \bmod y = 0$. But we'll see in Chapter 4

that this would be much less useful. Continuity is not an important aspect of the mod operation.)

We've already seen one special case of mod in disguise, when we wrote x in terms of its integer and fractional parts, $x = \lfloor x \rfloor + \{x\}$. The fractional part can also be written $x \bmod 1$, because we have

$$x = \lfloor x \rfloor + x \bmod 1 .$$

Notice that parentheses aren't needed in this formula; we take mod to bind more tightly than addition or subtraction.

The floor function has been used to define mod, and the ceiling function hasn't gotten equal time. We could perhaps use the ceiling to define a mod analog like

$$x \text{ mumble } y = y\lceil x/y \rceil - x ;$$

There was a time in the 70s when 'mod' was the fashion. Maybe the new mumble function should be called 'punk'?

No — I like 'mumble'.

in our circle analogy this represents the distance the traveler needs to continue, after going a distance x, to get back to the starting point 0. But of course we'd need a better name than 'mumble'. If sufficient applications come along, an appropriate name will probably suggest itself.

The distributive law is mod's most important algebraic property: We have

$$c(x \bmod y) = (cx) \bmod (cy) \tag{3.23}$$

for all real c, x, and y. (Those who like mod to bind less tightly than multiplication may remove the parentheses from the right side here, too.) It's easy to prove this law from definition (3.21), since

$$c(x \bmod y) = c(x - y\lfloor x/y \rfloor) = cx - cy\lfloor cx/cy \rfloor = cx \bmod cy ,$$

if $cy \neq 0$; and the zero-modulus cases are trivially true. Our four examples using ± 5 and ± 3 illustrate this law twice, with $c = -1$. An identity like (3.23) is reassuring, because it gives us reason to believe that 'mod' has not been defined improperly.

The remainder, eh?

In the remainder of this section, we'll consider an application in which 'mod' turns out to be helpful although it doesn't play a central role. The problem arises frequently in a variety of situations: We want to partition n things into m groups as equally as possible.

Suppose, for example, that we have n short lines of text that we'd like to arrange in m columns. For æsthetic reasons, we want the columns to be arranged in decreasing order of length (actually nonincreasing order); and the lengths should be approximately the same — no two columns should differ by

more than one line's worth of text. If 37 lines of text are being divided into five columns, we would therefore prefer the arrangement on the right:

8	8	8	8	5
line 1	line 9	line 17	line 25	line 33
line 2	line 10	line 18	line 26	line 34
line 3	line 11	line 19	line 27	line 35
line 4	line 12	line 20	line 28	line 36
line 5	line 13	line 21	line 29	line 37
line 6	line 14	line 22	line 30	
line 7	line 15	line 23	line 31	
line 8	line 16	line 24	line 32	

8	8	7	7	7
line 1	line 9	line 17	line 24	line 31
line 2	line 10	line 18	line 25	line 32
line 3	line 11	line 19	line 26	line 33
line 4	line 12	line 20	line 27	line 34
line 5	line 13	line 21	line 28	line 35
line 6	line 14	line 22	line 29	line 36
line 7	line 15	line 23	line 30	line 37
line 8	line 16			

Furthermore we want to distribute the lines of text columnwise — first deciding how many lines go into the first column and then moving on to the second, the third, and so on — because that's the way people read. Distributing row by row would give us the correct number of lines in each column, but the ordering would be wrong. (We would get something like the arrangement on the right, but column 1 would contain lines 1, 6, 11, ..., 36, instead of lines 1, 2, 3, ..., 8 as desired.)

A row-by-row distribution strategy can't be used, but it does tell us how many lines to put in each column. If n is not a multiple of m, the row-by-row procedure makes it clear that the long columns should each contain $\lceil n/m \rceil$ lines, and the short columns should each contain $\lfloor n/m \rfloor$. There will be exactly $n \bmod m$ long columns (and, as it turns out, there will be exactly n mumble m short ones).

Let's generalize the terminology and talk about 'things' and 'groups' instead of 'lines' and 'columns'. We have just decided that the first group should contain $\lceil n/m \rceil$ things; therefore the following sequential distribution scheme ought to work: To distribute n things into m groups, when $m > 0$, put $\lceil n/m \rceil$ things into one group, then use the same procedure recursively to put the remaining $n' = n - \lceil n/m \rceil$ things into $m' = m - 1$ additional groups.

For example, if $n = 314$ and $m = 6$, the distribution goes like this:

remaining things	remaining groups	$\lceil$things/groups$\rceil$
314	6	53
261	5	53
208	4	52
156	3	52
104	2	52
52	1	52

It works. We get groups of approximately the same size, even though the divisor keeps changing.

Why does it work? In general we can suppose that $n = qm + r$, where $q = \lfloor n/m \rfloor$ and $r = n \bmod m$. The process is simple if $r = 0$: We put $\lceil n/m \rceil = q$ things into the first group and replace n by $n' = n - q$, leaving

$n' = qm'$ things to put into the remaining $m' = m - 1$ groups. And if $r > 0$, we put $\lceil n/m \rceil = q + 1$ things into the first group and replace n by $n' = n - q - 1$, leaving $n' = qm' + r - 1$ things for subsequent groups. The new remainder is $r' = r - 1$, but q stays the same. It follows that there will be r groups with $q + 1$ things, followed by $m - r$ groups with q things.

How many things are in the kth group? We'd like a formula that gives $\lceil n/m \rceil$ when $k \leqslant n \bmod m$, and $\lfloor n/m \rfloor$ otherwise. It's not hard to verify that

$$\left\lceil \frac{n-k+1}{m} \right\rceil$$

has the desired properties, because this reduces to $q + \lceil (r-k+1)/m \rceil$ if we write $n = qm + r$ as in the preceding paragraph; here $q = \lfloor n/m \rfloor$. We have $\lceil (r-k+1)/m \rceil = [k \leqslant r]$, if $1 \leqslant k \leqslant m$ and $0 \leqslant r < m$. Therefore we can write an identity that expresses the partition of n into m as-equal-as-possible parts in nonincreasing order:

$$n = \left\lceil \frac{n}{m} \right\rceil + \left\lceil \frac{n-1}{m} \right\rceil + \cdots + \left\lceil \frac{n-m+1}{m} \right\rceil . \tag{3.24}$$

This identity is valid for all positive integers m, and for all integers n (whether positive, negative, or zero). We have already encountered the case $m = 2$ in (3.17), although we wrote it in a slightly different form, $n = \lceil n/2 \rceil + \lfloor n/2 \rfloor$.

If we had wanted the parts to be in nondecreasing order, with the small groups coming before the larger ones, we could have proceeded in the same way but with $\lfloor n/m \rfloor$ things in the first group. Then we would have derived the corresponding identity

$$n = \left\lfloor \frac{n}{m} \right\rfloor + \left\lfloor \frac{n+1}{m} \right\rfloor + \cdots + \left\lfloor \frac{n+m-1}{m} \right\rfloor . \tag{3.25}$$

It's possible to convert between (3.25) and (3.24) by using either (3.4) or the identity of exercise 12.

Some claim that it's too dangerous to replace anything by an mx.

Now if we replace n in (3.25) by $\lfloor mx \rfloor$, and apply rule (3.11) to remove floors inside of floors, we get an identity that holds for all real x:

$$\lfloor mx \rfloor = \lfloor x \rfloor + \left\lfloor x + \frac{1}{m} \right\rfloor + \cdots + \left\lfloor x + \frac{m-1}{m} \right\rfloor . \tag{3.26}$$

This is rather amazing, because the floor function is an integer approximation of a real value, but the single approximation on the left equals the sum of a bunch of them on the right. If we assume that $\lfloor x \rfloor$ is roughly $x - \frac{1}{2}$ on the average, the left-hand side is roughly $mx - \frac{1}{2}$, while the right-hand side comes to roughly $(x - \frac{1}{2}) + (x - \frac{1}{2} + \frac{1}{m}) + \cdots + (x - \frac{1}{2} + \frac{m-1}{m}) = mx - \frac{1}{2}$; the sum of all these rough approximations turns out to be exact!

3.5 FLOOR/CEILING SUMS

Equation (3.26) demonstrates that it's possible to get a closed form for at least one kind of sum that involves $\lfloor\ \rfloor$. Are there others? Yes. The trick that usually works in such cases is to get rid of the floor or ceiling by introducing a new variable.

For example, let's see if it's possible to do the sum

$$\sum_{0\leqslant k<n} \lfloor\sqrt{k}\rfloor$$

in closed form. One idea is to introduce the variable $m = \lfloor\sqrt{k}\rfloor$; we can do this "mechanically" by proceeding as we did in the roulette problem:

$$\begin{aligned}
\sum_{0\leqslant k<n} \lfloor\sqrt{k}\rfloor &= \sum_{k,m\geqslant 0} m[k<n]\big[m=\lfloor\sqrt{k}\rfloor\big] \\
&= \sum_{k,m\geqslant 0} m[k<n]\big[m\leqslant\sqrt{k}<m+1\big] \\
&= \sum_{k,m\geqslant 0} m[k<n]\big[m^2\leqslant k<(m+1)^2\big] \\
&= \sum_{k,m\geqslant 0} m\big[m^2\leqslant k<(m+1)^2\leqslant n\big] \\
&\qquad\qquad + \sum_{k,m\geqslant 0} m\big[m^2\leqslant k<n<(m+1)^2\big]\,.
\end{aligned}$$

Once again the boundary conditions are a bit delicate. Let's assume first that $n = a^2$ is a perfect square. Then the second sum is zero, and the first can be evaluated by our usual routine:

$$\begin{aligned}
&\sum_{k,m\geqslant 0} m\big[m^2\leqslant k<(m+1)^2\leqslant a^2\big] \\
&\quad= \sum_{m\geqslant 0} m\big((m+1)^2-m^2\big)[m+1\leqslant a] \\
&\quad= \sum_{m\geqslant 0} m(2m+1)[m<a] \\
&\quad= \sum_{m\geqslant 0} (2m^{\underline{2}}+3m^{\underline{1}})[m<a] \\
&\quad= \sum_0^a (2m^{\underline{2}}+3m^{\underline{1}})\,\delta m \\
&\quad= \tfrac{2}{3}a(a-1)(a-2)+\tfrac{3}{2}a(a-1) \;=\; \tfrac{1}{6}(4a+1)a(a-1)\,.
\end{aligned}$$

Falling powers make the sum come tumbling down.

In the general case we can let $a = \lfloor\sqrt{n}\rfloor$; then we merely need to add the terms for $a^2 \leqslant k < n$, which are all equal to a, so they sum to $(n - a^2)a$. This gives the desired closed form,

$$\sum_{0\leqslant k<n} \lfloor\sqrt{k}\rfloor \;=\; na - \tfrac{1}{3}a^3 - \tfrac{1}{2}a^2 - \tfrac{1}{6}a\,, \qquad a = \lfloor\sqrt{n}\rfloor. \tag{3.27}$$

Another approach to such sums is to replace an expression of the form $\lfloor x\rfloor$ by $\sum_j [1 \leqslant j \leqslant x]$; this is legal whenever $x \geqslant 0$. Here's how that method works in the sum of $\lfloor$square roots$\rfloor$, if we assume for convenience that $n = a^2$:

$$\begin{aligned}\sum_{0\leqslant k<n} \lfloor\sqrt{k}\rfloor \;&=\; \sum_{j,k} [1 \leqslant j \leqslant \sqrt{k}\,][0 \leqslant k < a^2] \\ &=\; \sum_{1\leqslant j<a} \sum_{k} [j^2 \leqslant k < a^2] \\ &=\; \sum_{1\leqslant j<a} (a^2 - j^2) \;=\; a^3 - \tfrac{1}{3}a(a+\tfrac{1}{2})(a+1)\,.\end{aligned}$$

Now here's another example where a change of variable leads to a transformed sum. A remarkable theorem was discovered independently by three mathematicians — Bohl [34], Sierpiński [326], and Weyl [368] — at about the same time in 1909: If α is irrational then the fractional parts $\{n\alpha\}$ are very uniformly distributed between 0 and 1, as $n \to \infty$. One way to state this is that

$$\lim_{n\to\infty} \frac{1}{n} \sum_{0\leqslant k<n} f\big(\{k\alpha\}\big) \;=\; \int_0^1 f(x)\,dx \tag{3.28}$$

for all irrational α and all functions f that are continuous almost everywhere. For example, the *average* value of $\{n\alpha\}$ can be found by setting $f(x) = x$; we get $\frac{1}{2}$. (That's exactly what we might expect; but it's nice to know that it is really, provably true, no matter how irrational α is.)

Warning: This stuff is fairly advanced. Better skim the next two pages on first reading; they aren't crucial.
— Friendly TA

Start Skimming

The theorem of Bohl, Sierpiński, and Weyl is proved by approximating $f(x)$ above and below by "step functions," which are linear combinations of the simple functions

$$f_\nu(x) \;=\; [0 \leqslant x < \nu]$$

when $0 \leqslant \nu \leqslant 1$. Our purpose here is not to prove the theorem; that's a job for calculus books. But let's try to figure out the basic reason why it holds, by seeing how well it works in the special case $f(x) = f_\nu(x)$. In other words, let's try to see how close the sum

$$\sum_{0\leqslant k<n} \big[\{k\alpha\} < \nu\big]$$

gets to the "ideal" value $n\nu$, when n is large and α is irrational.

For this purpose we define the *discrepancy* $D(\alpha, n)$ to be the maximum absolute value, over all $0 \leqslant \nu \leqslant 1$, of the sum

$$s(\alpha, n, \nu) \;=\; \sum_{0 \leqslant k < n} \Big(\big[\{k\alpha\} < \nu\big] - \nu \Big) . \tag{3.29}$$

Our goal is to show that $D(\alpha, n)$ is "not too large" when compared with n, by showing that $|s(\alpha, n, \nu)|$ is always reasonably small when α is irrational.

First we can rewrite $s(\alpha, n, \nu)$ in simpler form, then introduce a new index variable j:

$$\begin{aligned}
\sum_{0 \leqslant k < n} \Big(\big[\{k\alpha\} < \nu\big] - \nu \Big) \;&=\; \sum_{0 \leqslant k < n} \big(\lfloor k\alpha \rfloor - \lfloor k\alpha - \nu \rfloor - \nu \big) \\
&=\; -n\nu + \sum_{0 \leqslant k < n} \sum_{j} \big[k\alpha - \nu < j \leqslant k\alpha \big] \\
&=\; -n\nu + \sum_{0 \leqslant j < \lceil n\alpha \rceil} \sum_{k < n} \big[j\alpha^{-1} \leqslant k < (j+\nu)\alpha^{-1} \big] .
\end{aligned}$$

If we're lucky, we can do the sum on k. But we ought to introduce some new variables, so that the formula won't be such a mess. Without loss of generality, we can assume that $0 < \alpha < 1$; let us write

Right, name and conquer. The change of variable from k to j is the main point.
—Friendly TA

$$\begin{aligned}
a \;&=\; \lfloor \alpha^{-1} \rfloor , & \alpha^{-1} \;&=\; a + \alpha' ; \\
b \;&=\; \lceil \nu\alpha^{-1} \rceil , & \nu\alpha^{-1} \;&=\; b - \nu' .
\end{aligned}$$

Thus $\alpha' = \{\alpha^{-1}\}$ is the fractional part of α^{-1}, and ν' is the mumble-fractional part of $\nu\alpha^{-1}$.

Once again the boundary conditions are our only source of grief. For now, let's forget the restriction '$k < n$' and evaluate the sum on k without it:

$$\begin{aligned}
\sum_{k} \Big[k \in \big[j\alpha^{-1} \,..\, (j+\nu)\alpha^{-1} \big) \Big] \;&=\; \big\lceil (j+\nu)(a+\alpha') \big\rceil - \big\lceil j(a+\alpha') \big\rceil \\
&=\; b + \lceil j\alpha' - \nu' \rceil - \lceil j\alpha' \rceil .
\end{aligned}$$

OK, that's pretty simple; we plug it in and plug away:

$$s(\alpha, n, \nu) \;=\; -n\nu + \lceil n\alpha \rceil b + \sum_{0 \leqslant j < \lceil n\alpha \rceil} \big(\lceil j\alpha' - \nu' \rceil - \lceil j\alpha' \rceil \big) - S , \tag{3.30}$$

where S is a correction for the cases with $k \geqslant n$ that we have failed to exclude. The quantity $j\alpha'$ will never be an integer, since α (hence α') is irrational; and $j\alpha' - \nu'$ will be an integer for at most one value of j. So we can change the

ceiling terms to floors:

$$s(\alpha,n,\nu) \;=\; -n\nu + \lceil n\alpha\rceil b - \sum_{0\leqslant j<\lceil n\alpha\rceil} \big(\lfloor j\alpha'\rfloor - \lfloor j\alpha'-\nu'\rfloor\big) - S + \{0 \text{ or } 1\}.$$

(The formula $\{0 \text{ or } 1\}$ stands for something that's either 0 or 1; we needn't commit ourselves, because the details don't really matter.)

Interesting. Instead of a closed form, we're getting a sum that looks rather like $s(\alpha,n,\nu)$ but with different parameters: α' instead of α, $\lceil n\alpha\rceil$ instead of n, and ν' instead of ν. So we'll have a recurrence for $s(\alpha,n,\nu)$, which (hopefully) will lead to a recurrence for the discrepancy $D(\alpha,n)$. This means we want to get

$$s(\alpha',\lceil n\alpha\rceil,\nu') \;=\; \sum_{0\leqslant j<\lceil n\alpha\rceil} \big(\lfloor j\alpha'\rfloor - \lfloor j\alpha'-\nu'\rfloor - \nu'\big)$$

into the act:

$$s(\alpha,n,\nu) \;=\; -n\nu + \lceil n\alpha\rceil b - \lceil n\alpha\rceil\nu' - s(\alpha',\lceil n\alpha\rceil,\nu') - S + \{0 \text{ or } 1\}.$$

Recalling that $b-\nu' = \nu\alpha^{-1}$, we see that everything will simplify beautifully if we replace $\lceil n\alpha\rceil(b-\nu')$ by $n\alpha(b-\nu') = n\nu$:

$$s(\alpha,n,\nu) \;=\; -s(\alpha',\lceil n\alpha\rceil,\nu') - S + \epsilon + \{0 \text{ or } 1\}.$$

Here ϵ is a positive error of at most $\nu\alpha^{-1}$. Exercise 18 proves that S is, similarly, between 0 and $\lceil \nu\alpha^{-1}\rceil$. And we can remove the term for $j = \lceil n\alpha\rceil - 1 = \lfloor n\alpha\rfloor$ from the sum, since it contributes either ν' or $\nu'-1$. Hence, if we take the maximum of absolute values over all ν, we get

$$D(\alpha,n) \;\leqslant\; D(\alpha',\lfloor \alpha n\rfloor) + \alpha^{-1} + 2. \tag{3.31}$$

The methods we'll learn in succeeding chapters will allow us to conclude from this recurrence that $D(\alpha,n)$ is always much smaller than n, when n is sufficiently large. Hence the theorem (3.28) is not only true, it can also be strengthened: Convergence to the limit is very fast.

Stop Skimming

Whew; that was quite an exercise in manipulation of sums, floors, and ceilings. Readers who are not accustomed to "proving that errors are small" might find it hard to believe that anybody would have the courage to keep going, when faced with such weird-looking sums. But actually, a second look shows that there's a simple motivating thread running through the whole calculation. The main idea is that a certain sum $s(\alpha,n,\nu)$ of n terms can be reduced to a similar sum of at most $\lceil \alpha n\rceil$ terms. Everything else cancels out except for a small residual left over from terms near the boundaries.

Let's take a deep breath now and do one more sum, which is not trivial but has the great advantage (compared with what we've just been doing) that

it comes out in closed form so that we can easily check the answer. Our goal now will be to generalize the sum in (3.26) by finding an expression for

Is this a harder sum of floors, or a sum of harder floors?

$$\sum_{0\leqslant k<m} \left\lfloor \frac{nk+x}{m} \right\rfloor, \qquad \text{integer } m>0, \quad \text{integer } n.$$

Finding a closed form for this sum is tougher than what we've done so far (except perhaps for the discrepancy problem we just looked at). But it's instructive, so we'll hack away at it for the rest of this chapter.

Be forewarned: This is the beginning of a pattern, in that the last part of the chapter consists of the solution of some long, difficult problem, with little more motivation than curiosity.
—Students

As usual, especially with tough problems, we start by looking at small cases. The special case $n=1$ is (3.26), with x replaced by x/m:

$$\left\lfloor \frac{x}{m} \right\rfloor + \left\lfloor \frac{1+x}{m} \right\rfloor + \cdots + \left\lfloor \frac{m-1+x}{m} \right\rfloor = \lfloor x \rfloor .$$

And as in Chapter 1, we find it useful to get more data by generalizing downwards to the case $n=0$:

$$\left\lfloor \frac{x}{m} \right\rfloor + \left\lfloor \frac{x}{m} \right\rfloor + \cdots + \left\lfloor \frac{x}{m} \right\rfloor = m \left\lfloor \frac{x}{m} \right\rfloor .$$

Touché. But c'mon, gang, do you always need to be told about applications before you can get interested in something? This sum arises, for example, in the study of random number generation and testing. But mathematicians looked at it long before computers came along, because they found it natural to ask if there's a way to sum arithmetic progressions that have been "floored."
—Your instructor

Our problem has two parameters, m and n; let's look at some small cases for m. When $m=1$ there's just a single term in the sum and its value is $\lfloor x \rfloor$. When $m=2$ the sum is $\lfloor x/2 \rfloor + \lfloor (x+n)/2 \rfloor$. We can remove the interaction between x and n by removing n from inside the floor function, but to do that we must consider even and odd n separately. If n is even, $n/2$ is an integer, so we can remove it from the floor:

$$\left\lfloor \frac{x}{2} \right\rfloor + \left(\left\lfloor \frac{x}{2} \right\rfloor + \frac{n}{2} \right) = 2 \left\lfloor \frac{x}{2} \right\rfloor + \frac{n}{2} .$$

If n is odd, $(n-1)/2$ is an integer so we get

$$\left\lfloor \frac{x}{2} \right\rfloor + \left(\left\lfloor \frac{x+1}{2} \right\rfloor + \frac{n-1}{2} \right) = \lfloor x \rfloor + \frac{n-1}{2} .$$

The last step follows from (3.26) with $m=2$.

These formulas for even and odd n slightly resemble those for $n=0$ and 1, but no clear pattern has emerged yet; so we had better continue exploring some more small cases. For $m=3$ the sum is

$$\left\lfloor \frac{x}{3} \right\rfloor + \left\lfloor \frac{x+n}{3} \right\rfloor + \left\lfloor \frac{x+2n}{3} \right\rfloor ,$$

and we consider three cases for n: Either it's a multiple of 3, or it's 1 more than a multiple, or it's 2 more. That is, $n \bmod 3 = 0$, 1, or 2. If $n \bmod 3 = 0$

then $n/3$ and $2n/3$ are integers, so the sum is

$$\left\lfloor\frac{x}{3}\right\rfloor + \left(\left\lfloor\frac{x}{3}\right\rfloor + \frac{n}{3}\right) + \left(\left\lfloor\frac{x}{3}\right\rfloor + \frac{2n}{3}\right) = 3\left\lfloor\frac{x}{3}\right\rfloor + n\,.$$

If $n \bmod 3 = 1$ then $(n-1)/3$ and $(2n-2)/3$ are integers, so we have

$$\left\lfloor\frac{x}{3}\right\rfloor + \left(\left\lfloor\frac{x+1}{3}\right\rfloor + \frac{n-1}{3}\right) + \left(\left\lfloor\frac{x+2}{3}\right\rfloor + \frac{2n-2}{3}\right) = \lfloor x\rfloor + n - 1\,.$$

Again this last step follows from (3.26), this time with $m = 3$. And finally, if $n \bmod 3 = 2$ then

$$\left\lfloor\frac{x}{3}\right\rfloor + \left(\left\lfloor\frac{x+2}{3}\right\rfloor + \frac{n-2}{3}\right) + \left(\left\lfloor\frac{x+1}{3}\right\rfloor + \frac{2n-1}{3}\right) = \lfloor x\rfloor + n - 1\,.$$

"Inventive genius requires pleasurable mental activity as a condition for its vigorous exercise. 'Necessity is the mother of invention' is a silly proverb. 'Necessity is the mother of futile dodges' is much nearer to the truth. The basis of the growth of modern invention is science, and science is almost wholly the outgrowth of pleasurable intellectual curiosity."
—A. N. Whitehead [371]

The left hemispheres of our brains have finished the case $m = 3$, but the right hemispheres still can't recognize the pattern, so we proceed to $m = 4$:

$$\left\lfloor\frac{x}{4}\right\rfloor + \left\lfloor\frac{x+n}{4}\right\rfloor + \left\lfloor\frac{x+2n}{4}\right\rfloor + \left\lfloor\frac{x+3n}{4}\right\rfloor\,.$$

At least we know enough by now to consider cases based on $n \bmod m$. If $n \bmod 4 = 0$ then

$$\left\lfloor\frac{x}{4}\right\rfloor + \left(\left\lfloor\frac{x}{4}\right\rfloor + \frac{n}{4}\right) + \left(\left\lfloor\frac{x}{4}\right\rfloor + \frac{2n}{4}\right) + \left(\left\lfloor\frac{x}{4}\right\rfloor + \frac{3n}{4}\right) = 4\left\lfloor\frac{x}{4}\right\rfloor + \frac{3n}{2}\,.$$

And if $n \bmod 4 = 1$,

$$\left\lfloor\frac{x}{4}\right\rfloor + \left(\left\lfloor\frac{x+1}{4}\right\rfloor + \frac{n-1}{4}\right) + \left(\left\lfloor\frac{x+2}{4}\right\rfloor + \frac{2n-2}{4}\right) + \left(\left\lfloor\frac{x+3}{4}\right\rfloor + \frac{3n-3}{4}\right)$$
$$= \lfloor x\rfloor + \frac{3n}{2} - \frac{3}{2}\,.$$

The case $n \bmod 4 = 3$ turns out to give the same answer. Finally, in the case $n \bmod 4 = 2$ we get something a bit different, and this turns out to be an important clue to the behavior in general:

$$\left\lfloor\frac{x}{4}\right\rfloor + \left(\left\lfloor\frac{x+2}{4}\right\rfloor + \frac{n-2}{4}\right) + \left(\left\lfloor\frac{x}{4}\right\rfloor + \frac{2n}{4}\right) + \left(\left\lfloor\frac{x+2}{4}\right\rfloor + \frac{3n-2}{4}\right)$$
$$= 2\left(\left\lfloor\frac{x}{4}\right\rfloor + \left\lfloor\frac{x+2}{4}\right\rfloor\right) + \frac{3n}{2} - 1 = 2\left\lfloor\frac{x}{2}\right\rfloor + \frac{3n}{2} - 1\,.$$

This last step simplifies something of the form $\lfloor y/2\rfloor + \lfloor (y+1)/2\rfloor$, which again is a special case of (3.26).

To summarize, here's the value of our sum for small m:

m	$n \bmod m = 0$	$n \bmod m = 1$	$n \bmod m = 2$	$n \bmod m = 3$
1	$\lfloor x \rfloor$			
2	$2\left\lfloor \frac{x}{2} \right\rfloor + \frac{n}{2}$	$\lfloor x \rfloor + \frac{n}{2} - \frac{1}{2}$		
3	$3\left\lfloor \frac{x}{3} \right\rfloor + n$	$\lfloor x \rfloor + n - 1$	$\lfloor x \rfloor + n - 1$	
4	$4\left\lfloor \frac{x}{4} \right\rfloor + \frac{3n}{2}$	$\lfloor x \rfloor + \frac{3n}{2} - \frac{3}{2}$	$2\left\lfloor \frac{x}{2} \right\rfloor + \frac{3n}{2} - 1$	$\lfloor x \rfloor + \frac{3n}{2} - \frac{3}{2}$

It looks as if we're getting something of the form

$$a\left\lfloor \frac{x}{a} \right\rfloor + bn + c\,,$$

where a, b, and c somehow depend on m and n. Even the myopic among us can see that b is probably $(m-1)/2$. It's harder to discern an expression for a; but the case $n \bmod 4 = 2$ gives us a hint that a is probably $\gcd(m,n)$, the greatest common divisor of m and n. This makes sense because $\gcd(m,n)$ is the factor we remove from m and n when reducing the fraction n/m to lowest terms, and our sum involves the fraction n/m. (We'll look carefully at gcd operations in Chapter 4.) The value of c seems more mysterious, but perhaps it will drop out of our proofs for a and b.

In computing the sum for small m, we've effectively rewritten each term of the sum as

$$\left\lfloor \frac{x+kn}{m} \right\rfloor = \left\lfloor \frac{x + kn \bmod m}{m} \right\rfloor + \frac{kn}{m} - \frac{kn \bmod m}{m}\,,$$

because $(kn - kn \bmod m)/m$ is an integer that can be removed from inside the floor brackets. Thus the original sum can be expanded into the following tableau:

$$\begin{array}{ccccccc}
 & \left\lfloor \frac{x}{m} \right\rfloor & + & \frac{0}{m} & - & \frac{0 \bmod m}{m} \\
+ & \left\lfloor \frac{x + n \bmod m}{m} \right\rfloor & + & \frac{n}{m} & - & \frac{n \bmod m}{m} \\
+ & \left\lfloor \frac{x + 2n \bmod m}{m} \right\rfloor & + & \frac{2n}{m} & - & \frac{2n \bmod m}{m} \\
 & \vdots & & \vdots & & \vdots \\
+ & \left\lfloor \frac{x + (m-1)n \bmod m}{m} \right\rfloor & + & \frac{(m-1)n}{m} & - & \frac{(m-1)n \bmod m}{m}\,.
\end{array}$$

When we experimented with small values of m, these three columns led respectively to $a\lfloor x/a\rfloor$, bn, and c.

In particular, we can see how b arises. The second column is an arithmetic progression, whose sum we know — it's the average of the first and last terms, times the number of terms:

$$\frac{1}{2}\left(0+\frac{(m-1)n}{m}\right)\cdot m \;=\; \frac{(m-1)n}{2}\,.$$

So our guess that $b=(m-1)/2$ has been verified.

The first and third columns seem tougher; to determine a and c we must take a closer look at the sequence of numbers

$$0 \bmod m,\;\; n \bmod m,\;\; 2n \bmod m,\;\; \ldots,\;\; (m-1)n \bmod m.$$

Suppose, for example, that $m=12$ and $n=5$. If we think of the sequence as times on a clock, the numbers are 0 o'clock (we take 12 o'clock to be 0 o'clock), then 5 o'clock, 10 o'clock, 3 o'clock (= 15 o'clock), 8 o'clock, and so on. It turns out that we hit every hour exactly once.

Now suppose $m=12$ and $n=8$. The numbers are 0 o'clock, 8 o'clock, 4 o'clock (= 16 o'clock), but then 0, 8, and 4 repeat. Since both 8 and 12 are multiples of 4, and since the numbers start at 0 (also a multiple of 4), there's no way to break out of this pattern — they must all be multiples of 4.

Lemma now, dilemma later.

In these two cases we have $\gcd(12,5)=1$ and $\gcd(12,8)=4$. The general rule, which we will prove next chapter, states that if $d=\gcd(m,n)$ then we get the numbers $0,\ d,\ 2d,\ \ldots,\ m-d$ in some order, followed by $d-1$ more copies of the same sequence. For example, with $m=12$ and $n=8$ the pattern 0, 8, 4 occurs four times.

The first column of our sum now makes complete sense. It contains d copies of the terms $\lfloor x/m\rfloor,\ \lfloor(x+d)/m\rfloor,\ \ldots,\ \lfloor(x+m-d)/m\rfloor$, in some order, so its sum is

$$\begin{aligned}
&d\left(\left\lfloor\frac{x}{m}\right\rfloor+\left\lfloor\frac{x+d}{m}\right\rfloor+\cdots+\left\lfloor\frac{x+m-d}{m}\right\rfloor\right)\\
&\quad= d\left(\left\lfloor\frac{x/d}{m/d}\right\rfloor+\left\lfloor\frac{x/d+1}{m/d}\right\rfloor+\cdots+\left\lfloor\frac{x/d+m/d-1}{m/d}\right\rfloor\right)\\
&\quad= d\left\lfloor\frac{x}{d}\right\rfloor.
\end{aligned}$$

This last step is yet another application of (3.26). Our guess for a has been verified:

$$a \;=\; d \;=\; \gcd(m,n)\,.$$

Also, as we guessed, we can now compute c, because the third column has become easy to fathom. It contains d copies of the arithmetic progression $0/m$, d/m, $2d/m$, $\ldots$, $(m-d)/m$, so its sum is

$$d\left(\frac{1}{2}\left(0+\frac{m-d}{m}\right)\cdot\frac{m}{d}\right) = \frac{m-d}{2};$$

the third column is actually subtracted, not added, so we have

$$c = \frac{d-m}{2}.$$

End of mystery, end of quest. The desired closed form is

$$\sum_{0\leqslant k<m}\left\lfloor\frac{nk+x}{m}\right\rfloor = d\left\lfloor\frac{x}{d}\right\rfloor + \frac{m-1}{2}n + \frac{d-m}{2},$$

where $d=\gcd(m,n)$. As a check, we can make sure this works in the special cases $n=0$ and $n=1$ that we knew before: When $n=0$ we get $d=\gcd(m,0)=m$; the last two terms of the formula are zero so the formula properly gives $m\lfloor x/m\rfloor$. And for $n=1$ we get $d=\gcd(m,1)=1$; the last two terms cancel nicely, and the sum is just $\lfloor x\rfloor$.

By manipulating the closed form a bit, we can actually make it symmetric in m and n:

$$\begin{aligned}\sum_{0\leqslant k<m}\left\lfloor\frac{nk+x}{m}\right\rfloor &= d\left\lfloor\frac{x}{d}\right\rfloor + \frac{m-1}{2}n + \frac{d-m}{2}\\ &= d\left\lfloor\frac{x}{d}\right\rfloor + \frac{(m-1)(n-1)}{2} + \frac{m-1}{2} + \frac{d-m}{2}\\ &= d\left\lfloor\frac{x}{d}\right\rfloor + \frac{(m-1)(n-1)}{2} + \frac{d-1}{2}. \qquad (3.32)\end{aligned}$$

This is astonishing, because there's no algebraic reason to suspect that such a sum should be symmetrical. We have proved a "reciprocity law,"

Yup, I'm floored.

$$\sum_{0\leqslant k<m}\left\lfloor\frac{nk+x}{m}\right\rfloor = \sum_{0\leqslant k<n}\left\lfloor\frac{mk+x}{n}\right\rfloor, \qquad \text{integers } m,n>0.$$

For example, if $m=41$ and $n=127$, the left sum has 41 terms and the right has 127; but they still come out equal, for all real x.

Exercises

Warmups

1 When we analyzed the Josephus problem in Chapter 1, we represented an arbitrary positive integer n in the form $n = 2^m + l$, where $0 \leqslant l < 2^m$. Give explicit formulas for l and m as functions of n, using floor and/or ceiling brackets.

2 What is a formula for the nearest integer to a given real number x? In case of ties, when x is exactly halfway between two integers, give an expression that rounds (a) up — that is, to $\lceil x \rceil$; (b) down — that is, to $\lfloor x \rfloor$.

3 Evaluate $\lfloor \lfloor m\alpha \rfloor n/\alpha \rfloor$, when m and n are positive integers and α is an irrational number greater than n.

4 The text describes problems at levels 1 through 5. What is a level 0 problem? (This, by the way, is *not* a level 0 problem.)

5 Find a necessary and sufficient condition that $\lfloor nx \rfloor = n\lfloor x \rfloor$, when n is a positive integer. (Your condition should involve $\{x\}$.)

6 Can something interesting be said about $\lfloor f(x) \rfloor$ when $f(x)$ is a continuous, monotonically *decreasing* function that takes integer values only when x is an integer?

7 Solve the recurrence

$$\begin{aligned} X_n &= n\,, && \text{for } 0 \leqslant n < m; \\ X_n &= X_{n-m} + 1\,, && \text{for } n \geqslant m. \end{aligned}$$

You know you're in college when the book doesn't tell you how to pronounce 'Dirichlet'.

8 Prove the *Dirichlet box principle*: If n objects are put into m boxes, some box must contain $\geqslant \lceil n/m \rceil$ objects, and some box must contain $\leqslant \lfloor n/m \rfloor$.

9 Egyptian mathematicians in 1800 B.C. represented rational numbers between 0 and 1 as sums of unit fractions $1/x_1 + \cdots + 1/x_k$, where the x's were distinct positive integers. For example, they wrote $\frac{1}{3} + \frac{1}{15}$ instead of $\frac{2}{5}$. Prove that it is always possible to do this in a systematic way: If $0 < m/n < 1$, then

$$\frac{m}{n} = \frac{1}{q} + \left\{ \text{representation of } \frac{m}{n} - \frac{1}{q} \right\}, \qquad q = \left\lceil \frac{n}{m} \right\rceil .$$

(This is *Fibonacci's algorithm*, due to Leonardo Fibonacci, A.D. 1202.)

Basics

10 Show that the expression

$$\left\lceil \frac{2x+1}{2} \right\rceil - \left\lceil \frac{2x+1}{4} \right\rceil + \left\lfloor \frac{2x+1}{4} \right\rfloor$$

is always either $\lfloor x \rfloor$ or $\lceil x \rceil$. In what circumstances does each case arise?

11 Give details of the proof alluded to in the text, that the open interval $(\alpha\,..\,\beta)$ contains exactly $\lceil \beta \rceil - \lfloor \alpha \rfloor - 1$ integers when $\alpha < \beta$. Why does the case $\alpha = \beta$ have to be excluded in order to make the proof correct?

12 Prove that

$$\left\lceil \frac{n}{m} \right\rceil \;=\; \left\lfloor \frac{n+m-1}{m} \right\rfloor ,$$

for all integers n and all positive integers m. [This identity gives us another way to convert ceilings to floors and vice versa, instead of using the reflective law (3.4).]

13 Let α and β be positive real numbers. Prove that $\mathrm{Spec}(\alpha)$ and $\mathrm{Spec}(\beta)$ partition the positive integers if and only if α and β are irrational and $1/\alpha + 1/\beta = 1$.

14 Prove or disprove:

$$(x \bmod ny) \bmod y \;=\; x \bmod y\,, \qquad \text{integer } n.$$

15 Is there an identity analogous to (3.26) that uses ceilings instead of floors?

16 Prove that $n \bmod 2 = \big(1-(-1)^n\big)/2$. Find and prove a similar expression for $n \bmod 3$ in the form $a+b\omega^n+c\omega^{2n}$, where ω is the complex number $(-1+i\sqrt{3}\,)/2$. *Hint:* $\omega^3 = 1$ and $1+\omega+\omega^2 = 0$.

17 Evaluate the sum $\sum_{0\leqslant k<m} \lfloor x + k/m \rfloor$ in the case $x \geqslant 0$ by substituting $\sum_j [1 \leqslant j \leqslant x + k/m]$ for $\lfloor x + k/m \rfloor$ and summing first on k. Does your answer agree with (3.26)?

18 Prove that the boundary-value error term S in (3.30) is at most $\lceil \alpha^{-1}\nu \rceil$. *Hint:* Show that small values of j are not involved.

Homework exercises

19 Find a necessary and sufficient condition on the real number $b > 1$ such that

$$\lfloor \log_b x \rfloor \;=\; \big\lfloor \log_b \lfloor x \rfloor \big\rfloor$$

for all real $x \geqslant 1$.

20 Find the sum of all multiples of x in the closed interval $[\alpha\,..\,\beta]$, when $x > 0$.

21 How many of the numbers 2^m, for $0 \leqslant m \leqslant M$, have leading digit 1 in decimal notation?

22 Evaluate the sums $S_n = \sum_{k\geqslant 1} \lfloor n/2^k + \frac{1}{2} \rfloor$ and $T_n = \sum_{k\geqslant 1} 2^k \lfloor n/2^k + \frac{1}{2} \rfloor^2$.

23 Show that the nth element of the sequence

$$1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, \ldots$$

is $\lfloor \sqrt{2n} + \frac{1}{2} \rfloor$. (The sequence contains exactly m occurrences of m.)

24 Exercise 13 establishes an interesting relation between the two multisets $\mathrm{Spec}(\alpha)$ and $\mathrm{Spec}\big(\alpha/(\alpha-1)\big)$, when α is any irrational number > 1, because $1/\alpha + (\alpha-1)/\alpha = 1$. Find (and prove) an interesting relation between the two multisets $\mathrm{Spec}(\alpha)$ and $\mathrm{Spec}\big(\alpha/(\alpha+1)\big)$, when α is any positive real number.

25 Prove or disprove that the Knuth numbers, defined by (3.16), satisfy $K_n \geqslant n$ for all nonnegative n.

26 Show that the auxiliary Josephus numbers (3.20) satisfy

$$\left(\frac{q}{q-1}\right)^n \leqslant D_n^{(q)} \leqslant q\left(\frac{q}{q-1}\right)^n, \qquad \text{for } n \geqslant 0.$$

27 Prove that infinitely many of the numbers $D_n^{(3)}$ defined by (3.20) are even, and that infinitely many are odd.

28 Solve the recurrence

$$\begin{aligned} a_0 &= 1; \\ a_n &= a_{n-1} + \lfloor \sqrt{a_{n-1}} \rfloor, \qquad \text{for } n > 0. \end{aligned}$$

There's a discrepancy between this formula and (3.31).

29 Show that, in addition to (3.31), we have

$$D(\alpha, n) \;\geqslant\; D\big(\alpha', \lfloor \alpha n \rfloor\big) - \alpha^{-1} - 2\,.$$

30 Show that the recurrence

$$\begin{aligned} X_0 &= m\,, \\ X_n &= X_{n-1}^2 - 2\,, \qquad \text{for } n > 0, \end{aligned}$$

has the solution $X_n = \lceil \alpha^{2^n} \rceil$, if m is an integer greater than 2, where $\alpha + \alpha^{-1} = m$ and $\alpha > 1$. For example, if $m = 3$ the solution is

$$X_n = \lceil \phi^{2^{n+1}} \rceil, \qquad \phi = \frac{1+\sqrt{5}}{2}, \qquad \alpha = \phi^2\,.$$

31 Prove or disprove: $\lfloor x\rfloor + \lfloor y\rfloor + \lfloor x+y\rfloor \leqslant \lfloor 2x\rfloor + \lfloor 2y\rfloor$.

32 Let $\|x\| = \min\bigl(x-\lfloor x\rfloor, \lceil x\rceil - x\bigr)$ denote the distance from x to the nearest integer. What is the value of

$$\sum_k 2^k \bigl\|x/2^k\bigr\|^2 ?$$

(Note that this sum can be doubly infinite. For example, when $x = 1/3$ the terms are nonzero as $k \to -\infty$ and also as $k \to +\infty$.)

Exam problems

33 A circle, $2n-1$ units in diameter, has been drawn symmetrically on a $2n \times 2n$ chessboard, illustrated here for $n = 3$:

a How many cells of the board contain a segment of the circle?

b Find a function $f(n,k)$ such that exactly $\sum_{k=1}^{n-1} f(n,k)$ cells of the board lie entirely within the circle.

34 Let $f(n) = \sum_{k=1}^{n} \lceil \lg k\rceil$.

a Find a closed form for $f(n)$, when $n \geqslant 1$.

b Prove that $f(n) = n - 1 + f\bigl(\lceil n/2\rceil\bigr) + f\bigl(\lfloor n/2\rfloor\bigr)$ for all $n \geqslant 1$.

35 Simplify the formula $\lfloor (n+1)^2 n!\, e\rfloor \bmod n$.

Simplify it, but don't change the value.

36 Assuming that n is a nonnegative integer, find a closed form for the sum

$$\sum_{1<k<2^{2^n}} \frac{1}{2^{\lfloor \lg k\rfloor} 4^{\lfloor \lg\lg k\rfloor}} .$$

37 Prove the identity

$$\sum_{0\leqslant k<m} \left(\left\lfloor \frac{m+k}{n}\right\rfloor - \left\lfloor \frac{k}{n}\right\rfloor\right) = \left\lfloor \frac{m^2}{n}\right\rfloor - \left\lfloor \frac{\min\bigl(m \bmod n, (-m) \bmod n\bigr)^2}{n}\right\rfloor$$

for all positive integers m and n.

38 Let $x_1, \ldots, x_n$ be real numbers such that the identity

$$\sum_{k=1}^{n} \lfloor m x_k\rfloor = \left\lfloor m \sum_{1\leqslant k\leqslant n} x_k\right\rfloor$$

holds for all positive integers m. Prove something interesting about $x_1, \ldots, x_n$.

39 Prove that the double sum $\sum_{0\leqslant k\leqslant \log_b x}\sum_{0<j<b}\lceil (x+jb^k)/b^{k+1}\rceil$ equals $(b-1)\big(\lfloor \log_b x\rfloor+1\big)+\lceil x\rceil-1$, for every real number $x\geqslant 1$ and every integer $b>1$.

40 The spiral function $\sigma(n)$, indicated in the diagram below, maps a nonnegative integer n onto an ordered pair of integers $\big(x(n),y(n)\big)$. For example, it maps $n=9$ onto the ordered pair $(1,2)$.

People in the southern hemisphere use a different spiral.

a Prove that if $m=\lfloor\sqrt{n}\rfloor$,

$$x(n) \;=\; (-1)^m\Big(\big(n-m(m+1)\big)\cdot\big[\lfloor 2\sqrt{n}\rfloor \text{ is even}\big]+\lceil \tfrac12 m\rceil\Big),$$

and find a similar formula for $y(n)$. *Hint:* Classify the spiral into segments W_k, S_k, E_k, N_k according as $\lfloor 2\sqrt{n}\rfloor=4k-2$, $4k-1$, $4k$, $4k+1$.

b Prove that, conversely, we can determine n from $\sigma(n)$ by a formula of the form

$$n \;=\; (2k)^2\pm\big(2k+x(n)+y(n)\big),\quad k \;=\; \max\big(|x(n)|,|y(n)|\big).$$

Give a rule for when the sign is $+$ and when the sign is $-$.

Bonus problems

41 Let f and g be increasing functions such that the sets $\{f(1),f(2),\ldots\}$ and $\{g(1),g(2),\ldots\}$ partition the positive integers. Suppose that f and g are related by the condition $g(n)=f\big(f(n)\big)+1$ for all $n>0$. Prove that $f(n)=\lfloor n\phi\rfloor$ and $g(n)=\lfloor n\phi^2\rfloor$, where $\phi=(1+\sqrt{5})/2$.

42 Do there exist real numbers α, β, and γ such that $\mathrm{Spec}(\alpha)$, $\mathrm{Spec}(\beta)$, and $\mathrm{Spec}(\gamma)$ together partition the set of positive integers?

43 Find an interesting interpretation of the Knuth numbers, by unfolding the recurrence (3.16).

44 Show that there are integers $a_n^{(q)}$ and $d_n^{(q)}$ such that

$$a_n^{(q)} = \frac{D_{n-1}^{(q)} + d_n^{(q)}}{q-1} = \frac{D_n^{(q)} + d_n^{(q)}}{q}, \qquad \text{for } n > 0,$$

when $D_n^{(q)}$ is the solution to (3.20). Use this fact to obtain another form of the solution to the generalized Josephus problem:

$$J_q(n) = 1 + d_k^{(q)} + q(n - a_k^{(q)}), \qquad \text{for } a_k^{(q)} \leqslant n < a_{k+1}^{(q)}.$$

45 Extend the trick of exercise 30 to find a closed-form solution to

$$\begin{aligned} Y_0 &= m, \\ Y_n &= 2Y_{n-1}^2 - 1, \qquad \text{for } n > 0, \end{aligned}$$

if m is a positive integer.

46 Prove that if $n = \lfloor (\sqrt{2}^l + \sqrt{2}^{l-1})m \rfloor$, where m and l are nonnegative integers, then $\lfloor \sqrt{2n(n+1)} \rfloor = \lfloor (\sqrt{2}^{l+1} + \sqrt{2}^l)m \rfloor$. Use this remarkable property to find a closed form solution to the recurrence

$$\begin{aligned} L_0 &= a, && \text{integer } a > 0; \\ L_n &= \lfloor \sqrt{2L_{n-1}(L_{n-1}+1)} \rfloor, && \text{for } n > 0. \end{aligned}$$

Hint: $\lfloor \sqrt{2n(n+1)} \rfloor = \lfloor \sqrt{2}(n + \frac{1}{2}) \rfloor$.

47 The function $f(x)$ is said to be *replicative* if it satisfies

$$f(mx) = f(x) + f\Bigl(x + \frac{1}{m}\Bigr) + \cdots + f\Bigl(x + \frac{m-1}{m}\Bigr)$$

for every positive integer m. Find necessary and sufficient conditions on the real number c for the following functions to be replicative:

a $f(x) = x + c$.

b $f(x) = [x + c \text{ is an integer}]$.

c $f(x) = \max\bigl(\lfloor x \rfloor, c\bigr)$.

d $f(x) = x + c\lfloor x \rfloor - \frac{1}{2}[x \text{ is not an integer}]$.

48 Prove the identity

$$x^3 = 3x\lfloor x\lfloor x \rfloor \rfloor + 3\{x\}\bigl\{x\lfloor x \rfloor\bigr\} + \{x\}^3 - 3\lfloor x \rfloor \lfloor x\lfloor x \rfloor \rfloor + \lfloor x \rfloor^3,$$

and show how to obtain similar formulas for x^n when $n > 3$.

49 Find a necessary and sufficient condition on the real numbers $0 \leqslant \alpha < 1$ and $\beta \geqslant 0$ such that we can determine α and β from the infinite multiset of values

$$\{\, \lfloor n\alpha \rfloor + \lfloor n\beta \rfloor \;\mid\; n > 0 \,\}.$$

Research problems

50 Find a necessary and sufficient condition on the nonnegative real numbers α and β such that we can determine α and β from the infinite multiset of values

$$\{\, \lfloor \lfloor n\alpha \rfloor \beta \rfloor \;\mid\; n > 0 \,\}.$$

51 Let x be a real number $\geqslant \phi = \frac{1}{2}(1+\sqrt{5})$. The solution to the recurrence

$$\begin{aligned} Z_0(x) &= x\,, \\ Z_n(x) &= Z_{n-1}(x)^2 - 1\,, \qquad \text{for } n > 0, \end{aligned}$$

can be written $Z_n(x) = \lceil f(x)^{2^n} \rceil$, if x is an integer, where

$$f(x) = \lim_{n\to\infty} Z_n(x)^{1/2^n}\,,$$

because $Z_n(x) - 1 < f(x)^{2^n} < Z_n(x)$ in that case. What other interesting properties does this function $f(x)$ have?

52 Given nonnegative real numbers α and β, let

$$\mathrm{Spec}(\alpha;\beta) = \{\lfloor \alpha+\beta \rfloor, \lfloor 2\alpha+\beta \rfloor, \lfloor 3\alpha+\beta \rfloor, \ldots\}$$

Spec this be hard.

be a multiset that generalizes $\mathrm{Spec}(\alpha) = \mathrm{Spec}(\alpha;0)$. Prove or disprove: If the $m \geqslant 3$ multisets $\mathrm{Spec}(\alpha_1;\beta_1)$, $\mathrm{Spec}(\alpha_2;\beta_2)$, $\ldots$, $\mathrm{Spec}(\alpha_m;\beta_m)$ partition the positive integers, and if the parameters $\alpha_1 < \alpha_2 < \cdots < \alpha_m$ are rational, then

$$\alpha_k = \frac{2^m - 1}{2^{k-1}}\,, \qquad \text{for } 1 \leqslant k \leqslant m.$$

53 Fibonacci's algorithm (exercise 9) is "greedy" in the sense that it chooses the least conceivable q at every step. A more complicated algorithm is known by which every fraction m/n with n odd can be represented as a sum of distinct unit fractions $1/q_1 + \cdots + 1/q_k$ with *odd* denominators. Does the greedy algorithm for such a representation always terminate?

4

Number Theory

INTEGERS ARE CENTRAL to the discrete mathematics we are emphasizing in this book. Therefore we want to explore the *theory of numbers*, an important branch of mathematics concerned with the properties of integers.

We tested the number theory waters in the previous chapter, by introducing binary operations called 'mod' and 'gcd'. Now let's plunge in and really immerse ourselves in the subject.

In other words, be prepared to drown.

4.1 DIVISIBILITY

We say that m divides n (or n is divisible by m) if $m > 0$ and the ratio n/m is an integer. This property underlies all of number theory, so it's convenient to have a special notation for it. We therefore write

$$m \backslash n \qquad \Longleftrightarrow \qquad m > 0 \text{ and } n = mk \text{ for some integer } k. \tag{4.1}$$

(The notation '$m|n$' is actually much more common than '$m\backslash n$' in current mathematics literature. But vertical lines are overused — for absolute values, set delimiters, conditional probabilities, etc. — and backward slashes are underused. Moreover, '$m\backslash n$' gives an impression that m is the denominator of an implied ratio. So we shall boldly let our divisibility symbol lean leftward.)

If m does not divide n we write '$m \not\backslash n$'.

There's a similar relation, "n is a multiple of m," which means almost the same thing except that m doesn't have to be positive. In this case we simply mean that $n = mk$ for some integer k. Thus, for example, there's only one multiple of 0 (namely 0), but nothing is divisible by 0. Every integer is a multiple of -1, but no integer is divisible by -1 (strictly speaking). These definitions apply when m and n are any real numbers; for example, 2π is divisible by π. But we'll almost always be using them when m and n are integers. After all, this is number theory.

"... no integer is divisible by -1 (strictly speaking)." — Graham, Knuth, and Patashnik [161]

In Britain we call this 'hcf' (highest common factor).

The *greatest common divisor* of two integers m and n is the largest integer that divides them both:

$$\gcd(m,n) \;=\; \max\{\,k \mid k\backslash m \quad\text{and}\quad k\backslash n\,\}. \tag{4.2}$$

For example, $\gcd(12,18) = 6$. This is a familiar notion, because it's the common factor that fourth graders learn to take out of a fraction m/n when reducing it to lowest terms: $12/18 = (12/6)/(18/6) = 2/3$. Notice that if $n > 0$ we have $\gcd(0,n) = n$, because any positive number divides 0, and because n is the largest divisor of itself. The value of $\gcd(0,0)$ is undefined.

Not to be confused with the greatest common multiple.

Another familiar notion is the *least common multiple*,

$$\operatorname{lcm}(m,n) \;=\; \min\{\,k \mid k>0,\quad m\backslash k \quad\text{and}\quad n\backslash k\,\}; \tag{4.3}$$

this is undefined if $m \leqslant 0$ or $n \leqslant 0$. Students of arithmetic recognize this as the least common denominator, which is used when adding fractions with denominators m and n. For example, $\operatorname{lcm}(12,18) = 36$, and fourth graders know that $\frac{7}{12} + \frac{1}{18} = \frac{21}{36} + \frac{2}{36} = \frac{23}{36}$. The lcm is somewhat analogous to the gcd, but we don't give it equal time because the gcd has nicer properties.

One of the nicest properties of the gcd is that it is easy to compute, using a 2300-year-old method called *Euclid's algorithm*. To calculate $\gcd(m,n)$, for given values $0 \leqslant m < n$, Euclid's algorithm uses the recurrence

$$\begin{aligned}\gcd(0,n) \;&=\; n\,;\\ \gcd(m,n) \;&=\; \gcd(n \bmod m,\, m)\,, \qquad \text{for } m>0.\end{aligned} \tag{4.4}$$

Thus, for example, $\gcd(12,18) = \gcd(6,12) = \gcd(0,6) = 6$. The stated recurrence is valid, because any common divisor of m and n must also be a common divisor of both m and the number $n \bmod m$, which is $n - \lfloor n/m \rfloor m$. There doesn't seem to be any recurrence for $\operatorname{lcm}(m,n)$ that's anywhere near as simple as this. (See exercise 2.)

Euclid's algorithm also gives us more: We can extend it so that it will compute integers m' and n' satisfying

$$m'm + n'n \;=\; \gcd(m,n)\,. \tag{4.5}$$

(Remember that m' or n' can be negative.)

Here's how. If $m = 0$, we simply take $m' = 0$ and $n' = 1$. Otherwise we let $r = n \bmod m$ and apply the method recursively with r and m in place of m and n, computing $\bar{r}$ and $\overline{m}$ such that

$$\bar{r}\,r + \overline{m}\,m \;=\; \gcd(r,m)\,.$$

Since $r = n - \lfloor n/m \rfloor m$ and $\gcd(r,m) = \gcd(m,n)$, this equation tells us that

$$\bar{r}\,\big(n - \lfloor n/m \rfloor m\big) + \overline{m}\,m \;=\; \gcd(m,n)\,.$$

The left side can be rewritten to show its dependency on m and n:

$$\left(\overline{m} - \lfloor n/m \rfloor \overline{r}\right) m + \overline{r}\, n \;=\; \gcd(m,n)\,;$$

hence $m' = \overline{m} - \lfloor n/m \rfloor \overline{r}$ and $n' = \overline{r}$ are the integers we need in (4.5). For example, in our favorite case $m = 12$, $n = 18$, this method gives $6 = 0{\cdot}0 + 1{\cdot}6 = 1\cdot 6 + 0\cdot 12 = (-1)\cdot 12 + 1\cdot 18$.

But why is (4.5) such a neat result? The main reason is that there's a sense in which the numbers m' and n' actually *prove* that Euclid's algorithm has produced the correct answer in any particular case. Let's suppose that our computer has told us after a lengthy calculation that $\gcd(m,n) = d$ and that $m'm + n'n = d$; but we're skeptical and think that there's really a greater common divisor, which the machine has somehow overlooked. This cannot be, however, because any common divisor of m and n has to divide $m'm + n'n$; so it has to divide d; so it has to be $\leqslant d$. Furthermore we can easily check that d does divide both m and n. (Algorithms that output their own proofs of correctness are called *self-certifying*.)

We'll be using (4.5) a lot in the rest of this chapter. One of its important consequences is the following mini-theorem:

$$k \backslash m \quad \text{and} \quad k \backslash n \qquad \Longleftrightarrow \qquad k \backslash \gcd(m,n)\,. \tag{4.6}$$

(Proof: If k divides both m and n, it divides $m'm + n'n$, so it divides $\gcd(m,n)$. Conversely, if k divides $\gcd(m,n)$, it divides a divisor of m and a divisor of n, so it divides both m and n.) We always knew that any common divisor of m and n must be *less than or equal to* their gcd; that's the definition of greatest common divisor. But now we know that any common divisor is, in fact, *a divisor of* their gcd.

Sometimes we need to do sums over all divisors of n. In this case it's often useful to use the handy rule

$$\sum_{m \backslash n} a_m \;=\; \sum_{m \backslash n} a_{n/m}\,, \qquad \text{integer } n > 0, \tag{4.7}$$

which holds since n/m runs through all divisors of n when m does. For example, when $n = 12$ this says that $a_1 + a_2 + a_3 + a_4 + a_6 + a_{12} = a_{12} + a_6 + a_4 + a_3 + a_2 + a_1$.

There's also a slightly more general identity,

$$\sum_{m \backslash n} a_m \;=\; \sum_k \sum_{m>0} a_m [n = mk]\,, \tag{4.8}$$

which is an immediate consequence of the definition (4.1). If n is positive, the right-hand side of (4.8) is $\sum_{k \backslash n} a_{n/k}$; hence (4.8) implies (4.7). And equation

(4.8) works also when n is negative. (In such cases, the nonzero terms on the right occur when k is the negative of a divisor of n.)

Moreover, a double sum over divisors can be "interchanged" by the law

$$\sum_{m\backslash n}\sum_{k\backslash m} a_{k,m} = \sum_{k\backslash n}\sum_{l\backslash(n/k)} a_{k,kl}\,. \tag{4.9}$$

For example, this law takes the following form when $n = 12$:

$$\begin{aligned}
a_{1,1} &+ (a_{1,2}+a_{2,2}) + (a_{1,3}+a_{3,3}) \\
&\quad + (a_{1,4}+a_{2,4}+a_{4,4}) + (a_{1,6}+a_{2,6}+a_{3,6}+a_{6,6}) \\
&\quad + (a_{1,12}+a_{2,12}+a_{3,12}+a_{4,12}+a_{6,12}+a_{12,12}) \\
&= (a_{1,1}+a_{1,2}+a_{1,3}+a_{1,4}+a_{1,6}+a_{1,12}) \\
&\quad + (a_{2,2}+a_{2,4}+a_{2,6}+a_{2,12}) + (a_{3,3}+a_{3,6}+a_{3,12}) \\
&\quad + (a_{4,4}+a_{4,12}) + (a_{6,6}+a_{6,12}) + a_{12,12}\,.
\end{aligned}$$

We can prove (4.9) with Iversonian manipulation. The left-hand side is

$$\sum_{j,l}\sum_{k,m>0} a_{k,m}[n=jm][m=kl] = \sum_{j}\sum_{k,l>0} a_{k,kl}[n=jkl]\,;$$

the right-hand side is

$$\sum_{j,m}\sum_{k,l>0} a_{k,kl}[n=jk][n/k=ml] = \sum_{m}\sum_{k,l>0} a_{k,kl}[n=mlk]\,,$$

which is the same except for renaming the indices. This example indicates that the techniques we've learned in Chapter 2 will come in handy as we study number theory.

4.2 PRIMES

How about the p in 'explicitly'?

A positive integer p is called *prime* if it has just two divisors, namely 1 and p. *Throughout the rest of this chapter, the letter* p *will always stand for a prime number, even when we don't say so explicitly.* By convention, 1 isn't prime, so the sequence of primes starts out like this:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, \ldots\,.$$

Some numbers look prime but aren't, like 91 ($= 7{\cdot}13$) and 161 ($= 7{\cdot}23$). These numbers and others that have three or more divisors are called *composite*. Every integer greater than 1 is either prime or composite, but not both.

Primes are of great importance, because they're the fundamental building blocks of all the positive integers. Any positive integer n can be written as a

product of primes,

$$n = p_1 \ldots p_m = \prod_{k=1}^{m} p_k, \qquad p_1 \leqslant \cdots \leqslant p_m. \tag{4.10}$$

For example, $12 = 2 \cdot 2 \cdot 3$; $11011 = 7 \cdot 11 \cdot 11 \cdot 13$; $11111 = 41 \cdot 271$. (Products denoted by $\prod$ are analogous to sums denoted by $\sum$, as explained in exercise 2.25. If $m = 0$, we consider this to be an empty product, whose value is 1 by definition; that's the way $n = 1$ gets represented by (4.10).) Such a factorization is always possible because if $n > 1$ is not prime it has a divisor n_1 such that $1 < n_1 < n$; thus we can write $n = n_1 \cdot n_2$, and (by induction) we know that n_1 and n_2 can be written as products of primes.

Moreover, the expansion in (4.10) is *unique*: There's only one way to write n as a product of primes in nondecreasing order. This statement is called the Fundamental Theorem of Arithmetic, and it seems so obvious that we might wonder why it needs to be proved. How could there be two different sets of primes with the same product? Well, there can't, but the reason *isn't* simply "by definition of prime numbers." For example, if we consider the set of all real numbers of the form $m + n\sqrt{10}$ when m and n are integers, the product of any two such numbers is again of the same form, and we can call such a number "prime" if it can't be factored in a nontrivial way. The number 6 has two representations, $2 \cdot 3 = (4 + \sqrt{10})(4 - \sqrt{10})$; yet exercise 36 shows that 2, 3, $4 + \sqrt{10}$, and $4 - \sqrt{10}$ are all "prime" in this system.

Therefore we should prove rigorously that (4.10) is unique. There is certainly only one possibility when $n = 1$, since the product must be empty in that case; so let's suppose that $n > 1$ and that all smaller numbers factor uniquely. Suppose we have two factorizations

$$n = p_1 \ldots p_m = q_1 \ldots q_k, \qquad p_1 \leqslant \cdots \leqslant p_m \quad \text{and} \quad q_1 \leqslant \cdots \leqslant q_k,$$

where the p's and q's are all prime. We will prove that $p_1 = q_1$. If not, we can assume that $p_1 < q_1$, making p_1 smaller than all the q's. Since p_1 and q_1 are prime, their gcd must be 1; hence Euclid's self-certifying algorithm gives us integers a and b such that $ap_1 + bq_1 = 1$. Therefore

$$ap_1 q_2 \ldots q_k + bq_1 q_2 \ldots q_k = q_2 \ldots q_k.$$

Now p_1 divides both terms on the left, since $q_1 q_2 \ldots q_k = n$; hence p_1 divides the right-hand side, $q_2 \ldots q_k$. Thus $q_2 \ldots q_k / p_1$ is an integer, and $q_2 \ldots q_k$ has a prime factorization in which p_1 appears. But $q_2 \ldots q_k < n$, so it has a unique factorization (by induction). This contradiction shows that p_1 must be equal to q_1 after all. Therefore we can divide both of n's factorizations by p_1, obtaining $p_2 \ldots p_m = q_2 \ldots q_k < n$. The other factors must likewise be equal (by induction), so our proof of uniqueness is complete.

It's the factorization, not the theorem, that's unique.

Sometimes it's more useful to state the Fundamental Theorem in another way: *Every positive integer can be written uniquely in the form*

$$n \;=\; \prod_p p^{n_p}\,, \qquad \text{where each } n_p \geqslant 0. \tag{4.11}$$

The right-hand side is a product over infinitely many primes; but for any particular n all but a few exponents are zero, so the corresponding factors are 1. Therefore it's really a finite product, just as many "infinite" sums are really finite because their terms are mostly zero.

Formula (4.11) represents n uniquely, so we can think of the sequence $\langle n_2, n_3, n_5, \ldots\rangle$ as a *number system* for positive integers. For example, the prime-exponent representation of 12 is $\langle 2, 1, 0, 0, \ldots\rangle$ and the prime-exponent representation of 18 is $\langle 1, 2, 0, 0, \ldots\rangle$. To multiply two numbers, we simply add their representations. In other words,

$$k \;=\; mn \qquad\Longleftrightarrow\qquad k_p \;=\; m_p + n_p \quad \text{for all } p. \tag{4.12}$$

This implies that

$$m \backslash n \qquad\Longleftrightarrow\qquad m_p \;\leqslant\; n_p \quad \text{for all } p, \tag{4.13}$$

and it follows immediately that

$$k \;=\; \gcd(m,n) \qquad\Longleftrightarrow\qquad k_p \;=\; \min(m_p, n_p) \quad \text{for all } p; \tag{4.14}$$

$$k \;=\; \operatorname{lcm}(m,n) \qquad\Longleftrightarrow\qquad k_p \;=\; \max(m_p, n_p) \quad \text{for all } p. \tag{4.15}$$

For example, since $12 = 2^2 \cdot 3^1$ and $18 = 2^1 \cdot 3^2$, we can get their gcd and lcm by taking the min and max of common exponents:

$$\begin{aligned}\gcd(12,18) \;&=\; 2^{\min(2,1)} \cdot 3^{\min(1,2)} \;=\; 2^1 \cdot 3^1 \;=\; 6\,;\\ \operatorname{lcm}(12,18) \;&=\; 2^{\max(2,1)} \cdot 3^{\max(1,2)} \;=\; 2^2 \cdot 3^2 \;=\; 36\,.\end{aligned}$$

If the prime p divides a product mn then it divides either m or n, perhaps both, because of the unique factorization theorem. But composite numbers do not have this property. For example, the nonprime 4 divides $60 = 6 \cdot 10$, but it divides neither 6 nor 10. The reason is simple: In the factorization $60 = 6 \cdot 10 = (2 \cdot 3)(2 \cdot 5)$, the two prime factors of $4 = 2 \cdot 2$ have been split into two parts, hence 4 divides neither part. But a prime is unsplittable, so it must divide one of the original factors.

4.3 PRIME EXAMPLES

How many primes are there? A lot. In fact, infinitely many. Euclid proved this long ago in his Theorem 9 : 20, as follows. Suppose there were only

finitely many primes, say k of them — $2, 3, 5, \ldots, P_k$. Then, said Euclid, we should consider the number

$$M = 2 \cdot 3 \cdot 5 \cdot \ldots \cdot P_k + 1 .$$

None of the k primes can divide M, because each divides $M - 1$. Thus there must be some other prime that divides M; perhaps M itself is prime. This contradicts our assumption that $2, 3, \ldots, P_k$ are the only primes, so there must indeed be infinitely many.

"Οἱ πρῶτοι ἀριθμοὶ πλείους εἰσὶ παντὸς τοῦ προτεθέντος πλήθους πρώτων ἀριθμῶν."
— Euclid [98]
[Translation: "There are more primes than in any given set of primes."]

Euclid's proof suggests that we define *Euclid numbers* by the recurrence

$$e_n = e_1 e_2 \ldots e_{n-1} + 1 , \qquad \text{when } n \geqslant 1. \tag{4.16}$$

The sequence starts out

$$\begin{aligned} e_1 &= 1 + 1 = 2 ; \\ e_2 &= 2 + 1 = 3 ; \\ e_3 &= 2 \cdot 3 + 1 = 7 ; \\ e_4 &= 2 \cdot 3 \cdot 7 + 1 = 43 ; \end{aligned}$$

these are all prime. But the next case, e_5, is $1807 = 13 \cdot 139$. It turns out that $e_6 = 3263443$ is prime, while

$$\begin{aligned} e_7 &= 547 \cdot 607 \cdot 1033 \cdot 31051 ; \\ e_8 &= 29881 \cdot 67003 \cdot 9119521 \cdot 6212157481 . \end{aligned}$$

It is known that $e_9, \ldots, e_{17}$ are composite, and the remaining e_n are probably composite as well. However, the Euclid numbers are all *relatively prime* to each other; that is,

$$\gcd(e_m, e_n) = 1 , \qquad \text{when } m \neq n.$$

Euclid's algorithm (what else?) tells us this in three short steps, because $e_n \bmod e_m = 1$ when $n > m$:

$$\gcd(e_m, e_n) = \gcd(1, e_m) = \gcd(0, 1) = 1 .$$

Therefore, if we let q_j be the smallest factor of e_j for all $j \geqslant 1$, the primes q_1, q_2, q_3, ... are all different. This is a sequence of infinitely many primes.

Let's pause to consider the Euclid numbers from the standpoint of Chapter 1. Can we express e_n in closed form? Recurrence (4.16) can be simplified by removing the three dots: If $n > 1$ we have

$$e_n = e_1 \ldots e_{n-2} e_{n-1} + 1 = (e_{n-1} - 1) e_{n-1} + 1 = e_{n-1}^2 - e_{n-1} + 1 .$$

Thus e_n has about twice as many decimal digits as e_{n-1}. Exercise 37 proves that there's a constant $E \approx 1.264$ such that

$$e_n = \lfloor E^{2^n} + \tfrac{1}{2} \rfloor . \tag{4.17}$$

And exercise 60 provides a similar formula that gives nothing but primes:

$$p_n = \lfloor P^{3^n} \rfloor , \tag{4.18}$$

for some constant P. But equations like (4.17) and (4.18) cannot really be considered to be in closed form, because the constants E and P are computed from the numbers e_n and p_n in a sort of sneaky way. No independent relation is known (or likely) that would connect them with other constants of mathematical interest.

Indeed, nobody knows *any* useful formula that gives arbitrarily large primes but only primes. Computer scientists at Chevron Geosciences did, however, strike mathematical oil in 1984. Using a program developed by David Slowinski, they discovered the largest prime known at that time,

$$2^{216091} - 1 ,$$

while testing a new Cray X-MP supercomputer. It's easy to compute this number in a few milliseconds on a personal computer, because modern computers work in binary notation and this number is simply $(11\ldots1)_2$. All 216,091 of its bits are '1'. But it's much harder to prove that this number is prime. In fact, just about any computation with it takes a lot of time, because it's so large. For example, even a sophisticated algorithm requires several minutes just to convert $2^{216091} - 1$ to radix 10 on a PC. When printed out, its 65,050 decimal digits require 75 cents U.S. postage to mail first class.

Or probably more, by the time you read this.

Incidentally, $2^{216091} - 1$ is the number of moves necessary to solve the Tower of Hanoi problem when there are 216,091 disks. Numbers of the form

$$2^p - 1$$

(where p is prime, as always in this chapter) are called *Mersenne numbers*, after Father Marin Mersenne who investigated some of their properties in the seventeenth century [269]. . The Mersenne primes known to date occur for $p = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091$, and 756839.

The number $2^n - 1$ can't possibly be prime if n is composite, because $2^{km} - 1$ has $2^m - 1$ as a factor:

$$2^{km} - 1 = (2^m - 1)(2^{m(k-1)} + 2^{m(k-2)} + \cdots + 1) .$$

But $2^p - 1$ isn't always prime when p is prime; $2^{11} - 1 = 2047 = 23 \cdot 89$ is the smallest such nonprime. (Mersenne knew this.)

Factoring and primality testing of large numbers are hot topics nowadays. A summary of what was known up to 1981 appears in Section 4.5.4 of [208], and many new results continue to be discovered. Pages 391–394 of that book explain a special way to test Mersenne numbers for primality.

For most of the last two hundred years, the largest known prime has been a Mersenne prime, although only 31 Mersenne primes are known. Many people are trying to find larger ones, but it's getting tough. So those really interested in fame (if not fortune) and a spot in *The Guinness Book of World Records* might instead try numbers of the form $2^n k + 1$, for small values of k like 3 or 5. These numbers can be tested for primality almost as quickly as Mersenne numbers can; exercise 4.5.4–27 of [208] gives the details.

We haven't fully answered our original question about how many primes there are. There are infinitely many, but some infinite sets are "denser" than others. For instance, among the positive integers there are infinitely many even numbers and infinitely many perfect squares, yet in several important senses there are more even numbers than perfect squares. One such sense looks at the size of the nth value. The nth even integer is $2n$ and the nth perfect square is n^2; since $2n$ is much less than n^2 for large n, the nth even integer occurs much sooner than the nth perfect square, so we can say there are many more even integers than perfect squares. A similar sense looks at the number of values not exceeding x. There are $\lfloor x/2 \rfloor$ such even integers and $\lfloor \sqrt{x} \rfloor$ perfect squares; since $x/2$ is much larger than $\sqrt{x}$ for large x, again we can say there are many more even integers.

Weird. I thought there were the same number of even integers as perfect squares, since there's a one-to-one correspondence between them.

What can we say about the primes in these two senses? It turns out that the nth prime, P_n, is about n times the natural log of n:

$$P_n \sim n \ln n .$$

(The symbol '$\sim$' can be read "is asymptotic to"; it means that the limit of the ratio $P_n / n \ln n$ is 1 as n goes to infinity.) Similarly, for the number of primes $\pi(x)$ not exceeding x we have what's known as the prime number theorem:

$$\pi(x) \sim \frac{x}{\ln x} .$$

Proving these two facts is beyond the scope of this book, although we can show easily that each of them implies the other. In Chapter 9 we will discuss the rates at which functions approach infinity, and we'll see that the function $n \ln n$, our approximation to P_n, lies between $2n$ and n^2 asymptotically. Hence there are fewer primes than even integers, but there are more primes than perfect squares.

These formulas, which hold only in the limit as n or $x \to \infty$, can be replaced by more exact estimates. For example, Rosser and Schoenfeld [312] have established the handy bounds

$$\ln x - \tfrac{3}{2} \;<\; \tfrac{x}{\pi(x)} \;<\; \ln x - \tfrac{1}{2}\,, \qquad \text{for } x \geqslant 67; \tag{4.19}$$

$$n\bigl(\ln n + \ln\ln n - \tfrac{3}{2}\bigr) \;<\; P_n \;<\; n\bigl(\ln n + \ln\ln n - \tfrac{1}{2}\bigr), \quad \text{for } n \geqslant 20. \tag{4.20}$$

If we look at a "random" integer n, the chances of its being prime are about one in $\ln n$. For example, if we look at numbers near 10^{16}, we'll have to examine about $16\ln 10 \approx 36.8$ of them before finding a prime. (It turns out that there are exactly 10 primes between $10^{16} - 370$ and $10^{16} - 1$.) Yet the distribution of primes has many irregularities. For example, all the numbers between $P_1P_2\ldots P_n + 2$ and $P_1P_2\ldots P_n + P_{n+1} - 1$ inclusive are composite. Many examples of "twin primes" p and $p+2$ are known (5 and 7, 11 and 13, 17 and 19, 29 and 31, ..., 9999999999999641 and 9999999999999643, ...), yet nobody knows whether or not there are infinitely many pairs of twin primes. (See Hardy and Wright [181, §1.4 and §2.8].)

One simple way to calculate all $\pi(x)$ primes $\leqslant x$ is to form the so-called sieve of Eratosthenes: First write down all integers from 2 through x. Next circle 2, marking it prime, and cross out all other multiples of 2. Then repeatedly circle the smallest uncircled, uncrossed number and cross out its other multiples. When everything has been circled or crossed out, the circled numbers are the primes. For example when $x = 10$ we write down 2 through 10, circle 2, then cross out its multiples 4, 6, 8, and 10. Next 3 is the smallest uncircled, uncrossed number, so we circle it and cross out 6 and 9. Now 5 is smallest, so we circle it and cross out 10. Finally we circle 7. The circled numbers are 2, 3, 5, and 7; so these are the $\pi(10) = 4$ primes not exceeding 10.

"Je me sers de la notation très simple $n!$ pour désigner le produit de nombres décroissans depuis n jusqu'à l'unité, savoir $n(n-1)(n-2)\ldots.3.2.1$. L'emploi continuel de l'analyse combinatoire que je fais dans la plupart de mes démonstrations, a rendu cette notation indispensable."
— Ch. Kramp [228]

4.4 FACTORIAL FACTORS

Now let's take a look at the factorization of some interesting highly composite numbers, the factorials:

$$n! \;=\; 1\cdot 2\cdot\ldots\cdot n \;=\; \prod_{k=1}^{n} k\,, \qquad \text{integer } n \geqslant 0. \tag{4.21}$$

According to our convention for an empty product, this defines $0!$ to be 1. Thus $n! = (n-1)!\,n$ for every positive integer n. This is the number of permutations of n distinct objects. That is, it's the number of ways to arrange n things in a row: There are n choices for the first thing; for each choice of first thing, there are $n-1$ choices for the second; for each of these $n(n-1)$ choices, there are $n-2$ for the third; and so on, giving $n(n-1)(n-2)\ldots(1)$

arrangements in all. Here are the first few values of the factorial function.

n	0	1	2	3	4	5	6	7	8	9	10
$n!$	1	1	2	6	24	120	720	5040	40320	362880	3628800

It's useful to know a few factorial facts, like the first six or so values, and the fact that 10! is about $3\frac{1}{2}$ million plus change; another interesting fact is that the number of digits in $n!$ exceeds n when $n \geqslant 25$.

We can prove that $n!$ is plenty big by using something like Gauss's trick of Chapter 1:

$$n!^2 \;=\; (1\cdot 2\cdot \ldots \cdot n)(n\cdot \ldots \cdot 2\cdot 1) \;=\; \prod_{k=1}^{n} k(n+1-k)\,.$$

We have $n \leqslant k(n+1-k) \leqslant \frac{1}{4}(n+1)^2$, since the quadratic polynomial $k(n+1-k) = \frac{1}{4}(n+1)^2 - \big(k - \frac{1}{2}(n+1)\big)^2$ has its smallest value at $k=1$ and its largest value at $k = \frac{1}{2}(n+1)$. Therefore

$$\prod_{k=1}^{n} n \;\leqslant\; n!^2 \;\leqslant\; \prod_{k=1}^{n} \frac{(n+1)^2}{4}\,;$$

that is,

$$n^{n/2} \;\leqslant\; n! \;\leqslant\; \frac{(n+1)^n}{2^n}\,. \tag{4.22}$$

This relation tells us that the factorial function grows exponentially!!

To approximate $n!$ more accurately for large n we can use Stirling's formula, which we will derive in Chapter 9:

$$n! \;\sim\; \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\,. \tag{4.23}$$

And a still more precise approximation tells us the asymptotic relative error: Stirling's formula undershoots $n!$ by a factor of about $1/(12n)$. Even for fairly small n this more precise estimate is pretty good. For example, Stirling's approximation (4.23) gives a value near 3598696 when $n = 10$, and this is about $0.83\% \approx 1/120$ too small. Good stuff, asymptotics.

But let's get back to primes. We'd like to determine, for any given prime p, the largest power of p that divides $n!$; that is, we want the exponent of p in $n!$'s unique factorization. We denote this number by $\epsilon_p(n!)$, and we start our investigations with the small case $p = 2$ and $n = 10$. Since 10! is the product of ten numbers, $\epsilon_2(10!)$ can be found by summing the powers-of-2

contributions of those ten numbers; this calculation corresponds to summing the columns of the following array:

	1	2	3	4	5	6	7	8	9	10	powers of 2
divisible by 2		x		x		x		x		x	$5 = \lfloor 10/2 \rfloor$
divisible by 4				x				x			$2 = \lfloor 10/4 \rfloor$
divisible by 8								x			$1 = \lfloor 10/8 \rfloor$
powers of 2	0	1	0	2	0	1	0	3	0	1	8

A powerful ruler.

(The column sums form what's sometimes called the *ruler function* $\rho(k)$, because of their similarity to '⌈''''|''''|''''|''''|''''⌉', the lengths of lines marking fractions of an inch.) The sum of these ten sums is 8; hence 2^8 divides 10! but 2^9 doesn't.

There's also another way: We can sum the contributions of the rows. The first row marks the numbers that contribute a power of 2 (and thus are divisible by 2); there are $\lfloor 10/2 \rfloor = 5$ of them. The second row marks those that contribute an additional power of 2; there are $\lfloor 10/4 \rfloor = 2$ of them. And the third row marks those that contribute yet another; there are $\lfloor 10/8 \rfloor = 1$ of them. These account for all contributions, so we have $\epsilon_2(10!) = 5+2+1 = 8$.

For general n this method gives

$$\epsilon_2(n!) \;=\; \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \left\lfloor \frac{n}{8} \right\rfloor + \cdots \;=\; \sum_{k \geqslant 1} \left\lfloor \frac{n}{2^k} \right\rfloor .$$

This sum is actually finite, since the summand is zero when $2^k > n$. Therefore it has only $\lfloor \lg n \rfloor$ nonzero terms, and it's computationally quite easy. For instance, when $n = 100$ we have

$$\epsilon_2(100!) \;=\; 50 + 25 + 12 + 6 + 3 + 1 \;=\; 97 .$$

Each term is just the floor of half the previous term. This is true for all n, because as a special case of (3.11) we have $\lfloor n/2^{k+1} \rfloor = \lfloor \lfloor n/2^k \rfloor / 2 \rfloor$. It's especially easy to see what's going on here when we write the numbers in binary:

$$\begin{aligned}
100 &= (1100100)_2 = 100 \\
\lfloor 100/2 \rfloor &= (110010)_2 = 50 \\
\lfloor 100/4 \rfloor &= (11001)_2 = 25 \\
\lfloor 100/8 \rfloor &= (1100)_2 = 12 \\
\lfloor 100/16 \rfloor &= (110)_2 = 6 \\
\lfloor 100/32 \rfloor &= (11)_2 = 3 \\
\lfloor 100/64 \rfloor &= (1)_2 = 1
\end{aligned}$$

We merely drop the least significant bit from one term to get the next.

The binary representation also shows us how to derive another formula,

$$\epsilon_2(n!) \;=\; n - \nu_2(n)\,, \tag{4.24}$$

where $\nu_2(n)$ is the number of 1's in the binary representation of n. This simplification works because each 1 that contributes 2^m to the value of n contributes $2^{m-1} + 2^{m-2} + \cdots + 2^0 = 2^m - 1$ to the value of $\epsilon_2(n!)$.

Generalizing our findings to an arbitrary prime p, we have

$$\epsilon_p(n!) \;=\; \left\lfloor \frac{n}{p} \right\rfloor + \left\lfloor \frac{n}{p^2} \right\rfloor + \left\lfloor \frac{n}{p^3} \right\rfloor + \cdots \;=\; \sum_{k \geqslant 1} \left\lfloor \frac{n}{p^k} \right\rfloor \tag{4.25}$$

by the same reasoning as before.

About how large is $\epsilon_p(n!)$? We get an easy (but good) upper bound by simply removing the floor from the summand and then summing an infinite geometric progression:

$$\begin{aligned}
\epsilon_p(n!) \;&<\; \frac{n}{p} + \frac{n}{p^2} + \frac{n}{p^3} + \cdots \\
&=\; \frac{n}{p}\left(1 + \frac{1}{p} + \frac{1}{p^2} + \cdots\right) \\
&=\; \frac{n}{p}\left(\frac{p}{p-1}\right) \\
&=\; \frac{n}{p-1}\,.
\end{aligned}$$

For $p = 2$ and $n = 100$ this inequality says that $97 < 100$. Thus the upper bound 100 is not only correct, it's also close to the true value 97. In fact, the true value $n - \nu_2(n)$ is $\sim n$ in general, because $\nu_2(n) \leqslant \lceil \lg n \rceil$ is asymptotically much smaller than n.

When $p = 2$ and 3 our formulas give $\epsilon_2(n!) \sim n$ and $\epsilon_3(n!) \sim n/2$, so it seems reasonable that every once in awhile $\epsilon_3(n!)$ should be exactly half as big as $\epsilon_2(n!)$. For example, this happens when $n = 6$ and $n = 7$, because $6! = 2^4 \cdot 3^2 \cdot 5 = 7!/7$. But nobody has yet proved that such coincidences happen infinitely often.

The bound on $\epsilon_p(n!)$ in turn gives us a bound on $p^{\epsilon_p(n!)}$, which is p's contribution to $n!$:

$$p^{\epsilon_p(n!)} \;<\; p^{n/(p-1)}\,.$$

And we can simplify this formula (at the risk of greatly loosening the upper bound) by noting that $p \leqslant 2^{p-1}$; hence $p^{n/(p-1)} \leqslant (2^{p-1})^{n/(p-1)} = 2^n$. In other words, the contribution that any prime makes to $n!$ is less than 2^n.

We can use this observation to get another proof that there are infinitely many primes. For if there were only the k primes $2, 3, \ldots, P_k$, then we'd have $n! < (2^n)^k = 2^{nk}$ for all $n > 1$, since each prime can contribute at most a factor of $2^n - 1$. But we can easily contradict the inequality $n! < 2^{nk}$ by choosing n large enough, say $n = 2^{2k}$. Then

$$n! \;<\; 2^{nk} \;=\; 2^{2^{2k}k} \;=\; n^{n/2}\,,$$

contradicting the inequality $n! \geqslant n^{n/2}$ that we derived in (4.22). There are infinitely many primes, still.

We can even beef up this argument to get a crude bound on $\pi(n)$, the number of primes not exceeding n. Every such prime contributes a factor of less than 2^n to $n!$; so, as before,

$$n! \;<\; 2^{n\pi(n)}\,.$$

If we replace $n!$ here by Stirling's approximation (4.23), which is a lower bound, and take logarithms, we get

$$n\pi(n) \;>\; n\lg(n/e) + \tfrac{1}{2}\lg(2\pi n)\,;$$

hence

$$\pi(n) \;>\; \lg(n/e)\,.$$

This lower bound is quite weak, compared with the actual value $\pi(n) \sim n/\ln n$, because $\log n$ is much smaller than $n/\log n$ when n is large. But we didn't have to work very hard to get it, and a bound is a bound.

4.5 RELATIVE PRIMALITY

When $\gcd(m, n) = 1$, the integers m and n have no prime factors in common and we say that they're *relatively prime*.

This concept is so important in practice, we ought to have a special notation for it; but alas, number theorists haven't agreed on a very good one yet. Therefore we cry: HEAR US, O MATHEMATICIANS OF THE WORLD! LET US NOT WAIT ANY LONGER! WE CAN MAKE MANY FORMULAS CLEARER BY ADOPTING A NEW NOTATION NOW! LET US AGREE TO WRITE '$m \perp n$', AND TO SAY "m IS PRIME TO n," IF m AND n ARE RELATIVELY PRIME. In other words, let us declare that

Like perpendicular lines don't have a common direction, perpendicular numbers don't have common factors.

$$m \perp n \qquad \Longleftrightarrow \qquad m, n \text{ are integers and } \gcd(m, n) = 1. \tag{4.26}$$

A fraction m/n is in lowest terms if and only if $m \perp n$. Since we reduce fractions to lowest terms by casting out the largest common factor of numerator and denominator, we suspect that, in general,

$$m/\gcd(m,n) \quad \perp \quad n/\gcd(m,n)\,; \tag{4.27}$$

and indeed this is true. It follows from a more general law, $\gcd(km,kn) = k\gcd(m,n)$, proved in exercise 14.

The $\perp$ relation has a simple formulation when we work with the prime-exponent representations of numbers, because of the gcd rule (4.14):

$$m \perp n \qquad \Longleftrightarrow \qquad \min(m_p, n_p) = 0 \quad \text{for all } p. \tag{4.28}$$

Furthermore, since m_p and n_p are nonnegative, we can rewrite this as

The dot product is zero, like orthogonal vectors.

$$m \perp n \qquad \Longleftrightarrow \qquad m_p n_p = 0 \quad \text{for all } p. \tag{4.29}$$

And now we can prove an important law by which we can split and combine two $\perp$ relations with the same left-hand side:

$$k \perp m \quad \text{and} \quad k \perp n \qquad \Longleftrightarrow \qquad k \perp mn\,. \tag{4.30}$$

In view of (4.29), this law is another way of saying that $k_p m_p = 0$ and $k_p n_p = 0$ if and only if $k_p(m_p + n_p) = 0$, when m_p and n_p are nonnegative.

There's a beautiful way to construct the set of all nonnegative fractions m/n with $m \perp n$, called the *Stern–Brocot tree* because it was discovered independently by Moriz Stern [339], a German mathematician, and Achille Brocot [40], a French clockmaker. The idea is to start with the two fractions $(\frac{0}{1}, \frac{1}{0})$ and then to repeat the following operation as many times as desired:

Interesting how mathematicians will say "discovered" when absolutely anyone else would have said "invented."

$$\text{Insert } \frac{m+m'}{n+n'} \text{ between two adjacent fractions } \frac{m}{n} \text{ and } \frac{m'}{n'}\,.$$

The new fraction $(m+m')/(n+n')$ is called the *mediant* of m/n and m'/n'. For example, the first step gives us one new entry between $\frac{0}{1}$ and $\frac{1}{0}$,

$$\tfrac{0}{1},\ \tfrac{1}{1},\ \tfrac{1}{0}\,;$$

and the next gives two more:

$$\tfrac{0}{1},\ \tfrac{1}{2},\ \tfrac{1}{1},\ \tfrac{2}{1},\ \tfrac{1}{0}\,.$$

The next gives four more,

$$\tfrac{0}{1},\ \tfrac{1}{3},\ \tfrac{1}{2},\ \tfrac{2}{3},\ \tfrac{1}{1},\ \tfrac{3}{2},\ \tfrac{2}{1},\ \tfrac{3}{1},\ \tfrac{1}{0}\,;$$

I guess $1/0$ *is infinity, "in lowest terms."*

and then we'll get 8, 16, and so on. The entire array can be regarded as an infinite binary tree structure whose top levels look like this:

Each fraction is $\frac{m+m'}{n+n'}$, where $\frac{m}{n}$ is the nearest ancestor above and to the left, and $\frac{m'}{n'}$ is the nearest ancestor above and to the right. (An "ancestor" is a fraction that's reachable by following the branches upward.) Many patterns can be observed in this tree.

Conserve parody.

Why does this construction work? Why, for example, does each mediant fraction $(m+m')/(n+n')$ turn out to be in lowest terms when it appears in this tree? (If m, m', n, and n' were all odd, we'd get even/even; somehow the construction guarantees that fractions with odd numerators and denominators never appear next to each other.) And why do all possible fractions m/n occur exactly once? Why can't a particular fraction occur twice, or not at all?

All of these questions have amazingly simple answers, based on the following fundamental fact: *If* m/n *and* m'/n' *are consecutive fractions at any stage of the construction, we have*

$$m'n - mn' \;=\; 1\,. \tag{4.31}$$

This relation is true initially $(1\cdot 1 - 0\cdot 0 = 1)$; and when we insert a new mediant $(m+m')/(n+n')$, the new cases that need to be checked are

$$\begin{aligned}(m+m')n - m(n+n') \;&=\; 1\,;\\ m'(n+n') - (m+m')n' \;&=\; 1\,.\end{aligned}$$

Both of these equations are equivalent to the original condition (4.31) that they replace. Therefore (4.31) is invariant at all stages of the construction.

Furthermore, if $m/n < m'/n'$ and if all values are nonnegative, it's easy to verify that

$$m/n \;<\; (m+m')/(n+n') \;<\; m'/n'\,.$$

A mediant fraction isn't halfway between its progenitors, but it does lie somewhere in between. Therefore the construction preserves order, and we couldn't possibly get the same fraction in two different places.

True, but if you get a compound fracture you'd better go see a doctor.

One question still remains. Can any positive fraction a/b with $a \perp b$ possibly be omitted? The answer is no, because we can confine the construction to the immediate neighborhood of a/b, and in this region the behavior is easy to analyze: Initially we have

$$\tfrac{m}{n} = \tfrac{0}{1} < \left(\tfrac{a}{b}\right) < \tfrac{1}{0} = \tfrac{m'}{n'},$$

where we put parentheses around $\frac{a}{b}$ to indicate that it's not really present yet. Then if at some stage we have

$$\tfrac{m}{n} < \left(\tfrac{a}{b}\right) < \tfrac{m'}{n'},$$

the construction forms $(m+m')/(n+n')$ and there are three cases. Either $(m+m')/(n+n') = a/b$ and we win; or $(m+m')/(n+n') < a/b$ and we can set $m \leftarrow m+m'$, $n \leftarrow n+n'$; or $(m+m')/(n+n') > a/b$ and we can set $m' \leftarrow m+m'$, $n' \leftarrow n+n'$. This process cannot go on indefinitely, because the conditions

$$\tfrac{a}{b} - \tfrac{m}{n} > 0 \qquad \text{and} \qquad \tfrac{m'}{n'} - \tfrac{a}{b} > 0$$

imply that

$$an - bm \geqslant 1 \qquad \text{and} \qquad bm' - an' \geqslant 1;$$

hence

$$(m'+n')(an-bm) + (m+n)(bm'-an') \geqslant m'+n'+m+n;$$

and this is the same as $a+b \geqslant m'+n'+m+n$ by (4.31). Either m or n or m' or n' increases at each step, so we must win after at most $a+b$ steps.

The *Farey series* of order N, denoted by $\mathcal{F}_N$, is the set of all reduced fractions between 0 and 1 whose denominators are N or less, arranged in increasing order. For example, if $N = 6$ we have

$$\mathcal{F}_6 = \tfrac{0}{1}, \tfrac{1}{6}, \tfrac{1}{5}, \tfrac{1}{4}, \tfrac{1}{3}, \tfrac{2}{5}, \tfrac{1}{2}, \tfrac{3}{5}, \tfrac{2}{3}, \tfrac{3}{4}, \tfrac{4}{5}, \tfrac{5}{6}, \tfrac{1}{1}.$$

We can obtain $\mathcal{F}_N$ in general by starting with $\mathcal{F}_1 = \frac{0}{1}, \frac{1}{1}$ and then inserting mediants whenever it's possible to do so without getting a denominator that is too large. We don't miss any fractions in this way, because we know that the Stern–Brocot construction doesn't miss any, and because a mediant with denominator $\leqslant N$ is never formed from a fraction whose denominator is $> N$. (In other words, $\mathcal{F}_N$ defines a *subtree* of the Stern–Brocot tree, obtained by

pruning off unwanted branches.) It follows that $m'n - mn' = 1$ whenever m/n and m'/n' are consecutive elements of a Farey series.

This method of construction reveals that $\mathcal{F}_N$ can be obtained in a simple way from $\mathcal{F}_{N-1}$: We simply insert the fraction $(m + m')/N$ between consecutive fractions m/n, m'/n' of $\mathcal{F}_{N-1}$ whose denominators sum to N. For example, it's easy to obtain $\mathcal{F}_7$ from the elements of $\mathcal{F}_6$, by inserting $\frac{1}{7}$, $\frac{2}{7}$, $\ldots$, $\frac{6}{7}$ according to the stated rule:

$$\mathcal{F}_7 \;=\; \tfrac{0}{1}, \tfrac{1}{7}, \tfrac{1}{6}, \tfrac{1}{5}, \tfrac{1}{4}, \tfrac{2}{7}, \tfrac{1}{3}, \tfrac{2}{5}, \tfrac{3}{7}, \tfrac{1}{2}, \tfrac{4}{7}, \tfrac{3}{5}, \tfrac{2}{3}, \tfrac{5}{7}, \tfrac{3}{4}, \tfrac{4}{5}, \tfrac{5}{6}, \tfrac{6}{7}, \tfrac{1}{1}\,.$$

When N is prime, $N - 1$ new fractions will appear; but otherwise we'll have fewer than $N - 1$, because this process generates only numerators that are relatively prime to N.

Long ago in (4.5) we proved — in different words — that whenever $m \perp n$ and $0 < m \leqslant n$ we can find integers a and b such that

$$ma - nb \;=\; 1\,. \tag{4.32}$$

(Actually we said $m'm + n'n = \gcd(m, n)$, but we can write 1 for $\gcd(m, n)$, a for m', and b for $-n'$.) The Farey series gives us another proof of (4.32), because we can let b/a be the fraction that precedes m/n in $\mathcal{F}_n$. Thus (4.5) is just (4.31) again. For example, one solution to $3a - 7b = 1$ is $a = 5$, $b = 2$, since $\frac{2}{5}$ precedes $\frac{3}{7}$ in $\mathcal{F}_7$. This construction implies that we can always find a solution to (4.32) with $0 \leqslant b < a < n$, if $0 < m \leqslant n$. Similarly, if $0 \leqslant n < m$ and $m \perp n$, we can solve (4.32) with $0 < a \leqslant b \leqslant m$ by letting a/b be the fraction that *follows* n/m in $\mathcal{F}_m$.

Sequences of three consecutive terms in a Farey series have an amazing property that is proved in exercise 61. But we had better not discuss the Farey series any further, because the entire Stern–Brocot tree turns out to be even more interesting.

Farey 'nough.

We can, in fact, regard the Stern–Brocot tree as a *number system* for representing rational numbers, because each positive, reduced fraction occurs exactly once. Let's use the letters L and R to stand for going down to the left or right branch as we proceed from the root of the tree to a particular fraction; then a string of L's and R's uniquely identifies a place in the tree. For example, LRRL means that we go left from $\frac{1}{1}$ down to $\frac{1}{2}$, then right to $\frac{2}{3}$, then right to $\frac{3}{4}$, then left to $\frac{5}{7}$. We can consider LRRL to be a representation of $\frac{5}{7}$. Every positive fraction gets represented in this way as a unique string of L's and R's.

Well, actually there's a slight problem: The fraction $\frac{1}{1}$ corresponds to the *empty* string, and we need a notation for that. Let's agree to call it I, because that looks something like 1 and it stands for "identity."

This representation raises two natural questions: (1) Given positive integers m and n with $m \perp n$, what is the string of L's and R's that corresponds to m/n? (2) Given a string of L's and R's, what fraction corresponds to it? Question 2 seems easier, so let's work on it first. We define

$$f(S) \;=\; \text{fraction corresponding to } S$$

when S is a string of L's and R's. For example, $f(\mathrm{LRRL}) = \frac{5}{7}$.

According to the construction, $f(S) = (m+m')/(n+n')$ if m/n and m'/n' are the closest fractions preceding and following S in the upper levels of the tree. Initially $m/n = 0/1$ and $m'/n' = 1/0$; then we successively replace either m/n or m'/n' by the mediant $(m+m')/(n+n')$ as we move right or left in the tree, respectively.

How can we capture this behavior in mathematical formulas that are easy to deal with? A bit of experimentation suggests that the best way is to maintain a 2×2 matrix

$$M(S) \;=\; \begin{pmatrix} n & n' \\ m & m' \end{pmatrix}$$

that holds the four quantities involved in the ancestral fractions m/n and m'/n' enclosing $f(S)$. We could put the m's on top and the n's on the bottom, fractionwise; but this upside-down arrangement works out more nicely because we have $M(I) = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ when the process starts, and $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ is traditionally called the identity matrix I.

A step to the left replaces n' by $n+n'$ and m' by $m+m'$; hence

$$M(SL) \;=\; \begin{pmatrix} n & n+n' \\ m & m+m' \end{pmatrix} \;=\; \begin{pmatrix} n & n' \\ m & m' \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \;=\; M(S)\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

(This is a special case of the general rule

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} w & x \\ y & z \end{pmatrix} \;=\; \begin{pmatrix} aw+by & ax+bz \\ cw+dy & cx+dz \end{pmatrix}$$

for multiplying 2×2 matrices.) Similarly it turns out that

If you're clueless about matrices, don't panic; this book uses them only here.

$$M(SR) \;=\; \begin{pmatrix} n+n' & n' \\ m+m' & m' \end{pmatrix} \;=\; M(S)\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

Therefore if we define L and R as 2×2 matrices,

$$L \;=\; \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \qquad R \;=\; \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \tag{4·33}$$

we get the simple formula $M(S) = S$, by induction on the length of S. Isn't that nice? (The letters L and R serve dual roles, as matrices and as letters in the string representation.) For example,

$$M(\mathrm{LRRL}) = \mathrm{LRRL} = \left(\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right)\left(\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right)\left(\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right)\left(\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right) = \left(\begin{smallmatrix}2&1\\1&1\end{smallmatrix}\right)\left(\begin{smallmatrix}1&1\\1&2\end{smallmatrix}\right) = \left(\begin{smallmatrix}3&4\\2&3\end{smallmatrix}\right);$$

the ancestral fractions that enclose $\mathrm{LRRL} = \frac{5}{7}$ are $\frac{2}{3}$ and $\frac{3}{4}$. And this construction gives us the answer to Question 2:

$$f(S) = f\left(\begin{pmatrix} n & n' \\ m & m' \end{pmatrix}\right) = \frac{m+m'}{n+n'}. \tag{4·34}$$

How about Question 1? That's easy, now that we understand the fundamental connection between tree nodes and 2×2 matrices. Given a pair of positive integers m and n, with $m \perp n$, we can find the position of m/n in the Stern–Brocot tree by "binary search" as follows:

$S := I;$
while $m/n \neq f(S)$ **do**
  if $m/n < f(S)$ **then** $\big(\mathrm{output}(L);\ S := SL\big)$
    else $\big(\mathrm{output}(R);\ S := SR\big)$.

This outputs the desired string of L's and R's.

There's also another way to do the same job, by changing m and n instead of maintaining the state S. If S is any 2×2 matrix, we have

$$f(RS) = f(S) + 1$$

because RS is like S but with the top row added to the bottom row. (Let's look at it in slow motion:

$$S = \begin{pmatrix} n & n' \\ m & m' \end{pmatrix}; \qquad RS = \begin{pmatrix} n & n' \\ m+n & m'+n' \end{pmatrix};$$

hence $f(S) = (m+m')/(n+n')$ and $f(RS) = \big((m+n)+(m'+n')\big)/(n+n')$.) If we carry out the binary search algorithm on a fraction m/n with $m > n$, the first output will be R; hence the subsequent behavior of the algorithm will have $f(S)$ exactly 1 greater than if we had begun with $(m-n)/n$ instead of m/n. A similar property holds for L, and we have

$$\frac{m}{n} = f(RS) \iff \frac{m-n}{n} = f(S), \qquad \text{when } m > n;$$
$$\frac{m}{n} = f(LS) \iff \frac{m}{n-m} = f(S), \qquad \text{when } m < n.$$

This means that we can transform the binary search algorithm to the following matrix-free procedure:

$$\begin{array}{l} \textbf{while}\ \ m \neq n\ \ \textbf{do} \\ \quad \textbf{if}\ \ m < n\ \ \textbf{then}\ \big(\text{output}(L);\ \ n := n - m\big) \\ \qquad\qquad\ \ \textbf{else}\ \ \big(\text{output}(R);\ \ m := m - n\big)\,. \end{array}$$

For example, given $m/n = 5/7$, we have successively

$$\begin{array}{rccccc} m = & 5 & 5 & 3 & 1 & 1 \\ n = & 7 & 2 & 2 & 2 & 1 \\ \text{output} & & L & R & R & L \end{array}$$

in the simplified algorithm.

Irrational numbers don't appear in the Stern–Brocot tree, but all the rational numbers that are "close" to them do. For example, if we try the binary search algorithm with the number $e = 2.71828\ldots$, instead of with a fraction m/n, we'll get an infinite string of L's and R's that begins

$$\mathrm{RRLRRLRLLLLRLRRRRRRLRLLLLLLLLRLR}\ \ldots\,.$$

We can consider this infinite string to be the representation of e in the Stern–Brocot number system, just as we can represent e as an infinite decimal $2.718281828459\ldots$ or as an infinite binary fraction $(10.101101111110\ldots)_2$. Incidentally, it turns out that e's representation has a regular pattern in the Stern–Brocot system:

$$e \;=\; \mathrm{RL}^0\mathrm{RLR}^2\mathrm{LRL}^4\mathrm{RLR}^6\mathrm{LRL}^8\mathrm{RLR}^{10}\mathrm{LRL}^{12}\mathrm{RL}\ \ldots\,;$$

this is equivalent to a special case of something that Euler [105] discovered when he was 24 years old.

From this representation we can deduce that the fractions

$$\begin{array}{ccccccccccccccccccc} & \mathrm{R} & \mathrm{R} & \mathrm{L} & \mathrm{R} & \mathrm{R} & \mathrm{L} & \mathrm{R} & \mathrm{L} & \mathrm{L} & \mathrm{L} & \mathrm{L} & \mathrm{R} & \mathrm{L} & \mathrm{R} & \mathrm{R} & \mathrm{R} & \mathrm{R} & \\ \frac{1}{1}, & \frac{2}{1}, & \frac{3}{1}, & \frac{5}{2}, & \frac{8}{3}, & \frac{11}{4}, & \frac{19}{7}, & \frac{30}{11}, & \frac{49}{18}, & \frac{68}{25}, & \frac{87}{32}, & \frac{106}{39}, & \frac{193}{71}, & \frac{299}{110}, & \frac{492}{181}, & \frac{685}{252}, & \frac{878}{323}, & \ldots \end{array}$$

are the simplest rational upper and lower approximations to e. For if m/n does not appear in this list, then some fraction in this list whose numerator is $\leqslant m$ and whose denominator is $\leqslant n$ lies between m/n and e. For example, $\frac{27}{10}$ is not as simple an approximation as $\frac{19}{7} = 2.714\ldots$, which appears in the list and is closer to e. We can see this because the Stern–Brocot tree not only includes all rationals, it includes them in order, and because all fractions with small numerator and denominator appear above all less simple ones. Thus, $\frac{27}{10} = \mathrm{RRLRRLL}$ is less than $\frac{19}{7} = \mathrm{RRLRRL}$, which is less than

$e = \mathrm{RRLRRLR}\ldots$. Excellent approximations can be found in this way. For example, $\frac{1264}{465} \approx 2.718280$ agrees with e to six decimal places; we obtained this fraction from the first 19 letters of e's Stern–Brocot representation, and the accuracy is about what we would get with 19 bits of e's binary representation.

We can find the infinite representation of an irrational number α by a simple modification of the matrix-free binary search procedure:

$$\begin{aligned} &\textbf{if } \alpha < 1 \textbf{ then } \big(\mathrm{output}(L);\ \ \alpha := \alpha/(1-\alpha)\big) \\ &\qquad\qquad \textbf{else } \ \big(\mathrm{output}(R);\ \ \alpha := \alpha - 1\big)\,. \end{aligned}$$

(These steps are to be repeated infinitely many times, or until we get tired.) If α is rational, the infinite representation obtained in this way is the same as before but with RL^∞ appended at the right of α's (finite) representation. For example, if $\alpha = 1$, we get $\mathrm{RLLL}\ldots$, corresponding to the infinite sequence of fractions $\frac{1}{1}, \frac{2}{1}, \frac{3}{2}, \frac{4}{3}, \frac{5}{4}, \ldots$, which approach 1 in the limit. This situation is exactly analogous to ordinary binary notation, if we think of L as 0 and R as 1: Just as every real number x in $[0\,..\,1)$ has an infinite binary representation $(.b_1b_2b_3\ldots)_2$ not ending with all 1's, every real number α in $[0\,..\,\infty)$ has an infinite Stern–Brocot representation $B_1B_2B_3\ldots$ not ending with all R's. Thus we have a one-to-one order-preserving correspondence between $[0\,..\,1)$ and $[0\,..\,\infty)$ if we let $0 \leftrightarrow L$ and $1 \leftrightarrow R$.

There's an intimate relationship between Euclid's algorithm and the Stern–Brocot representations of rationals. Given $\alpha = m/n$, we get $\lfloor m/n \rfloor$ R's, then $\lfloor n/(m \bmod n) \rfloor$ L's, then $\lfloor (m \bmod n)/\big(n \bmod (m \bmod n)\big) \rfloor$ R's, and so on. These numbers $m \bmod n$, $n \bmod (m \bmod n)$, ... are just the values examined in Euclid's algorithm. (A little fudging is needed at the end to make sure that there aren't infinitely many R's.) We will explore this relationship further in Chapter 6.

4.6 'MOD': THE CONGRUENCE RELATION

"Numerorum congruentiam hoc signo, $\equiv$, in posterum denotabimus, modulum ubi opus erit in clausulis adiungentes, $-16 \equiv 9 \pmod{5}$, $-7 \equiv 15 \pmod{11}$."
— C.F. Gauss [142]

Modular arithmetic is one of the main tools provided by number theory. We got a glimpse of it in Chapter 3 when we used the binary operation 'mod', usually as one operation amidst others in an expression. In this chapter we will use 'mod' also with entire equations, for which a slightly different notation is more convenient:

$$a \equiv b \pmod{m} \qquad \Longleftrightarrow \qquad a \bmod m = b \bmod m\,. \tag{4.35}$$

For example, $9 \equiv -16 \pmod{5}$, because $9 \bmod 5 = 4 = (-16) \bmod 5$. The formula '$a \equiv b \pmod{m}$' can be read "$a$ is congruent to b modulo m." The definition makes sense when a, b, and m are arbitrary real numbers, but we almost always use it with integers only.

Since $x \bmod m$ differs from x by a multiple of m, we can understand congruences in another way:

$$a \equiv b \pmod{m} \qquad \Longleftrightarrow \qquad a - b \text{ is a multiple of } m. \tag{4.36}$$

For if $a \bmod m = b \bmod m$, then the definition of 'mod' in (3.21) tells us that $a - b = a \bmod m + km - (b \bmod m + lm) = (k - l)m$ for some integers k and l. Conversely if $a - b = km$, then $a = b$ if $m = 0$; otherwise

$$\begin{aligned} a \bmod m \;=\; a - \lfloor a/m \rfloor m \;&=\; b + km - \lfloor (b + km)/m \rfloor m \\ &=\; b - \lfloor b/m \rfloor m \;=\; b \bmod m\,. \end{aligned}$$

The characterization of $\equiv$ in (4.36) is often easier to apply than (4.35). For example, we have $8 \equiv 23 \pmod{5}$ because $8 - 23 = -15$ is a multiple of 5; we don't have to compute both $8 \bmod 5$ and $23 \bmod 5$.

The congruence sign '$\equiv$' looks conveniently like '$=$', because congruences are almost like equations. For example, congruence is an *equivalence relation*; that is, it satisfies the reflexive law '$a \equiv a$', the symmetric law '$a \equiv b \;\Rightarrow\; b \equiv a$', and the transitive law '$a \equiv b \equiv c \;\Rightarrow\; a \equiv c$'. All these properties are easy to prove, because any relation '$\equiv$' that satisfies '$a \equiv b \iff f(a) = f(b)$' for some function f is an equivalence relation. (In our case, $f(x) = x \bmod m$.) Moreover, we can add and subtract congruent elements without losing congruence:

"I feel fine today modulo a slight headache."
—The Hacker's Dictionary [337]

$$\begin{aligned} a \equiv b \quad\text{and}\quad c \equiv d \qquad &\Longrightarrow \qquad a + c \equiv b + d \qquad \pmod{m}\,; \\ a \equiv b \quad\text{and}\quad c \equiv d \qquad &\Longrightarrow \qquad a - c \equiv b - d \qquad \pmod{m}\,. \end{aligned}$$

For if $a - b$ and $c - d$ are both multiples of m, so are $(a + c) - (b + d) = (a - b) + (c - d)$ and $(a - c) - (b - d) = (a - b) - (c - d)$. Incidentally, it isn't necessary to write '$\pmod{m}$' once for every appearance of '$\equiv$'; if the modulus is constant, we need to name it only once in order to establish the context. This is one of the great conveniences of congruence notation.

Multiplication works too, provided that we are dealing with integers:

$$a \equiv b \quad\text{and}\quad c \equiv d \qquad \Longrightarrow \qquad ac \equiv bd \qquad \pmod{m}\,, \quad \text{integers } b, c.$$

Proof: $ac - bd = (a - b)c + b(c - d)$. Repeated application of this multiplication property now allows us to take powers:

$$a \equiv b \qquad \Longrightarrow \qquad a^n \equiv b^n \qquad \pmod{m}\,, \qquad \text{integers } a, b;\ \text{integer } n \geqslant 0.$$

For example, since $2 \equiv -1 \pmod 3$, we have $2^n \equiv (-1)^n \pmod 3$; this means that $2^n - 1$ is a multiple of 3 if and only if n is even.

Thus, most of the algebraic operations that we customarily do with equations can also be done with congruences. Most, but not all. The operation of division is conspicuously absent. If $ad \equiv bd \pmod m$, we can't always conclude that $a \equiv b$. For example, $3 \cdot 2 \equiv 5 \cdot 2 \pmod 4$, but $3 \not\equiv 5$.

We can salvage the cancellation property for congruences, however, in the common case that d and m are relatively prime:

$$ad \equiv bd \iff a \equiv b \pmod m, \quad \text{integers } a, b, d, m \text{ and } d \perp m. \tag{4.37}$$

For example, it's legit to conclude from $15 \equiv 35 \pmod m$ that $3 \equiv 7 \pmod m$, unless the modulus m is a multiple of 5.

To prove this property, we use the extended gcd law (4.5) again, finding d' and m' such that $d'd + m'm = 1$. Then if $ad \equiv bd$ we can multiply both sides of the congruence by d', obtaining $ad'd \equiv bd'd$. Since $d'd \equiv 1$, we have $ad'd \equiv a$ and $bd'd \equiv b$; hence $a \equiv b$. This proof shows that the number d' acts almost like $1/d$ when congruences are considered $\pmod m$; therefore we call it the "inverse of d modulo m."

Another way to apply division to congruences is to divide the modulus as well as the other numbers:

$$ad \equiv bd \pmod{md} \iff a \equiv b \pmod m, \quad \text{for } d \neq 0. \tag{4.38}$$

This law holds for all real a, b, d, and m, because it depends only on the distributive law $(a \bmod m)d = ad \bmod md$: We have $a \bmod m = b \bmod m \iff (a \bmod m)d = (b \bmod m)d \iff ad \bmod md = bd \bmod md$. Thus, for example, from $3 \cdot 2 \equiv 5 \cdot 2 \pmod 4$ we conclude that $3 \equiv 5 \pmod 2$.

We can combine (4.37) and (4.38) to get a general law that changes the modulus as little as possible:

$$\begin{aligned} &ad \equiv bd \pmod m \\ &\quad \iff a \equiv b \left(\bmod \frac{m}{\gcd(d,m)}\right), \quad \text{integers } a, b, d, m. \end{aligned} \tag{4.39}$$

For we can multiply $ad \equiv bd$ by d', where $d'd + m'm = \gcd(d,m)$; this gives the congruence $a \cdot \gcd(d,m) \equiv b \cdot \gcd(d,m) \pmod m$, which can be divided by $\gcd(d,m)$.

Let's look a bit further into this idea of changing the modulus. If we know that $a \equiv b \pmod{100}$, then we also must have $a \equiv b \pmod{10}$, or modulo any divisor of 100. It's stronger to say that $a - b$ is a multiple of 100

than to say that it's a multiple of 10. In general,

$$a \equiv b \pmod{md} \quad \Longrightarrow \quad a \equiv b \pmod{m}, \quad \text{integer } d, \tag{4.40}$$

because any multiple of md is a multiple of m.

Conversely, if we know that $a \equiv b$ with respect to two small moduli, can we conclude that $a \equiv b$ with respect to a larger one? Yes; the rule is *Modulitos?*

$$\begin{aligned} &a \equiv b \pmod{m} \quad \text{and} \quad a \equiv b \pmod{n} \\ &\qquad \Longleftrightarrow \quad a \equiv b \bigl(\bmod \operatorname{lcm}(m,n)\bigr), \qquad \text{integers } m,n > 0. \end{aligned} \tag{4.41}$$

For example, if we know that $a \equiv b$ modulo 12 and 18, we can safely conclude that $a \equiv b \pmod{36}$. The reason is that if $a - b$ is a common multiple of m and n, it is a multiple of $\operatorname{lcm}(m,n)$. This follows from the principle of unique factorization.

The special case $m \perp n$ of this law is extremely important, because $\operatorname{lcm}(m,n) = mn$ when m and n are relatively prime. Therefore we will state it explicitly:

$$\begin{aligned} &a \equiv b \pmod{mn} \\ &\quad \Longleftrightarrow \quad a \equiv b \pmod{m} \ \text{ and } \ a \equiv b \pmod{n}, \quad \text{if } m \perp n. \end{aligned} \tag{4.42}$$

For example, $a \equiv b \pmod{100}$ if and only if $a \equiv b \pmod{25}$ and $a \equiv b \pmod{4}$. Saying this another way, if we know $x \bmod 25$ and $x \bmod 4$, then we have enough facts to determine $x \bmod 100$. This is a special case of the *Chinese Remainder Theorem* (see exercise 30), so called because it was discovered by Sun Tsŭ in China, about A.D. 350.

The moduli m and n in (4.42) can be further decomposed into relatively prime factors until every distinct prime has been isolated. Therefore

$$a \equiv b \pmod{m} \qquad \Longleftrightarrow \qquad a \equiv b \pmod{p^{m_p}} \ \text{ for all } p,$$

if the prime factorization (4.11) of m is $\prod_p p^{m_p}$. Congruences modulo powers of primes are the building blocks for all congruences modulo integers.

4.7 INDEPENDENT RESIDUES

One of the important applications of congruences is a *residue number system*, in which an integer x is represented as a sequence of residues (or remainders) with respect to moduli that are prime to each other:

$$\operatorname{Res}(x) \;=\; (x \bmod m_1, \ldots, x \bmod m_r), \qquad \text{if } m_j \perp m_k \text{ for } 1 \leqslant j < k \leqslant r.$$

Knowing $x \bmod m_1$, ..., $x \bmod m_r$ doesn't tell us everything about x. But it does allow us to determine $x \bmod m$, where m is the product $m_1 \ldots m_r$.

In practical applications we'll often know that x lies in a certain range; then we'll know everything about x if we know $x \bmod m$ and if m is large enough.

For example, let's look at a small case of a residue number system that has only two moduli, 3 and 5:

$x \bmod 15$	$x \bmod 3$	$x \bmod 5$
0	0	0
1	1	1
2	2	2
3	0	3
4	1	4
5	2	0
6	0	1
7	1	2
8	2	3
9	0	4
10	1	0
11	2	1
12	0	2
13	1	3
14	2	4

Each ordered pair $(x \bmod 3, x \bmod 5)$ is different, because $x \bmod 3 = y \bmod 3$ and $x \bmod 5 = y \bmod 5$ if and only if $x \bmod 15 = y \bmod 15$.

We can perform addition, subtraction, and multiplication on the two components *independently*, because of the rules of congruences. For example, if we want to multiply $7 = (1,2)$ by $13 = (1,3)$ modulo 15, we calculate $1{\cdot}1 \bmod 3 = 1$ and $2{\cdot}3 \bmod 5 = 1$. The answer is $(1,1) = 1$; hence $7{\cdot}13 \bmod 15$ must equal 1. Sure enough, it does.

This independence principle is useful in computer applications, because different components can be worked on separately (for example, by different computers). If each modulus m_k is a distinct prime p_k, chosen to be slightly less than 2^{31}, then a computer whose basic arithmetic operations handle integers in the range $[-2^{31} \mathinner{..} 2^{31})$ can easily compute sums, differences, and products modulo p_k. A set of r such primes makes it possible to add, subtract, and multiply "multiple-precision numbers" of up to almost $31r$ bits, and the residue system makes it possible to do this faster than if such large numbers were added, subtracted, or multiplied in other ways.

For example, the Mersenne prime $2^{31} - 1$ *works well.*

We can even do division, in appropriate circumstances. For example, suppose we want to compute the exact value of a large determinant of integers. The result will be an integer D, and bounds on $|D|$ can be given based on the size of its entries. But the only fast ways known for calculating determinants

require division, and this leads to fractions (and loss of accuracy, if we resort to binary approximations). The remedy is to evaluate $D \bmod p_k = D_k$, for various large primes p_k. We can safely divide modulo p_k unless the divisor happens to be a multiple of p_k. That's very unlikely, but if it does happen we can choose another prime. Finally, knowing D_k for sufficiently many primes, we'll have enough information to determine D.

But we haven't explained how to get from a given sequence of residues $(x \bmod m_1, \ldots, x \bmod m_r)$ back to $x \bmod m$. We've shown that this conversion can be done in principle, but the calculations might be so formidable that they might rule out the idea in practice. Fortunately, there is a reasonably simple way to do the job, and we can illustrate it in the situation $(x \bmod 3, x \bmod 5)$ shown in our little table. The key idea is to solve the problem in the two cases $(1,0)$ and $(0,1)$; for if $(1,0) = a$ and $(0,1) = b$, then $(x,y) = (ax + by) \bmod 15$, since congruences can be multiplied and added.

In our case $a = 10$ and $b = 6$, by inspection of the table; but how could we find a and b when the moduli are huge? In other words, if $m \perp n$, what is a good way to find numbers a and b such that the equations

$$a \bmod m \;=\; 1, \quad a \bmod n \;=\; 0, \quad b \bmod m \;=\; 0, \quad b \bmod n \;=\; 1$$

all hold? Once again, (4.5) comes to the rescue: With Euclid's algorithm, we can find m' and n' such that

$$m'm + n'n \;=\; 1\,.$$

Therefore we can take $a = n'n$ and $b = m'm$, reducing them both mod mn if desired.

Further tricks are needed in order to minimize the calculations when the moduli are large; the details are beyond the scope of this book, but they can be found in [208, page 274]. Conversion from residues to the corresponding original numbers is feasible, but it is sufficiently slow that we save total time only if a sequence of operations can all be done in the residue number system before converting back.

Let's firm up these congruence ideas by trying to solve a little problem: How many solutions are there to the congruence

$$x^2 \;\equiv\; 1 \pmod{m}\,, \tag{4.43}$$

if we consider two solutions x and x' to be the same when $x \equiv x'$?

According to the general principles explained earlier, we should consider first the case that m is a prime power, p^k, where $k > 0$. Then the congruence $x^2 \equiv 1$ can be written

$$(x-1)(x+1) \;\equiv\; 0 \pmod{p^k}\,,$$

so p must divide either $x-1$ or $x+1$, or both. But p can't divide both $x-1$ and $x+1$ unless $p=2$; we'll leave that case for later. If $p>2$, then $p^k \backslash (x-1)(x+1) \iff p^k \backslash (x-1)$ or $p^k \backslash (x+1)$; so there are exactly two solutions, $x \equiv +1$ and $x \equiv -1$.

The case $p=2$ is a little different. If $2^k \backslash (x-1)(x+1)$ then either $x-1$ or $x+1$ is divisible by 2 but not by 4, so the other one must be divisible by 2^{k-1}. This means that we have four solutions when $k \geqslant 3$, namely $x \equiv \pm 1$ and $x \equiv 2^{k-1} \pm 1$. (For example, when $p^k = 8$ the four solutions are $x \equiv 1$, 3, 5, 7 (mod 8); it's often useful to know that *the square of any odd integer has the form* $8n+1$.)

Now $x^2 \equiv 1 \pmod{m}$ if and only if $x^2 \equiv 1 \pmod{p^{m_p}}$ for all primes p with $m_p > 0$ in the complete factorization of m. Each prime is independent of the others, and there are exactly two possibilities for $x \bmod p^{m_p}$ except when $p=2$. Therefore if m has exactly r different prime divisors, the total number of solutions to $x^2 \equiv 1$ is 2^r, except for a correction when m is even. The exact number in general is

All primes are odd except 2, which is the oddest of all.

$$2^{r+[8\backslash m]+[4\backslash m]-[2\backslash m]}\,. \tag{4.44}$$

For example, there are four "square roots of unity modulo 12," namely 1, 5, 7, and 11. When $m=15$ the four are those whose residues mod 3 and mod 5 are ± 1, namely $(1,1)$, $(1,4)$, $(2,1)$, and $(2,4)$ in the residue number system. These solutions are 1, 4, 11, and 14 in the ordinary (decimal) number system.

4.8 ADDITIONAL APPLICATIONS

There's some unfinished business left over from Chapter 3: We wish to prove that the m numbers

$$0 \bmod m,\quad n \bmod m,\quad 2n \bmod m,\quad \ldots,\quad (m-1)n \bmod m \tag{4.45}$$

consist of precisely d copies of the m/d numbers

$$0,\quad d,\quad 2d,\quad \ldots,\quad m-d$$

in some order, where $d = \gcd(m,n)$. For example, when $m=12$ and $n=8$ we have $d=4$, and the numbers are 0, 8, 4, 0, 8, 4, 0, 8, 4, 0, 8, 4.

The first part of the proof — to show that we get d copies of the first m/d values — is now trivial. We have

Mathematicians love to say that things are trivial.

$$jn \equiv kn \pmod{m} \qquad \iff \qquad j(n/d) \equiv k(n/d) \pmod{m/d}$$

by (4.38); hence we get d copies of the values that occur when $0 \leqslant k < m/d$.

Now we must show that those m/d numbers are $\{0, d, 2d, \ldots, m-d\}$ in some order. Let's write $m = m'd$ and $n = n'd$. Then $kn \bmod m = d(kn' \bmod m')$, by the distributive law (3.23); so the values that occur when $0 \leqslant k < m'$ are d times the numbers

$$0 \bmod m', \quad n' \bmod m', \quad 2n' \bmod m', \quad \ldots, \quad (m'-1)n' \bmod m'.$$

But we know that $m' \perp n'$ by (4.27); we've divided out their gcd. Therefore we need only consider the case $d = 1$, namely the case that m and n are relatively prime.

So let's assume that $m \perp n$. In this case it's easy to see that the numbers (4.45) are just $\{0, 1, \ldots, m-1\}$ in some order, by using the "pigeonhole principle." This principle states that if m pigeons are put into m pigeonholes, there is an empty hole if and only if there's a hole with more than one pigeon. (Dirichlet's box principle, proved in exercise 3.8, is similar.) We know that the numbers (4.45) are distinct, because

$$jn \equiv kn \pmod{m} \qquad \Longleftrightarrow \qquad j \equiv k \pmod{m}$$

when $m \perp n$; this is (4.37). Therefore the m different numbers must fill all the pigeonholes $0, 1, \ldots, m-1$. Therefore the unfinished business of Chapter 3 is finished.

The proof is complete, but we can prove even more if we use a direct method instead of relying on the indirect pigeonhole argument. If $m \perp n$ and if a value $j \in [0\,..\,m)$ is given, we can explicitly compute $k \in [0\,..\,m)$ such that $kn \bmod m = j$ by solving the congruence

$$kn \equiv j \pmod{m}$$

for k. We simply multiply both sides by n', where $m'm + n'n = 1$, to get

$$k \equiv jn' \pmod{m};$$

hence $k = jn' \bmod m$.

We can use the facts just proved to establish an important result discovered by Pierre de Fermat in 1640. Fermat was a great mathematician who contributed to the discovery of calculus and many other parts of mathematics. He left notebooks containing dozens of theorems stated without proof, and each of those theorems has subsequently been verified — with the possible exception of one that became the most famous of all, because it baffled the world's best mathematicians for 350 years. The famous one, called "Fermat's Last Theorem," states that

$$a^n + b^n \neq c^n \tag{4.46}$$

NEWS FLASH

Euler [115] conjectured that $a^4+b^4+c^4 \neq d^4$, *but Noam Elkies [92] found infinitely many solutions in August, 1987.*

Now Roger Frye has done an exhaustive computer search, proving (after about 110 hours on a Connection Machine) that the only solution with $d < 1000000$ *is:*

$$95800^4 + 217519^4 + 414560^4 = 422481^4 .$$

for all positive integers a, b, c, and n, when $n > 2$. (Of course there are lots of solutions to the equations $a + b = c$ and $a^2 + b^2 = c^2$.) Andrew Wiles culminated many years of research by announcing a proof of (4.46) in 1993; his proof is currently being subjected to intense scrutiny.

Fermat's theorem of 1640 is much easier to verify. It's now called Fermat's Little Theorem (or just Fermat's theorem, for short), and it states that

$$n^{p-1} \equiv 1 \pmod{p}, \qquad \text{if } n \perp p. \tag{4.47}$$

Proof: As usual, we assume that p denotes a prime. We know that the $p-1$ numbers $n \bmod p$, $2n \bmod p$, ..., $(p-1)n \bmod p$ are the numbers 1, 2, ..., $p-1$ in some order. Therefore if we multiply them together we get

$$\begin{aligned} &n \cdot (2n) \cdot \ldots \cdot \big((p-1)n\big) \\ &\qquad \equiv (n \bmod p) \cdot (2n \bmod p) \cdot \ldots \cdot \big((p-1)n \bmod p\big) \\ &\qquad \equiv (p-1)!\,, \end{aligned}$$

where the congruence is modulo p. This means that

$$(p-1)!\, n^{p-1} \equiv (p-1)! \pmod{p},$$

and we can cancel the $(p-1)!$ since it's not divisible by p. QED.

An alternative form of Fermat's theorem is sometimes more convenient:

$$n^p \equiv n \pmod{p}, \qquad \text{integer } n. \tag{4.48}$$

This congruence holds for all integers n. The proof is easy: If $n \perp p$ we simply multiply (4.47) by n. If not, $p \backslash n$, so $n^p \equiv 0 \equiv n$.

In the same year that he discovered (4.47), Fermat wrote a letter to Mersenne, saying he suspected that the number

$$f_n = 2^{2^n} + 1$$

"... laquelle proposition, si elle est vraie, est de très grand usage."
—P. de Fermat [121]

would turn out to be prime for all $n \geqslant 0$. He knew that the first five cases gave primes:

$$2^1+1 = 3;\ 2^2+1 = 5;\ 2^4+1 = 17;\ 2^8+1 = 257;\ 2^{16}+1 = 65537;$$

but he couldn't see how to prove that the next case, $2^{32} + 1 = 4294967297$, would be prime.

It's interesting to note that Fermat could have proved that $2^{32}+1$ is *not* prime, using his own recently discovered theorem, if he had taken time to perform a few dozen multiplications: We can set $n = 3$ in (4.47), deducing that

$$3^{2^{32}} \equiv 1 \pmod{2^{32}+1}, \qquad \text{if } 2^{32}+1 \text{ is prime.}$$

And it's possible to test this relation by hand, beginning with 3 and squaring 32 times, keeping only the remainders mod $2^{32}+1$. First we have $3^2 = 9$, then $3^{2^2} = 81$, then $3^{2^3} = 6561$, and so on until we reach

If this is Fermat's Little Theorem, the other one was last but not least.

$$3^{2^{32}} \equiv 3029026160 \pmod{2^{32}+1}.$$

The result isn't 1, so $2^{32}+1$ isn't prime. This method of disproof gives us no clue about what the factors might be, but it does prove that factors exist. (They are 641 and 6700417, first found by Euler in 1732 [102].)

If $3^{2^{32}}$ had turned out to be 1, modulo $2^{32}+1$, the calculation wouldn't have proved that $2^{32}+1$ is prime; it just wouldn't have disproved it. But exercise 47 discusses a converse to Fermat's theorem by which we *can* prove that large prime numbers are prime, without doing an enormous amount of laborious arithmetic.

We proved Fermat's theorem by cancelling $(p-1)!$ from both sides of a congruence. It turns out that $(p-1)!$ is always congruent to -1, modulo p; this is part of a classical result known as Wilson's theorem:

$$(n-1)! \equiv -1 \pmod{n} \quad \Longleftrightarrow \quad n \text{ is prime}, \quad \text{if } n > 1. \tag{4.49}$$

One half of this theorem is trivial: If $n > 1$ is not prime, it has a prime divisor p that appears as a factor of $(n-1)!$, so $(n-1)!$ cannot be congruent to -1. (If $(n-1)!$ were congruent to -1 modulo n, it would also be congruent to -1 modulo p, but it isn't.)

The other half of Wilson's theorem states that $(p-1)! \equiv -1 \pmod{p}$. We can prove this half by pairing up numbers with their inverses mod p. If $n \perp p$, we know that there exists n' such that

$$n'n \equiv 1 \pmod{p};$$

here n' is the inverse of n, and n is also the inverse of n'. Any two inverses of n must be congruent to each other, since $nn' \equiv nn''$ implies $n' \equiv n''$.

If p is prime, is p' prime prime?

Now suppose we pair up each number between 1 and $p-1$ with its inverse. Since the product of a number and its inverse is congruent to 1, the product of all the numbers in all pairs of inverses is also congruent to 1; so it seems that $(p-1)!$ is congruent to 1. Let's check, say for $p = 5$. We get $4! = 24$; but this is congruent to 4, not 1, modulo 5. Oops — what went wrong? Let's take a closer look at the inverses:

$$1' = 1, \qquad 2' = 3, \qquad 3' = 2, \qquad 4' = 4.$$

Ah so; 2 and 3 pair up but 1 and 4 don't — they're their own inverses.

To resurrect our analysis we must determine which numbers are their own inverses. If x is its own inverse, then $x^2 \equiv 1 \pmod{p}$; and we have

already proved that this congruence has exactly two roots when $p > 2$. (If $p = 2$ it's obvious that $(p-1)! \equiv -1$, so we needn't worry about that case.) The roots are 1 and $p-1$, and the other numbers (between 1 and $p-1$) pair up; hence

$$(p-1)! \;\equiv\; 1 \cdot (p-1) \;\equiv\; -1\,,$$

as desired.

Unfortunately, we can't compute factorials efficiently, so Wilson's theorem is of no use as a practical test for primality. It's just a theorem.

4.9 PHI AND MU

How many of the integers $\{0, 1, \ldots, m-1\}$ are relatively prime to m? This is an important quantity called $\varphi(m)$, the "totient" of m (so named by J. J. Sylvester [347], a British mathematician who liked to invent new words). We have $\varphi(1) = 1$, $\varphi(p) = p - 1$, and $\varphi(m) < m - 1$ for all composite numbers m.

The φ function is called *Euler's totient function*, because Euler was the first person to study it. Euler discovered, for example, that Fermat's theorem (4.47) can be generalized to nonprime moduli in the following way:

"Si fuerit N *ad* x *numerus primus et* n *numerus partium ad* N *primarum, tum potestas* x^n *unitate minuta semper per numerum* N *erit divisibilis."*
—L. Euler [111]

$$n^{\varphi(m)} \;\equiv\; 1 \pmod{m}\,, \qquad \text{if } n \perp m. \tag{4.50}$$

(Exercise 32 asks for a proof of Euler's theorem.)

If m is a prime power p^k, it's easy to compute $\varphi(m)$, because $n \perp p^k \iff p \nmid n$. The multiples of p in $\{0, 1, \ldots, p^k - 1\}$ are $\{0, p, 2p, \ldots, p^k - p\}$; hence there are p^{k-1} of them, and $\varphi(p^k)$ counts what is left:

$$\varphi(p^k) \;=\; p^k - p^{k-1}\,.$$

Notice that this formula properly gives $\varphi(p) = p - 1$ when $k = 1$.

If $m > 1$ is not a prime power, we can write $m = m_1 m_2$ where $m_1 \perp m_2$. Then the numbers $0 \leqslant n < m$ can be represented in a residue number system as $(n \bmod m_1, n \bmod m_2)$. We have

$$n \perp m \qquad \iff \qquad n \bmod m_1 \perp m_1 \quad \text{and} \quad n \bmod m_2 \perp m_2$$

by (4.30) and (4.4). Hence, $n \bmod m$ is "good" if and only if $n \bmod m_1$ and $n \bmod m_2$ are both "good," if we consider relative primality to be a virtue. The total number of good values modulo m can now be computed, recursively: It is $\varphi(m_1)\varphi(m_2)$, because there are $\varphi(m_1)$ good ways to choose the first component $n \bmod m_1$ and $\varphi(m_2)$ good ways to choose the second component $n \bmod m_2$ in the residue representation.

For example, $\varphi(12) = \varphi(4)\varphi(3) = 2\cdot 2 = 4$, because n is prime to 12 if and only if $n \bmod 4 = (1 \text{ or } 3)$ and $n \bmod 3 = (1 \text{ or } 2)$. The four values prime to 12 are $(1,1)$, $(1,2)$, $(3,1)$, $(3,2)$ in the residue number system; they are 1, 5, 7, 11 in ordinary decimal notation. Euler's theorem states that $n^4 \equiv 1 \pmod{12}$ whenever $n \perp 12$.

"Si sint A *et* B *numeri inter se primi et numerus partium ad* A *primarum sit* $= a$, *numerus vero partium ad* B *primarum sit* $= b$, *tum numerus partium ad productum* AB *primarum erit* $= ab$*."*
—*L. Euler [111]*

A function $f(m)$ of positive integers is called *multiplicative* if $f(1) = 1$ and

$$f(m_1 m_2) \;=\; f(m_1)f(m_2) \qquad \text{whenever } m_1 \perp m_2. \tag{4.51}$$

We have just proved that $\varphi(m)$ is multiplicative. We've also seen another instance of a multiplicative function earlier in this chapter: The number of incongruent solutions to $x^2 \equiv 1 \pmod{m}$ is multiplicative. Still another example is $f(m) = m^\alpha$ for any power α.

A multiplicative function is defined completely by its values at prime powers, because we can decompose any positive integer m into its prime-power factors, which are relatively prime to each other. The general formula

$$f(m) \;=\; \prod_p f(p^{m_p}), \qquad \text{if } m = \prod_p p^{m_p} \tag{4.52}$$

holds if and only if f is multiplicative.

In particular, this formula gives us the value of Euler's totient function for general m:

$$\varphi(m) \;=\; \prod_{p\backslash m}(p^{m_p} - p^{m_p-1}) \;=\; m\prod_{p\backslash m}\Big(1 - \frac{1}{p}\Big). \tag{4.53}$$

For example, $\varphi(12) = (4-2)(3-1) = 12(1-\frac{1}{2})(1-\frac{1}{3})$.

Now let's look at an application of the φ function to the study of rational numbers mod 1. We say that the fraction m/n is *basic* if $0 \leqslant m < n$. Therefore $\varphi(n)$ is the number of reduced basic fractions with denominator n; and the Farey series $\mathcal{F}_n$ contains all the reduced basic fractions with denominator n or less, as well as the non-basic fraction $\frac{1}{1}$.

The set of *all* basic fractions with denominator 12, before reduction to lowest terms, is

$$\tfrac{0}{12},\ \tfrac{1}{12},\ \tfrac{2}{12},\ \tfrac{3}{12},\ \tfrac{4}{12},\ \tfrac{5}{12},\ \tfrac{6}{12},\ \tfrac{7}{12},\ \tfrac{8}{12},\ \tfrac{9}{12},\ \tfrac{10}{12},\ \tfrac{11}{12}.$$

Reduction yields

$$\tfrac{0}{1},\ \tfrac{1}{12},\ \tfrac{1}{6},\ \tfrac{1}{4},\ \tfrac{1}{3},\ \tfrac{5}{12},\ \tfrac{1}{2},\ \tfrac{7}{12},\ \tfrac{2}{3},\ \tfrac{3}{4},\ \tfrac{5}{6},\ \tfrac{11}{12},$$

and we can group these fractions by their denominators:

$$\tfrac{0}{1};\quad \tfrac{1}{2};\quad \tfrac{1}{3},\ \tfrac{2}{3};\quad \tfrac{1}{4},\ \tfrac{3}{4};\quad \tfrac{1}{6},\ \tfrac{5}{6};\quad \tfrac{1}{12},\ \tfrac{5}{12},\ \tfrac{7}{12},\ \tfrac{11}{12}.$$

What can we make of this? Well, every divisor d of 12 occurs as a denominator, together with all $\varphi(d)$ of its numerators. The only denominators that occur are divisors of 12. Thus

$$\varphi(1)+\varphi(2)+\varphi(3)+\varphi(4)+\varphi(6)+\varphi(12) \;=\; 12\,.$$

A similar thing will obviously happen if we begin with the unreduced fractions $\frac{0}{m}, \frac{1}{m}, \ldots, \frac{m-1}{m}$ for any m, hence

$$\sum_{d\backslash m} \varphi(d) \;=\; m\,. \tag{4.54}$$

We said near the beginning of this chapter that problems in number theory often require sums over the divisors of a number. Well, (4.54) is one such sum, so our claim is vindicated. (We will see other examples.)

Now here's a curious fact: If f is any function such that the sum

$$g(m) \;=\; \sum_{d\backslash m} f(d)$$

is multiplicative, then f itself is multiplicative. (This result, together with (4.54) and the fact that $g(m) = m$ is obviously multiplicative, gives another reason why $\varphi(m)$ is multiplicative.) We can prove this curious fact by induction on m: The basis is easy because $f(1) = g(1) = 1$. Let $m > 1$, and assume that $f(m_1 m_2) = f(m_1)f(m_2)$ whenever $m_1 \perp m_2$ and $m_1 m_2 < m$. If $m = m_1 m_2$ and $m_1 \perp m_2$, we have

$$g(m_1 m_2) \;=\; \sum_{d\backslash m_1 m_2} f(d) \;=\; \sum_{d_1\backslash m_1}\ \sum_{d_2\backslash m_2} f(d_1 d_2)\,,$$

and $d_1 \perp d_2$ since all divisors of m_1 are relatively prime to all divisors of m_2. By the induction hypothesis, $f(d_1 d_2) = f(d_1)f(d_2)$ except possibly when $d_1 = m_1$ and $d_2 = m_2$; hence we obtain

$$\begin{aligned}\Bigl(\sum_{d_1\backslash m_1} f(d_1) \sum_{d_2\backslash m_2} f(d_2)\Bigr) &- f(m_1)f(m_2) + f(m_1 m_2)\\ &= g(m_1)g(m_2) - f(m_1)f(m_2) + f(m_1 m_2)\,.\end{aligned}$$

But this equals $g(m_1 m_2) = g(m_1)g(m_2)$, so $f(m_1 m_2) = f(m_1)f(m_2)$.

Conversely, if $f(m)$ is multiplicative, the corresponding sum-over-divisors function $g(m) = \sum_{d\backslash m} f(d)$ is always multiplicative. In fact, exercise 33 shows that even more is true. Hence the curious fact and its converse are both facts.

The *Möbius function* $\mu(m)$, named after the nineteenth-century mathematician August Möbius who also had a famous band, can be defined for all integers $m \geqslant 1$ by the equation

$$\sum_{d\backslash m} \mu(d) = [m=1]. \tag{4.55}$$

This equation is actually a recurrence, since the left-hand side is a sum consisting of $\mu(m)$ and certain values of $\mu(d)$ with $d < m$. For example, if we plug in $m = 1, 2, \ldots, 12$ successively we can compute the first twelve values:

m	1	2	3	4	5	6	7	8	9	10	11	12
$\mu(m)$	1	−1	−1	0	−1	1	−1	0	0	1	−1	0

Richard Dedekind [77] and Joseph Liouville [251] noticed the following important "inversion principle" in 1857:

$$g(m) = \sum_{d\backslash m} f(d) \qquad \Longleftrightarrow \qquad f(m) = \sum_{d\backslash m} \mu(d) g(\frac{m}{d}). \tag{4.56}$$

According to this principle, the μ function gives us a new way to understand any function $f(m)$ for which we know $\sum_{d\backslash m} f(d)$.

Now is a good time to try warmup exercise 11.

The proof of (4.56) uses two tricks (4.7) and (4.9) that we described near the beginning of this chapter: If $g(m) = \sum_{d\backslash m} f(d)$ then

$$\begin{aligned}
\sum_{d\backslash m} \mu(d) g(\frac{m}{d}) &= \sum_{d\backslash m} \mu(\frac{m}{d}) g(d) \\
&= \sum_{d\backslash m} \mu(\frac{m}{d}) \sum_{k\backslash d} f(k) \\
&= \sum_{k\backslash m} \sum_{d\backslash (m/k)} \mu(\frac{m}{kd}) f(k) \\
&= \sum_{k\backslash m} \sum_{d\backslash (m/k)} \mu(d) f(k) \\
&= \sum_{k\backslash m} [m/k=1] f(k) = f(m).
\end{aligned}$$

The other half of (4.56) is proved similarly (see exercise 12).

Relation (4.56) gives us a useful property of the Möbius function, and we have tabulated the first twelve values; but what is the value of $\mu(m)$ when m is large? How can we solve the recurrence (4.55)? Well, the function $g(m) = [m=1]$ is obviously multiplicative — after all, it's zero except when $m = 1$. So the Möbius function defined by (4.55) must be multiplicative, by the curious fact we proved a minute or two ago. Therefore we can figure out what $\mu(m)$ is if we compute $\mu(p^k)$.

Depending on how fast you read.

When $m = p^k$, (4.55) says that

$$\mu(1) + \mu(p) + \mu(p^2) + \cdots + \mu(p^k) \;=\; 0$$

for all $k \geqslant 1$, since the divisors of p^k are $1, \ldots, p^k$. It follows that

$$\mu(p) \;=\; -1\,; \qquad \mu(p^k) \;=\; 0 \quad \text{for } k > 1.$$

Therefore by (4.52), we have the general formula

$$\mu(m) \;=\; \prod_{p \backslash m} \mu(p^{m_p}) \;=\; \begin{cases} (-1)^r, & \text{if } m = p_1 p_2 \ldots p_r; \\ 0, & \text{if } m \text{ is divisible by some } p^2. \end{cases} \tag{4.57}$$

That's μ.

If we regard (4.54) as a recurrence for the function $\varphi(m)$, we can solve that recurrence by applying the Dedekind-Liouville rule (4.56). We get

$$\varphi(m) \;=\; \sum_{d \backslash m} \mu(d)\, \frac{m}{d}\,. \tag{4.58}$$

For example,

$$\begin{aligned} \varphi(12) \;&=\; \mu(1)\cdot 12 + \mu(2)\cdot 6 + \mu(3)\cdot 4 + \mu(4)\cdot 3 + \mu(6)\cdot 2 + \mu(12)\cdot 1 \\ &=\; 12 - 6 - 4 + 0 + 2 + 0 \;=\; 4\,. \end{aligned}$$

If m is divisible by r different primes, say $\{p_1, \ldots, p_r\}$, the sum (4.58) has only 2^r nonzero terms, because the μ function is often zero. Thus we can see that (4.58) checks with formula (4.53), which reads

$$\varphi(m) \;=\; m\Big(1 - \frac{1}{p_1}\Big) \ldots \Big(1 - \frac{1}{p_r}\Big)\,;$$

if we multiply out the r factors $(1 - 1/p_j)$, we get precisely the 2^r nonzero terms of (4.58). The advantage of the Möbius function is that it applies in many situations besides this one.

For example, let's try to figure out how many fractions are in the Farey series $\mathcal{F}_n$. This is the number of reduced fractions in $[0\,..\,1]$ whose denominators do not exceed n, so it is 1 greater than $\Phi(n)$ where we define

$$\Phi(x) \;=\; \sum_{1\leqslant k\leqslant x} \varphi(k)\,. \tag{4.59}$$

(We must add 1 to $\Phi(n)$ because of the final fraction $\frac{1}{1}$.) The sum in (4.59) looks difficult, but we can determine $\Phi(x)$ indirectly by observing that

$$\sum_{d\geqslant 1} \Phi\Bigl(\frac{x}{d}\Bigr) \;=\; \frac{1}{2}\lfloor x\rfloor\lfloor 1+x\rfloor \tag{4.60}$$

for all real $x \geqslant 0$. Why does this identity hold? Well, it's a bit awesome yet not really beyond our ken. There are $\frac{1}{2}\lfloor x\rfloor\lfloor 1+x\rfloor$ basic fractions m/n with $0 \leqslant m < n \leqslant x$, counting both reduced and unreduced fractions; that gives us the right-hand side. The number of such fractions with $\gcd(m,n) = d$ is $\Phi(x/d)$, because such fractions are m'/n' with $0 \leqslant m' < n' \leqslant x/d$ after replacing m by $m'd$ and n by $n'd$. So the left-hand side counts the same fractions in a different way, and the identity must be true.

Let's look more closely at the situation, so that equations (4.59) and (4.60) become clearer. The definition of $\Phi(x)$ implies that $\Phi(x) = \Phi\bigl(\lfloor x\rfloor\bigr)$; but it turns out to be convenient to define $\Phi(x)$ for arbitrary real values, not just for integers. At integer values we have the table

(This extension to real values is a useful trick for many recurrences that arise in the analysis of algorithms.)

n	0	1	2	3	4	5	6	7	8	9	10	11	12
$\varphi(n)$	–	1	1	2	2	4	2	6	4	6	4	10	4
$\Phi(n)$	0	1	2	4	6	10	12	18	22	28	32	42	46

and we can check (4.60) when $x = 12$:

$$\begin{aligned}&\Phi(12)+\Phi(6)+\Phi(4)+\Phi(3)+\Phi(2)+\Phi(2)+6\cdot\Phi(1)\\ &\qquad = 46+12+6+4+2+2+6 \;=\; 78 \;=\; \tfrac{1}{2}\cdot 12\cdot 13\,.\end{aligned}$$

Amazing.

Identity (4.60) can be regarded as an implicit recurrence for $\Phi(x)$; for example, we've just seen that we could have used it to calculate $\Phi(12)$ from certain values of $\Phi(m)$ with $m < 12$. And we can solve such recurrences by using another beautiful property of the Möbius function:

In fact, Möbius [273] invented his function because of (4.61), not (4.56).

$$g(x) \;=\; \sum_{d\geqslant 1} f(x/d) \qquad\Longleftrightarrow\qquad f(x) \;=\; \sum_{d\geqslant 1} \mu(d)\,g(x/d)\,. \tag{4.61}$$

This inversion law holds for all functions f such that $\sum_{k,d\geqslant 1}\bigl|f(x/kd)\bigr| < \infty$; we can prove it as follows. Suppose $g(x) = \sum_{d\geqslant 1} f(x/d)$. Then

$$\begin{aligned}
\sum_{d\geqslant 1}\mu(d)g(x/d) &= \sum_{d\geqslant 1}\mu(d)\sum_{k\geqslant 1} f(x/kd) \\
&= \sum_{m\geqslant 1} f(x/m)\sum_{d,k\geqslant 1}\mu(d)[m=kd] \\
&= \sum_{m\geqslant 1} f(x/m)\sum_{d\backslash m}\mu(d) \;=\; \sum_{m\geqslant 1} f(x/m)[m=1] \;=\; f(x)\,.
\end{aligned}$$

The proof in the other direction is essentially the same.

So now we can solve the recurrence (4.60) for $\Phi(x)$:

$$\Phi(x) \;=\; \frac{1}{2}\sum_{d\geqslant 1}\mu(d)\lfloor x/d\rfloor\lfloor 1+x/d\rfloor\,. \tag{4.62}$$

This is always a finite sum. For example,

$$\begin{aligned}
\Phi(12) &= \tfrac{1}{2}(12\cdot 13 - 6\cdot 7 - 4\cdot 5 + 0 - 2\cdot 3 + 2\cdot 3 \\
&\qquad - 1\cdot 2 + 0 + 0 + 1\cdot 2 - 1\cdot 2 + 0) \\
&= 78 - 21 - 10 - 3 + 3 - 1 + 1 - 1 \;=\; 46\,.
\end{aligned}$$

In Chapter 9 we'll see how to use (4.62) to get a good approximation to $\Phi(x)$; in fact, we'll prove a result due to Mertens in 1874 [270],

$$\Phi(x) \;=\; \frac{3}{\pi^2}x^2 + O(x\log x)\,.$$

Therefore the function $\Phi(x)$ grows "smoothly"; it averages out the erratic behavior of $\varphi(k)$.

In keeping with the tradition established last chapter, let's conclude this chapter with a problem that illustrates much of what we've just seen and that also points ahead to the next chapter. Suppose we have beads of n different colors; our goal is to count how many different ways there are to string them into circular necklaces of length m. We can try to "name and conquer" this problem by calling the number of possible necklaces $N(m,n)$.

For example, with two colors of beads R and B, we can make necklaces of length 4 in $N(4,2) = 6$ different ways:

All other ways are equivalent to one of these, because rotations of a necklace do not change it. However, reflections are considered to be different; in the

case $m = 6$, for example,

$$\begin{array}{ccc} & \text{B} & \\ \text{R} & & \text{R} \\ | & & | \\ \text{R} & & \text{B} \\ & \text{B} & \end{array} \quad \text{is different from} \quad \begin{array}{ccc} & \text{B} & \\ \text{R} & & \text{R} \\ | & & | \\ \text{B} & & \text{R} \\ & \text{B} & \end{array}.$$

The problem of counting these configurations was first solved by P. A. MacMahon in 1892 [264].

There's no obvious recurrence for $N(m, n)$, but we can count the necklaces by breaking them each into linear strings in m ways and considering the resulting fragments. For example, when $m = 4$ and $n = 2$ we get

RRRR	RRRR	RRRR	RRRR
RRBR	RRRB	BRRR	RBRR
RBBR	RRBB	BRRB	BBRR
RBRB	BRBR	RBRB	BRBR
RBBB	BRBB	BBRB	BBBR
BBBB	BBBB	BBBB	BBBB

Each of the n^m possible patterns appears at least once in this array of $mN(m, n)$ strings, and some patterns appear more than once. How many times does a pattern $a_0 \ldots a_{m-1}$ appear? That's easy: It's the number of cyclic shifts $a_k \ldots a_{m-1} a_0 \ldots a_{k-1}$ that produce the same pattern as the original $a_0 \ldots a_{m-1}$. For example, BRBR occurs twice, because the four ways to cut the necklace formed from BRBR produce four cyclic shifts (BRBR, RBRB, BRBR, RBRB); two of these coincide with BRBR itself. This argument shows that

$$\begin{aligned} mN(m, n) &= \sum_{a_0, \ldots, a_{m-1} \in S_n} \sum_{0 \leqslant k < m} \left[a_0 \ldots a_{m-1} = a_k \ldots a_{m-1} a_0 \ldots a_{k-1} \right] \\ &= \sum_{0 \leqslant k < m} \sum_{a_0, \ldots, a_{m-1} \in S_n} \left[a_0 \ldots a_{m-1} = a_k \ldots a_{m-1} a_0 \ldots a_{k-1} \right]. \end{aligned}$$

Here S_n is a set of n different colors.

Let's see how many patterns satisfy $a_0 \ldots a_{m-1} = a_k \ldots a_{m-1} a_0 \ldots a_{k-1}$, when k is given. For example, if $m = 12$ and $k = 8$, we want to count the number of solutions to

$$a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} \;=\; a_8 a_9 a_{10} a_{11} a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 \,.$$

This means $a_0 = a_8 = a_4$; $a_1 = a_9 = a_5$; $a_2 = a_{10} = a_6$; and $a_3 = a_{11} = a_7$. So the values of a_0, a_1, a_2, and a_3 can be chosen in n^4 ways, and the

remaining a's depend on them. Does this look familiar? In general, the solution to

$$a_j \;=\; a_{(j+k) \bmod m}\,, \qquad \text{for } 0 \leqslant j < m$$

makes us equate a_j with $a_{(j+kl) \bmod m}$ for $l = 1, 2, \ldots$; and we know that the multiples of k modulo m are $\{0, d, 2d, \ldots, m-d\}$, where $d = \gcd(k, m)$. Therefore the general solution is to choose $a_0, \ldots, a_{d-1}$ independently and then to set $a_j = a_{j-d}$ for $d \leqslant j < m$. There are n^d solutions.

We have just proved that

$$mN(m,n) \;=\; \sum_{0 \leqslant k < m} n^{\gcd(k,m)}\,.$$

This sum can be simplified, since it includes only terms n^d where $d \backslash m$. Substituting $d = \gcd(k, m)$ yields

$$\begin{aligned} N(m,n) \;&=\; \frac{1}{m} \sum_{d \backslash m} n^d \sum_{0 \leqslant k < m} \bigl[d = \gcd(k,m)\bigr] \\ &=\; \frac{1}{m} \sum_{d \backslash m} n^d \sum_{0 \leqslant k < m} \bigl[k/d \perp m/d\bigr] \\ &=\; \frac{1}{m} \sum_{d \backslash m} n^d \sum_{0 \leqslant k < m/d} \bigl[k \perp m/d\bigr]\,. \end{aligned}$$

(We are allowed to replace k/d by k because k must be a multiple of d.) Finally, we have $\sum_{0 \leqslant k < m/d} [k \perp m/d] = \varphi(m/d)$ by definition, so we obtain MacMahon's formula:

$$N(m,n) \;=\; \frac{1}{m} \sum_{d \backslash m} n^d\, \varphi\Bigl(\frac{m}{d}\Bigr) \;=\; \frac{1}{m} \sum_{d \backslash m} \varphi(d)\, n^{m/d}\,. \tag{4.63}$$

When $m = 4$ and $n = 2$, for example, the number of necklaces is $\frac{1}{4}(1 \cdot 2^4 + 1 \cdot 2^2 + 2 \cdot 2^1) = 6$, just as we suspected.

It's not immediately obvious that the value $N(m,n)$ defined by MacMahon's sum is an integer! Let's try to prove directly that

$$\sum_{d \backslash m} \varphi(d)\, n^{m/d} \;\equiv\; 0 \qquad (\bmod\ m)\,, \tag{4.64}$$

without using the clue that this is related to necklaces. In the special case that m is prime, this congruence reduces to $n^p + (p-1)n \equiv 0 \pmod p$; that is, it reduces to $n^p \equiv n$. We've seen in (4.48) that this congruence is an alternative form of Fermat's theorem. Therefore (4.64) holds when $m = p$;

we can regard it as a generalization of Fermat's theorem to the case when the modulus is not prime. (Euler's generalization (4.50) is different.)

We've proved (4.64) for all prime moduli, so let's look at the smallest case left, $m = 4$. We must prove that

$$n^4 + n^2 + 2n \equiv 0 \pmod{4}.$$

The proof is easy if we consider even and odd cases separately. If n is even, all three terms on the left are congruent to 0 modulo 4, so their sum is too. If n is odd, n^4 and n^2 are each congruent to 1, and $2n$ is congruent to 2; hence the left side is congruent to $1+1+2$ and thus to 0 modulo 4, and we're done.

Next, let's be a bit daring and try $m = 12$. This value of m ought to be interesting because it has lots of factors, including the square of a prime, yet it is fairly small. (Also there's a good chance we'll be able to generalize a proof for 12 to a proof for general m.) The congruence we must prove is

$$n^{12} + n^6 + 2n^4 + 2n^3 + 2n^2 + 4n \equiv 0 \pmod{12}.$$

Now what? By (4.42) this congruence holds if and only if it also holds modulo 3 and modulo 4. So let's prove that it holds modulo 3. Our congruence (4.64) holds for primes, so we have $n^3 + 2n \equiv 0 \pmod{3}$. Careful scrutiny reveals that we can use this fact to group terms of the larger sum:

$$\begin{aligned} &n^{12} + n^6 + 2n^4 + 2n^3 + 2n^2 + 4n \\ &\quad = (n^{12} + 2n^4) + (n^6 + 2n^2) + 2(n^3 + 2n) \\ &\quad \equiv 0 + 0 + 2\cdot 0 \equiv 0 \pmod{3}. \end{aligned}$$

So it works modulo 3.

We're half done. To prove congruence modulo 4 we use the same trick. We've proved that $n^4 + n^2 + 2n \equiv 0 \pmod{4}$, so we use this pattern to group:

$$\begin{aligned} &n^{12} + n^6 + 2n^4 + 2n^3 + 2n^2 + 4n \\ &\quad = (n^{12} + n^6 + 2n^3) + 2(n^4 + n^2 + 2n) \\ &\quad \equiv 0 + 2\cdot 0 \equiv 0 \pmod{4}. \end{aligned}$$

QED for the case $m = 12$.

QED: Quite Easily Done.

So far we've proved our congruence for prime m, for $m = 4$, and for $m = 12$. Now let's try to prove it for prime powers. For concreteness we may suppose that $m = p^3$ for some prime p. Then the left side of (4.64) is

$$\begin{aligned} &n^{p^3} + \varphi(p)n^{p^2} + \varphi(p^2)n^p + \varphi(p^3)n \\ &\quad = n^{p^3} + (p-1)n^{p^2} + (p^2 - p)n^p + (p^3 - p^2)n \\ &\quad = (n^{p^3} - n^{p^2}) + p(n^{p^2} - n^p) + p^2(n^p - n) + p^3 n. \end{aligned}$$

We can show that this is congruent to 0 modulo p^3 if we can prove that $n^{p^3} - n^{p^2}$ is divisible by p^3, that $n^{p^2} - n^p$ is divisible by p^2, and that $n^p - n$ is divisible by p, because the whole thing will then be divisible by p^3. By the alternative form of Fermat's theorem we have $n^p \equiv n \pmod{p}$, so p divides $n^p - n$; hence there is an integer q such that

$$n^p = n + pq\,.$$

Now we raise both sides to the pth power, expand the right side according to the binomial theorem (which we'll meet in Chapter 5), and regroup, giving

$$\begin{aligned} n^{p^2} &= (n+pq)^p = n^p + (pq)^1 n^{p-1}\binom{p}{1} + (pq)^2 n^{p-2}\binom{p}{2} + \cdots \\ &= n^p + p^2 Q \end{aligned}$$

for some other integer Q. We're able to pull out a factor of p^2 here because $\binom{p}{1} = p$ in the second term, and because a factor of $(pq)^2$ appears in all the terms that follow. So we find that p^2 divides $n^{p^2} - n^p$.

Again we raise both sides to the pth power, expand, and regroup, to get

$$\begin{aligned} n^{p^3} &= (n^p + p^2 Q)^p \\ &= n^{p^2} + (p^2 Q)^1 n^{p(p-1)}\binom{p}{1} + (p^2 Q)^2 n^{p(p-2)}\binom{p}{2} + \cdots \\ &= n^{p^2} + p^3 \mathbf{Q} \end{aligned}$$

for yet another integer $\mathbf{Q}$. So p^3 divides $n^{p^3} - n^{p^2}$. This finishes the proof for $m = p^3$, because we've shown that p^3 divides the left-hand side of (4.64).

Moreover we can prove by induction that

$$n^{p^k} = n^{p^{k-1}} + p^k \mathfrak{Q}$$

for some final integer $\mathfrak{Q}$ (final because we're running out of fonts); hence

$$n^{p^k} \equiv n^{p^{k-1}} \pmod{p^k}, \qquad \text{for } k > 0. \tag{4.65}$$

Thus the left side of (4.64), which is

$$(n^{p^k} - n^{p^{k-1}}) + p(n^{p^{k-1}} - n^{p^{k-2}}) + \cdots + p^{k-1}(n^p - n) + p^k n\,,$$

is divisible by p^k and so is congruent to 0 modulo p^k.

We're almost there. Now that we've proved (4.64) for prime powers, all that remains is to prove it when $m = m_1 m_2$, where $m_1 \perp m_2$, assuming that the congruence is true for m_1 and m_2. Our examination of the case $m = 12$, which factored into instances of $m = 3$ and $m = 4$, encourages us to think that this approach will work.

We know that the φ function is multiplicative, so we can write

$$\begin{aligned}\sum_{d\backslash m}\varphi(d)\,n^{m/d} &= \sum_{d_1\backslash m_1,\,d_2\backslash m_2}\varphi(d_1d_2)\,n^{m_1m_2/d_1d_2}\\ &= \sum_{d_1\backslash m_1}\varphi(d_1)\Bigg(\sum_{d_2\backslash m_2}\varphi(d_2)(n^{m_1/d_1})^{m_2/d_2}\Bigg).\end{aligned}$$

But the inner sum is congruent to 0 modulo m_2, because we've assumed that (4.64) holds for m_2; so the entire sum is congruent to 0 modulo m_2. By a symmetric argument, we find that the entire sum is congruent to 0 modulo m_1 as well. Thus by (4.42) it's congruent to 0 modulo m. QED.

Exercises

Warmups

1 What is the smallest positive integer that has exactly k divisors, for $1 \leqslant k \leqslant 6$?

2 Prove that $\gcd(m,n)\cdot\operatorname{lcm}(m,n) = m\cdot n$, and use this identity to express $\operatorname{lcm}(m,n)$ in terms of $\operatorname{lcm}(n \bmod m, m)$, when $n \bmod m \neq 0$. *Hint:* Use (4.12), (4.14), and (4.15).

3 Let $\pi(x)$ be the number of primes not exceeding x. Prove or disprove:

$$\pi(x) - \pi(x-1) \;=\; [x \text{ is prime}]\,.$$

4 What would happen if the Stern–Brocot construction started with the five fractions $\left(\frac{0}{1}, \frac{1}{0}, \frac{0}{-1}, \frac{-1}{0}, \frac{0}{1}\right)$ instead of with $\left(\frac{0}{1}, \frac{1}{0}\right)$?

5 Find simple formulas for L^k and R^k, when L and R are the 2×2 matrices of (4.33).

6 What does '$a \equiv b \pmod 0$' mean?

7 Ten people numbered 1 to 10 are lined up in a circle as in the Josephus problem, and every mth person is executed. (The value of m may be much larger than 10.) Prove that the first three people to go cannot be 10, k, and $k+1$ (in this order), for any k.

8 The residue number system $(x \bmod 3, x \bmod 5)$ considered in the text has the curious property that 13 corresponds to $(1,3)$, which looks almost the same. Explain how to find all instances of such a coincidence, without calculating all fifteen pairs of residues. In other words, find all solutions to the congruences

$$10x + y \;\equiv\; x \pmod 3\,, \qquad 10x + y \;\equiv\; y \pmod 5\,.$$

Hint: Use the facts that $10u+6v \equiv u \pmod 3$ and $10u+6v \equiv v \pmod 5$.

9 Show that $(3^{77}-1)/2$ is odd and composite. *Hint:* What is $3^{77} \bmod 4$?

10 Compute $\varphi(999)$.

11 Find a function $\sigma(n)$ with the property that

$$g(n) = \sum_{0\leqslant k\leqslant n} f(k) \qquad \Longleftrightarrow \qquad f(n) = \sum_{0\leqslant k\leqslant n} \sigma(k)\, g(n-k)\,.$$

(This is analogous to the Möbius function; see (4.56).)

12 Simplify the formula $\sum_{d\backslash m}\sum_{k\backslash d}\mu(k)\,g(d/k)$.

13 A positive integer n is called *squarefree* if it is not divisible by m^2 for any $m>1$. Find a necessary and sufficient condition that n is squarefree,

a in terms of the prime-exponent representation (4.11) of n;

b in terms of $\mu(n)$.

Basics

14 Prove or disprove:

a $\gcd(km,kn) = k\gcd(m,n)$;

b $\operatorname{lcm}(km,kn) = k\operatorname{lcm}(m,n)$.

15 Does every prime occur as a factor of some Euclid number e_n?

16 What is the sum of the reciprocals of the first n Euclid numbers?

17 Let f_n be the "Fermat number" $2^{2^n}+1$. Prove that $f_m \perp f_n$ if $m<n$.

18 Show that if 2^n+1 is prime then n is a power of 2.

19 Prove the following identities when n is a positive integer:

$$\sum_{1\leqslant k<n}\left\lfloor\frac{\varphi(k+1)}{k}\right\rfloor = \sum_{1<m\leqslant n}\left\lfloor\left(\sum_{1\leqslant k<m}\lfloor(m/k)/\lceil m/k\rceil\rfloor\right)^{-1}\right\rfloor$$
$$= n-1-\sum_{k=1}^{n}\left\lceil\left\{\frac{(k-1)!+1}{k}\right\}\right\rceil.$$

Hint: This is a trick question and the answer is pretty easy.

20 For every positive integer n there's a prime p such that $n<p\leqslant 2n$. (This is essentially "Bertrand's postulate," which Joseph Bertrand verified for $n<3000000$ in 1845 and Chebyshev proved for all n in 1850.) Use Bertrand's postulate to prove that there's a constant $b\approx 1.25$ such that the numbers

$$\lfloor 2^b\rfloor,\ \lfloor 2^{2^b}\rfloor,\ \lfloor 2^{2^{2^b}}\rfloor,\ \ldots$$

are all prime.

21 Let P_n be the nth prime number. Find a constant K such that

$$\lfloor (10^{n^2} K) \bmod 10^n \rfloor \;=\; P_n\,.$$

22 The number 111111111111111111 is prime. Prove that, in any radix b, $(11\ldots1)_b$ can be prime only if the number of 1's is prime. *Is this a test for strabismus?*

23 State a recurrence for $\rho(k)$, the ruler function in the text's discussion of $\epsilon_2(n!)$. Show that there's a connection between $\rho(k)$ and the disk that's moved at step k when an n-disk Tower of Hanoi is being transferred in $2^n - 1$ moves, for $1 \leqslant k \leqslant 2^n - 1$.

24 Express $\epsilon_p(n!)$ in terms of $\nu_p(n)$, the sum of the digits in the radix p representation of n, thereby generalizing (4.24). *Look, ma, sideways addition.*

25 We say that m *exactly divides* n, written $m \backslash\backslash n$, if $m \backslash n$ and $m \perp n/m$. For example, in the text's discussion of factorial factors, $p^{\epsilon_p(n!)} \backslash\backslash n!$. Prove or disprove the following:

a $k \backslash\backslash n$ and $m \backslash\backslash n \iff km \backslash\backslash n$, if $k \perp m$.

b For all $m, n > 0$, either $\gcd(m,n) \backslash\backslash m$ or $\gcd(m,n) \backslash\backslash n$.

26 Consider the sequence $\mathcal{G}_N$ of all nonnegative reduced fractions m/n such that $mn \leqslant N$. For example,

$$\mathcal{G}_{10} \;=\; \tfrac{0}{1}, \tfrac{1}{10}, \tfrac{1}{9}, \tfrac{1}{8}, \tfrac{1}{7}, \tfrac{1}{6}, \tfrac{1}{5}, \tfrac{1}{4}, \tfrac{1}{3}, \tfrac{2}{5}, \tfrac{1}{2}, \tfrac{2}{3}, \tfrac{1}{1}, \tfrac{3}{2}, \tfrac{2}{1}, \tfrac{5}{2}, \tfrac{3}{1}, \tfrac{4}{1}, \tfrac{5}{1}, \tfrac{6}{1}, \tfrac{7}{1}, \tfrac{8}{1}, \tfrac{9}{1}, \tfrac{10}{1}.$$

Is it true that $m'n - mn' = 1$ whenever m/n immediately precedes m'/n' in $\mathcal{G}_N$?

27 Give a simple rule for comparing rational numbers based on their representations as L's and R's in the Stern–Brocot number system.

28 The Stern–Brocot representation of π is

$$\pi \;=\; R^3L^7R^{15}LR^{292}LRLR^2LR^3LR^{14}L^2R\ldots;$$

use it to find all the simplest rational approximations to π whose denominators are less than 50. Is $\frac{22}{7}$ one of them?

29 The text describes a correspondence between binary real numbers $x = (.b_1b_2b_3\ldots)_2$ in $[0\,..\,1)$ and Stern–Brocot real numbers $\alpha = B_1B_2B_3\ldots$ in $[0\,..\,\infty)$. If x corresponds to α and $x \neq 0$, what number corresponds to $1 - x$?

30 Prove the following statement (the Chinese Remainder Theorem): Let $m_1, \ldots, m_r$ be integers with $m_j \perp m_k$ for $1 \leqslant j < k \leqslant r$; let $m = m_1 \ldots m_r$; and let $a_1, \ldots, a_r, A$ be integers. Then there is exactly one integer a such that

$$a \equiv a_k \pmod{m_k} \text{ for } 1 \leqslant k \leqslant r \quad\text{and}\quad A \leqslant a < A + m\,.$$

31 A number in decimal notation is divisible by 3 if and only if the sum of its digits is divisible by 3. Prove this well-known rule, and generalize it.

Why is "Euler" pronounced "Oiler" when "Euclid" is "Yooklid"?

32 Prove Euler's theorem (4.50) by generalizing the proof of (4.47).

33 Show that if $f(m)$ and $g(m)$ are multiplicative functions, then so is $h(m) = \sum_{d \backslash m} f(d)\, g(m/d)$.

34 Prove that (4.56) is a special case of (4.61).

Homework exercises

35 Let $I(m,n)$ be a function that satisfies the relation

$$I(m,n)m + I(n,m)n \;=\; \gcd(m,n)\,,$$

when m and n are nonnegative integers with $m \neq n$. Thus, $I(m,n) = m'$ and $I(n,m) = n'$ in (4.5); the value of $I(m,n)$ is an *inverse* of m with respect to n. Find a recurrence that defines $I(m,n)$.

36 Consider the set $Z(\sqrt{10}) = \{\, m + n\sqrt{10} \mid \text{integer } m,n \,\}$. The number $m + n\sqrt{10}$ is called a *unit* if $m^2 - 10n^2 = \pm 1$, since it has an inverse (that is, since $(m+n\sqrt{10}\,)\cdot\pm(m-n\sqrt{10}\,) = 1$). For example, $3+\sqrt{10}$ is a unit, and so is $19-6\sqrt{10}$. Pairs of cancelling units can be inserted into any factorization, so we ignore them. Nonunit numbers of $Z(\sqrt{10}\,)$ are called prime if they cannot be written as a product of two nonunits. Show that 2, 3, and $4 \pm \sqrt{10}$ are primes of $Z(\sqrt{10}\,)$. *Hint:* If $2 = (k + l\sqrt{10}\,) \times (m + n\sqrt{10}\,)$ then $4 = (k^2 - 10l^2)(m^2 - 10n^2)$. Furthermore, the square of any integer mod 10 is 0, 1, 4, 5, 6, or 9.

37 Prove (4.17). *Hint:* Show that $e_n - \frac{1}{2} = (e_{n-1} - \frac{1}{2})^2 + \frac{1}{4}$, and consider $2^{-n}\log(e_n - \frac{1}{2})$.

38 Prove that if $a \perp b$ and $a > b$ then

$$\gcd(a^m - b^m,\, a^n - b^n) \;=\; a^{\gcd(m,n)} - b^{\gcd(m,n)}\,, \qquad 0 \leqslant m < n.$$

(All variables are integers.) *Hint:* Use Euclid's algorithm.

39 Let $S(m)$ be the smallest positive integer n for which there exists an increasing sequence of integers

$$m \;=\; a_1 \;<\; a_2 \;<\; \cdots \;<\; a_t \;=\; n$$

such that $a_1 a_2 \ldots a_t$ is a perfect square. (If m is a perfect square, we can let $t = 1$ and $n = m$.) For example, $S(2) = 6$ because the best such sequence is $a_1 = 2$, $a_2 = 3$, $a_3 = 6$. We have

n	1	2	3	4	5	6	7	8	9	10	11	12
$S(n)$	1	6	8	4	10	12	14	15	9	18	22	20

Prove that $S(m) \neq S(m')$ whenever $0 < m < m'$.

40 If the radix p representation of n is $(a_m \ldots a_1 a_0)_p$, prove that

$$n!/p^{\epsilon_p(n!)} \equiv (-1)^{\epsilon_p(n!)} a_m! \ldots a_1!\, a_0! \qquad (\text{mod } p)\,.$$

(The left side is simply $n!$ with all p factors removed. When $n = p$ this reduces to Wilson's theorem.)

Wilson's theorem: "Martha, that boy is a menace."

41 **a** Show that if $p \bmod 4 = 3$, there is no integer n such that p divides $n^2 + 1$. *Hint:* Use Fermat's theorem.

b But show that if $p \bmod 4 = 1$, there is such an integer. *Hint:* Write $(p-1)!$ as $\left(\prod_{k=1}^{(p-1)/2} k(p-k)\right)$ and think about Wilson's theorem.

42 Consider two fractions m/n and m'/n' in lowest terms. Prove that when the sum $m/n + m'/n'$ is reduced to lowest terms, the denominator will be nn' if and only if $n \perp n'$. (In other words, $(mn' + m'n)/nn'$ will already be in lowest terms if and only if n and n' have no common factor.)

43 There are 2^k nodes at level k of the Stern–Brocot tree, corresponding to the matrices L^k, $L^{k-1}R$, ..., R^k. Show that this sequence can be obtained by starting with L^k and then multiplying successively by

$$\begin{pmatrix} 0 & -1 \\ 1 & 2\rho(n)+1 \end{pmatrix}$$

for $1 \leqslant n < 2^k$, where $\rho(n)$ is the ruler function.

Radio announcer: "... pitcher Mark LeChiffre hits a two-run single! Mark, who was batting .080, gets his second hit of the year." Anything wrong?

44 Prove that a baseball player whose batting average is .316 must have batted at least 19 times. (If he has m hits in n times at bat, then $m/n \in [0.3155\,..\,0.3165)$.)

45 The number 9376 has the peculiar self-reproducing property that

$$9376^2 = 87909376\,.$$

How many 4-digit numbers x satisfy the equation $x^2 \bmod 10000 = x$? How many n-digit numbers x satisfy the equation $x^2 \bmod 10^n = x$?

46 **a** Prove that if $n^j \equiv 1$ and $n^k \equiv 1 \pmod m$, then $n^{\gcd(j,k)} \equiv 1$.

b Show that $2^n \not\equiv 1 \pmod n$, if $n > 1$. *Hint:* Consider the least prime factor of n.

47 Show that if $n^{m-1} \equiv 1 \pmod m$ and if $n^{(m-1)/p} \not\equiv 1 \pmod m$ for all primes such that $p \backslash (m-1)$, then m is prime. *Hint:* Show that if this condition holds, the numbers $n^k \bmod m$ are distinct, for $1 \leqslant k < m$.

48 Generalize Wilson's theorem (4.49) by ascertaining the value of the expression $\left(\prod_{1 \leqslant n < m,\, n \perp m} n\right) \bmod m$, when $m > 1$.

49 Let $R(N)$ be the number of pairs of integers (m, n) such that $0 \leqslant m < N$, $0 \leqslant n < N$, and $m \perp n$.

a Express $R(N)$ in terms of the Φ function.

b Prove that $R(N) = \sum_{d \geqslant 1} \lfloor N/d \rfloor^2 \mu(d)$.

50 Let m be a positive integer and let

$$\omega \;=\; e^{2\pi i/m} \;=\; \cos(2\pi/m) + i\sin(2\pi/m)\,.$$

What are the roots of disunity?

We say that ω is an mth *root of unity*, since $\omega^m = e^{2\pi i} = 1$. In fact, each of the m complex numbers ω^0, ω^1, $\ldots$, ω^{m-1} is an mth root of unity, because $(\omega^k)^m = e^{2\pi k i} = 1$; therefore $z - \omega^k$ is a factor of the polynomial $z^m - 1$, for $0 \leqslant k < m$. Since these factors are distinct, the complete factorization of $z^m - 1$ over the complex numbers must be

$$z^m - 1 \;=\; \prod_{0 \leqslant k < m} (z - \omega^k)\,.$$

a Let $\Psi_m(z) = \prod_{0 \leqslant k < m,\, k \perp m} (z - \omega^k)$. (This polynomial of degree $\varphi(m)$ is called the *cyclotomic polynomial of order* m.) Prove that

$$z^m - 1 \;=\; \prod_{d \backslash m} \Psi_d(z)\,.$$

b Prove that $\Psi_m(z) = \prod_{d \backslash m} (z^d - 1)^{\mu(m/d)}$.

Exam problems

51 Prove Fermat's theorem (4.48) by expanding $(1 + 1 + \cdots + 1)^p$ via the multinomial theorem.

52 Let n and x be positive integers such that x has no divisors $\leqslant n$ (except 1), and let p be a prime number. Prove that at least $\lfloor n/p \rfloor$ of the numbers $\{x - 1, x^2 - 1, \ldots, x^{n-1} - 1\}$ are multiples of p.

53 Find all positive integers n such that $n \backslash \lfloor (n-1)!/(n+1) \rfloor$.

54 Determine the value of $1000! \bmod 10^{250}$ by hand calculation.

55 Let P_n be the product of the first n factorials, $\prod_{k=1}^{n} k!$. Prove that P_{2n}/P_n^4 is an integer, for all positive integers n.

56 Show that

$$\left(\prod_{k=1}^{2n-1} k^{\min(k,\,2n-k)}\right) \Big/ \left(\prod_{k=1}^{n-1} (2k+1)^{2n-2k-1}\right)$$

is a power of 2.

57 Let $S(m,n)$ be the set of all integers k such that

$$m \bmod k + n \bmod k \;\geqslant\; k\,.$$

For example, $S(7,9) = \{2,4,5,8,10,11,12,13,14,15,16\}$. Prove that

$$\sum_{k\in S(m,n)} \varphi(k) \;=\; mn\,.$$

Hint: Prove first that $\sum_{1\leqslant m\leqslant n}\sum_{d\backslash m}\varphi(d) = \sum_{d\geqslant 1}\varphi(d)\lfloor n/d\rfloor$. Then consider $\lfloor (m+n)/d\rfloor - \lfloor m/d\rfloor - \lfloor n/d\rfloor$.

58 Let $f(m) = \sum_{d\backslash m} d$. Find a necessary and sufficient condition that $f(m)$ is a power of 2.

Bonus problems

59 Prove that if $x_1, \ldots, x_n$ are positive integers with $1/x_1 + \cdots + 1/x_n = 1$, then $\max(x_1,\ldots,x_n) < e_n$. *Hint:* Prove the following stronger result by induction: "If $1/x_1 + \cdots + 1/x_n + 1/\alpha = 1$, where $x_1, \ldots, x_n$ are positive integers and α is a rational number $\geqslant \max(x_1,\ldots,x_n)$, then $\alpha+1 \leqslant e_{n+1}$ and $x_1\ldots x_n(\alpha+1) \leqslant e_1\ldots e_n e_{n+1}$." (The proof is nontrivial.)

60 Prove that there's a constant P such that (4.18) gives only primes. You may use the following (highly nontrivial) fact: There is a prime between p and $p + p^\theta$, for all sufficiently large p, if $\theta > \frac{6}{11}$.

61 Prove that if m/n, m'/n', and m''/n'' are consecutive elements of $\mathcal{F}_N$, then

$$\begin{aligned} m'' &= \lfloor (n+N)/n' \rfloor m' - m\,, \\ n'' &= \lfloor (n+N)/n' \rfloor n' - n\,. \end{aligned}$$

(This recurrence allows us to compute the elements of $\mathcal{F}_N$ in order, starting with $\frac{0}{1}$ and $\frac{1}{N}$.)

62 What binary number corresponds to e, in the binary $\leftrightarrow$ Stern–Brocot correspondence? (Express your answer as an infinite sum; you need not evaluate it in closed form.)

63 Using only the methods of this chapter, show that if Fermat's Last Theorem (4.46) were false, the least n for which it fails would have to be prime. (You may assume that (4.46) holds when $n = 4$.) Furthermore, if $a^p + b^p = c^p$ is the smallest counterexample, show that

$$a + b \;=\; \begin{cases} m^p, & \text{if } p\not\backslash c, \\ p^{p-1}m^p, & \text{if } p\backslash c, \end{cases}$$

for some integer m. Thus $c \geqslant m^p/2$ must be really huge. *Hint:* Let $x = a + b$, and note that $\gcd\bigl(x, (a^p + (x-a)^p)/x\bigr) = \gcd(x, pa^{p-1})$.

64 The *Peirce sequence* $\mathcal{P}_N$ *of order* N is an infinite string of fractions separated by '$<$' or '$=$' signs, containing all the nonnegative fractions m/n with $m \geqslant 0$ and $n \leqslant N$ (including fractions that are not reduced). It is defined recursively by starting with

$$\mathcal{P}_1 \;=\; \tfrac{0}{1}<\tfrac{1}{1}<\tfrac{2}{1}<\tfrac{3}{1}<\tfrac{4}{1}<\tfrac{5}{1}<\tfrac{6}{1}<\tfrac{7}{1}<\tfrac{8}{1}<\tfrac{9}{1}<\tfrac{10}{1}<\cdots.$$

For $N \geqslant 1$, we form $\mathcal{P}_{N+1}$ by inserting two symbols just before the kNth symbol of $\mathcal{P}_N$, for all $k > 0$. The two inserted symbols are

$$\begin{array}{lll} \dfrac{k-1}{N+1} & = , & \text{if } kN \text{ is odd;} \\ \mathcal{P}_{N,kN} & \dfrac{k-1}{N+1}, & \text{if } kN \text{ is even.} \end{array}$$

Here $\mathcal{P}_{N,j}$ denotes the jth symbol of $\mathcal{P}_N$, which will be either '$<$' or '$=$' when j is even; it will be a fraction when j is odd. For example,

$$\begin{aligned}
\mathcal{P}_2 &= \tfrac{0}{2}=\tfrac{0}{1}<\tfrac{1}{2}<\tfrac{2}{2}=\tfrac{1}{1}<\tfrac{3}{2}<\tfrac{4}{2}=\tfrac{2}{1}<\tfrac{5}{2}<\tfrac{6}{2}=\tfrac{3}{1}<\tfrac{7}{2}<\tfrac{8}{2}=\tfrac{4}{1}<\tfrac{9}{2}<\tfrac{10}{2}=\tfrac{5}{1}<\cdots;\\
\mathcal{P}_3 &= \tfrac{0}{2}=\tfrac{0}{3}=\tfrac{0}{1}<\tfrac{1}{3}<\tfrac{1}{2}<\tfrac{2}{3}<\tfrac{2}{2}=\tfrac{3}{3}=\tfrac{1}{1}<\tfrac{4}{3}<\tfrac{3}{2}<\tfrac{5}{3}<\tfrac{4}{2}=\tfrac{6}{3}=\tfrac{2}{1}<\tfrac{7}{3}<\tfrac{5}{2}<\cdots;\\
\mathcal{P}_4 &= \tfrac{0}{2}=\tfrac{0}{4}=\tfrac{0}{3}=\tfrac{0}{1}<\tfrac{1}{4}<\tfrac{1}{3}<\tfrac{2}{4}=\tfrac{1}{2}<\tfrac{2}{3}<\tfrac{3}{4}<\tfrac{2}{2}=\tfrac{4}{4}=\tfrac{3}{3}=\tfrac{1}{1}<\tfrac{5}{4}<\tfrac{4}{3}<\tfrac{6}{4}=\cdots;\\
\mathcal{P}_5 &= \tfrac{0}{2}=\tfrac{0}{4}=\tfrac{0}{5}=\tfrac{0}{3}=\tfrac{0}{1}<\tfrac{1}{5}<\tfrac{1}{4}<\tfrac{1}{3}<\tfrac{2}{5}<\tfrac{2}{4}=\tfrac{1}{2}<\tfrac{2}{5}<\tfrac{2}{3}<\tfrac{3}{4}<\tfrac{4}{5}<\tfrac{2}{2}=\tfrac{4}{4}=\cdots;\\
\mathcal{P}_6 &= \tfrac{0}{2}=\tfrac{0}{4}=\tfrac{0}{6}=\tfrac{0}{5}=\tfrac{0}{3}=\tfrac{0}{1}<\tfrac{1}{6}<\tfrac{1}{5}<\tfrac{1}{4}<\tfrac{2}{6}=\tfrac{1}{3}<\tfrac{2}{5}<\tfrac{2}{4}=\tfrac{3}{6}=\tfrac{1}{2}<\tfrac{3}{5}<\tfrac{4}{6}=\cdots.
\end{aligned}$$

(Equal elements occur in a slightly peculiar order.) Prove that the '$<$' and '$=$' signs defined by the rules above correctly describe the relations between adjacent fractions in the Peirce sequence.

Research problems

65 Are the Euclid numbers e_n all squarefree?

66 Are the Mersenne numbers $2^p - 1$ all squarefree?

67 Prove or disprove that $\max_{1\leqslant j<k\leqslant n} a_k/\gcd(a_j, a_k) \geqslant n$, for all sequences of integers $0 < a_1 < \cdots < a_n$.

68 Is there a constant Q such that $\lfloor Q^{2^n} \rfloor$ is prime for all $n \geqslant 0$?

69 Let P_n denote the nth prime. Prove or disprove that $P_{n+1} - P_n = O(\log P_n)^2$.

70 Does $\epsilon_3(n!) = \epsilon_2(n!)/2$ for infinitely many n?

71 Prove or disprove: If $k \neq 1$ there exists $n > 1$ such that $2^n \equiv k \pmod{n}$. Are there infinitely many such n?

72 Prove or disprove: For all integers a, there exist infinitely many n such that $\varphi(n)\backslash(n + a)$.

73 If the $\Phi(n)+1$ terms of the Farey series

$$\mathcal{F}_n \;=\; \bigl\langle \mathcal{F}_n(0), \mathcal{F}_n(1), \ldots, \mathcal{F}_n\bigl(\Phi(n)\bigr)\bigr\rangle$$

were fairly evenly distributed, we would expect $\mathcal{F}_n(k) \approx k/\Phi(n)$. Therefore the sum $D(n) = \sum_{k=0}^{\Phi(n)} \bigl|\mathcal{F}_n(k) - k/\Phi(n)\bigr|$ measures the "deviation of $\mathcal{F}_n$ from uniformity." Is it true that $D(n) = O(n^{1/2+\epsilon})$ for all $\epsilon > 0$?

74 Approximately how many distinct values are there in the set $\{0! \bmod p, 1! \bmod p, \ldots, (p-1)! \bmod p\}$, as $p \to \infty$?

5

Binomial Coefficients

LET'S TAKE A BREATHER. The previous chapters have seen some heavy going, with sums involving floor, ceiling, mod, phi, and mu functions. Now we're going to study binomial coefficients, which turn out to be (a) more important in applications, and (b) easier to manipulate, than all those other quantities.

Lucky us!

5.1 BASIC IDENTITIES

The symbol $\binom{n}{k}$ is a binomial coefficient, so called because of an important property we look at later this section, the binomial theorem. But we read the symbol "n choose k." This incantation arises from its combinatorial interpretation — it is the number of ways to choose a k-element subset from an n-element set. For example, from the set $\{1,2,3,4\}$ we can choose two elements in six ways,

Otherwise known as combinations of n things, k at a time.

$$\{1,2\}, \quad \{1,3\}, \quad \{1,4\}, \quad \{2,3\}, \quad \{2,4\}, \quad \{3,4\};$$

so $\binom{4}{2} = 6$.

To express the number $\binom{n}{k}$ in more familiar terms it's easiest to first determine the number of k-element *sequences*, rather than subsets, chosen from an n-element set; for sequences, the order of the elements counts. We use the same argument we used in Chapter 4 to show that $n!$ is the number of permutations of n objects. There are n choices for the first element of the sequence; for each, there are $n-1$ choices for the second; and so on, until there are $n-k+1$ choices for the kth. This gives $n(n-1)\ldots(n-k+1) = n^{\underline{k}}$ choices in all. And since each k-element subset has exactly $k!$ different orderings, this number of *sequences* counts each *subset* exactly $k!$ times. To get our answer, we simply divide by $k!$:

$$\binom{n}{k} = \frac{n(n-1)\ldots(n-k+1)}{k(k-1)\ldots(1)}.$$

For example,

$$\binom{4}{2} = \frac{4\cdot 3}{2\cdot 1} = 6;$$

this agrees with our previous enumeration.

We call n the *upper index* and k the *lower index*. The indices are restricted to be nonnegative integers by the combinatorial interpretation, because sets don't have negative or fractional numbers of elements. But the binomial coefficient has many uses besides its combinatorial interpretation, so we will remove some of the restrictions. It's most useful, it turns out, to allow an arbitrary real (or even complex) number to appear in the upper index, and to allow an arbitrary integer in the lower. Our formal definition therefore takes the following form:

$$\binom{r}{k} = \begin{cases} \dfrac{r(r-1)\ldots(r-k+1)}{k(k-1)\ldots(1)} = \dfrac{r^{\underline{k}}}{k!}, & \text{integer } k \geqslant 0; \\ 0, & \text{integer } k < 0. \end{cases} \tag{5.1}$$

This definition has several noteworthy features. First, the upper index is called r, not n; the letter r emphasizes the fact that binomial coefficients make sense when any real number appears in this position. For instance, we have $\binom{-1}{3} = (-1)(-2)(-3)/(3\cdot 2\cdot 1) = -1$. There's no combinatorial interpretation here, but $r = -1$ turns out to be an important special case. A noninteger index like $r = -1/2$ also turns out to be useful.

Second, we can view $\binom{r}{k}$ as a kth-degree polynomial in r. We'll see that this viewpoint is often helpful.

Third, we haven't defined binomial coefficients for noninteger lower indices. A reasonable definition can be given, but actual applications are rare, so we will defer this generalization to later in the chapter.

Final note: We've listed the restrictions 'integer $k \geqslant 0$' and 'integer $k < 0$' at the right of the definition. Such restrictions will be listed in all the identities we will study, so that the range of applicability will be clear. In general the fewer restrictions the better, because an unrestricted identity is most useful; still, any restrictions that apply are an important part of the identity. When we manipulate binomial coefficients, it's easier to ignore difficult-to-remember restrictions temporarily and to check later that nothing has been violated. But the check needs to be made.

For example, almost every time we encounter $\binom{n}{n}$ it equals 1, so we can get lulled into thinking that it's always 1. But a careful look at definition (5.1) tells us that $\binom{n}{n}$ is 1 only when $n \geqslant 0$ (assuming that n is an integer); when $n < 0$ we have $\binom{n}{n} = 0$. Traps like this can (and will) make life adventuresome.

Before getting to the identities that we will use to tame binomial coefficients, let's take a peek at some small values. The numbers in Table 155 form the beginning of *Pascal's triangle*, named after Blaise Pascal (1623–1662)

Table 155 Pascal's triangle.

n	$\binom{n}{0}$	$\binom{n}{1}$	$\binom{n}{2}$	$\binom{n}{3}$	$\binom{n}{4}$	$\binom{n}{5}$	$\binom{n}{6}$	$\binom{n}{7}$	$\binom{n}{8}$	$\binom{n}{9}$	$\binom{n}{10}$
0	1										
1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1

Binomial coefficients were well known in Asia, many centuries before Pascal was born [90], but he had no way to know that.

because he wrote an influential treatise about them [285]. The empty entries in this table are actually 0's, because of a zero in the numerator of (5.1); for example, $\binom{1}{2} = (1\cdot 0)/(2\cdot 1) = 0$. These entries have been left blank simply to help emphasize the rest of the table.

It's worthwhile to memorize formulas for the first three columns,

$$\binom{r}{0} = 1\,, \qquad \binom{r}{1} = r\,, \qquad \binom{r}{2} = \frac{r(r-1)}{2}\,; \tag{5.2}$$

these hold for arbitrary reals. (Recall that $\binom{n+1}{2} = \frac{1}{2}n(n+1)$ is the formula we derived for triangular numbers in Chapter 1; triangular numbers are conspicuously present in the $\binom{n}{2}$ column of Table 155.) It's also a good idea to memorize the first five rows or so of Pascal's triangle, so that when the pattern 1, 4, 6, 4, 1 appears in some problem we will have a clue that binomial coefficients probably lurk nearby.

In Italy it's called Tartaglia's triangle.

The numbers in Pascal's triangle satisfy, practically speaking, infinitely many identities, so it's not too surprising that we can find some surprising relationships by looking closely. For example, there's a curious "hexagon property," illustrated by the six numbers 56, 28, 36, 120, 210, 126 that surround 84 in the lower right portion of Table 155. Both ways of multiplying alternate numbers from this hexagon give the same product: $56\cdot 36\cdot 210 = 28\cdot 120\cdot 126 = 423360$. The same thing holds if we extract such a hexagon from any other part of Pascal's triangle.

And now the identities. Our goal in this section will be to learn a few simple rules by which we can solve the vast majority of practical problems involving binomial coefficients.

"C'est une chose estrange combien il est fertile en proprietez."
—B. Pascal [285]

Definition (5.1) can be recast in terms of factorials in the common case that the upper index r is an integer, n, that's greater than or equal to the lower index k:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}, \qquad \text{integers } n \geqslant k \geqslant 0. \tag{5.3}$$

To get this formula, we just multiply the numerator and denominator of (5.1) by $(n-k)!$. It's occasionally useful to expand a binomial coefficient into this factorial form (for example, when proving the hexagon property). And we often want to go the other way, changing factorials into binomials.

The factorial representation hints at a symmetry in Pascal's triangle: Each row reads the same left-to-right as right-to-left. The identity reflecting this — called the *symmetry* identity — is obtained by changing k to $n-k$:

$$\binom{n}{k} = \binom{n}{n-k}, \qquad \begin{array}{c}\text{integer } n \geqslant 0,\\ \text{integer } k.\end{array} \tag{5.4}$$

This formula makes combinatorial sense, because by specifying the k chosen things out of n we're in effect specifying the $n-k$ unchosen things.

The restriction that n and k be integers in identity (5.4) is obvious, since each lower index must be an integer. But why can't n be negative? Suppose, for example, that $n = -1$. Is

$$\binom{-1}{k} \stackrel{?}{=} \binom{-1}{-1-k}$$

a valid equation? No. For instance, when $k = 0$ we get 1 on the left and 0 on the right. In fact, for any integer $k \geqslant 0$ the left side is

$$\binom{-1}{k} = \frac{(-1)(-2)\ldots(-k)}{k!} = (-1)^k,$$

which is either 1 or -1; but the right side is 0, because the lower index is negative. And for negative k the left side is 0 but the right side is

$$\binom{-1}{-1-k} = (-1)^{-1-k},$$

which is either 1 or -1. So the equation '$\binom{-1}{k} = \binom{-1}{-1-k}$' is always false!

The symmetry identity fails for all other negative integers n, too. But unfortunately it's all too easy to forget this restriction, since the expression in the upper index is sometimes negative only for obscure (but legal) values

I just hope I don't fall into this trap during the midterm.

of its variables. Everyone who's manipulated binomial coefficients much has fallen into this trap at least three times.

But the symmetry identity does have a big redeeming feature: It works for all values of k, even when $k < 0$ or $k > n$. (Because both sides are zero in such cases.) Otherwise $0 \leqslant k \leqslant n$, and symmetry follows immediately from (5.3):

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} = \frac{n!}{(n-(n-k))!\;(n-k)!} = \binom{n}{n-k}.$$

Our next important identity lets us move things in and out of binomial coefficients:

$$\binom{r}{k} = \frac{r}{k}\binom{r-1}{k-1}, \qquad \text{integer } k \neq 0. \tag{5.5}$$

The restriction on k prevents us from dividing by 0 here. We call (5.5) an *absorption* identity, because we often use it to absorb a variable into a binomial coefficient when that variable is a nuisance outside. The equation follows from definition (5.1), because $r^{\underline{k}} = r(r-1)^{\underline{k-1}}$ and $k! = k(k-1)!$ when $k > 0$; both sides are zero when $k < 0$.

If we multiply both sides of (5.5) by k, we get an absorption identity that works even when $k = 0$:

$$k\binom{r}{k} = r\binom{r-1}{k-1}, \qquad \text{integer } k. \tag{5.6}$$

This one also has a companion that keeps the lower index intact:

$$(r-k)\binom{r}{k} = r\binom{r-1}{k}, \qquad \text{integer } k. \tag{5.7}$$

We can derive (5.7) by sandwiching an application of (5.6) between two applications of symmetry:

$$\begin{aligned}
(r-k)\binom{r}{k} &= (r-k)\binom{r}{r-k} && \text{(by symmetry)}\\
&= r\binom{r-1}{r-k-1} && \text{(by (5.6))}\\
&= r\binom{r-1}{k}. && \text{(by symmetry)}
\end{aligned}$$

But wait a minute. We've claimed that the identity holds for *all* real r, yet the derivation we just gave holds only when r is a positive integer. (The upper index $r-1$ must be a nonnegative integer if we're to use the symmetry

property (5.4) with impunity.) Have we been cheating? No. It's true that the derivation is valid only for positive integers r; but we can claim that the identity holds for all values of r, because both sides of (5.7) are polynomials in r of degree $k+1$. A nonzero polynomial of degree d or less can have at most d distinct zeros; therefore the difference of two such polynomials, which also has degree d or less, cannot be zero at more than d points unless it is identically zero. In other words, if two polynomials of degree d or less agree at more than d points, they must agree everywhere. We have shown that $(r-k)\binom{r}{k} = r\binom{r-1}{k}$ whenever r is a positive integer; so these two polynomials agree at infinitely many points, and they must be identically equal.

(Well, not here anyway.)

The proof technique in the previous paragraph, which we will call the *polynomial argument*, is useful for extending many identities from integers to reals; we'll see it again and again. Some equations, like the symmetry identity (5.4), are not identities between polynomials, so we can't always use this method. But many identities do have the necessary form.

For example, here's another polynomial identity, perhaps the most important binomial identity of all, known as the *addition formula*:

$$\binom{r}{k} = \binom{r-1}{k} + \binom{r-1}{k-1}, \qquad \text{integer } k. \tag{5.8}$$

When r is a positive integer, the addition formula tells us that every number in Pascal's triangle is the sum of two numbers in the previous row, one directly above it and the other just to the left. And the formula applies also when r is negative, real, or complex; the only restriction is that k be an integer, so that the binomial coefficients are defined.

One way to prove the addition formula is to assume that r is a positive integer and to use the combinatorial interpretation. Recall that $\binom{r}{k}$ is the number of possible k-element subsets chosen from an r-element set. If we have a set of r eggs that includes exactly one bad egg, there are $\binom{r}{k}$ ways to select k of the eggs. Exactly $\binom{r-1}{k}$ of these selections involve nothing but good eggs; and $\binom{r-1}{k-1}$ of them contain the bad egg, because such selections have $k-1$ of the $r-1$ good eggs. Adding these two numbers together gives (5.8). This derivation assumes that r is a positive integer, and that $k \geqslant 0$. But both sides of the identity are zero when $k < 0$, and the polynomial argument establishes (5.8) in all remaining cases.

We can also derive (5.8) by adding together the two absorption identities (5.7) and (5.6):

$$(r-k)\binom{r}{k} + k\binom{r}{k} = r\binom{r-1}{k} + r\binom{r-1}{k-1};$$

the left side is $r\binom{r}{k}$, and we can divide through by r. This derivation is valid for everything but $r = 0$, and it's easy to check that remaining case.

Those of us who tend not to discover such slick proofs, or who are otherwise into tedium, might prefer to derive (5.8) by a straightforward manipulation of the definition. If $k > 0$,

$$\begin{aligned}\binom{r-1}{k} + \binom{r-1}{k-1} &= \frac{(r-1)^{\underline{k}}}{k!} + \frac{(r-1)^{\underline{k-1}}}{(k-1)!} \\ &= \frac{(r-1)^{\underline{k-1}}(r-k)}{k!} + \frac{(r-1)^{\underline{k-1}}k}{k!} \\ &= \frac{(r-1)^{\underline{k-1}}r}{k!} = \frac{r^{\underline{k}}}{k!} = \binom{r}{k}.\end{aligned}$$

Again, the cases for $k \leqslant 0$ are easy to handle.

We've just seen three rather different proofs of the addition formula. This is not surprising; binomial coefficients have many useful properties, several of which are bound to lead to proofs of an identity at hand.

The addition formula is essentially a recurrence for the numbers of Pascal's triangle, so we'll see that it is especially useful for proving other identities by induction. We can also get a new identity immediately by unfolding the recurrence. For example,

$$\begin{aligned}\binom{5}{3} &= \binom{4}{3} + \binom{4}{2} \\ &= \binom{4}{3} + \binom{3}{2} + \binom{3}{1} \\ &= \binom{4}{3} + \binom{3}{2} + \binom{2}{1} + \binom{2}{0} \\ &= \binom{4}{3} + \binom{3}{2} + \binom{2}{1} + \binom{1}{0} + \binom{1}{-1}.\end{aligned}$$

Since $\binom{1}{-1} = 0$, that term disappears and we can stop. This method yields the general formula

$$\begin{aligned}\sum_{k \leqslant n} \binom{r+k}{k} &= \binom{r}{0} + \binom{r+1}{1} + \cdots + \binom{r+n}{n} \\ &= \binom{r+n+1}{n}, \qquad \text{integer } n. \end{aligned} \tag{5.9}$$

Notice that we don't need the lower limit $k \geqslant 0$ on the index of summation, because the terms with $k < 0$ are zero.

This formula expresses one binomial coefficient as the sum of others whose upper and lower indices stay the same distance apart. We found it by repeatedly expanding the binomial coefficient with the smallest lower index: first

$\binom{5}{3}$, then $\binom{4}{2}$, then $\binom{3}{1}$, then $\binom{2}{0}$. What happens if we unfold the other way, repeatedly expanding the one with largest lower index? We get

$$\begin{aligned}
\binom{5}{3} &= \binom{4}{3} + \binom{4}{2} \\
&= \binom{3}{3} + \binom{3}{2} + \binom{4}{2} \\
&= \binom{2}{3} + \binom{2}{2} + \binom{3}{2} + \binom{4}{2} \\
&= \binom{1}{3} + \binom{1}{2} + \binom{2}{2} + \binom{3}{2} + \binom{4}{2} \\
&= \binom{0}{3} + \binom{0}{2} + \binom{1}{2} + \binom{2}{2} + \binom{3}{2} + \binom{4}{2}.
\end{aligned}$$

Now $\binom{0}{3}$ is zero (so are $\binom{0}{2}$ and $\binom{1}{2}$, but these make the identity nicer), and we can spot the general pattern:

$$\begin{aligned}
\sum_{0 \leqslant k \leqslant n} \binom{k}{m} &= \binom{0}{m} + \binom{1}{m} + \cdots + \binom{n}{m} \\
&= \binom{n+1}{m+1}, \qquad \text{integers } m, n \geqslant 0. \qquad (5.10)
\end{aligned}$$

This identity, which we call *summation on the upper index*, expresses a binomial coefficient as the sum of others whose lower indices are constant. In this case the sum needs the lower limit $k \geqslant 0$, because the terms with $k < 0$ *aren't* zero. Also, m and n can't in general be negative.

Identity (5.10) has an interesting combinatorial interpretation. If we want to choose $m+1$ tickets from a set of $n+1$ tickets numbered 0 through n, there are $\binom{k}{m}$ ways to do this when the largest ticket selected is number k.

We can prove both (5.9) and (5.10) by induction using the addition formula, but we can also prove them from each other. For example, let's prove (5.9) from (5.10); our proof will illustrate some common binomial coefficient manipulations. Our general plan will be to massage the left side $\sum \binom{r+k}{k}$ of (5.9) so that it looks like the left side $\sum \binom{k}{m}$ of (5.10); then we'll invoke that identity, replacing the sum by a single binomial coefficient; finally we'll transform that coefficient into the right side of (5.9).

We can assume for convenience that r and n are nonnegative integers; the general case of (5.9) follows from this special case, by the polynomial argument. Let's write m instead of r, so that this variable looks more like a nonnegative integer. The plan can now be carried out systematically as

follows:

$$\begin{aligned}
\sum_{k\leqslant n}\binom{m+k}{k} &= \sum_{-m\leqslant k\leqslant n}\binom{m+k}{k} \\
&= \sum_{-m\leqslant k\leqslant n}\binom{m+k}{m} \\
&= \sum_{0\leqslant k\leqslant m+n}\binom{k}{m} \\
&= \binom{m+n+1}{m+1} \quad = \quad \binom{m+n+1}{n}.
\end{aligned}$$

Let's look at this derivation blow by blow. The key step is in the second line, where we apply the symmetry law (5.4) to replace $\binom{m+k}{k}$ by $\binom{m+k}{m}$. We're allowed to do this only when $m+k \geqslant 0$, so our first step restricts the range of k by discarding the terms with $k < -m$. (This is legal because those terms are zero.) Now we're almost ready to apply (5.10); the third line sets this up, replacing k by $k-m$ and tidying up the range of summation. This step, like the first, merely plays around with $\sum$-notation. Now k appears by itself in the upper index and the limits of summation are in the proper form, so the fourth line applies (5.10). One more use of symmetry finishes the job.

Certain sums that we did in Chapters 1 and 2 were actually special cases of (5.10), or disguised versions of this identity. For example, the case $m = 1$ gives the sum of the nonnegative integers up through n:

$$\binom{0}{1}+\binom{1}{1}+\cdots+\binom{n}{1} = 0+1+\cdots+n = \frac{(n+1)n}{2} = \binom{n+1}{2}.$$

And the general case is equivalent to Chapter 2's rule

$$\sum_{0\leqslant k\leqslant n} k^{\underline{m}} = \frac{(n+1)^{\underline{m+1}}}{m+1}, \qquad \text{integers } m, n \geqslant 0,$$

if we divide both sides of this formula by $m!$. In fact, the addition formula (5.8) tells us that

$$\Delta\left(\binom{x}{m}\right) = \binom{x+1}{m} - \binom{x}{m} = \binom{x}{m-1},$$

if we replace r and k respectively by $x+1$ and m. Hence the methods of Chapter 2 give us the handy indefinite summation formula

$$\sum \binom{x}{m}\,\delta x = \binom{x}{m+1} + C. \tag{5.11}$$

Binomial coefficients get their name from the *binomial theorem*, which deals with powers of the binomial expression $x+y$. Let's look at the smallest cases of this theorem:

"At the age of twenty-one he [Moriarty] wrote a treatise upon the Binomial Theorem, which has had a European vogue. On the strength of it, he won the Mathematical Chair at one of our smaller Universities."
—S. Holmes [84]

$$\begin{aligned}
(x+y)^0 &= 1x^0y^0 \\
(x+y)^1 &= 1x^1y^0 + 1x^0y^1 \\
(x+y)^2 &= 1x^2y^0 + 2x^1y^1 + 1x^0y^2 \\
(x+y)^3 &= 1x^3y^0 + 3x^2y^1 + 3x^1y^2 + 1x^0y^3 \\
(x+y)^4 &= 1x^4y^0 + 4x^3y^1 + 6x^2y^2 + 4x^1y^3 + 1x^0y^4 \,.
\end{aligned}$$

It's not hard to see why these coefficients are the same as the numbers in Pascal's triangle: When we expand the product

$$(x+y)^n = \overbrace{(x+y)(x+y)\ldots(x+y)}^{n \text{ factors}},$$

every term is itself the product of n factors, each either an x or y. The number of such terms with k factors of x and $n-k$ factors of y is the coefficient of $x^k y^{n-k}$ after we combine like terms. And this is exactly the number of ways to choose k of the n binomials from which an x will be contributed; that is, it's $\binom{n}{k}$.

Some textbooks leave the quantity 0^0 undefined, because the functions x^0 and 0^x have different limiting values when x decreases to 0. But this is a mistake. We must define

$$x^0 = 1\,, \qquad \text{for all } x,$$

if the binomial theorem is to be valid when $x=0$, $y=0$, and/or $x=-y$. The theorem is too important to be arbitrarily restricted! By contrast, the function 0^x is quite unimportant. (See [220] for further discussion.)

But what exactly is the binomial theorem? In its full glory it is the following identity:

$$(x+y)^r = \sum_k \binom{r}{k} x^k y^{r-k}\,, \qquad \begin{array}{l}\text{integer } r \geqslant 0 \\ \text{or } |x/y| < 1.\end{array} \tag{5.12}$$

The sum is over all integers k; but it is really a finite sum when r is a nonnegative integer, because all terms are zero except those with $0 \leqslant k \leqslant r$. On the other hand, the theorem is also valid when r is negative, or even when r is an arbitrary real or complex number. In such cases the sum really is infinite, and we must have $|x/y| < 1$ to guarantee the sum's absolute convergence.

Two special cases of the binomial theorem are worth special attention, even though they are extremely simple. If $x = y = 1$ and $r = n$ is nonnegative, we get

$$2^n = \binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n}, \qquad \text{integer } n \geqslant 0.$$

This equation tells us that row n of Pascal's triangle sums to 2^n. And when x is -1 instead of $+1$, we get

$$0^n = \binom{n}{0} - \binom{n}{1} + \cdots + (-1)^n \binom{n}{n}, \qquad \text{integer } n \geqslant 0.$$

For example, $1 - 4 + 6 - 4 + 1 = 0$; the elements of row n sum to zero if we give them alternating signs, except in the top row (when $n = 0$ and $0^0 = 1$).

When r is not a nonnegative integer, we most often use the binomial theorem in the special case $y = 1$. Let's state this special case explicitly, writing z instead of x to emphasize the fact that an arbitrary complex number can be involved here:

$$(1+z)^r = \sum_k \binom{r}{k} z^k, \qquad |z| < 1. \tag{5.13}$$

The general formula in (5.12) follows from this one if we set $z = x/y$ and multiply both sides by y^r.

We have proved the binomial theorem only when r is a nonnegative integer, by using a combinatorial interpretation. We can't deduce the general case from the nonnegative-integer case by using the polynomial argument, because the sum is infinite in the general case. But when r is arbitrary, we can use Taylor series and the theory of complex variables:

$$\begin{aligned} f(z) &= \frac{f(0)}{0!} z^0 + \frac{f'(0)}{1!} z^1 + \frac{f''(0)}{2!} z^2 + \cdots \\ &= \sum_{k \geqslant 0} \frac{f^{(k)}(0)}{k!} z^k. \end{aligned}$$

The derivatives of the function $f(z) = (1+z)^r$ are easily evaluated; in fact, $f^{(k)}(z) = r^{\underline{k}}(1+z)^{r-k}$. Setting $z = 0$ gives (5.13).

(Chapter 9 tells the meaning of O*.)*

We also need to prove that the infinite sum converges, when $|z| < 1$. It does, because $\binom{r}{k} = O(k^{-1-r})$ by equation (5.83) below.

Now let's look more closely at the values of $\binom{n}{k}$ when n is a negative integer. One way to approach these values is to use the addition law (5.8) to fill in the entries that lie above the numbers in Table 155, thereby obtaining Table 164. For example, we must have $\binom{-1}{0} = 1$, since $\binom{0}{0} = \binom{-1}{0} + \binom{-1}{-1}$ and $\binom{-1}{-1} = 0$; then we must have $\binom{-1}{1} = -1$, since $\binom{0}{1} = \binom{-1}{1} + \binom{-1}{0}$; and so on.

Table 164 Pascal's triangle, extended upward.

n	$\binom{n}{0}$	$\binom{n}{1}$	$\binom{n}{2}$	$\binom{n}{3}$	$\binom{n}{4}$	$\binom{n}{5}$	$\binom{n}{6}$	$\binom{n}{7}$	$\binom{n}{8}$	$\binom{n}{9}$	$\binom{n}{10}$
-4	1	-4	10	-20	35	-56	84	-120	165	-220	286
-3	1	-3	6	-10	15	-21	28	-36	45	-55	66
-2	1	-2	3	-4	5	-6	7	-8	9	-10	11
-1	1	-1	1	-1	1	-1	1	-1	1	-1	1
0	1	0	0	0	0	0	0	0	0	0	0

All these numbers are familiar. Indeed, the rows and columns of Table 164 appear as columns in Table 155 (but minus the minus signs). So there must be a connection between the values of $\binom{n}{k}$ for negative n and the values for positive n. The general rule is

$$\binom{r}{k} = (-1)^k \binom{k-r-1}{k}, \qquad \text{integer } k; \tag{5.14}$$

it is easily proved, since

$$\begin{aligned} r^{\underline{k}} &= r(r-1)\ldots(r-k+1) \\ &= (-1)^k(-r)(1-r)\ldots(k-1-r) \;=\; (-1)^k(k-r-1)^{\underline{k}} \end{aligned}$$

when $k \geqslant 0$, and both sides are zero when $k < 0$.

Identity (5.14) is particularly valuable because it holds without any restriction. (Of course, the lower index must be an integer so that the binomial coefficients are defined.) The transformation in (5.14) is called *negating the upper index*, or "upper negation."

But how can we remember this important formula? The other identities we've seen — symmetry, absorption, addition, etc. — are pretty simple, but this one looks rather messy. Still, there's a mnemonic that's not too bad: To negate the upper index, we begin by writing down $(-1)^k$, where k is the lower index. (The lower index doesn't change.) Then we immediately write k again, twice, in both lower and upper index positions. Then we negate the original upper index by *subtracting* it from the new upper index. And we complete the job by *subtracting* 1 more (always subtracting, not adding, because this is a negation process).

You call this a mnemonic? I'd call it pneumatic — full of air. It does help me remember, though.

Let's negate the upper index twice in succession, for practice. We get

(Now is a good time to do warmup exercise 4.)

$$\begin{aligned} \binom{r}{k} &= (-1)^k \binom{k-r-1}{k} \\ &= (-1)^{2k} \binom{k-(k-r-1)-1}{k} \;=\; \binom{r}{k}, \end{aligned}$$

It's also frustrating, if we're trying to get somewhere else.

so we're right back where we started. This is probably not what the framers of the identity intended; but it's reassuring to know that we haven't gone astray.

Some applications of (5.14) are, of course, more useful than this. We can use upper negation, for example, to move quantities between upper and lower index positions. The identity has a symmetric formulation,

$$(-1)^m\binom{-n-1}{m} = (-1)^n\binom{-m-1}{n}, \qquad \text{integers } m,n \geqslant 0, \tag{5.15}$$

which holds because both sides are equal to $\binom{m+n}{n}$.

Upper negation can also be used to derive the following interesting sum:

$$\begin{aligned}\sum_{k\leqslant m}\binom{r}{k}(-1)^k &= \binom{r}{0}-\binom{r}{1}+\cdots+(-1)^m\binom{r}{m}\\ &= (-1)^m\binom{r-1}{m}, \qquad \text{integer } m.\end{aligned} \tag{5.16}$$

The idea is to negate the upper index, then apply (5.9), and negate again:

(Here double negation helps, because we've sandwiched another operation in between.)

$$\begin{aligned}\sum_{k\leqslant m}\binom{r}{k}(-1)^k &= \sum_{k\leqslant m}\binom{k-r-1}{k}\\ &= \binom{-r+m}{m}\\ &= (-1)^m\binom{r-1}{m}.\end{aligned}$$

This formula gives us a partial sum of the rth row of Pascal's triangle, provided that the entries of the row have been given alternating signs. For instance, if $r=5$ and $m=2$ the formula gives $1-5+10=6=(-1)^2\binom{4}{2}$.

Notice that if $m \geqslant r$, (5.16) gives the alternating sum of the entire row, and this sum is zero when r is a positive integer. We proved this before, when we expanded $(1-1)^r$ by the binomial theorem; it's interesting to know that the partial sums of this expression can also be evaluated in closed form.

How about the simpler partial sum,

$$\sum_{k\leqslant m}\binom{n}{k} = \binom{n}{0}+\binom{n}{1}+\cdots+\binom{n}{m}; \tag{5.17}$$

surely if we can evaluate the corresponding sum with alternating signs, we ought to be able to do this one? But no; there is no closed form for the partial sum of a row of Pascal's triangle. We can do columns — that's (5.10) — but

not rows. Curiously, however, there is a way to partially sum the row elements if they have been multiplied by their distance from the center:

$$\sum_{k\leqslant m}\binom{r}{k}\left(\frac{r}{2}-k\right) = \frac{m+1}{2}\binom{r}{m+1}, \qquad \text{integer } m. \tag{5.18}$$

(This formula is easily verified by induction on m.) The relation between these partial sums with and without the factor of $(r/2-k)$ in the summand is analogous to the relation between the integrals

$$\int_{-\infty}^{\alpha} xe^{-x^2}dx = -\tfrac{1}{2}e^{-\alpha^2} \qquad \text{and} \qquad \int_{-\infty}^{\alpha} e^{-x^2}dx\,.$$

The apparently more complicated integral on the left, with the factor of x, has a closed form, while the simpler-looking integral on the right, without the factor, has none. Appearances can be deceiving.

(Well, the right-hand integral is $\frac{1}{2}\sqrt{\pi}(1+\operatorname{erf}\alpha)$, a constant plus a multiple of the "error function" of α, if we're willing to accept that as a closed form.)

Near the end of this chapter, we'll study a method by which it's possible to determine whether or not there is a closed form for the partial sums of a given series involving binomial coefficients, in a fairly general setting. This method is capable of discovering identities (5.16) and (5.18), and it also will tell us that (5.17) is a dead end.

Partial sums of the binomial series lead to a curious relationship of another kind:

$$\sum_{k\leqslant m}\binom{m+r}{k}x^k y^{m-k} = \sum_{k\leqslant m}\binom{-r}{k}(-x)^k(x+y)^{m-k}, \quad \text{integer } m. \tag{5.19}$$

This identity isn't hard to prove by induction: Both sides are zero when $m<0$ and 1 when $m=0$. If we let S_m stand for the sum on the left, we can apply the addition formula (5.8) and show easily that

$$S_m = \sum_{k\leqslant m}\binom{m-1+r}{k}x^k y^{m-k} + \sum_{k\leqslant m}\binom{m-1+r}{k-1}x^k y^{m-k};$$

and

$$\sum_{k\leqslant m}\binom{m-1+r}{k}x^k y^{m-k} = yS_{m-1} + \binom{m-1+r}{m}x^m,$$
$$\sum_{k\leqslant m}\binom{m-1+r}{k-1}x^k y^{m-k} = xS_{m-1},$$

when $m>0$. Hence

$$S_m = (x+y)S_{m-1} + \binom{-r}{m}(-x)^m,$$

and this recurrence is satisfied also by the right-hand side of (5.19). By induction, both sides must be equal; QED.

But there's a neater proof. When r is an integer in the range $0 \geqslant r \geqslant -m$, the binomial theorem tells us that both sides of (5.19) are $(x+y)^{m+r}y^{-r}$. And since both sides are polynomials in r of degree m or less, agreement at $m+1$ different values is enough (but just barely!) to prove equality in general.

It may seem foolish to have an identity where one sum equals another. Neither side is in closed form. But sometimes one side turns out to be easier to evaluate than the other. For example, if we set $x = -1$ and $y = 1$, we get

$$\sum_{k \leqslant m} \binom{m+r}{k}(-1)^k = \binom{-r}{m}, \qquad \text{integer } m \geqslant 0,$$

an alternative form of identity (5.16). And if we set $x = y = 1$ and $r = m+1$, we get

$$\sum_{k \leqslant m} \binom{2m+1}{k} = \sum_{k \leqslant m} \binom{m+k}{k} 2^{m-k}.$$

The left-hand side sums just half of the binomial coefficients with upper index $2m+1$, and these are equal to their counterparts in the other half because Pascal's triangle has left-right symmetry. Hence the left-hand side is just $\frac{1}{2}2^{2m+1} = 2^{2m}$. This yields a formula that is quite unexpected,

(There's a nice combinatorial proof of this formula [247].)

$$\sum_{k \leqslant m} \binom{m+k}{k} 2^{-k} = 2^m, \qquad \text{integer } m \geqslant 0. \tag{5.20}$$

Let's check it when $m = 2$: $\binom{2}{0} + \frac{1}{2}\binom{3}{1} + \frac{1}{4}\binom{4}{2} = 1 + \frac{3}{2} + \frac{6}{4} = 4$. Astounding.

So far we've been looking either at binomial coefficients by themselves or at sums of terms in which there's only one binomial coefficient per term. But many of the challenging problems we face involve products of two or more binomial coefficients, so we'll spend the rest of this section considering how to deal with such cases.

Here's a handy rule that often helps to simplify the product of two binomial coefficients:

$$\binom{r}{m}\binom{m}{k} = \binom{r}{k}\binom{r-k}{m-k}, \qquad \text{integers } m,\ k. \tag{5.21}$$

We've already seen the special case $k = 1$; it's the absorption identity (5.6). Although both sides of (5.21) are products of binomial coefficients, one side often is easier to sum because of interactions with the rest of a formula. For example, the left side uses m twice, the right side uses it only once. Therefore we usually want to replace $\binom{r}{m}\binom{m}{k}$ by $\binom{r}{k}\binom{r-k}{m-k}$ when summing on m.

Equation (5.21) holds primarily because of cancellation between $m!$'s in the factorial representations of $\binom{r}{m}$ and $\binom{m}{k}$. If all variables are integers and $r \geqslant m \geqslant k \geqslant 0$, we have

$$\begin{aligned}\binom{r}{m}\binom{m}{k} &= \frac{r!}{m!\,(r-m)!}\,\frac{m!}{k!\,(m-k)!}\\ &= \frac{r!}{k!\,(m-k)!\,(r-m)!}\\ &= \frac{r!}{k!\,(r-k)!}\,\frac{(r-k)!}{(m-k)!\,(r-m)!} = \binom{r}{k}\binom{r-k}{m-k}.\end{aligned}$$

That was easy. Furthermore, if $m < k$ or $k < 0$, both sides of (5.21) are zero; so the identity holds for all integers m and k. Finally, the polynomial argument extends its validity to all real r.

Yeah, right.

A binomial coefficient $\binom{r}{k} = r!/(r-k)!\,k!$ can be written in the form $(a+b)!/a!\,b!$ after a suitable renaming of variables. Similarly, the quantity in the middle of the derivation above, $r!/k!\,(m-k)!\,(r-m)!$, can be written in the form $(a+b+c)!/a!\,b!\,c!$. This is a "trinomial coefficient," which arises in the "trinomial theorem":

$$\begin{aligned}(x+y+z)^n &= \sum_{\substack{0\leqslant a,b,c\leqslant n\\ a+b+c=n}} \frac{(a+b+c)!}{a!\,b!\,c!}x^a y^b z^c\\ &= \sum_{\substack{0\leqslant a,b,c\leqslant n\\ a+b+c=n}} \binom{a+b+c}{b+c}\binom{b+c}{c}x^a y^b z^c\,.\end{aligned}$$

"Excogitavi autem olim mirabilem regulam pro numeris coefficientibus potestatum, non tantum a binomio $x+y$, sed et a trinomio $x+y+z$, imo a polynomio quocunque, ut data potentia gradus cujuscunque v. gr. decimi, et potentia in ejus valore comprehensa, ut $x^5y^3z^2$, possim statim assignare numerum coefficientem, quem habere debet, sine ulla Tabula jam calculata."
—G. W. Leibniz [245]

So $\binom{r}{m}\binom{m}{k}$ is really a trinomial coefficient in disguise. Trinomial coefficients pop up occasionally in applications, and we can conveniently write them as

$$\binom{a+b+c}{a,\,b,\,c} = \frac{(a+b+c)!}{a!\,b!\,c!}$$

in order to emphasize the symmetry present.

Binomial and trinomial coefficients generalize to *multinomial coefficients*, which are always expressible as products of binomial coefficients:

$$\begin{aligned}\binom{a_1+a_2+\cdots+a_m}{a_1,a_2,\ldots,a_m} &= \frac{(a_1+a_2+\cdots+a_m)!}{a_1!\,a_2!\,\ldots\,a_m!}\\ &= \binom{a_1+a_2+\cdots+a_m}{a_2+\cdots+a_m}\cdots\binom{a_{m-1}+a_m}{a_m}.\end{aligned}$$

Therefore, when we run across such a beastie, our standard techniques apply.

Table 169 Sums of products of binomial coefficients.

$$\sum_k \binom{r}{m+k}\binom{s}{n-k} = \binom{r+s}{m+n}, \qquad \text{integers } m, n. \tag{5.22}$$

$$\sum_k \binom{l}{m+k}\binom{s}{n+k} = \binom{l+s}{l-m+n}, \qquad \begin{array}{c}\text{integer } l \geqslant 0,\\ \text{integers } m, n.\end{array} \tag{5.23}$$

$$\sum_k \binom{l}{m+k}\binom{s+k}{n}(-1)^k = (-1)^{l+m}\binom{s-m}{n-l}, \qquad \begin{array}{c}\text{integer } l \geqslant 0,\\ \text{integers } m, n.\end{array} \tag{5.24}$$

$$\sum_{k\leqslant l} \binom{l-k}{m}\binom{s}{k-n}(-1)^k = (-1)^{l+m}\binom{s-m-1}{l-m-n}, \qquad \begin{array}{c}\text{integers}\\ l, m, n \geqslant 0.\end{array} \tag{5.25}$$

$$\sum_{0\leqslant k\leqslant l} \binom{l-k}{m}\binom{q+k}{n} = \binom{l+q+1}{m+n+1}, \qquad \begin{array}{c}\text{integers } l, m \geqslant 0,\\ \text{integers } n \geqslant q \geqslant 0.\end{array} \tag{5.26}$$

Now we come to Table 169, which lists identities that are among the most important of our standard techniques. These are the ones we rely on when struggling with a sum involving a product of two binomial coefficients. Each of these identities is a sum over k, with one appearance of k in each binomial coefficient; there also are four nearly independent parameters, called m, n, r, etc., one in each index position. Different cases arise depending on whether k appears in the upper or lower index, and on whether it appears with a plus or minus sign. Sometimes there's an additional factor of $(-1)^k$, which is needed to make the terms summable in closed form.

Fold down the corner on this page, so you can find the table quickly later. You'll need it!

Table 169 is far too complicated to memorize in full; it is intended only for reference. But the first identity in this table is by far the most memorable, and it should be remembered. It states that the sum (over all integers k) of the product of two binomial coefficients, in which the upper indices are constant and the lower indices have a constant sum for all k, is the binomial coefficient obtained by summing both lower and upper indices. This identity is known as *Vandermonde's convolution*, because Alexandre Vandermonde wrote a significant paper about it in the late 1700s [357]; it was, however, known to Chu Shih-Chieh in China as early as 1303. All of the other identities in Table 169 can be obtained from Vandermonde's convolution by doing things like negating upper indices or applying the symmetry law, etc., with care; therefore Vandermonde's convolution is the most basic of all.

We can prove Vandermonde's convolution by giving it a nice combinatorial interpretation. If we replace k by $k-m$ and n by $n-m$, we can assume

that $m = 0$; hence the identity to be proved is

$$\sum_k \binom{r}{k}\binom{s}{n-k} = \binom{r+s}{n}, \qquad \text{integer } n. \tag{5.27}$$

Let r and s be nonnegative integers; the general case then follows by the polynomial argument. On the right side, $\binom{r+s}{n}$ is the number of ways to choose n people from among r men and s women. On the left, each term of the sum is the number of ways to choose k of the men and $n-k$ of the women. Summing over all k counts each possibility exactly once.

Sexist! You mentioned men first.

Much more often than not we use these identities left to right, since that's the direction of simplification. But every once in a while it pays to go the other direction, temporarily making an expression more complicated. When this works, we've usually created a double sum for which we can interchange the order of summation and then simplify.

Before moving on let's look at proofs for two more of the identities in Table 169. It's easy to prove (5.23); all we need to do is replace the first binomial coefficient by $\binom{l}{l-m-k}$, then Vandermonde's (5.22) applies.

The next one, (5.24), is a bit more difficult. We can reduce it to Vandermonde's convolution by a sequence of transformations, but we can just as easily prove it by resorting to the old reliable technique of mathematical induction. Induction is often the first thing to try when nothing else obvious jumps out at us, and induction on l works just fine here.

For the basis $l = 0$, all terms are zero except when $k = -m$; so both sides of the equation are $(-1)^m\binom{s-m}{n}$. Now suppose that the identity holds for all values less than some fixed l, where $l > 0$. We can use the addition formula to replace $\binom{l}{m+k}$ by $\binom{l-1}{m+k} + \binom{l-1}{m+k-1}$; the original sum now breaks into two sums, each of which can be evaluated by the induction hypothesis:

$$\sum_k \binom{l-1}{m+k}\binom{s+k}{n}(-1)^k + \sum_k \binom{l-1}{m+k-1}\binom{s+k}{n}(-1)^k$$
$$= (-1)^{l-1+m}\binom{s-m}{n-l+1} + (-1)^{l+m}\binom{s-m+1}{n-l+1}.$$

And this simplifies to the right-hand side of (5.24), if we apply the addition formula once again.

Two things about this derivation are worthy of note. First, we see again the great convenience of summing over all integers k, not just over a certain range, because there's no need to fuss over boundary conditions. Second, the addition formula works nicely with mathematical induction, because it's a recurrence for binomial coefficients. A binomial coefficient whose upper index is l is expressed in terms of two whose upper indices are $l-1$, and that's exactly what we need to apply the induction hypothesis.

So much for Table 169. What about sums with three or more binomial coefficients? If the index of summation is spread over all the coefficients, our chances of finding a closed form aren't great: Only a few closed forms are known for sums of this kind, hence the sum we need might not match the given specs. One of these rarities, proved in exercise 43, is

$$\sum_k \binom{m-r+s}{k}\binom{n+r-s}{n-k}\binom{r+k}{m+n} = \binom{r}{m}\binom{s}{n}, \qquad \text{integers } m,n \geqslant 0. \tag{5.28}$$

Here's another, more symmetric example:

$$\sum_k \binom{a+b}{a+k}\binom{b+c}{b+k}\binom{c+a}{c+k}(-1)^k = \frac{(a+b+c)!}{a!\,b!\,c!}, \qquad \text{integers } a,b,c \geqslant 0. \tag{5.29}$$

This one has a two-coefficient counterpart,

$$\sum_k \binom{a+b}{a+k}\binom{b+a}{b+k}(-1)^k = \frac{(a+b)!}{a!\,b!}, \qquad \text{integers } a,b \geqslant 0, \tag{5.30}$$

which incidentally doesn't appear in Table 169. The analogous four-coefficient sum doesn't have a closed form, but a similar sum does:

$$\sum_k (-1)^k \binom{a+b}{a+k}\binom{b+c}{b+k}\binom{c+d}{c+k}\binom{d+a}{d+k}\Big/\binom{2a+2b+2c+2d}{a+b+c+d+k} = \frac{(a{+}b{+}c{+}d)!\,(a{+}b{+}c)!\,(a{+}b{+}d)!\,(a{+}c{+}d)!\,(b{+}c{+}d)!}{(2a{+}2b{+}2c{+}2d)!\,(a{+}c)!\,(b{+}d)!\,a!\,b!\,c!\,d!}, \qquad \text{integers } a,b,c,d \geqslant 0.$$

This was discovered by John Dougall [82] early in the twentieth century.

Is Dougall's identity the hairiest sum of binomial coefficients known? No! The champion so far is

$$\sum_{k_{ij}} (-1)^{\Sigma_{i<j}k_{ij}} \left(\prod_{1\leqslant i<j<n}\binom{a_i+a_j}{a_j+k_{ij}}\right)\left(\prod_{1\leqslant j<n}\binom{a_j+a_n}{a_n+\Sigma_{i<j}k_{ij}-\Sigma_{i>j}k_{ji}}\right) = \binom{a_1+\cdots+a_n}{a_1,a_2,\ldots,a_n}, \qquad \text{integers } a_1,a_2,\ldots,a_n \geqslant 0. \tag{5.31}$$

Here the sum is over $\binom{n-1}{2}$ index variables k_{ij} for $1 \leqslant i < j < n$. Equation (5.29) is the special case $n = 3$; the case $n = 4$ can be written out as follows,

if we use (a,b,c,d) for (a_1,a_2,a_3,a_4) and (i,j,k) for (k_{12},k_{13},k_{23}):

$$\sum_{i,j,k}(-1)^{i+j+k}\binom{a+b}{b+i}\binom{a+c}{c+j}\binom{b+c}{c+k}\binom{a+d}{d-i-j}\binom{b+d}{d+i-k}\binom{c+d}{d+j+k}$$
$$= \frac{(a+b+c+d)!}{a!\,b!\,c!\,d!}, \qquad \text{integers } a,b,c,d \geqslant 0.$$

The left side of (5.31) is the coefficient of $z_1^0 z_2^0 \ldots z_n^0$ after the product of $n(n-1)$ fractions

$$\prod_{\substack{1\leqslant i,j\leqslant n\\ i\neq j}}\left(1-\frac{z_i}{z_j}\right)^{a_i}$$

has been fully expanded into positive and negative powers of the z's. The right side of (5.31) was conjectured by Freeman Dyson in 1962 and proved by several people shortly thereafter. Exercise 89 gives a "simple" proof of (5.31).

Another noteworthy identity involving lots of binomial coefficients is

$$\sum_{j,k}(-1)^{j+k}\binom{j+k}{k+l}\binom{r}{j}\binom{n}{k}\binom{s+n-j-k}{m-j}$$
$$= (-1)^l\binom{n+r}{n+l}\binom{s-r}{m-n-l}, \quad \text{integers } l,m,n; \quad n\geqslant 0. \quad (5.32)$$

This one, proved in exercise 83, even has a chance of arising in practical applications. But we're getting far afield from our theme of "basic identities," so we had better stop and take stock of what we've learned.

We've seen that binomial coefficients satisfy an almost bewildering variety of identities. Some of these, fortunately, are easily remembered, and we can use the memorable ones to derive most of the others in a few steps. Table 174 collects ten of the most useful formulas, all in one place; these are the best identities to know.

5.2 BASIC PRACTICE

In the previous section we derived a bunch of identities by manipulating sums and plugging in other identities. It wasn't too tough to find those derivations — we knew what we were trying to prove, so we could formulate a general plan and fill in the details without much trouble. Usually, however, out in the real world, we're not faced with an identity to prove; we're faced with a sum to simplify. And we don't know what a simplified form might look like (or even if one exists). By tackling many such sums in this section and the next, we will hone our binomial coefficient tools.

To start, let's try our hand at a few sums involving a single binomial coefficient.

Algorithm
<u>self-teach</u>:
1 read problem
2 attempt solution
3 skim book solution
4 <u>if</u> attempt failed <u>goto</u> 1
<u>else</u> <u>goto</u> next problem

Problem 1: A sum of ratios.

We'd like to have a closed form for

$$\sum_{k=0}^{m} \binom{m}{k} \Big/ \binom{n}{k}, \qquad \text{integers } n \geqslant m \geqslant 0.$$

At first glance this sum evokes panic, because we haven't seen any identities that deal with a quotient of binomial coefficients. (Furthermore the sum involves two binomial coefficients, which seems to contradict the sentence preceding this problem.) However, just as we can use the factorial representations to reexpress a product of binomial coefficients as another product — that's how we got identity (5.21) — we can do likewise with a quotient. In fact we can avoid the grubby factorial representations by letting $r = n$ and dividing both sides of equation (5.21) by $\binom{n}{k}\binom{n}{m}$; this yields

Unfortunately, that algorithm can put you in an infinite loop.

Suggested patches:
0 <u>set</u> $c \leftarrow 0$
3a <u>set</u> $c \leftarrow c + 1$
3b <u>if</u> $c = N$ <u>goto</u> your TA

$$\binom{m}{k} \Big/ \binom{n}{k} = \binom{n-k}{m-k} \Big/ \binom{n}{m}.$$

So we replace the quotient on the left, which appears in our sum, by the one on the right; the sum becomes

$$\sum_{k=0}^{m} \binom{n-k}{m-k} \Big/ \binom{n}{m}.$$

We still have a quotient, but the binomial coefficient in the denominator doesn't involve the index of summation k, so we can remove it from the sum. We'll restore it later.

— E. W. Dijkstra

We can also simplify the boundary conditions by summing over all $k \geqslant 0$; the terms for $k > m$ are zero. The sum that's left isn't so intimidating:

$$\sum_{k \geqslant 0} \binom{n-k}{m-k}.$$

... But this subchapter is called BASIC practice.

It's similar to the one in identity (5.9), because the index k appears twice with the same sign. But here it's $-k$ and in (5.9) it's not. The next step should therefore be obvious; there's only one reasonable thing to do:

$$\begin{aligned}\sum_{k \geqslant 0} \binom{n-k}{m-k} &= \sum_{m-k \geqslant 0} \binom{n-(m-k)}{m-(m-k)} \\ &= \sum_{k \leqslant m} \binom{n-m+k}{k}.\end{aligned}$$

Table 174 The top ten binomial coefficient identities.

$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$,	integers $n \geqslant k \geqslant 0$.	*factorial expansion*
$\binom{n}{k} = \binom{n}{n-k}$,	integer $n \geqslant 0$, integer k.	*symmetry*
$\binom{r}{k} = \frac{r}{k}\binom{r-1}{k-1}$,	integer $k \neq 0$.	*absorption/extraction*
$\binom{r}{k} = \binom{r-1}{k} + \binom{r-1}{k-1}$,	integer k.	*addition/induction*
$\binom{r}{k} = (-1)^k\binom{k-r-1}{k}$,	integer k.	*upper negation*
$\binom{r}{m}\binom{m}{k} = \binom{r}{k}\binom{r-k}{m-k}$,	integers m, k.	*trinomial revision*
$\sum_k \binom{r}{k} x^k y^{r-k} = (x+y)^r$,	integer $r \geqslant 0$, or $\lvert x/y\rvert < 1$.	*binomial theorem*
$\sum_{k \leqslant n} \binom{r+k}{k} = \binom{r+n+1}{n}$,	integer n.	*parallel summation*
$\sum_{0 \leqslant k \leqslant n} \binom{k}{m} = \binom{n+1}{m+1}$,	integers $m, n \geqslant 0$.	*upper summation*
$\sum_k \binom{r}{k}\binom{s}{n-k} = \binom{r+s}{n}$,	integer n.	*Vandermonde convolution*

And now we can apply the parallel summation identity, (5.9):

$$\sum_{k \leqslant m} \binom{n-m+k}{k} = \binom{(n-m)+m+1}{m} = \binom{n+1}{m}.$$

Finally we reinstate the $\binom{n}{m}$ in the denominator that we removed from the sum earlier, and then apply (5.7) to get the desired closed form:

$$\binom{n+1}{m} \Big/ \binom{n}{m} = \frac{n+1}{n+1-m}.$$

This derivation actually works for any real value of n, as long as no division by zero occurs; that is, as long as n isn't one of the integers $0, 1, \ldots, m-1$.

The more complicated the derivation, the more important it is to check the answer. This one wasn't too complicated but we'll check anyway. In the small case $m = 2$ and $n = 4$ we have

$$\binom{2}{0}\Big/\binom{4}{0} + \binom{2}{1}\Big/\binom{4}{1} + \binom{2}{2}\Big/\binom{4}{2} = 1 + \frac{1}{2} + \frac{1}{6} = \frac{5}{3};$$

yes, this agrees perfectly with our closed form $(4+1)/(4+1-2)$.

Problem 2: From the literature of sorting.

Our next sum appeared way back in ancient times (the early 1970s) before people were fluent with binomial coefficients. A paper that introduced an improved merging technique [196] concludes with the following remarks: "It can be shown that the expected number of saved transfers ... is given by the expression

$$T = \sum_{r=0}^{n} r \, \frac{{}_{m-r-1}C_{m-n-1}}{{}_{m}C_{n}}$$

Here m and n are as defined above, and ${}_{m}C_{n}$ is the symbol for the number of combinations of m objects taken n at a time. ... The author is grateful to the referee for reducing a more complex equation for expected transfers saved to the form given here."

We'll see that this is definitely not a final answer to the author's problem. It's not even a midterm answer.

Please, don't remind me of the midterm.

First we should translate the sum into something we can work with; the ghastly notation ${}_{m-r-1}C_{m-n-1}$ is enough to stop anybody, save the enthusiastic referee (please). In our language we'd write

$$T = \sum_{k=0}^{n} k\binom{m-k-1}{m-n-1}\Big/\binom{m}{n}, \qquad \text{integers } m > n \geqslant 0.$$

The binomial coefficient in the denominator doesn't involve the index of summation, so we can remove it and work with the new sum

$$S = \sum_{k=0}^{n} k\binom{m-k-1}{m-n-1}.$$

What next? The index of summation appears in the upper index of the binomial coefficient but not in the lower index. So if the other k weren't there, we could massage the sum and apply summation on the upper index (5.10). With the extra k, though, we can't. If we could somehow absorb that k into the binomial coefficient, using one of our absorption identities, we could then

sum on the upper index. Unfortunately those identities don't work here. But if the k were instead $m-k$, we could use absorption identity (5.6):

$$(m-k)\binom{m-k-1}{m-n-1} = (m-n)\binom{m-k}{m-n}.$$

So here's the key: We'll rewrite k as $m-(m-k)$ and split the sum S into two sums:

$$\begin{aligned}
\sum_{k=0}^{n} k\binom{m-k-1}{m-n-1} &= \sum_{k=0}^{n} (m-(m-k))\binom{m-k-1}{m-n-1} \\
&= \sum_{k=0}^{n} m\binom{m-k-1}{m-n-1} - \sum_{k=0}^{n} (m-k)\binom{m-k-1}{m-n-1} \\
&= m\sum_{k=0}^{n} \binom{m-k-1}{m-n-1} - \sum_{k=0}^{n} (m-n)\binom{m-k}{m-n} \\
&= mA-(m-n)B\,,
\end{aligned}$$

where

$$A = \sum_{k=0}^{n} \binom{m-k-1}{m-n-1}, \qquad B = \sum_{k=0}^{n} \binom{m-k}{m-n}.$$

The sums A and B that remain are none other than our old friends in which the upper index varies while the lower index stays fixed. Let's do B first, because it looks simpler. A little bit of massaging is enough to make the summand match the left side of (5.10):

$$\begin{aligned}
\sum_{0\leqslant k\leqslant n} \binom{m-k}{m-n} &= \sum_{0\leqslant m-k\leqslant n} \binom{m-(m-k)}{m-n} \\
&= \sum_{m-n\leqslant k\leqslant m} \binom{k}{m-n} \\
&= \sum_{0\leqslant k\leqslant m} \binom{k}{m-n}.
\end{aligned}$$

In the last step we've included the terms with $0 \leqslant k < m-n$ in the sum; they're all zero, because the upper index is less than the lower. Now we sum on the upper index, using (5.10), and get

$$B = \sum_{0\leqslant k\leqslant m} \binom{k}{m-n} = \binom{m+1}{m-n+1}.$$

The other sum A is the same, but with m replaced by $m-1$. Hence we have a closed form for the given sum S, which can be further simplified:

$$\begin{aligned} S &= mA-(m-n)B = m\binom{m}{m-n}-(m-n)\binom{m+1}{m-n+1} \\ &= \left(m-(m-n)\frac{m+1}{m-n+1}\right)\binom{m}{m-n} \\ &= \left(\frac{n}{m-n+1}\right)\binom{m}{m-n}. \end{aligned}$$

And this gives us a closed form for the original sum:

$$\begin{aligned} T &= S\Big/\binom{m}{n} \\ &= \frac{n}{m-n+1}\binom{m}{m-n}\Big/\binom{m}{n} \\ &= \frac{n}{m-n+1}. \end{aligned}$$

Even the referee can't simplify this.

Again we use a small case to check the answer. When $m=4$ and $n=2$, we have

$$T = 0\cdot\tbinom{3}{1}/\tbinom{4}{2} + 1\cdot\tbinom{2}{1}/\tbinom{4}{2} + 2\cdot\tbinom{1}{1}/\tbinom{4}{2} = 0+\tfrac{2}{6}+\tfrac{2}{6} = \tfrac{2}{3},$$

which agrees with our formula $2/(4-2+1)$.

Problem 3: From an old exam.

Do old exams ever die?

Let's do one more sum that involves a single binomial coefficient. This one, unlike the last, originated in the halls of academia; it was a problem on a take-home test. We want the value of $Q_{1000000}$, when

$$Q_n = \sum_{k\leqslant 2^n}\binom{2^n-k}{k}(-1)^k, \qquad \text{integer } n\geqslant 0.$$

This one's harder than the others; we can't apply *any* of the identities we've seen so far. And we're faced with a sum of $2^{1000000}$ terms, so we can't just add them up. The index of summation k appears in both indices, upper and lower, but with opposite signs. Negating the upper index doesn't help, either; it removes the factor of $(-1)^k$, but it introduces a $2k$ in the upper index.

When nothing obvious works, we know that it's best to look at small cases. If we can't spot a pattern and prove it by induction, at least we'll have

some data for checking our results. Here are the nonzero terms and their sums for the first four values of n.

n			Q_n
0	$\binom{1}{0}$	$= 1$	$= 1$
1	$\binom{2}{0} - \binom{1}{1}$	$= 1 - 1$	$= 0$
2	$\binom{4}{0} - \binom{3}{1} + \binom{2}{2}$	$= 1 - 3 + 1$	$= -1$
3	$\binom{8}{0} - \binom{7}{1} + \binom{6}{2} - \binom{5}{3} + \binom{4}{4}$	$= 1 - 7 + 15 - 10 + 1$	$= 0$

We'd better not try the next case, $n = 4$; the chances of making an arithmetic error are too high. (Computing terms like $\binom{12}{4}$ and $\binom{11}{5}$ by hand, let alone combining them with the others, is worthwhile only if we're desperate.)

So the pattern starts out 1, 0, -1, 0. Even if we knew the next term or two, the closed form wouldn't be obvious. But if we could find and prove a recurrence for Q_n we'd probably be able to guess and prove its closed form. To find a recurrence, we need to relate Q_n to Q_{n-1} (or to $Q_{\text{smaller values}}$); but to do this we need to relate a term like $\binom{128-13}{13}$, which arises when $n = 7$ and $k = 13$, to terms like $\binom{64-13}{13}$. This doesn't look promising; we don't know any neat relations between entries in Pascal's triangle that are 64 rows apart. The addition formula, our main tool for induction proofs, only relates entries that are one row apart.

But this leads us to a key observation: There's no need to deal with entries that are 2^{n-1} rows apart. The variable n never appears by itself, it's always in the context 2^n. So the 2^n is a red herring! If we replace 2^n by m, all we need to do is find a closed form for the more general (but easier) sum

Oh, the sneakiness of the instructor who set that exam.

$$R_m = \sum_{k \leqslant m} \binom{m-k}{k} (-1)^k, \qquad \text{integer } m \geqslant 0;$$

then we'll also have a closed form for $Q_n = R_{2^n}$. And there's a good chance that the addition formula will give us a recurrence for the sequence R_m.

Values of R_m for small m can be read from Table 155, if we alternately add and subtract values that appear in a southwest-to-northeast diagonal. The results are:

m	0	1	2	3	4	5	6	7	8	9	10
R_m	1	1	0	-1	-1	0	1	1	0	-1	-1

There seems to be a lot of cancellation going on.

Let's look now at the formula for R_m and see if it defines a recurrence. Our strategy is to apply the addition formula (5.8) and to find sums that

have the form R_k in the resulting expression, somewhat as we did in the perturbation method of Chapter 2:

$$\begin{aligned}
R_m &= \sum_{k\leqslant m}\binom{m-k}{k}(-1)^k \\
&= \sum_{k\leqslant m}\binom{m-1-k}{k}(-1)^k \;+\; \sum_{k\leqslant m}\binom{m-1-k}{k-1}(-1)^k \\
&= \sum_{k\leqslant m}\binom{m-1-k}{k}(-1)^k \;+\; \sum_{k+1\leqslant m}\binom{m-2-k}{k}(-1)^{k+1} \\
&= \sum_{k\leqslant m-1}\binom{m-1-k}{k}(-1)^k \;+\; \binom{-1}{m}(-1)^m \\
&\qquad - \sum_{k\leqslant m-2}\binom{m-2-k}{k}(-1)^k \;-\; \binom{-1}{m-1}(-1)^{m-1} \\
&= R_{m-1} + (-1)^{2m} - R_{m-2} - (-1)^{2(m-1)} \;=\; R_{m-1} - R_{m-2}\,.
\end{aligned}$$

(In the next-to-last step we've used the formula $\binom{-1}{m} = (-1)^m$, which we know is true when $m \geqslant 0$.) This derivation is valid for $m \geqslant 2$.

Anyway those of us who've done warmup exercise 4 know it.

From this recurrence we can generate values of R_m quickly, and we soon perceive that the sequence is periodic. Indeed,

$$R_m = \begin{cases} 1 \\ 1 \\ 0 \\ -1 \\ -1 \\ 0 \end{cases} \quad \text{if } m \bmod 6 = \begin{cases} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{cases}.$$

The proof by induction is by inspection. Or, if we must give a more academic proof, we can unfold the recurrence one step to obtain

$$R_m = (R_{m-2} - R_{m-3}) - R_{m-2} = -R_{m-3}\,,$$

whenever $m \geqslant 3$. Hence $R_m = R_{m-6}$ whenever $m \geqslant 6$.

Finally, since $Q_n = R_{2^n}$, we can determine Q_n by determining $2^n \bmod 6$ and using the closed form for R_m. When $n = 0$ we have $2^0 \bmod 6 = 1$; after that we keep multiplying by 2 (mod 6), so the pattern 2, 4 repeats. Thus

$$Q_n = R_{2^n} = \begin{cases} R_1 = 1, & \text{if } n = 0; \\ R_2 = 0, & \text{if } n \text{ is odd;} \\ R_4 = -1, & \text{if } n > 0 \text{ is even.} \end{cases}$$

This closed form for Q_n agrees with the first four values we calculated when we started on the problem. We conclude that $Q_{1000000} = R_4 = -1$.

Problem 4: A sum involving two binomial coefficients.

Our next task is to find a closed form for

$$\sum_{k=0}^{n} k\binom{m-k-1}{m-n-1}, \qquad \text{integers } m > n \geqslant 0.$$

Wait a minute. Where's the second binomial coefficient promised in the title of this problem? And why should we try to simplify a sum we've already simplified? (This is the sum S from Problem 2.)

Well, this is a sum that's easier to simplify if we view the summand as a product of two binomial coefficients, and then use one of the general identities found in Table 169. The second binomial coefficient materializes when we rewrite k as $\binom{k}{1}$:

$$\sum_{k=0}^{n} k\binom{m-k-1}{m-n-1} = \sum_{0\leqslant k\leqslant n} \binom{k}{1}\binom{m-k-1}{m-n-1}.$$

And identity (5.26) is the one to apply, since its index of summation appears in both upper indices and with opposite signs.

But our sum isn't quite in the correct form yet. The upper limit of summation should be $m-1$, if we're to have a perfect match with (5.26). No problem; the terms for $n < k \leqslant m-1$ are zero. So we can plug in, with $(l, m, n, q) \leftarrow (m-1, m-n-1, 1, 0)$; the answer is

$$S = \binom{m}{m-n+1}.$$

This is cleaner than the formula we got before. We can convert it to the previous formula by using (5.7):

$$\binom{m}{m-n+1} = \frac{n}{m-n+1}\binom{m}{m-n}.$$

Similarly, we can get interesting results by plugging special values into the other general identities we've seen. Suppose, for example, that we set $m = n = 1$ and $q = 0$ in (5.26). Then the identity reads

$$\sum_{0\leqslant k\leqslant l} (l-k)k = \binom{l+1}{3}.$$

The left side is $l\big((l+1)l/2\big) - (1^2+2^2+\cdots+l^2)$, so this gives us a brand new way to solve the sum-of-squares problem that we beat to death in Chapter 2.

The moral of this story is: Special cases of very general sums are sometimes best handled in the general form. When learning general forms, it's wise to learn their simple specializations.

Problem 5: A sum with three factors.

Here's another sum that isn't too bad. We wish to simplify

$$\sum_k \binom{n}{k}\binom{s}{k}k\,, \qquad \text{integer } n \geqslant 0.$$

The index of summation k appears in both lower indices and with the same sign; therefore identity (5.23) in Table 169 looks close to what we need. With a bit of manipulation, we should be able to use it.

The biggest difference between (5.23) and what we have is the extra k in our sum. But we can absorb k into one of the binomial coefficients by using one of the absorption identities:

$$\begin{aligned}\sum_k \binom{n}{k}\binom{s}{k}k &= \sum_k \binom{n}{k}\binom{s-1}{k-1}s \\ &= s\sum_k \binom{n}{k}\binom{s-1}{k-1}.\end{aligned}$$

We don't care that the s appears when the k disappears, because it's constant. And now we're ready to apply the identity and get the closed form,

$$s\sum_k \binom{n}{k}\binom{s-1}{k-1} = s\binom{n+s-1}{n-1}.$$

If we had chosen in the first step to absorb k into $\binom{n}{k}$, not $\binom{s}{k}$, we wouldn't have been allowed to apply (5.23) directly, because $n-1$ might be negative; the identity requires a nonnegative value in at least one of the upper indices.

Problem 6: A sum with menacing characteristics.

The next sum is more challenging. We seek a closed form for

$$\sum_{k\geqslant 0} \binom{n+k}{2k}\binom{2k}{k}\frac{(-1)^k}{k+1}\,, \qquad \text{integer } n \geqslant 0.$$

So we should deep six this sum, right?

One useful measure of a sum's difficulty is the number of times the index of summation appears. By this measure we're in deep trouble — k appears six times. Furthermore, the key step that worked in the previous problem — to absorb something outside the binomial coefficients into one of them — won't work here. If we absorb the $k+1$ we just get another occurrence of k in its place. And not only that: Our index k is twice shackled with the coefficient 2 inside a binomial coefficient. Multiplicative constants are usually harder to remove than additive constants.

We're lucky this time, though. The $2k$'s are right where we need them for identity (5.21) to apply, so we get

$$\sum_{k\geqslant 0}\binom{n+k}{2k}\binom{2k}{k}\frac{(-1)^k}{k+1} = \sum_{k\geqslant 0}\binom{n+k}{k}\binom{n}{k}\frac{(-1)^k}{k+1}.$$

The two 2's disappear, and so does one occurrence of k. So that's one down and five to go.

The $k+1$ in the denominator is the most troublesome characteristic left, and now we can absorb it into $\binom{n}{k}$ using identity (5.6):

$$\begin{aligned}\sum_{k\geqslant 0}\binom{n+k}{k}\binom{n}{k}\frac{(-1)^k}{k+1} &= \sum_k\binom{n+k}{k}\binom{n+1}{k+1}\frac{(-1)^k}{n+1}\\ &= \frac{1}{n+1}\sum_k\binom{n+k}{k}\binom{n+1}{k+1}(-1)^k.\end{aligned}$$

(Recall that $n\geqslant 0$.) Two down, four to go.

To eliminate another k we have two promising options. We could use symmetry on $\binom{n+k}{k}$; or we could negate the upper index $n+k$, thereby eliminating that k as well as the factor $(-1)^k$. Let's explore both possibilities, starting with the symmetry option:

$$\frac{1}{n+1}\sum_k\binom{n+k}{k}\binom{n+1}{k+1}(-1)^k = \frac{1}{n+1}\sum_k\binom{n+k}{n}\binom{n+1}{k+1}(-1)^k.$$

Third down, three to go, and we're in position to make a big gain by plugging into (5.24): Replacing (l,m,n,s) by $(n+1,1,n,n)$, we get

For a minute I thought we'd have to punt.

$$\frac{1}{n+1}\sum_k\binom{n+k}{n}\binom{n+1}{k+1}(-1)^k = \frac{1}{n+1}(-1)^n\binom{n-1}{-1} = 0.$$

Zero, eh? After all that work? Let's check it when $n=2$: $\binom{2}{0}\binom{0}{0}\frac{1}{1}-\binom{3}{2}\binom{2}{1}\frac{1}{2}+\binom{4}{4}\binom{4}{2}\frac{1}{3}=1-\frac{6}{2}+\frac{6}{3}=0$. It checks.

Just for the heck of it, let's explore our other option, negating the upper index of $\binom{n+k}{k}$:

$$\frac{1}{n+1}\sum_k\binom{n+k}{k}\binom{n+1}{k+1}(-1)^k = \frac{1}{n+1}\sum_k\binom{-n-1}{k}\binom{n+1}{k+1}.$$

Now (5.23) applies, with $(l,m,n,s)\leftarrow(n+1,1,0,-n-1)$, and

$$\frac{1}{n+1}\sum_k\binom{-n-1}{k}\binom{n+1}{k+1} = \frac{1}{n+1}\binom{0}{n}.$$

Hey wait. This is zero when $n > 0$, but it's 1 when $n = 0$. Our other path to the solution told us that the sum was zero in all cases! What gives? The sum actually does turn out to be 1 when $n = 0$, so the correct answer is '$[n=0]$'. We must have made a mistake in the previous derivation.

Try binary search: Replay the middle formula first, to see if the mistake was early or late.

Let's do an instant replay on that derivation when $n = 0$, in order to see where the discrepancy first arises. Ah yes; we fell into the old trap mentioned earlier: We tried to apply symmetry when the upper index could be negative! We were not justified in replacing $\binom{n+k}{k}$ by $\binom{n+k}{n}$ when k ranges over all integers, because this converts zero into a nonzero value when $k < -n$. (Sorry about that.)

The other factor in the sum, $\binom{n+1}{k+1}$, turns out to be zero when $k < -n$, except when $n = 0$ and $k = -1$. Hence our error didn't show up when we checked the case $n = 2$. Exercise 6 explains what we should have done.

Problem 7: A new obstacle.

This one's even tougher; we want a closed form for

$$\sum_{k\geqslant 0}\binom{n+k}{m+2k}\binom{2k}{k}\frac{(-1)^k}{k+1}, \qquad \text{integers } m, n > 0.$$

If m were 0 we'd have the sum from the problem we just finished. But it's not, and we're left with a real mess — nothing we used in Problem 6 works here. (Especially not the crucial first step.)

However, if we could somehow get rid of the m, we could use the result just derived. So our strategy is: Replace $\binom{n+k}{m+2k}$ by a sum of terms like $\binom{l+k}{2k}$ for some nonnegative integer l; the summand will then look like the summand in Problem 6, and we can interchange the order of summation.

What should we substitute for $\binom{n+k}{m+2k}$? A painstaking examination of the identities derived earlier in this chapter turns up only one suitable candidate, namely equation (5.26) in Table 169. And one way to use it is to replace the parameters (l, m, n, q, k) by $(n+k-1, 2k, m-1, 0, j)$, respectively:

$$\begin{aligned}
&\sum_{k\geqslant 0}\binom{n+k}{m+2k}\binom{2k}{k}\frac{(-1)^k}{k+1}\\
&\quad=\sum_{k\geqslant 0}\sum_{0\leqslant j\leqslant n+k-1}\binom{n+k-1-j}{2k}\binom{j}{m-1}\binom{2k}{k}\frac{(-1)^k}{k+1}\\
&\quad=\sum_{j\geqslant 0}\binom{j}{m-1}\sum_{\substack{k\geqslant j-n+1\\ k\geqslant 0}}\binom{n+k-1-j}{2k}\binom{2k}{k}\frac{(-1)^k}{k+1}.
\end{aligned}$$

In the last step we've changed the order of summation, manipulating the conditions below the $\sum$'s according to the rules of Chapter 2.

We can't quite replace the inner sum using the result of Problem 6, because it has the extra condition $k \geqslant j - n + 1$. But this extra condition is superfluous unless $j - n + 1 > 0$; that is, unless $j \geqslant n$. And when $j \geqslant n$, the first binomial coefficient of the inner sum is zero, because its upper index is between 0 and $k - 1$, thus strictly less than the lower index $2k$. We may therefore place the additional restriction $j < n$ on the outer sum, without affecting which nonzero terms are included. This makes the restriction $k \geqslant j - n + 1$ superfluous, and we can use the result of Problem 6. The double sum now comes tumbling down:

$$\begin{aligned}
&\sum_{j\geqslant 0}\binom{j}{m-1}\sum_{\substack{k\geqslant j-n+1\\ k\geqslant 0}}\binom{n+k-1-j}{2k}\binom{2k}{k}\frac{(-1)^k}{k+1}\\
&\quad=\sum_{0\leqslant j<n}\binom{j}{m-1}\sum_{k\geqslant 0}\binom{n+k-1-j}{2k}\binom{2k}{k}\frac{(-1)^k}{k+1}\\
&\quad=\sum_{0\leqslant j<n}\binom{j}{m-1}[n-1-j=0] \;=\; \binom{n-1}{m-1}.
\end{aligned}$$

The inner sums vanish except when $j = n - 1$, so we get a simple closed form as our answer.

Problem 8: A different obstacle.

Let's branch out from Problem 6 in another way by considering the sum

$$S_m \;=\; \sum_{k\geqslant 0}\binom{n+k}{2k}\binom{2k}{k}\frac{(-1)^k}{k+1+m}, \qquad \text{integers } m, n \geqslant 0.$$

Again, when $m = 0$ we have the sum we did before; but now the m occurs in a different place. This problem is a bit harder yet than Problem 7, but (fortunately) we're getting better at finding solutions. We can begin as in Problem 6,

$$S_m \;=\; \sum_{k\geqslant 0}\binom{n+k}{k}\binom{n}{k}\frac{(-1)^k}{k+1+m}.$$

Now (as in Problem 7) we try to expand the part that depends on m into terms that we know how to deal with. When m was zero, we absorbed $k + 1$ into $\binom{n}{k}$; if $m > 0$, we can do the same thing if we expand $1/(k + 1 + m)$ into absorbable terms. And our luck still holds: We proved a suitable identity

$$\sum_{j=0}^{m}\binom{m}{j}\binom{r}{j}^{-1} \;=\; \frac{r+1}{r+1-m}, \qquad \begin{array}{c}\text{integer } m \geqslant 0,\\ r \notin \{0, 1, \ldots, m-1\}.\end{array} \tag{5·33}$$

in Problem 1. Replacing r by $-k-2$ gives the desired expansion,

$$S_m = \sum_{k\geqslant 0}\binom{n+k}{k}\binom{n}{k}\frac{(-1)^k}{k+1}\sum_{j\geqslant 0}\binom{m}{j}\binom{-k-2}{j}^{-1}.$$

Now the $(k+1)^{-1}$ can be absorbed into $\binom{n}{k}$, as planned. In fact, it could also be absorbed into $\binom{-k-2}{j}^{-1}$. Double absorption suggests that even more cancellation might be possible behind the scenes. Yes — expanding everything in our new summand into factorials and going back to binomial coefficients gives a formula that we can sum on k:

They expect us to check this on a sheet of scratch paper.

$$\begin{aligned} S_m &= \frac{m!\,n!}{(m+n+1)!}\sum_{j\geqslant 0}(-1)^j\binom{m+n+1}{n+1+j}\sum_k\binom{n+1+j}{k+j+1}\binom{-n-1}{k} \\ &= \frac{m!\,n!}{(m+n+1)!}\sum_{j\geqslant 0}(-1)^j\binom{m+n+1}{n+1+j}\binom{j}{n}. \end{aligned}$$

The sum over all integers j is zero, by (5.24). Hence $-S_m$ is the sum for $j<0$.

To evaluate $-S_m$ for $j<0$, let's replace j by $-k-1$ and sum for $k\geqslant 0$:

$$\begin{aligned} S_m &= \frac{m!\,n!}{(m+n+1)!}\sum_{k\geqslant 0}(-1)^k\binom{m+n+1}{n-k}\binom{-k-1}{n} \\ &= \frac{m!\,n!}{(m+n+1)!}\sum_{k\leqslant n}(-1)^{n-k}\binom{m+n+1}{k}\binom{k-n-1}{n} \\ &= \frac{m!\,n!}{(m+n+1)!}\sum_{k\leqslant n}(-1)^k\binom{m+n+1}{k}\binom{2n-k}{n} \\ &= \frac{m!\,n!}{(m+n+1)!}\sum_{k\leqslant 2n}(-1)^k\binom{m+n+1}{k}\binom{2n-k}{n}. \end{aligned}$$

Finally (5.25) applies, and we have our answer:

$$S_m = (-1)^n\frac{m!\,n!}{(m+n+1)!}\binom{m}{n} = (-1)^n m^{\underline{n}} m^{\underline{-n-1}}.$$

Whew; we'd better check it. When $n=2$ we find

$$S_m = \frac{1}{m+1}-\frac{6}{m+2}+\frac{6}{m+3} = \frac{m(m-1)}{(m+1)(m+2)(m+3)}.$$

Our derivation requires m to be an integer, but the result holds for all real m, because the quantity $(m+1)^{\overline{n+1}}\,S_m$ is a polynomial in m of degree $\leqslant n$.

5.3 TRICKS OF THE TRADE

Let's look next at three techniques that significantly amplify the methods we have already learned.

Trick 1: Going halves.

This should really be called Trick 1/2.

Many of our identities involve an arbitrary real number r. When r has the special form "integer minus one half," the binomial coefficient $\binom{r}{k}$ can be written as a quite different-looking product of binomial coefficients. This leads to a new family of identities that can be manipulated with surprising ease.

One way to see how this works is to begin with the *duplication formula*

$$r^{\underline{k}}(r-\tfrac{1}{2})^{\underline{k}} = (2r)^{\underline{2k}}/2^{2k}, \qquad \text{integer } k \geqslant 0. \tag{5·34}$$

This identity is obvious if we expand the falling powers and interleave the factors on the left side:

$$\begin{aligned} &r(r-\tfrac{1}{2})(r-1)(r-\tfrac{3}{2})\ldots(r-k+1)(r-k+\tfrac{1}{2}) \\ &\qquad = \frac{(2r)(2r-1)\ldots(2r-2k+1)}{2\cdot 2\cdot\ldots\cdot 2}. \end{aligned}$$

Now we can divide both sides by $k!^2$, and we get

$$\binom{r}{k}\binom{r-1/2}{k} = \binom{2r}{2k}\binom{2k}{k}\Big/2^{2k}, \qquad \text{integer } k. \tag{5·35}$$

If we set $k = r = n$, where n is an integer, this yields

$$\binom{n-1/2}{n} = \binom{2n}{n}\Big/2^{2n}, \qquad \text{integer } n. \tag{5·36}$$

And negating the upper index gives yet another useful formula,

$$\binom{-1/2}{n} = \left(\frac{-1}{4}\right)^n\binom{2n}{n}, \qquad \text{integer } n. \tag{5·37}$$

For example, when $n = 4$ we have

. . . we halve. . .

$$\begin{aligned} \binom{-1/2}{4} &= \frac{(-1/2)(-3/2)(-5/2)(-7/2)}{4!} \\ &= \left(\frac{-1}{2}\right)^4\frac{1\cdot3\cdot5\cdot7}{1\cdot2\cdot3\cdot4} \\ &= \left(\frac{-1}{4}\right)^4\frac{1\cdot3\cdot5\cdot7\cdot2\cdot4\cdot6\cdot8}{1\cdot2\cdot3\cdot4\cdot1\cdot2\cdot3\cdot4} = \left(\frac{-1}{4}\right)^4\binom{8}{4}. \end{aligned}$$

Notice how we've changed a product of odd numbers into a factorial.

Identity (5.35) has an amusing corollary. Let $r = \frac{1}{2}n$, and take the sum over all integers k. The result is

$$\begin{aligned}\sum_k \binom{n}{2k}\binom{2k}{k}2^{-2k} &= \sum_k \binom{n/2}{k}\binom{(n-1)/2}{k} \\ &= \binom{n-1/2}{\lfloor n/2\rfloor}, \qquad \text{integer } n \geqslant 0 \end{aligned} \tag{5.38}$$

by (5.23), because either $n/2$ or $(n-1)/2$ is $\lfloor n/2\rfloor$, a nonnegative integer!

We can also use Vandermonde's convolution (5.27) to deduce that

$$\sum_k \binom{-1/2}{k}\binom{-1/2}{n-k} = \binom{-1}{n} = (-1)^n, \qquad \text{integer } n \geqslant 0.$$

Plugging in the values from (5.37) gives

$$\begin{aligned}\binom{-1/2}{k}\binom{-1/2}{n-k} &= \left(\frac{-1}{4}\right)^k\binom{2k}{k}\left(\frac{-1}{4}\right)^{n-k}\binom{2(n-k)}{n-k} \\ &= \frac{(-1)^n}{4^n}\binom{2k}{k}\binom{2n-2k}{n-k};\end{aligned}$$

this is what sums to $(-1)^n$. Hence we have a remarkable property of the "middle" elements of Pascal's triangle:

$$\sum_k \binom{2k}{k}\binom{2n-2k}{n-k} = 4^n, \qquad \text{integer } n \geqslant 0. \tag{5.39}$$

For example, $\binom{0}{0}\binom{6}{3}+\binom{2}{1}\binom{4}{2}+\binom{4}{2}\binom{2}{1}+\binom{6}{3}\binom{0}{0} = 1{\cdot}20+2{\cdot}6+6{\cdot}2+20{\cdot}1 = 64 = 4^3$.

These illustrations of our first trick indicate that it's wise to try changing binomial coefficients of the form $\binom{2k}{k}$ into binomial coefficients of the form $\binom{n-1/2}{k}$, where n is some appropriate integer (usually 0, 1, or k); the resulting formula might be much simpler.

Trick 2: High-order differences.

We saw earlier that it's possible to evaluate partial sums of the series $\binom{n}{k}(-1)^k$, but not of the series $\binom{n}{k}$. It turns out that there are many important applications of binomial coefficients with alternating signs, $\binom{n}{k}(-1)^k$. One of the reasons for this is that such coefficients are intimately associated with the difference operator Δ defined in Section 2.6.

The difference Δf of a function f at the point x is

$$\Delta f(x) = f(x+1) - f(x);$$

if we apply Δ again, we get the second difference

$$\begin{aligned}\Delta^2 f(x) &= \Delta f(x+1) - \Delta f(x) = \big(f(x{+}2) - f(x{+}1)\big) - \big(f(x{+}1) - f(x)\big)\\ &= f(x+2) - 2f(x+1) + f(x)\,,\end{aligned}$$

which is analogous to the second derivative. Similarly, we have

$$\begin{aligned}\Delta^3 f(x) &= f(x+3) - 3f(x+2) + 3f(x+1) - f(x)\,;\\ \Delta^4 f(x) &= f(x+4) - 4f(x+3) + 6f(x+2) - 4f(x+1) + f(x)\,;\end{aligned}$$

and so on. Binomial coefficients enter these formulas with alternating signs.

In general, the nth difference is

$$\Delta^n f(x) = \sum_k \binom{n}{k}(-1)^{n-k} f(x+k)\,, \qquad \text{integer } n \geqslant 0. \tag{5.40}$$

This formula is easily proved by induction, but there's also a nice way to prove it directly using the elementary theory of operators. Recall that Section 2.6 defines the shift operator E by the rule

$$Ef(x) = f(x+1)\,;$$

hence the operator Δ is $E-1$, where 1 is the identity operator defined by the rule $1f(x) = f(x)$. By the binomial theorem,

$$\Delta^n = (E-1)^n = \sum_k \binom{n}{k} E^k (-1)^{n-k}\,.$$

This is an equation whose elements are operators; it is equivalent to (5.40), since E^k is the operator that takes $f(x)$ into $f(x+k)$.

An interesting and important case arises when we consider negative falling powers. Let $f(x) = (x-1)^{\underline{-1}} = 1/x$. Then, by rule (2.45), we have $\Delta f(x) = (-1)(x-1)^{\underline{-2}}$, $\Delta^2 f(x) = (-1)(-2)(x-1)^{\underline{-3}}$, and in general

$$\Delta^n\big((x-1)^{\underline{-1}}\big) = (-1)^{\underline{n}}(x-1)^{\underline{-n-1}} = (-1)^n \frac{n!}{x(x+1)\ldots(x+n)}\,.$$

Equation (5.40) now tells us that

$$\begin{aligned}\sum_k \binom{n}{k}\frac{(-1)^k}{x+k} &= \frac{n!}{x(x+1)\ldots(x+n)}\\ &= x^{-1}\binom{x+n}{n}^{-1}, \qquad x \notin \{0,-1,\ldots,-n\}.\end{aligned} \tag{5.41}$$

For example,

$$\frac{1}{x} - \frac{4}{x+1} + \frac{6}{x+2} - \frac{4}{x+3} + \frac{1}{x+4}$$
$$= \frac{4!}{x(x+1)(x+2)(x+3)(x+4)} = 1\Big/x\binom{x+4}{4}.$$

The sum in (5.41) is the partial fraction expansion of $n!/\big(x(x+1)\dots(x+n)\big)$.

Significant results can be obtained from positive falling powers too. If $f(x)$ is a polynomial of degree d, the difference $\Delta f(x)$ is a polynomial of degree $d-1$; therefore $\Delta^d f(x)$ is a constant, and $\Delta^n f(x) = 0$ if $n > d$. This extremely important fact simplifies many formulas.

A closer look gives further information: Let

$$f(x) = a_d x^d + a_{d-1}x^{d-1} + \cdots + a_1 x^1 + a_0 x^0$$

be any polynomial of degree d. We will see in Chapter 6 that we can express ordinary powers as sums of falling powers (for example, $x^2 = x^{\underline{2}} + x^{\underline{1}}$); hence there are coefficients b_d, b_{d-1}, ..., b_1, b_0 such that

$$f(x) = b_d x^{\underline{d}} + b_{d-1}x^{\underline{d-1}} + \cdots + b_1 x^{\underline{1}} + b_0 x^{\underline{0}}.$$

(It turns out that $b_d = a_d$ and $b_0 = a_0$, but the intervening coefficients are related in a more complicated way.) Let $c_k = k!\,b_k$ for $0 \leqslant k \leqslant d$. Then

$$f(x) = c_d\binom{x}{d} + c_{d-1}\binom{x}{d-1} + \cdots + c_1\binom{x}{1} + c_0\binom{x}{0};$$

thus, any polynomial can be represented as a sum of multiples of binomial coefficients. Such an expansion is called the *Newton series* of $f(x)$, because Isaac Newton used it extensively.

We observed earlier in this chapter that the addition formula implies

$$\Delta\left(\binom{x}{k}\right) = \binom{x}{k-1}.$$

Therefore, by induction, the nth difference of a Newton series is very simple:

$$\Delta^n f(x) = c_d\binom{x}{d-n} + c_{d-1}\binom{x}{d-1-n} + \cdots + c_1\binom{x}{1-n} + c_0\binom{x}{-n}.$$

If we now set $x = 0$, all terms $c_k\binom{x}{k-n}$ on the right side are zero, except the term with $k - n = 0$; hence

$$\Delta^n f(0) = \begin{cases} c_n, & \text{if } n \leqslant d; \\ 0, & \text{if } n > d. \end{cases}$$

The Newton series for $f(x)$ is therefore

$$f(x) \;=\; \Delta^d f(0)\binom{x}{d} + \Delta^{d-1} f(0)\binom{x}{d-1} + \cdots + \Delta f(0)\binom{x}{1} + f(0)\binom{x}{0}.$$

For example, suppose $f(x) = x^3$. It's easy to calculate

$$\begin{aligned} f(0) = 0, \qquad f(1) = 1, \qquad f(2) = 8, \qquad f(3) = 27;\\ \Delta f(0) = 1, \qquad \Delta f(1) = 7, \qquad \Delta f(2) = 19;\\ \Delta^2 f(0) = 6, \quad \Delta^2 f(1) = 12;\\ \Delta^3 f(0) = 6. \end{aligned}$$

So the Newton series is $x^3 = 6\binom{x}{3} + 6\binom{x}{2} + 1\binom{x}{1} + 0\binom{x}{0}$.

Our formula $\Delta^n f(0) = c_n$ can also be stated in the following way, using (5.40) with $x = 0$:

$$\sum_k \binom{n}{k}(-1)^k\left(c_0\binom{k}{0} + c_1\binom{k}{1} + c_2\binom{k}{2} + \cdots\right) \;=\; (-1)^n c_n\,, \qquad \text{integer } n \geqslant 0.$$

Here $\langle c_0, c_1, c_2, \ldots\rangle$ is an arbitrary sequence of coefficients; the infinite sum $c_0\binom{k}{0} + c_1\binom{k}{1} + c_2\binom{k}{2} + \cdots$ is actually finite for all $k \geqslant 0$, so convergence is not an issue. In particular, we can prove the important identity

$$\sum_k \binom{n}{k}(-1)^k(a_0 + a_1 k + \cdots + a_n k^n) \;=\; (-1)^n n!\, a_n\,, \qquad \text{integer } n \geqslant 0, \tag{5.42}$$

because the polynomial $a_0 + a_1 k + \cdots + a_n k^n$ can always be written as a Newton series $c_0\binom{k}{0} + c_1\binom{k}{1} + \cdots + c_n\binom{k}{n}$ with $c_n = n!\, a_n$.

Many sums that appear to be hopeless at first glance can actually be summed almost trivially by using the idea of nth differences. For example, let's consider the identity

$$\sum_k \binom{n}{k}\binom{r-sk}{n}(-1)^k \;=\; s^n\,, \qquad \text{integer } n \geqslant 0. \tag{5.43}$$

This looks very impressive, because it's quite different from anything we've seen so far. But it really is easy to understand, once we notice the telltale factor $\binom{n}{k}(-1)^k$ in the summand, because the function

$$f(k) \;=\; \binom{r-sk}{n} \;=\; \frac{1}{n!}(-1)^n s^n k^n + \cdots \;=\; (-1)^n s^n \binom{k}{n} + \cdots$$

is a polynomial in k of degree n, with leading coefficient $(-1)^n s^n/n!$. Therefore (5.43) is nothing more than an application of (5.42).

We have discussed Newton series under the assumption that $f(x)$ is a polynomial. But we've also seen that infinite Newton series

$$f(x) \;=\; c_0\binom{x}{0} + c_1\binom{x}{1} + c_2\binom{x}{2} + \cdots$$

make sense too, because such sums are always finite when x is a nonnegative integer. Our derivation of the formula $\Delta^n f(0) = c_n$ works in the infinite case, just as in the polynomial case; so we have the general identity

$$f(x) \;=\; f(0)\binom{x}{0} + \Delta f(0)\binom{x}{1} + \Delta^2 f(0)\binom{x}{2} + \Delta^3 f(0)\binom{x}{3} + \cdots, \qquad \text{integer } x \geqslant 0. \tag{5.44}$$

This formula is valid for any function $f(x)$ that is defined for nonnegative integers x. Moreover, if the right-hand side converges for other values of x, it defines a function that "interpolates" $f(x)$ in a natural way. (There are infinitely many ways to interpolate function values, so we cannot assert that (5.44) is true for all x that make the infinite series converge. For example, if we let $f(x) = \sin(\pi x)$, we have $f(x) = 0$ at all integer points, so the right-hand side of (5.44) is identically zero; but the left-hand side is nonzero at all noninteger x.)

A Newton series is finite calculus's answer to infinite calculus's Taylor series. Just as a Taylor series can be written

$$g(a+x) \;=\; \frac{g(a)}{0!}x^0 + \frac{g'(a)}{1!}x^1 + \frac{g''(a)}{2!}x^2 + \frac{g'''(a)}{3!}x^3 + \cdots,$$

(Since $E = 1 + \Delta$, $E^x = \Sigma_k \binom{x}{k} \Delta^k$; *and* $E^x g(a) = g(a+x)$.*)*

the Newton series for $f(x) = g(a+x)$ can be written

$$g(a+x) \;=\; \frac{g(a)}{0!}x^{\underline{0}} + \frac{\Delta g(a)}{1!}x^{\underline{1}} + \frac{\Delta^2 g(a)}{2!}x^{\underline{2}} + \frac{\Delta^3 g(a)}{3!}x^{\underline{3}} + \cdots. \tag{5.45}$$

(This is the same as (5.44), because $\Delta^n f(0) = \Delta^n g(a)$ for all $n \geqslant 0$ when $f(x) = g(a+x)$.) Both the Taylor and Newton series are finite when g is a polynomial, or when $x = 0$; in addition, the Newton series is finite when x is a positive integer. Otherwise the sums may or may not converge for particular values of x. If the Newton series converges when x is not a nonnegative integer, it might actually converge to a value that's *different* from $g(a+x)$, because the Newton series (5.45) depends only on the spaced-out function values $g(a)$, $g(a+1)$, $g(a+2)$, $\ldots$.

One example of a convergent Newton series is provided by the binomial theorem. Let $g(x) = (1+z)^x$, where z is a fixed complex number such that $|z| < 1$. Then $\Delta g(x) = (1+z)^{x+1} - (1+z)^x = z(1+z)^x$, hence $\Delta^n g(x) = z^n(1+z)^x$. In this case the infinite Newton series

$$g(a+x) \;=\; \sum_n \Delta^n g(a)\binom{x}{n} \;=\; (1+z)^a \sum_n \binom{x}{n} z^n$$

converges to the "correct" value $(1+z)^{a+x}$, for all x.

James Stirling tried to use Newton series to generalize the factorial function to noninteger values. First he found coefficients S_n such that

$$x! \;=\; \sum_n S_n \binom{x}{n} \;=\; S_0\binom{x}{0} + S_1\binom{x}{1} + S_2\binom{x}{2} + \cdots \tag{5.46}$$

is an identity for $x = 0$, $x = 1$, $x = 2$, etc. But he discovered that the resulting series doesn't converge except when x is a nonnegative integer. So he tried again, this time writing

"Forasmuch as these terms increase very fast, their differences will make a diverging progression, which hinders the ordinate of the parabola from approaching to the truth; therefore in this and the like cases, I interpolate the logarithms of the terms, whose differences constitute a series swiftly converging."
—J. Stirling [343]

$$\ln x! \;=\; \sum_n s_n \binom{x}{n} \;=\; s_0\binom{x}{0} + s_1\binom{x}{1} + s_2\binom{x}{2} + \cdots . \tag{5.47}$$

Now $\Delta(\ln x!) = \ln(x+1)! - \ln x! = \ln(x+1)$, hence

$$\begin{aligned} s_n &= \Delta^n(\ln x!)\big|_{x=0} \\ &= \Delta^{n-1}\big(\ln(x+1)\big)\big|_{x=0} \\ &= \sum_k \binom{n-1}{k}(-1)^{n-1-k}\ln(k+1) \end{aligned}$$

by (5.40). The coefficients are therefore $s_0 = s_1 = 0$; $s_2 = \ln 2$; $s_3 = \ln 3 - 2\ln 2 = \ln\frac{3}{4}$; $s_4 = \ln 4 - 3\ln 3 + 3\ln 2 = \ln\frac{32}{27}$; etc. In this way Stirling obtained a series that does converge (although he didn't prove it); in fact, his series converges for all $x > -1$. He was thereby able to evaluate $\frac{1}{2}!$ satisfactorily. Exercise 88 tells the rest of the story.

(Proofs of convergence were not invented until the nineteenth century.)

Trick 3: Inversion.

A special case of the rule (5.45) we've just derived for Newton's series can be rewritten in the following way:

$$g(n) \;=\; \sum_k \binom{n}{k}(-1)^k f(k) \quad\Longleftrightarrow\quad f(n) \;=\; \sum_k \binom{n}{k}(-1)^k g(k)\,. \tag{5.48}$$

This dual relationship between f and g is called an *inversion formula*; it's rather like the Möbius inversion formulas (4.56) and (4.61) that we encountered in Chapter 4. Inversion formulas tell us how to solve "implicit recurrences," where an unknown sequence is embedded in a sum.

Invert this: 'zinb ppo'.

For example, $g(n)$ might be a known function, and $f(n)$ might be unknown; and we might have found a way to show that $g(n) = \sum_k \binom{n}{k}(-1)^k f(k)$. Then (5.48) lets us express $f(n)$ as a sum of known values.

We can prove (5.48) directly by using the basic methods at the beginning of this chapter. If $g(n) = \sum_k \binom{n}{k}(-1)^k f(k)$ for all $n \geqslant 0$, then

$$\begin{aligned}
\sum_k \binom{n}{k}(-1)^k g(k) &= \sum_k \binom{n}{k}(-1)^k \sum_j \binom{k}{j}(-1)^j f(j) \\
&= \sum_j f(j) \sum_k \binom{n}{k}(-1)^{k+j}\binom{k}{j} \\
&= \sum_j f(j) \sum_k \binom{n}{j}(-1)^{k+j}\binom{n-j}{k-j} \\
&= \sum_j f(j)\binom{n}{j} \sum_k (-1)^k \binom{n-j}{k} \\
&= \sum_j f(j)\binom{n}{j}[n-j=0] \quad = \quad f(n)\,.
\end{aligned}$$

The proof in the other direction is, of course, the same, because the relation between f and g is symmetric.

Let's illustrate (5.48) by applying it to the "football victory problem": A group of n fans of the winning football team throw their hats high into the air. The hats come back randomly, one hat to each of the n fans. How many ways $h(n,k)$ are there for exactly k fans to get their own hats back?

For example, if $n = 4$ and if the hats and fans are named A, B, C, D, the $4! = 24$ possible ways for hats to land generate the following numbers of rightful owners:

ABCD	4	BACD	2	CABD	1	DABC	0
ABDC	2	BADC	0	CADB	0	DACB	1
ACBD	2	BCAD	1	CBAD	2	DBAC	1
ACDB	1	BCDA	0	CBDA	1	DBCA	2
ADBC	1	BDAC	0	CDAB	0	DCAB	0
ADCB	2	BDCA	1	CDBA	0	DCBA	0

Therefore $h(4,4) = 1$; $h(4,3) = 0$; $h(4,2) = 6$; $h(4,1) = 8$; $h(4,0) = 9$.

We can determine $h(n,k)$ by noticing that it is the number of ways to choose k lucky hat owners, namely $\binom{n}{k}$, times the number of ways to arrange the remaining $n-k$ hats so that none of them goes to the right owner, namely $h(n-k,0)$. A permutation is called a *derangement* if it moves every item, and the number of derangements of n objects is sometimes denoted by the symbol '$n_¡$', read "n subfactorial." Therefore $h(n-k,0) = (n-k)_¡$, and we have the general formula

$$h(n,k) \;=\; \binom{n}{k} h(n-k,0) \;=\; \binom{n}{k}(n-k)_¡\,.$$

(Subfactorial notation isn't standard, and it's not clearly a great idea; but let's try it awhile to see if we grow to like it. We can always resort to 'D_n' or something, if '$n_¡$' doesn't work out.)

Our problem would be solved if we had a closed form for $n_¡$, so let's see what we can find. There's an easy way to get a recurrence, because the sum of $h(n,k)$ for all k is the total number of permutations of n hats:

$$\begin{aligned} n! \;=\; \sum_k h(n,k) \;&=\; \sum_k \binom{n}{k}(n-k)_¡ \\ &=\; \sum_k \binom{n}{k} k_¡\,, \qquad \text{integer } n \geqslant 0. \end{aligned} \tag{5.49}$$

(We've changed k to $n-k$ and $\binom{n}{n-k}$ to $\binom{n}{k}$ in the last step.) With this implicit recurrence we can compute all the $h(n,k)$'s we like:

n	$h(n,0)$	$h(n,1)$	$h(n,2)$	$h(n,3)$	$h(n,4)$	$h(n,5)$	$h(n,6)$
0	1						
1	0	1					
2	1	0	1				
3	2	3	0	1			
4	9	8	6	0	1		
5	44	45	20	10	0	1	
6	265	264	135	40	15	0	1

For example, here's how the row for $n=4$ can be computed: The two rightmost entries are obvious — there's just one way for all hats to land correctly, and there's no way for just three fans to get their own. (Whose hat would the fourth fan get?) When $k=2$ and $k=1$, we can use our equation for $h(n,k)$, giving $h(4,2) = \binom{4}{2}h(2,0) = 6\cdot 1 = 6$, and $h(4,1) = \binom{4}{1}h(3,0) = 4\cdot 2 = 8$. We can't use this equation for $h(4,0)$; rather, we can, but it gives us $h(4,0) = \binom{4}{0}h(4,0)$, which is true but useless. Taking another tack, we can use the relation $h(4,0)+8+6+0+1 = 4!$ to deduce that $h(4,0) = 9$; this is the value of $4_¡$. Similarly $n_¡$ depends on the values of $k_¡$ for $k < n$.

The art of mathematics, as of life, is knowing which truths are useless.

How can we solve a recurrence like (5.49)? Easy; it has the form of (5.48), with $g(n) = n!$ and $f(k) = (-1)^k k_!$. Hence its solution is

$$n_! \;=\; (-1)^n \sum_k \binom{n}{k} (-1)^k k!\,.$$

Well, this isn't really a solution; it's a sum that should be put into closed form if possible. But it's better than a recurrence. The sum can be simplified, since $k!$ cancels with a hidden $k!$ in $\binom{n}{k}$, so let's try that: We get

$$n_! \;=\; \sum_{0\leqslant k\leqslant n} \frac{n!}{(n-k)!}(-1)^{n+k} \;=\; n! \sum_{0\leqslant k\leqslant n} \frac{(-1)^k}{k!}\,. \tag{5.50}$$

The remaining sum converges rapidly to the number $\sum_{k\geqslant 0}(-1)^k/k! = e^{-1}$. In fact, the terms that are excluded from the sum are

$$\begin{aligned} n! \sum_{k>n} \frac{(-1)^k}{k!} \;&=\; \frac{(-1)^{n+1}}{n+1} \sum_{k\geqslant 0} (-1)^k \frac{(n+1)!}{(k+n+1)!} \\ &=\; \frac{(-1)^{n+1}}{n+1}\left(1 - \frac{1}{n+2} + \frac{1}{(n+2)(n+3)} - \cdots\right), \end{aligned}$$

and the parenthesized quantity lies between 1 and $1 - \frac{1}{n+2} = \frac{n+1}{n+2}$. Therefore the difference between $n_!$ and $n!/e$ is roughly $1/n$ in absolute value; more precisely, it lies between $1/(n+1)$ and $1/(n+2)$. But $n_!$ is an integer. Therefore it must be what we get when we round $n!/e$ to the nearest integer, if $n > 0$. So we have the closed form we seek:

$$n_! \;=\; \left\lfloor \frac{n!}{e} + \frac{1}{2} \right\rfloor + [n{=}0]\,. \tag{5.51}$$

This is the number of ways that no fan gets the right hat back. When n is large, it's more meaningful to know the *probability* that this happens. If we assume that each of the $n!$ arrangements is equally likely — because the hats were thrown extremely high — this probability is

Baseball fans: .367 is also Ty Cobb's lifetime batting average, the all-time record. Can this be a coincidence?

$$\frac{n_!}{n!} \;=\; \frac{n!/e + O(1)}{n!} \;\sim\; \frac{1}{e} \;=\; .367\ldots\,.$$

So when n gets large the probability that all hats are misplaced is almost 37%.

(Hey wait, you're fudging. Cobb's average was $4191/11429 \approx .366699$, while $1/e \approx .367879$. But maybe if Wade Boggs has a few really good seasons...)

Incidentally, recurrence (5.49) for subfactorials is exactly the same as (5.46), the first recurrence considered by Stirling when he was trying to generalize the factorial function. Hence $S_k = k_!$. These coefficients are so large, it's no wonder the infinite series (5.46) diverges for noninteger x.

Before leaving this problem, let's look briefly at two interesting patterns that leap out at us in the table of small $h(n,k)$. First, it seems that the numbers 1, 3, 6, 10, 15, ... below the all-0 diagonal are the triangular numbers.

This observation is easy to prove, since those table entries are the $h(n, n-2)$'s, and we have

$$h(n, n-2) \;=\; \binom{n}{n-2} 2_{\text{¡}} \;=\; \binom{n}{2}.$$

It also seems that the numbers in the first two columns differ by ± 1. Is this always true? Yes,

$$\begin{aligned} h(n,0) - h(n,1) \;&=\; n_{\text{¡}} - n(n-1)_{\text{¡}} \\ &=\; \left(n! \sum_{0 \leqslant k \leqslant n} \frac{(-1)^k}{k!}\right) - \left(n(n-1)! \sum_{0 \leqslant k \leqslant n-1} \frac{(-1)^k}{k!}\right) \\ &=\; n! \frac{(-1)^n}{n!} \;=\; (-1)^n . \end{aligned}$$

In other words, $n_{\text{¡}} = n(n-1)_{\text{¡}} + (-1)^n$. This is a much simpler recurrence for the derangement numbers than we had before.

Now let's invert something else. If we apply inversion to the formula

But inversion is the source of smog.

$$\sum_k \binom{n}{k} \frac{(-1)^k}{x+k} \;=\; \frac{1}{x} \binom{x+n}{n}^{-1}$$

that we derived in (5.41), we find

$$\frac{x}{x+n} \;=\; \sum_{k \geqslant 0} \binom{n}{k} (-1)^k \binom{x+k}{k}^{-1} .$$

This is interesting, but not really new. If we negate the upper index in $\binom{x+k}{k}$, we have merely discovered identity (5.33) again.

5.4 GENERATING FUNCTIONS

We come now to the most important idea in this whole book, the notion of a *generating function*. An infinite sequence $\langle a_0, a_1, a_2, \ldots \rangle$ that we wish to deal with in some way can conveniently be represented as a *power series* in an auxiliary variable z,

$$A(z) \;=\; a_0 + a_1 z + a_2 z^2 + \cdots \;=\; \sum_{k \geqslant 0} a_k z^k . \tag{5.52}$$

It's appropriate to use the letter z as the name of the auxiliary variable, because we'll often be thinking of z as a complex number. The theory of complex variables conventionally uses 'z' in its formulas; power series (a.k.a. analytic functions or holomorphic functions) are central to that theory.

We will be seeing lots of generating functions in subsequent chapters. Indeed, Chapter 7 is entirely devoted to them. Our present goal is simply to introduce the basic concepts, and to demonstrate the relevance of generating functions to the study of binomial coefficients.

A generating function is useful because it's a single quantity that represents an entire infinite sequence. We can often solve problems by first setting up one or more generating functions, then by fooling around with those functions until we know a lot about them, and finally by looking again at the coefficients. With a little bit of luck, we'll know enough about the function to understand what we need to know about its coefficients.

(See [223] for a discussion of the history and usefulness of this notation.)

If $A(z)$ is any power series $\sum_{k\geqslant 0} a_k z^k$, we will find it convenient to write

$$[z^n]\,A(z) \;=\; a_n\,; \tag{5.53}$$

in other words, $[z^n]\,A(z)$ denotes the coefficient of z^n in $A(z)$.

Let $A(z)$ be the generating function for $\langle a_0, a_1, a_2, \ldots\rangle$ as in (5.52), and let $B(z)$ be the generating function for another sequence $\langle b_0, b_1, b_2, \ldots\rangle$. Then the product $A(z)B(z)$ is the power series

$$\begin{aligned}&(a_0 + a_1 z + a_2 z^2 + \cdots)(b_0 + b_1 z + b_2 z^2 + \cdots)\\ &\qquad = a_0 b_0 \;+\; (a_0 b_1 + a_1 b_0)z \;+\; (a_0 b_2 + a_1 b_1 + a_2 b_0)z^2 \;+\; \cdots\,;\end{aligned}$$

the coefficient of z^n in this product is

$$a_0 b_n + a_1 b_{n-1} + \cdots + a_n b_0 \;=\; \sum_{k=0}^{n} a_k b_{n-k}\,.$$

Therefore if we wish to evaluate any sum that has the general form

$$c_n \;=\; \sum_{k=0}^{n} a_k b_{n-k}\,, \tag{5.54}$$

and if we know the generating functions $A(z)$ and $B(z)$, we have

$$c_n \;=\; [z^n]\,A(z)B(z)\,.$$

The sequence $\langle c_n\rangle$ defined by (5.54) is called the *convolution* of the sequences $\langle a_n\rangle$ and $\langle b_n\rangle$; two sequences are "convolved" by forming the sums of all products whose subscripts add up to a given amount. The gist of the previous paragraph is that convolution of sequences corresponds to multiplication of their generating functions.

Generating functions give us powerful ways to discover and/or prove identities. For example, the binomial theorem tells us that $(1+z)^r$ is the generating function for the sequence $\langle \binom{r}{0}, \binom{r}{1}, \binom{r}{2}, \ldots \rangle$:

$$(1+z)^r = \sum_{k \geqslant 0} \binom{r}{k} z^k .$$

Similarly,

$$(1+z)^s = \sum_{k \geqslant 0} \binom{s}{k} z^k .$$

If we multiply these together, we get another generating function:

$$(1+z)^r (1+z)^s = (1+z)^{r+s} .$$

And now comes the punch line: Equating coefficients of z^n on both sides of this equation gives us

$$\sum_{k=0}^{n} \binom{r}{k} \binom{s}{n-k} = \binom{r+s}{n} .$$

We've discovered Vandermonde's convolution, (5.27)!

(5.27)! = (5.27)(4.27) (3.27)(2.27) (1.27)(0.27)!.

That was nice and easy; let's try another. This time we use $(1-z)^r$, which is the generating function for the sequence $\langle (-1)^n \binom{r}{n} \rangle = \langle \binom{r}{0}, -\binom{r}{1}, \binom{r}{2}, \ldots \rangle$. Multiplying by $(1+z)^r$ gives another generating function whose coefficients we know:

$$(1-z)^r (1+z)^r = (1-z^2)^r .$$

Equating coefficients of z^n now gives the equation

$$\sum_{k=0}^{n} \binom{r}{k} \binom{r}{n-k} (-1)^k = (-1)^{n/2} \binom{r}{n/2} [n \text{ even}] . \tag{5.55}$$

We should check this on a small case or two. When $n = 3$, for example, the result is

$$\binom{r}{0}\binom{r}{3} - \binom{r}{1}\binom{r}{2} + \binom{r}{2}\binom{r}{1} - \binom{r}{3}\binom{r}{0} = 0 .$$

Each positive term is cancelled by a corresponding negative term. And the same thing happens whenever n is odd, in which case the sum isn't very

interesting. But when n is even, say $n = 2$, we get a nontrivial sum that's different from Vandermonde's convolution:

$$\binom{r}{0}\binom{r}{2} - \binom{r}{1}\binom{r}{1} + \binom{r}{2}\binom{r}{0} = 2\binom{r}{2} - r^2 = -r\,.$$

So (5.55) checks out fine when $n = 2$. It turns out that (5.30) is a special case of our new identity (5.55).

Binomial coefficients also show up in some other generating functions, most notably the following important identities in which the lower index stays fixed and the upper index varies:

If you have a highlighter pen, these two equations have got to be marked.

$$\frac{1}{(1-z)^{n+1}} = \sum_{k\geqslant 0}\binom{n+k}{n}z^k\,, \qquad \text{integer } n \geqslant 0 \tag{5.56}$$

$$\frac{z^n}{(1-z)^{n+1}} = \sum_{k\geqslant 0}\binom{k}{n}z^k\,, \qquad \text{integer } n \geqslant 0. \tag{5.57}$$

The second identity here is just the first one multiplied by z^n, that is, "shifted right" by n places. The first identity is just a special case of the binomial theorem in slight disguise: If we expand $(1-z)^{-n-1}$ by (5.13), the coefficient of z^k is $\binom{-n-1}{k}(-1)^k$, which can be rewritten as $\binom{k+n}{k}$ or $\binom{n+k}{n}$ by negating the upper index. These special cases are worth noting explicitly, because they arise so frequently in applications.

When $n = 0$ we get a special case of a special case, the geometric series:

$$\frac{1}{1-z} = 1 + z + z^2 + z^3 + \cdots = \sum_{k\geqslant 0} z^k\,.$$

This is the generating function for the sequence $\langle 1, 1, 1, \ldots\rangle$, and it is especially useful because the convolution of any other sequence with this one is the sequence of sums: When $b_k = 1$ for all k, (5.54) reduces to

$$c_n = \sum_{k=0}^{n} a_k\,.$$

Therefore if $A(z)$ is the generating function for the summands $\langle a_0, a_1, a_2, \ldots\rangle$, then $A(z)/(1-z)$ is the generating function for the sums $\langle c_0, c_1, c_2, \ldots\rangle$.

The problem of derangements, which we solved by inversion in connection with hats and football fans, can be resolved with generating functions in an interesting way. The basic recurrence

$$n! = \sum_k \binom{n}{k}(n-k)_!$$

can be put into the form of a convolution if we expand $\binom{n}{k}$ in factorials and divide both sides by $n!$:

$$1 = \sum_{k=0}^{n} \frac{1}{k!} \frac{(n-k)_{\text{¡}}}{(n-k)!}.$$

The generating function for the sequence $\langle \frac{1}{0!}, \frac{1}{1!}, \frac{1}{2!}, \ldots \rangle$ is e^z; hence if we let

$$D(z) = \sum_{k \geqslant 0} \frac{k_{\text{¡}}}{k!} z^k,$$

the convolution/recurrence tells us that

$$\frac{1}{1-z} = e^z D(z).$$

Solving for $D(z)$ gives

$$D(z) = \frac{1}{1-z} e^{-z} = \frac{1}{1-z}\left(\frac{1}{0!}z^0 - \frac{1}{1!}z^1 + \frac{1}{2!}z^2 + \cdots\right).$$

Equating coefficients of z^n now tells us that

$$\frac{n_{\text{¡}}}{n!} = \sum_{k=0}^{n} \frac{(-1)^k}{k!};$$

this is the formula we derived earlier by inversion.

So far our explorations with generating functions have given us slick proofs of things that we already knew how to derive by more cumbersome methods. But we haven't used generating functions to obtain any new results, except for (5.55). Now we're ready for something new and more surprising. There are two families of power series that generate an especially rich class of binomial coefficient identities: Let us define the *generalized binomial series* $\mathcal{B}_t(z)$ and the *generalized exponential series* $\mathcal{E}_t(z)$ as follows:

$$\mathcal{B}_t(z) = \sum_{k \geqslant 0} (tk)^{\underline{k-1}} \frac{z^k}{k!}; \qquad \mathcal{E}_t(z) = \sum_{k \geqslant 0} (tk+1)^{k-1} \frac{z^k}{k!}. \tag{5.58}$$

It can be shown that these functions satisfy the identities

$$\mathcal{B}_t(z)^{1-t} - \mathcal{B}_t(z)^{-t} = z; \qquad \mathcal{E}_t(z)^{-t} \ln \mathcal{E}_t(z) = z. \tag{5.59}$$

In the special case $t = 0$, we have

$$\mathcal{B}_0(z) = 1 + z; \qquad \mathcal{E}_0(z) = e^z;$$

this explains why the series with parameter t are called "generalized" binomials and exponentials.

The generalized binomial series $\mathcal{B}_t(z)$ was discovered in the 1750s by J. H. Lambert [236, §38], who noticed a few years later [237] that its powers satisfy the first identity in (5.60).

The following pairs of identities are valid for all real r:

$$\mathcal{B}_t(z)^r = \sum_{k\geqslant 0}\binom{tk+r}{k}\frac{r}{tk+r}z^k;$$

$$\mathcal{E}_t(z)^r = \sum_{k\geqslant 0} r\frac{(tk+r)^{k-1}}{k!}z^k; \qquad (5.60)$$

$$\frac{\mathcal{B}_t(z)^r}{1-t+t\mathcal{B}_t(z)^{-1}} = \sum_{k\geqslant 0}\binom{tk+r}{k}z^k;$$

$$\frac{\mathcal{E}_t(z)^r}{1-zt\mathcal{E}_t(z)^t} = \sum_{k\geqslant 0}\frac{(tk+r)^k}{k!}z^k. \qquad (5.61)$$

(When $tk+r=0$, we have to be a little careful about how the coefficient of z^k is interpreted; each coefficient is a polynomial in r. For example, the constant term of $\mathcal{E}_t(z)^r$ is $r(0+r)^{-1}$, and this is equal to 1 even when $r=0$.)

Since equations (5.60) and (5.61) hold for all r, we get very general identities when we multiply together the series that correspond to different powers r and s. For example,

$$\begin{aligned}\mathcal{B}_t(z)^r\frac{\mathcal{B}_t(z)^s}{1-t+t\mathcal{B}_t(z)^{-1}} &= \sum_{k\geqslant 0}\binom{tk+r}{k}\frac{r}{tk+r}z^k\sum_{j\geqslant 0}\binom{tj+s}{j}z^j\\ &= \sum_{n\geqslant 0}z^n\sum_{k\geqslant 0}\binom{tk+r}{k}\frac{r}{tk+r}\binom{t(n-k)+s}{n-k}.\end{aligned}$$

This power series must equal

$$\frac{\mathcal{B}_t(z)^{r+s}}{1-t+t\mathcal{B}_t(z)^{-1}} = \sum_{n\geqslant 0}\binom{tn+r+s}{n}z^n;$$

hence we can equate coefficients of z^n and get the identity

$$\sum_k\binom{tk+r}{k}\binom{t(n-k)+s}{n-k}\frac{r}{tk+r} = \binom{tn+r+s}{n}, \qquad \text{integer } n,$$

valid for all real r, s, and t. When $t=0$ this identity reduces to Vandermonde's convolution. (If by chance $tk+r$ happens to equal zero in this formula, the denominator factor $tk+r$ should be considered to cancel with the $tk+r$ in the numerator of the binomial coefficient. Both sides of the identity are polynomials in r, s, and t.) Similar identities hold when we multiply $\mathcal{B}_t(z)^r$ by $\mathcal{B}_t(z)^s$, etc.; Table 202 presents the results.

Table 202 General convolution identities, valid for integer $n \geqslant 0$.

$$\sum_k \binom{tk+r}{k}\binom{tn-tk+s}{n-k}\frac{r}{tk+r} = \binom{tn+r+s}{n}. \tag{5.62}$$

$$\sum_k \binom{tk+r}{k}\binom{tn-tk+s}{n-k}\frac{r}{tk+r}\cdot\frac{s}{tn-tk+s} = \binom{tn+r+s}{n}\frac{r+s}{tn+r+s}. \tag{5.63}$$

$$\sum_k \binom{n}{k}(tk+r)^k(tn-tk+s)^{n-k}\frac{r}{tk+r} = (tn+r+s)^n. \tag{5.64}$$

$$\sum_k \binom{n}{k}(tk+r)^k(tn-tk+s)^{n-k}\frac{r}{tk+r}\cdot\frac{s}{tn-tk+s} = (tn+r+s)^n\frac{r+s}{tn+r+s}. \tag{5.65}$$

We have learned that it's generally a good idea to look at special cases of general results. What happens, for example, if we set $t = 1$? The generalized binomial $\mathcal{B}_1(z)$ is very simple — it's just

$$\mathcal{B}_1(z) = \sum_{k\geqslant 0} z^k = \frac{1}{1-z};$$

therefore $\mathcal{B}_1(z)$ doesn't give us anything we didn't already know from Vandermonde's convolution. But $\mathcal{E}_1(z)$ is an important function,

$$\mathcal{E}(z) = \sum_{k\geqslant 0}(k+1)^{k-1}\frac{z^k}{k!} = 1 + z + \frac{3}{2}z^2 + \frac{8}{3}z^3 + \frac{125}{24}z^4 + \cdots \tag{5.66}$$

which we haven't seen before; it satisfies the basic identity

$$\mathcal{E}(z) = e^{z\mathcal{E}(z)}. \tag{5.67}$$

Aha! This is the iterated power function $\mathcal{E}(\ln z) = z^{z^{z^{\cdot^{\cdot^{\cdot}}}}}$ that I've often wondered about.

This function, first studied by Euler [117] and Eisenstein [91], arises in a great many applications [203, 193].

Zzzzzz...

The special cases $t = 2$ and $t = -1$ of the generalized binomial are of particular interest, because their coefficients occur again and again in problems that have a recursive structure. Therefore it's useful to display these

The power series for $\mathcal{B}_{1/2}(z)^r = (\sqrt{z^2+4}+z)^{2r}/4^r$ is noteworthy too.

series explicitly for future reference:

$$\begin{aligned}\mathcal{B}_2(z) &= \sum_k \binom{2k}{k}\frac{z^k}{1+k} \\ &= \sum_k \binom{2k+1}{k}\frac{z^k}{1+2k} = \frac{1-\sqrt{1-4z}}{2z}. \qquad (5.68)\end{aligned}$$

$$\begin{aligned}\mathcal{B}_{-1}(z) &= \sum_k \binom{1-k}{k}\frac{z^k}{1-k} \\ &= \sum_k \binom{2k-1}{k}\frac{(-z)^k}{1-2k} = \frac{1+\sqrt{1+4z}}{2}. \qquad (5.69)\end{aligned}$$

$$\mathcal{B}_2(z)^r = \sum_k \binom{2k+r}{k}\frac{r}{2k+r}z^k. \qquad (5.70)$$

$$\mathcal{B}_{-1}(z)^r = \sum_k \binom{r-k}{k}\frac{r}{r-k}z^k. \qquad (5.71)$$

$$\frac{\mathcal{B}_2(z)^r}{\sqrt{1-4z}} = \sum_k \binom{2k+r}{k}z^k. \qquad (5.72)$$

$$\frac{\mathcal{B}_{-1}(z)^{r+1}}{\sqrt{1+4z}} = \sum_k \binom{r-k}{k}z^k. \qquad (5.73)$$

The coefficients $\binom{2n}{n}\frac{1}{n+1}$ of $\mathcal{B}_2(z)$ are called the *Catalan numbers* C_n, because Eugène Catalan wrote an influential paper about them in the 1830s [52]. The sequence begins as follows:

n	0	1	2	3	4	5	6	7	8	9	10
C_n	1	1	2	5	14	42	132	429	1430	4862	16796

The coefficients of $\mathcal{B}_{-1}(z)$ are essentially the same, but there's an extra 1 at the beginning and the other numbers alternate in sign: $\langle 1, 1, -1, 2, -5, 14, \ldots\rangle$. Thus $\mathcal{B}_{-1}(z) = 1 + z\mathcal{B}_2(-z)$. We also have $\mathcal{B}_{-1}(z) = \mathcal{B}_2(-z)^{-1}$.

Let's close this section by deriving an important consequence of (5.72) and (5.73), a relation that shows further connections between the functions $\mathcal{B}_{-1}(z)$ and $\mathcal{B}_2(-z)$:

$$\frac{\mathcal{B}_{-1}(z)^{n+1} - (-z)^{n+1}\mathcal{B}_2(-z)^{n+1}}{\sqrt{1+4z}} = \sum_{k\leqslant n}\binom{n-k}{k}z^k.$$

This holds because the coefficient of z^k in $(-z)^{n+1}\mathcal{B}_2(-z)^{n+1}/\sqrt{1+4z}$ is

$$\begin{aligned}
[z^k]\,\frac{(-z)^{n+1}\mathcal{B}_2(-z)^{n+1}}{\sqrt{1+4z}} &= (-1)^{n+1}[z^{k-n-1}]\,\frac{\mathcal{B}_2(-z)^{n+1}}{\sqrt{1+4z}} \\
&= (-1)^{n+1}(-1)^{k-n-1}[z^{k-n-1}]\,\frac{\mathcal{B}_2(z)^{n+1}}{\sqrt{1-4z}} \\
&= (-1)^k\binom{2(k-n-1)+n+1}{k-n-1} \\
&= (-1)^k\binom{2k-n-1}{k-n-1} \;=\; (-1)^k\binom{2k-n-1}{k} \\
&= \binom{n-k}{k} \;=\; [z^k]\,\frac{\mathcal{B}_{-1}(z)^{n+1}}{\sqrt{1+4z}}
\end{aligned}$$

when $k > n$. The terms nicely cancel each other out. We can now use (5.68) and (5.69) to obtain the closed form

$$\sum_{k\leqslant n}\binom{n-k}{k}z^k \;=\; \frac{1}{\sqrt{1+4z}}\left(\left(\frac{1+\sqrt{1+4z}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{1+4z}}{2}\right)^{n+1}\right), \qquad \text{integer } n \geqslant 0. \tag{5.74}$$

(The special case $z = -1$ came up in Problem 3 of Section 5.2. Since the numbers $\frac{1}{2}(1 \pm \sqrt{-3}\,)$ are sixth roots of unity, the sums $\sum_{k\leqslant n}\binom{n-k}{k}(-1)^k$ have the periodic behavior we observed in that problem.) Similarly we can combine (5.70) with (5.71) to cancel the large coefficients and get

$$\sum_{k<n}\binom{n-k}{k}\frac{n}{n-k}z^k \;=\; \left(\frac{1+\sqrt{1+4z}}{2}\right)^n + \left(\frac{1-\sqrt{1+4z}}{2}\right)^n, \qquad \text{integer } n > 0. \tag{5.75}$$

5.5 HYPERGEOMETRIC FUNCTIONS

The methods we've been applying to binomial coefficients are very effective, when they work, but we must admit that they often appear to be ad hoc — more like tricks than techniques. When we're working on a problem, we often have many directions to pursue, and we might find ourselves going around in circles. Binomial coefficients are like chameleons, changing their appearance easily. Therefore it's natural to ask if there isn't some unifying principle that will systematically handle a great variety of binomial coefficient summations all at once. Fortunately, the answer is yes. The unifying principle is based on the theory of certain infinite sums called *hypergeometric series*.

They're even more versatile than chameleons; we can dissect them and put them back together in different ways.

The study of hypergeometric series was launched many years ago by Euler, Gauss, and Riemann; such series, in fact, are still the subject of considerable research. But hypergeometrics have a somewhat formidable notation, which takes a little time to get used to.

Anything that has survived for centuries with such awesome notation must be really useful.

The general hypergeometric series is a power series in z with $m+n$ parameters, and it is defined as follows in terms of rising factorial powers:

$$F\left(\begin{matrix} a_1, \ldots, a_m \\ b_1, \ldots, b_n \end{matrix}\middle| z\right) = \sum_{k\geqslant 0} \frac{a_1^{\overline{k}} \ldots a_m^{\overline{k}}}{b_1^{\overline{k}} \ldots b_n^{\overline{k}}} \frac{z^k}{k!}. \tag{5.76}$$

To avoid division by zero, none of the b's may be zero or a negative integer. Other than that, the a's and b's may be anything we like. The notation '$F(a_1, \ldots, a_m; b_1, \ldots, b_n; z)$' is also used as an alternative to the two-line form (5.76), since a one-line form sometimes works better typographically. The a's are said to be *upper parameters*; they occur in the numerator of the terms of F. The b's are *lower parameters*, and they occur in the denominator. The final quantity z is called the *argument*.

Standard reference books often use '${}_mF_n$' instead of 'F' as the name of a hypergeometric with m upper parameters and n lower parameters. But the extra subscripts tend to clutter up the formulas and waste our time, if we're compelled to write them over and over. We can count how many parameters there are, so we usually don't need extra additional unnecessary redundancy.

Many important functions occur as special cases of the general hypergeometric; indeed, that's why hypergeometrics are so powerful. For example, the simplest case occurs when $m = n = 0$: There are no parameters at all, and we get the familiar series

$$F\left(\ \middle| z\right) = \sum_{k\geqslant 0} \frac{z^k}{k!} = e^z.$$

Actually the notation looks a bit unsettling when m or n is zero. We can add an extra '1' above and below in order to avoid this:

$$F\left(\begin{matrix} 1 \\ 1 \end{matrix}\middle| z\right) = e^z.$$

In general we don't change the function if we cancel a parameter that occurs in both numerator and denominator, or if we insert two identical parameters.

The next simplest case has $m = 1$, $a_1 = 1$, and $n = 0$; we change the parameters to $m = 2$, $a_1 = a_2 = 1$, $n = 1$, and $b_1 = 1$, so that $n > 0$. This series also turns out to be familiar, because $1^{\overline{k}} = k!$:

$$F\left(\begin{matrix} 1, 1 \\ 1 \end{matrix}\middle| z\right) = \sum_{k\geqslant 0} z^k = \frac{1}{1-z}.$$

It's our old friend, the geometric series; $F(a_1,\ldots,a_m;b_1,\ldots,b_n;z)$ is called hypergeometric because it includes the geometric series $F(1,1;1;z)$ as a very special case.

The general case $m=1$ and $n=0$ is, in fact, easy to sum in closed form,

$$F\left(\begin{matrix}a,\,1\\1\end{matrix}\middle|\,z\right) = \sum_{k\geqslant 0} a^{\overline{k}}\,\frac{z^k}{k!} = \sum_k \binom{a+k-1}{k} z^k = \frac{1}{(1-z)^a}\,, \tag{5.77}$$

using (5.56). If we replace a by $-a$ and z by $-z$, we get the binomial theorem,

$$F\left(\begin{matrix}-a,\,1\\1\end{matrix}\middle|\,-z\right) = (1+z)^a\,.$$

A negative integer as upper parameter causes the infinite series to become finite, since $(-a)^{\overline{k}}=0$ whenever $k>a\geqslant 0$ and a is an integer.

The general case $m=0$, $n=1$ is another famous series, but it's not as well known in the literature of discrete mathematics:

$$F\left(\begin{matrix}1\\b,1\end{matrix}\middle|\,z\right) = \sum_{k\geqslant 0}\frac{(b-1)!}{(b-1+k)!}\,\frac{z^k}{k!} = I_{b-1}\big(2\sqrt{z}\big)\,\frac{(b-1)!}{z^{(b-1)/2}}\,. \tag{5.78}$$

This function I_{b-1} is called a "modified Bessel function" of order $b-1$. The special case $b=1$ gives us $F\big(\begin{smallmatrix}1\\1,1\end{smallmatrix}\big|z\big)=I_0(2\sqrt{z})$, which is the interesting series $\sum_{k\geqslant 0} z^k/k!^2$.

The special case $m=n=1$ is called a "confluent hypergeometric series" and often denoted by the letter M:

$$F\left(\begin{matrix}a\\b\end{matrix}\middle|\,z\right) = \sum_{k\geqslant 0}\frac{a^{\overline{k}}}{b^{\overline{k}}}\,\frac{z^k}{k!} = M(a,b,z)\,. \tag{5.79}$$

This function, which has important applications to engineering, was introduced by Ernst Kummer.

By now a few of us are wondering why we haven't discussed convergence of the infinite series (5.76). The answer is that we can ignore convergence if we are using z simply as a formal symbol. It is not difficult to verify that formal infinite sums of the form $\sum_{k\geqslant n}\alpha_k z^k$ form a field, if the coefficients α_k lie in a field. We can add, subtract, multiply, divide, differentiate, and do functional composition on such formal sums without worrying about convergence; any identities we derive will still be formally true. For example, the hypergeometric $F\big(\begin{smallmatrix}1,1,1\\1\end{smallmatrix}\big|z\big)=\sum_{k\geqslant 0}k!\,z^k$ doesn't converge for any nonzero z; yet we'll see in Chapter 7 that we can still use it to solve problems. On the other hand, whenever we replace z by a particular numerical value, we do have to be sure that the infinite sum is well defined.

The next step up in complication is actually the most famous hypergeometric of all. In fact, it was *the* hypergeometric series until about 1870, when everything was generalized to arbitrary m and n. This one has two upper parameters and one lower parameter:

$$F\left(\begin{matrix} a, b \\ c \end{matrix}\middle| z\right) = \sum_{k\geqslant 0} \frac{a^{\overline{k}} b^{\overline{k}} z^k}{c^{\overline{k}}\, k!}. \tag{5.80}$$

It is often called the Gaussian hypergeometric, because many of its subtle properties were first proved by Gauss in his doctoral dissertation of 1812 [143], although Euler [118] and Pfaff [292] had already discovered some remarkable things about it. One of its important special cases is

"There must be many universities to-day where 95 per cent, if not 100 per cent, of the functions studied by physics, engineering, and even mathematics students, are covered by this single symbol $F(a, b; c; x)$."
— W. W. Sawyer [318]

$$\begin{aligned} \ln(1+z) &= z\,F\left(\begin{matrix} 1, 1 \\ 2 \end{matrix}\middle| -z\right) = z \sum_{k\geqslant 0} \frac{k!\,k!}{(k+1)!} \frac{(-z)^k}{k!} \\ &= z - \frac{z^2}{2} + \frac{z^3}{3} - \frac{z^4}{4} + \cdots. \end{aligned}$$

Notice that $z^{-1}\ln(1+z)$ is a hypergeometric function, but $\ln(1+z)$ itself cannot be hypergeometric, since a hypergeometric series always has the value 1 when $z = 0$.

So far hypergeometrics haven't actually done anything for us except provide an excuse for name-dropping. But we've seen that several very different functions can all be regarded as hypergeometric; this will be the main point of interest in what follows. We'll see that a large class of sums can be written as hypergeometric series in a "canonical" way, hence we will have a good filing system for facts about binomial coefficients.

What series are hypergeometric? It's easy to answer this question if we look at the ratio between consecutive terms:

$$F\left(\begin{matrix} a_1, \ldots, a_m \\ b_1, \ldots, b_n \end{matrix}\middle| z\right) = \sum_{k\geqslant 0} t_k, \qquad t_k = \frac{a_1^{\overline{k}} \ldots a_m^{\overline{k}}\, z^k}{b_1^{\overline{k}} \ldots b_n^{\overline{k}}\, k!}.$$

The first term is $t_0 = 1$, and the other terms have ratios given by

$$\begin{aligned} \frac{t_{k+1}}{t_k} &= \frac{a_1^{\overline{k+1}} \ldots a_m^{\overline{k+1}}}{a_1^{\overline{k}} \ldots a_m^{\overline{k}}} \frac{b_1^{\overline{k}} \ldots b_n^{\overline{k}}}{b_1^{\overline{k+1}} \ldots b_n^{\overline{k+1}}} \frac{k!}{(k+1)!} \frac{z^{k+1}}{z^k} \\ &= \frac{(k+a_1)\ldots(k+a_m)\,z}{(k+b_1)\ldots(k+b_n)(k+1)}. \end{aligned} \tag{5.81}$$

This is a *rational function* of k, that is, a quotient of polynomials in k. According to the Fundamental Theorem of Algebra, any rational function

of k can be factored over the complex numbers and put into this form. The a's are the negatives of the roots of the polynomial in the numerator, and the b's are the negatives of the roots of the polynomial in the denominator. If the denominator doesn't already contain the special factor $(k+1)$, we can include $(k+1)$ in both numerator and denominator. A constant factor remains, and we can call it z. Therefore hypergeometric series are precisely those series whose first term is 1 and whose term ratio t_{k+1}/t_k is a rational function of k.

Suppose, for example, that we're given an infinite series with term ratio

$$\frac{t_{k+1}}{t_k} = \frac{k^2+7k+10}{4k^2+1},$$

a rational function of k. The numerator polynomial splits nicely into two factors, $(k+2)(k+5)$, and the denominator is $4(k+i/2)(k-i/2)$. Since the denominator is missing the required factor $(k+1)$, we write the term ratio as

$$\frac{t_{k+1}}{t_k} = \frac{(k+2)(k+5)(k+1)(1/4)}{(k+i/2)(k-i/2)(k+1)},$$

and we can read off the results: The given series is

$$\sum_{k\geqslant 0} t_k = t_0\, F\left(\begin{matrix}2,\,5,\,1\\ i/2,\,-i/2\end{matrix}\,\middle|\,1/4\right).$$

Thus, we have a general method for finding the hypergeometric representation of a given quantity S, when such a representation is possible: First we write S as an infinite series whose first term is nonzero. We choose a notation so that the series is $\sum_{k\geqslant 0} t_k$ with $t_0 \neq 0$. Then we calculate t_{k+1}/t_k. If the term ratio is not a rational function of k, we're out of luck. Otherwise we express it in the form (5.81); this gives parameters $a_1, \ldots, a_m, b_1, \ldots, b_n$, and an argument z, such that $S = t_0\, F(a_1, \ldots, a_m; b_1, \ldots, b_n; z)$.

(Now is a good time to do warmup exercise 11.)

Gauss's hypergeometric series can be written in the recursively factored form

$$F\left(\begin{matrix}a,b\\ c\end{matrix}\,\middle|\,z\right) = 1 + \frac{a}{1}\frac{b}{c}z\left(1 + \frac{a+1}{2}\frac{b+1}{c+1}z\left(1 + \frac{a+2}{3}\frac{b+2}{c+2}z(1+\cdots)\right)\right)$$

if we wish to emphasize the importance of term ratios.

Let's try now to reformulate the binomial coefficient identities derived earlier in this chapter, expressing them as hypergeometrics. For example, let's figure out what the parallel summation law,

$$\sum_{k\leqslant n}\binom{r+k}{k} = \binom{r+n+1}{n}, \qquad \text{integer } n,$$

looks like in hypergeometric notation. We need to write the sum as an infinite series that starts at $k = 0$, so we replace k by $n - k$:

$$\sum_{k\geqslant 0}\binom{r+n-k}{n-k} = \sum_{k\geqslant 0}\frac{(r+n-k)!}{r!\,(n-k)!} = \sum_{k\geqslant 0} t_k\,.$$

This series is formally infinite but actually finite, because the $(n-k)!$ in the denominator will make $t_k = 0$ when $k > n$. (We'll see later that $1/x!$ is defined for all x, and that $1/x! = 0$ when x is a negative integer. But for now, let's blithely disregard such technicalities until we gain more hypergeometric experience.) The term ratio is

$$\begin{aligned}\frac{t_{k+1}}{t_k} &= \frac{(r+n-k-1)!\,r!\,(n-k)!}{r!\,(n-k-1)!\,(r+n-k)!} = \frac{n-k}{r+n-k}\\ &= \frac{(k+1)(k-n)(1)}{(k-n-r)(k+1)}\,.\end{aligned}$$

Furthermore $t_0 = \binom{r+n}{n}$. Hence the parallel summation law is equivalent to the hypergeometric identity

$$\binom{r+n}{n} F\left(\begin{matrix}1,\ -n\\ -n-r\end{matrix}\middle|\,1\right) = \binom{r+n+1}{n}\,.$$

Dividing through by $\binom{r+n}{n}$ gives a slightly simpler version,

$$F\left(\begin{matrix}1,\ -n\\ -n-r\end{matrix}\middle|\,1\right) = \frac{r+n+1}{r+1}\,, \qquad \text{if } \binom{r+n}{n} \neq 0. \tag{5.82}$$

Let's do another one. The term ratio of identity (5.16),

$$\sum_{k\leqslant m}\binom{r}{k}(-1)^k = (-1)^m\binom{r-1}{m}\,, \qquad \text{integer } m,$$

is $(k-m)/(r-m+k+1) = (k+1)(k-m)(1)/(k-m+r+1)(k+1)$, after we replace k by $m-k$; hence (5.16) gives a closed form for

$$F\left(\begin{matrix}1,\ -m\\ -m+r+1\end{matrix}\middle|\,1\right).$$

This is essentially the same as the hypergeometric function on the left of (5.82), but with m in place of n and $r+1$ in place of $-r$. Therefore identity (5.16) could have been derived from (5.82), the hypergeometric version of (5.9). (No wonder we found it easy to prove (5.16) by using (5.9).)

First derangements, now degenerates.

Before we go further, we should think about degenerate cases, because hypergeometrics are not defined when a lower parameter is zero or a negative

integer. We usually apply the parallel summation identity when r and n are positive integers; but then $-n-r$ is a negative integer and the hypergeometric (5.76) is undefined. How then can we consider (5.82) to be legitimate? The answer is that we can take the limit of $F\left(\begin{smallmatrix}1,-n\\-n-r+\epsilon\end{smallmatrix}\middle|1\right)$ as $\epsilon \to 0$.

We will look at such things more closely later in this chapter, but for now let's just be aware that some denominators can be dynamite. It is interesting, however, that the very first sum we've tried to express hypergeometrically has turned out to be degenerate.

(We proved the identities originally for integer r, and used the polynomial argument to show that they hold in general. Now we're proving them first for irrational r, and using a limiting argument to show that they hold for integers!)

Another possibly sore point in our derivation of (5.82) is that we expanded $\binom{r+n-k}{n-k}$ as $(r+n-k)!/r!\,(n-k)!$. This expansion fails when r is a negative integer, because $(-m)!$ has to be ∞ if the law

$$0! \;=\; 0\cdot(-1)\cdot(-2)\cdot\ldots\cdot(-m+1)\cdot(-m)!$$

is going to hold. Again, we need to approach integer results by considering a limit of $r+\epsilon$ as $\epsilon \to 0$.

But we defined the factorial representation $\binom{r}{k} = r!/k!\,(r-k)!$ only when r is an integer! If we want to work effectively with hypergeometrics, we need a factorial function that is defined for all complex numbers. Fortunately there is such a function, and it can be defined in many ways. Here's one of the most useful definitions of $z!$, actually a definition of $1/z!$:

$$\frac{1}{z!} \;=\; \lim_{n\to\infty}\binom{n+z}{n}n^{-z}\,. \tag{5.83}$$

(See exercise 21. Euler [99, 100, 72] discovered this when he was 22 years old.) The limit can be shown to exist for all complex z, and it is zero only when z is a negative integer. Another significant definition is

$$z! \;=\; \int_0^\infty t^z e^{-t}\,dt\,, \qquad \text{if } \Re z > -1. \tag{5.84}$$

This integral exists only when the real part of z exceeds -1, but we can use the formula

$$z! \;=\; z\,(z-1)! \tag{5.85}$$

to extend the definition to all complex z (except negative integers). Still another definition comes from Stirling's interpolation of $\ln z!$ in (5.47). All of these approaches lead to the same generalized factorial function.

There's a very similar function called the *Gamma function*, which relates to ordinary factorials somewhat as rising powers relate to falling powers. Standard reference books often use factorials and Gamma functions simultaneously, and it's convenient to convert between them if necessary using the

following formulas:

$$\Gamma(z+1) = z!\,; \tag{5.86}$$

$$(-z)!\,\Gamma(z) = \frac{\pi}{\sin \pi z}\,. \tag{5.87}$$

How do you write z to the $\overline{w}$ power, when $\overline{w}$ is the complex conjugate of w?

$z^{(\overline{w})}$.

We can use these generalized factorials to define generalized factorial powers, when z and w are arbitrary complex numbers:

$$z^{\underline{w}} = \frac{z!}{(z-w)!}\,; \tag{5.88}$$

$$z^{\overline{w}} = \frac{\Gamma(z+w)}{\Gamma(z)}\,. \tag{5.89}$$

The only proviso is that we must use appropriate limiting values when these formulas give ∞/∞. (The formulas never give $0/0$, because factorials and Gamma-function values are never zero.) A binomial coefficient can be written

$$\binom{z}{w} = \lim_{\zeta\to z}\lim_{\omega\to w}\frac{\zeta!}{\omega!\,(\zeta-\omega)!} \tag{5.90}$$

I see, the lower index arrives at its limit first. That's why $\binom{z}{w}$ is zero when w is a negative integer.

when z and w are any complex numbers whatever.

Armed with generalized factorial tools, we can return to our goal of reducing the identities derived earlier to their hypergeometric essences. The binomial theorem (5.13) turns out to be neither more nor less than (5.77), as we might expect. So the next most interesting identity to try is Vandermonde's convolution (5.27):

$$\sum_k \binom{r}{k}\binom{s}{n-k} = \binom{r+s}{n}, \qquad \text{integer } n.$$

The kth term here is

$$t_k = \frac{r!}{(r-k)!\,k!}\,\frac{s!}{(s-n+k)!\,(n-k)!}\,,$$

and we are no longer too shy to use generalized factorials in these expressions. Whenever t_k contains a factor like $(\alpha+k)!$, with a plus sign before the k, we get $(\alpha+k+1)!/(\alpha+k)! = k+\alpha+1$ in the term ratio t_{k+1}/t_k, by (5.85); this contributes the parameter '$\alpha+1$' to the corresponding hypergeometric — as an upper parameter if $(\alpha+k)!$ was in the numerator of t_k, but as a lower parameter otherwise. Similarly, a factor like $(\alpha-k)!$ leads to $(\alpha-k-1)!/(\alpha-k)! = (-1)/(k-\alpha)$; this contributes '$-\alpha$' to the opposite set of parameters (reversing the roles of upper and lower), and negates the hypergeometric argument. Factors like $r!$, which are independent of k, go

into t_0 but disappear from the term ratio. Using such tricks we can predict without further calculation that the term ratio of (5.27) is

$$\frac{t_{k+1}}{t_k} = \frac{k-r}{k+1}\,\frac{k-n}{k+s-n+1}$$

times $(-1)^2 = 1$, and Vandermonde's convolution becomes

$$\binom{s}{n} F\left(\begin{matrix}-r,\ -n\\ s-n+1\end{matrix}\middle|\,1\right) = \binom{r+s}{n}. \tag{5.91}$$

We can use this equation to determine $F(a, b; c; z)$ in general, when $z = 1$ and when b is a negative integer.

Let's rewrite (5.91) in a form so that table lookup is easy when a new sum needs to be evaluated. The result turns out to be

$$F\left(\begin{matrix}a, b\\ c\end{matrix}\middle|\,1\right) = \frac{\Gamma(c-a-b)\,\Gamma(c)}{\Gamma(c-a)\,\Gamma(c-b)}; \qquad \begin{matrix}\text{integer } b \leqslant 0\\ \text{or } \Re c > \Re a + \Re b.\end{matrix} \tag{5.92}$$

Vandermonde's convolution (5.27) covers only the case that one of the upper parameters, say b, is a nonpositive integer; but Gauss proved that (5.92) is valid also when a, b, c are complex numbers whose real parts satisfy $\Re c > \Re a + \Re b$. In other cases, the infinite series $F\left(\begin{smallmatrix}a,b\\ c\end{smallmatrix}\middle|\,1\right)$ doesn't converge. When $b = -n$, the identity can be written more conveniently with factorial powers instead of Gamma functions:

A few weeks ago, we were studying what Gauss had done in kindergarten. Now we're studying stuff beyond his Ph.D. thesis. Is this intimidating or what?

$$F\left(\begin{matrix}a, -n\\ c\end{matrix}\middle|\,1\right) = \frac{(c-a)^{\overline{n}}}{c^{\overline{n}}} = \frac{(a-c)^{\underline{n}}}{(-c)^{\underline{n}}}, \qquad \text{integer } n \geqslant 0. \tag{5.93}$$

It turns out that all five of the identities in Table 169 are special cases of Vandermonde's convolution; formula (5.93) covers them all, when proper attention is paid to degenerate situations.

Notice that (5.82) is just the special case $a = 1$ of (5.93). Therefore we don't really need to remember (5.82); and we don't really need the identity (5.9) that led us to (5.82), even though Table 174 said that it was memorable. A computer program for formula manipulation, faced with the problem of evaluating $\sum_{k\leqslant n}\binom{r+k}{k}$, could convert the sum to a hypergeometric and plug into the general identity for Vandermonde's convolution.

Problem 1 in Section 5.2 asked for the value of

$$\sum_{k\geqslant 0}\binom{m}{k}\Big/\binom{n}{k}.$$

This problem is a natural for hypergeometrics, and after a bit of practice any hypergeometer can read off the parameters immediately as $F(1, -m; -n; 1)$. Hmmm; that problem was yet another special takeoff on Vandermonde!

The sum in Problem 2 and Problem 4 likewise yields $F(2, 1-n; 2-m; 1)$. (We need to replace k by $k+1$ first.) And the "menacing" sum in Problem 6 turns out to be just $F(n+1, -n; 2; 1)$. Is there nothing more to sum, besides disguised versions of Vandermonde's powerful convolution?

Well, yes, Problem 3 is a bit different. It deals with a special case of the general sum $\sum_k \binom{n-k}{k} z^k$ considered in (5.74), and this leads to a closed-form expression for

$$F\left(\begin{matrix} 1+2\lceil n/2\rceil, -n \\ 1/2 \end{matrix} \middle| -z/4\right).$$

We also proved something new in (5.55), when we looked at the coefficients of $(1-z)^r(1+z)^r$:

$$F\left(\begin{matrix} 1-c-2n, -2n \\ c \end{matrix} \middle| -1\right) = (-1)^n \frac{(2n)!}{n!} \frac{(c-1)!}{(c+n-1)!}, \qquad \text{integer } n \geqslant 0.$$

Kummer was a summer.

This is called *Kummer's formula* when it's generalized to complex numbers:

$$F\left(\begin{matrix} a, b \\ 1+b-a \end{matrix} \middle| -1\right) = \frac{(b/2)!}{b!}(b-a)^{\underline{b/2}}. \tag{5.94}$$

The summer of '36.

(Ernst Kummer [229] proved this in 1836.)

It's interesting to compare these two formulas. Replacing c by $1-2n-a$, we find that the results are consistent if and only if

$$(-1)^n \frac{(2n)!}{n!} = \lim_{b\to -2n} \frac{(b/2)!}{b!} = \lim_{x\to -n} \frac{x!}{(2x)!} \tag{5.95}$$

when n is a positive integer. Suppose, for example, that $n = 3$; then we should have $-6!/3! = \lim_{x\to -3} x!/(2x)!$. We know that $(-3)!$ and $(-6)!$ are both infinite; but we might choose to ignore that difficulty and to imagine that $(-3)! = (-3)(-4)(-5)(-6)!$, so that the two occurrences of $(-6)!$ will cancel. Such temptations must, however, be resisted, because they lead to the wrong answer! The limit of $x!/(2x)!$ as $x \to -3$ is not $(-3)(-4)(-5)$ but rather $-6!/3! = (-4)(-5)(-6)$, according to (5.95).

The right way to evaluate the limit in (5.95) is to use equation (5.87), which relates negative-argument factorials to positive-argument Gamma functions. If we replace x by $-n-\epsilon$ and let $\epsilon \to 0$, two applications of (5.87) give

$$\frac{(-n-\epsilon)!}{(-2n-2\epsilon)!} \frac{\Gamma(n+\epsilon)}{\Gamma(2n+2\epsilon)} = \frac{\sin(2n+2\epsilon)\pi}{\sin(n+\epsilon)\pi}.$$

Now $\sin(x+y) = \sin x \cos y + \cos x \sin y$; so this ratio of sines is

$$\frac{\cos 2n\pi \sin 2\epsilon\pi}{\cos n\pi \sin \epsilon\pi} = (-1)^n\big(2 + O(\epsilon)\big),$$

by the methods of Chapter 9. Therefore, by (5.86), we have

$$\lim_{\epsilon\to 0} \frac{(-n-\epsilon)!}{(-2n-2\epsilon)!} = 2(-1)^n \frac{\Gamma(2n)}{\Gamma(n)} = 2(-1)^n \frac{(2n-1)!}{(n-1)!} = (-1)^n \frac{(2n)!}{n!},$$

as desired.

Let's complete our survey by restating the other identities we've seen so far in this chapter, clothing them in hypergeometric garb. The triple-binomial sum in (5.29) can be written

$$F\left(\begin{matrix} 1-a-2n,\ 1-b-2n,\ -2n \\ a,\ b \end{matrix}\middle|\, 1\right)$$
$$= (-1)^n \frac{(2n)!}{n!} \frac{(a+b+2n-1)^{\overline{n}}}{a^{\overline{n}}\, b^{\overline{n}}}, \qquad \text{integer } n \geqslant 0.$$

When this one is generalized to complex numbers, it is called *Dixon's formula*:

$$F\left(\begin{matrix} a,\ b,\ c \\ 1+c-a,\ 1+c-b \end{matrix}\middle|\, 1\right) = \frac{(c/2)!}{c!} \frac{(c-a)^{\underline{c/2}}\,(c-b)^{\underline{c/2}}}{(c-a-b)^{\underline{c/2}}}, \qquad (5.96)$$
$$\Re a + \Re b < 1 + \Re c/2.$$

One of the most general formulas we've encountered is the triple-binomial sum (5.28), which yields *Saalschütz's identity*:

(Historical note: Saalschütz [315] independently discovered this formula almost 100 years after Pfaff [292] had first published it. Taking the limit as $n \to \infty$ yields equation (5.92).)

$$F\left(\begin{matrix} a,\ b,\ -n \\ c,\ a+b-c-n+1 \end{matrix}\middle|\, 1\right) = \frac{(c-a)^{\overline{n}}\,(c-b)^{\overline{n}}}{c^{\overline{n}}\,(c-a-b)^{\overline{n}}} \qquad (5.97)$$
$$= \frac{(a-c)^{\underline{n}}\,(b-c)^{\underline{n}}}{(-c)^{\underline{n}}\,(a+b-c)^{\underline{n}}}, \qquad \text{integer } n \geqslant 0.$$

This formula gives the value at $z = 1$ of the general hypergeometric series with three upper parameters and two lower parameters, provided that one of the upper parameters is a nonpositive integer and that $b_1 + b_2 = a_1 + a_2 + a_3 + 1$. (If the sum of the lower parameters exceeds the sum of the upper parameters by 2 instead of by 1, the formula of exercise 25 can be used to express $F(a_1, a_2, a_3; b_1, b_2; 1)$ in terms of two hypergeometrics that satisfy Saalschütz's identity.)

Our hard-won identity in Problem 8 of Section 5.2 reduces to

$$\frac{1}{1+x} F\left(\begin{matrix} x+1,\ n+1,\ -n \\ 1,\ x+2 \end{matrix}\middle|\, 1\right) = (-1)^n x^{\underline{n}}\, x^{\underline{-n-1}}.$$

Sigh. This is just the special case $c = 1$ of Saalschütz's identity (5.97), so we could have saved a lot of work by going to hypergeometrics directly!

What about Problem 7? That extra-menacing sum gives us the formula

$$F\left(\begin{matrix} n+1,\ m-n,\ 1,\ \frac{1}{2} \\ \frac{1}{2}m+1,\ \frac{1}{2}m+\frac{1}{2},\ 2 \end{matrix}\middle|\, 1\right) = \frac{m}{n},$$

which is the first case we've seen with three lower parameters. So it looks new. But it really isn't; the left-hand side can be replaced by

$$F\left(\begin{matrix} n,\ m-n-1,\ -\frac{1}{2} \\ \frac{1}{2}m,\ \frac{1}{2}m-\frac{1}{2} \end{matrix}\middle|\, 1\right) - 1,$$

using exercise 26, and Saalschütz's identity wins again.

(Historical note: The great relevance of hypergeometric series to binomial coefficient identities was first pointed out by George Andrews in 1974 [9, section 5].)

Well, that's another deflating experience, but it's also another reason to appreciate the power of hypergeometric methods.

The convolution identities in Table 202 do not have hypergeometric equivalents, because their term ratios are rational functions of k only when t is an integer. Equations (5.64) and (5.65) aren't hypergeometric even when $t = 1$. But we can take note of what (5.62) tells us when t has small integer values:

$$F\left(\begin{matrix} \frac{1}{2}r,\ \frac{1}{2}r+\frac{1}{2},\ -n,\ -n-s \\ r+1,\ -n-\frac{1}{2}s,\ -n-\frac{1}{2}s+\frac{1}{2} \end{matrix}\middle|\, 1\right) = \binom{r+s+2n}{n}\Big/\binom{s+2n}{n};$$

$$F\left(\begin{matrix} \frac{1}{3}r,\ \frac{1}{3}r+\frac{1}{3},\ \frac{1}{3}r+\frac{2}{3},\ -n,\ -n-\frac{1}{2}s,\ -n-\frac{1}{2}s-\frac{1}{2} \\ \frac{1}{2}r+\frac{1}{2},\ \frac{1}{2}r+1,\ -n-\frac{1}{3}s,\ -n-\frac{1}{3}s+\frac{1}{3},\ -n-\frac{1}{3}s+\frac{2}{3} \end{matrix}\middle|\, 1\right)$$
$$= \binom{r+s+3n}{n}\Big/\binom{s+3n}{n}.$$

The first of these formulas gives the result of Problem 7 again, when the quantities (r, s, n) are replaced respectively by $(1, 2n+1-m, -1-n)$.

Finally, the "unexpected" sum (5.20) gives us an unexpected hypergeometric identity that turns out to be quite instructive. Let's look at it in slow motion. First we convert to an infinite sum,

$$\sum_{k\leqslant m}\binom{m+k}{k}2^{-k} = 2^m \qquad\Longleftrightarrow\qquad \sum_{k\geqslant 0}\binom{2m-k}{m-k}2^k = 2^{2m}.$$

The term ratio from $(2m-k)!\,2^k/m!\,(m-k)!$ is $2(k-m)/(k-2m)$, so we have a hypergeometric identity with $z = 2$:

$$\binom{2m}{m}F\left(\begin{matrix} 1,\ -m \\ -2m \end{matrix}\middle|\, 2\right) = 2^{2m}, \qquad \text{integer } m \geqslant 0. \tag{5.98}$$

But look at the lower parameter '$-2m$'. Negative integers are verboten, so this identity is undefined!

It's high time to look at such limiting cases carefully, as promised earlier, because degenerate hypergeometrics can often be evaluated by approaching them from nearby nondegenerate points. We must be careful when we do this, because different results can be obtained if we take limits in different ways. For example, here are two limits that turn out to be quite different when one of the upper parameters is increased by ϵ:

$$\begin{aligned}
\lim_{\epsilon\to 0} F\left(\begin{matrix}-1+\epsilon,\ -3\\ -2+\epsilon\end{matrix}\middle|\,1\right) &= \lim_{\epsilon\to 0}\Big(1+\tfrac{(-1+\epsilon)(-3)}{(-2+\epsilon)\,1!}+\tfrac{(-1+\epsilon)(\epsilon)(-3)(-2)}{(-2+\epsilon)(-1+\epsilon)\,2!}\\
&\qquad\qquad +\tfrac{(-1+\epsilon)(\epsilon)(1+\epsilon)(-3)(-2)(-1)}{(-2+\epsilon)(-1+\epsilon)(\epsilon)\,3!}\Big)\\
&= 1-\tfrac{3}{2}+0+\tfrac{1}{2} \;=\; 0;\\
\lim_{\epsilon\to 0} F\left(\begin{matrix}-1,\ -3\\ -2+\epsilon\end{matrix}\middle|\,1\right) &= \lim_{\epsilon\to 0}\Big(1+\tfrac{(-1)(-3)}{(-2+\epsilon)\,1!}+0+0\Big)\\
&= 1-\tfrac{3}{2}+0+0 \;=\; -\tfrac{1}{2}\,.
\end{aligned}$$

Similarly, we have defined $\binom{-1}{-1} = 0 = \lim_{\epsilon\to 0}\binom{-1+\epsilon}{-1}$; this is not the same as $\lim_{\epsilon\to 0}\binom{-1+\epsilon}{-1+\epsilon} = 1$. The proper way to treat (5.98) as a limit is to realize that the upper parameter $-m$ is being used to make all terms of the series $\sum_{k\geqslant 0}\binom{2m-k}{m-k}2^k$ zero for $k > m$; this means that we want to make the following more precise statement:

$$\binom{2m}{m}\lim_{\epsilon\to 0} F\left(\begin{matrix}1,\ -m\\ -2m+\epsilon\end{matrix}\middle|\,2\right) = 2^{2m}, \qquad \text{integer } m \geqslant 0. \tag{5.99}$$

Each term of this limit is well defined, because the denominator factor $(-2m)^{\overline{k}}$ does not become zero until $k > 2m$. Therefore this limit gives us exactly the sum (5.20) we began with.

5.6 HYPERGEOMETRIC TRANSFORMATIONS

It should be clear by now that a database of known hypergeometric closed forms is a useful tool for doing sums of binomial coefficients. We simply convert any given sum into its canonical hypergeometric form, then look it up in the table. If it's there, fine, we've got the answer. If not, we can add it to the database if the sum turns out to be expressible in closed form. We might also include entries in the table that say, "This sum does not have a simple closed form in general." For example, the sum $\sum_{k\leqslant m}\binom{n}{k}$

corresponds to the hypergeometric

$$\binom{n}{m} F\left(\begin{matrix}1, -m\\ n-m+1\end{matrix}\middle| -1\right), \qquad \text{integers } n \geqslant m \geqslant 0; \tag{5.100}$$

this has a simple closed form only if m is near 0, $\frac{1}{2}n$, or n.

But there's more to the story, since hypergeometric functions also obey identities of their own. This means that every closed form for hypergeometrics leads to additional closed forms and to additional entries in the database. For example, the identities in exercises 25 and 26 tell us how to transform one hypergeometric into two others with similar but different parameters. These can in turn be transformed again.

The hypergeometric database should really be a "knowledge base."

In 1797, J. F. Pfaff [292] discovered a surprising *reflection law*,

$$\frac{1}{(1-z)^a} F\left(\begin{matrix}a, b\\ c\end{matrix}\middle| \frac{-z}{1-z}\right) = F\left(\begin{matrix}a, c-b\\ c\end{matrix}\middle| z\right), \tag{5.101}$$

which is a transformation of another type. This is a formal identity in power series, if the quantity $(-z)^k/(1-z)^{k+a}$ is replaced by the infinite series $(-z)^k\big(1 + \binom{k+a}{1}z + \binom{k+a+1}{2}z^2 + \cdots\big)$ when the left-hand side is expanded (see exercise 50). We can use this law to derive new formulas from the identities we already know, when $z \neq 1$.

For example, Kummer's formula (5.94) can be combined with the reflection law (5.101) if we choose the parameters so that both identities apply:

$$\begin{aligned} 2^{-a} F\left(\begin{matrix}a, 1-a\\ 1+b-a\end{matrix}\middle| \frac{1}{2}\right) &= F\left(\begin{matrix}a, b\\ 1+b-a\end{matrix}\middle| -1\right) \\ &= \frac{(b/2)!}{b!}(b-a)^{\underline{b/2}}. \end{aligned} \tag{5.102}$$

We can now set $a = -n$ and go back from this equation to a new identity in binomial coefficients that we might need some day:

$$\begin{aligned} \sum_{k\geqslant 0} \frac{(-n)^{\overline{k}}\,(1+n)^{\overline{k}}}{(1+b+n)^{\overline{k}}}\,\frac{2^{-k}}{k!} &= \sum_k \binom{n}{k}\left(\frac{-1}{2}\right)^k \binom{n+k}{k}\Big/\binom{n+b+k}{k} \\ &= 2^{-n}\frac{(b/2)!\,(b+n)!}{b!\,(b/2+n)!}, \quad \text{integer } n \geqslant 0. \end{aligned} \tag{5.103}$$

For example, when $n = 3$ this identity says that

$$\begin{aligned} 1 - 3\,\frac{4}{2(4+b)} + 3\,\frac{4\cdot 5}{4(4+b)(5+b)} - \frac{4\cdot 5\cdot 6}{8(4+b)(5+b)(6+b)} \\ = \frac{(b+3)(b+2)(b+1)}{(b+6)(b+4)(b+2)}. \end{aligned}$$

It's almost unbelievable, but true, for all b. (Except when a factor in the denominator vanishes.)

This is fun; let's try again. Maybe we'll find a formula that will really astonish our friends. What does Pfaff's reflection law tell us if we apply it to the strange form (5.99), where $z = 2$? In this case we set $a = -m$, $b = 1$, and $c = -2m + \epsilon$, obtaining

$$\begin{aligned}(-1)^m \lim_{\epsilon\to 0} F\left(\begin{matrix}-m, 1\\ -2m+\epsilon\end{matrix}\middle| 2\right) &= \lim_{\epsilon\to 0} F\left(\begin{matrix}-m, -2m-1+\epsilon\\ -2m+\epsilon\end{matrix}\middle| 2\right)\\ &= \lim_{\epsilon\to 0}\sum_{k\geqslant 0}\frac{(-m)^{\overline{k}}(-2m-1+\epsilon)^{\overline{k}}}{(-2m+\epsilon)^{\overline{k}}}\frac{2^k}{k!}\\ &= \sum_{k\leqslant m}\binom{m}{k}\frac{(2m+1)^{\underline{k}}}{(2m)^{\underline{k}}}(-2)^k,\end{aligned}$$

because none of the limiting terms is close to zero. This leads to another miraculous formula,

(Hysterical note: See exercise 51 if you get a different result.)

$$\begin{aligned}\sum_{k\leqslant m}\binom{m}{k}\frac{2m+1}{2m+1-k}(-2)^k &= (-1)^m 2^{2m}\Big/\binom{2m}{m}\\ &= 1\Big/\binom{-1/2}{m}, \quad \text{integer } m \geqslant 0. \qquad (5.104)\end{aligned}$$

When $m = 3$, for example, the sum is

$$1 - 7 + \frac{84}{5} - 14 = -\frac{16}{5},$$

and $\binom{-1/2}{3}$ is indeed equal to $-\frac{5}{16}$.

When we looked at our binomial coefficient identities and converted them to hypergeometric form, we overlooked (5.19) because it was a relation between two sums instead of a closed form. But now we can regard (5.19) as an identity between hypergeometric series. If we differentiate it n times with respect to y and then replace k by $m - n - k$, we get

$$\begin{aligned}&\sum_{k\geqslant 0}\binom{m+r}{m-n-k}\binom{n+k}{n}x^{m-n-k}y^k\\ &\quad= \sum_{k\geqslant 0}\binom{-r}{m-n-k}\binom{n+k}{n}(-x)^{m-n-k}(x+y)^k.\end{aligned}$$

This yields the following hypergeometric transformation:

$$F\left(\begin{matrix}a, -n\\ c\end{matrix}\middle| z\right) = \frac{(a-c)^{\underline{n}}}{(-c)^{\underline{n}}}F\left(\begin{matrix}a, -n\\ 1-n+a-c\end{matrix}\middle| 1-z\right), \quad \begin{matrix}\text{integer}\\ n \geqslant 0.\end{matrix} \qquad (5.105)$$

Notice that when $z = 1$ this reduces to Vandermonde's convolution, (5.93).

Differentiation seems to be useful, if this example is any indication; we also found it helpful in Chapter 2, when summing $x + 2x^2 + \cdots + nx^n$. Let's see what happens when a general hypergeometric series is differentiated with respect to z:

$$\begin{aligned}
\frac{d}{dz} F\left(\begin{matrix} a_1, \ldots, a_m \\ b_1, \ldots, b_n \end{matrix}\middle| z\right) &= \sum_{k \geqslant 1} \frac{a_1^{\overline{k}} \ldots a_m^{\overline{k}}\, z^{k-1}}{b_1^{\overline{k}} \ldots b_n^{\overline{k}}\, (k-1)!} \\
&= \sum_{k+1 \geqslant 1} \frac{a_1^{\overline{k+1}} \ldots a_m^{\overline{k+1}}\, z^k}{b_1^{\overline{k+1}} \ldots b_n^{\overline{k+1}}\, k!} \\
&= \sum_{k \geqslant 0} \frac{a_1(a_1+1)^{\overline{k}} \ldots a_m(a_m+1)^{\overline{k}}\, z^k}{b_1(b_1+1)^{\overline{k}} \ldots b_n(b_n+1)^{\overline{k}}\, k!} \\
&= \frac{a_1 \ldots a_m}{b_1 \ldots b_n} F\left(\begin{matrix} a_1+1, \ldots, a_m+1 \\ b_1+1, \ldots, b_n+1 \end{matrix}\middle| z\right). \qquad (5.106)
\end{aligned}$$

The parameters move out and shift up.

It's also possible to use differentiation to tweak just one of the parameters while holding the rest of them fixed. For this we use the operator

How do you pronounce ϑ? (Dunno, but TEX calls it 'vartheta'.)

$$\vartheta = z \frac{d}{dz},$$

which acts on a function by differentiating it and then multiplying by z. This operator gives

$$\vartheta F\left(\begin{matrix} a_1, \ldots, a_m \\ b_1, \ldots, b_n \end{matrix}\middle| z\right) = z \sum_{k \geqslant 1} \frac{a_1^{\overline{k}} \ldots a_m^{\overline{k}}\, z^{k-1}}{b_1^{\overline{k}} \ldots b_n^{\overline{k}}\, (k-1)!} = \sum_{k \geqslant 0} \frac{k\, a_1^{\overline{k}} \ldots a_m^{\overline{k}}\, z^k}{b_1^{\overline{k}} \ldots b_n^{\overline{k}}\, k!},$$

which by itself isn't too useful. But if we multiply F by one of its upper parameters, say a_1, and add it to ϑF, we get

$$\begin{aligned}
(\vartheta + a_1) F\left(\begin{matrix} a_1, \ldots, a_m \\ b_1, \ldots, b_n \end{matrix}\middle| z\right) &= \sum_{k \geqslant 0} \frac{(k + a_1) a_1^{\overline{k}} \ldots a_m^{\overline{k}}\, z^k}{b_1^{\overline{k}} \ldots b_n^{\overline{k}}\, k!}, \\
&= \sum_{k \geqslant 0} \frac{a_1(a_1+1)^{\overline{k}}\, a_2^{\overline{k}} \ldots a_m^{\overline{k}}\, z^k}{b_1^{\overline{k}} \ldots b_n^{\overline{k}}\, k!} \\
&= a_1 F\left(\begin{matrix} a_1+1, a_2, \ldots, a_m \\ b_1, \ldots, b_n \end{matrix}\middle| z\right).
\end{aligned}$$

Only one parameter has been shifted.

A similar trick works with lower parameters, but in this case things shift down instead of up:

$$
\begin{aligned}
(\vartheta + b_1 - 1)\, F\left(\begin{matrix} a_1, \ldots, a_m \\ b_1, \ldots, b_n \end{matrix}\middle| z\right) &= \sum_{k \geqslant 0} \frac{(k + b_1 - 1) a_1^{\overline{k}} \ldots a_m^{\overline{k}}\, z^k}{b_1^{\overline{k}} \ldots b_n^{\overline{k}}\, k!}\,, \\
&= \sum_{k \geqslant 0} \frac{(b_1 - 1)\, a_1^{\overline{k}} \ldots a_m^{\overline{k}}\, z^k}{(b_1 - 1)^{\overline{k}}\, b_2^{\overline{k}} \ldots b_n^{\overline{k}}\, k!} \\
&= (b_1 - 1)\, F\left(\begin{matrix} a_1, \ldots, a_m \\ b_1 - 1, b_2, \ldots, b_n \end{matrix}\middle| z\right).
\end{aligned}
$$

We can now combine all these operations and make a mathematical "pun" by expressing the same quantity in two different ways. Namely, we have

Ever hear the one about the brothers who named their cattle ranch Focus, because it's where the sons raise meat?

$$
(\vartheta + a_1) \ldots (\vartheta + a_m) F = a_1 \ldots a_m\, F\left(\begin{matrix} a_1 + 1, \ldots, a_m + 1 \\ b_1, \ldots, b_n \end{matrix}\middle| z\right),
$$

and

$$
\begin{aligned}
&(\vartheta + b_1 - 1) \ldots (\vartheta + b_n - 1) F \\
&\qquad = (b_1 - 1) \ldots (b_n - 1)\, F\left(\begin{matrix} a_1, \ldots, a_m \\ b_1 - 1, \ldots, b_n - 1 \end{matrix}\middle| z\right),
\end{aligned}
$$

where $F = F(a_1, \ldots, a_m; b_1, \ldots, b_n; z)$. And (5.106) tells us that the top line is the derivative of the bottom line. Therefore the general hypergeometric function F satisfies the differential equation

$$
D(\vartheta + b_1 - 1) \ldots (\vartheta + b_n - 1) F = (\vartheta + a_1) \ldots (\vartheta + a_m) F\,, \tag{5.107}
$$

where D is the operator $\frac{d}{dz}$.

This cries out for an example. Let's find the differential equation satisfied by the standard 2-over-1 hypergeometric series $F(z) = F(a, b; c; z)$. According to (5.107), we have

$$
D(\vartheta + c - 1) F = (\vartheta + a)(\vartheta + b) F\,.
$$

What does this mean in ordinary notation? Well, $(\vartheta + c - 1)F$ is $zF'(z) + (c - 1)F(z)$, and the derivative of this gives the left-hand side,

$$
F'(z) + zF''(z) + (c - 1)F'(z)\,.
$$

On the right-hand side we have

$$\begin{aligned}(\vartheta+a)\big(zF'(z)+bF(z)\big) &= z\frac{d}{dz}\big(zF'(z)+bF(z)\big) + a\big(zF'(z)+bF(z)\big)\\ &= zF'(z)+z^2F''(z)+bzF'(z)+azF'(z)+abF(z)\,.\end{aligned}$$

Equating the two sides tells us that

$$z(1-z)F''(z) + \big(c - z(a+b+1)\big)F'(z) - abF(z) \;=\; 0\,. \tag{5.108}$$

This equation is equivalent to the factored form (5.107).

Conversely, we can go back from the differential equation to the power series. Let's assume that $F(z) = \sum_{k\geqslant 0} t_k z^k$ is a power series satisfying (5.107). A straightforward calculation shows that we must have

$$\frac{t_{k+1}}{t_k} \;=\; \frac{(k+a_1)\ldots(k+a_m)}{(k+b_1)\ldots(k+b_n)(k+1)}\,;$$

hence $F(z)$ must be $t_0\,F(a_1,\ldots,a_m;b_1,\ldots,b_n;z)$. We've proved that the hypergeometric series (5.76) is the only formal power series that satisfies the differential equation (5.107) and has the constant term 1.

It would be nice if hypergeometrics solved all the world's differential equations, but they don't quite. The right-hand side of (5.107) always expands into a sum of terms of the form $\alpha_k z^k F^{(k)}(z)$, where $F^{(k)}(z)$ is the kth derivative $D^kF(k)$; the left-hand side always expands into a sum of terms of the form $\beta_k z^{k-1}F^{(k)}(z)$ with $k > 0$. So the differential equation (5.107) always takes the special form

$$z^{n-1}(\beta_n - z\alpha_n)F^{(n)}(z) + \cdots + (\beta_1 - z\alpha_1)F'(z) - \alpha_0 F(z) \;=\; 0\,.$$

Equation (5.108) illustrates this in the case $n = 2$. Conversely, we will prove in exercise 6.13 that any differential equation of this form can be factored in terms of the ϑ operator, to give an equation like (5.107). So these are the differential equations whose solutions are power series with rational term ratios.

The function $F(z) = (1-z)^r$ satisfies $\vartheta F = z(\vartheta - r)F$. This gives another proof of the binomial theorem.

Multiplying both sides of (5.107) by z dispenses with the D operator and gives us an instructive all-ϑ form,

$$\vartheta(\vartheta+b_1-1)\ldots(\vartheta+b_n-1)F \;=\; z(\vartheta+a_1)\ldots(\vartheta+a_m)F\,. \tag{5.109}$$

The first factor $\vartheta = (\vartheta + 1 - 1)$ on the left corresponds to the $(k+1)$ in the term ratio (5.81), which corresponds to the $k!$ in the denominator of the kth term in a general hypergeometric series. The other factors $(\vartheta + b_j - 1)$ correspond to the denominator factor $(k + b_j)$, which corresponds to $b_j^{\overline{k}}$ in (5.76). On the right, the z corresponds to z^k, and $(\vartheta + a_j)$ corresponds to $a_j^{\overline{k}}$.

One use of this differential theory is to find and prove new transformations. For example, we can readily verify that both of the hypergeometrics

$$F\left(\begin{matrix}2a,\,2b\\ a+b+\frac{1}{2}\end{matrix}\,\middle|\,z\right) \quad\text{and}\quad F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\,\middle|\,4z(1-z)\right)$$

satisfy the differential equation

$$z(1-z)F''(z)+(a+b+\tfrac{1}{2})(1-2z)F'(z)-4abF(z) \;=\; 0;$$

hence *Gauss's identity* [143, equation 102]

$$F\left(\begin{matrix}2a,\,2b\\ a+b+\frac{1}{2}\end{matrix}\,\middle|\,z\right) \;=\; F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\,\middle|\,4z(1-z)\right) \tag{5.110}$$

must be true. In particular,

(Caution: We can't use (5.110) safely when $|z| > 1/2$, unless both sides are polynomials; see exercise 53.)

$$F\left(\begin{matrix}2a,\,2b\\ a+b+\frac{1}{2}\end{matrix}\,\middle|\,\frac{1}{2}\right) \;=\; F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\,\middle|\,1\right), \tag{5.111}$$

whenever both infinite sums converge.

Every new identity for hypergeometrics has consequences for binomial coefficients, and this one is no exception. Let's consider the sum

$$\sum_{k\leqslant m}\binom{m-k}{n}\binom{m+n+1}{k}\left(\frac{-1}{2}\right)^k, \qquad \text{integers } m\geqslant n\geqslant 0.$$

The terms are nonzero for $0\leqslant k\leqslant m-n$, and with a little delicate limit-taking as before we can express this sum as the hypergeometric

$$\lim_{\epsilon\to 0}\binom{m}{n}F\left(\begin{matrix}n-m,\,-n-m-1+\alpha\epsilon\\ -m+\epsilon\end{matrix}\,\middle|\,\frac{1}{2}\right).$$

The value of α doesn't affect the limit, since the nonpositive upper parameter $n-m$ cuts the sum off early. We can set $\alpha=2$, so that (5.111) applies. The limit can now be evaluated because the right-hand side is a special case of (5.92). The result can be expressed in simplified form,

$$\begin{aligned}&\sum_{k\leqslant m}\binom{m-k}{n}\binom{m+n+1}{k}\left(\frac{-1}{2}\right)^k\\ &\qquad=\binom{(m+n)/2}{n}2^{n-m}[m+n\text{ is even}], \qquad \begin{matrix}\text{integers}\\ m\geqslant n\geqslant 0,\end{matrix}\end{aligned} \tag{5.112}$$

as shown in exercise 54. For example, when $m=5$ and $n=2$ we get $\binom{5}{2}\binom{8}{0}-\binom{4}{2}\binom{8}{1}/2+\binom{3}{2}\binom{8}{2}/4-\binom{2}{2}\binom{8}{3}/8=10-24+21-7=0$; when $m=4$ and $n=2$, both sides give $\frac{3}{4}$.

We can also find cases where (5.110) gives binomial sums when $z = -1$, but these are really weird. If we set $a = \frac{1}{6} - \frac{n}{3}$ and $b = -n$, we get the monstrous formula

$$F\left(\begin{matrix}\frac{1}{3}-\frac{2}{3}n, -2n\\ \frac{2}{3}-\frac{4}{3}n\end{matrix}\middle|-1\right) = F\left(\begin{matrix}\frac{1}{6}-\frac{1}{3}n, -n\\ \frac{2}{3}-\frac{4}{3}n\end{matrix}\middle|-8\right).$$

These hypergeometrics are nondegenerate polynomials when $n \not\equiv 2 \pmod 3$; and the parameters have been cleverly chosen so that the left-hand side can be evaluated by (5.94). We are therefore led to a truly mind-boggling result,

$$\sum_k \binom{n}{k}\binom{\frac{1}{3}n-\frac{1}{6}}{k} 8^k \Big/ \binom{\frac{4}{3}n-\frac{2}{3}}{k} = \binom{2n}{n}\Big/\binom{\frac{4}{3}n-\frac{2}{3}}{n}, \quad \text{integer } n \geqslant 0,\ n \not\equiv 2 \pmod 3. \tag{5.113}$$

This is the most startling identity in binomial coefficients that we've seen. Small cases of the identity aren't even easy to check by hand. (It turns out that both sides do give $\frac{81}{7}$ when $n = 3$.) But the identity is completely useless, of course; surely it will never arise in a practical problem.

The only use of (5.113) is to demonstrate the existence of incredibly useless identities.

So that's our hype for hypergeometrics. We've seen that hypergeometric series provide a high-level way to understand what's going on in binomial coefficient sums. A great deal of additional information can be found in the classic book by Bailey [18] and its sequel by Gasper and Rahman [141].

5.7 PARTIAL HYPERGEOMETRIC SUMS

Most of the sums we've evaluated in this chapter range over all indices $k \geqslant 0$, but sometimes we've been able to find a closed form that works over a general range $a \leqslant k < b$. For example, we know from (5.16) that

$$\sum_{k<m} \binom{n}{k}(-1)^k = (-1)^{m-1}\binom{n-1}{m-1}, \qquad \text{integer } m. \tag{5.114}$$

The theory in Chapter 2 gives us a nice way to understand formulas like this: If $f(k) = \Delta g(k) = g(k+1) - g(k)$, then we've agreed to write $\sum f(k)\,\delta k = g(k) + C$, and

$$\sum\nolimits_a^b f(k)\,\delta k = g(k)\Big|_a^b = g(b) - g(a).$$

Furthermore, when a and b are integers with $a \leqslant b$, we have

$$\sum\nolimits_a^b f(k)\,\delta k = \sum_{a\leqslant k<b} f(k) = g(b) - g(a).$$

Therefore identity (5.114) corresponds to the indefinite summation formula

$$\sum \binom{n}{k}(-1)^k\,\delta k \;=\; (-1)^{k-1}\binom{n-1}{k-1} + C\,,$$

and to the difference formula

$$\Delta\Bigl((-1)^k\binom{n}{k}\Bigr) \;=\; (-1)^{k+1}\binom{n+1}{k+1}\,.$$

It's easy to start with a function $g(k)$ and to compute $\Delta g(k) = f(k)$, a function whose sum will be $g(k) + C$. But it's much harder to start with $f(k)$ and to figure out its indefinite sum $\sum f(k)\,\delta k = g(k) + C$; this function g might not have a simple form. For example, there is apparently no simple form for $\sum \binom{n}{k}\,\delta k$; otherwise we could evaluate sums like $\sum_{k\leqslant n/3}\binom{n}{k}$, about which we're clueless. Yet maybe there is a simple form for $\sum \binom{n}{k}\,\delta k$ and we just haven't thought of it; how can we be sure?

In 1977, R. W. Gosper [154] discovered a beautiful way to find indefinite sums $\sum f(k)\,\delta k = g(k) + C$ whenever f and g belong to a general class of functions called hypergeometric terms. Let us write

$$F\left(\begin{matrix} a_1, \ldots, a_m \\ b_1, \ldots, b_n \end{matrix}\middle| z\right)_k \;=\; \frac{a_1^{\overline{k}}\ldots a_m^{\overline{k}}}{b_1^{\overline{k}}\ldots b_n^{\overline{k}}}\,\frac{z^k}{k!} \tag{5.115}$$

for the kth term of the hypergeometric series $F(a_1,\ldots,a_m;\,b_1,\ldots,b_n;\,z)$. We will regard $F(a_1,\ldots,a_m;\,b_1,\ldots,b_n;\,z)_k$ as a function of k, not of z. In many cases it turns out that there are parameters c, A_1, ..., A_M, B_1, ..., B_N, and Z such that

$$\sum F\left(\begin{matrix} a_1, \ldots, a_m \\ b_1, \ldots, b_n \end{matrix}\middle| z\right)_k \delta k \;=\; c\,F\left(\begin{matrix} A_1, \ldots, A_M \\ B_1, \ldots, B_N \end{matrix}\middle| Z\right)_k + C\,, \tag{5.116}$$

given a_1, ..., a_m, b_1, ..., b_n, and z. We will say that a given function $F(a_1,\ldots,a_m;\,b_1,\ldots,b_n;\,z)_k$ is *summable in hypergeometric terms* if such constants c, A_1, ..., A_M, B_1, ..., B_N, Z exist. Gosper's algorithm either finds the unknown constants or proves that no such constants exist.

In general, we say that $t(k)$ is a *hypergeometric term* if $t(k+1)/t(k)$ is a rational function of k, not identically zero. This means, in essence, that $t(k)$ is a constant multiple of a term like (5.115). (A technicality arises, however, with respect to zeros, because we want $t(k)$ to be meaningful when k is negative and when one or more of the b's in (5.115) is zero or a negative integer. Strictly speaking, we get the most general hypergeometric term by multiplying (5.115) by a nonzero constant times a power of 0, then cancelling zeros of the numerator with zeros of the denominator. The examples in exercise 12 help clarify this general rule.)

Suppose we want to find $\sum t(k)\,\delta k$, when $t(k)$ is a hypergeometric term. Gosper's algorithm proceeds in two steps, each of which is fairly straightforward. Step 1 is to express the term ratio in the special form

$$\frac{t(k+1)}{t(k)} = \frac{p(k+1)}{p(k)}\,\frac{q(k)}{r(k+1)}\,, \tag{5.117}$$

(Divisibility of polynomials is analogous to divisibility of integers. For example, $(k+\alpha)\backslash q(k)$ means that the quotient $q(k)/(k+\alpha)$ is a polynomial. It's easy to see that $(k+\alpha)\backslash q(k)$ if and only if $q(-\alpha)=0$.)

where p, q, and r are polynomials subject to the following condition:

$$\begin{aligned}&(k+\alpha)\backslash q(k) \quad\text{and}\quad (k+\beta)\backslash r(k)\\ &\qquad\Longrightarrow\ \alpha-\beta \text{ is not a positive integer.}\end{aligned} \tag{5.118}$$

This condition is easy to achieve: We start by provisionally setting $p(k)=1$, and we set $q(k)$ and $r(k+1)$ to the numerator and denominator of the term ratio, factoring them into linear factors. For example, if $t(k)$ has the form (5.115), we start with the factorizations $q(k) = (k+a_1)\ldots(k+a_m)z$ and $r(k) = (k+b_1-1)\ldots(k+b_n-1)k$. Then we check if (5.118) is violated. If q and r have factors $(k+\alpha)$ and $(k+\beta)$ where $\alpha-\beta = N > 0$, we divide them out of q and r and replace $p(k)$ by

$$p(k)(k+\alpha-1)^{\underline{N-1}} = p(k)(k+\alpha-1)(k+\alpha-2)\ldots(k+\beta+1)\,. \tag{5.119}$$

The new p, q, and r still satisfy (5.117), and we can repeat this process until (5.118) holds. We'll see in a moment why (5.118) is important.

Step 2 of Gosper's algorithm is to finish the job—to find a hypergeometric term $T(k)$ such that

$$t(k) = T(k+1) - T(k)\,, \tag{5.120}$$

whenever possible. But it's not obvious how to do this; we need to develop some theory before we know how to proceed. Gosper noticed, after studying a lot of special cases, that it is wise to write the unknown function $T(k)$ in the form

$$T(k) = \frac{r(k)\,s(k)\,t(k)}{p(k)}\,, \tag{5.121}$$

(Exercise 55 gives a clue about why we might want to make this magic substitution.)

where $s(k)$ is a secret function that must be discovered somehow. Plugging (5.121) into (5.120) and applying (5.117) gives

$$\begin{aligned} t(k) &= \frac{r(k+1)s(k+1)t(k+1)}{p(k+1)} - \frac{r(k)s(k)t(k)}{p(k)}\\ &= \frac{q(k)s(k+1)t(k)}{p(k)} - \frac{r(k)s(k)t(k)}{p(k)}\,; \end{aligned}$$

so we need to have

$$p(k) \;=\; q(k)s(k+1) - r(k)s(k)\,. \tag{5.122}$$

If we can find $s(k)$ satisfying this fundamental recurrence relation, we've found $\sum t(k)\,\delta k$. If we can't, there's no T.

We're assuming that $T(k)$ is a hypergeometric term, which means that $T(k+1)/T(k)$ is a rational function of k. Therefore, by (5.121) and (5.120), $r(k)s(k)/p(k) = T(k)/\bigl(T(k+1) - T(k)\bigr)$ is a rational function of k, and $s(k)$ itself must be a quotient of polynomials:

$$s(k) \;=\; f(k)/g(k)\,.$$

But in fact we can prove that $s(k)$ is itself a polynomial. For if $g(k)$ is not constant, and if $f(k)$ and $g(k)$ have no common factors, let N be the largest integer such that $(k+\beta)$ and $(k+\beta+N-1)$ both occur as factors of $g(k)$ for some complex number β. The value of N is positive, since $N=1$ always satisfies this condition. Equation (5.122) can be rewritten

$$p(k)g(k{+}1)g(k) \;=\; q(k)f(k{+}1)g(k) - r(k)g(k{+}1)f(k)\,,$$

and if we set $k=-\beta$ and $k=-\beta-N$ we get

$$r(-\beta)g(1{-}\beta)f(-\beta) \;=\; 0 \;=\; q(-\beta{-}N)f(1{-}\beta{-}N)g(-\beta{-}N)\,.$$

Now $f(-\beta)\neq 0$ and $f(1-\beta-N)\neq 0$, because f and g have no common roots. Also $g(1-\beta)\neq 0$ and $g(-\beta-N)\neq 0$, because $g(k)$ would otherwise contain the factor $(k+\beta-1)$ or $(k+\beta+N)$, contrary to the maximality of N. Therefore

$$r(-\beta) \;=\; q(-\beta-N) \;=\; 0\,.$$

But this contradicts condition (5.118). Hence $s(k)$ must be a polynomial.

I see: Gosper came up with condition (5.118) in order to make this proof go through.

Our task now boils down to finding a polynomial $s(k)$ that satisfies (5.122), when $p(k)$, $q(k)$, and $r(k)$ are given polynomials, or proving that no such polynomial exists. It's easy to do this when $s(k)$ has any particular degree d, since we can write

$$s(k) \;=\; \alpha_d k^d + \alpha_{d-1}k^{d-1} + \cdots + \alpha_0\,, \qquad \alpha_d \;\neq\; 0 \tag{5.123}$$

for unknown coefficients $(\alpha_d,\ldots,\alpha_0)$ and plug this expression into the fundamental recurrence (5.122). The polynomial $s(k)$ will satisfy the recurrence if and only if the α's satisfy the linear equations that result when we equate coefficients of each power of k in (5.122).

But how can we determine the degree of s? It turns out that there actually are at most two possibilities. We can rewrite (5.122) in the form

$$2p(k) \;=\; Q(k)\big(s(k+1)+s(k)\big) + R(k)\big(s(k+1)-s(k)\big)\,, \qquad (5.124)$$
$$\text{where} \quad Q(k) \;=\; q(k)-r(k) \quad \text{and} \quad R(k) \;=\; q(k)+r(k)\,.$$

If $s(k)$ has degree d, then the sum $s(k+1)+s(k) = 2\alpha_d k^d + \cdots$ also has degree d, while the difference $s(k+1)-s(k) = \Delta s(k) = d\alpha_d k^{d-1} + \cdots$ has degree $d-1$. (The zero polynomial can be assumed to have degree -1.) Let's write $\deg(P)$ for the degree of a polynomial P. If $\deg(Q) \geqslant \deg(R)$, then the degree of the right-hand side of (5.124) is $\deg(Q)+d$, so we must have $d = \deg(p) - \deg(Q)$. On the other hand if $\deg(Q) < \deg(R) = d'$, we can write $Q(k) = \lambda' k^{d'-1} + \cdots$ and $R(k) = \lambda k^{d'} + \cdots$ where $\lambda \neq 0$; the right-hand side of (5.124) has the form

$$(2\lambda'\alpha_d + \lambda d\,\alpha_d)k^{d+d'-1} + \cdots\,.$$

Ergo, two possibilities: Either $2\lambda' + \lambda d \neq 0$, and $d = \deg(p) - \deg(R) + 1$; or $2\lambda' + \lambda d = 0$, and $d > \deg(p) - \deg(R) + 1$. The second case needs to be examined only if $-2\lambda'/\lambda$ is an integer d greater than $\deg(p) - \deg(R) + 1$.

OK, we now have enough facts to perform Step 2 of Gosper's two-step algorithm: By trying at most two values of d, we can discover $s(k)$, whenever equation (5.122) has a polynomial solution. If $s(k)$ exists, we can plug it into (5.121) and we have our T. If it doesn't, we've proved that $t(k)$ is not summable in hypergeometric terms.

Time for an example: Let's try the partial sum (5.114). Gosper's method should be able to deduce the value of

$$\sum \binom{n}{k}(-1)^k\,\delta k$$

for any fixed n, so we seek the sum of

$$t(k) \;=\; \binom{n}{k}(-1)^k \;=\; \frac{n!\,(-1)^k}{k!\,(n-k)!}\,.$$

Step 1 is to put the term ratio into the required form (5.117); we have

$$\frac{t(k+1)}{t(k)} \;=\; \frac{k-n}{k+1} \;=\; \frac{p(k+1)\,q(k)}{p(k)\,r(k+1)}$$

Why isn't it $r(k) = k+1$? Oh, I see.

so we simply take $p(k) = 1$, $q(k) = k-n$, and $r(k) = k$. This choice of p, q, and r satisfies (5.118), unless n is a negative integer; let's suppose it isn't.

Now we do Step 2. According to (5.124), we should consider the polynomials $Q(k) = -n$ and $R(k) = 2k-n$. Since R has larger degree than Q,

we need to look at two cases. Either $d = \deg(p) - \deg(R) + 1$, which is 0; or $d = -2\lambda'/\lambda$ where $\lambda' = -n$ and $\lambda = 2$, hence $d = n$. The first case is nicer, because it doesn't require n to be a positive integer, so let's try it first; we'll need to try the other possibility for d only if the first case fails. Assuming that $d = 0$, the value of $s(k)$ is simply α_0, and equation (5.122) reduces to

$$1 \;=\; (k-n)\alpha_0 - k\alpha_0\,.$$

Hence we choose $\alpha_0 = -1/n$. This satisfies the equation and gives

$$\begin{aligned} T(k) \;&=\; \frac{r(k)\,s(k)\,t(k)}{p(k)} \\ &=\; k\cdot\left(\frac{-1}{n}\right)\cdot\binom{n}{k}(-1)^k \\ &=\; \binom{n-1}{k-1}(-1)^{k-1}\,, \qquad \text{if } n \neq 0, \end{aligned}$$

precisely the answer we were hoping to confirm.

If we apply the same method to find the indefinite sum $\sum \binom{n}{k}\,\delta k$, without the $(-1)^k$, everything will be almost the same except that $q(k)$ will be $n-k$; hence $Q(k) = n - 2k$ will have greater degree than $R(k) = n$, and we will conclude that d has the impossible value $\deg(p) - \deg(Q) = -1$. (The polynomial $s(k)$ cannot have negative degree, because it cannot be zero.) Therefore the function $\binom{n}{k}$ is not summable in hypergeometric terms.

However, once we have eliminated the impossible, whatever remains — however improbable — must be the truth (according to S. Holmes [83]). When we defined p, q, and r in Step 1, we decided to ignore the possibility that n might be a negative integer. What if it is? Let's set $n = -N$, where N is positive. Then the term ratio for $\sum \binom{n}{k}\,\delta k$ is

$$\frac{t(k+1)}{t(k)} \;=\; \frac{-(k+N)}{(k+1)} \;=\; \frac{p(k+1)}{p(k)}\,\frac{q(k)}{r(k+1)}$$

and it should be represented by $p(k) = (k+1)^{\overline{N-1}}$, $q(k) = -1$, $r(k) = 1$, according to (5.119). Step 2 of Gosper's algorithm now tells us to look for a polynomial $s(k)$ of degree $d = N - 1$; maybe there's hope after all. For example, when $N = 2$ recurrence (5.122) says that we should solve

$$k+1 \;=\; -\big((k+1)\alpha_1 + \alpha_0\big) - (k\alpha_1 + \alpha_0)\,.$$

Equating coefficients of k and 1 tells us that

$$1 \;=\; -\alpha_1 - \alpha_1; \qquad 1 \;=\; -\alpha_1 - \alpha_0 - \alpha_0;$$

hence $s(k) = -\frac{1}{2}k - \frac{1}{4}$ is a solution, and

$$T(k) \;=\; \frac{1 \cdot \left(-\frac{1}{2}k - \frac{1}{4}\right) \cdot \binom{-2}{k}}{k+1} \;=\; (-1)^{k-1}\frac{2k+1}{4}\,.$$

"Excellent, Holmes!" "Elementary, my dear Watson."

Can this be the desired sum? Yes, it checks out:

$$(-1)^k\frac{2k+3}{4} - (-1)^{k-1}\frac{2k+1}{4} \;=\; (-1)^k(k+1) \;=\; \binom{-2}{k}\,.$$

Incidentally, we can write this summation formula in another form, by attaching an upper limit:

$$\begin{aligned}\sum_{k<m}\binom{-2}{k} \;&=\; (-1)^{k-1}\frac{2k+1}{4}\Big|_0^m \\ &=\; \frac{(-1)^{m-1}}{2}\left(m + \frac{1-(-1)^m}{2}\right) \\ &=\; (-1)^{m-1}\left\lceil\frac{m}{2}\right\rceil, \qquad \text{integer } m \geqslant 0.\end{aligned}$$

This representation conceals the fact that $\binom{-2}{k}$ is summable in hypergeometric terms, because $\lceil m/2 \rceil$ is not a hypergeometric term. (See exercise 12.)

A problem might arise in the denominator of (5.121) if $p(k) = 0$ for some integer k. Exercise 97 gives some insight into what can be done in such situations.

Notice that we need not bother to compile a catalog of indefinitely summable hypergeometric terms, analogous to the database of definite hypergeometric sums mentioned earlier in this chapter, because Gosper's algorithm provides a quick, uniform method that works in all summable cases.

Marko Petkovšek [291] has found a nice way to generalize Gosper's algorithm to more complicated inversion problems, by showing how to determine all hypergeometric terms $T(k)$ that satisfy the lth-order recurrence

$$t(k) \;=\; p_l(k)T(k+l) + \cdots + p_1(k)T(k+1) + p_0(k)T(k)\,, \tag{5.125}$$

given any hypergeometric term $t(k)$ and polynomials $p_l(k), \ldots, p_1(k), p_0(k)$.

5.8 MECHANICAL SUMMATION

Gosper's algorithm, beautiful as it is, finds a closed form for only a few of the binomial sums we meet in practice. But we need not stop there. Doron Zeilberger [383] showed how to extend Gosper's algorithm so that it becomes even more beautiful, making it succeed in vastly more cases. With

Zeilberger's extension we can handle summation over all k, not just partial sums, so we have an alternative to the hypergeometric methods of Sections 5.5 and 5.6. Moreover, as with Gosper's original method, the calculations can be done by computer, almost blindly; we need not rely on cleverness and luck.

The basic idea is to regard the term we want to sum as a function $t(n,k)$ of two variables n and k. (In Gosper's algorithm we wrote just $t(k)$.) When $t(n,k)$ does not turn out to be indefinitely summable in hypergeometric terms, with respect to k — and let's face it, relatively few functions are — Zeilberger noticed that we can often modify $t(n,k)$ in order to obtain another term that *is* indefinitely summable. For example, it often turns out in practice that $\beta_0(n)t(n,k)+\beta_1(n)t(n+1,k)$ is indefinitely summable with respect to k, for appropriate polynomials $\beta_0(n)$ and $\beta_1(n)$. And when we carry out the sum with respect to k, we obtain a recurrence in n that solves our problem.

Let's start with a simple case in order to get familiar with this general approach. Suppose we have forgotten the binomial theorem, and we want to evaluate $\sum_k \binom{n}{k} z^k$. How could we discover the answer, without clairvoyance or inspired guesswork? Earlier in this chapter, for example in Problem 3 of Section 5.2, we learned how to replace $\binom{n}{k}$ by $\binom{n-1}{k}+\binom{n-1}{k-1}$ and to fiddle around with the result. But there's a more systematic way to proceed.

Or without looking on page 174.

Let $t(n,k)=\binom{n}{k}z^k$ be the quantity we want to sum. Gosper's algorithm tells us that we can't evaluate the partial sums $\sum_{k\leqslant m} t(n,k)$ for arbitrary n in hypergeometric terms, except in the case $z=-1$. So let's consider a more general term

$$\hat{t}(n,k) \;=\; \beta_0(n)t(n,k)+\beta_1(n)t(n+1,k) \tag{5.126}$$

instead. We'll look for values of $\beta_0(n)$ and $\beta_1(n)$ that make Gosper's algorithm succeed. First we want to simplify (5.126) by using the relation between $t(n+1,k)$ and $t(n,k)$ to eliminate $t(n+1,k)$ from the expression. Since

$$\begin{aligned}\frac{t(n+1,k)}{t(n,k)} &= \frac{(n+1)!\,z^k}{(n+1-k)!\,k!}\,\frac{(n-k)!\,k!}{n!\,z^k}\\ &= \frac{n+1}{n+1-k}\,,\end{aligned}$$

we have

$$\hat{t}(n,k) \;=\; p(n,k)\frac{t(n,k)}{n+1-k}\,,$$

where

$$p(n,k) \;=\; (n+1-k)\beta_0(n)+(n+1)\beta_1(n)\,.$$

We now apply Gosper's algorithm to $\hat{t}(n,k)$, with n held fixed, first writing

$$\frac{\hat{t}(n,k+1)}{\hat{t}(n,k)} = \frac{\hat{p}(n,k+1)}{\hat{p}(n,k)}\frac{q(n,k)}{r(n,k+1)} \tag{5.127}$$

as in (5.117). Gosper's method would find such a representation by starting with $\hat{p}(n,k) = 1$, but with Zeilberger's extension we are better off starting with $\hat{p}(n,k) = p(n,k)$. Notice that if we set $\bar{t}(n,k) = \hat{t}(n,k)/p(n,k)$ and $\bar{p}(n,k) = \hat{p}(n,k)/p(n,k)$, equation (5.127) is equivalent to

$$\frac{\bar{t}(n,k+1)}{\bar{t}(n,k)} = \frac{\bar{p}(n,k+1)}{\bar{p}(n,k)}\frac{q(n,k)}{r(n,k+1)}\,. \tag{5.128}$$

So we can find $\hat{p}$, q and r satisfying (5.127) by finding $\bar{p}$, q and r satisfying (5.128), starting with $\bar{p}(n,k) = 1$. This makes life easy, because $\bar{t}(n,k)$ does not involve the unknown quantities $\beta_0(n)$ and $\beta_1(n)$ that appear in $\hat{t}(n,k)$. In our case $\bar{t}(n,k) = t(n,k)/(n+1-k) = n!\,z^k/(n+1-k)!\,k!$, so we have

$$\frac{\bar{t}(n,k+1)}{\bar{t}(n,k)} = \frac{(n+1-k)z}{k+1}\,;$$

This time I remembered why $r(n,k)$ *isn't* $k+1$.

we may take $q(n,k) = (n+1-k)z$ and $r(n,k) = k$. These polynomials in k are supposed to satisfy condition (5.118). If they don't, we're supposed to remove factors from q and r and include corresponding factors (5.119) in $\bar{p}(n,k)$; but we should do this only when the quantity $\alpha - \beta$ in (5.118) is a positive integer constant, independent of n, because we want our calculations to be valid for arbitrary n. (The formulas we derive will, in fact, be valid even when n and k are not integers, using the generalized factorials (5.83).)

Our first choices of q and r do satisfy (5.118), in this sense, so we can move right on to Step 2 of Gosper's algorithm: We want to solve the analog of (5.122), using (5.127) in place of (5.117). So we want to solve

$$\hat{p}(n,k) = q(n,k)s(n,k+1) - r(n,k)s(n,k) \tag{5.129}$$

for the secret polynomial

$$s(n,k) = \alpha_d(n)k^d + \alpha_{d-1}(n)k^{d-1} + \cdots + \alpha_0(n)\,. \tag{5.130}$$

(The coefficients of s are considered to be functions of n, not just constants.) In our case equation (5.129) is

$$\begin{aligned}&(n+1-k)\beta_0(n) + (n+1)\beta_1(n)\\ &\qquad = (n+1-k)zs(n,k+1) - ks(n,k)\,,\end{aligned}$$

and we regard this as a polynomial equation in k with coefficients that are functions of n. As before, we determine the degree d of s by considering

$Q(n,k) = q(n,k) - r(n,k)$ and $R(n,k) = q(n,k) + r(n,k)$. Since $\deg(Q) = \deg(R) = 1$ (assuming that $z \neq -1$), we have $d = \deg(\hat{p}) - \deg(Q) = 0$ and $s(n,k) = \alpha_0(n)$ is independent of k. Our equation becomes

The degree function $\deg(Q)$ *refers here to the degree in* k*, treating* n *as constant.*

$$(n+1-k)\beta_0(n) + (n+1)\beta_1(n) \;=\; (n+1-k)z\alpha_0(n) - k\alpha_0(n)\,;$$

and by equating powers of k we get the equivalent k-free equations

$$\begin{aligned}(n+1)\beta_0(n) + (n+1)\beta_1(n) - (n+1)z\alpha_0(n) &= 0\,,\\ -\beta_0(n) \qquad\qquad\qquad\qquad + (z+1)\alpha_0(n) &= 0\,.\end{aligned}$$

Hence we have a solution to (5.129) with

$$\beta_0(n) \;=\; z+1\,, \qquad \beta_1(n) \;=\; -1\,, \qquad \alpha_0(n) \;=\; s(n,k) \;=\; 1\,.$$

(By chance, n has dropped out.)

We have discovered, by a purely mechanical method, that the term $\hat{t}(n,k) = (z+1)t(n,k) - t(n+1,k)$ is summable in hypergeometric terms. In other words,

$$\hat{t}(n,k) \;=\; T(n,k+1) - T(n,k)\,, \tag{5.131}$$

where $T(n,k)$ is a hypergeometric term in k. What is this $T(n,k)$? According to (5.121) and (5.128), we have

$$T(n,k) \;=\; \frac{r(n,k)s(n,k)\hat{t}(n,k)}{\hat{p}(n,k)} \;=\; r(n,k)s(n,k)\bar{t}(n,k)\,, \tag{5.132}$$

because $\bar{p}(n,k) = 1$. (Indeed, $\bar{p}(n,k)$ almost always turns out to be 1 in practice.) Hence

$$T(n,k) \;=\; \frac{k}{n+1-k}\,t(n,k) \;=\; \frac{k}{n+1-k}\binom{n}{k}z^k \;=\; \binom{n}{k-1}z^k.$$

And sure enough, everything checks out — equation (5.131) is true:

$$(z+1)\binom{n}{k}z^k - \binom{n+1}{k}z^k \;=\; \binom{n}{k}z^{k+1} - \binom{n}{k-1}z^k\,.$$

But we don't actually need to know $T(n,k)$ precisely, because we are going to sum $t(n,k)$ over all integers k. All we need to know is that $T(n,k)$ is nonzero for only finitely many values of k, when n is any given nonnegative integer. Then the sum of $T(n,k+1) - T(n,k)$ over all k must telescope to 0.

Let $S_n = \sum_k t(n,k) = \sum_k \binom{n}{k}z^k$; this is the sum we started with, and we're now ready to compute it, because we now know a lot about $t(n,k)$. The

Gosper-Zeilberger procedure has deduced that

$$\sum_k \big((z+1)t(n,k) - t(n+1,k)\big) \;=\; 0\,.$$

In fact, $\lim_{k\to\infty} T(n,k) = 0$ *when* $|z| < 1$ *and* n *is any complex number. So (5.133) is true for all* n*, and in particular* $S_n = (z+1)^n$ *when* n *is a negative integer.*

But this sum is $(z+1)\sum_k t(n,k) - \sum_k t(n+1,k) = (z+1)S_n - S_{n+1}$. Therefore we have

$$S_{n+1} \;=\; (z+1)S_n\,. \tag{5.133}$$

Aha! This is a recurrence we know how to solve, provided that we know S_0. And obviously $S_0 = 1$. Hence we deduce that $S_n = (z+1)^n$, for all integers $n \geqslant 0$. QED.

Let's look back at this computation and summarize what we did, in a form that will apply also to other summands $t(n,k)$. The Gosper-Zeilberger algorithm can be formalated as follows, when $t(n,k)$ is given:

0 Set $l := 0$. (We'll seek recurrences in n of order l.)

1 Let $\hat{t}(n,k) = \beta_0(n)t(n,k) + \cdots + \beta_l(n)t(n+l,k)$, where $\beta_0(n), \ldots, \beta_l(n)$ are unknown functions. Use properties of $t(n,k)$ to find a linear combination $p(n,k)$ of $\beta_0(n), \ldots, \beta_l(n)$ with coefficients that are polynomials in n and k, so that $\hat{t}(n,k)$ can be written in the form $p(n,k)\bar{t}(n,k)$, where $\bar{t}(n,k)$ is a hypergeometric term in k. Find polynomials $\bar{p}(n,k)$, $q(n,k)$, $r(n,k)$ so that the term ratio of $\bar{t}(n,k)$ is expressed in the form (5.128), where $q(n,k)$ and $r(n,k)$ satisfy Gosper's condition (5.118). Set $\hat{p}(n,k) = p(n,k)\bar{p}(n,k)$.

2a Set $d_Q := \deg(q-r)$, $d_R := \deg(q+r)$, and

$$d \;:=\; \begin{cases} \deg(\hat{p}) - d_Q, & \text{if } d_Q \geqslant d_R; \\ \deg(\hat{p}) - d_R + 1, & \text{if } d_Q < d_R. \end{cases}$$

2b If $d \geqslant 0$, define $s(n,k)$ by (5.130), and consider the linear equations in $\alpha_0, \ldots, \alpha_d, \beta_0, \ldots \beta_l$ obtained by equating coefficients of powers of k in the fundamental equation (5.129). If these equations have a solution with $\beta_0, \ldots, \beta_l$ not all zero, go to Step 4. Otherwise, if $d_Q < d_R$ and if $-2\lambda'/\lambda$ is an integer greater than d, where λ is the coefficient of k^{d_R} in $q+r$ and λ' is the coefficient of k^{d_R-1} in $q-r$, set $d := -2\lambda'/\lambda$ and repeat Step 2b.

3 (The term $\hat{t}(n,k)$ isn't hypergeometrically summable.) Increase l by 1 and go back to Step 1.

4 (Success.) Set $T(n,k) := r(n,k)s(n,k)\bar{t}(n,k)/\bar{p}(n,k)$. The algorithm has discovered that $\hat{t}(n,k) = T(n,k+1) - T(n,k)$.

We'll prove later that this algorithm terminates successfully whenever $t(n,k)$ belongs to a large class of terms called proper terms.

The binomial theorem can be derived in many ways, so our first example of the Gosper-Zeilberger approach was more instructive than impressive. Let's tackle Vandermonde's convolution next. Can Gosper and Zeilberger deduce algorithmically that $\sum_k \binom{a}{k}\binom{b}{n-k}$ has a simple form? The algorithm starts with $l = 0$, which essentially reproduces Gosper's original algorithm, trying to see if $\binom{a}{k}\binom{b}{n-k}$ is summable in hypergeometric terms. Surprise: That term actually does turn out to be summable, if $a+b$ is a specific nonnegative integer (see exercise 94). But we are interested in general values of a and b, and the algorithm quickly discovers that the indefinite sum is not a hypergeometric term in general. So l is increased from 0 to 1, and the algorithm proceeds to try $\hat{t}(n,k) = \beta_0(n)t(n,k) + \beta_1(n)t(n+1,k)$ instead. The next step, as in our derivation of the binomial theorem, is to write $\hat{t}(n,k) = p(n,k)\bar{t}(n,k)$, where $p(n,k)$ is obtained by clearing fractions in $t(n+1,k)/t(n,k)$. In this case — the reader should please work along on a piece of scratch paper to check all these calculations — they aren't as hard as they look — everything goes through in an analogous fashion, but now with

$$\begin{aligned}
p(n,k) &= (n+1-k)\beta_0(n) + (b-n+k)\beta_1(n) \;=\; \hat{p}(n,k)\,,\\
\bar{t}(n,k) &= t(n,k)/(n{+}1{-}k) \;=\; a!\,b!/(a{-}k)!\,k!\,(b{-}n{+}k)!\,(n{+}1{-}k)!\,,\\
q(n,k) &= (n+1-k)(a-k)\,,\\
r(n,k) &= (b-n+k)k\,.
\end{aligned}$$

Step 2a finds $\deg(q-r) < \deg(q+r)$, and $d = \deg(\hat{p}) - \deg(q+r) + 1 = 0$, so $s(n,k)$ is again independent of k. Gosper's fundamental equation (5.129) is equivalent to two equations in three unknowns,

The crucial point is that the Gosper-Zeilberger method always leads to equations that are linear in the unknown α's and β's, because the left side of (5.129) is linear in the β's and the right side is linear in the α's.

$$\begin{aligned}
(n+1)\beta_0(n) + (b-n)\beta_1(n) \quad - (n+1)a\,\alpha_0(n) &= 0\,,\\
-\beta_0(n) \qquad\qquad + \beta_1(n) + (a+b+1)\alpha_0(n) &= 0\,,
\end{aligned}$$

which have the solution

$$\beta_0(n) \;=\; a+b-n\,, \qquad \beta_1(n) \;=\; -n-1\,, \qquad \alpha_0(n) \;=\; 1\,.$$

We conclude that $(a+b-n)t(n,k) - (n+1)t(n+1,k)$ is summable with respect to k; hence if $S_n = \sum_k \binom{a}{k}\binom{b}{n-k}$ the recurrence

$$S_{n+1} \;=\; \frac{a+b-n}{n+1}\,S_n$$

holds; thus $S_n = \binom{a+b}{n}$ since $S_0 = 1$. A piece of cake.

What about the Saalschützian triple-binomial identity in (5.28)? The proof of (5.28) in exercise 43 is interesting, but it requires inspiration. When we transform an art into a science, we aim to replace inspiration by perspiration; so let's see if the Gosper-Zeilberger approach to summation is able to

discover and prove (5.28) in a purely mechanical way. For convenience we make the substitutions $m = b + d$, $n = a$, $r = a + b + c + d$, $s = a + b + c$, so that (5.28) takes the more symmetrical form

$$\sum_k \frac{(a+b+c+d+k)!}{(a-k)!\,(b-k)!\,(c+k)!\,(d+k)!\,k!} = \frac{(a+b+c+d)!\,(a+b+c)!\,(a+b+d)!}{a!\,b!\,(a+c)!\,(a+d)!\,(b+c)!\,(b+d)!}\,. \tag{5.134}$$

To make the sum finite, we assume that either a or b is a nonnegative integer.

Deciding what parameter to call n is the only nonmechanical part.

Let $t(n,k) = (n+b+c+d+k)!/(n-k)!\,(b-k)!\,(c+k)!\,(d+k)!\,k!$ and $\hat{t}(n,k) = \beta_0(n)t(n,k) + \beta_1(n)t(n+1,k)$. Proceeding along a path that is beginning to become well worn, we set

$$\begin{aligned}
p(n,k) &= (n+1-k)\beta_0(n) + (n+1+b+c+d+k)\beta_1(n) \;=\; \hat{p}(n,k)\,,\\
\bar{t}(n,k) &= \frac{t(n,k)}{n+1-k} \;=\; \frac{(n+b+c+d+k)!}{(n+1-k)!\,(b-k)!\,(c+k)!\,(d+k)!\,k!}\,,\\
q(n,k) &= (n+b+c+d+k+1)(n+1-k)(b-k)\,,\\
r(n,k) &= (c+k)(d+k)k\,,
\end{aligned}$$

and we try to solve (5.129) for $s(n,k)$. Again $\deg(q-r) < \deg(q+r)$, but this time $\deg(\hat{p}) - \deg(q+r) + 1 = -1$ so it looks like we're stuck. However, Step 2b has an important second choice, $d = -2\lambda'/\lambda$, for the degree of s; we had better try it now before we give up. Here $R(n,k) = q(n,k) + r(n,k) = 2k^3 + \cdots$, so $\lambda = 2$, while the polynomial $Q(n,k) = q(n,k) - r(n,k)$ almost miraculously turns out to have degree 1 in k—the coefficient of k^2 vanishes! Therefore $\lambda' = 0$; Gosper allows us to take $d = 0$ and $s(n,k) = \alpha_0(n)$.

Notice that λ' is not the leading coefficient of Q, although λ is the leading coefficient of R. The number λ' is the coefficient of $k^{\deg(R)-1}$ in Q.

The equations to be solved are now

$$\begin{aligned}
&(n+1)\beta_0(n) + (n+1+b+c+d)\beta_1(n)\\
&\qquad - (n+1)(n+1+b+c+d)b\alpha_0(n) \;=\; 0\,,\\
&-\beta_0(n) + \beta_1(n)\\
&\qquad - \big((n+1)b - (n+1+b)(n+1+b+c+d) - cd\big)\alpha_0(n) \;=\; 0\,;
\end{aligned}$$

and we find

$$\begin{aligned}
\beta_0(n) &= (n+1+b+c)(n+1+b+d)(n+1+b+c+d)\,,\\
\beta_1(n) &= -(n+1)(n+1+c)(n+1+d)\,,\\
\alpha_0(n) &= 2n+2+b+c+d\,,
\end{aligned}$$

Perspiration flows, identity follows.

after only a modest amount of perspiration. The identity (5.134) follows immediately.

A similar proof of (5.134) can be obtained if we work with $n = d$ instead of $n = a$. (See exercise 99.)

The Gosper-Zeilberger approach helps us evaluate definite sums over a restricted range as well as sums over all k. For example, let's consider

$$S_n(z) \;=\; \sum_{k=0}^{n} \binom{n+k}{k} z^k . \tag{5.135}$$

When $z = \frac{1}{2}$ we obtained an "unexpected" result in (5.20); would Gosper and Zeilberger have expected it? Putting $t(n,k) = \binom{n+k}{k} z^k$ leads us to

$$\begin{aligned}
p(n,k) &= (n+1)\beta_0(n) + (n+1+k)\beta_1(n) \;=\; \hat{p}(n,k)\,,\\
\bar{t}(n,k) &= t(n,k)/(n+1) \;=\; (n+k)!\,z^k/k!\,(n+1)!\,,\\
q(n,k) &= (n+1+k)z\,,\\
r(n,k) &= k\,,
\end{aligned}$$

and $\deg(s) = \deg(\hat{p}) - \deg(q-r) = 0$. Equation (5.129) is solved by $\beta_0(n) = 1$, $\beta_1(n) = z-1$, $s(n,k) = 1$. Therefore we find

$$t(n,k) + (z-1)t(n+1,k) \;=\; T(n,k+1) - T(n,k)\,, \tag{5.136}$$

where $T(n,k) = r(n,k)s(n,k)\hat{t}(n,k)/\hat{p}(n,k) = \binom{n+k}{k-1} z^k$. We can now sum (5.136) for $0 \leqslant k \leqslant n+1$, getting

$$\begin{aligned}
S_n(z) + t(n,n+1) + (z-1)S_{n+1}(z) &= T(n,n+2) - T(n,0)\\
&= \binom{2n+2}{n+1} z^{n+2}\\
&= 2\binom{2n+1}{n} z^{n+2}\,.
\end{aligned}$$

But $t(n,n+1) = \binom{2n+1}{n+1} z^{n+1} = \binom{2n+1}{n} z^{n+1}$, so

$$S_{n+1}(z) \;=\; \frac{1}{1-z}\left(S_n(z) + (1-2z)\binom{2n+1}{n} z^{n+1}\right). \tag{5.137}$$

We see immediately that the case $z = \frac{1}{2}$ is special, and that $S_{n+1}(\frac{1}{2}) = 2S_n(\frac{1}{2})$. Moreover, the recurrence (5.137) can be simplified by applying the summation factor $(1-z)^{n+1}$ to both sides; this yields the general identity

$$(1-z)^n \sum_{k=0}^{n} \binom{n+k}{k} z^k \;=\; 1 + \frac{1-2z}{2-2z} \sum_{k=1}^{n} \binom{2k}{k} \big(z(1-z)\big)^k\,, \tag{5.138}$$

which comparatively few people would have expected before Gosper and Zeilberger came along. Now the production of such identities is routine.

How about the similar sum

$$S_n(z) \;=\; \sum_{k=0}^{n} \binom{n-k}{k} z^k\,, \tag{5.139}$$

which we encountered in (5.74)? Flushed with confidence, we set $t(n,k) = \binom{n-k}{k} z^k$ and proceed to calculate

$$\begin{aligned}
p(n,k) &= (n+1-2k)\beta_0(n) + (n+1-k)\beta_1(n) \;=\; \hat{p}(n,k)\,,\\
\bar{t}(n,k) &= t(n,k)/(n+1-2k) \;=\; (n-k)!\,z^k/k!\,(n+1-2k)!\,,\\
q(n,k) &= (n+1-2k)(n-2k)z\,,\\
r(n,k) &= (n+1-k)k\,.
\end{aligned}$$

$S_n(-\frac{1}{4})$ *equals* $(n+1)/2^n$.

But whoa — there's no way to solve (5.129), if we assume that $z \neq -\frac{1}{4}$, because the degree of s would have to be $\deg(\hat{p}) - \deg(q-r) = -1$.

No problem. We simply add another parameter $\beta_2(n)$ and try $\hat{t}(n,k) = \beta_0(n)t(n,k) + \beta_1(n)t(n+1,k) + \beta_2(n)t(n+2,k)$ instead:

$$\begin{aligned}
p(n,k) &= (n+1-2k)(n+2-2k)\beta_0(n)\\
&\quad + (n+1-k)(n+2-2k)\beta_1(n)\\
&\quad + (n+1-k)(n+2-k)\beta_2(n) \;=\; \hat{p}(n,k)\,,\\
\bar{t}(n,k) &= t(n,k)/(n{+}1{-}2k)(n{+}2{-}2k) \;=\; (n{-}k)!\,z^k/k!\,(n{+}2{-}2k)!\,,\\
q(n,k) &= (n+2-2k)(n+1-2k)z\,,\\
r(n,k) &= (n+1-k)k\,.
\end{aligned}$$

Now we can try $s(n,k) = \alpha_0(n)$ and (5.129) does have a solution:

$$\beta_0(n) \;=\; z\,, \qquad \beta_1(n) \;=\; 1\,, \qquad \beta_2(n) \;=\; -1\,, \qquad \alpha_0(n) \;=\; 1\,.$$

We have discovered that

$$zt(n,k) + t(n+1,k) - t(n+2,k) \;=\; T(n,k+1) - T(n,k)\,,$$

where $T(n,k)$ equals $r(n,k)s(n,k)\hat{t}(n,k)/\hat{p}(n,k) = (n+1-k)k\bar{t}(n,k) = \binom{n+1-k}{k-1} z^k$. Summing from $k=0$ to $k=n$ gives

$$\begin{aligned}
&zS_n(z) + \Bigl(S_{n+1}(z) - \tbinom{0}{n+1} z^{n+1}\Bigr) - \Bigl(S_{n+2}(z) - \tbinom{0}{n+2} z^{n+2} - \tbinom{1}{n+1} z^{n+1}\Bigr)\\
&\qquad = T(n,n+1) - T(n,0)\,.
\end{aligned}$$

And $\binom{1}{n+1} = \binom{0}{n} z^{n+1} = T(n,n+1)$ for all $n \geqslant 0$, so we obtain

$$S_{n+2}(z) \;=\; S_{n+1}(z) + zS_n(z)\,, \qquad n \geqslant 0. \tag{5.140}$$

We will study the solution of such recurrences in Chapters 6 and 7; the methods of those chapters lead directly from (5.140) to the closed form (5.74), when $S_0(z) = S_1(z) = 1$.

One more example — a famous one — will complete the picture. The French mathematician Roger Apéry solved a long-standing problem in 1978 when he proved that the number $\zeta(3) = 1 + 2^{-3} + 3^{-3} + 4^{-3} + \cdots$ is irrational [14]. One of the main components of his proof involved the binomial sums

$$A_n = \sum_k \binom{n}{k}^2 \binom{n+k}{k}^2 , \tag{5.141}$$

for which he announced a recurrence that other mathematicians were unable to verify at the time. (The numbers A_n have since become known as Apéry numbers; we have $A_0 = 1$, $A_1 = 5$, $A_2 = 73$, $A_3 = 1445$, $A_4 = 33001$.) Finally [356] Don Zagier and Henri Cohen found a proof of Apéry's claim, and their proof for this special (but difficult) sum was one of the key clues that ultimately led Zeilberger to discover the general approach we are discussing.

By now, in fact, we have seen enough examples to make the sum in (5.141) almost trivial. Putting $t(n,k) = \binom{n}{k}^2\binom{n+k}{k}^2$ and $\hat{t}(n,k) = \beta_0(n)t(n,k) + \beta_1(n)t(n+1,k) + \beta_2(n)t(n+2,k)$, we try to solve (5.129) with

(First we try doing without β_2, but that attempt quickly peters out.)

$$\begin{aligned}
p(n,k) &= (n+1-k)^2(n+2-k)^2\beta_0(n) \\
&\quad + (n+1+k)^2(n+2-k)^2\beta_1(n) \\
&\quad + (n+1+k)^2(n+2+k)^2\beta_2(n) \;=\; \hat{p}(n,k)\,, \\
\bar{t}(n,k) &= t(n,k)/(n{+}1{-}k)^2(n{+}2{-}k)^2 \;=\; (n{+}k)!^2/k!^4(n{+}2{-}k)!^2\,, \\
q(n,k) &= (n+1+k)^2(n+2-k)^2\,, \\
r(n,k) &= k^4\,.
\end{aligned}$$

(We don't worry about the fact that q has the factor $(k+n+1)$ while r has the factor k; this does not violate (5.118), because we are regarding n as a variable parameter, not a fixed integer.) Since $q(n,k) - r(n,k) = -2k^3 + \cdots$, we are allowed to set $\deg(s) = -2\lambda'/\lambda = 2$, so we take

$$s(n,k) = \alpha_2(n)k^2 + \alpha_1(n)k + \alpha_0(n)\,.$$

With this choice of s, the recurrence (5.129) boils down to five equations in the six unknown quantities $\beta_0(n)$, $\beta_1(n)$, $\beta_2(n)$, $\alpha_0(n)$, $\alpha_1(n)$, $\alpha_2(n)$. For example, the equation arising from the coefficients of k^0 simplifies to

$$\beta_0 + \beta_1 + \beta_2 - \alpha_0 - \alpha_1 - \alpha_2 = 0\,;$$

the equation arising from the coefficients of k^4 is

$$\beta_0 + \beta_1 + \beta_2 + \alpha_1 + (6 + 6n + 2n^2)\alpha_2 = 0\,.$$

The other three equations are more complicated. But the main point is that these linear equations — like all the equations that arise when we come to this stage of the Gosper-Zeilberger algorithm — are *homogeneous* (their right-hand sides are 0). So they always have a nonzero solution when the number of unknowns exceeds the number of equations. A solution, in our case, turns out to be

$$\begin{aligned}
\beta_0(n) &= (n+1)^3\,,\\
\beta_1(n) &= -(2n+3)(17n^2+51n+39)\,,\\
\beta_2(n) &= (n+2)^3\,,\\
\alpha_0(n) &= -16(n+1)(n+2)(2n+3)\,,\\
\alpha_1(n) &= -12(2n+3)\,,\\
\alpha_2(n) &= 8(2n+3)\,.
\end{aligned}$$

Consequently

$$\begin{aligned}
(n+1)^3 t(n,k) - (2n+3)(17n^2+51n+39)t(n+1,k)\\
\qquad + (n+2)^3 t(n+2,k) \;=\; T(n,k+1) - T(n,k)\,,
\end{aligned}$$

where $T(n,k) = k^4 s(n,k)\bar{t}(n,k) = (2n+3)(8k^2-12k-16(n+1)(n+2)) \times (n+k)!^2/(k-1)!^4(n+2-k)!^2$. Summing on k gives Apéry's once-incredible recurrence,

$$(n+1)^3 A_n + (n+2)^3 A_{n+2} \;=\; (2n+3)(17n^2+51n+39)A_{n+1}\,. \qquad (5.142)$$

"Professor Littlewood, when he makes use of an algebraic identity, always saves himself the trouble of proving it; he maintains that an identity, if true, can be verified in a few lines by anybody obtuse enough to feel the need of verification. My object in the following pages is to confute this assertion."
— F. J. Dyson [89]

Does the Gosper-Zeilberger method work with all the sums we've encountered in this chapter? No. It doesn't apply when $t(n,k)$ is the summand $\binom{n}{k}(k+1)^{k-1}(n-k+1)^{n-k-1}$ in (5.65), because the term ratio $t(n,k+1)/t(n,k)$ is not a rational function of k. It also fails to handle cases like $t(n,k) = \binom{n}{k}n^k$, because the other term ratio $t(n+1,k)/t(n,k)$ is not a rational function of k. (We can do that one, however, by summing $\binom{n}{k}z^k$ and then setting $z = n$.) And it fails on a comparatively simple summand like $t(n,k) = 1/(nk+1)$, even though both $t(n,k+1)/t(n,k)$ and $t(n+1,k)/t(n,k)$ are rational functions of n and k; see exercise 107.

But the Gosper-Zeilberger algorithm is guaranteed to succeed in an enormous number of cases, namely whenever the summand $t(n,k)$ is a so-called *proper term* — a term that can be written in the form

$$t(n,k) \;=\; f(n,k)\frac{(a_1n+a_1'k+a_1'')!\,\ldots(a_pn+a_p'k+a_p'')!}{(b_1n+b_1'k+b_1'')!\,\ldots(b_qn+b_q'k+b_q'')!}\,w^n z^k\,. \qquad (5.143)$$

Here $f(n,k)$ is a polynomial in n and k; the coefficients a_1, a_1', ..., a_p, a_p', b_1, b_1', ..., b_q, b_q' are specific integer constants; the parameters w and z

are nonzero; and the other quantities $a''_1, \ldots, a''_p, b''_1, \ldots, b''_q$ are arbitrary complex numbers. We will prove that whenever $t(n,k)$ is a proper term, there exist polynomials $\beta_0(n), \ldots, \beta_l(n)$, not all zero, and a proper term $T(n,k)$, such that

What happens if $t(n,k)$ is independent of n?

$$\beta_0(n)t(n,k) + \cdots + \beta_l(n)t(n+l,k) \;=\; T(n,k+1) - T(n,k)\,. \qquad (5.144)$$

The following proof is due to Wilf and Zeilberger [374].

Let N be the operator that increases n by 1, and let K be the operator that increases k by 1, so that, for example, $N^2K^3t(n,k) = t(n+2,k+3)$. We will study linear difference operators in N, K, and n, namely operator polynomials of the form

$$H(N,K,n) \;=\; \sum_{i=0}^{I}\sum_{j=0}^{J} \alpha_{i,j}(n)N^iK^j\,, \qquad (5.145)$$

where each $\alpha_{i,j}(n)$ is a polynomial in n. Our first observation is that, if $t(n,k)$ is any proper term and $H(N,K,n)$ is any linear difference operator, then $H(N,K,n)t(n,k)$ is a proper term. Suppose t and H are given respectively by (5.143) and (5.145); then we define a "base term"

$$\bar{t}(n,k)_{I,J} \;=\; \frac{\prod_{i=1}^{p}\bigl(a_in + a'_ik + a_iI[a_i<0] + a'_iJ[a'_i<0] + a''_i\bigr)!}{\prod_{i=1}^{q}\bigl(b_in + b'_ik + b_iI[b_i>0] + b'_iJ[b'_i>0] + b''_i\bigr)!}\, w^nz^k\,.$$

For example, if $t(n,k)$ is $\binom{n-2k}{k} = (n-2k)!/k!\,(n-3k)!$, the base term corresponding to a linear difference operator of degrees I and J is $\bar{t}(n,k)_{I,J} = (n-2k-2J)!/(k+J)!\,(n-3k+I)!$. The point is that $\alpha_{i,j}(n)N^iK^jt(n,k)$ is equal to $\bar{t}(n,k)_{I,J}$ times a polynomial in n and k, whenever $0 \leqslant i \leqslant I$ and $0 \leqslant j \leqslant J$. A finite sum of polynomials is a polynomial, so $H(N,K,n)t(n,k)$ has the required form (5.143).

The next step is to show that whenever $t(n,k)$ is a proper term, there is always a nonzero linear difference operator $H(N,K,n)$ such that

$$H(N,K,n)t(n,k) \;=\; 0\,.$$

If $0 \leqslant i \leqslant I$ and $0 \leqslant j \leqslant J$, the shifted term $N^iK^jt(n,k)$ is $\bar{t}(n,k)_{I,J}$ times a polynomial in n and k that has degree at most

$$\begin{aligned} D_{I,J} \;=\; \deg(f) &+ |a_1|I + |a'_1|J + \cdots + |a_p|I + |a'_p|J \\ &+ |b_1|I + |b'_1|J + \cdots + |b_q|I + |b'_q|J \end{aligned}$$

in the variable k. Hence the desired H exists if we can solve $D_{I,J}+1$ homogeneous linear equations in the $(I+1)(J+1)$ variables $\alpha_{i,j}(n)$, with coefficients

that are polynomials in n. All we need to do is choose I and J large enough that $(I+1)(J+1) > D_{I,J} + 1$. For example, we can take $I = 2A' + 1$ and $J = 2A + \deg(f)$, where

$$\begin{aligned} A &= |a_1| + \cdots + |a_p| + |b_1| + \cdots + |b_q|; \\ A' &= |a'_1| + \cdots + |a'_p| + |b'_1| + \cdots + |b'_q|. \end{aligned}$$

The last step in the proof is to go from the equation $H(N,K,n)t(n,k) = 0$ to a solution of (5.144). Let H be chosen so that J is minimized, i.e., so that H has the smallest possible degree in K. We can write

The trick here is based on regarding H as a polynomial in K and then replacing K by $\Delta + 1$.

$$H(N,K,n) = H(N,1,n) - (K-1)G(N,K,n)$$

for some linear difference operator $G(N,K,n)$. Let $H(N,1,n) = \beta_0(n) + \beta_1(n)N + \cdots + \beta_l(n)N^l$ and $T(n,k) = G(N,K,n)t(n,k)$. Then $T(n,k)$ is a proper term, and (5.144) holds.

The proof is almost complete; we still have to verify that $H(N,1,n)$ is not simply the zero operator. If it is, then $T(n,k)$ is independent of k. So there are polynomials $\beta_0(n)$ and $\beta_1(n)$ such that $\big(\beta_0(n) + \beta_1(n)N\big)T(n,k) = 0$. But then $\big(\beta_0(n) + \beta_1(n)N\big)G(N,K,n)$ is a nonzero linear difference operator of degree $J-1$ that annihilates $t(n,k)$; this contradicts the minimality of J, and our proof of (5.144) is complete.

Once we know that (5.144) holds, for some proper term T, we can be sure that Gosper's algorithm will succeed in finding T (or T plus a constant). Although we proved Gosper's algorithm only for the case of hypergeometric terms $t(k)$ in a single variable k, our proof can be extended to the two-variable case, as follows: There are infinitely many complex numbers n for which condition (5.118) holds when $q(n,k)$ and $r(n,k)$ are completely factored as polynomials in k, and for which the calculations of d in Step 2 agree with the calculations of Gosper's one-variable algorithm. For all such n, our previous proof shows that a suitable polynomial $s(n,k)$ in k exists; therefore a suitable polynomial $s(n,k)$ in n and k exists; QED.

We have proved that the Gosper-Zeilberger algorithm will discover a solution to (5.144), for some l, where l is as small as possible. That solution gives us a recurrence in n for evaluating the sum over k of any proper term $t(n,k)$, provided that $t(n,k)$ is nonzero for only finitely many k. And the roles of n and k can, of course, be reversed, because the definition of proper term in (5.143) is symmetrical in n and k.

Exercises 99–108 provide additional examples of the Gosper-Zeilberger algorithm, illustrating some of its versatility. Wilf and Zeilberger [374] have significantly extended these results to methods that handle generalized binomial coefficients and multiple indices of summation.

Exercises

Warmups

1 What is 11^4? Why is this number easy to compute, for a person who knows binomial coefficients?

2 For which value(s) of k is $\binom{n}{k}$ a maximum, when n is a given positive integer? Prove your answer.

3 Prove the hexagon property,

$$\binom{n-1}{k-1}\binom{n}{k+1}\binom{n+1}{k} = \binom{n-1}{k}\binom{n+1}{k+1}\binom{n}{k-1}.$$

4 Evaluate $\binom{-1}{k}$ by negating (actually un-negating) its upper index.

5 Let p be prime. Show that $\binom{p}{k} \bmod p = 0$ for $0 < k < p$. What does this imply about the binomial coefficients $\binom{p-1}{k}$?

6 Fix up the text's derivation in Problem 6, Section 5.2, by correctly applying symmetry. *A case of mistaken identity.*

7 Is (5.34) true also when $k < 0$?

8 Evaluate

$$\sum_k \binom{n}{k}(-1)^k(1-k/n)^n.$$

What is the approximate value of this sum, when n is very large? *Hint:* The sum is $\Delta^n f(0)$ for some function f.

9 Show that the generalized exponentials of (5.58) obey the law

$$\mathcal{E}_t(z) = \mathcal{E}(tz)^{1/t}, \qquad \text{if } t \neq 0,$$

where $\mathcal{E}(z)$ is an abbreviation for $\mathcal{E}_1(z)$.

10 Show that $-2\big(\ln(1-z)+z\big)/z^2$ is a hypergeometric function.

11 Express the two functions

$$\sin z = z - \frac{z^3}{3!} + \frac{z^5}{5!} - \frac{z^7}{7!} + \cdots$$

$$\arcsin z = z + \frac{1\cdot z^3}{2\cdot 3} + \frac{1\cdot 3\cdot z^5}{2\cdot 4\cdot 5} + \frac{1\cdot 3\cdot 5\cdot z^7}{2\cdot 4\cdot 6\cdot 7} + \cdots$$

in terms of hypergeometric series.

12 Which of the following functions of k is a hypergeometric term, as defined in Section 5.7? Explain why or why not.

a n^k.
b k^n.
c $\big(k! + (k+1)!\big)/2$.
d H_k, that is, $\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{k}$.
e $1/\binom{n}{k}$.
f $t(k)T(k)$, when t and T are hypergeometric terms.
g $t(k) + T(k)$, when t and T are hypergeometric terms.
h $t(n-k)$, when t is a hypergeometric term.
i $a\,t(k) + b\,t(k{+}1) + c\,t(k{+}2)$, when t is a hypergeometric term.
j $\lceil k/2 \rceil$.
k $k\,[k>0]$.

(Here t and T aren't necessarily related as in (5.120).)

Basics

13 Find relations between the superfactorial function $P_n = \prod_{k=1}^n k!$ of exercise 4.55, the hyperfactorial function $Q_n = \prod_{k=1}^n k^k$, and the product $R_n = \prod_{k=0}^n \binom{n}{k}$.

14 Prove identity (5.25) by negating the upper index in Vandermonde's convolution (5.22). Then show that another negation yields (5.26).

15 What is $\sum_k \binom{n}{k}^3 (-1)^k$? *Hint:* See (5.29).

16 Evaluate the sum

$$\sum_k \binom{2a}{a+k}\binom{2b}{b+k}\binom{2c}{c+k}(-1)^k$$

when a, b, c are nonnegative integers.

17 Find a simple relation between $\binom{2n-1/2}{n}$ and $\binom{2n-1/2}{2n}$.

18 Find an alternative form analogous to (5.35) for the product

$$\binom{r}{k}\binom{r-1/3}{k}\binom{r-2/3}{k}.$$

19 Show that the generalized binomials of (5.58) obey the law

$$\mathcal{B}_t(z) \;=\; \mathcal{B}_{1-t}(-z)^{-1}\,.$$

20 Define a "generalized bloopergeometric series" by the formula

$$G\left(\begin{matrix} a_1, \ldots, a_m \\ b_1, \ldots, b_n \end{matrix}\middle| z\right) \;=\; \sum_{k\geqslant 0} \frac{a_1^{\underline{k}} \ldots a_m^{\underline{k}}}{b_1^{\underline{k}} \ldots b_n^{\underline{k}}} \frac{z^k}{k!}\,,$$

using falling powers instead of the rising ones in (5.76). Explain how G is related to F.

21 Show that Euler's definition of factorials is consistent with the ordinary definition, by showing that the limit in (5.83) is $1/\big((m-1)\ldots(1)\big)$ when $z = m$ is a positive integer.

22 Use (5.83) to prove the *factorial duplication formula*:

$$x!\,(x-\tfrac{1}{2})! \;=\; (2x)!\,(-\tfrac{1}{2})!/2^{2x}\,.$$

By the way, $(-\frac{1}{2})! = \sqrt{\pi}$.

23 What is the value of $F(-n, 1;; 1)$?

24 Find $\sum_k \binom{n}{m+k}\binom{m+k}{2k}4^k$ by using hypergeometric series.

25 Show that

$$(a_1 - b_1)\,F\left(\begin{matrix} a_1,\, a_2,\, \ldots,\, a_m \\ b_1{+}1,\, b_2,\, \ldots,\, b_n \end{matrix}\middle|\, z\right)$$
$$= a_1\,F\left(\begin{matrix} a_1{+}1,\, a_2,\, \ldots,\, a_m \\ b_1{+}1,\, b_2,\, \ldots,\, b_n \end{matrix}\middle|\, z\right) - b_1\,F\left(\begin{matrix} a_1,\, a_2,\, \ldots,\, a_m \\ b_1,\, b_2,\, \ldots,\, b_n \end{matrix}\middle|\, z\right).$$

Find a similar relation between the hypergeometrics

$$F\left(\begin{matrix} a_1,\, a_2,\, a_3,\, \ldots,\, a_m \\ b_1,\, \ldots,\, b_n \end{matrix}\middle|\, z\right),$$
$$F\left(\begin{matrix} a_1{+}1,\, a_2,\, a_3,\, \ldots,\, a_m \\ b_1,\, \ldots,\, b_n \end{matrix}\middle|\, z\right), \qquad \text{and}$$
$$F\left(\begin{matrix} a_1,\, a_2{+}1,\, a_3,\, \ldots,\, a_m \\ b_1,\, \ldots,\, b_n \end{matrix}\middle|\, z\right).$$

26 Express the function $G(z)$ in the formula

$$F\left(\begin{matrix} a_1,\, \ldots,\, a_m \\ b_1,\, \ldots,\, b_n \end{matrix}\middle|\, z\right) \;=\; 1 + G(z)$$

as a multiple of a hypergeometric series.

27 Prove that

$$F\left(\begin{matrix} a_1,\, a_1{+}\frac{1}{2},\, \ldots,\, a_m,\, a_m{+}\frac{1}{2} \\ b_1,\, b_1{+}\frac{1}{2},\, \ldots,\, b_n,\, b_n{+}\frac{1}{2},\, \frac{1}{2} \end{matrix}\middle|\, (2^{m-n-1}z)^2\right)$$
$$= \frac{1}{2}\left(F\left(\begin{matrix} 2a_1,\ldots,2a_m \\ 2b_1,\ldots,2b_n \end{matrix}\middle|\, z\right) + F\left(\begin{matrix} 2a_1,\ldots,2a_m \\ 2b_1,\ldots,2b_n \end{matrix}\middle|\, -z\right)\right).$$

28 Prove *Euler's identity*

$$F\left(\begin{matrix} a, b \\ c \end{matrix}\middle|\, z\right) \;=\; (1-z)^{c-a-b}\,F\left(\begin{matrix} c-a,\, c-b \\ c \end{matrix}\middle|\, z\right)$$

by applying Pfaff's reflection law (5.101) twice.

29 Show that confluent hypergeometrics satisfy

$$e^z F\left(\begin{matrix} a \\ b \end{matrix}\middle| -z\right) = F\left(\begin{matrix} b-a \\ b \end{matrix}\middle| z\right).$$

30 What hypergeometric series F satisfies $zF'(z) + F(z) = 1/(1-z)$?

31 Show that if $f(k)$ is any function summable in hypergeometric terms, then f itself is a hypergeometric term. For example, if $\sum f(k)\,\delta k = cF(A_1, \ldots, A_M; B_1, \ldots, B_N; Z)_k + C$, then there are constants $a_1, \ldots, a_m$, $b_1, \ldots, b_n$, and z such that $f(k)$ is a multiple of (5.115).

32 Find $\sum k^2\,\delta k$ by Gosper's method.

33 Use Gosper's method to find $\sum \delta k/(k^2-1)$.

34 Show that a partial hypergeometric sum can always be represented as a limit of ordinary hypergeometrics:

$$\sum_{k\leqslant c} F\left(\begin{matrix} a_1, \ldots, a_m \\ b_1, \ldots, b_n \end{matrix}\middle| z\right)_k = \lim_{\epsilon\to 0} F\left(\begin{matrix} -c, a_1, \ldots, a_m \\ \epsilon-c, b_1, \ldots, b_n \end{matrix}\middle| z\right),$$

when c is a nonnegative integer. (See (5.115▷) Use this idea to evaluate $\sum_{k\leqslant m}\binom{n}{k}(-1)^k$.

Homework exercises

35 The notation $\sum_{k\leqslant n}\binom{n}{k}2^{k-n}$ is ambiguous without context. Evaluate it

a as a sum on k;

b as a sum on n.

36 Let p^k be the largest power of the prime p that divides $\binom{m+n}{m}$, when m and n are nonnegative integers. Prove that k is the number of carries that occur when m is added to n in the radix p number system. *Hint:* Exercise 4.24 helps here.

37 Show that an analog of the binomial theorem holds for factorial powers. That is, prove the identities

$$(x+y)^{\underline{n}} = \sum_k \binom{n}{k} x^{\underline{k}}\, y^{\underline{n-k}},$$

$$(x+y)^{\overline{n}} = \sum_k \binom{n}{k} x^{\overline{k}}\, y^{\overline{n-k}},$$

for all nonnegative integers n.

38 Show that all nonnegative integers n can be represented uniquely in the form $n = \binom{a}{1}+\binom{b}{2}+\binom{c}{3}$ where a, b, and c are integers with $0 \leqslant a < b < c$. (This is called the *binomial number system.*)

39 Show that if $xy = ax + by$ then

$$x^n y^n \;=\; \sum_{k=1}^{n} \binom{2n-1-k}{n-1} (a^n b^{n-k} x^k + a^{n-k} b^n y^k)$$

for all $n > 0$. Find a similar formula for the more general product $x^m y^n$. (These formulas give useful partial fraction expansions, for example when $x = 1/(z-c)$ and $y = 1/(z-d)$.)

40 Find a closed form for

$$\sum_{j=1}^{m} (-1)^{j+1} \binom{r}{j} \sum_{k=1}^{n} \binom{-j+rk+s}{m-j}, \qquad \text{integers } m, n \geqslant 0.$$

41 Evaluate $\sum_k \binom{n}{k} k!/(n+1+k)!$ when n is a nonnegative integer.

42 Find the indefinite sum $\sum \left((-1)^x / \binom{n}{x}\right) \delta x$, and use it to compute the sum $\sum_{k=0}^{n} (-1)^k / \binom{n}{k}$ in closed form.

43 Prove the triple-binomial identity (5.28). *Hint:* First replace $\binom{r+k}{m+n}$ by $\sum_j \binom{r}{m+n-j}\binom{k}{j}$.

44 Use identity (5.32) to find closed forms for the double sums

$$\sum_{j,k} (-1)^{j+k} \binom{j+k}{j} \binom{a}{j} \binom{b}{k} \binom{m+n-j-k}{m-j} \quad \text{and}$$
$$\sum_{j,k \geqslant 0} (-1)^{j+k} \binom{a}{j} \binom{m}{j} \binom{b}{k} \binom{n}{k} \Big/ \binom{m+n}{j+k},$$

given integers $m \geqslant a \geqslant 0$ and $n \geqslant b \geqslant 0$.

45 Find a closed form for $\sum_{k \leqslant n} \binom{2k}{k} 4^{-k}$.

46 Evaluate the following sum in closed form, when n is a positive integer:

$$\sum_k \binom{2k-1}{k} \binom{4n-2k-1}{2n-k} \frac{(-1)^{k-1}}{(2k-1)(4n-2k-1)}.$$

Hint: Generating functions win again.

47 The sum

$$\sum_k \binom{rk+s}{k} \binom{rn-rk-s}{n-k}$$

is a polynomial in r and s. Show that it doesn't depend on s.

48 The identity $\sum_{k\leqslant n}\binom{n+k}{n}2^{-k} = 2^n$ can be combined with the formula $\sum_{k\geqslant 0}\binom{n+k}{n}z^k = 1/(1-z)^{n+1}$ to yield

$$\sum_{k>n}\binom{n+k}{n}2^{-k} \;=\; 2^n\,.$$

What is the hypergeometric form of the latter identity?

49 Use the hypergeometric method to evaluate

$$\sum_k (-1)^k\binom{x}{k}\binom{x+n-k}{n-k}\frac{y}{y+n-k}\,.$$

50 Prove Pfaff's reflection law (5.101) by comparing the coefficients of z^n on both sides of the equation.

51 The derivation of (5.104) shows that

$$\lim_{\epsilon\to 0} F(-m,-2m-1+\epsilon;-2m+\epsilon;2) \;=\; 1\Big/\binom{-1/2}{m}\,.$$

In this exercise we will see that slightly different limiting processes lead to distinctly different answers for the degenerate hypergeometric series $F(-m,-2m-1;-2m;2)$.

a Show that $\lim_{\epsilon\to 0} F(-m+\epsilon,-2m-1;-2m+2\epsilon;2) = 0$, by using Pfaff's reflection law to prove the identity $F(a,-2m-1;2a;2) = 0$ for all integers $m \geqslant 0$.

b What is $\lim_{\epsilon\to 0} F(-m+\epsilon,-2m-1;-2m+\epsilon;2)$?

52 Prove that if N is a nonnegative integer,

$$\begin{aligned} &b_1^{\overline{N}}\ldots b_n^{\overline{N}}\,F\left(\begin{matrix}a_1,\ldots,a_m,-N\\ b_1,\ldots,b_n\end{matrix}\,\middle|\,z\right)\\ &\quad= a_1^{\overline{N}}\ldots a_m^{\overline{N}}(-z)^N\,F\left(\begin{matrix}1-b_1-N,\ldots,1-b_n-N,-N\\ 1-a_1-N,\ldots,1-a_m-N\end{matrix}\,\middle|\,\frac{(-1)^{m+n}}{z}\right).\end{aligned}$$

53 If we put $b = -\frac{1}{2}$ and $z = 1$ in Gauss's identity (5.110), the left side reduces to -1 while the right side is $+1$. Why doesn't this prove that $-1 = +1$?

54 Explain how the right-hand side of (5.112) was obtained.

55 If the hypergeometric terms $t(k) = F(a_1,\ldots,a_m;\,b_1,\ldots,b_n;\,z)_k$ and $T(k) = F(A_1,\ldots,A_M;\,B_1,\ldots,B_N;\,Z)_k$ satisfy $t(k) = c\big(T(k+1)-T(k)\big)$ for all $k \geqslant 0$, show that $z = Z$ and $m-n = M-N$.

56 Find a general formula for $\sum\binom{-3}{k}\,\delta k$ using Gosper's method. Show that $(-1)^{k-1}\lfloor\frac{k+1}{2}\rfloor\lfloor\frac{k+2}{2}\rfloor$ is also a solution.

57 Use Gosper's method to find a constant θ such that

$$\sum \binom{n}{k} z^k (k+\theta)\,\delta k$$

is summable in hypergeometric terms.

58 If m and n are integers with $0 \leqslant m \leqslant n$, let

$$T_{m,n} \;=\; \sum_{0 \leqslant k < n} \binom{k}{m} \frac{1}{n-k}\,.$$

Find a relation between $T_{m,n}$ and $T_{m-1,n-1}$, then solve your recurrence by applying a summation factor.

Exam problems

59 Find a closed form for

$$\sum_{k \geqslant 1} \binom{n}{\lfloor \log_m k \rfloor}$$

when m and n are positive integers.

60 Use Stirling's approximation (4.23) to estimate $\binom{m+n}{n}$ when m and n are both large. What does your formula reduce to when $m = n$?

61 Prove that when p is prime, we have

$$\binom{n}{m} \;\equiv\; \binom{\lfloor n/p \rfloor}{\lfloor m/p \rfloor} \binom{n \bmod p}{m \bmod p} \pmod{p}\,,$$

for all nonnegative integers m and n.

62 Assuming that p is prime and that m and n are positive integers, determine the value of $\binom{np}{mp} \bmod p^2$. *Hint:* You may wish to use the following generalization of Vandermonde's convolution:

$$\sum_{k_1+k_2+\cdots+k_m=n} \binom{r_1}{k_1}\binom{r_2}{k_2}\cdots\binom{r_m}{k_m} \;=\; \binom{r_1+r_2+\cdots+r_m}{n}\,.$$

63 Find a closed form for

$$\sum_{k=0}^{n} (-4)^k \binom{n+k}{2k}\,,$$

given an integer $n \geqslant 0$.

64 Evaluate $\sum_{k=0}^{n} \binom{n}{k} \Big/ \left\lceil \frac{k+1}{2} \right\rceil$, given an integer $n \geqslant 0$.

65 Prove that

$$\sum_k \binom{n-1}{k} n^{-k}(k+1)! \;=\; n\,.$$

66 Evaluate "Harry's double sum,"

$$\sum_{0 \leqslant j \leqslant k} \binom{-1}{j - \lfloor\sqrt{k-j}\rfloor} \binom{j}{m} \frac{1}{2^j}\,, \qquad \text{integer } m \geqslant 0,$$

as a function of m. (The sum is over both j and k.)

67 Find a closed form for

$$\sum_{k=0}^{n} \binom{\binom{k}{2}}{2} \binom{2n-k}{n}\,, \qquad \text{integer } n \geqslant 0.$$

68 Find a closed form for

$$\sum_k \binom{n}{k} \min(k, n-k)\,, \qquad \text{integer } n \geqslant 0.$$

69 Find a closed form for

$$\min_{\substack{k_1,\ldots,k_m \geqslant 0 \\ k_1+\cdots+k_m=n}} \sum_{j=1}^{m} \binom{k_j}{2}$$

as a function of m and n.

70 Find a closed form for

$$\sum_k \binom{n}{k} \binom{2k}{k} \left(\frac{-1}{2}\right)^k\,, \qquad \text{integer } n \geqslant 0.$$

71 Let

$$S_n \;=\; \sum_{k \geqslant 0} \binom{n+k}{m+2k} a_k\,,$$

where m and n are nonnegative integers, and let $A(z) = \sum_{k \geqslant 0} a_k z^k$ be the generating function for the sequence $\langle a_0, a_1, a_2, \ldots \rangle$.

a Express the generating function $S(z) = \sum_{n \geqslant 0} S_n z^n$ in terms of $A(z)$.

b Use this technique to solve Problem 7 in Section 5.2.

72 Prove that, if m, n, and k are integers and $n > 0$,

$$\binom{m/n}{k} n^{2k-\nu(k)} \quad \text{is an integer,}$$

where $\nu(k)$ is the number of 1's in the binary representation of k.

73 Use the repertoire method to solve the recurrence

$$\begin{aligned} X_0 &= \alpha; \qquad X_1 = \beta; \\ X_n &= (n-1)(X_{n-1} + X_{n-2}), \qquad \text{for } n > 1. \end{aligned}$$

Hint: Both $n!$ and $n_{¡}$ satisfy this recurrence.

74 This problem concerns a deviant version of Pascal's triangle in which the sides consist of the numbers 1, 2, 3, 4, ... instead of all 1's, although the interior numbers still satisfy the addition formula:

$$\begin{array}{ccccccccccc} & & & & & 1 & & & & & \\ & & & & 2 & & 2 & & & & \\ & & & 3 & & 4 & & 3 & & & \\ & & 4 & & 7 & & 7 & & 4 & & \\ & 5 & & 11 & & 14 & & 11 & & 5 & \\ \cdot & & \cdot & & \cdot & & \cdot & & \cdot & & \cdot \end{array}$$

If $\left(\!\binom{n}{k}\!\right)$ denotes the kth number in row n, for $1 \leqslant k \leqslant n$, we have $\left(\!\binom{n}{1}\!\right) = \left(\!\binom{n}{n}\!\right) = n$, and $\left(\!\binom{n}{k}\!\right) = \left(\!\binom{n-1}{k}\!\right) + \left(\!\binom{n-1}{k-1}\!\right)$ for $1 < k < n$. Express the quantity $\left(\!\binom{n}{k}\!\right)$ in closed form.

75 Find a relation between the functions

$$\begin{aligned} S_0(n) &= \sum_k \binom{n}{3k}, \\ S_1(n) &= \sum_k \binom{n}{3k+1}, \\ S_2(n) &= \sum_k \binom{n}{3k+2} \end{aligned}$$

and the quantities $\lfloor 2^n/3 \rfloor$ and $\lceil 2^n/3 \rceil$.

76 Solve the following recurrence for $n, k \geqslant 0$:

$$\begin{aligned} Q_{n,0} &= 1; \qquad Q_{0,k} = [k=0]; \\ Q_{n,k} &= Q_{n-1,k} + Q_{n-1,k-1} + \binom{n}{k}, \qquad \text{for } n, k > 0. \end{aligned}$$

77 What is the value of

$$\sum_{0\leqslant k_1,\ldots,k_m\leqslant n}\ \prod_{1\leqslant j<m}\binom{k_{j+1}}{k_j}, \qquad \text{if } m>1?$$

78 Assuming that m is a positive integer, find a closed form for

$$\sum_{k=0}^{2m^2}\binom{k \bmod m}{(2k+1) \bmod (2m+1)}.$$

79 **a** What is the greatest common divisor of $\binom{2n}{1}$, $\binom{2n}{3}$, $\ldots$, $\binom{2n}{2n-1}$? *Hint:* Consider the sum of these n numbers.

b Show that the least common multiple of $\binom{n}{0}$, $\binom{n}{1}$, $\ldots$, $\binom{n}{n}$ is equal to $L(n+1)/(n+1)$, where $L(n)=\operatorname{lcm}(1,2,\ldots,n)$.

Handy to know.

80 Prove that $\binom{n}{k}\leqslant(en/k)^k$ for all integers $k,n\geqslant 0$.

81 If $0<\theta<1$ and $0\leqslant x\leqslant 1$, and if l,m,n are nonnegative integers with $m<n$, prove the inequality

$$(-1)^{n-m-1}\sum_k\binom{l}{k}\binom{m+\theta}{n+k}x^k \;>\; 0\,.$$

Hint: Consider taking the derivative with respect to x.

Bonus problems

82 Prove that Pascal's triangle has an even more surprising hexagon property than the one cited in the text:

$$\gcd\Bigl(\tbinom{n-1}{k-1},\tbinom{n}{k+1},\tbinom{n+1}{k}\Bigr) \;=\; \gcd\Bigl(\tbinom{n-1}{k},\tbinom{n+1}{k+1},\tbinom{n}{k-1}\Bigr),$$

if $0<k<n$. For example, $\gcd(56,36,210)=\gcd(28,120,126)=2$.

83 Prove the amazing five-parameter double-sum identity (5.32).

84 Show that the second pair of convolution formulas, (5.61), follows from the first pair, (5.60). *Hint:* Differentiate with respect to z.

85 Prove that

$$\sum_{m=1}^{n}(-1)^m\sum_{1\leqslant k_1<k_2<\cdots<k_m\leqslant n}\binom{k_1^3+k_2^3+\cdots+k_m^3+2^n}{n} \;=\; (-1)^n n!^3-\binom{2^n}{n}.$$

(The left side is a sum of 2^n-1 terms.) *Hint:* Much more is true.

86 Let $a_1, \ldots, a_n$ be nonnegative integers, and let $C(a_1, \ldots, a_n)$ be the coefficient of the constant term $z_1^0 \ldots z_n^0$ when the $n(n-1)$ factors

$$\prod_{\substack{1 \leqslant i,j \leqslant n \\ i \neq j}} \left(1 - \frac{z_i}{z_j}\right)^{a_i}$$

are fully expanded into positive and negative powers of the complex variables $z_1, \ldots, z_n$.

a Prove that $C(a_1, \ldots, a_n)$ equals the left-hand side of (5.31).

b Prove that if $z_1, \ldots, z_n$ are distinct complex numbers, then the polynomial

$$f(z) = \sum_{k=1}^{n} \prod_{\substack{1 \leqslant j \leqslant n \\ j \neq k}} \frac{z - z_j}{z_k - z_j}$$

is identically equal to 1.

c Multiply the original product of $n(n-1)$ factors by $f(0)$ and deduce that $C(a_1, a_2, \ldots, a_n)$ is equal to

$$\begin{aligned} &C(a_1 - 1, a_2, \ldots, a_n) + C(a_1, a_2 - 1, \ldots, a_n) \\ &\qquad + \cdots + C(a_1, a_2, \ldots, a_n - 1)\,. \end{aligned}$$

(This recurrence defines multinomial coefficients, so $C(a_1, \ldots, a_n)$ must equal the right-hand side of (5.31).)

87 Let m be a positive integer and let $\zeta = e^{\pi i/m}$. Show that

$$\begin{aligned} &\sum_{k \leqslant n/m} \binom{n - mk}{k} z^{mk} \\ &\quad = \frac{\mathcal{B}_{-m}(z^m)^{n+1}}{(1+m)\mathcal{B}_{-m}(z^m) - m} \\ &\qquad - \sum_{0 \leqslant j < m} \frac{\left(\zeta^{2j+1} z\, \mathcal{B}_{1+1/m}(\zeta^{2j+1} z)^{1/m}\right)^{n+1}}{(m+1)\mathcal{B}_{1+1/m}(\zeta^{2j+1} z)^{-1} - 1}\,. \end{aligned}$$

(This reduces to (5.74) in the special case $m = 1$.)

88 Prove that the coefficients s_k in (5.47) are equal to

$$(-1)^k \int_0^\infty e^{-t}(1 - e^{-t})^{k-1} \frac{dt}{t}\,,$$

for all $k > 1$; hence $|s_k| < 1/(k-1)$.

89 Prove that (5.19) has an infinite counterpart,

$$\sum_{k>m}\binom{m+r}{k}x^k y^{m-k} = \sum_{k>m}\binom{-r}{k}(-x)^k(x+y)^{m-k}, \quad \text{integer } m,$$

if $|x| < |y|$ and $|x| < |x+y|$. Differentiate this identity n times with respect to y and express it in terms of hypergeometrics; what relation do you get?

90 Problem 1 in Section 5.2 considers $\sum_{k\geqslant 0}\binom{r}{k}\Big/\binom{s}{k}$ when r and s are integers with $s \geqslant r \geqslant 0$. What is the value of this sum if r and s aren't integers?

91 Prove *Whipple's identity*,

$$F\left(\begin{matrix}\frac{1}{2}a,\ \frac{1}{2}a+\frac{1}{2},\ 1+a-b-c\\ 1+a-b,\ 1+a-c\end{matrix}\middle|\ \frac{-4z}{(1-z)^2}\right)$$
$$= (1-z)^a F\left(\begin{matrix}a,\ b,\ c\\ 1+a-b, 1+a-c\end{matrix}\middle|\ z\right),$$

by showing that both sides satisfy the same differential equation.

92 Prove *Clausen's product identities*

$$F\left(\begin{matrix}a,\ b\\ a+b+\frac{1}{2}\end{matrix}\middle|\ z\right)^2 = F\left(\begin{matrix}2a,\ a+b,\ 2b\\ 2a+2b,\ a+b+\frac{1}{2}\end{matrix}\middle|\ z\right);$$

$$F\left(\begin{matrix}\frac{1}{4}+a,\ \frac{1}{4}+b\\ 1+a+b\end{matrix}\middle|\ z\right) F\left(\begin{matrix}\frac{1}{4}-a,\ \frac{1}{4}-b\\ 1-a-b\end{matrix}\middle|\ z\right)$$
$$= F\left(\begin{matrix}\frac{1}{2},\ \frac{1}{2}+a-b,\ \frac{1}{2}-a+b\\ 1+a+b,\ 1-a-b\end{matrix}\middle|\ z\right).$$

What identities result when the coefficients of z^n on both sides of these formulas are equated?

93 Show that the indefinite sum

$$\sum\left(\prod_{j=1}^{k-1}(f(j)+\alpha)\Big/\prod_{j=1}^{k}f(j)\right)\delta k$$

has a (fairly) simple form, given any function f and any constant α.

94 Find $\sum\binom{a}{k}\binom{-a}{n-k}\delta k$.

95 What conditions in addition to (5.118) will make the polynomials p, q, r of (5.117) uniquely determined?

96 Prove that if Gosper's algorithm finds no solution to (5.120), given a hypergeometric term $t(k)$, then there is no solution to the more general equation

$$t(k) \;=\; \bigl(T_1(k+1)+\cdots+T_m(k+1)\bigr)-\bigl(T_1(k)+\cdots+T_m(k)\bigr)\,,$$

where $T_1(k)$, ..., $T_m(k)$ are hypergeometric terms.

97 Find all complex numbers z such that $k!^2/\prod_{j=1}^{k}(j^2+jz+1)$ is summable in hypergeometric terms.

98 What recurrence does the Gosper-Zeilberger method give for the sum $S_n=\sum_k\binom{n}{2k}$?

99 Use the Gosper-Zeilberger method to discover a closed form for $\sum_k t(n,k)$ when $t(n,k)=(n+a+b+c+k)!/(n-k)!\,(c+k)!\,(b-k)!\,(a-k)!\,k!$, assuming that a is a nonnegative integer.

100 Find a recurrence relation for the sum

$$S_n \;=\; \sum_{k=0}^{n}\frac{1}{\binom{n}{k}}\,,$$

and use the recurrence to find another formula for S_n.

101 Find recurrence relations satisfied by the sums

Better use computer algebra for this one (and the next few).

a $S_{m,n}(z)=\sum_k\binom{m}{k}\binom{n}{k}z^k\,;$

b $S_n(z)=S_{n,n}(z)=\sum_k\binom{n}{k}^2 z^k\,.$

102 Use the Gosper-Zeilberger procedure to generalize the "useless" identity (5.113): Find additional values of a, b, and z such that

$$\sum_k\binom{n}{k}\binom{\frac{1}{3}n-a}{k}z^k\Bigg/\binom{\frac{4}{3}n-b}{k}$$

has a simple closed form.

103 Let $t(n,k)$ be the proper term (5.143). What are the degrees of $\hat{p}(n,k)$, $q(n,k)$, and $r(n,k)$ in terms of the variable k, when the Gosper-Zeilberger procedure is applied to $\hat{t}(n,k)=\beta_0(n)t(n,k)+\cdots+\beta_l(n)t(n+l,k)$? (Ignore the rare, exceptional cases.)

104 Use the Gosper-Zeilberger procedure to verify the remarkable identity

$$\sum_k (-1)^k \binom{r-s-k}{k}\binom{r-2k}{n-k}\frac{1}{r-n-k+1} = \binom{s}{n}\frac{1}{r-2n+1}.$$

Explain why the simplest recurrence for this sum is not found.

105 Show that if $\omega = e^{2\pi i/3}$ we have

$$\sum_{k+l+m=3n} \binom{3n}{k,l,m}^2 \omega^{l-m} = \binom{4n}{n,n,2n}, \qquad \text{integer } n \geqslant 0.$$

106 Prove the amazing identity (5.32) by letting $t(r,j,k)$ be the summand divided by the right-hand side, then showing that there are functions $T(r,j,k)$ and $U(r,j,k)$ for which

$$\begin{aligned} t(r+1,j,k) - t(r,j,k) &= T(r,j+1,k) - T(r,j,k) \\ &\quad + U(r,j,k+1) - U(r,j,k). \end{aligned}$$

107 Prove that $1/(nk+1)$ is not a proper term.

108 Show that the Apéry numbers A_n of (5.141) are the diagonal elements $A_{n,n}$ of a matrix of numbers defined by

$$A_{m,n} = \sum_{j,k} \binom{m}{j}^2 \binom{m}{k}^2 \binom{2m+n-j-k}{2m}.$$

Prove, in fact, that this matrix is symmetric, and that

$$\begin{aligned} A_{m,n} &= \sum_k \binom{m+n-k}{k}^2 \binom{m+n-2k}{m-k}^2 \\ &= \sum_k \binom{m}{k}\binom{n}{k}\binom{m+k}{k}\binom{n+k}{k}. \end{aligned}$$

109 Prove that the Apéry numbers (5.141) satisfy

$$A_n \equiv A_{\lfloor n/p \rfloor} A_{n \bmod p} \pmod{p}$$

for all primes p and all integers $n \geqslant 0$.

Research problems

110 For what values of n is $\binom{2n}{n} \equiv (-1)^n \pmod{(2n+1)}$?

111 Let $q(n)$ be the smallest odd prime factor of the middle binomial coefficient $\binom{2n}{n}$. According to exercise 36, the odd primes p that do *not* divide $\binom{2n}{n}$ are those for which all digits in n's radix p representation are $(p-1)/2$ or less. Computer experiments have shown that $q(n) \leqslant 11$ for $1 < n < 10^{10000}$, except that $q(3160) = 13$.

a Is $q(n) \leqslant 11$ for all $n > 3160$?

b Is $q(n) = 11$ for infinitely many n?

A reward of $\$7 \cdot 11 \cdot 13$ is offered for a solution to either (a) or (b).

112 Is $\binom{2n}{n}$ divisible either by 4 or by 9, for all $n > 4$ except $n = 64$ and $n = 256$?

113 If $t(n+1,k)/t(n,k)$ and $t(n,k+1)/t(n,k)$ are rational functions of n and k, and if there is a nonzero linear difference operator $H(N,K,n)$ such that $H(N,K,n)t(n,k) = 0$, does it follow that $t(n,k)$ is a proper term?

114 Let m be a positive integer, and define the sequence $c_n^{(m)}$ by the recurrence

$$\sum_k \binom{n}{k}^m \binom{n+k}{k}^m = \sum_k \binom{n}{k}\binom{n+k}{k} c_k^{(m)} .$$

Are these numbers $c_n^{(m)}$ integers?

6

Special Numbers

SOME SEQUENCES of numbers arise so often in mathematics that we recognize them instantly and give them special names. For example, everybody who learns arithmetic knows the sequence of square numbers $\langle 1, 4, 9, 16, \ldots \rangle$. In Chapter 1 we encountered the triangular numbers $\langle 1, 3, 6, 10, \ldots \rangle$; in Chapter 4 we studied the prime numbers $\langle 2, 3, 5, 7, \ldots \rangle$; in Chapter 5 we looked briefly at the Catalan numbers $\langle 1, 2, 5, 14, \ldots \rangle$.

In the present chapter we'll get to know a few other important sequences. First on our agenda will be the Stirling numbers $\left\{ {n \atop k} \right\}$ and $\left[{n \atop k} \right]$, and the Eulerian numbers $\left\langle {n \atop k} \right\rangle$; these form triangular patterns of coefficients analogous to the binomial coefficients $\binom{n}{k}$ in Pascal's triangle. Then we'll take a good look at the harmonic numbers H_n, and the Bernoulli numbers B_n; these differ from the other sequences we've been studying because they're fractions, not integers. Finally, we'll examine the fascinating Fibonacci numbers F_n and some of their important generalizations.

6.1 STIRLING NUMBERS

We begin with some close relatives of the binomial coefficients, the Stirling numbers, named after James Stirling (1692–1770). These numbers come in two flavors, traditionally called by the no-frills names "Stirling numbers of the first and second kind." Although they have a venerable history and numerous applications, they still lack a standard notation. Following Jovan Karamata, we will write $\left\{ {n \atop k} \right\}$ for Stirling numbers of the second kind and $\left[{n \atop k} \right]$ for Stirling numbers of the first kind; these symbols turn out to be more user-friendly than the many other notations that people have tried.

"... par cette notation, les formules deviennent plus symétriques."
—J. Karamata [199]

Tables 258 and 259 show what $\left\{ {n \atop k} \right\}$ and $\left[{n \atop k} \right]$ look like when n and k are small. A problem that involves the numbers "1, 7, 6, 1" is likely to be related to $\left\{ {n \atop k} \right\}$, and a problem that involves "6, 11, 6, 1" is likely to be related to $\left[{n \atop k} \right]$, just as we assume that a problem involving "1, 4, 6, 4, 1" is likely to be related to $\binom{n}{k}$; these are the trademark sequences that appear when $n = 4$.

Table 258 Stirling's triangle for subsets.

n	$\left\{ n \atop 0 \right\}$	$\left\{ n \atop 1 \right\}$	$\left\{ n \atop 2 \right\}$	$\left\{ n \atop 3 \right\}$	$\left\{ n \atop 4 \right\}$	$\left\{ n \atop 5 \right\}$	$\left\{ n \atop 6 \right\}$	$\left\{ n \atop 7 \right\}$	$\left\{ n \atop 8 \right\}$	$\left\{ n \atop 9 \right\}$
0	1									
1	0	1								
2	0	1	1							
3	0	1	3	1						
4	0	1	7	6	1					
5	0	1	15	25	10	1				
6	0	1	31	90	65	15	1			
7	0	1	63	301	350	140	21	1		
8	0	1	127	966	1701	1050	266	28	1	
9	0	1	255	3025	7770	6951	2646	462	36	1

Stirling numbers of the second kind show up more often than those of the other variety, so let's consider last things first. The symbol $\left\{ n \atop k \right\}$ stands for the number of ways to partition a set of n things into k nonempty subsets. For example, there are seven ways to split a four-element set into two parts:

(Stirling himself considered this kind first in his book [343].)

$$\begin{aligned} &\{1,2,3\}\cup\{4\}, \quad &&\{1,2,4\}\cup\{3\}, \quad &&\{1,3,4\}\cup\{2\}, \quad &&\{2,3,4\}\cup\{1\}, \\ &\{1,2\}\cup\{3,4\}, \quad &&\{1,3\}\cup\{2,4\}, \quad &&\{1,4\}\cup\{2,3\}; \end{aligned} \tag{6.1}$$

thus $\left\{ 4 \atop 2 \right\} = 7$. Notice that curly braces are used to denote sets as well as the numbers $\left\{ n \atop k \right\}$. This notational kinship helps us remember the meaning of $\left\{ n \atop k \right\}$, which can be read "$n$ subset k."

Let's look at small k. There's just one way to put n elements into a single nonempty set; hence $\left\{ n \atop 1 \right\} = 1$, for all $n > 0$. On the other hand $\left\{ 0 \atop 1 \right\} = 0$, because a 0-element set is empty.

The case $k = 0$ is a bit tricky. Things work out best if we agree that there's just one way to partition an empty set into zero nonempty parts; hence $\left\{ 0 \atop 0 \right\} = 1$. But a nonempty set needs at least one part, so $\left\{ n \atop 0 \right\} = 0$ for $n > 0$.

What happens when $k = 2$? Certainly $\left\{ 0 \atop 2 \right\} = 0$. If a set of $n > 0$ objects is divided into two nonempty parts, one of those parts contains the last object and some subset of the first $n-1$ objects. There are 2^{n-1} ways to choose the latter subset, since each of the first $n-1$ objects is either in it or out of it; but we mustn't put all of those objects in it, because we want to end up with two nonempty parts. Therefore we subtract 1:

$$\left\{ n \atop 2 \right\} = 2^{n-1} - 1, \qquad \text{integer } n > 0. \tag{6.2}$$

(This tallies with our enumeration of $\left\{ 4 \atop 2 \right\} = 7 = 2^3 - 1$ ways above.)

Table 259 Stirling's triangle for cycles.

n	$\left[{n \atop 0}\right]$	$\left[{n \atop 1}\right]$	$\left[{n \atop 2}\right]$	$\left[{n \atop 3}\right]$	$\left[{n \atop 4}\right]$	$\left[{n \atop 5}\right]$	$\left[{n \atop 6}\right]$	$\left[{n \atop 7}\right]$	$\left[{n \atop 8}\right]$	$\left[{n \atop 9}\right]$
0	1									
1	0	1								
2	0	1	1							
3	0	2	3	1						
4	0	6	11	6	1					
5	0	24	50	35	10	1				
6	0	120	274	225	85	15	1			
7	0	720	1764	1624	735	175	21	1		
8	0	5040	13068	13132	6769	1960	322	28	1	
9	0	40320	109584	118124	67284	22449	4536	546	36	1

A modification of this argument leads to a recurrence by which we can compute $\left\{{n \atop k}\right\}$ for all k: Given a set of $n > 0$ objects to be partitioned into k nonempty parts, we either put the last object into a class by itself (in $\left\{{n-1 \atop k-1}\right\}$ ways), or we put it together with some nonempty subset of the first $n-1$ objects. There are $k\left\{{n-1 \atop k}\right\}$ possibilities in the latter case, because each of the $\left\{{n-1 \atop k}\right\}$ ways to distribute the first $n-1$ objects into k nonempty parts gives k subsets that the nth object can join. Hence

$$\left\{{n \atop k}\right\} = k\left\{{n-1 \atop k}\right\} + \left\{{n-1 \atop k-1}\right\}, \qquad \text{integer } n > 0. \tag{6.3}$$

This is the law that generates Table 258; without the factor of k it would reduce to the addition formula (5.8) that generates Pascal's triangle.

And now, Stirling numbers of the first kind. These are somewhat like the others, but $\left[{n \atop k}\right]$ counts the number of ways to arrange n objects into k *cycles* instead of subsets. We verbalize '$\left[{n \atop k}\right]$' by saying "n cycle k."

Cycles are cyclic arrangements, like the necklaces we considered in Chapter 4. The cycle

can be written more compactly as '$[A, B, C, D]$', with the understanding that

$$[A, B, C, D] = [B, C, D, A] = [C, D, A, B] = [D, A, B, C];$$

a cycle "wraps around" because its end is joined to its beginning. On the other hand, the cycle $[A, B, C, D]$ is not the same as $[A, B, D, C]$ or $[D, C, B, A]$.

There are eleven different ways to make two cycles from four elements:

"There are nine and sixty ways of constructing tribal lays, And-every-single-one-of-them-is-right."
—Rudyard Kipling

$$\begin{array}{llll} [1,2,3]\,[4]\,, & [1,2,4]\,[3]\,, & [1,3,4]\,[2]\,, & [2,3,4]\,[1]\,, \\ [1,3,2]\,[4]\,, & [1,4,2]\,[3]\,, & [1,4,3]\,[2]\,, & [2,4,3]\,[1]\,, \\ [1,2]\,[3,4]\,, & [1,3]\,[2,4]\,, & [1,4]\,[2,3]\,; & \end{array} \tag{6.4}$$

hence $\left[{4 \atop 2}\right] = 11$.

A singleton cycle (that is, a cycle with only one element) is essentially the same as a singleton set (a set with only one element). Similarly, a 2-cycle is like a 2-set, because we have $[A, B] = [B, A]$ just as $\{A, B\} = \{B, A\}$. But there are two *different* 3-cycles, $[A, B, C]$ and $[A, C, B]$. Notice, for example, that the eleven cycle pairs in (6.4) can be obtained from the seven set pairs in (6.1) by making two cycles from each of the 3-element sets.

In general, $n!/n = (n-1)!$ different n-cycles can be made from any n-element set, whenever $n > 0$. (There are $n!$ permutations, and each n-cycle corresponds to n of them because any one of its elements can be listed first.) Therefore we have

$$\left[{n \atop 1}\right] = (n-1)!\,, \qquad \text{integer } n > 0. \tag{6.5}$$

This is much larger than the value $\left\{{n \atop 1}\right\} = 1$ we had for Stirling subset numbers. In fact, it is easy to see that the cycle numbers must be at least as large as the subset numbers,

$$\left[{n \atop k}\right] \geqslant \left\{{n \atop k}\right\}, \qquad \text{integers } n, k \geqslant 0, \tag{6.6}$$

because every partition into nonempty subsets leads to at least one arrangement of cycles.

Equality holds in (6.6) when all the cycles are necessarily singletons or doubletons, because cycles are equivalent to subsets in such cases. This happens when $k = n$ and when $k = n-1$; hence

$$\left[{n \atop n}\right] = \left\{{n \atop n}\right\}; \qquad \left[{n \atop n-1}\right] = \left\{{n \atop n-1}\right\}.$$

In fact, it is easy to see that

$$\left[{n \atop n}\right] = \left\{{n \atop n}\right\} = 1\,; \qquad \left[{n \atop n-1}\right] = \left\{{n \atop n-1}\right\} = \binom{n}{2}. \tag{6.7}$$

(The number of ways to arrange n objects into $n-1$ cycles or subsets is the number of ways to choose the two objects that will be in the same cycle or subset.) The triangular numbers $\binom{n}{2} = 1, 3, 6, 10, \ldots$ are conspicuously present in both Table 258 and Table 259.

We can derive a recurrence for $\left[{n \atop k}\right]$ by modifying the argument we used for $\left\{{n \atop k}\right\}$. Every arrangement of n objects in k cycles either puts the last object into a cycle by itself (in $\left[{n-1 \atop k-1}\right]$ ways) or inserts that object into one of the $\left[{n-1 \atop k}\right]$ cycle arrangements of the first $n-1$ objects. In the latter case, there are $n-1$ different ways to do the insertion. (This takes some thought, but it's not hard to verify that there are j ways to put a new element into a j-cycle in order to make a $(j+1)$-cycle. When $j=3$, for example, the cycle $[A, B, C]$ leads to

$$[A, B, C, D], \qquad [A, B, D, C], \qquad \text{or} \qquad [A, D, B, C]$$

when we insert a new element D, and there are no other possibilities. Summing over all j gives a total of $n-1$ ways to insert an nth object into a cycle decomposition of $n-1$ objects.) The desired recurrence is therefore

$$\begin{bmatrix} n \\ k \end{bmatrix} = (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}, \qquad \text{integer } n > 0. \tag{6.8}$$

This is the addition-formula analog that generates Table 259.

Comparison of (6.8) and (6.3) shows that the first term on the right side is multiplied by its upper index $(n-1)$ in the case of Stirling cycle numbers, but by its lower index k in the case of Stirling subset numbers. We can therefore perform "absorption" in terms like $n\left[{n \atop k}\right]$ and $k\left\{{n \atop k}\right\}$, when we do proofs by mathematical induction.

Every permutation is equivalent to a set of cycles. For example, consider the permutation that takes 123456789 into 384729156. We can conveniently represent it in two rows,

$$\begin{array}{l} 1\,2\,3\,4\,5\,6\,7\,8\,9 \\ 3\,8\,4\,7\,2\,9\,1\,5\,6, \end{array}$$

showing that 1 becomes 3 and 2 becomes 8, etc. The cycle structure comes about because 1 becomes 3, which becomes 4, which becomes 7, which becomes the original element 1; that's the cycle $[1, 3, 4, 7]$. Another cycle in this permutation is $[2, 8, 5]$; still another is $[6, 9]$. Therefore the permutation 384729156 is equivalent to the cycle arrangement

$$[1, 3, 4, 7]\,[2, 8, 5]\,[6, 9]\,.$$

If we have any permutation $\pi_1\pi_2\ldots\pi_n$ of $\{1, 2, \ldots, n\}$, every element is in a unique cycle. For if we start with $m_0 = m$ and look at $m_1 = \pi_{m_0}$, $m_2 = \pi_{m_1}$, etc., we must eventually come back to $m_k = m_0$. (The numbers must repeat sooner or later, and the first number to reappear must be m_0 because we know the unique predecessors of the other numbers m_1, m_2, $\ldots$, m_{k-1}.)

Therefore every permutation defines a cycle arrangement. Conversely, every cycle arrangement obviously defines a permutation if we reverse the construction, and this one-to-one correspondence shows that permutations and cycle arrangements are essentially the same thing.

Therefore $\left[{n \atop k}\right]$ is the number of permutations of n objects that contain exactly k cycles. If we sum $\left[{n \atop k}\right]$ over all k, we must get the total number of permutations:

$$\sum_{k=0}^{n} \left[{n \atop k}\right] = n!\,, \qquad \text{integer } n \geqslant 0. \tag{6.9}$$

For example, $6 + 11 + 6 + 1 = 24 = 4!$.

Stirling numbers are useful because the recurrence relations (6.3) and (6.8) arise in a variety of problems. For example, if we want to represent ordinary powers x^n by falling powers $x^{\underline{n}}$, we find that the first few cases are

$$\begin{aligned}
x^0 &= x^{\underline{0}};\\
x^1 &= x^{\underline{1}};\\
x^2 &= x^{\underline{2}} + x^{\underline{1}};\\
x^3 &= x^{\underline{3}} + 3x^{\underline{2}} + x^{\underline{1}};\\
x^4 &= x^{\underline{4}} + 6x^{\underline{3}} + 7x^{\underline{2}} + x^{\underline{1}}.
\end{aligned}$$

These coefficients look suspiciously like the numbers in Table 258, reflected between left and right; therefore we can be pretty confident that the general formula is

$$x^n = \sum_k \left\{{n \atop k}\right\} x^{\underline{k}}, \qquad \text{integer } n \geqslant 0. \tag{6.10}$$

We'd better define $\left\{{n \atop k}\right\} = {n \atop k} = 0$ *when* $k < 0$ *and* $n \geqslant 0$.

And sure enough, a simple proof by induction clinches the argument: We have $x \cdot x^{\underline{k}} = x^{\underline{k+1}} + kx^{\underline{k}}$, because $x^{\underline{k+1}} = x^{\underline{k}}(x-k)$; hence $x \cdot x^{n-1}$ is

$$\begin{aligned}
x\sum_k \left\{{n-1 \atop k}\right\} x^{\underline{k}} &= \sum_k \left\{{n-1 \atop k}\right\} x^{\underline{k+1}} + \sum_k \left\{{n-1 \atop k}\right\} kx^{\underline{k}}\\
&= \sum_k \left\{{n-1 \atop k-1}\right\} x^{\underline{k}} + \sum_k \left\{{n-1 \atop k}\right\} kx^{\underline{k}}\\
&= \sum_k \left(k\left\{{n-1 \atop k}\right\} + \left\{{n-1 \atop k-1}\right\}\right) x^{\underline{k}} = \sum_k \left\{{n \atop k}\right\} x^{\underline{k}}.
\end{aligned}$$

In other words, Stirling subset numbers are the coefficients of factorial powers that yield ordinary powers.

We can go the other way too, because Stirling cycle numbers are the coefficients of ordinary powers that yield factorial powers:

$$\begin{aligned}
x^{\overline{0}} &= x^0\,; \\
x^{\overline{1}} &= x^1\,; \\
x^{\overline{2}} &= x^2 + x^1\,; \\
x^{\overline{3}} &= x^3 + 3x^2 + 2x^1\,; \\
x^{\overline{4}} &= x^4 + 6x^3 + 11x^2 + 6x^1\,.
\end{aligned}$$

We have $(x+n-1)\cdot x^k = x^{k+1} + (n-1)x^k$, so a proof like the one just given shows that

$$(x+n-1)x^{\overline{n-1}} \;=\; (x+n-1)\sum_k \begin{bmatrix} n-1 \\ k \end{bmatrix} x^k \;=\; \sum_k \begin{bmatrix} n \\ k \end{bmatrix} x^k\,.$$

This leads to a proof by induction of the general formula

$$x^{\overline{n}} \;=\; \sum_k \begin{bmatrix} n \\ k \end{bmatrix} x^k\,, \qquad \text{integer } n \geqslant 0. \tag{6.11}$$

(Setting $x = 1$ gives (6.9) again.)

But wait, you say. This equation involves rising factorial powers $x^{\overline{n}}$, while (6.10) involves falling factorials $x^{\underline{n}}$. What if we want to express $x^{\underline{n}}$ in terms of ordinary powers, or if we want to express x^n in terms of rising powers? Easy; we just throw in some minus signs and get

$$x^n \;=\; \sum_k \left\{ n \atop k \right\} (-1)^{n-k} x^{\overline{k}}\,, \qquad \text{integer } n \geqslant 0; \tag{6.12}$$

$$x^{\underline{n}} \;=\; \sum_k \begin{bmatrix} n \\ k \end{bmatrix} (-1)^{n-k} x^k\,, \qquad \text{integer } n \geqslant 0. \tag{6.13}$$

This works because, for example, the formula

$$x^{\underline{4}} \;=\; x(x-1)(x-2)(x-3) \;=\; x^4 - 6x^3 + 11x^2 - 6x$$

is just like the formula

$$x^{\overline{4}} \;=\; x(x+1)(x+2)(x+3) \;=\; x^4 + 6x^3 + 11x^2 + 6x$$

but with alternating signs. The general identity

$$x^{\underline{n}} \;=\; (-1)^n (-x)^{\overline{n}} \tag{6.14}$$

of exercise 2.17 converts (6.10) to (6.12) and (6.11) to (6.13) if we negate x.

Table 264 Basic Stirling number identities, for integer $n \geqslant 0$.

Recurrences:

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = k\begin{Bmatrix} n-1 \\ k \end{Bmatrix} + \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix}.$$

$$\begin{bmatrix} n \\ k \end{bmatrix} = (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}.$$

Special values:

$$\begin{Bmatrix} n \\ 0 \end{Bmatrix} = \begin{bmatrix} n \\ 0 \end{bmatrix} = [n=0].$$

$$\begin{Bmatrix} n \\ 1 \end{Bmatrix} = [n>0]; \qquad \begin{bmatrix} n \\ 1 \end{bmatrix} = (n-1)!\,[n>0].$$

$$\begin{Bmatrix} n \\ 2 \end{Bmatrix} = (2^{n-1}-1)[n>0]; \qquad \begin{bmatrix} n \\ 2 \end{bmatrix} = (n-1)!\,H_{n-1}\,[n>0].$$

$$\begin{Bmatrix} n \\ n-1 \end{Bmatrix} = \begin{bmatrix} n \\ n-1 \end{bmatrix} = \binom{n}{2}.$$

$$\begin{Bmatrix} n \\ n \end{Bmatrix} = \begin{bmatrix} n \\ n \end{bmatrix} = \binom{n}{n} = 1.$$

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = \begin{bmatrix} n \\ k \end{bmatrix} = \binom{n}{k} = 0, \qquad \text{if } k > n.$$

Converting between powers:

$$x^n = \sum_k \begin{Bmatrix} n \\ k \end{Bmatrix} x^{\underline{k}} = \sum_k \begin{Bmatrix} n \\ k \end{Bmatrix} (-1)^{n-k} x^{\overline{k}}.$$

$$x^{\underline{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} (-1)^{n-k} x^k;$$

$$x^{\overline{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} x^k.$$

Inversion formulas:

$$\sum_k \begin{bmatrix} n \\ k \end{bmatrix} \begin{Bmatrix} k \\ m \end{Bmatrix} (-1)^{n-k} = [m=n];$$

$$\sum_k \begin{Bmatrix} n \\ k \end{Bmatrix} \begin{bmatrix} k \\ m \end{bmatrix} (-1)^{n-k} = [m=n].$$

Table 265 Additional Stirling number identities, for integers $l, m, n \geqslant 0$.

$$\left\{ {n+1 \atop m+1} \right\} = \sum_k \binom{n}{k} \left\{ {k \atop m} \right\}. \tag{6.15}$$

$$\left[{n+1 \atop m+1} \right] = \sum_k \left[{n \atop k} \right] \binom{k}{m}. \tag{6.16}$$

$$\left\{ {n \atop m} \right\} = \sum_k \binom{n}{k} \left\{ {k+1 \atop m+1} \right\} (-1)^{n-k}. \tag{6.17}$$

$$\left[{n \atop m} \right] = \sum_k \left[{n+1 \atop k+1} \right] \binom{k}{m} (-1)^{m-k}. \tag{6.18}$$

$$n^m (-1)^{n-m} \binom{n}{m} = \sum_k \binom{n}{k} \binom{-m}{k-m} n^k.$$

$$m! \left\{ {n \atop m} \right\} = \sum_k \binom{m}{k} k^n (-1)^{m-k}. \tag{6.19}$$

$$\left\{ {n+1 \atop m+1} \right\} = \sum_{k=0}^{n} \left\{ {k \atop m} \right\} (m+1)^{n-k}. \tag{6.20}$$

$$\left[{n+1 \atop m+1} \right] = \sum_{k=0}^{n} \left[{k \atop m} \right] n^{\underline{n-k}} = n! \sum_{k=0}^{n} \left[{k \atop m} \right] / k!. \tag{6.21}$$

$$\left\{ {m+n+1 \atop m} \right\} = \sum_{k=0}^{m} k \left\{ {n+k \atop k} \right\}. \tag{6.22}$$

$$\left[{m+n+1 \atop m} \right] = \sum_{k=0}^{m} (n+k) \left[{n+k \atop k} \right]. \tag{6.23}$$

$$\binom{n}{m} = \sum_k \left\{ {n+1 \atop k+1} \right\} \left[{k \atop m} \right] (-1)^{m-k}. \tag{6.24}$$

Also,

$$\left[{n \atop m} \right] (n-1)^{\underline{n-m}} = \sum_k \left[{n \atop k} \right] \left\{ {k \atop m} \right\},$$

a generalization of (6.9).

$$n^{\underline{n-m}} [n \geqslant m] = \sum_k \left[{n+1 \atop k+1} \right] \left\{ {k \atop m} \right\} (-1)^{m-k}. \tag{6.25}$$

$$\left\{ {n \atop n-m} \right\} = \sum_k \binom{m-n}{m+k} \binom{m+n}{n+k} \left[{m+k \atop k} \right]. \tag{6.26}$$

$$\left[{n \atop n-m} \right] = \sum_k \binom{m-n}{m+k} \binom{m+n}{n+k} \left\{ {m+k \atop k} \right\}. \tag{6.27}$$

$$\left\{ {n \atop l+m} \right\} \binom{l+m}{l} = \sum_k \left\{ {k \atop l} \right\} \left\{ {n-k \atop m} \right\} \binom{n}{k}. \tag{6.28}$$

$$\left[{n \atop l+m} \right] \binom{l+m}{l} = \sum_k \left[{k \atop l} \right] \left[{n-k \atop m} \right] \binom{n}{k}. \tag{6.29}$$

We can remember when to stick the $(-1)^{n-k}$ factor into a formula like (6.12) because there's a natural ordering of powers when x is large:

$$x^{\overline{n}} > x^n > x^{\underline{n}}, \qquad \text{for all } x > n > 1. \tag{6.30}$$

The Stirling numbers $\left[{n \atop k}\right]$ and $\left\{{n \atop k}\right\}$ are nonnegative, so we have to use minus signs when expanding a "small" power in terms of "large" ones.

We can plug (6.11) into (6.12) and get a double sum:

$$x^n = \sum_k \left\{{n \atop k}\right\}(-1)^{n-k}x^{\overline{k}} = \sum_{k,m} \left\{{n \atop k}\right\}\left[{k \atop m}\right](-1)^{n-k}x^m .$$

This holds for all x, so the coefficients of x^0, x^1, ..., x^{n-1}, x^{n+1}, x^{n+2}, ... on the right must all be zero and we must have the identity

$$\sum_k \left\{{n \atop k}\right\}\left[{k \atop m}\right](-1)^{n-k} = [m=n], \qquad \text{integers } m, n \geqslant 0. \tag{6.31}$$

Stirling numbers, like binomial coefficients, satisfy many surprising identities. But these identities aren't as versatile as the ones we had in Chapter 5, so they aren't applied nearly as often. Therefore it's best for us just to list the simplest ones, for future reference when a tough Stirling nut needs to be cracked. Tables 264 and 265 contain the formulas that are most frequently useful; the principal identities we have already derived are repeated there.

When we studied binomial coefficients in Chapter 5, we found that it was advantageous to define $\binom{n}{k}$ for negative n in such a way that the identity $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ is valid without any restrictions. Using that identity to extend the $\binom{n}{k}$'s beyond those with combinatorial significance, we discovered (in Table 164) that Pascal's triangle essentially reproduces itself in a rotated form when we extend it upward. Let's try the same thing with Stirling's triangles: What happens if we decide that the basic recurrences

$$\begin{aligned} \left\{{n \atop k}\right\} &= k\left\{{n-1 \atop k}\right\} + \left\{{n-1 \atop k-1}\right\} \\ \left[{n \atop k}\right] &= (n-1)\left[{n-1 \atop k}\right] + \left[{n-1 \atop k-1}\right] \end{aligned}$$

are valid for all integers n and k? The solution becomes unique if we make the reasonable additional stipulations that

$$\left\{{0 \atop k}\right\} = \left[{0 \atop k}\right] = [k=0] \quad \text{and} \quad \left\{{n \atop 0}\right\} = \left[{n \atop 0}\right] = [n=0]. \tag{6.32}$$

Table 267 Stirling's triangles in tandem.

n	$\left\{ {n \atop -5} \right\}$	$\left\{ {n \atop -4} \right\}$	$\left\{ {n \atop -3} \right\}$	$\left\{ {n \atop -2} \right\}$	$\left\{ {n \atop -1} \right\}$	$\left\{ {n \atop 0} \right\}$	$\left\{ {n \atop 1} \right\}$	$\left\{ {n \atop 2} \right\}$	$\left\{ {n \atop 3} \right\}$	$\left\{ {n \atop 4} \right\}$	$\left\{ {n \atop 5} \right\}$
−5	1										
−4	10	1									
−3	35	6	1								
−2	50	11	3	1							
−1	24	6	2	1	1						
0	0	0	0	0	0	1					
1	0	0	0	0	0	0	1				
2	0	0	0	0	0	0	1	1			
3	0	0	0	0	0	0	1	3	1		
4	0	0	0	0	0	0	1	7	6	1	
5	0	0	0	0	0	0	1	15	25	10	1

In fact, a surprisingly pretty pattern emerges: Stirling's triangle for cycles appears above Stirling's triangle for subsets, and vice versa! The two kinds of Stirling numbers are related by an extremely simple law [220, 221]:

$$\left[{n \atop k} \right] = \left\{ {-k \atop -n} \right\}, \qquad \text{integers } k, n. \tag{6.33}$$

We have "duality," something like the relations between min and max, between $\lfloor x \rfloor$ and $\lceil x \rceil$, between $x^{\underline{n}}$ and $x^{\overline{n}}$, between gcd and lcm. It's easy to check that both of the recurrences $\left[{n \atop k} \right] = (n-1)\left[{n-1 \atop k} \right] + \left[{n-1 \atop k-1} \right]$ and $\left\{ {n \atop k} \right\} = k\left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}$ amount to the same thing, under this correspondence.

6.2 EULERIAN NUMBERS

Another triangle of values pops up now and again, this one due to Euler [104, §13; 110, page 485], and we denote its elements by $\left\langle {n \atop k} \right\rangle$. The angle brackets in this case suggest "less than" and "greater than" signs; $\left\langle {n \atop k} \right\rangle$ is the number of permutations $\pi_1\pi_2\ldots\pi_n$ of $\{1, 2, \ldots, n\}$ that have k *ascents*, namely, k places where $\pi_j < \pi_{j+1}$. (Caution: This notation is less standard than our notations $\left[{n \atop k} \right]$, $\left\{ {n \atop k} \right\}$ for Stirling numbers. But we'll see that it makes good sense.)

(Knuth [209, first edition] used $\left\langle {n \atop k+1} \right\rangle$ *for* $\left\langle {n \atop k} \right\rangle$*.)*

For example, eleven permutations of $\{1, 2, 3, 4\}$ have two ascents:

$$\begin{array}{llllll} 1324, & 1423, & 2314, & 2413, & 3412; & \\ 1243, & 1342, & 2341; & 2134, & 3124, & 4123. \end{array}$$

(The first row lists the permutations with $\pi_1 < \pi_2 > \pi_3 < \pi_4$; the second row lists those with $\pi_1 < \pi_2 < \pi_3 > \pi_4$ and $\pi_1 > \pi_2 < \pi_3 < \pi_4$.) Hence $\left\langle {4 \atop 2} \right\rangle = 11$.

Table 268 Euler's triangle.

n	$\left\langle {n \atop 0} \right\rangle$	$\left\langle {n \atop 1} \right\rangle$	$\left\langle {n \atop 2} \right\rangle$	$\left\langle {n \atop 3} \right\rangle$	$\left\langle {n \atop 4} \right\rangle$	$\left\langle {n \atop 5} \right\rangle$	$\left\langle {n \atop 6} \right\rangle$	$\left\langle {n \atop 7} \right\rangle$	$\left\langle {n \atop 8} \right\rangle$	$\left\langle {n \atop 9} \right\rangle$
0	1									
1	1	0								
2	1	1	0							
3	1	4	1	0						
4	1	11	11	1	0					
5	1	26	66	26	1	0				
6	1	57	302	302	57	1	0			
7	1	120	1191	2416	1191	120	1	0		
8	1	247	4293	15619	15619	4293	247	1	0	
9	1	502	14608	88234	156190	88234	14608	502	1	0

Table 268 lists the smallest Eulerian numbers; notice that the trademark sequence is 1, 11, 11, 1 this time. There can be at most $n-1$ ascents, when $n > 0$, so we have $\left\langle {n \atop n} \right\rangle = [n=0]$ on the diagonal of the triangle.

Euler's triangle, like Pascal's, is symmetric between left and right. But in this case the symmetry law is slightly different:

$$\left\langle {n \atop k} \right\rangle = \left\langle {n \atop n-1-k} \right\rangle, \qquad \text{integer } n > 0; \tag{6.34}$$

The permutation $\pi_1\pi_2\ldots\pi_n$ has $n-1-k$ ascents if and only if its "reflection" $\pi_n\ldots\pi_2\pi_1$ has k ascents.

Let's try to find a recurrence for $\left\langle {n \atop k} \right\rangle$. Each permutation $\rho = \rho_1\ldots\rho_{n-1}$ of $\{1,\ldots,n-1\}$ leads to n permutations of $\{1,2,\ldots,n\}$ if we insert the new element n in all possible ways. Suppose we put n in position j, obtaining the permutation $\pi = \rho_1\ldots\rho_{j-1}\,n\,\rho_j\ldots\rho_{n-1}$. The number of ascents in π is the same as the number in ρ, if $j = 1$ or if $\rho_{j-1} < \rho_j$; it's one greater than the number in ρ, if $\rho_{j-1} > \rho_j$ or if $j = n$. Therefore π has k ascents in a total of $(k+1)\left\langle {n-1 \atop k} \right\rangle$ ways from permutations ρ that have k ascents, plus a total of $\big((n-2)-(k-1)+1\big)\left\langle {n-1 \atop k-1} \right\rangle$ ways from permutations ρ that have $k-1$ ascents. The desired recurrence is

$$\left\langle {n \atop k} \right\rangle = (k+1)\left\langle {n-1 \atop k} \right\rangle + (n-k)\left\langle {n-1 \atop k-1} \right\rangle, \quad \text{integer } n > 0. \tag{6.35}$$

Once again we start the recurrence off by setting

$$\left\langle {0 \atop k} \right\rangle = [k=0], \qquad \text{integer } k, \tag{6.36}$$

and we will assume that $\left\langle {n \atop k} \right\rangle = 0$ when $k < 0$.

Eulerian numbers are useful primarily because they provide an unusual connection between ordinary powers and consecutive binomial coefficients:

$$x^n = \sum_k \left\langle {n \atop k} \right\rangle \binom{x+k}{n}, \qquad \text{integer } n \geqslant 0. \tag{6.37}$$

Western scholars have recently learned of a significant Chinese book by Li Shan-Lan [249; 265, pages 320–325], published in 1867, which contains the first known appearance of formula (6.37).

(This is called "Worpitzky's identity" [378].) For example, we have

$$\begin{aligned} x^2 &= \binom{x}{2} + \binom{x+1}{2}, \\ x^3 &= \binom{x}{3} + 4\binom{x+1}{3} + \binom{x+2}{3}, \\ x^4 &= \binom{x}{4} + 11\binom{x+1}{4} + 11\binom{x+2}{4} + \binom{x+3}{4}, \end{aligned}$$

and so on. It's easy to prove (6.37) by induction (exercise 14).

Incidentally, (6.37) gives us yet another way to obtain the sum of the first n squares: We have $k^2 = \left\langle {2 \atop 0} \right\rangle \binom{k}{2} + \left\langle {2 \atop 1} \right\rangle \binom{k+1}{2} = \binom{k}{2} + \binom{k+1}{2}$, hence

$$\begin{aligned} 1^2 + 2^2 + \cdots + n^2 &= \left(\binom{1}{2} + \binom{2}{2} + \cdots + \binom{n}{2}\right) + \left(\binom{2}{2} + \binom{3}{2} + \cdots + \binom{n+1}{2}\right) \\ &= \binom{n+1}{3} + \binom{n+2}{3} = \tfrac{1}{6}(n+1)n\big((n-1) + (n+2)\big). \end{aligned}$$

The Eulerian recurrence (6.35) is a bit more complicated than the Stirling recurrences (6.3) and (6.8), so we don't expect the numbers $\left\langle {n \atop k} \right\rangle$ to satisfy as many simple identities. Still, there are a few:

$$\left\langle {n \atop m} \right\rangle = \sum_{k=0}^{m} \binom{n+1}{k} (m+1-k)^n (-1)^k; \tag{6.38}$$

$$m! \left\{ {n \atop m} \right\} = \sum_k \left\langle {n \atop k} \right\rangle \binom{k}{n-m}; \tag{6.39}$$

$$\left\langle {n \atop m} \right\rangle = \sum_k \left\{ {n \atop k} \right\} \binom{n-k}{m} (-1)^{n-k-m} k!. \tag{6.40}$$

If we multiply (6.39) by z^{n-m} and sum on m, we get $\sum_m \left\{ {n \atop m} \right\} m!\, z^{n-m} = \sum_k \left\langle {n \atop k} \right\rangle (z+1)^k$. Replacing z by $z-1$ and equating coefficients of z^k gives (6.40). Thus the last two of these identities are essentially equivalent. The first identity, (6.38), gives us special values when m is small:

$$\left\langle {n \atop 0} \right\rangle = 1; \quad \left\langle {n \atop 1} \right\rangle = 2^n - n - 1; \quad \left\langle {n \atop 2} \right\rangle = 3^n - (n+1)2^n + \binom{n+1}{2}.$$

Table 270 Second-order Eulerian triangle.

n	$\left\langle\!\left\langle {n \atop 0} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {n \atop 1} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {n \atop 2} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {n \atop 3} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {n \atop 4} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {n \atop 5} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {n \atop 6} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {n \atop 7} \right\rangle\!\right\rangle$	$\left\langle\!\left\langle {n \atop 8} \right\rangle\!\right\rangle$
0	1								
1	1	0							
2	1	2	0						
3	1	8	6	0					
4	1	22	58	24	0				
5	1	52	328	444	120	0			
6	1	114	1452	4400	3708	720	0		
7	1	240	5610	32120	58140	33984	5040	0	
8	1	494	19950	195800	644020	785304	341136	40320	0

We needn't dwell further on Eulerian numbers here; it's usually sufficient simply to know that they exist, and to have a list of basic identities to fall back on when the need arises. However, before we leave this topic, we should take note of yet another triangular pattern of coefficients, shown in Table 270. We call these "second-order Eulerian numbers" $\left\langle\!\left\langle {n \atop k} \right\rangle\!\right\rangle$, because they satisfy a recurrence similar to (6.35) but with n replaced by $2n-1$ in one place:

$$\left\langle\!\left\langle {n \atop k} \right\rangle\!\right\rangle = (k+1)\left\langle\!\left\langle {n-1 \atop k} \right\rangle\!\right\rangle + (2n-1-k)\left\langle\!\left\langle {n-1 \atop k-1} \right\rangle\!\right\rangle . \tag{6.41}$$

These numbers have a curious combinatorial interpretation, first noticed by Gessel and Stanley [147]: If we form permutations of the multiset $\{1,1,2,2,\ldots,n,n\}$ with the special property that all numbers between the two occurrences of m are greater than m, for $1 \leqslant m \leqslant n$, then $\left\langle\!\left\langle {n \atop k} \right\rangle\!\right\rangle$ is the number of such permutations that have k ascents. For example, there are eight suitable single-ascent permutations of $\{1,1,2,2,3,3\}$:

113322, 133221, 221331, 221133, 223311, 233211, 331122, 331221.

Thus $\left\langle\!\left\langle {3 \atop 1} \right\rangle\!\right\rangle = 8$. The multiset $\{1,1,2,2,\ldots,n,n\}$ has a total of

$$\sum_k \left\langle\!\left\langle {n \atop k} \right\rangle\!\right\rangle = (2n-1)(2n-3)\ldots(1) = \frac{(2n)^{\underline{n}}}{2^n} \tag{6.42}$$

suitable permutations, because the two appearances of n must be adjacent and there are $2n-1$ places to insert them within a permutation for $n-1$. For example, when $n=3$ the permutation 1221 has five insertion points, yielding 331221, 133221, 123321, 122331, and 122133. Recurrence (6.41) can be proved by extending the argument we used for ordinary Eulerian numbers.

Second-order Eulerian numbers are important chiefly because of their connection with Stirling numbers [148]: We have, by induction on n,

$$\left\{ {x \atop x-n} \right\} = \sum_k \left\langle\!\!\left\langle {n \atop k} \right\rangle\!\!\right\rangle \binom{x+n-1-k}{2n}, \qquad \text{integer } n \geqslant 0; \tag{6.43}$$

$$\left[{x \atop x-n} \right] = \sum_k \left\langle\!\!\left\langle {n \atop k} \right\rangle\!\!\right\rangle \binom{x+k}{2n}, \qquad \text{integer } n \geqslant 0. \tag{6.44}$$

For example,

$$\begin{aligned}
\left\{ {x \atop x-1} \right\} &= \binom{x}{2}, & \left[{x \atop x-1} \right] &= \binom{x}{2};\\
\left\{ {x \atop x-2} \right\} &= \binom{x+1}{4} + 2\binom{x}{4}, & \left[{x \atop x-2} \right] &= \binom{x}{4} + 2\binom{x+1}{4};\\
\left\{ {x \atop x-3} \right\} &= \binom{x+2}{6} + 8\binom{x+1}{6} + 6\binom{x}{6}, & & \\
& & \left[{x \atop x-3} \right] &= \binom{x}{6} + 8\binom{x+1}{6} + 6\binom{x+2}{6}.
\end{aligned}$$

(We already encountered the case $n = 1$ in (6.7).) These identities hold whenever x is an integer and n is a nonnegative integer. Since the right-hand sides are polynomials in x, we can use (6.43) and (6.44) to define Stirling numbers $\left\{ {x \atop x-n} \right\}$ and $\left[{x \atop x-n} \right]$ for arbitrary real (or complex) values of x.

If $n > 0$, these polynomials $\left\{ {x \atop x-n} \right\}$ and $\left[{x \atop x-n} \right]$ are zero when $x = 0$, $x = 1$, $\ldots$, and $x = n$; therefore they are divisible by $(x-0)$, $(x-1)$, $\ldots$, and $(x-n)$. It's interesting to look at what's left after these known factors are divided out. We define the *Stirling polynomials* $\sigma_n(x)$ by the rule

$$\sigma_n(x) = \left[{x \atop x-n} \right] \Big/ \big(x(x-1)\ldots(x-n)\big). \tag{6.45}$$

(The degree of $\sigma_n(x)$ is $n-1$.) The first few cases are

So $1/x$ is a polynomial?

(Sorry about that.)

$$\begin{aligned}
\sigma_0(x) &= 1/x;\\
\sigma_1(x) &= 1/2;\\
\sigma_2(x) &= (3x-1)/24;\\
\sigma_3(x) &= (x^2-x)/48;\\
\sigma_4(x) &= (15x^3-30x^2+5x+2)/5760.
\end{aligned}$$

They can be computed via the second-order Eulerian numbers; for example,

$$\sigma_3(x) = \big((x-4)(x-5) + 8(x-4)(x+1) + 6(x+2)(x+1)\big)/6!.$$

Table 272 Stirling convolution formulas.

$$rs\sum_{k=0}^{n}\sigma_k(r+tk)\,\sigma_{n-k}(s+t(n-k)) \;=\; (r+s)\,\sigma_n(r+s+tn) \tag{6.46}$$

$$s\sum_{k=0}^{n}k\sigma_k(r+tk)\,\sigma_{n-k}(s+t(n-k)) \;=\; n\sigma_n(r+s+tn) \tag{6.47}$$

$$\left\{{n \atop m}\right\} \;=\; (-1)^{n-m+1}\frac{n!}{(m-1)!}\sigma_{n-m}(-m) \tag{6.48}$$

$$\left[{n \atop m}\right] \;=\; \frac{n!}{(m-1)!}\sigma_{n-m}(n) \tag{6.49}$$

It turns out that these polynomials satisfy two very pretty identities:

$$\left(\frac{ze^z}{e^z-1}\right)^x \;=\; x\sum_{n\geqslant 0}\sigma_n(x)\,z^n\,; \tag{6.50}$$

$$\left(\frac{1}{z}\ln\frac{1}{1-z}\right)^x \;=\; x\sum_{n\geqslant 0}\sigma_n(x+n)\,z^n\,. \tag{6.51}$$

And in general, if $\mathcal{S}_t(z)$ is the power series that satisfies

$$\ln\bigl(1-z\mathcal{S}_t(z)^{t-1}\bigr) \;=\; -z\mathcal{S}_t(z)^t\,, \tag{6.52}$$

then

$$\mathcal{S}_t(z)^x \;=\; x\sum_{n\geqslant 0}\sigma_n(x+tn)\,z^n\,. \tag{6.53}$$

Therefore we can obtain general convolution formulas for Stirling numbers, as we did for binomial coefficients in Table 202; the results appear in Table 272. When a sum of Stirling numbers doesn't fit the identities of Table 264 or 265, Table 272 may be just the ticket. (An example appears later in this chapter, following equation (6.100). Exercise 7.19 discusses the general principles of convolutions based on identities like (6.50) and (6.53).)

6.3 HARMONIC NUMBERS

It's time now to take a closer look at harmonic numbers, which we first met back in Chapter 2:

$$H_n \;=\; 1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n} \;=\; \sum_{k=1}^{n}\frac{1}{k}\,,\quad \text{integer } n\geqslant 0. \tag{6.54}$$

These numbers appear so often in the analysis of algorithms that computer scientists need a special notation for them. We use H_n, the 'H' standing for "harmonic," since a tone of wavelength $1/n$ is called the nth harmonic of a tone whose wavelength is 1. The first few values look like this:

n	0	1	2	3	4	5	6	7	8	9	10
H_n	0	1	$\frac{3}{2}$	$\frac{11}{6}$	$\frac{25}{12}$	$\frac{137}{60}$	$\frac{49}{20}$	$\frac{363}{140}$	$\frac{761}{280}$	$\frac{7129}{2520}$	$\frac{7381}{2520}$

Exercise 21 shows that H_n is never an integer when $n > 1$.

Here's a card trick, based on an idea by R. T. Sharp [325], that illustrates how the harmonic numbers arise naturally in simple situations. Given n cards and a table, we'd like to create the largest possible overhang by stacking the cards up over the table's edge, subject to the laws of gravity:

This must be Table 273.

To define the problem a bit more, we require the edges of the cards to be parallel to the edge of the table; otherwise we could increase the overhang by rotating the cards so that their corners stick out a little farther. And to make the answer simpler, we assume that each card is 2 units long.

With one card, we get maximum overhang when its center of gravity is just above the edge of the table. The center of gravity is in the middle of the card, so we can create half a cardlength, or 1 unit, of overhang.

With two cards, it's not hard to convince ourselves that we get maximum overhang when the center of gravity of the top card is just above the edge of the second card, and the center of gravity of both cards combined is just above the edge of the table. The joint center of gravity of two cards will be in the middle of their common part, so we are able to achieve an additional half unit of overhang.

This pattern suggests a general method, where we place cards so that the center of gravity of the top k cards lies just above the edge of the $k+1$st card (which supports those top k). The table plays the role of the $n+1$st card. To express this condition algebraically, we can let d_k be the distance from the extreme edge of the top card to the corresponding edge of the kth card from the top. Then $d_1 = 0$, and we want to make d_{k+1} the center of gravity of the first k cards:

$$d_{k+1} = \frac{(d_1+1)+(d_2+1)+\cdots+(d_k+1)}{k}, \quad \text{for } 1 \leqslant k \leqslant n. \qquad (6.55)$$

(The center of gravity of k objects, having respective weights $w_1, \ldots, w_k$ and having respective centers of gravity at positions $p_1, \ldots p_k$, is at position $(w_1p_1 + \cdots + w_kp_k)/(w_1 + \cdots + w_k)$.) We can rewrite this recurrence in two equivalent forms

$$\begin{aligned} kd_{k+1} &= k + d_1 + \cdots + d_{k-1} + d_k\,, & k \geqslant 0; \\ (k-1)d_k &= k - 1 + d_1 + \cdots + d_{k-1}\,, & k \geqslant 1. \end{aligned}$$

Subtracting these equations tells us that

$$kd_{k+1} - (k-1)d_k = 1 + d_k\,, \qquad k \geqslant 1;$$

hence $d_{k+1} = d_k + 1/k$. The second card will be offset half a unit past the third, which is a third of a unit past the fourth, and so on. The general formula

$$d_{k+1} = H_k \tag{6.56}$$

follows by induction, and if we set $k = n$ we get $d_{n+1} = H_n$ as the total overhang when n cards are stacked as described.

Could we achieve greater overhang by holding back, not pushing each card to an extreme position but storing up "potential gravitational energy" for a later advance? No; any well-balanced card placement has

$$d_{k+1} \leqslant \frac{(1 + d_1) + (1 + d_2) + \cdots + (1 + d_k)}{k}\,, \qquad 1 \leqslant k \leqslant n.$$

Furthermore $d_1 = 0$. It follows by induction that $d_{k+1} \leqslant H_k$.

Notice that it doesn't take too many cards for the top one to be completely past the edge of the table. We need an overhang of more than one cardlength, which is 2 units. The first harmonic number to exceed 2 is $H_4 = \frac{25}{12}$, so we need only four cards.

And with 52 cards we have an H_{52}-unit overhang, which turns out to be $H_{52}/2 \approx 2.27$ cardlengths. (We will soon learn a formula that tells us how to compute an approximate value of H_n for large n without adding up a whole bunch of fractions.)

Anyone who actually tries to achieve this maximum overhang with 52 cards is probably not dealing with a full deck — or maybe he's a real joker.

An amusing problem called the "worm on the rubber band" shows harmonic numbers in another guise. A slow but persistent worm, W, starts at one end of a meter-long rubber band and crawls one centimeter per minute toward the other end. At the end of each minute, an equally persistent keeper of the band, K, whose sole purpose in life is to frustrate W, stretches it one meter. Thus after one minute of crawling, W is 1 centimeter from the start and 99 from the finish; then K stretches it one meter. During the stretching operation W maintains his relative position, 1% from the start and 99% from

the finish; so W is now 2 cm from the starting point and 198 cm from the goal. After W crawls for another minute the score is 3 cm traveled and 197 to go; but K stretches, and the distances become 4.5 and 295.5. And so on. Does the worm ever reach the finish? He keeps moving, but the goal seems to move away even faster. (We're assuming an infinite longevity for K and W, an infinite elasticity of the band, and an infinitely tiny worm.)

Metric units make this problem more scientific.

Let's write down some formulas. When K stretches the rubber band, the fraction of it that W has crawled stays the same. Thus he crawls 1/100th of it the first minute, 1/200th the second, 1/300th the third, and so on. After n minutes the fraction of the band that he's crawled is

$$\frac{1}{100}\left(\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}\right) = \frac{H_n}{100}\,. \tag{6.57}$$

So he reaches the finish if H_n ever surpasses 100.

We'll see how to estimate H_n for large n soon; for now, let's simply check our analysis by considering how "Superworm" would perform in the same situation. Superworm, unlike W, can crawl 50 cm per minute; so she will crawl $H_n/2$ of the band length after n minutes, according to the argument we just gave. If our reasoning is correct, Superworm should finish before n reaches 4, since $H_4 > 2$. And yes, a simple calculation shows that Superworm has only $33\frac{1}{3}$ cm left to travel after three minutes have elapsed. She finishes in 3 minutes and 40 seconds flat.

A flatworm, eh?

Harmonic numbers appear also in Stirling's triangle. Let's try to find a closed form for $\left[{n \atop 2}\right]$, the number of permutations of n objects that have exactly two cycles. Recurrence (6.8) tells us that

$$\begin{aligned}\left[{n+1 \atop 2}\right] &= n\left[{n \atop 2}\right] + \left[{n \atop 1}\right] \\ &= n\left[{n \atop 2}\right] + (n-1)!\,, \qquad \text{if } n > 0;\end{aligned}$$

and this recurrence is a natural candidate for the summation factor technique of Chapter 2:

$$\frac{1}{n!}\left[{n+1 \atop 2}\right] = \frac{1}{(n-1)!}\left[{n \atop 2}\right] + \frac{1}{n}\,.$$

Unfolding this recurrence tells us that $\frac{1}{n!}\left[{n+1 \atop 2}\right] = H_n$; hence

$$\left[{n+1 \atop 2}\right] = n!H_n\,. \tag{6.58}$$

We proved in Chapter 2 that the harmonic series $\sum_k 1/k$ diverges, which means that H_n gets arbitrarily large as $n \to \infty$. But our proof was indirect;

we found that a certain infinite sum (2.58) gave different answers when it was rearranged, hence $\sum_k 1/k$ could not be bounded. The fact that $H_n \to \infty$ seems counter-intuitive, because it implies among other things that a large enough stack of cards will overhang a table by a mile or more, and that the worm W will eventually reach the end of his rope. Let us therefore take a closer look at the size of H_n when n is large.

The simplest way to see that $H_n \to \infty$ is probably to group its terms according to powers of 2. We put one term into group 1, two terms into group 2, four into group 3, eight into group 4, and so on:

$$\underbrace{\frac{1}{1}}_{\text{group 1}} + \underbrace{\frac{1}{2}+\frac{1}{3}}_{\text{group 2}} + \underbrace{\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}}_{\text{group 3}} + \underbrace{\frac{1}{8}+\frac{1}{9}+\frac{1}{10}+\frac{1}{11}+\frac{1}{12}+\frac{1}{13}+\frac{1}{14}+\frac{1}{15}}_{\text{group 4}} + \cdots .$$

Both terms in group 2 are between $\frac{1}{4}$ and $\frac{1}{2}$, so the sum of that group is between $2 \cdot \frac{1}{4} = \frac{1}{2}$ and $2 \cdot \frac{1}{2} = 1$. All four terms in group 3 are between $\frac{1}{8}$ and $\frac{1}{4}$, so their sum is also between $\frac{1}{2}$ and 1. In fact, each of the 2^{k-1} terms in group k is between 2^{-k} and 2^{1-k}; hence the sum of each individual group is between $\frac{1}{2}$ and 1.

This grouping procedure tells us that if n is in group k, we must have $H_n > k/2$ and $H_n \leqslant k$ (by induction on k). Thus $H_n \to \infty$, and in fact

$$\frac{\lfloor \lg n \rfloor + 1}{2} < H_n \leqslant \lfloor \lg n \rfloor + 1 . \tag{6.59}$$

We now know H_n within a factor of 2. Although the harmonic numbers approach infinity, they approach it only logarithmically — that is, quite slowly.

We should call them the worm numbers, they're so slow.

Better bounds can be found with just a little more work and a dose of calculus. We learned in Chapter 2 that H_n is the discrete analog of the continuous function $\ln n$. The natural logarithm is defined as the area under a curve, so a geometric comparison is suggested:

The area under the curve between 1 and n, which is $\int_1^n dx/x = \ln n$, is less than the area of the n rectangles, which is $\sum_{k=1}^n 1/k = H_n$. Thus $\ln n < H_n$; this is a sharper result than we had in (6.59). And by placing the rectangles

"I now see a way too how y^e aggregate of y^e termes of Musicall progressions may bee found (much after y^e same manner) by Logarithms, but y^e calculations for finding out those rules would bee still more troublesom."
—I. Newton [280]

a little differently, we get a similar upper bound:

This time the area of the n rectangles, H_n, is less than the area of the first rectangle plus the area under the curve. We have proved that

$$\ln n \;<\; H_n \;<\; \ln n + 1\,, \qquad \text{for } n > 1. \tag{6.60}$$

We now know the value of H_n with an error of at most 1.

"Second order" harmonic numbers $H_n^{(2)}$ arise when we sum the squares of the reciprocals, instead of summing simply the reciprocals:

$$H_n^{(2)} \;=\; 1 + \frac{1}{4} + \frac{1}{9} + \cdots + \frac{1}{n^2} \;=\; \sum_{k=1}^{n} \frac{1}{k^2}\,.$$

Similarly, we define harmonic numbers of order r by summing $(-r)$th powers:

$$H_n^{(r)} \;=\; \sum_{k=1}^{n} \frac{1}{k^r}\,. \tag{6.61}$$

If $r > 1$, these numbers approach a limit as $n \to \infty$; we noted in exercise 2.31 that this limit is conventionally called Riemann's zeta function:

$$\zeta(r) \;=\; H_\infty^{(r)} \;=\; \sum_{k\geqslant 1} \frac{1}{k^r}\,. \tag{6.62}$$

Euler [103] discovered a neat way to use generalized harmonic numbers to approximate the ordinary ones, $H_n^{(1)}$. Let's consider the infinite series

$$\ln\!\left(\frac{k}{k-1}\right) \;=\; \frac{1}{k} + \frac{1}{2k^2} + \frac{1}{3k^3} + \frac{1}{4k^4} + \cdots\,, \tag{6.63}$$

which converges when $k > 1$. The left-hand side is $\ln k - \ln(k-1)$; therefore if we sum both sides for $2 \leqslant k \leqslant n$ the left-hand sum telescopes and we get

$$\begin{aligned} \ln n - \ln 1 \;&=\; \sum_{k=2}^{n} \left(\frac{1}{k} + \frac{1}{2k^2} + \frac{1}{3k^3} + \frac{1}{4k^4} + \cdots\right) \\ &=\; \left(H_n - 1\right) + \tfrac{1}{2}\left(H_n^{(2)} - 1\right) + \tfrac{1}{3}\left(H_n^{(3)} - 1\right) + \tfrac{1}{4}\left(H_n^{(4)} - 1\right) + \cdots\,. \end{aligned}$$

Rearranging, we have an expression for the difference between H_n and $\ln n$:

$$H_n - \ln n \;=\; 1 - \tfrac{1}{2}\big(H_n^{(2)}-1\big) - \tfrac{1}{3}\big(H_n^{(3)}-1\big) - \tfrac{1}{4}\big(H_n^{(4)}-1\big) - \cdots .$$

When $n \to \infty$, the right-hand side approaches the limiting value

$$1 - \tfrac{1}{2}\big(\zeta(2)-1\big) - \tfrac{1}{3}\big(\zeta(3)-1\big) - \tfrac{1}{4}\big(\zeta(4)-1\big) - \cdots ,$$

which is now known as *Euler's constant* and conventionally denoted by the Greek letter γ. In fact, $\zeta(r) - 1$ is approximately $1/2^r$, so this infinite series converges rather rapidly and we can compute the decimal value

"Huius igitur quantitatis constantis C valorem detexímus, quippe est $C = 0,577218$."
—L. Euler [103]

$$\gamma \;=\; 0.5772156649\ldots . \tag{6.64}$$

Euler's argument establishes the limiting relation

$$\lim_{n\to\infty}\big(H_n - \ln n\big) \;=\; \gamma ; \tag{6.65}$$

thus H_n lies about 58% of the way between the two extremes in (6.60). We are gradually homing in on its value.

Further refinements are possible, as we will see in Chapter 9. We will prove, for example, that

$$H_n \;=\; \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{\epsilon_n}{120n^4}, \qquad 0 < \epsilon_n < 1. \tag{6.66}$$

This formula allows us to conclude that the millionth harmonic number is

$$H_{1000000} \;\approx\; 14.392726722865723631381127 5 ,$$

without adding up a million fractions. Among other things, this implies that a stack of a million cards can overhang the edge of a table by more than seven cardlengths.

What does (6.66) tell us about the worm on the rubber band? Since H_n is unbounded, the worm will definitely reach the end, when H_n first exceeds 100. Our approximation to H_n says that this will happen when n is approximately

$$e^{100-\gamma} \;\approx\; e^{99.423} .$$

In fact, exercise 9.49 proves that the critical value of n is either $\lfloor e^{100-\gamma} \rfloor$ or $\lceil e^{100-\gamma} \rceil$. We can imagine W's triumph when he crosses the finish line at last, much to K's chagrin, some 287 decillion centuries after his long crawl began. (The rubber band will have stretched to more than 10^{27} light years long; its molecules will be pretty far apart.)

Well, they can't really go at it this long; the world will have ended much earlier, when the Tower of Brahma is fully transferred.

6.4 HARMONIC SUMMATION

Now let's look at some sums involving harmonic numbers, starting with a review of a few ideas we learned in Chapter 2. We proved in (2.36) and (2.57) that

$$\sum_{0\leqslant k<n} H_k = nH_n - n\,; \tag{6.67}$$

$$\sum_{0\leqslant k<n} kH_k = \frac{n(n-1)}{2}H_n - \frac{n(n-1)}{4}\,. \tag{6.68}$$

Let's be bold and take on a more general sum, which includes both of these as special cases: What is the value of

$$\sum_{0\leqslant k<n} \binom{k}{m} H_k\,,$$

when m is a nonnegative integer?

The approach that worked best for (6.67) and (6.68) in Chapter 2 was called *summation by parts*. We wrote the summand in the form $u(k)\Delta v(k)$, and we applied the general identity

$$\sum\nolimits_a^b u(x)\Delta v(x)\,\delta x = u(x)v(x)\Big|_a^b - \sum\nolimits_a^b v(x+1)\Delta u(x)\,\delta x\,. \tag{6.69}$$

Remember? The sum that faces us now, $\sum_{0\leqslant k<n}\binom{k}{m}H_k$, is a natural for this method because we can let

$$u(k) = H_k\,, \qquad \Delta u(k) = H_{k+1} - H_k = \frac{1}{k+1}\,;$$

$$v(k) = \binom{k}{m+1}, \qquad \Delta v(k) = \binom{k+1}{m+1} - \binom{k}{m+1} = \binom{k}{m}.$$

(In other words, harmonic numbers have a simple Δ and binomial coefficients have a simple Δ^{-1}, so we're in business.) Plugging into (6.69) yields

$$\begin{aligned}\sum_{0\leqslant k<n}\binom{k}{m}H_k = \sum\nolimits_0^n \binom{x}{m}H_x\,\delta x &= \binom{x}{m+1}H_x\Big|_0^n - \sum\nolimits_0^n \binom{x+1}{m+1}\frac{\delta x}{x+1}\\ &= \binom{n}{m+1}H_n - \sum_{0\leqslant k<n}\binom{k+1}{m+1}\frac{1}{k+1}\,.\end{aligned}$$

The remaining sum is easy, since we can absorb the $(k+1)^{-1}$ using our old standby, equation (5.5):

$$\sum_{0\leqslant k<n}\binom{k+1}{m+1}\frac{1}{k+1} = \sum_{0\leqslant k<n}\binom{k}{m}\frac{1}{m+1} = \binom{n}{m+1}\frac{1}{m+1}\,.$$

Thus we have the answer we seek:

$$\sum_{0\leqslant k<n}\binom{k}{m}H_k = \binom{n}{m+1}\left(H_n - \frac{1}{m+1}\right). \tag{6.70}$$

(This checks nicely with (6.67) and (6.68) when $m=0$ and $m=1$.)

The next example sum uses division instead of multiplication: Let us try to evaluate

$$S_n = \sum_{k=1}^{n}\frac{H_k}{k}.$$

If we expand H_k by its definition, we obtain a double sum,

$$S_n = \sum_{1\leqslant j\leqslant k\leqslant n}\frac{1}{j\cdot k}.$$

Now another method from Chapter 2 comes to our aid; equation (2.33) tells us that

$$S_n = \frac{1}{2}\left(\left(\sum_{k=1}^{n}\frac{1}{k}\right)^2 + \sum_{k=1}^{n}\frac{1}{k^2}\right) = \frac{1}{2}\left(H_n^2 + H_n^{(2)}\right). \tag{6.71}$$

It turns out that we could also have obtained this answer in another way if we had tried to sum by parts (see exercise 26).

Now let's try our hands at a more difficult problem [354], which doesn't submit to summation by parts:

$$U_n = \sum_{k\geqslant 1}\binom{n}{k}\frac{(-1)^{k-1}}{k}(n-k)^n, \qquad \text{integer } n\geqslant 1.$$

(This sum doesn't explicitly mention harmonic numbers either; but who knows when they might turn up?)

(Not to give the answer away or anything.)

We will solve this problem in two ways, one by grinding out the answer and the other by being clever and/or lucky. First, the grinder's approach. We expand $(n-k)^n$ by the binomial theorem, so that the troublesome k in the denominator will combine with the numerator:

$$\begin{aligned} U_n &= \sum_{k\geqslant 1}\binom{n}{k}\frac{(-1)^{k-1}}{k}\sum_j\binom{n}{j}(-k)^j n^{n-j} \\ &= \sum_j\binom{n}{j}(-1)^{j-1}n^{n-j}\sum_{k\geqslant 1}\binom{n}{k}(-1)^k k^{j-1}. \end{aligned}$$

This isn't quite the mess it seems, because the k^{j-1} in the inner sum is a polynomial in k, and identity (5.40) tells us that we are simply taking the

nth difference of this polynomial. Almost; first we must clean up a few things. For one, k^{j-1} isn't a polynomial if $j = 0$; so we will need to split off that term and handle it separately. For another, we're missing the term $k = 0$ from the formula for nth difference; that term is nonzero when $j = 1$, so we had better restore it (and subtract it out again). The result is

$$\begin{aligned} U_n &= \sum_{j\geqslant 1}\binom{n}{j}(-1)^{j-1}n^{n-j}\sum_{k\geqslant 0}\binom{n}{k}(-1)^k k^{j-1} \\ &\quad -\sum_{j\geqslant 1}\binom{n}{j}(-1)^{j-1}n^{n-j}\binom{n}{0}0^{j-1} \\ &\quad -\binom{n}{0}n^n\sum_{k\geqslant 1}\binom{n}{k}(-1)^k k^{-1}\,. \end{aligned}$$

OK, now the top line (the only remaining double sum) is zero: It's the sum of multiples of nth differences of polynomials of degree less than n, and such nth differences are zero. The second line is zero except when $j = 1$, when it equals $-n^n$. So the third line is the only residual difficulty; we have reduced the original problem to a much simpler sum:

$$U_n = n^n(T_n - 1)\,, \qquad \text{where } T_n = \sum_{k\geqslant 1}\binom{n}{k}\frac{(-1)^{k-1}}{k}\,. \tag{6.72}$$

For example, $U_3 = \binom{3}{1}\frac{8}{1} - \binom{3}{2}\frac{1}{2} = \frac{45}{2}$; $T_3 = \binom{3}{1}\frac{1}{1} - \binom{3}{2}\frac{1}{2} + \binom{3}{3}\frac{1}{3} = \frac{11}{6}$; hence $U_3 = 27(T_3 - 1)$ as claimed.

How can we evaluate T_n? One way is to replace $\binom{n}{k}$ by $\binom{n-1}{k} + \binom{n-1}{k-1}$, obtaining a simple recurrence for T_n in terms of T_{n-1}. But there's a more instructive way: We had a similar formula in (5.41), namely

$$\sum_k \binom{n}{k}\frac{(-1)^k}{x+k} = \frac{n!}{x(x+1)\ldots(x+n)}\,.$$

If we subtract out the term for $k = 0$ and set $x = 0$, we get $-T_n$. So let's do it:

$$\begin{aligned} T_n &= \left(\frac{1}{x} - \frac{n!}{x(x+1)\ldots(x+n)}\right)\Bigg|_{x=0} \\ &= \left(\frac{(x+1)\ldots(x+n) - n!}{x(x+1)\ldots(x+n)}\right)\Bigg|_{x=0} \\ &= \left(\frac{x^n\left[{n+1 \atop n+1}\right] + \cdots + x\left[{n+1 \atop 2}\right] + \left[{n+1 \atop 1}\right] - n!}{x(x+1)\ldots(x+n)}\right)\Bigg|_{x=0} = \frac{1}{n!}\left[{n+1 \atop 2}\right]. \end{aligned}$$

(We have used the expansion (6.11) of $(x+1)\ldots(x+n) = x^{\overline{n+1}}/x$; we can divide x out of the numerator because $\left[{n+1 \atop 1}\right] = n!$.) But we know from (6.58) that $\left[{n+1 \atop 2}\right] = n!\,H_n$; hence $T_n = H_n$, and we have the answer:

$$U_n \;=\; n^n(H_n - 1)\,. \tag{6.73}$$

That's one approach. The other approach will be to try to evaluate a much more general sum,

$$U_n(x,y) \;=\; \sum_{k\geqslant 1}\binom{n}{k}\frac{(-1)^{k-1}}{k}(x+ky)^n\,, \qquad \text{integer } n \geqslant 0; \tag{6.74}$$

the value of the original U_n will drop out as the special case $U_n(n,-1)$. (We are encouraged to try for more generality because the previous derivation "threw away" most of the details of the given problem; somehow those details must be irrelevant, because the nth difference wiped them away.)

We could replay the previous derivation with small changes and discover the value of $U_n(x,y)$. Or we could replace $(x+ky)^n$ by $(x+ky)^{n-1}(x+ky)$ and then replace $\binom{n}{k}$ by $\binom{n-1}{k} + \binom{n-1}{k-1}$, leading to the recurrence

$$U_n(x,y) \;=\; xU_{n-1}(x,y) + x^n/n + yx^{n-1}\,; \tag{6.75}$$

this can readily be solved with a summation factor (exercise 5).

But it's easiest to use another trick that worked to our advantage in Chapter 2: differentiation. The derivative of $U_n(x,y)$ with respect to y brings out a k that cancels with the k in the denominator, and the resulting sum is trivial:

$$\begin{aligned}\frac{\partial}{\partial y}U_n(x,y) \;&=\; \sum_{k\geqslant 1}\binom{n}{k}(-1)^{k-1}n(x+ky)^{n-1}\\ &=\; \binom{n}{0}nx^{n-1} - \sum_{k\geqslant 0}\binom{n}{k}(-1)^k n(x+ky)^{n-1} \;=\; nx^{n-1}\,.\end{aligned}$$

(Once again, the nth difference of a polynomial of degree $< n$ has vanished.)

We've proved that the derivative of $U_n(x,y)$ with respect to y is nx^{n-1}, independent of y. In general, if $f'(y) = c$ then $f(y) = f(0) + cy$; therefore we must have $U_n(x,y) = U_n(x,0) + nx^{n-1}y$.

The remaining task is to determine $U_n(x,0)$. But $U_n(x,0)$ is just x^n times the sum $T_n = H_n$ we've already considered in (6.72); therefore the general sum in (6.74) has the closed form

$$U_n(x,y) \;=\; x^nH_n + nx^{n-1}y\,. \tag{6.76}$$

In particular, the solution to the original problem is $U_n(n,-1) = n^n(H_n - 1)$.

6.5 BERNOULLI NUMBERS

The next important sequence of numbers on our agenda is named after Jakob Bernoulli (1654–1705), who discovered curious relationships while working out the formulas for sums of mth powers [26]. Let's write

$$S_m(n) \;=\; 0^m + 1^m + \cdots + (n-1)^m \;=\; \sum_{k=0}^{n-1} k^m \;=\; \sum\nolimits_0^n x^m\,\delta x\,. \qquad (6.77)$$

(Thus, when $m > 0$ we have $S_m(n) = H_{n-1}^{(-m)}$ in the notation of generalized harmonic numbers.) Bernoulli looked at the following sequence of formulas and spotted a pattern:

$$\begin{aligned}
S_0(n) &= n \\
S_1(n) &= \tfrac{1}{2}n^2 - \tfrac{1}{2}n \\
S_2(n) &= \tfrac{1}{3}n^3 - \tfrac{1}{2}n^2 + \tfrac{1}{6}n \\
S_3(n) &= \tfrac{1}{4}n^4 - \tfrac{1}{2}n^3 + \tfrac{1}{4}n^2 \\
S_4(n) &= \tfrac{1}{5}n^5 - \tfrac{1}{2}n^4 + \tfrac{1}{3}n^3 - \tfrac{1}{30}n \\
S_5(n) &= \tfrac{1}{6}n^6 - \tfrac{1}{2}n^5 + \tfrac{5}{12}n^4 - \tfrac{1}{12}n^2 \\
S_6(n) &= \tfrac{1}{7}n^7 - \tfrac{1}{2}n^6 + \tfrac{1}{2}n^5 - \tfrac{1}{6}n^3 + \tfrac{1}{42}n \\
S_7(n) &= \tfrac{1}{8}n^8 - \tfrac{1}{2}n^7 + \tfrac{7}{12}n^6 - \tfrac{7}{24}n^4 + \tfrac{1}{12}n^2 \\
S_8(n) &= \tfrac{1}{9}n^9 - \tfrac{1}{2}n^8 + \tfrac{2}{3}n^7 - \tfrac{7}{15}n^5 + \tfrac{2}{9}n^3 - \tfrac{1}{30}n \\
S_9(n) &= \tfrac{1}{10}n^{10} - \tfrac{1}{2}n^9 + \tfrac{3}{4}n^8 - \tfrac{7}{10}n^6 + \tfrac{1}{2}n^4 - \tfrac{3}{20}n^2 \\
S_{10}(n) &= \tfrac{1}{11}n^{11} - \tfrac{1}{2}n^{10} + \tfrac{5}{6}n^9 - n^7 + n^5 - \tfrac{1}{2}n^3 + \tfrac{5}{66}n
\end{aligned}$$

Can you see it too? The coefficient of n^{m+1} in $S_m(n)$ is always $1/(m+1)$. The coefficient of n^m is always $-1/2$. The coefficient of n^{m-1} is always ... let's see ... $m/12$. The coefficient of n^{m-2} is always zero. The coefficient of n^{m-3} is always ... let's see ... hmmm ... yes, it's $-m(m-1)(m-2)/720$. The coefficient of n^{m-4} is always zero. And it looks as if the pattern will continue, with the coefficient of n^{m-k} always being some constant times $m^{\underline{k}}$.

That was Bernoulli's empirical discovery. (He did not give a proof.) In modern notation we write the coefficients in the form

$$\begin{aligned}
S_m(n) &= \frac{1}{m+1}\left(B_0\,n^{m+1} + \binom{m+1}{1}B_1\,n^m + \cdots + \binom{m+1}{m}B_m\,n\right) \\
&= \frac{1}{m+1}\sum_{k=0}^{m}\binom{m+1}{k}B_k\,n^{m+1-k}\,. \qquad (6.78)
\end{aligned}$$

Bernoulli numbers are defined by an implicit recurrence relation,

$$\sum_{j=0}^{m} \binom{m+1}{j} B_j \;=\; [m=0]\,, \qquad \text{for all } m \geqslant 0. \tag{6.79}$$

For example, $\binom{2}{0}B_0 + \binom{2}{1}B_1 = 0$. The first few values turn out to be

n	0	1	2	3	4	5	6	7	8	9	10	11	12
B_n	1	$-\frac{1}{2}$	$\frac{1}{6}$	0	$\frac{-1}{30}$	0	$\frac{1}{42}$	0	$\frac{-1}{30}$	0	$\frac{5}{66}$	0	$\frac{-691}{2730}$

(All conjectures about a simple closed form for B_n are wiped out by the appearance of the strange fraction $-691/2730$.)

We can prove Bernoulli's formula (6.78) by induction on m, using the perturbation method (one of the ways we found $S_2(n) = \square_n$ in Chapter 2):

$$\begin{aligned} S_{m+1}(n) + n^{m+1} \;&=\; \sum_{k=0}^{n-1} (k+1)^{m+1} \\ &=\; \sum_{k=0}^{n-1} \sum_{j=0}^{m+1} \binom{m+1}{j} k^j \;=\; \sum_{j=0}^{m+1} \binom{m+1}{j} S_j(n)\,. \end{aligned} \tag{6.80}$$

Let $\widehat{S}_m(n)$ be the right-hand side of (6.78); we wish to show that $S_m(n) = \widehat{S}_m(n)$, assuming that $S_j(n) = \widehat{S}_j(n)$ for $0 \leqslant j < m$. We begin as we did for $m = 2$ in Chapter 2, subtracting $S_{m+1}(n)$ from both sides of (6.80). Then we expand each $S_j(n)$ using (6.78), and regroup so that the coefficients of powers of n on the right-hand side are brought together and simplified:

$$\begin{aligned} n^{m+1} \;&=\; \sum_{j=0}^{m} \binom{m+1}{j} S_j(n) \;=\; \sum_{j=0}^{m} \binom{m+1}{j} \widehat{S}_j(n) + \binom{m+1}{m} \Delta \\ &=\; \sum_{j=0}^{m} \binom{m+1}{j} \frac{1}{j+1} \sum_{k=0}^{j} \binom{j+1}{k} B_k n^{j+1-k} + (m+1)\,\Delta \\ &=\; \sum_{0 \leqslant k \leqslant j \leqslant m} \binom{m+1}{j} \binom{j+1}{k} \frac{B_k}{j+1} n^{j+1-k} + (m+1)\,\Delta \\ &=\; \sum_{0 \leqslant k \leqslant j \leqslant m} \binom{m+1}{j} \binom{j+1}{j-k} \frac{B_{j-k}}{j+1} n^{k+1} + (m+1)\,\Delta \\ &=\; \sum_{0 \leqslant k \leqslant j \leqslant m} \binom{m+1}{j} \binom{j+1}{k+1} \frac{B_{j-k}}{j+1} n^{k+1} + (m+1)\,\Delta \\ &=\; \sum_{0 \leqslant k \leqslant m} \frac{n^{k+1}}{k+1} \sum_{k \leqslant j \leqslant m} \binom{m+1}{j} \binom{j}{k} B_{j-k} + (m+1)\,\Delta \end{aligned}$$

$$
\begin{aligned}
&= \sum_{0\leqslant k\leqslant m} \frac{n^{k+1}}{k+1}\binom{m+1}{k} \sum_{k\leqslant j\leqslant m} \binom{m+1-k}{j-k} B_{j-k} + (m+1)\,\Delta \\
&= \sum_{0\leqslant k\leqslant m} \frac{n^{k+1}}{k+1}\binom{m+1}{k} \sum_{0\leqslant j\leqslant m-k} \binom{m+1-k}{j} B_j + (m+1)\,\Delta \\
&= \sum_{0\leqslant k\leqslant m} \frac{n^{k+1}}{k+1}\binom{m+1}{k} [m-k=0] + (m+1)\,\Delta \\
&= \frac{n^{m+1}}{m+1}\binom{m+1}{m} + (m+1)\,\Delta \\
&= n^{m+1} + (m+1)\,\Delta\,, \qquad \text{where } \Delta = S_m(n) - \widehat{S}_m(n).
\end{aligned}
$$

(This derivation is a good review of the standard manipulations we learned in Chapter 5.) Thus $\Delta = 0$ and $S_m(n) = \widehat{S}_m(n)$, QED.

Here's some more neat stuff that you'll probably want to skim through the first time.
—Friendly TA

In Chapter 7 we'll use generating functions to obtain a much simpler proof of (6.78). The key idea will be to show that the Bernoulli numbers are the coefficients of the power series

$$\frac{z}{e^z-1} = \sum_{n\geqslant 0} B_n \frac{z^n}{n!}\,. \tag{6.81}$$

Start Skimming

Let's simply assume for now that equation (6.81) holds, so that we can derive some of its amazing consequences. If we add $\frac{1}{2}z$ to both sides, thereby cancelling the term $B_1 z/1! = -\frac{1}{2}z$ from the right, we get

$$\frac{z}{e^z-1} + \frac{z}{2} = \frac{z}{2}\,\frac{e^z+1}{e^z-1} = \frac{z}{2}\,\frac{e^{z/2}+e^{-z/2}}{e^{z/2}-e^{-z/2}} = \frac{z}{2}\coth\frac{z}{2}\,. \tag{6.82}$$

Here coth is the "hyperbolic cotangent" function, otherwise known in calculus books as $\cosh z/\sinh z$; we have

$$\sinh z = \frac{e^z-e^{-z}}{2}\,; \qquad \cosh z = \frac{e^z+e^{-z}}{2}\,. \tag{6.83}$$

Changing z to $-z$ gives $\left(\frac{-z}{2}\right)\coth\left(\frac{-z}{2}\right) = \frac{z}{2}\coth\frac{z}{2}$; hence every odd-numbered coefficient of $\frac{z}{2}\coth\frac{z}{2}$ must be zero, and we have

$$B_3 = B_5 = B_7 = B_9 = B_{11} = B_{13} = \cdots = 0\,. \tag{6.84}$$

Furthermore (6.82) leads to a closed form for the coefficients of coth:

$$z\coth z = \frac{2z}{e^{2z}-1} + \frac{2z}{2} = \sum_{n\geqslant 0} B_{2n}\frac{(2z)^{2n}}{(2n)!} = \sum_{n\geqslant 0} 4^n B_{2n}\frac{z^{2n}}{(2n)!}\,. \tag{6.85}$$

But there isn't much of a market for hyperbolic functions; people are more interested in the "real" functions of trigonometry. We can express ordinary

trigonometric functions in terms of their hyperbolic cousins by using the rules

$$\sin z = -i\sinh iz\,, \qquad \cos z = \cosh iz\,; \tag{6.86}$$

the corresponding power series are

$$\begin{aligned}
\sin z &= \frac{z^1}{1!} - \frac{z^3}{3!} + \frac{z^5}{5!} - \cdots\,, & \sinh z &= \frac{z^1}{1!} + \frac{z^3}{3!} + \frac{z^5}{5!} + \cdots\,;\\
\cos z &= \frac{z^0}{0!} - \frac{z^2}{2!} + \frac{z^4}{4!} - \cdots\,, & \cosh z &= \frac{z^0}{0!} + \frac{z^2}{2!} + \frac{z^4}{4!} + \cdots\,.
\end{aligned}$$

Hence $\cot z = \cos z/\sin z = i\cosh iz/\sinh iz = i\coth iz$, and we have

I see, we get "real" functions by using imaginary numbers.

$$z\cot z = \sum_{n\geqslant 0} B_{2n}\frac{(2iz)^{2n}}{(2n)!} = \sum_{n\geqslant 0}(-4)^n B_{2n}\frac{z^{2n}}{(2n)!}\,. \tag{6.87}$$

Another remarkable formula for $z\cot z$ was found by Euler (exercise 73):

$$z\cot z = 1 - 2\sum_{k\geqslant 1}\frac{z^2}{k^2\pi^2 - z^2}\,. \tag{6.88}$$

We can expand Euler's formula in powers of z^2, obtaining

$$\begin{aligned}
z\cot z &= 1 - 2\sum_{k\geqslant 1}\left(\frac{z^2}{k^2\pi^2} + \frac{z^4}{k^4\pi^4} + \frac{z^6}{k^6\pi^6} + \cdots\right)\\
&= 1 - 2\left(\frac{z^2}{\pi^2}H_\infty^{(2)} + \frac{z^4}{\pi^4}H_\infty^{(4)} + \frac{z^6}{\pi^6}H_\infty^{(6)} + \cdots\right).
\end{aligned}$$

Equating coefficients of z^{2n} with those in our other formula, (6.87), gives us an almost miraculous closed form for infinitely many infinite sums:

$$\zeta(2n) = H_\infty^{(2n)} = (-1)^{n-1}\frac{2^{2n-1}\pi^{2n}B_{2n}}{(2n)!}\,, \qquad \text{integer } n > 0. \tag{6.89}$$

For example,

$$\zeta(2) = H_\infty^{(2)} = 1 + \tfrac{1}{4} + \tfrac{1}{9} + \cdots = \pi^2 B_2 = \pi^2/6\,; \tag{6.90}$$

$$\zeta(4) = H_\infty^{(4)} = 1 + \tfrac{1}{16} + \tfrac{1}{81} + \cdots = -\pi^4 B_4/3 = \pi^4/90\,. \tag{6.91}$$

Formula (6.89) is not only a closed form for $H_\infty^{(2n)}$, it also tells us the approximate size of B_{2n}, since $H_\infty^{(2n)}$ is very near 1 when n is large. And it tells us that $(-1)^{n-1}B_{2n} > 0$ for all $n > 0$; thus the nonzero Bernoulli numbers alternate in sign.

And that's not all. Bernoulli numbers also appear in the coefficients of the tangent function,

Start Skipping

$$\tan z = \frac{\sin z}{\cos z} = \sum_{n\geqslant 0}(-1)^{n-1}4^n(4^n-1)B_{2n}\frac{z^{2n-1}}{(2n)!}, \tag{6.92}$$

as well as other trigonometric functions (exercise 72). Formula (6.92) leads to another important fact about the Bernoulli numbers, namely that

$$T_{2n-1} = (-1)^{n-1}\frac{4^n(4^n-1)}{2n}B_{2n} \quad \text{is a positive integer.} \tag{6.93}$$

We have, for example:

n	1	3	5	7	9	11	13
T_n	1	2	16	272	7936	353792	22368256

(The T's are called *tangent numbers.*)

One way to prove (6.93), following an idea of B. F. Logan, is to consider the power series

$$\begin{aligned}\frac{\sin z + x\cos z}{\cos z - x\sin z} &= x + (1+x^2)z + (2x^3+2x)\frac{z^2}{2} + (6x^4+8x^2+2)\frac{z^3}{6} + \cdots \\ &= \sum_{n\geqslant 0} T_n(x)\frac{z^n}{n!},\end{aligned} \tag{6.94}$$

When $x = \tan w$, this is $\tan(z+w)$. Hence, by Taylor's theorem, the nth derivative of $\tan w$ is $T_n(\tan w)$.

where $T_n(x)$ is a polynomial in x; setting $x = 0$ gives $T_n(0) = T_n$, the nth tangent number. If we differentiate (6.94) with respect to x, we get

$$\frac{1}{(\cos z - x\sin z)^2} = \sum_{n\geqslant 0} T_n'(x)\frac{z^n}{n!};$$

but if we differentiate with respect to z, we get

$$\frac{1+x^2}{(\cos z - x\sin z)^2} = \sum_{n\geqslant 1} T_n(x)\frac{z^{n-1}}{(n-1)!} = \sum_{n\geqslant 0} T_{n+1}(x)\frac{z^n}{n!}.$$

(Try it—the cancellation is very pretty.) Therefore we have

$$T_{n+1}(x) = (1+x^2)T_n'(x), \qquad T_0(x) = x, \tag{6.95}$$

a simple recurrence from which it follows that the coefficients of $T_n(x)$ are nonnegative integers. Moreover, we can easily prove that $T_n(x)$ has degree $n+1$, and that its coefficients are alternately zero and positive. Therefore $T_{2n+1}(0) = T_{2n+1}$ is a positive integer, as claimed in (6.93).

Recurrence (6.95) gives us a simple way to calculate Bernoulli numbers, via tangent numbers, using only simple operations on integers; by contrast, the defining recurrence (6.79) involves difficult arithmetic with fractions.

If we want to compute the sum of nth powers from a to $b-1$ instead of from 0 to $n-1$, the theory of Chapter 2 tells us that

$$\sum_{k=a}^{b-1} k^m \;=\; \sum\nolimits_a^b x^m\,\delta x \;=\; S_m(b) - S_m(a)\,. \tag{6.96}$$

This identity has interesting consequences when we consider negative values of k: We have

$$\sum_{k=-n+1}^{-1} k^m \;=\; (-1)^m \sum_{k=0}^{n-1} k^m\,, \qquad \text{when } m > 0,$$

hence

$$S_m(0) - S_m(-n+1) \;=\; (-1)^m\big(S_m(n) - S_m(0)\big)\,.$$

But $S_m(0) = 0$, so we have the identity

$$S_m(1-n) \;=\; (-1)^{m+1} S_m(n)\,, \qquad m > 0. \tag{6.97}$$

Johann Faulhaber implicitly used (6.97) in 1635 [119] to find simple formulas for $S_m(n)$ as polynomials in $n(n+1)/2$ when $m \leqslant 17$; see [222].)

Therefore $S_m(1) = 0$. If we write the polynomial $S_m(n)$ in factored form, it will always have the factors n and $(n-1)$, because it has the roots 0 and 1. In general, $S_m(n)$ is a polynomial of degree $m+1$ with leading term $\frac{1}{m+1}n^{m+1}$. Moreover, we can set $n = \frac{1}{2}$ in (6.97) to deduce that $S_m(\frac{1}{2}) = (-1)^{m+1}S_m(\frac{1}{2})$; if m is even, this makes $S_m(\frac{1}{2}) = 0$, so $(n - \frac{1}{2})$ will be an additional factor. These observations explain why we found the simple factorization

$$S_2(n) \;=\; \tfrac{1}{3}n(n-\tfrac{1}{2})(n-1)$$

in Chapter 2; we could have used such reasoning to deduce the value of $S_2(n)$ without calculating it! Furthermore, (6.97) implies that the polynomial with the remaining factors, $\hat{S}_m(n) = S_m(n)/(n-\frac{1}{2})$, always satisfies

$$\hat{S}_m(1-n) \;=\; \hat{S}_m(n)\,, \qquad m \text{ even}, \quad m > 0.$$

It follows that $S_m(n)$ can always be written in the factored form

$$S_m(n) \;=\; \begin{cases} \dfrac{1}{m+1} \displaystyle\prod_{k=1}^{\lceil m/2 \rceil} (n - \tfrac{1}{2} - \alpha_k)(n - \tfrac{1}{2} + \alpha_k)\,, & m \text{ odd;} \\[2ex] \dfrac{(n-\frac{1}{2})}{m+1} \displaystyle\prod_{k=1}^{m/2} (n - \tfrac{1}{2} - \alpha_k)(n - \tfrac{1}{2} + \alpha_k)\,, & m \text{ even.} \end{cases} \tag{6.98}$$

Here $\alpha_1 = \frac{1}{2}$, and $\alpha_2, \ldots, \alpha_{\lceil m/2 \rceil}$ are appropriate complex numbers whose values depend on m. For example,

$$\begin{aligned}
S_3(n) &= n^2(n-1)^2/4\,; \\
S_4(n) &= n(n-\tfrac{1}{2})(n-1)(n-\tfrac{1}{2}+\sqrt{7/12}\,)(n-\tfrac{1}{2}-\sqrt{7/12}\,)/5\,; \\
S_5(n) &= n^2(n-1)^2(n-\tfrac{1}{2}+\sqrt{3/4}\,)(n-\tfrac{1}{2}-\sqrt{3/4}\,)/6\,; \\
S_6(n) &= n(n-\tfrac{1}{2})(n-1)(n-\tfrac{1}{2}+\alpha)(n-\tfrac{1}{2}-\alpha)(n-\tfrac{1}{2}+\overline{\alpha})(n-\tfrac{1}{2}-\overline{\alpha})\,, \\
&\qquad \text{where } \alpha = 2^{-5/2}3^{-1/2}31^{1/4}\bigl(\sqrt{\sqrt{31}+\sqrt{27}}+i\sqrt{\sqrt{31}-\sqrt{27}}\bigr).
\end{aligned}$$

Stop Skipping

If m is odd and greater than 1, we have $B_m = 0$; hence $S_m(n)$ is divisible by n^2 (and by $(n-1)^2$). Otherwise the roots of $S_m(n)$ don't seem to obey a simple law.

Let's conclude our study of Bernoulli numbers by looking at how they relate to Stirling numbers. One way to compute $S_m(n)$ is to change ordinary powers to falling powers, since the falling powers have easy sums. After doing those easy sums we can convert back to ordinary powers:

$$\begin{aligned}
S_m(n) = \sum_{k=0}^{n-1} k^m &= \sum_{k=0}^{n-1}\sum_{j\geqslant 0}\left\{{m \atop j}\right\}k^{\underline{j}} = \sum_{j\geqslant 0}\left\{{m \atop j}\right\}\sum_{k=0}^{n-1} k^{\underline{j}} \\
&= \sum_{j\geqslant 0}\left\{{m \atop j}\right\}\frac{n^{\underline{j+1}}}{j+1} \\
&= \sum_{j\geqslant 0}\left\{{m \atop j}\right\}\frac{1}{j+1}\sum_{k\geqslant 0}(-1)^{j+1-k}\left[{j+1 \atop k}\right]n^k\,.
\end{aligned}$$

Therefore, equating coefficients with those in (6.78), we must have the identity

$$\sum_{j\geqslant 0}\left\{{m \atop j}\right\}\left[{j+1 \atop k}\right]\frac{(-1)^{j+1-k}}{j+1} = \frac{1}{m+1}\binom{m+1}{k}B_{m+1-k}\,. \tag{6.99}$$

It would be nice to prove this relation directly, thereby discovering Bernoulli numbers in a new way. But the identities in Tables 264 or 265 don't give us any obvious handle on a proof by induction that the left-hand sum in (6.99) is a constant times $m^{\underline{k-1}}$. If $k = m+1$, the left-hand sum is just $\left\{{m \atop m}\right\}\left[{m+1 \atop m+1}\right]/(m+1) = 1/(m+1)$, so that case is easy. And if $k = m$, the left-hand side sums to $\left\{{m \atop m-1}\right\}\left[{m \atop m}\right]m^{-1} - \left\{{m \atop m}\right\}\left[{m+1 \atop m}\right](m+1)^{-1} = \frac{1}{2}(m-1) - \frac{1}{2}m = -\frac{1}{2}$; so that case is pretty easy too. But if $k < m$, the left-hand sum looks hairy. Bernoulli would probably not have discovered his numbers if he had taken this route.

One thing we can do is replace $\left\{ {m \atop j} \right\}$ by $\left\{ {m+1 \atop j+1} \right\} - (j+1)\left\{ {m \atop j+1} \right\}$. The $(j+1)$ nicely cancels with the awkward denominator, and the left-hand side becomes

$$\sum_{j\geqslant 0} \left\{ {m+1 \atop j+1} \right\} \left[{j+1 \atop k} \right] \frac{(-1)^{j+1-k}}{j+1} \quad - \quad \sum_{j\geqslant 0} \left\{ {m \atop j+1} \right\} \left[{j+1 \atop k} \right] (-1)^{j+1-k} .$$

The second sum is zero, when $k < m$, by (6.31). That leaves us with the first sum, which cries out for a change in notation; let's rename all variables so that the index of summation is k, and so that the other parameters are m and n. Then identity (6.99) is equivalent to

$$\sum_{k} \left\{ {n \atop k} \right\} \left[{k \atop m} \right] \frac{(-1)^{k-m}}{k} \;=\; \frac{1}{n}\binom{n}{m} B_{n-m} \;+\; [m = n-1] . \tag{6.100}$$

Good, we have something that looks more pleasant — although Table 265 still doesn't suggest any obvious next step.

The convolution formulas in Table 272 now come to the rescue. We can use (6.49) and (6.48) to rewrite the summand in terms of Stirling polynomials:

$$\begin{aligned} \left\{ {n \atop k} \right\} \left[{k \atop m} \right] &= (-1)^{n-k+1} \frac{n!}{(k-1)!} \sigma_{n-k}(-k) \cdot \frac{k!}{(m-1)!} \sigma_{k-m}(k) ; \\ \left\{ {n \atop k} \right\} \left[{k \atop m} \right] \frac{(-1)^{k-m}}{k} &= (-1)^{n+1-m} \frac{n!}{(m-1)!} \sigma_{n-k}(-k)\, \sigma_{k-m}(k) . \end{aligned}$$

Things are looking up; the convolution in (6.46), with $t = 1$, yields

$$\begin{aligned} \sum_{k=0}^{n} \sigma_{n-k}(-k)\, \sigma_{k-m}(k) &= \sum_{k=0}^{n-m} \sigma_{n-m-k}\big(-n + (n-m-k)\big)\, \sigma_k(m+k) \\ &= \frac{m-n}{(m)(-n)} \sigma_{n-m}\big(m - n + (n-m)\big) . \end{aligned}$$

Formula (6.100) is now verified, and we find that Bernoulli numbers are related to the constant terms in the Stirling polynomials:

Stop Skimming

$$\frac{B_m}{m!} \;=\; -m\sigma_m(0) . \tag{6.101}$$

6.6 FIBONACCI NUMBERS

Now we come to a special sequence of numbers that is perhaps the most pleasant of all, the Fibonacci sequence $\langle F_n \rangle$:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F_n	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377

Unlike the harmonic numbers and the Bernoulli numbers, the Fibonacci numbers are nice simple integers. They are defined by the recurrence

$$\begin{aligned} F_0 &= 0; \\ F_1 &= 1; \\ F_n &= F_{n-1} + F_{n-2}, \qquad \text{for } n > 1. \end{aligned} \tag{6.102}$$

The simplicity of this rule — the simplest possible recurrence in which each number depends on the previous two — accounts for the fact that Fibonacci numbers occur in a wide variety of situations.

The back-to-nature nature of this example is shocking. This book should be banned.

"Bee trees" provide a good example of how Fibonacci numbers can arise naturally. Let's consider the pedigree of a male bee. Each male (also known as a drone) is produced asexually from a female (also known as a queen); each female, however, has two parents, a male and a female. Here are the first few levels of the tree:

The drone has one grandfather and one grandmother; he has one great-grandfather and two great-grandmothers; he has two great-great-grandfathers and three great-great-grandmothers. In general, it is easy to see by induction that he has exactly F_{n+1} great^n-grandpas and F_{n+2} great^n-grandmas.

Fibonacci numbers are often found in nature, perhaps for reasons similar to the bee-tree law. For example, a typical sunflower has a large head that contains spirals of tightly packed florets, usually with 34 winding in one direction and 55 in another. Smaller heads will have 21 and 34, or 13 and 21; a gigantic sunflower with 89 and 144 spirals was once exhibited in England. Similar patterns are found in some species of pine cones.

Phyllotaxis, n. The love of taxis.

And here's an example of a different nature [277]: Suppose we put two panes of glass back-to-back. How many ways a_n are there for light rays to pass through or be reflected after changing direction n times? The first few

cases are:

When n is even, we have an even number of bounces and the ray passes through; when n is odd, the ray is reflected and it re-emerges on the same side it entered. The a_n's seem to be Fibonacci numbers, and a little staring at the figure tells us why: For $n \geqslant 2$, the n-bounce rays either take their first bounce off the opposite surface and continue in a_{n-1} ways, or they begin by bouncing off the middle surface and then bouncing back again to finish in a_{n-2} ways. Thus we have the Fibonacci recurrence $a_n = a_{n-1} + a_{n-2}$. The initial conditions are different, but not very different, because we have $a_0 = 1 = F_2$ and $a_1 = 2 = F_3$; therefore everything is simply shifted two places, and $a_n = F_{n+2}$.

Leonardo Fibonacci introduced these numbers in 1202, and mathematicians gradually began to discover more and more interesting things about them. Édouard Lucas, the perpetrator of the Tower of Hanoi puzzle discussed in Chapter 1, worked with them extensively in the last half of the nineteenth century (in fact it was Lucas who popularized the name "Fibonacci numbers"). One of his amazing results was to use properties of Fibonacci numbers to prove that the 39-digit Mersenne number $2^{127} - 1$ is prime.

"La suite de Fibonacci possède des propriétés nombreuses fort intéressantes."
—E. Lucas [259]

One of the oldest theorems about Fibonacci numbers, due to the French astronomer Jean-Dominique Cassini in 1680 [51], is the identity

$$F_{n+1} F_{n-1} - F_n^2 \;=\; (-1)^n\,, \qquad \text{for } n > 0. \tag{6.103}$$

When $n = 6$, for example, Cassini's identity correctly claims that $13 \cdot 5 - 8^2 = 1$.

A polynomial formula that involves Fibonacci numbers of the form $F_{n\pm k}$ for small values of k can be transformed into a formula that involves only F_n and F_{n+1}, because we can use the rule

$$F_m \;=\; F_{m+2} - F_{m+1} \tag{6.104}$$

to express F_m in terms of higher Fibonacci numbers when $m < n$, and we can use

$$F_m \;=\; F_{m-2} + F_{m-1} \tag{6.105}$$

to replace F_m by lower Fibonacci numbers when $m > n+1$. Thus, for example, we can replace F_{n-1} by $F_{n+1} - F_n$ in (6.103) to get Cassini's identity in the

form

$$F_{n+1}^2 - F_{n+1}F_n - F_n^2 \;=\; (-1)^n\,. \tag{6.106}$$

Moreover, Cassini's identity reads

$$F_{n+2}F_n - F_{n+1}^2 \;=\; (-1)^{n+1}$$

when n is replaced by $n+1$; this is the same as $(F_{n+1}+F_n)F_n - F_{n+1}^2 = (-1)^{n+1}$, which is the same as (6.106). Thus Cassini(n) is true if and only if Cassini(n+1) is true; equation (6.103) holds for all n by induction.

Cassini's identity is the basis of a geometrical paradox that was one of Lewis Carroll's favorite puzzles [63], [319], [364]. The idea is to take a chessboard and cut it into four pieces as shown here, then to reassemble the pieces into a rectangle:

The paradox is explained because . . . well, magic tricks aren't supposed to be explained.

Presto: The original area of $8 \times 8 = 64$ squares has been rearranged to yield $5 \times 13 = 65$ squares! A similar construction dissects any $F_n \times F_n$ square into four pieces, using F_{n+1}, F_n, F_{n-1}, and F_{n-2} as dimensions wherever the illustration has 13, 8, 5, and 3 respectively. The result is an $F_{n-1} \times F_{n+1}$ rectangle; by (6.103), one square has therefore been gained or lost, depending on whether n is even or odd.

Strictly speaking, we can't apply the reduction (6.105) unless $m \geqslant 2$, because we haven't defined F_n for negative n. A lot of maneuvering becomes easier if we eliminate this boundary condition and use (6.104) and (6.105) to define Fibonacci numbers with negative indices. For example, F_{-1} turns out to be $F_1 - F_0 = 1$; then F_{-2} is $F_0 - F_{-1} = -1$. In this way we deduce the values

n	0	−1	−2	−3	−4	−5	−6	−7	−8	−9	−10	−11
F_n	0	1	−1	2	−3	5	−8	13	−21	34	−55	89

and it quickly becomes clear (by induction) that

$$F_{-n} \;=\; (-1)^{n-1}F_n\,, \qquad \text{integer } n. \tag{6.107}$$

Cassini's identity (6.103) is true for *all* integers n, not just for $n > 0$, when we extend the Fibonacci sequence in this way.

The process of reducing $F_{n\pm k}$ to a combination of F_n and F_{n+1} by using (6.105) and (6.104) leads to the sequence of formulas

$$\begin{aligned} F_{n+2} &= F_{n+1} + F_n & F_{n-1} &= F_{n+1} - F_n \\ F_{n+3} &= 2F_{n+1} + F_n & F_{n-2} &= -F_{n+1} + 2F_n \\ F_{n+4} &= 3F_{n+1} + 2F_n & F_{n-3} &= 2F_{n+1} - 3F_n \\ F_{n+5} &= 5F_{n+1} + 3F_n & F_{n-4} &= -3F_{n+1} + 5F_n \end{aligned}$$

in which another pattern becomes obvious:

$$F_{n+k} = F_k F_{n+1} + F_{k-1} F_n \,. \tag{6.108}$$

This identity, easily proved by induction, holds for all integers k and n (positive, negative, or zero).

If we set $k = n$ in (6.108), we find that

$$F_{2n} = F_n F_{n+1} + F_{n-1} F_n \,; \tag{6.109}$$

hence F_{2n} is a multiple of F_n. Similarly,

$$F_{3n} = F_{2n} F_{n+1} + F_{2n-1} F_n \,,$$

and we may conclude that F_{3n} is also a multiple of F_n. By induction,

$$F_{kn} \text{ is a multiple of } F_n \,, \tag{6.110}$$

for all integers k and n. This explains, for example, why F_{15} (which equals 610) is a multiple of both F_3 and F_5 (which are equal to 2 and 5). Even more is true, in fact; exercise 27 proves that

$$\gcd(F_m, F_n) = F_{\gcd(m,n)} \,. \tag{6.111}$$

For example, $\gcd(F_{12}, F_{18}) = \gcd(144, 2584) = 8 = F_6$.

We can now prove a converse of (6.110): If $n > 2$ and if F_m is a multiple of F_n, then m is a multiple of n. For if $F_n \backslash F_m$ then $F_n \backslash \gcd(F_m, F_n) = F_{\gcd(m,n)} \leqslant F_n$. This is possible only if $F_{\gcd(m,n)} = F_n$; and our assumption that $n > 2$ makes it mandatory that $\gcd(m, n) = n$. Hence $n \backslash m$.

An extension of these divisibility ideas was used by Yuri Matijasevich in his famous proof [266] that there is no algorithm to decide if a given multivariate polynomial equation with integer coefficients has a solution in integers. Matijasevich's lemma states that, if $n > 2$, the Fibonacci number F_m is a multiple of F_n^2 if and only if m is a multiple of nF_n.

Let's prove this by looking at the sequence $\langle F_{kn} \bmod F_n^2 \rangle$ for $k = 1, 2, 3, \ldots$, and seeing when $F_{kn} \bmod F_n^2 = 0$. (We know that m must have the

form kn if $F_m \bmod F_n = 0$.) First we have $F_n \bmod F_n^2 = F_n$; that's not zero. Next we have

$$F_{2n} \;=\; F_nF_{n+1} + F_{n-1}F_n \;\equiv\; 2F_nF_{n+1} \pmod{F_n^2},$$

by (6.108), since $F_{n+1} \equiv F_{n-1} \pmod{F_n}$. Similarly

$$F_{2n+1} \;=\; F_{n+1}^2 + F_n^2 \;\equiv\; F_{n+1}^2 \pmod{F_n^2}.$$

This congruence allows us to compute

$$\begin{aligned} F_{3n} &= F_{2n+1}F_n + F_{2n}F_{n-1} \\ &\equiv F_{n+1}^2F_n + (2F_nF_{n+1})F_{n+1} \;=\; 3F_{n+1}^2F_n \qquad \pmod{F_n^2}; \\ F_{3n+1} &= F_{2n+1}F_{n+1} + F_{2n}F_n \\ &\equiv F_{n+1}^3 + (2F_nF_{n+1})F_n \;\equiv\; F_{n+1}^3 \qquad\qquad \pmod{F_n^2}. \end{aligned}$$

In general, we find by induction on k that

$$F_{kn} \;\equiv\; kF_nF_{n+1}^{k-1} \quad\text{and}\quad F_{kn+1} \;\equiv\; F_{n+1}^k \pmod{F_n^2}.$$

Now F_{n+1} is relatively prime to F_n, so

$$\begin{aligned} F_{kn} \;\equiv\; 0 \pmod{F_n^2} \quad &\Longleftrightarrow \quad kF_n \;\equiv\; 0 \pmod{F_n^2} \\ &\Longleftrightarrow \quad k \;\equiv\; 0 \pmod{F_n}. \end{aligned}$$

We have proved Matijasevich's lemma.

One of the most important properties of the Fibonacci numbers is the special way in which they can be used to represent integers. Let's write

$$j \gg k \qquad \Longleftrightarrow \qquad j \geqslant k+2. \tag{6.112}$$

Then *every positive integer has a unique representation of the form*

$$n \;=\; F_{k_1} + F_{k_2} + \cdots + F_{k_r}, \qquad k_1 \gg k_2 \gg \cdots \gg k_r \gg 0. \tag{6.113}$$

(This is "Zeckendorf's theorem" [246], [381].) For example, the representation of one million turns out to be

$$\begin{aligned} 1000000 &= 832040 + 121393 + 46368 + 144 + 55 \\ &= \quad F_{30} \;+\; F_{26} \;+\; F_{24} \;+ F_{12} + F_{10}. \end{aligned}$$

We can always find such a representation by using a "greedy" approach, choosing F_{k_1} to be the largest Fibonacci number $\leqslant n$, then choosing F_{k_2} to be the largest that is $\leqslant n - F_{k_1}$, and so on. (More precisely, suppose that

$F_k \leqslant n < F_{k+1}$; then we have $0 \leqslant n - F_k < F_{k+1} - F_k = F_{k-1}$. If n is a Fibonacci number, (6.113) holds with $r = 1$ and $k_1 = k$. Otherwise $n - F_k$ has a Fibonacci representation $F_{k_2} + \cdots + F_{k_r}$, by induction on n; and (6.113) holds if we set $k_1 = k$, because the inequalities $F_{k_2} \leqslant n - F_k < F_{k-1}$ imply that $k \gg k_2$.) Conversely, any representation of the form (6.113) implies that

$$F_{k_1} \leqslant n < F_{k_1+1}\,,$$

because the largest possible value of $F_{k_2} + \cdots + F_{k_r}$ when $k \gg k_2 \gg \cdots \gg k_r \gg 0$ is

$$F_{k-2} + F_{k-4} + \cdots + F_{k \bmod 2+2} = F_{k-1} - 1\,, \qquad \text{if } k \geqslant 2. \tag{6.114}$$

(This formula is easy to prove by induction on k; the left-hand side is zero when k is 2 or 3.) Therefore k_1 is the greedily chosen value described earlier, and the representation must be unique.

Any unique system of representation is a number system; therefore Zeckendorf's theorem leads to the *Fibonacci number system*. We can represent any nonnegative integer n as a sequence of 0's and 1's, writing

$$n = (b_m b_{m-1} \ldots b_2)_F \quad \Longleftrightarrow \quad n = \sum_{k=2}^{m} b_k F_k\,. \tag{6.115}$$

This number system is something like binary (radix 2) notation, except that there never are two adjacent 1's. For example, here are the numbers from 1 to 20, expressed Fibonacci-wise:

$1 = (000001)_F$	$6 = (001001)_F$	$11 = (010100)_F$	$16 = (100100)_F$
$2 = (000010)_F$	$7 = (001010)_F$	$12 = (010101)_F$	$17 = (100101)_F$
$3 = (000100)_F$	$8 = (010000)_F$	$13 = (100000)_F$	$18 = (101000)_F$
$4 = (000101)_F$	$9 = (010001)_F$	$14 = (100001)_F$	$19 = (101001)_F$
$5 = (001000)_F$	$10 = (010010)_F$	$15 = (100010)_F$	$20 = (101010)_F$

The Fibonacci representation of a million, shown a minute ago, can be contrasted with its binary representation $2^{19} + 2^{18} + 2^{17} + 2^{16} + 2^{14} + 2^9 + 2^6$:

$$\begin{aligned}(1000000)_{10} &= (1000101000000000010100000000)_F \\ &= (11110100001001000000)_2\,.\end{aligned}$$

The Fibonacci representation needs a few more bits because adjacent 1's are not permitted; but the two representations are analogous.

To add 1 in the Fibonacci number system, there are two cases: If the "units digit" is 0, we change it to 1; that adds $F_2 = 1$, since the units digit

refers to F_2. Otherwise the two least significant digits will be 01, and we change them to 10 (thereby adding $F_3 - F_2 = 1$). Finally, we must "carry" as much as necessary by changing the digit pattern '011' to '100' until there are no two 1's in a row. (This carry rule is equivalent to replacing $F_{m+1} + F_m$ by F_{m+2}.) For example, to go from $5 = (1000)_F$ to $6 = (1001)_F$ or from $6 = (1001)_F$ to $7 = (1010)_F$ requires no carrying; but to go from $7 = (1010)_F$ to $8 = (10000)_F$ we must carry twice.

So far we've been discussing lots of properties of the Fibonacci numbers, but we haven't come up with a closed formula for them. We haven't found closed forms for Stirling numbers, Eulerian numbers, or Bernoulli numbers either; but we were able to discover the closed form $H_n = \left[{n+1 \atop 2}\right]/n!$ for harmonic numbers. Is there a relation between F_n and other quantities we know? Can we "solve" the recurrence that defines F_n?

"Sit $1 + x + 2xx + 3x^3 + 5x^4 + 8x^5 + 13x^6 + 21x^7 + 34x^8$ &c Series nata ex divisione Unitatis per Trinomium $1 - x - xx$."
—A. de Moivre [76]

The answer is yes. In fact, there's a simple way to solve the recurrence by using the idea of *generating function* that we looked at briefly in Chapter 5. Let's consider the infinite series

$$F(z) = F_0 + F_1 z + F_2 z^2 + \cdots = \sum_{n \geqslant 0} F_n z^n . \tag{6.116}$$

If we can find a simple formula for $F(z)$, chances are reasonably good that we can find a simple formula for its coefficients F_n.

"The quantities r, s, t, which show the relation of the terms, are the same as those in the denominator of the fraction. This property, howsoever obvious it may be, M. DeMoivre was the first that applied it to use, in the solution of problems about infinite series, which otherwise would have been very intricate."
—J. Stirling [343]

In Chapter 7 we will focus on generating functions in detail, but it will be helpful to have this example under our belts by the time we get there. The power series $F(z)$ has a nice property if we look at what happens when we multiply it by z and by z^2:

$$\begin{aligned} F(z) &= F_0 + F_1 z + F_2 z^2 + F_3 z^3 + F_4 z^4 + F_5 z^5 + \cdots , \\ zF(z) &= \phantom{F_0 +{}} F_0 z + F_1 z^2 + F_2 z^3 + F_3 z^4 + F_4 z^5 + \cdots , \\ z^2 F(z) &= \phantom{F_0 + F_1 z +{}} F_0 z^2 + F_1 z^3 + F_2 z^4 + F_3 z^5 + \cdots . \end{aligned}$$

If we now subtract the last two equations from the first, the terms that involve z^2, z^3, and higher powers of z will all disappear, because of the Fibonacci recurrence. Furthermore the constant term F_0 never actually appeared in the first place, because $F_0 = 0$. Therefore all that's left after the subtraction is $(F_1 - F_0)z$, which is just z. In other words,

$$F(z) - zF(z) - z^2 F(z) = z ,$$

and solving for $F(z)$ gives us the compact formula

$$F(z) = \frac{z}{1 - z - z^2} . \tag{6.117}$$

We have now boiled down all the information in the Fibonacci sequence to a simple (although unrecognizable) expression $z/(1-z-z^2)$. This, believe it or not, is progress, because we can factor the denominator and then use partial fractions to achieve a formula that we can easily expand in power series. The coefficients in this power series will be a closed form for the Fibonacci numbers.

The plan of attack just sketched can perhaps be understood better if we approach it backwards. If we have a simpler generating function, say $1/(1-\alpha z)$ where α is a constant, we know the coefficients of all powers of z, because

$$\frac{1}{1-\alpha z} = 1 + \alpha z + \alpha^2 z^2 + \alpha^3 z^3 + \cdots .$$

Similarly, if we have a generating function of the form $A/(1-\alpha z)+B/(1-\beta z)$, the coefficients are easily determined, because

$$\begin{aligned}\frac{A}{1-\alpha z} + \frac{B}{1-\beta z} &= A\sum_{n\geqslant 0}(\alpha z)^n + B\sum_{n\geqslant 0}(\beta z)^n \\ &= \sum_{n\geqslant 0}(A\alpha^n + B\beta^n)z^n . \qquad (6.118)\end{aligned}$$

Therefore all we have to do is find constants A, B, α, and β such that

$$\frac{A}{1-\alpha z} + \frac{B}{1-\beta z} = \frac{z}{1-z-z^2},$$

and we will have found a closed form $A\alpha^n + B\beta^n$ for the coefficient F_n of z^n in $F(z)$. The left-hand side can be rewritten

$$\frac{A}{1-\alpha z} + \frac{B}{1-\beta z} = \frac{A - A\beta z + B - B\alpha z}{(1-\alpha z)(1-\beta z)},$$

so the four constants we seek are the solutions to two polynomial equations:

$$(1-\alpha z)(1-\beta z) = 1 - z - z^2 ; \qquad (6.119)$$

$$(A+B) - (A\beta + B\alpha)z = z . \qquad (6.120)$$

We want to factor the denominator of $F(z)$ into the form $(1-\alpha z)(1-\beta z)$; then we will be able to express $F(z)$ as the sum of two fractions in which the factors $(1-\alpha z)$ and $(1-\beta z)$ are conveniently separated from each other.

Notice that the denominator factors in (6.119) have been written in the form $(1-\alpha z)(1-\beta z)$, instead of the more usual form $c(z-\rho_1)(z-\rho_2)$ where ρ_1 and ρ_2 are the roots. The reason is that $(1-\alpha z)(1-\beta z)$ leads to nicer expansions in power series.

As usual, the authors can't resist a trick.

We can find α and β in several ways, one of which uses a slick trick: Let us introduce a new variable w and try to find the factorization

$$w^2 - wz - z^2 \;=\; (w - \alpha z)(w - \beta z)\,.$$

Then we can simply set $w = 1$ and we'll have the factors of $1 - z - z^2$. The roots of $w^2 - wz - z^2 = 0$ can be found by the quadratic formula; they are

$$\frac{z \pm \sqrt{z^2 + 4z^2}}{2} \;=\; \frac{1 \pm \sqrt{5}}{2}\,z\,.$$

Therefore

$$w^2 - wz - z^2 \;=\; \left(w - \frac{1+\sqrt{5}}{2}\,z\right)\left(w - \frac{1-\sqrt{5}}{2}\,z\right)$$

and we have the constants α and β we were looking for.

The ratio of one's height to the height of one's navel is approximately 1.618, *according to extensive empirical observations by European scholars [136].*

The number $(1+\sqrt{5})/2 \approx 1.61803$ is important in many parts of mathematics as well as in the art world, where it has been considered since ancient times to be the most pleasing ratio for many kinds of design. Therefore it has a special name, the *golden ratio*. We denote it by the Greek letter ϕ, in honor of Phidias who is said to have used it consciously in his sculpture. The other root $(1-\sqrt{5})/2 = -1/\phi \approx -.61803$ shares many properties of ϕ, so it has the special name $\hat{\phi}$, "phi hat." These numbers are roots of the equation $w^2 - w - 1 = 0$, so we have

$$\phi^2 \;=\; \phi + 1\,; \qquad \hat{\phi}^2 \;=\; \hat{\phi} + 1\,. \tag{6.121}$$

(More about ϕ and $\hat{\phi}$ later.)

We have found the constants $\alpha = \phi$ and $\beta = \hat{\phi}$ needed in (6.119); now we merely need to find A and B in (6.120). Setting $z = 0$ in that equation tells us that $B = -A$, so (6.120) boils down to

$$-\hat{\phi}A + \phi A \;=\; 1\,.$$

The solution is $A = 1/(\phi - \hat{\phi}) = 1/\sqrt{5}$; the partial fraction expansion of (6.117) is therefore

$$F(z) \;=\; \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi} z}\right). \tag{6.122}$$

Good, we've got $F(z)$ right where we want it. Expanding the fractions into power series as in (6.118) gives a closed form for the coefficient of z^n:

$$F_n \;=\; \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)\,. \tag{6.123}$$

(This formula was first published by Leonhard Euler [113] in 1765, but people forgot about it until it was rediscovered by Jacques Binet [31] in 1843.)

Before we stop to marvel at our derivation, we should check its accuracy. For $n = 0$ the formula correctly gives $F_0 = 0$; for $n = 1$, it gives $F_1 = (\phi - \hat{\phi})/\sqrt{5}$, which is indeed 1. For higher powers, equations (6.121) show that the numbers defined by (6.123) satisfy the Fibonacci recurrence, so they must be the Fibonacci numbers by induction. (We could also expand ϕ^n and $\hat{\phi}^n$ by the binomial theorem and chase down the various powers of $\sqrt{5}$; but that gets pretty messy. The point of a closed form is not necessarily to provide us with a fast method of calculation, but rather to tell us how F_n relates to other quantities in mathematics.)

With a little clairvoyance we could simply have guessed formula (6.123) and proved it by induction. But the method of generating functions is a powerful way to discover it; in Chapter 7 we'll see that the same method leads us to the solution of recurrences that are considerably more difficult. Incidentally, we never worried about whether the infinite sums in our derivation of (6.123) were convergent; it turns out that most operations on the coefficients of power series can be justified rigorously whether or not the sums actually converge [182]. Still, skeptical readers who suspect fallacious reasoning with infinite sums can take comfort in the fact that equation (6.123), once found by using infinite series, can be verified by a solid induction proof.

One of the interesting consequences of (6.123) is that the integer F_n is extremely close to the irrational number $\phi^n/\sqrt{5}$ when n is large. (Since $\hat{\phi}$ is less than 1 in absolute value, $\hat{\phi}^n$ becomes exponentially small and its effect is almost negligible.) For example, $F_{10} = 55$ and $F_{11} = 89$ are very near

$$\frac{\phi^{10}}{\sqrt{5}} \approx 55.00364 \quad \text{and} \quad \frac{\phi^{11}}{\sqrt{5}} \approx 88.99775\,.$$

We can use this observation to derive another closed form,

$$F_n = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor = \frac{\phi^n}{\sqrt{5}} \quad \text{rounded to the nearest integer,} \tag{6.124}$$

because $\left|\hat{\phi}^n/\sqrt{5}\right| < \frac{1}{2}$ for all $n \geqslant 0$. When n is even, F_n is a little bit less than $\phi^n/\sqrt{5}$; otherwise it is a little greater.

Cassini's identity (6.103) can be rewritten

$$\frac{F_{n+1}}{F_n} - \frac{F_n}{F_{n-1}} = \frac{(-1)^n}{F_{n-1}\,F_n}\,.$$

When n is large, $1/F_{n-1}F_n$ is very small, so F_{n+1}/F_n must be very nearly the same as F_n/F_{n-1}; and (6.124) tells us that this ratio approaches ϕ. In fact, we have

$$F_{n+1} = \phi F_n + \hat{\phi}^n\,. \tag{6.125}$$

(This identity is true by inspection when $n = 0$ or $n = 1$, and by induction when $n > 1$; we can also prove it directly by plugging in (6.123).) The ratio F_{n+1}/F_n is very close to ϕ, which it alternately overshoots and undershoots.

By coincidence, ϕ is also very nearly the number of kilometers in a mile. (The exact number is 1.609344, since 1 inch is exactly 2.54 centimeters.) This gives us a handy way to convert mentally between kilometers and miles, because a distance of F_{n+1} kilometers is (very nearly) a distance of F_n miles.

If the USA ever goes metric, our speed limit signs will go from 55 mi/hr to 89 km/hr. Or maybe the highway people will be generous and let us go 90.

Suppose we want to convert a non-Fibonacci number from kilometers to miles; what is 30 km, American style? Easy: We just use the Fibonacci number system and mentally convert 30 to its Fibonacci representation $21 + 8 + 1$ by the greedy approach explained earlier. Now we can shift each number down one notch, getting $13 + 5 + 1$. (The former '1' was F_2, since $k_r \gg 0$ in (6.113); the new '1' is F_1.) Shifting down divides by ϕ, more or less. Hence 19 miles is our estimate. (That's pretty close; the correct answer is about 18.64 miles.) Similarly, to go from miles to kilometers we can shift up a notch; 30 miles is approximately $34 + 13 + 2 = 49$ kilometers. (That's not quite as close; the correct number is about 48.28.)

It turns out that this shift-down rule gives the correctly rounded number of miles per n kilometers for all $n \leqslant 100$, except in the cases $n = 4$, 12, 62, 75, 91, and 96, when it is off by less than 2/3 mile. And the shift-up rule gives either the correctly rounded number of kilometers for n miles, or 1 km too many, for all $n \leqslant 126$. (The only really embarrassing case is $n = 4$, where the individual rounding errors for $n = 3 + 1$ both go the same direction instead of cancelling each other out.)

The "shift down" rule changes n to $f(n/\phi)$ and the "shift up" rule changes n to $f(n\phi)$, where $f(x) = \lfloor x + \phi^{-1} \rfloor$.

6.7 CONTINUANTS

Fibonacci numbers have important connections to the Stern–Brocot tree that we studied in Chapter 4, and they have important generalizations to a sequence of polynomials that Euler studied extensively. These polynomials are called *continuants*, because they are the key to the study of continued fractions like

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cfrac{1}{a_5 + \cfrac{1}{a_6 + \cfrac{1}{a_7}}}}}}}. \tag{6.126}$$

The continuant polynomial $K_n(x_1, x_2, \ldots, x_n)$ has n parameters, and it is defined by the following recurrence:

$$\begin{aligned} K_0() &= 1\,; \\ K_1(x_1) &= x_1\,; \\ K_n(x_1, \ldots, x_n) &= K_{n-1}(x_1, \ldots, x_{n-1})x_n + K_{n-2}(x_1, \ldots, x_{n-2})\,. \end{aligned} \tag{6.127}$$

For example, the next three cases after $K_1(x_1)$ are

$$\begin{aligned} K_2(x_1, x_2) &= x_1x_2 + 1\,; \\ K_3(x_1, x_2, x_3) &= x_1x_2x_3 + x_1 + x_3\,; \\ K_4(x_1, x_2, x_3, x_4) &= x_1x_2x_3x_4 + x_1x_2 + x_1x_4 + x_3x_4 + 1\,. \end{aligned}$$

It's easy to see, inductively, that the number of terms is a Fibonacci number:

$$K_n(1, 1, \ldots, 1) = F_{n+1}\,. \tag{6.128}$$

When the number of parameters is implied by the context, we can write simply 'K' instead of 'K_n', just as we can omit the number of parameters when we use the hypergeometric functions F of Chapter 5. For example, $K(x_1, x_2) = K_2(x_1, x_2) = x_1x_2 + 1$. The subscript n is of course necessary in formulas like (6.128).

Euler observed that $K(x_1, x_2, \ldots, x_n)$ can be obtained by starting with the product $x_1x_2 \ldots x_n$ and then striking out adjacent pairs x_kx_{k+1} in all possible ways. We can represent Euler's rule graphically by constructing all "Morse code" sequences of dots and dashes having length n, where each dot contributes 1 to the length and each dash contributes 2; here are the Morse code sequences of length 4:

$$\bullet\bullet\bullet\bullet \qquad \bullet\bullet - \qquad \bullet - \bullet \qquad - \bullet\bullet \qquad - -$$

These dot-dash patterns correspond to the terms of $K(x_1, x_2, x_3, x_4)$; a dot signifies a variable that's included and a dash signifies a pair of variables that's excluded. For example, $\bullet - \bullet$ corresponds to x_1x_4.

A Morse code sequence of length n that has k dashes has $n-2k$ dots and $n-k$ symbols altogether. These dots and dashes can be arranged in $\binom{n-k}{k}$ ways; therefore if we replace each dot by z and each dash by 1 we get

$$K_n(z, z, \ldots, z) = \sum_{k=0}^{n} \binom{n-k}{k} z^{n-2k}\,. \tag{6.129}$$

We also know that the total number of terms in a continuant is a Fibonacci number; hence we have the identity

$$F_{n+1} \;=\; \sum_{k=0}^{n} \binom{n-k}{k} . \tag{6.130}$$

(A closed form for (6.129), generalizing the Euler–Binet formula (6.123) for Fibonacci numbers, appears in (5.74).)

The relation between continuant polynomials and Morse code sequences shows that continuants have a mirror symmetry:

$$K(x_n, \ldots, x_2, x_1) \;=\; K(x_1, x_2, \ldots, x_n) . \tag{6.131}$$

Therefore they obey a recurrence that adjusts parameters at the left, in addition to the right-adjusting recurrence in definition (6.127):

$$K_n(x_1, \ldots, x_n) \;=\; x_1 K_{n-1}(x_2, \ldots, x_n) + K_{n-2}(x_3, \ldots, x_n) . \tag{6.132}$$

Both of these recurrences are special cases of a more general law:

$$\begin{aligned} &K_{m+n}(x_1, \ldots, x_m, x_{m+1}, \ldots, x_{m+n}) \\ &\qquad = K_m(x_1, \ldots, x_m)\, K_n(x_{m+1}, \ldots, x_{m+n}) \\ &\qquad\quad + K_{m-1}(x_1, \ldots, x_{m-1})\, K_{n-1}(x_{m+2}, \ldots, x_{m+n}) . \end{aligned} \tag{6.133}$$

This law is easily understood from the Morse code analogy: The first product $K_m K_n$ yields the terms of K_{m+n} in which there is no dash in the $[m, m+1]$ position, while the second product yields the terms in which there is a dash there. If we set all the x's equal to 1, this identity tells us that $F_{m+n+1} = F_{m+1}F_{n+1} + F_m F_n$; thus, (6.108) is a special case of (6.133).

Euler [112] discovered that continuants obey an even more remarkable law, which generalizes Cassini's identity:

$$\begin{aligned} &K_{m+n}(x_1, \ldots, x_{m+n})\, K_k(x_{m+1}, \ldots, x_{m+k}) \\ &\quad = K_{m+k}(x_1, \ldots, x_{m+k})\, K_n(x_{m+1}, \ldots, x_{m+n}) \\ &\qquad + (-1)^k K_{m-1}(x_1, \ldots, x_{m-1})\, K_{n-k-1}(x_{m+k+2}, \ldots, x_{m+n}) . \end{aligned} \tag{6.134}$$

This law (proved in exercise 29) holds whenever the subscripts on the K's are all nonnegative. For example, when $k = 2$, $m = 1$, and $n = 3$, we have

$$K(x_1, x_2, x_3, x_4)\, K(x_2, x_3) \;=\; K(x_1, x_2, x_3)\, K(x_2, x_3, x_4) + 1 .$$

Continuant polynomials are intimately connected with Euclid's algorithm. Suppose, for example, that the computation of $\gcd(m, n)$ finishes

in four steps:

$$\begin{aligned}
\gcd(m,n) &= \gcd(n_0,n_1) & n_0 &= m\,, \quad n_1 = n\,;\\
&= \gcd(n_1,n_2) & n_2 &= n_0 \bmod n_1 = n_0 - q_1 n_1\,;\\
&= \gcd(n_2,n_3) & n_3 &= n_1 \bmod n_2 = n_1 - q_2 n_2\,;\\
&= \gcd(n_3,n_4) & n_4 &= n_2 \bmod n_3 = n_2 - q_3 n_3\,;\\
&= \gcd(n_4,0) = n_4\,. & 0 &= n_3 \bmod n_4 = n_3 - q_4 n_4\,.
\end{aligned}$$

Then we have

$$\begin{aligned}
n_4 &= n_4 &&= K()n_4\,;\\
n_3 &= q_4 n_4 &&= K(q_4)n_4\,;\\
n_2 &= q_3 n_3 + n_4 &&= K(q_3,q_4)n_4\,;\\
n_1 &= q_2 n_2 + n_3 &&= K(q_2,q_3,q_4)n_4\,;\\
n_0 &= q_1 n_1 + n_2 &&= K(q_1,q_2,q_3,q_4)n_4\,.
\end{aligned}$$

In general, if Euclid's algorithm finds the greatest common divisor d in k steps, after computing the sequence of quotients $q_1, \ldots, q_k$, then the starting numbers were $K(q_1,q_2,\ldots,q_k)d$ and $K(q_2,\ldots,q_k)d$. (This fact was noticed early in the eighteenth century by Thomas Fantet de Lagny [232], who seems to have been the first person to consider continuants explicitly. Lagny pointed out that consecutive Fibonacci numbers, which occur as continuants when the q's take their minimum values, are therefore the smallest inputs that cause Euclid's algorithm to take a given number of steps.)

Continuants are also intimately connected with continued fractions, from which they get their name. We have, for example,

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3}}} = \frac{K(a_0,a_1,a_2,a_3)}{K(a_1,a_2,a_3)}\,. \tag{6.135}$$

The same pattern holds for continued fractions of any depth. It is easily proved by induction; we have, for example,

$$\frac{K(a_0,a_1,a_2,a_3+1/a_4)}{K(a_1,a_2,a_3+1/a_4)} = \frac{K(a_0,a_1,a_2,a_3,a_4)}{K(a_1,a_2,a_3,a_4)}\,,$$

because of the identity

$$\begin{aligned}
&K_n(x_1,\ldots,x_{n-1},x_n+y)\\
&\qquad = K_n(x_1,\ldots,x_{n-1},x_n) + K_{n-1}(x_1,\ldots,x_{n-1})\,y\,.
\end{aligned} \tag{6.136}$$

(This identity is proved and generalized in exercise 30.)

Moreover, continuants are closely connected with the Stern–Brocot tree discussed in Chapter 4. Each node in that tree can be represented as a sequence of L's and R's, say

$$R^{a_0}L^{a_1}R^{a_2}L^{a_3}\ldots R^{a_{n-2}}L^{a_{n-1}}\,, \tag{6.137}$$

where $a_0 \geqslant 0$, $a_1 \geqslant 1$, $a_2 \geqslant 1$, $a_3 \geqslant 1$, $\ldots$, $a_{n-2} \geqslant 1$, $a_{n-1} \geqslant 0$, and n is even. Using the 2×2 matrices L and R of (4.33), it is not hard to prove by induction that the matrix equivalent of (6.137) is

$$\begin{pmatrix} K_{n-2}(a_1,\ldots,a_{n-2}) & K_{n-1}(a_1,\ldots,a_{n-2},a_{n-1}) \\ K_{n-1}(a_0,a_1,\ldots,a_{n-2}) & K_n(a_0,a_1,\ldots,a_{n-2},a_{n-1}) \end{pmatrix}. \tag{6.138}$$

(The proof is part of exercise 87.) For example,

$$R^aL^bR^cL^d \;=\; \begin{pmatrix} bc+1 & bcd+b+d \\ abc+a+c & abcd+ab+ad+cd+1 \end{pmatrix}.$$

Finally, therefore, we can use (4.34) to write a closed form for the fraction in the Stern–Brocot tree whose L-and-R representation is (6.137):

$$f(R^{a_0}\ldots L^{a_{n-1}}) \;=\; \frac{K_{n+1}(a_0,a_1,\ldots,a_{n-1},1)}{K_n(a_1,\ldots,a_{n-1},1)}\,. \tag{6.139}$$

(This is "Halphen's theorem" [174].) For example, to find the fraction for LRRL we have $a_0 = 0$, $a_1 = 1$, $a_2 = 2$, $a_3 = 1$, and $n = 4$; equation (6.139) gives

$$\frac{K(0,1,2,1,1)}{K(1,2,1,1)} \;=\; \frac{K(2,1,1)}{K(1,2,1,1)} \;=\; \frac{K(2,2)}{K(3,2)} \;=\; \frac{5}{7}\,.$$

(We have used the rule $K_n(x_1,\ldots,x_{n-1},x_n+1) = K_{n+1}(x_1,\ldots,x_{n-1},x_n,1)$ to absorb leading and trailing 1's in the parameter lists; this rule is obtained by setting $y = 1$ in (6.136).)

A comparison of (6.135) and (6.139) shows that the fraction corresponding to a general node (6.137) in the Stern–Brocot tree has the continued fraction representation

$$f(R^{a_0}\ldots L^{a_{n-1}}) \;=\; a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ldots + \cfrac{1}{a_{n-1} + \cfrac{1}{1}}}}}\,. \tag{6.140}$$

Thus we can convert at sight between continued fractions and the corresponding nodes in the Stern–Brocot tree. For example,

$$f(\mathrm{LRRL}) \;=\; 0 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1}}}}\,.$$

We observed in Chapter 4 that irrational numbers define infinite paths in the Stern–Brocot tree, and that they can be represented as an infinite string of L's and R's. If the infinite string for α is $R^{a_0}L^{a_1}R^{a_2}L^{a_3}\ldots$, there is a corresponding infinite continued fraction

$$\alpha \;=\; a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cfrac{1}{a_5 + \cfrac{1}{\ddots}}}}}}\,. \tag{6.141}$$

This infinite continued fraction can also be obtained directly: Let $\alpha_0 = \alpha$ and for $k \geqslant 0$ let

$$a_k \;=\; \lfloor \alpha_k \rfloor\,; \qquad \alpha_k \;=\; a_k + \frac{1}{\alpha_{k+1}}\,. \tag{6.142}$$

The a's are called the "partial quotients" of α. If α is rational, say m/n, this process runs through the quotients found by Euclid's algorithm and then stops (with $\alpha_{k+1} = \infty$).

Is Euler's constant γ rational or irrational? Nobody knows. We can get partial information about this famous unsolved problem by looking for γ in the Stern–Brocot tree; if it's rational we will find it, and if it's irrational we will find all the closest rational approximations to it. The continued fraction for γ begins with the following partial quotients:

Or if they do, they're not talking.

k	0	1	2	3	4	5	6	7	8
a_k	0	1	1	2	1	2	1	4	3

Therefore its Stern–Brocot representation begins LRLLRLLRLLLLRRRL…; no pattern is evident. Calculations by Richard Brent [38] have shown that, if γ is rational, its denominator must be more than 10,000 decimal digits long.

Well, γ must be irrational, because of a little-known Einsteinian assertion: "God does not throw huge denominators at the universe."

Therefore nobody believes that γ is rational; but nobody so far has been able to prove that it isn't.

Let's conclude this chapter by proving a remarkable identity that ties a lot of these ideas together. We introduced the notion of spectrum in Chapter 3; the spectrum of α is the multiset of numbers $\lfloor n\alpha \rfloor$, where α is a given constant. The infinite series

$$\sum_{n\geqslant 1} z^{\lfloor n\phi \rfloor} \;=\; z + z^3 + z^4 + z^6 + z^8 + z^9 + \cdots$$

can therefore be said to be the generating function for the spectrum of ϕ, where $\phi = (1+\sqrt{5})/2$ is the golden ratio. The identity we will prove, discovered in 1976 by J. L. Davison [73], is an infinite continued fraction that relates this generating function to the Fibonacci sequence:

$$\cfrac{z^{F_1}}{1+\cfrac{z^{F_2}}{1+\cfrac{z^{F_3}}{1+\cfrac{z^{F_4}}{\ddots}}}} \;=\; (1-z)\sum_{n\geqslant 1} z^{\lfloor n\phi \rfloor}\,. \tag{6.143}$$

Both sides of (6.143) are interesting; let's look first at the numbers $\lfloor n\phi \rfloor$. If the Fibonacci representation (6.113) of n is $F_{k_1} + \cdots + F_{k_r}$, we expect $n\phi$ to be approximately $F_{k_1+1} + \cdots + F_{k_r+1}$, the number we get from shifting the Fibonacci representation left (as when converting from miles to kilometers). In fact, we know from (6.125) that

$$n\phi \;=\; F_{k_1+1} + \cdots + F_{k_r+1} - \bigl(\hat\phi^{k_1} + \cdots + \hat\phi^{k_r}\bigr)\,.$$

Now $\hat\phi = -1/\phi$ and $k_1 \gg \cdots \gg k_r \gg 0$, so we have

$$\begin{aligned}\bigl|\hat\phi^{k_1} + \cdots + \hat\phi^{k_r}\bigr| \;&<\; \phi^{-k_r} + \phi^{-k_r-2} + \phi^{-k_r-4} + \cdots \\ &=\; \frac{\phi^{-k_r}}{1-\phi^{-2}} \;=\; \phi^{1-k_r} \;\leqslant\; \phi^{-1} \;<\; 1\,;\end{aligned}$$

and $\hat\phi^{k_1} + \cdots + \hat\phi^{k_r}$ has the same sign as $(-1)^{k_r}$, by a similar argument. Hence

$$\lfloor n\phi \rfloor \;=\; F_{k_1+1} + \cdots + F_{k_r+1} \;-\; \bigl[k_r(n) \text{ is even}\bigr]\,. \tag{6.144}$$

Let us say that a number n is *Fibonacci odd* (or F-odd for short) if its least significant Fibonacci bit is 1; this is the same as saying that $k_r(n) = 2$. Otherwise n is *Fibonacci even* (F-even). For example, the smallest F-odd

numbers are 1, 4, 6, 9, 12, 14, 17, and 19. If $k_r(n)$ is even, then $n-1$ is F-even, by (6.114); similarly, if $k_r(n)$ is odd, then $n-1$ is F-odd. Therefore

$$k_r(n) \text{ is even} \quad \Longleftrightarrow \quad n-1 \text{ is F-even.}$$

Furthermore, if $k_r(n)$ is even, (6.144) implies that $k_r(\lfloor n\phi \rfloor) = 2$; if $k_r(n)$ is odd, (6.144) says that $k_r(\lfloor n\phi \rfloor) = k_r(n)+1$. Therefore $k_r(\lfloor n\phi \rfloor)$ is always even, and we have proved that

$$\lfloor n\phi \rfloor - 1 \text{ is always F-even.}$$

Conversely, if m is any F-even number, we can reverse this computation and find an n such that $m+1 = \lfloor n\phi \rfloor$. (First add 1 in F-notation as explained earlier. If no carries occur, n is $(m+2)$ shifted right; otherwise n is $(m+1)$ shifted right.) The right-hand sum of (6.143) can therefore be written

$$\sum_{n\geqslant 1} z^{\lfloor n\phi \rfloor} \;=\; z \sum_{m\geqslant 0} z^m [m \text{ is F-even}]\,. \tag{6.145}$$

How about the fraction on the left? Let's rewrite (6.143) so that the continued fraction looks like (6.141), with all numerators 1:

$$\cfrac{1}{z^{-F_0} + \cfrac{1}{z^{-F_1} + \cfrac{1}{z^{-F_2} + \cfrac{1}{\ddots}}}} \;=\; \frac{1-z}{z} \sum_{n\geqslant 1} z^{\lfloor n\phi \rfloor}\,. \tag{6.146}$$

(This transformation is a bit tricky! The numerator and denominator of the original fraction having z^{F_n} as numerator should be divided by $z^{F_{n-1}}$.) If we stop this new continued fraction at $1/z^{-F_n}$, its value will be a ratio of continuants,

$$\frac{K_{n+2}(0, z^{-F_0}, z^{-F_1}, \ldots, z^{-F_n})}{K_{n+1}(z^{-F_0}, z^{-F_1}, \ldots, z^{-F_n})} \;=\; \frac{K_n(z^{-F_1}, \ldots, z^{-F_n})}{K_{n+1}(z^{-F_0}, z^{-F_1}, \ldots, z^{-F_n})}\,,$$

as in (6.135). Let's look at the denominator first, in hopes that it will be tractable. Setting $Q_n = K_{n+1}(z^{-F_0}, \ldots, z^{-F_n})$, we find $Q_0 = 1$, $Q_1 = 1+z^{-1}$, $Q_2 = 1+z^{-1}+z^{-2}$, $Q_3 = 1+z^{-1}+z^{-2}+z^{-3}+z^{-4}$, and in general everything fits beautifully and gives a geometric series

$$Q_n \;=\; 1 + z^{-1} + z^{-2} + \cdots + z^{-(F_{n+2}-1)}\,.$$

The corresponding numerator is $P_n = K_n(z^{-F_1}, \ldots, z^{-F_n})$; this turns out to be like Q_n but with fewer terms. For example, we have

$$P_5 = z^{-1} + z^{-2} + z^{-4} + z^{-5} + z^{-7} + z^{-9} + z^{-10} + z^{-12},$$

compared with $Q_5 = 1 + z^{-1} + \cdots + z^{-12}$. A closer look reveals the pattern governing which terms are present: We have

$$P_5 = \frac{1+z^2+z^3+z^5+z^7+z^8+z^{10}+z^{11}}{z^{12}} = z^{-12} \sum_{m=0}^{12} z^m \,[m \text{ is F-even}];$$

and in general we can prove by induction that

$$P_n = z^{1-F_{n+2}} \sum_{m=0}^{F_{n+2}-1} z^m \,[m \text{ is F-even}].$$

Therefore

$$\frac{P_n}{Q_n} = \frac{\sum_{m=0}^{F_{n+2}-1} z^m \,[m \text{ is F-even}]}{\sum_{m=0}^{F_{n+2}-1} z^m}.$$

Taking the limit as $n \to \infty$ now gives (6.146), because of (6.145).

Exercises

Warmups

1 What are the $\binom{4}{2} = 11$ permutations of $\{1, 2, 3, 4\}$ that have exactly two cycles? (The cyclic forms appear in (6.4); non-cyclic forms like 2314 are desired instead.)

2 There are m^n functions from a set of n elements into a set of m elements. How many of them range over exactly k different function values?

3 Card stackers in the real world know that it's wise to allow a bit of slack so that the cards will not topple over when a breath of wind comes along. Suppose the center of gravity of the top k cards is required to be at least ϵ units from the edge of the $k+1$st card. (Thus, for example, the first card can overhang the second by at most $1-\epsilon$ units.) Can we still achieve arbitrarily large overhang, if we have enough cards?

4 Express $1/1 + 1/3 + \cdots + 1/(2n+1)$ in terms of harmonic numbers.

5 Explain how to get the recurrence (6.75) from the definition of $U_n(x, y)$ in (6.74), and solve the recurrence.

6 An explorer has left a pair of baby rabbits on an island. If baby rabbits become adults after one month, and if each pair of adult rabbits produces one pair of baby rabbits every month, how many pairs of rabbits are present after n months? (After two months there are two pairs, one of which is newborn.) Find a connection between this problem and the "bee tree" in the text.

If the harmonic numbers are worm numbers, the Fibonacci numbers are rabbit numbers.

7 Show that Cassini's identity (6.103) is a special case of (6.108), and a special case of (6.134).

8 Use the Fibonacci number system to convert 65 mi/hr into an approximate number of km/hr.

9 About how many square kilometers are in 8 square miles?

10 What is the continued fraction representation of ϕ?

Basics

11 What is $\sum_k (-1)^k \left[{n \atop k}\right]$, the row sum of Stirling's cycle-number triangle with alternating signs, when n is a nonnegative integer?

12 Prove that Stirling numbers have an inversion law analogous to (5.48):

$$g(n) \;=\; \sum_k \left\{{n \atop k}\right\}(-1)^k f(k) \quad\Longleftrightarrow\quad f(n) \;=\; \sum_k \left[{n \atop k}\right](-1)^k g(k)\,.$$

13 The differential operators $D = \frac{d}{dz}$ and $\vartheta = zD$ are mentioned in Chapters 2 and 5. We have

$$\vartheta^2 \;=\; z^2D^2 + zD\,,$$

because $\vartheta^2 f(z) = \vartheta z f'(z) = z\frac{d}{dz}zf'(z) = z^2f''(z) + zf'(z)$, which is $(z^2D^2+zD)f(z)$. Similarly it can be shown that $\vartheta^3 = z^3D^3+3z^2D^2+zD$. Prove the general formulas

$$\vartheta^n \;=\; \sum_k \left\{{n \atop k}\right\} z^k D^k\,,$$

$$z^nD^n \;=\; \sum_k \left[{n \atop k}\right](-1)^{n-k}\vartheta^k\,,$$

for all $n \geqslant 0$. (These can be used to convert between differential expressions of the forms $\sum_k \alpha_k z^k f^{(k)}(z)$ and $\sum_k \beta_k \vartheta^k f(z)$, as in (5.109).)

14 Prove the power identity (6.37) for Eulerian numbers.

15 Prove the Eulerian identity (6.39) by taking the mth difference of (6.37).

16 What is the general solution of the double recurrence

$$\begin{aligned} A_{n,0} &= a_n\,[n \geqslant 0]\,; \qquad A_{0,k} = 0\,, \qquad \text{if } k > 0; \\ A_{n,k} &= kA_{n-1,k} + A_{n-1,k-1}\,, \qquad \text{integers } k, n, \end{aligned}$$

when k and n range over the set of *all* integers?

17 Solve the following recurrences, assuming that $\left|{n \atop k}\right|$ is zero when $n < 0$ or $k < 0$:

a $$\left|{n \atop k}\right| = \left|{n-1 \atop k}\right| + n\left|{n-1 \atop k-1}\right| + [n=k=0]\,, \qquad \text{for } n, k \geqslant 0.$$

b $$\left|{n \atop k}\right| = (n-k)\left|{n-1 \atop k}\right| + \left|{n-1 \atop k-1}\right| + [n=k=0]\,, \qquad \text{for } n, k \geqslant 0.$$

c $$\left|{n \atop k}\right| = k\left|{n-1 \atop k}\right| + k\left|{n-1 \atop k-1}\right| + [n=k=0]\,, \qquad \text{for } n, k \geqslant 0.$$

18 Prove that the Stirling polynomials satisfy

$$(x+1)\,\sigma_n(x+1) = (x-n)\,\sigma_n(x) + x\sigma_{n-1}(x)\,.$$

19 Prove that the generalized Stirling numbers satisfy

$$\sum_{k=0}^{n} \left\{{x+k \atop x}\right\} \left[{x \atop x-n+k}\right] (-1)^k \Big/ \binom{x+k}{n+1} = 0\,, \quad \text{integer } n > 0.$$

$$\sum_{k=0}^{n} \left[{x+k \atop x}\right] \left\{{x \atop x-n+k}\right\} (-1)^k \Big/ \binom{x+k}{n+1} = 0\,, \quad \text{integer } n > 0.$$

20 Find a closed form for $\sum_{k=1}^{n} H_k^{(2)}$.

21 Show that if $H_n = a_n/b_n$, where a_n and b_n are integers, the denominator b_n is a multiple of $2^{\lfloor \lg n \rfloor}$. *Hint:* Consider the number $2^{\lfloor \lg n \rfloor - 1} H_n - \frac{1}{2}$.

22 Prove that the infinite sum

$$\sum_{k \geqslant 1} \left(\frac{1}{k} - \frac{1}{k+z} \right)$$

converges for all complex numbers z, except when z is a negative integer; and show that it equals H_z when z is a nonnegative integer. (Therefore we can use this formula to define harmonic numbers H_z when z is complex.)

23 Equation (6.81) gives the coefficients of $z/(e^z - 1)$, when expanded in powers of z. What are the coefficients of $z/(e^z + 1)$? *Hint:* Consider the identity $(e^z + 1)(e^z - 1) = e^{2z} - 1$.

24 Prove that the tangent number T_{2n+1} is a multiple of 2^n. *Hint:* Prove that all coefficients of $T_{2n}(x)$ and $T_{2n+1}(x)$ are multiples of 2^n.

25 Equation (6.57) proves that the worm will eventually reach the end of the rubber band at some time N. Therefore there must come a first time n when he's closer to the end after n minutes than he was after $n-1$ minutes. Show that $n < \frac{1}{2}N$.

26 Use summation by parts to evaluate $S_n = \sum_{k=1}^{n} H_k/k$. *Hint:* Consider also the related sum $\sum_{k=1}^{n} H_{k-1}/k$.

27 Prove the gcd law (6.111) for Fibonacci numbers.

28 The *Lucas number* L_n is defined to be $F_{n+1} + F_{n-1}$. Thus, according to (6.109), we have $F_{2n} = F_n L_n$. Here is a table of the first few values:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13
L_n	2	1	3	4	7	11	18	29	47	76	123	199	322	521

a Use the repertoire method to show that the solution Q_n to the general recurrence

$$Q_0 = \alpha; \qquad Q_1 = \beta; \qquad Q_n = Q_{n-1} + Q_{n-2}, \quad n > 1$$

can be expressed in terms of F_n and L_n.

b Find a closed form for L_n in terms of ϕ and $\hat{\phi}$.

29 Prove Euler's identity for continuants, equation (6.134).

30 Generalize (6.136) to find an expression for the incremented continuant $K(x_1, \ldots, x_{m-1}, x_m + y, x_{m+1}, \ldots, x_n)$, when $1 \leqslant m \leqslant n$.

Homework exercises

31 Find a closed form for the coefficients $\left|{n \atop k}\right|$ in the representation of rising powers by falling powers:

$$x^{\overline{n}} = \sum_k \left|{n \atop k}\right| x^{\underline{k}}, \qquad \text{integer } n \geqslant 0.$$

(For example, $x^{\overline{4}} = x^{\underline{4}} + 12x^{\underline{3}} + 36x^{\underline{2}} + 24x^{\underline{1}}$, hence $\left|{4 \atop 2}\right| = 36$.).

32 In Chapter 5 we obtained the formulas

$$\sum_{k \leqslant m} \binom{n+k}{k} = \binom{n+m+1}{m} \quad \text{and} \quad \sum_{0 \leqslant k \leqslant m} \binom{k}{n} = \binom{m+1}{n+1}$$

by unfolding the recurrence $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ in two ways. What identities appear when the analogous recurrence $\left\{{n \atop k}\right\} = k\left\{{n-1 \atop k}\right\} + \left\{{n-1 \atop k-1}\right\}$ is unwound?

33 Table 264 gives the values of $\left[{n \atop 2}\right]$ and $\left\{{n \atop 2}\right\}$. What are closed forms (not involving Stirling numbers) for the next cases, $\left[{n \atop 3}\right]$ and $\left\{{n \atop 3}\right\}$?

34 What are $\left\langle{-1 \atop k}\right\rangle$ and $\left\langle{-2 \atop k}\right\rangle$, if the basic recursion relation (6.35) is assumed to hold for all integers k and n, and if $\left\langle{n \atop k}\right\rangle = 0$ for all $k < 0$?

35 Prove that, for every $\epsilon > 0$, there exists an integer $n > 1$ (depending on ϵ) such that $H_n \bmod 1 < \epsilon$.

36 Is it possible to stack n bricks in such a way that the topmost brick is not above any point of the bottommost brick, yet a person who weighs the same as 100 bricks can balance on the middle of the top brick without toppling the pile?

37 Express $\sum_{k=1}^{mn}(k \bmod m)/k(k+1)$ in terms of harmonic numbers, assuming that m and n are positive integers. What is the limiting value as $n \to \infty$?

38 Find the indefinite sum $\sum \binom{r}{k}(-1)^k H_k \,\delta k$.

39 Express $\sum_{k=1}^{n} H_k^2$ in terms of n and H_n.

40 Prove that 1979 divides the numerator of $\sum_{k=1}^{1319}(-1)^{k-1}/k$, and give a similar result for 1987. *Hint:* Use Gauss's trick to obtain a sum of fractions whose numerators are 1979. See also exercise 4.

Ah! Those were prime years.

41 Evaluate the sum

$$\sum_k \binom{\lfloor (n+k)/2 \rfloor}{k}$$

in closed form, when n is an integer (possibly negative).

42 If S is a set of integers, let $S+1$ be the "shifted" set $\{x+1 \mid x \in S\}$. How many subsets of $\{1, 2, \ldots, n\}$ have the property that $S \cup (S+1) = \{1, 2, \ldots, n+1\}$?

43 Prove that the infinite sum

$$\begin{aligned} &.1 \\ +&.01 \\ +&.002 \\ +&.0003 \\ +&.00005 \\ +&.000008 \\ +&.0000013 \\ &\vdots \end{aligned}$$

converges to a rational number.

44 Prove the converse of Cassini's identity (6.106): If k and m are integers such that $|m^2 - km - k^2| = 1$, then there is an integer n such that $k = \pm F_n$ and $m = \pm F_{n+1}$.

45 Use the repertoire method to solve the general recurrence

$$X_0 = \alpha; \qquad X_1 = \beta; \qquad X_n = X_{n-1} + X_{n-2} + \gamma n + \delta.$$

46 What are $\cos 36^\circ$ and $\cos 72^\circ$?

47 Show that

$$2^{n-1} F_n = \sum_k \binom{n}{2k+1} 5^k,$$

and use this identity to deduce the values of $F_p \bmod p$ and $F_{p+1} \bmod p$ when p is prime.

48 Prove that zero-valued parameters can be removed from continuant polynomials by collapsing their neighbors together:

$$\begin{aligned} &K_n(x_1, \ldots, x_{m-1}, 0, x_{m+1}, \ldots, x_n) \\ &\quad = K_{n-2}(x_1, \ldots, x_{m-2}, x_{m-1} + x_{m+1}, x_{m+2}, \ldots, x_n), \qquad 1 < m < n. \end{aligned}$$

49 Find the continued fraction representation of the number $\sum_{n \geqslant 1} 2^{-\lfloor n\phi \rfloor}$.

50 Define $f(n)$ for all positive integers n by the recurrence

$$\begin{aligned} f(1) &= 1; \\ f(2n) &= f(n); \\ f(2n+1) &= f(n) + f(n+1). \end{aligned}$$

a For which n is $f(n)$ even?

b Show that $f(n)$ can be expressed in terms of continuants.

Exam problems

51 Let p be a prime number.

a Prove that $\left\{ {p \atop k} \right\} \equiv \left[{p \atop k} \right] \equiv 0 \pmod p$, for $1 < k < p$.

b Prove that $\left[{p-1 \atop k} \right] \equiv 1 \pmod p$, for $1 \leqslant k < p$.

c Prove that $\left\{ {2p-2 \atop p} \right\} \equiv \left[{2p-2 \atop p} \right] \equiv 0 \pmod p$, if $p > 2$.

d Prove that if $p > 3$ we have $\left[{p \atop 2} \right] \equiv 0 \pmod{p^2}$. *Hint:* Consider $p^{\underline{p}}$.

52 Let H_n be written in lowest terms as a_n / b_n.

a Prove that $p \backslash b_n \iff p \not\backslash a_{\lfloor n/p \rfloor}$, if p is prime.

b Find all $n > 0$ such that a_n is divisible by 5.

53 Find a closed form for $\sum_{k=0}^{m}\binom{n}{k}^{-1}(-1)^k H_k$, when $0 \leqslant m \leqslant n$. *Hint:* Exercise 5.42 has the sum without the H_k factor.

54 Let $n > 0$. The purpose of this exercise is to show that the denominator of B_{2n} is the product of all primes p such that $(p-1)\backslash(2n)$.

a Show that $S_m(p) + \big[(p-1)\backslash m\big]$ is a multiple of p, when p is prime and $m > 0$.

b Use the result of part (a) to show that

$$B_{2n} + \sum_{p \text{ prime}} \frac{\big[(p-1)\backslash(2n)\big]}{p} = I_{2n} \quad \text{is an integer.}$$

Hint: It suffices to prove that, if p is any prime, the denominator of the fraction $B_{2n} + \big[(p-1)\backslash(2n)\big]/p$ is not divisible by p.

c Prove that the denominator of B_{2n} is always an odd multiple of 6, and it is equal to 6 for infinitely many n.

55 Prove (6.70) as a corollary of a more general identity, by summing

$$\sum_{0\leqslant k<n}\binom{k}{m}\binom{x+k}{k}$$

and differentiating with respect to x.

56 Evaluate $\sum_{k\neq m}\binom{n}{k}(-1)^k k^{n+1}/(k-m)$ in closed form as a function of the integers m and n. (The sum is over all integers k except for the value $k = m$.)

57 The "wraparound binomial coefficients of order 5" are defined by

$$\left(\!\!\binom{n}{k}\!\!\right) = \left(\!\!\binom{n-1}{k}\!\!\right) + \left(\!\!\binom{n-1}{(k-1) \bmod 5}\!\!\right), \qquad n > 0,$$

and $\left(\!\binom{0}{k}\!\right) = [k{=}0]$. Let Q_n be the difference between the largest and smallest of these numbers in row n:

$$Q_n = \max_{0\leqslant k<5}\left(\!\!\binom{n}{k}\!\!\right) - \min_{0\leqslant k<5}\left(\!\!\binom{n}{k}\!\!\right).$$

Find and prove a relation between Q_n and the Fibonacci numbers.

58 Find closed forms for $\sum_{n\geqslant 0} F_n^2 z^n$ and $\sum_{n\geqslant 0} F_n^3 z^n$. What do you deduce about the quantity $F_{n+1}^3 - 4F_n^3 - F_{n-1}^3$?

59 Prove that if m and n are positive integers, there exists an integer x such that $F_x \equiv m \pmod{3^n}$.

60 Find all positive integers n such that either $F_n + 1$ or $F_n - 1$ is a prime number.

61 Prove the identity

$$\sum_{k=0}^{n} \frac{1}{F_{2^k}} = 3 - \frac{F_{2^n-1}}{F_{2^n}}, \qquad \text{integer } n \geqslant 1.$$

What is $\sum_{k=0}^{n} 1/F_{3\cdot 2^k}$?

62 Let $A_n = \phi^n + \phi^{-n}$ and $B_n = \phi^n - \phi^{-n}$.

a Find constants α and β such that $A_n = \alpha A_{n-1} + \beta A_{n-2}$ and $B_n = \alpha B_{n-1} + \beta B_{n-2}$ for all $n \geqslant 0$.

b Express A_n and B_n in terms of F_n and L_n (see exercise 28).

c Prove that $\sum_{k=1}^{n} 1/(F_{2k+1} + 1) = B_n/A_{n+1}$.

d Find a closed form for $\sum_{k=1}^{n} 1/(F_{2k+1} - 1)$.

Bonus problems *Bogus problems*

63 How many permutations $\pi_1\pi_2\ldots\pi_n$ of $\{1, 2, \ldots, n\}$ have exactly k indices j such that

a $\pi_i < \pi_j$ for all $i < j$? (Such j are called "left-to-right maxima.")

b $\pi_j > j$? (Such j are called "excedances.")

64 What is the denominator of $\left[{1/2 \atop 1/2-n}\right]$, when this fraction is reduced to lowest terms?

65 Prove the identity

$$\int_0^1 \ldots \int_0^1 f\big(\lfloor x_1 + \cdots + x_n \rfloor\big)\, dx_1 \ldots dx_n = \sum_k \left\langle {n \atop k} \right\rangle \frac{f(k)}{n!}.$$

66 What is $\sum_k (-1)^k \left\langle {n \atop k} \right\rangle$, the nth alternating row sum of Euler's triangle?

67 Prove that

$$\sum_k \left\{ {n+1 \atop k+1} \right\} \binom{n-k}{m-k} (-1)^{m-k} k! = \left\langle {n \atop m} \right\rangle.$$

68 Show that $\left\langle\!\left\langle {n \atop 1} \right\rangle\!\right\rangle = 2\left\langle {n \atop 1} \right\rangle$, and find a closed form for $\left\langle\!\left\langle {n \atop 2} \right\rangle\!\right\rangle$.

69 Find a closed form for $\sum_{k=1}^{n} k^2 H_{n+k}$.

70 Show that the complex harmonic numbers of exercise 22 have the power series expansion $H_z = \sum_{n\geqslant 2} (-1)^n H_\infty^{(n)} z^{n-1}$.

71 Prove that the generalized factorial of equation (5.83) can be written

$$\prod_{k\geqslant 1} \Big(1 + \frac{z}{k}\Big) e^{-z/k} = \frac{e^{\gamma z}}{z!},$$

by considering the limit as $n \to \infty$ of the first n factors of this infinite product. Show that $\frac{d}{dz}(z!)$ is related to the general harmonic numbers of exercise 22.

72 Prove that the tangent function has the power series (6.92), and find the corresponding series for $z/\sin z$ and $\ln\bigl((\tan z)/z\bigr)$.

73 Prove that $z\cot z$ is equal to

$$\frac{z}{2^n}\cot\frac{z}{2^n}-\frac{z}{2^n}\tan\frac{z}{2^n}+\sum_{k=1}^{2^n-1}\frac{z}{2^n}\left(\cot\frac{z+k\pi}{2^n}+\cot\frac{z-k\pi}{2^n}\right),$$

for all integers $n \geqslant 1$, and show that the limit of the kth summand is $2z^2/(z^2-k^2\pi^2)$ for fixed k as $n\to\infty$.

74 Find a relation between the numbers $T_n(1)$ and the coefficients of $1/\cos z$.

75 Prove that the tangent numbers and the coefficients of $1/\cos z$ appear at the edges of the infinite triangle that begins as follows:

$$\begin{array}{ccccccccccccc}
 & & & & & & 1 & & & & & & \\
 & & & & & 0 & & 1 & & & & & \\
 & & & & 1 & & 1 & & 0 & & & & \\
 & & & 0 & & 1 & & 2 & & 2 & & & \\
 & & 5 & & 5 & & 4 & & 2 & & 0 & & \\
 & 0 & & 5 & & 10 & & 14 & & 16 & & 16 & \\
61 & & 61 & & 56 & & 46 & & 32 & & 16 & & 0
\end{array}$$

Each row contains partial sums of the previous row, going alternately left-to-right and right-to-left. *Hint:* Consider the coefficients of the power series $(\sin z+\cos z)/\cos(w+z)$.

76 Find a closed form for the sum

$$\sum_k (-1)^k \left\{ {n \atop k} \right\} 2^{n-k} k!\,,$$

and show that it is zero when n is even.

77 When m and n are integers, $n \geqslant 0$, the value of $\sigma_n(m)$ is given by (6.48) if $m<0$, by (6.49) if $m \geqslant n$, and by (6.101) if $m=0$. Show that in the remaining cases we have

$$\sigma_n(m) \;=\; \frac{(-1)^{m+n-1}}{m!\,(n-m)!}\sum_{k=0}^{m-1}\left[{m \atop m-k}\right]\frac{B_{n-k}}{n-k}, \quad \text{integer } n>m>0.$$

78 Prove the following relation that connects Stirling numbers, Bernoulli numbers, and Catalan numbers:

$$\sum_{k=0}^{n}\left\{ {n+k \atop k} \right\}\binom{2n}{n+k}\frac{(-1)^k}{k+1} \;=\; B_n\binom{2n}{n}\frac{1}{n+1}\,.$$

79 Show that the four chessboard pieces of the $64=65$ paradox can also be reassembled to prove that $64=63$.

80 A sequence defined by the recurrence $A_1 = x$, $A_2 = y$, and $A_n = A_{n-1} + A_{n-2}$ has $A_m = 1000000$ for some m. What positive integers x and y make m as large as possible?

81 The text describes a way to change a formula involving $F_{n\pm k}$ to a formula that involves F_n and F_{n+1} only. Therefore it's natural to wonder if two such "reduced" formulas can be equal when they aren't identical in form. Let $P(x, y)$ be a polynomial in x and y with integer coefficients. Find a necessary and sufficient condition that $P(F_{n+1}, F_n) = 0$ for all $n \geqslant 0$.

82 Explain how to add positive integers, working entirely in the Fibonacci number system.

83 Is it possible that a sequence $\langle A_n \rangle$ satisfying the Fibonacci recurrence $A_n = A_{n-1} + A_{n-2}$ can contain no prime numbers, if A_0 and A_1 are relatively prime?

84 Let m and n be odd, positive integers. Find closed forms for

$$S^+_{m,n} = \sum_{k\geqslant 0} \frac{1}{F_{2mk+n} + F_m}; \qquad S^-_{m,n} = \sum_{k\geqslant 0} \frac{1}{F_{2mk+n} - F_m}.$$

Hint: The sums in exercise 62 are $S^+_{1,3} - S^+_{1,2n+3}$ and $S^-_{1,3} - S^-_{1,2n+3}$.

85 Characterize all N such that the Fibonacci residues $F_n \bmod N$ for $n \geqslant 0$ form the complete set $\{0, 1, \ldots, N-1\}$. (See exercise 59.)

86 Let C_1, C_2, ... be a sequence of nonzero integers such that

$$\gcd(C_m, C_n) = C_{\gcd(m,n)}$$

for all positive integers m and n. Prove that the generalized binomial coefficients

$$\binom{n}{k}_{\mathcal{C}} = \frac{C_n C_{n-1} \ldots C_{n-k+1}}{C_k C_{k-1} \ldots C_1}$$

are all integers. (In particular, the "Fibonomial coefficients" formed in this way from Fibonacci numbers are integers, by (6.111).)

87 Show that continuant polynomials appear in the matrix product

$$\begin{pmatrix} 0 & 1 \\ 1 & x_1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & x_2 \end{pmatrix} \cdots \begin{pmatrix} 0 & 1 \\ 1 & x_n \end{pmatrix}$$

and in the determinant

$$\det\begin{pmatrix} x_1 & 1 & 0 & 0 & \dots & 0 \\ -1 & x_2 & 1 & 0 & & 0 \\ 0 & -1 & x_3 & 1 & & \vdots \\ \vdots & & -1 & & & \\ & & & & \ddots & 1 \\ 0 & 0 & \dots & & -1 & x_n \end{pmatrix}.$$

88 Generalizing (6.146), find a continued fraction related to the generating function $\sum_{n\geqslant 1} z^{\lfloor n\alpha\rfloor}$, when α is any positive irrational number.

89 Let α be an irrational number in $(0\,..\,1)$ and let a_1, a_2, a_3, ... be the partial quotients in its continued fraction representation. Show that $\bigl|D(\alpha,n)\bigr| < 2$ when $n = K(a_1,\dots,a_m)$, where D is the discrepancy defined in Chapter 3.

90 Let Q_n be the largest denominator on level n of the Stern–Brocot tree. (Thus $\langle Q_0, Q_1, Q_2, Q_3, Q_4, \dots\rangle = \langle 1,2,3,5,8,\dots\rangle$ according to the diagram in Chapter 4.) Prove that $Q_n = F_{n+2}$.

Research problems

91 What is the best way to extend the definition of $\left\{{n \atop k}\right\}$ to arbitrary real values of n and k?

92 Let H_n be written in lowest terms as a_n/b_n, as in exercise 52.

a Are there infinitely many n with $11\backslash a_n$?

b Are there infinitely many n with $b_n = \operatorname{lcm}(1,2,\dots,n)$? (Two such values are $n = 250$ and $n = 1000$.)

93 Prove that γ and e^γ are irrational.

94 Develop a general theory of the solutions to the two-parameter recurrence

$$\begin{aligned}\left|{n \atop k}\right| &= (\alpha n + \beta k + \gamma)\left|{n-1 \atop k}\right| \\ &\qquad + (\alpha' n + \beta' k + \gamma')\left|{n-1 \atop k-1}\right| + [n{=}k{=}0]\,, \quad \text{for } n,k \geqslant 0,\end{aligned}$$

assuming that $\left|{n \atop k}\right| = 0$ when $n < 0$ or $k < 0$. (Binomial coefficients, Stirling numbers, Eulerian numbers, and the sequences of exercises 17 and 31 are special cases.) What special values $(\alpha,\beta,\gamma,\alpha',\beta',\gamma')$ yield "fundamental solutions" in terms of which the general solution can be expressed?

95 Find an efficient way to extend the Gosper-Zeilberger algorithm from hypergeometric terms to terms that may involve Stirling numbers.

7

Generating Functions

THE MOST POWERFUL WAY to deal with sequences of numbers, as far as anybody knows, is to manipulate infinite series that "generate" those sequences. We've learned a lot of sequences and we've seen a few generating functions; now we're ready to explore generating functions in depth, and to see how remarkably useful they are.

7.1 DOMINO THEORY AND CHANGE

Generating functions are important enough, and for many of us new enough, to justify a relaxed approach as we begin to look at them more closely. So let's start this chapter with some fun and games as we try to develop our intuitions about generating functions. We will study two applications of the ideas, one involving dominoes and the other involving coins.

How many ways T_n are there to completely cover a $2 \times n$ rectangle with 2×1 dominoes? We assume that the dominoes are identical (either because they're face down, or because someone has rendered them indistinguishable, say by painting them all red); thus only their orientations — vertical or horizontal — matter, and we can imagine that we're working with domino-shaped tiles. For example, there are three tilings of a 2×3 rectangle, namely , , and ; so $T_3 = 3$.

"Let me count the ways."
— E. B. Browning

To find a closed form for general T_n we do our usual first thing, look at small cases. When $n = 1$ there's obviously just one tiling, ; and when $n = 2$ there are two, and .

How about when $n = 0$; how many tilings of a 2×0 rectangle are there? It's not immediately clear what this question means, but we've seen similar situations before: There is one permutation of zero objects (namely the empty permutation), so $0! = 1$. There is one way to choose zero things from n things (namely to choose nothing), so $\binom{n}{0} = 1$. There is one way to partition the empty set into zero nonempty subsets, but there are no such ways to partition a nonempty set; so $\left\{ {n \atop 0} \right\} = [n = 0]$. By such reasoning we can conclude that

there's just one way to tile a 2×0 rectangle with dominoes, namely to use no dominoes; therefore $T_0 = 1$. (This spoils the simple pattern $T_n = n$ that holds when $n = 1$, 2, and 3; but that pattern was probably doomed anyway, since T_0 wants to be 1 according to the logic of the situation.) A proper understanding of the null case turns out to be useful whenever we want to solve an enumeration problem.

Let's look at one more small case, $n = 4$. There are two possibilities for tiling the left edge of the rectangle — we put either a vertical domino or two horizontal dominoes there. If we choose a vertical one, the partial solution is and the remaining 2×3 rectangle can be covered in T_3 ways. If we choose two horizontals, the partial solution can be completed in T_2 ways. Thus $T_4 = T_3 + T_2 = 5$. (The five tilings are , , , , and .)

We now know the first five values of T_n:

n	0	1	2	3	4
T_n	1	1	2	3	5

These look suspiciously like the Fibonacci numbers, and it's not hard to see why: The reasoning we used to establish $T_4 = T_3 + T_2$ easily generalizes to $T_n = T_{n-1} + T_{n-2}$, for $n \geqslant 2$. Thus we have the same recurrence here as for the Fibonacci numbers, except that the initial values $T_0 = 1$ and $T_1 = 1$ are a little different. But these initial values are the consecutive Fibonacci numbers F_1 and F_2, so the T's are just Fibonacci numbers shifted up one place:

$$T_n \;=\; F_{n+1}\,, \qquad \text{for } n \geqslant 0.$$

(We consider this to be a closed form for T_n, because the Fibonacci numbers are important enough to be considered "known." Also, F_n itself has a closed form (6.123) in terms of algebraic operations.) Notice that this equation confirms the wisdom of setting $T_0 = 1$.

But what does all this have to do with generating functions? Well, we're about to get to that — there's another way to figure out what T_n is. This new way is based on a bold idea. Let's consider the "sum" of all possible $2 \times n$ tilings, for all $n \geqslant 0$, and call it T:

To boldly go where no tiling has gone before.

$$T \;=\; \mathrm{I} + \square + \square\square + \square\square\square + \square\square\square\square + \square\square\square\square\square + \cdots\,. \tag{7.1}$$

(The first term '|' on the right stands for the null tiling of a 2×0 rectangle.) This sum T represents lots of information. It's useful because it lets us prove things about T as a whole rather than forcing us to prove them (by induction) about its individual terms.

The terms of this sum stand for tilings, which are combinatorial objects. We won't be fussy about what's considered legal when infinitely many tilings

are added together; everything can be made rigorous, but our goal right now is to expand our consciousness beyond conventional algebraic formulas.

We've added the patterns together, and we can also multiply them — by juxtaposition. For example, we can multiply the tilings $\square$ and $\boxminus$ to get the new tiling $\boxminus\!\!\square$. But notice that multiplication is not commutative; that is, the order of multiplication counts: $\boxminus\!\!\square$ is different from $\square\!\!\boxminus$.

Using this notion of multiplication it's not hard to see that the null tiling plays a special role — it is the multiplicative identity. For instance, $| \times \boxminus = \boxminus \times | = \boxminus$.

Now we can use domino arithmetic to manipulate the infinite sum T:

$$\begin{aligned} \mathsf{T} &= | + \square + \square\!\!\square + \boxminus + \square\!\!\square\!\!\square + \boxminus\!\!\square + \square\!\!\boxminus + \cdots \\ &= | + \square(| + \square + \square\!\!\square + \boxminus + \cdots) + \boxminus(| + \square + \square\!\!\square + \boxminus + \cdots) \\ &= | + \square\mathsf{T} + \boxminus\mathsf{T}. \end{aligned} \tag{7.2}$$

Every valid tiling occurs exactly once in each right side, so what we've done is reasonable even though we're ignoring the cautions in Chapter 2 about "absolute convergence." The bottom line of this equation tells us that everything in T is either the null tiling, or is a vertical tile followed by something else in T, or is two horizontal tiles followed by something else in T.

I have a gut feeling that these sums must converge, as long as the dominoes are small enough.

So now let's try to solve the equation for T. Replacing the T on the left by $|\mathsf{T}$ and subtracting the last two terms on the right from both sides of the equation, we get

$$(| - \square - \boxminus)\mathsf{T} = |. \tag{7.3}$$

For a consistency check, here's an expanded version:

$$\begin{array}{ccccccccccccccc} & | & + & \square & + & \square\!\!\square & + & \boxminus & + & \square\!\!\square\!\!\square & + & \boxminus\!\!\square & + & \square\!\!\boxminus & + \cdots \\ - & \square & - & \square\!\!\square & - & \square\!\!\square\!\!\square & - & \boxminus\!\!\square & - & \square\!\!\square\!\!\square\!\!\square & - & \square\!\!\boxminus\!\!\square & - & \boxminus\!\!\square\!\!\boxminus & - \cdots \\ - & \boxminus & - & \boxminus\!\!\square & - & \boxminus\!\!\square\!\!\square & - & \boxminus\!\!\boxminus\!\!\square & - & \boxminus\!\!\square\!\!\square\!\!\square & - & \boxminus\!\!\boxminus\!\!\boxminus & - & \boxminus\!\!\boxminus\!\!\square\!\!\boxminus & - \cdots \\ \hline & | & & & & & & & & & & & & & \end{array}$$

Every term in the top row, except the first, is cancelled by a term in either the second or third row, so our equation is correct.

So far it's been fairly easy to make combinatorial sense of the equations we've been working with. Now, however, to get a compact expression for T we cross a combinatorial divide. With a leap of algebraic faith we divide both sides of equation (7.3) by $| - \square - \boxminus$ to get

$$\mathsf{T} = \frac{|}{| - \square - \boxminus}. \tag{7.4}$$

(Multiplication isn't commutative, so we're on the verge of cheating, by not distinguishing between left and right division. In our application it doesn't matter, because $|$ commutes with everything. But let's not be picky, unless our wild ideas lead to paradoxes.)

The next step is to expand this fraction as a power series, using the rule

$$\frac{1}{1-z} = 1 + z + z^2 + z^3 + \cdots .$$

The null tiling $|$, which is the multiplicative identity for our combinatorial arithmetic, plays the part of 1, the usual multiplicative identity; and $\square + \boxminus$ plays z. So we get the expansion

$$\begin{aligned}\frac{|}{|-\square-\boxminus} &= |+(\square+\boxminus)+(\square+\boxminus)^2+(\square+\boxminus)^3+\cdots\\ &= |+(\square+\boxminus)+(\square\square+\square\boxminus+\boxminus\square+\boxminus\boxminus)\\ &\qquad +(\square\square\square+\square\square\boxminus+\square\boxminus\square+\square\boxminus\boxminus+\boxminus\square\square+\boxminus\square\boxminus+\boxminus\boxminus\square+\boxminus\boxminus\boxminus)+\cdots .\end{aligned}$$

This is T, but the tilings are arranged in a different order than we had before. Every tiling appears exactly once in this sum; for example, $\boxminus\boxminus\square\square\boxminus\boxminus$ appears in the expansion of $(\square+\boxminus)^7$.

We can get useful information from this infinite sum by compressing it down, ignoring details that are not of interest. For example, we can imagine that the patterns become unglued and that the individual dominoes commute with each other; then a term like $\boxminus\boxminus\square\square\boxminus\boxminus$ becomes $\square^4\boxminus^6$, because it contains four verticals and six horizontals. Collecting like terms gives us the series

$$T = |+\square+\square^2+\boxminus^2+\square^3+2\square\boxminus^2+\square^4+3\square^2\boxminus^2+\boxminus^4+\cdots .$$

The $2\square\boxminus^2$ here represents the two terms of the old expansion, $\square\boxminus$ and $\boxminus\square$, that have one vertical and two horizontal dominoes; similarly $3\square^2\boxminus^2$ represents the three terms $\square\square\boxminus$, $\square\boxminus\square$, and $\boxminus\square\square$. We're essentially treating $\square$ and $\boxminus$ as ordinary (commutative) variables.

We can find a closed form for the coefficients in the commutative version of T by using the binomial theorem:

$$\begin{aligned}\frac{|}{|-(\square+\boxminus^2)} &= |+(\square+\boxminus^2)+(\square+\boxminus^2)^2+(\square+\boxminus^2)^3+\cdots\\ &= \sum_{k\geqslant 0}(\square+\boxminus^2)^k\\ &= \sum_{j,k\geqslant 0}\binom{k}{j}\square^j\boxminus^{2k-2j}\\ &= \sum_{j,m\geqslant 0}\binom{j+m}{j}\square^j\boxminus^{2m} . \qquad (7.5)\end{aligned}$$

(The last step replaces $k-j$ by m; this is legal because we have $\binom{k}{j} = 0$ when $0 \leqslant k < j$.) We conclude that $\binom{j+m}{j}$ is the number of ways to tile a $2\times(j+2m)$ rectangle with j vertical dominoes and $2m$ horizontal dominoes. For example, we recently looked at the 2×10 tiling ⊟⊟⊟⊟, which involves four verticals and six horizontals; there are $\binom{4+3}{4} = 35$ such tilings in all, so one of the terms in the commutative version of T is $35\square^4\square^6$.

We can suppress even more detail by ignoring the orientation of the dominoes. Suppose we don't care about the horizontal/vertical breakdown; we only want to know about the total number of $2 \times n$ tilings. (This, in fact, is the number T_n we started out trying to discover.) We can collect the necessary information by simply substituting a single quantity, z, for $\square$ and $\square$. And we might as well also replace $|$ by 1, getting

Now I'm disoriented.

$$T = \frac{1}{1-z-z^2}\,. \tag{7.6}$$

This is the generating function (6.117) for Fibonacci numbers, except for a missing factor of z in the numerator; so we conclude that the coefficient of z^n in T is F_{n+1}.

The compact representations $|/(|-\square-\boxminus)$, $|/(|-\square-\square^2)$, and $1/(1-z-z^2)$ that we have deduced for T are called *generating functions*, because they generate the coefficients of interest.

Incidentally, our derivation implies that the number of $2 \times n$ domino tilings with exactly m pairs of horizontal dominoes is $\binom{n-m}{m}$. (This follows because there are $j = n - 2m$ vertical dominoes, hence there are

$$\binom{j+m}{j} = \binom{j+m}{m} = \binom{n-m}{m}$$

ways to do the tiling according to our formula.) We observed in Chapter 6 that $\binom{n-m}{m}$ is the number of Morse code sequences of length n that contain m dashes; in fact, it's easy to see that $2\times n$ domino tilings correspond directly to Morse code sequences. (The tiling ⊟⊟⊟⊟ corresponds to '•−−−••−•'.) Thus domino tilings are closely related to the continuant polynomials we studied in Chapter 6. It's a small world.

We have solved the T_n problem in two ways. The first way, guessing the answer and proving it by induction, was easier; the second way, using infinite sums of domino patterns and distilling out the coefficients of interest, was fancier. But did we use the second method only because it was amusing to play with dominoes as if they were algebraic variables? No; the real reason for introducing the second way was that the infinite-sum approach is a lot more powerful. The second method applies to many more problems, because it doesn't require us to make magic guesses.

Let's generalize up a notch, to a problem where guesswork will be beyond us. How many ways U_n are there to tile a $3 \times n$ rectangle with dominoes?

The first few cases of this problem tell us a little: The null tiling gives $U_0 = 1$. There is no valid tiling when $n = 1$, since a 2×1 domino doesn't fill a 3×1 rectangle, and since there isn't room for two. The next case, $n = 2$, can easily be done by hand; there are three tilings, , , and , so $U_2 = 3$. (Come to think of it we already knew this, because the previous problem told us that $T_3 = 3$; the number of ways to tile a 3×2 rectangle is the same as the number to tile a 2×3.) When $n = 3$, as when $n = 1$, there are no tilings. We can convince ourselves of this either by making a quick exhaustive search or by looking at the problem from a higher level: The area of a 3×3 rectangle is odd, so we can't possibly tile it with dominoes whose area is even. (The same argument obviously applies to any odd n.) Finally, when $n = 4$ there seem to be about a dozen tilings; it's difficult to be sure about the exact number without spending a lot of time to guarantee that the list is complete.

So let's try the infinite-sum approach that worked last time:

$U =$ $+ \cdots .$ (7.7)

Every non-null tiling begins with either or or ; but unfortunately the first two of these three possibilities don't simply factor out and leave us with U again. The sum of all terms in U that begin with can, however, be written as V, where

$V =$ $+ \cdots$

is the sum of all domino tilings of a mutilated $3 \times n$ rectangle that has its lower left corner missing. Similarly, the terms of U that begin with can be written Λ, where

$\Lambda =$ $+ \cdots$

consists of all rectangular tilings lacking their upper left corner. The series Λ is a mirror image of V. These factorizations allow us to write

$U =$.

And we can factor V and Λ as well, because such tilings can begin in only two ways:

$V =$ $U + $ V,

$\Lambda =$ $U + $ Λ.

Now we have three equations in three unknowns (U, V, and Λ). We can solve them by first solving for V and Λ in terms of U, then plugging the results into the equation for U:

$$V = (|-\square\!\!\square)^{-1}\square U, \qquad \Lambda = (|-\square\!\!\square)^{-1}\square U;$$
$$U = | + \square\!\!\square(|-\square\!\!\square)^{-1}\square U + \square\!\!\square(|-\square\!\!\square)^{-1}\square U + \square\!\!\square U.$$

And the final equation can be solved for U, giving the compact formula

$$U = \frac{|}{|-\square\!\!\square(|-\square\!\!\square)^{-1}\square - \square\!\!\square(|-\square\!\!\square)^{-1}\square - \square\!\!\square}. \tag{7.8}$$

I learned in another class about "regular expressions." If I'm not mistaken, we can write $U = (\square\!\!\square^*\square + \square\!\!\square^*\square + \square\!\!\square)^*$ *in the language of regular expressions; so there must be some connection between regular expressions and generating functions.*

This expression defines the infinite sum U, just as (7.4) defines T.

The next step is to go commutative. Everything simplifies beautifully when we detach all the dominoes and use only powers of $\square$ and $\square$:

$$\begin{aligned}
U &= \frac{1}{1-\square^2\square(1-\square^3)^{-1}-\square^2\square(1-\square^3)^{-1}-\square^3} \\
&= \frac{1-\square^3}{(1-\square^3)^2-2\square^2\square} \\
&= \frac{(1-\square^3)^{-1}}{1-2\square^2\square(1-\square^3)^{-2}} \\
&= \frac{1}{1-\square^3}+\frac{2\square^2\square}{(1-\square^3)^3}+\frac{4\square^4\square^2}{(1-\square^3)^5}+\frac{8\square^6\square^3}{(1-\square^3)^7}+\cdots \\
&= \sum_{k\geqslant 0}\frac{2^k\square^{2k}\square^k}{(1-\square^3)^{2k+1}} \\
&= \sum_{k,m\geqslant 0}\binom{m+2k}{m}2^k\square^{2k}\square^{k+3m}.
\end{aligned}$$

(This derivation deserves careful scrutiny. The last step uses the formula $(1-w)^{-2k-1}=\sum_m\binom{m+2k}{m}w^m$, identity (5.56).) Let's take a good look at the bottom line to see what it tells us. First, it says that every $3\times n$ tiling uses an even number of vertical dominoes. Moreover, if there are $2k$ verticals, there must be at least k horizontals, and the total number of horizontals must be $k+3m$ for some $m\geqslant 0$. Finally, the number of possible tilings with $2k$ verticals and $k+3m$ horizontals is exactly $\binom{m+2k}{m}2^k$.

We now are able to analyze the 3×4 tilings that left us doubtful when we began looking at the $3\times n$ problem. When $n=4$ the total area is 12, so we need six dominoes altogether. There are $2k$ verticals and $k+3m$ horizontals,

for some k and m; hence $2k + k + 3m = 6$. In other words, $k + m = 2$. If we use no verticals, then $k = 0$ and $m = 2$; the number of possibilities is $\binom{2+0}{2}2^0 = 1$. (This accounts for the tiling ⊞.) If we use two verticals, then $k = 1$ and $m = 1$; there are $\binom{1+2}{1}2^1 = 6$ such tilings. And if we use four verticals, then $k = 2$ and $m = 0$; there are $\binom{0+4}{0}2^2 = 4$ such tilings, making a total of $U_4 = 11$. In general if n is even, this reasoning shows that $k + m = \frac{1}{2}n$, hence $\binom{m+2k}{m} = \binom{n/2+k}{n/2-k}$ and the total number of $3 \times n$ tilings is

$$U_n = \sum_k \binom{n/2+k}{n/2-k} 2^k = \sum_m \binom{n-m}{m} 2^{n/2-m}. \tag{7.9}$$

As before, we can also substitute z for both ▯ and ▭, getting a generating function that doesn't discriminate between dominoes of particular persuasions. The result is

$$U = \frac{1}{1 - z^3(1-z^3)^{-1} - z^3(1-z^3)^{-1} - z^3} = \frac{1-z^3}{1-4z^3+z^6}. \tag{7.10}$$

If we expand this quotient into a power series, we get

$$U = 1 + U_2 z^3 + U_4 z^6 + U_6 z^9 + U_8 z^{12} + \cdots,$$

a generating function for the numbers U_n. (There's a curious mismatch between subscripts and exponents in this formula, but it is easily explained. The coefficient of z^9, for example, is U_6, which counts the tilings of a 3×6 rectangle. This is what we want, because every such tiling contains nine dominoes.)

We could proceed to analyze (7.10) and get a closed form for the coefficients, but it's better to save that for later in the chapter after we've gotten more experience. So let's divest ourselves of dominoes for the moment and proceed to the next advertised problem, "change."

How many ways are there to pay 50 cents? We assume that the payment must be made with pennies ①, nickels ⑤, dimes ⑩, quarters ㉕, and half-dollars ㊿. George Pólya [298] popularized this problem by showing that it can be solved with generating functions in an instructive way.

Ah yes, I remember when we had half-dollars.

Let's set up infinite sums that represent all possible ways to give change, just as we tackled the domino problems by working with infinite sums that represent all possible domino patterns. It's simplest to start by working with fewer varieties of coins, so let's suppose first that we have nothing but pennies. The sum of all ways to leave some number of pennies (but just pennies) in change can be written

$$\begin{aligned} P &= \not{1} + ① + ①① + ①①① + ①①①① + \cdots \\ &= \not{1} + ① + ①^2 + ①^3 + ①^4 + \cdots. \end{aligned}$$

The first term stands for the way to leave no pennies, the second term stands for one penny, then two pennies, three pennies, and so on. Now if we're allowed to use both pennies and nickels, the sum of all possible ways is

$$\begin{aligned} N &= P + ⑤P + ⑤⑤P + ⑤⑤⑤P + ⑤⑤⑤⑤P + \cdots \\ &= (\not{5} + ⑤ + ⑤^2 + ⑤^3 + ⑤^4 + \cdots)P, \end{aligned}$$

since each payment has a certain number of nickels chosen from the first factor and a certain number of pennies chosen from P. (Notice that N is *not* the sum $\not{1} + ① + ⑤ + (① + ⑤)^2 + (① + ⑤)^3 + \cdots$, because such a sum includes many types of payment more than once. For example, the term $(① + ⑤)^2 = ①① + ①⑤ + ⑤① + ⑤⑤$ treats ①⑤ and ⑤① as if they were different, but we want to list each set of coins only once without respect to order.)

Similarly, if dimes are permitted as well, we get the infinite sum

$$D = (\not{10} + ⑩ + ⑩^2 + ⑩^3 + ⑩^4 + \cdots)N,$$

which includes terms like $⑩^3⑤^3①^5 = ⑩⑩⑩⑤⑤⑤①①①①①$ when it is expanded in full. Each of these terms is a different way to make change.

Coins of the realm. Adding quarters and then half-dollars to the realm of possibilities gives

$$\begin{aligned} Q &= (\not{25} + ㉕ + ㉕^2 + ㉕^3 + ㉕^4 + \cdots)D; \\ C &= (\not{50} + ㊿ + ㊿^2 + ㊿^3 + ㊿^4 + \cdots)Q. \end{aligned}$$

Our problem is to find the number of terms in C worth exactly 50¢.

A simple trick solves this problem nicely: We can replace ① by z, ⑤ by z^5, ⑩ by z^{10}, ㉕ by z^{25}, and ㊿ by z^{50}. Then each term is replaced by z^n, where n is the monetary value of the original term. For example, the term ㊿⑩⑤⑤① becomes $z^{50+10+5+5+1} = z^{71}$. The four ways of paying 13 cents, namely $⑩①^3$, $⑤①^8$, $⑤^2①^3$, and $①^{13}$, each reduce to z^{13}; hence the coefficient of z^{13} will be 4 after the z-substitutions are made.

Let P_n, N_n, D_n, Q_n, and C_n be the numbers of ways to pay n cents when we're allowed to use coins that are worth at most 1, 5, 10, 25, and 50 cents, respectively. Our analysis tells us that these are the coefficients of z^n in the respective power series

$$\begin{aligned} P &= 1 + z + z^2 + z^3 + z^4 + \cdots, \\ N &= (1 + z^5 + z^{10} + z^{15} + z^{20} + \cdots)P, \\ D &= (1 + z^{10} + z^{20} + z^{30} + z^{40} + \cdots)N, \\ Q &= (1 + z^{25} + z^{50} + z^{75} + z^{100} + \cdots)D, \\ C &= (1 + z^{50} + z^{100} + z^{150} + z^{200} + \cdots)Q. \end{aligned}$$

Obviously $P_n = 1$ for all $n \geqslant 0$. And a little thought proves that we have $N_n = \lfloor n/5 \rfloor + 1$: To make n cents out of pennies and nickels, we must choose either 0 or 1 or ... or $\lfloor n/5 \rfloor$ nickels, after which there's only one way to supply the requisite number of pennies. Thus P_n and N_n are simple; but the values of D_n, Q_n, and C_n are increasingly more complicated.

How many pennies are there, really? If n is greater than, say, 10^{10}, I bet that $P_n = 0$ in the "real world."

One way to deal with these formulas is to realize that $1 + z^m + z^{2m} + \cdots$ is just $1/(1 - z^m)$. Thus we can write

$$\begin{aligned} P &= 1/(1-z)\,, \\ N &= P/(1-z^5)\,, \\ D &= N/(1-z^{10})\,, \\ Q &= D/(1-z^{25})\,, \\ C &= Q/(1-z^{50})\,. \end{aligned}$$

Multiplying by the denominators, we have

$$\begin{aligned} (1-z)\,P &= 1\,, \\ (1-z^5)\,N &= P\,, \\ (1-z^{10})\,D &= N\,, \\ (1-z^{25})\,Q &= D\,, \\ (1-z^{50})\,C &= Q\,. \end{aligned}$$

Now we can equate coefficients of z^n in these equations, getting recurrence relations from which the desired coefficients can quickly be computed:

$$\begin{aligned} P_n &= P_{n-1} + [n{=}0]\,, \\ N_n &= N_{n-5} + P_n\,, \\ D_n &= D_{n-10} + N_n\,, \\ Q_n &= Q_{n-25} + D_n\,, \\ C_n &= C_{n-50} + Q_n\,. \end{aligned}$$

For example, the coefficient of z^n in $D = (1 - z^{25})Q$ is equal to $Q_n - Q_{n-25}$; so we must have $Q_n - Q_{n-25} = D_n$, as claimed.

We could unfold these recurrences and find, for example, that $Q_n = D_n + D_{n-25} + D_{n-50} + D_{n-75} + \cdots$, stopping when the subscripts get negative. But the non-iterated form is convenient because each coefficient is computed with just one addition, as in Pascal's triangle.

Let's use the recurrences to find C_{50}. First, $C_{50} = C_0 + Q_{50}$; so we want to know Q_{50}. Then $Q_{50} = Q_{25} + D_{50}$, and $Q_{25} = Q_0 + D_{25}$; so we also want to know D_{50} and D_{25}. These D_n depend in turn on D_{40}, D_{30}, D_{20}, D_{15}, D_{10}, D_5, and on N_{50}, N_{45}, ..., N_5. A simple calculation therefore suffices

to determine all the necessary coefficients:

n	0	5	10	15	20	25	30	35	40	45	50
P_n	1	1	1	1	1	1	1	1	1	1	1
N_n	1	2	3	4	5	6	7	8	9	10	11
D_n	1	2	4	6	9	12	16		25		36
Q_n	1					13					49
C_n	1										50

The final value in the table gives us our answer, C_{50}: There are exactly 50 ways to leave a 50-cent tip.

(Not counting the option of charging the tip to a credit card.)

How about a closed form for C_n? Multiplying the equations together gives us the compact expression

$$C = \frac{1}{1-z}\,\frac{1}{1-z^5}\,\frac{1}{1-z^{10}}\,\frac{1}{1-z^{25}}\,\frac{1}{1-z^{50}}\,, \tag{7.11}$$

but it's not obvious how to get from here to the coefficient of z^n. Fortunately there is a way; we'll return to this problem later in the chapter.

More elegant formulas arise if we consider the problem of giving change when we live in a land that mints coins of every positive integer denomination (①, ②, ③, ...) instead of just the five we allowed before. The corresponding generating function is an infinite product of fractions,

$$\frac{1}{(1-z)(1-z^2)(1-z^3)\ldots}\,,$$

and the coefficient of z^n when these factors are fully multiplied out is called $p(n)$, the number of *partitions* of n. A partition of n is a representation of n as a sum of positive integers, disregarding order. For example, there are seven different partitions of 5, namely

$$5 = 4+1 = 3+2 = 3+1+1 = 2+2+1 = 2+1+1+1 = 1+1+1+1+1\,;$$

hence $p(5) = 7$. (Also $p(2) = 2$, $p(3) = 3$, $p(4) = 5$, and $p(6) = 11$; it begins to look as if $p(n)$ is always a prime number. But $p(7) = 15$, spoiling the pattern.) There is no closed form for $p(n)$, but the theory of partitions is a fascinating branch of mathematics in which many remarkable discoveries have been made. For example, Ramanujan proved that $p(5n+4) \equiv 0 \pmod 5$, $p(7n+5) \equiv 0 \pmod 7$, and $p(11n+6) \equiv 0 \pmod{11}$, by making ingenious transformations of generating functions (see Andrews [11, Chapter 10]).

7.2 BASIC MANEUVERS

Now let's look more closely at some of the techniques that make power series powerful.

First a few words about terminology and notation. Our generic generating function has the form

$$G(z) = g_0 + g_1 z + g_2 z^2 + \cdots = \sum_{n \geqslant 0} g_n z^n , \tag{7.12}$$

and we say that $G(z)$, or G for short, is the generating function for the sequence $\langle g_0, g_1, g_2, \ldots \rangle$, which we also call $\langle g_n \rangle$. The coefficient g_n of z^n in $G(z)$ is often denoted $[z^n]\, G(z)$, as in Section 5.4.

The sum in (7.12) runs over all $n \geqslant 0$, but we often find it more convenient to extend the sum over all integers n. We can do this by simply regarding $g_{-1} = g_{-2} = \cdots = 0$. In such cases we might still talk about the sequence $\langle g_0, g_1, g_2, \ldots \rangle$, as if the g_n's didn't exist for negative n.

Two kinds of "closed forms" come up when we work with generating functions. We might have a closed form for $G(z)$, expressed in terms of z; or we might have a closed form for g_n, expressed in terms of n. For example, the generating function for Fibonacci numbers has the closed form $z/(1-z-z^2)$; the Fibonacci numbers themselves have the closed form $(\phi^n - \hat{\phi}^n)/\sqrt{5}$. The context will explain what kind of closed form is meant.

Now a few words about perspective. The generating function $G(z)$ appears to be two different entities, depending on how we view it. Sometimes it is a function of a complex variable z, satisfying all the standard properties proved in calculus books. And sometimes it is simply a formal power series, with z acting as a placeholder. In the previous section, for example, we used the second interpretation; we saw several examples in which z was substituted for some feature of a combinatorial object in a "sum" of such objects. The coefficient of z^n was then the number of combinatorial objects having n occurrences of that feature.

If physicists can get away with viewing light sometimes as a wave and sometimes as a particle, mathematicians should be able to view generating functions in two different ways.

When we view $G(z)$ as a function of a complex variable, its convergence becomes an issue. We said in Chapter 2 that the infinite series $\sum_{n \geqslant 0} g_n z^n$ converges (absolutely) if and only if there's a bounding constant A such that the finite sums $\sum_{0 \leqslant n \leqslant N} |g_n z^n|$ never exceed A, for any N. Therefore it's easy to see that if $\sum_{n \geqslant 0} g_n z^n$ converges for some value $z = z_0$, it also converges for all z with $|z| < |z_0|$. Furthermore, we must have $\lim_{n \to \infty} |g_n z_0^n| = 0$; hence, in the notation of Chapter 9, $g_n = O\big(|1/z_0|^n\big)$ if there is convergence at z_0. And conversely if $g_n = O(M^n)$, the series $\sum_{n \geqslant 0} g_n z^n$ converges for all $|z| < 1/M$. These are the basic facts about convergence of power series.

But for our purposes convergence is usually a red herring, unless we're trying to study the asymptotic behavior of the coefficients. Nearly every

operation we perform on generating functions can be justified rigorously as an operation on formal power series, and such operations are legal even when the series don't converge. (The relevant theory can be found, for example, in Bell [23], Niven [282], and Henrici [182, Chapter 1].)

Even if we remove the tags from our mattresses.

Furthermore, even if we throw all caution to the winds and derive formulas without any rigorous justification, we generally can take the results of our derivation and prove them by induction. For example, the generating function for the Fibonacci numbers converges only when $|z| < 1/\phi \approx 0.618$, but we didn't need to know that when we proved the formula $F_n = (\phi^n - \hat{\phi}^n)/\sqrt{5}$. The latter formula, once discovered, can be verified directly, if we don't trust the theory of formal power series. Therefore we'll ignore questions of convergence in this chapter; it's more a hindrance than a help.

So much for perspective. Next we look at our main tools for reshaping generating functions — adding, shifting, changing variables, differentiating, integrating, and multiplying. In what follows we assume that, unless stated otherwise, $F(z)$ and $G(z)$ are the generating functions for the sequences $\langle f_n \rangle$ and $\langle g_n \rangle$. We also assume that the f_n's and g_n's are zero for negative n, since this saves us some bickering with the limits of summation.

It's pretty obvious what happens when we add constant multiples of F and G together:

$$\begin{aligned} \alpha F(z) + \beta G(z) &= \alpha \sum_n f_n z^n \;+\; \beta \sum_n g_n z^n \\ &= \sum_n (\alpha f_n + \beta g_n)\, z^n . \end{aligned} \tag{7.13}$$

This gives us the generating function for the sequence $\langle \alpha f_n + \beta g_n \rangle$.

Shifting a generating function isn't much harder. To shift $G(z)$ right by m places, that is, to form the generating function for the sequence $\langle 0, \ldots, 0, g_0, g_1, \ldots \rangle = \langle g_{n-m} \rangle$ with m leading 0's, we simply multiply by z^m:

$$z^m G(z) = \sum_n g_n\, z^{n+m} = \sum_n g_{n-m}\, z^n , \qquad \text{integer } m \geqslant 0. \tag{7.14}$$

This is the operation we used (twice), along with addition, to deduce the equation $(1 - z - z^2)F(z) = z$ on our way to finding a closed form for the Fibonacci numbers in Chapter 6.

And to shift $G(z)$ left m places — that is, to form the generating function for the sequence $\langle g_m, g_{m+1}, g_{m+2}, \ldots \rangle = \langle g_{n+m} \rangle$ with the first m elements discarded — we subtract off the first m terms and then divide by z^m:

$$\frac{G(z) - g_0 - g_1 z - \cdots - g_{m-1} z^{m-1}}{z^m} = \sum_{n \geqslant m} g_n z^{n-m} = \sum_{n \geqslant 0} g_{n+m} z^n \tag{7.15}$$

(We can't extend this last sum over all n unless $g_0 = \cdots = g_{m-1} = 0$.)

Replacing the z by a constant multiple is another of our tricks:

$$G(cz) \;=\; \sum_n g_n (cz)^n \;=\; \sum_n c^n g_n z^n\,; \tag{7.16}$$

this yields the generating function for the sequence $\langle c^n g_n \rangle$. The special case $c = -1$ is particularly useful.

Often we want to bring down a factor of n into the coefficient. Differentiation is what lets us do that:

I fear d*generating-function* dz*'s.*

$$G'(z) \;=\; g_1 + 2g_2 z + 3g_3 z^2 + \cdots \;=\; \sum_n (n+1) g_{n+1}\, z^n\,. \tag{7.17}$$

Shifting this right one place gives us a form that's sometimes more useful,

$$zG'(z) \;=\; \sum_n n g_n\, z^n\,. \tag{7.18}$$

This is the generating function for the sequence $\langle n g_n \rangle$. Repeated differentiation would allow us to multiply g_n by any desired polynomial in n.

Integration, the inverse operation, lets us divide the terms by n:

$$\int_0^z G(t)\,dt \;=\; g_0 z + \frac{1}{2} g_1 z^2 + \frac{1}{3} g_2 z^3 + \cdots \;=\; \sum_{n \geqslant 1} \frac{1}{n} g_{n-1}\, z^n\,. \tag{7.19}$$

(Notice that the constant term is zero.) If we want the generating function for $\langle g_n/n \rangle$ instead of $\langle g_{n-1}/n \rangle$, we should first shift left one place, replacing $G(t)$ by $\big(G(t) - g_0\big)/t$ in the integral.

Finally, here's how we multiply generating functions together:

$$\begin{aligned} F(z)G(z) \;&=\; (f_0 + f_1 z + f_2 z^2 + \cdots)(g_0 + g_1 z + g_2 z^2 + \cdots) \\ &=\; (f_0 g_0) \;+\; (f_0 g_1 + f_1 g_0) z \;+\; (f_0 g_2 + f_1 g_1 + f_2 g_0) z^2 \;+\; \cdots \\ &=\; \sum_n \Big(\sum_k f_k g_{n-k} \Big) z^n\,. \end{aligned} \tag{7.20}$$

As we observed in Chapter 5, this gives the generating function for the sequence $\langle h_n \rangle$, the *convolution* of $\langle f_n \rangle$ and $\langle g_n \rangle$. The sum $h_n = \sum_k f_k g_{n-k}$ can also be written $h_n = \sum_{k=0}^{n} f_k g_{n-k}$, because $f_k = 0$ when $k < 0$ and $g_{n-k} = 0$ when $k > n$. Multiplication/convolution is a little more complicated than the other operations, but it's very useful — so useful that we will spend all of Section 7.5 below looking at examples of it.

Multiplication has several special cases that are worth considering as operations in themselves. We've already seen one of these: When $F(z) = z^m$ we get the shifting operation (7.14). In that case the sum h_n becomes the single term g_{n-m}, because all f_k's are 0 except for $f_m = 1$.

Table 335 Generating function manipulations.

$$\alpha F(z) + \beta G(z) = \sum_n (\alpha f_n + \beta g_n) z^n$$

$$z^m G(z) = \sum_n g_{n-m} z^n, \qquad \text{integer } m \geqslant 0$$

$$\frac{G(z) - g_0 - g_1 z - \cdots - g_{m-1} z^{m-1}}{z^m} = \sum_{n \geqslant 0} g_{n+m} z^n, \qquad \text{integer } m \geqslant 0$$

$$G(cz) = \sum_n c^n g_n z^n$$

$$G'(z) = \sum_n (n+1) g_{n+1} z^n$$

$$zG'(z) = \sum_n n g_n z^n$$

$$\int_0^z G(t)\,dt = \sum_{n \geqslant 1} \frac{1}{n} g_{n-1} z^n$$

$$F(z)G(z) = \sum_n \Bigl(\sum_k f_k g_{n-k}\Bigr) z^n$$

$$\frac{1}{1-z} G(z) = \sum_n \Bigl(\sum_{k \leqslant n} g_k\Bigr) z^n$$

Another useful special case arises when $F(z)$ is the familiar function $1/(1-z) = 1 + z + z^2 + \cdots$; then all f_k's (for $k \geqslant 0$) are 1 and we have the important formula

$$\frac{1}{1-z} G(z) = \sum_n \Bigl(\sum_{k \geqslant 0} g_{n-k}\Bigr) z^n = \sum_n \Bigl(\sum_{k \leqslant n} g_k\Bigr) z^n. \tag{7.21}$$

Multiplying a generating function by $1/(1-z)$ gives us the generating function for the cumulative sums of the original sequence.

Table 335 summarizes the operations we've discussed so far. To use all these manipulations effectively it helps to have a healthy repertoire of generating functions in stock. Table 336 lists the simplest ones; we can use those to get started and to solve quite a few problems.

Each of the generating functions in Table 336 is important enough to be memorized. Many of them are special cases of the others, and many of

Table 336 Simple sequences and their generating functions.

sequence	generating function	closed form
$\langle 1,0,0,0,0,0,\ldots\rangle$	$\sum_{n\geqslant 0}[n=0]\,z^n$	1
$\langle 0,\ldots,0,1,0,0,\ldots\rangle$	$\sum_{n\geqslant 0}[n=m]\,z^n$	z^m
$\langle 1,1,1,1,1,1,\ldots\rangle$	$\sum_{n\geqslant 0} z^n$	$\dfrac{1}{1-z}$
$\langle 1,-1,1,-1,1,-1,\ldots\rangle$	$\sum_{n\geqslant 0}(-1)^n\,z^n$	$\dfrac{1}{1+z}$
$\langle 1,0,1,0,1,0,\ldots\rangle$	$\sum_{n\geqslant 0}[2\backslash n]\,z^n$	$\dfrac{1}{1-z^2}$
$\langle 1,0,\ldots,0,1,0,\ldots,0,1,0,\ldots\rangle$	$\sum_{n\geqslant 0}[m\backslash n]\,z^n$	$\dfrac{1}{1-z^m}$
$\langle 1,2,3,4,5,6,\ldots\rangle$	$\sum_{n\geqslant 0}(n+1)\,z^n$	$\dfrac{1}{(1-z)^2}$
$\langle 1,2,4,8,16,32,\ldots\rangle$	$\sum_{n\geqslant 0}2^n\,z^n$	$\dfrac{1}{1-2z}$
$\langle 1,4,6,4,1,0,0,\ldots\rangle$	$\sum_{n\geqslant 0}\binom{4}{n}\,z^n$	$(1+z)^4$
$\langle 1,c,\binom{c}{2},\binom{c}{3},\ldots\rangle$	$\sum_{n\geqslant 0}\binom{c}{n}\,z^n$	$(1+z)^c$
$\left\langle 1,c,\binom{c+1}{2},\binom{c+2}{3},\ldots\right\rangle$	$\sum_{n\geqslant 0}\binom{c+n-1}{n}\,z^n$	$\dfrac{1}{(1-z)^c}$
$\langle 1,c,c^2,c^3,\ldots\rangle$	$\sum_{n\geqslant 0}c^n\,z^n$	$\dfrac{1}{1-cz}$
$\left\langle 1,\binom{m+1}{m},\binom{m+2}{m},\binom{m+3}{m},\ldots\right\rangle$	$\sum_{n\geqslant 0}\binom{m+n}{m}\,z^n$	$\dfrac{1}{(1-z)^{m+1}}$
$\langle 0,1,\frac{1}{2},\frac{1}{3},\frac{1}{4},\ldots\rangle$	$\sum_{n\geqslant 1}\dfrac{1}{n}\,z^n$	$\ln\dfrac{1}{1-z}$
$\langle 0,1,-\frac{1}{2},\frac{1}{3},-\frac{1}{4},\ldots\rangle$	$\sum_{n\geqslant 1}\dfrac{(-1)^{n+1}}{n}\,z^n$	$\ln(1+z)$
$\langle 1,1,\frac{1}{2},\frac{1}{6},\frac{1}{24},\frac{1}{120},\ldots\rangle$	$\sum_{n\geqslant 0}\dfrac{1}{n!}\,z^n$	e^z

them can be derived quickly from the others by using the basic operations of Table 335; therefore the memory work isn't very hard.

Hint: If the sequence consists of binomial coefficients, its generating function usually involves a binomial, $1\pm z$.

For example, let's consider the sequence $\langle 1,2,3,4,\ldots\rangle$, whose generating function $1/(1-z)^2$ is often useful. This generating function appears near the

middle of Table 336, and it's also the special case $m = 1$ of $\langle 1, \binom{m+1}{m}, \binom{m+2}{m}, \binom{m+3}{m}, \ldots \rangle$, which appears further down; it's also the special case $c = 2$ of the closely related sequence $\langle 1, c, \binom{c+1}{2}, \binom{c+2}{3}, \ldots \rangle$. We can derive it from the generating function for $\langle 1, 1, 1, 1, \ldots \rangle$ by taking cumulative sums as in (7.21); that is, by dividing $1/(1-z)$ by $(1-z)$. Or we can derive it from $\langle 1, 1, 1, 1, \ldots \rangle$ by differentiation, using (7.17).

OK, OK, I'm convinced already.

The sequence $\langle 1, 0, 1, 0, \ldots \rangle$ is another one whose generating function can be obtained in many ways. We can obviously derive the formula $\sum_n z^{2n} = 1/(1-z^2)$ by substituting z^2 for z in the identity $\sum_n z^n = 1/(1-z)$; we can also apply cumulative summation to the sequence $\langle 1, -1, 1, -1, \ldots \rangle$, whose generating function is $1/(1+z)$, getting $1/(1+z)(1-z) = 1/(1-z^2)$. And there's also a third way, which is based on a general method for extracting the even-numbered terms $\langle g_0, 0, g_2, 0, g_4, 0, \ldots \rangle$ of *any* given sequence: If we add $G(-z)$ to $G(+z)$ we get

$$G(z) + G(-z) \;=\; \sum_n g_n\big(1 + (-1)^n\big)z^n \;=\; 2\sum_n g_n[n \text{ even}]z^n\,;$$

therefore

$$\frac{G(z) + G(-z)}{2} \;=\; \sum_n g_{2n}\, z^{2n}\,. \tag{7.22}$$

The odd-numbered terms can be extracted in a similar way,

$$\frac{G(z) - G(-z)}{2} \;=\; \sum_n g_{2n+1}\, z^{2n+1}\,. \tag{7.23}$$

In the special case where $g_n = 1$ and $G(z) = 1/(1-z)$, the generating function for $\langle 1, 0, 1, 0, \ldots \rangle$ is $\frac{1}{2}\big(G(z) + G(-z)\big) = \frac{1}{2}\big(\frac{1}{1-z} + \frac{1}{1+z}\big) = \frac{1}{1-z^2}$.

Let's try this extraction trick on the generating function for Fibonacci numbers. We know that $\sum_n F_n z^n = z/(1-z-z^2)$; hence

$$\begin{aligned}\sum_n F_{2n} z^{2n} \;&=\; \frac{1}{2}\left(\frac{z}{1-z-z^2} + \frac{-z}{1+z-z^2}\right)\\ &=\; \frac{1}{2}\left(\frac{z + z^2 - z^3 - z + z^2 + z^3}{(1-z^2)^2 - z^2}\right) \;=\; \frac{z^2}{1 - 3z^2 + z^4}\,.\end{aligned}$$

This generates the sequence $\langle F_0, 0, F_2, 0, F_4, \ldots \rangle$; hence the sequence of alternate F's, $\langle F_0, F_2, F_4, F_6, \ldots \rangle = \langle 0, 1, 3, 8, \ldots \rangle$, has a simple generating function:

$$\sum_n F_{2n} z^n \;=\; \frac{z}{1 - 3z + z^2}\,. \tag{7.24}$$

7.3 SOLVING RECURRENCES

Now let's focus our attention on one of the most important uses of generating functions: the solution of recurrence relations.

Given a sequence $\langle g_n \rangle$ that satisfies a given recurrence, we seek a closed form for g_n in terms of n. A solution to this problem via generating functions proceeds in four steps that are almost mechanical enough to be programmed on a computer:

1 Write down a single equation that expresses g_n in terms of other elements of the sequence. This equation should be valid for all integers n, assuming that $g_{-1} = g_{-2} = \cdots = 0$.

2 Multiply both sides of the equation by z^n and sum over all n. This gives, on the left, the sum $\sum_n g_n z^n$, which is the generating function $G(z)$. The right-hand side should be manipulated so that it becomes some other expression involving $G(z)$.

3 Solve the resulting equation, getting a closed form for $G(z)$.

4 Expand $G(z)$ into a power series and read off the coefficient of z^n; this is a closed form for g_n.

This method works because the single function $G(z)$ represents the entire sequence $\langle g_n \rangle$ in such a way that many manipulations are possible.

Example 1: Fibonacci numbers revisited.

For example, let's rerun the derivation of Fibonacci numbers from Chapter 6. In that chapter we were feeling our way, learning a new method; now we can be more systematic. The given recurrence is

$$\begin{aligned} g_0 &= 0; \qquad g_1 = 1; \\ g_n &= g_{n-1} + g_{n-2}, \qquad \text{for } n \geqslant 2. \end{aligned}$$

We will find a closed form for g_n by using the four steps above.

Step 1 tells us to write the recurrence as a "single equation" for g_n. We could say

$$g_n = \begin{cases} 0, & \text{if } n \leqslant 0; \\ 1, & \text{if } n = 1; \\ g_{n-1} + g_{n-2}, & \text{if } n > 1; \end{cases}$$

but this is cheating. Step 1 really asks for a formula that doesn't involve a case-by-case construction. The single equation

$$g_n = g_{n-1} + g_{n-2}$$

works for $n \geqslant 2$, and it also holds when $n \leqslant 0$ (because we have $g_0 = 0$ and $g_{\text{negative}} = 0$). But when $n = 1$ we get 1 on the left and 0 on the right.

Fortunately the problem is easy to fix, since we can add $[n=1]$ to the right; this adds 1 when $n=1$, and it makes no change when $n \neq 1$. So, we have

$$g_n = g_{n-1} + g_{n-2} + [n=1];$$

this is the equation called for in Step 1.

Step 2 now asks us to transform the equation for $\langle g_n \rangle$ into an equation for $G(z) = \sum_n g_n z^n$. The task is not difficult:

$$\begin{aligned} G(z) &= \sum_n g_n z^n = \sum_n g_{n-1} z^n + \sum_n g_{n-2} z^n + \sum_n [n=1] z^n \\ &= \sum_n g_n z^{n+1} + \sum_n g_n z^{n+2} + z \\ &= zG(z) + z^2 G(z) + z. \end{aligned}$$

Step 3 is also simple in this case; we have

$$G(z) = \frac{z}{1-z-z^2},$$

which of course comes as no surprise.

Step 4 is the clincher. We carried it out in Chapter 6 by having a sudden flash of inspiration; let's go more slowly now, so that we can get through Step 4 safely later, when we meet problems that are more difficult. What is

$$[z^n] \frac{z}{1-z-z^2},$$

the coefficient of z^n when $z/(1-z-z^2)$ is expanded in a power series? More generally, if we are given any rational function

$$R(z) = \frac{P(z)}{Q(z)},$$

where P and Q are polynomials, what is the coefficient $[z^n]\, R(z)$?

There's one kind of rational function whose coefficients are particularly nice, namely

$$\frac{a}{(1-\rho z)^{m+1}} = \sum_{n \geqslant 0} \binom{m+n}{m} a \rho^n z^n. \tag{7.25}$$

(The case $\rho = 1$ appears in Table 336, and we can get the general formula shown here by substituting ρz for z.) A finite sum of functions like (7.25),

$$S(z) = \frac{a_1}{(1-\rho_1 z)^{m_1+1}} + \frac{a_2}{(1-\rho_2 z)^{m_2+1}} + \cdots + \frac{a_l}{(1-\rho_l z)^{m_l+1}}, \tag{7.26}$$

also has nice coefficients,

$$[z^n]\,S(z) \;=\; a_1\binom{m_1+n}{m_1}\rho_1^n + a_2\binom{m_2+n}{m_2}\rho_2^n + \cdots + a_l\binom{m_l+n}{m_l}\rho_l^n\,. \tag{7.27}$$

We will show that every rational function $R(z)$ such that $R(0) \neq \infty$ can be expressed in the form

$$R(z) \;=\; S(z) + T(z)\,, \tag{7.28}$$

where $S(z)$ has the form (7.26) and $T(z)$ is a polynomial. Therefore there is a closed form for the coefficients $[z^n]\,R(z)$. Finding $S(z)$ and $T(z)$ is equivalent to finding the "partial fraction expansion" of $R(z)$.

Notice that $S(z) = \infty$ when z has the values $1/\rho_1, \ldots, 1/\rho_l$. Therefore the numbers ρ_k that we need to find, if we're going to succeed in expressing $R(z)$ in the desired form $S(z) + T(z)$, must be the reciprocals of the numbers α_k where $Q(\alpha_k) = 0$. (Recall that $R(z) = P(z)/Q(z)$, where P and Q are polynomials; we have $R(z) = \infty$ only if $Q(z) = 0$.)

Suppose $Q(z)$ has the form

$$Q(z) \;=\; q_0 + q_1 z + \cdots + q_m z^m\,, \qquad \text{where } q_0 \neq 0 \text{ and } q_m \neq 0.$$

The "reflected" polynomial

$$Q^R(z) \;=\; q_0 z^m + q_1 z^{m-1} + \cdots + q_m$$

has an important relation to $Q(z)$:

$$\begin{aligned} Q^R(z) \;&=\; q_0(z-\rho_1)\ldots(z-\rho_m) \\ \iff\quad Q(z) \;&=\; q_0(1-\rho_1 z)\ldots(1-\rho_m z)\,. \end{aligned}$$

Thus, the roots of Q^R are the reciprocals of the roots of Q, and vice versa. We can therefore find the numbers ρ_k we seek by factoring the reflected polynomial $Q^R(z)$.

For example, in the Fibonacci case we have

$$Q(z) \;=\; 1 - z - z^2\,; \qquad Q^R(z) \;=\; z^2 - z - 1\,.$$

The roots of Q^R can be found by setting $(a, b, c) = (1, -1, -1)$ in the quadratic formula $\bigl(-b \pm \sqrt{b^2 - 4ac}\bigr)/2a$; we find that they are

$$\phi \;=\; \frac{1+\sqrt{5}}{2} \quad\text{and}\quad \hat{\phi} \;=\; \frac{1-\sqrt{5}}{2}\,.$$

Therefore $Q^R(z) = (z-\phi)(z-\hat{\phi})$ and $Q(z) = (1-\phi z)(1-\hat{\phi} z)$.

Once we've found the ρ's, we can proceed to find the partial fraction expansion. It's simplest if all the roots are distinct, so let's consider that special case first. We might as well state and prove the general result formally:

Rational Expansion Theorem for Distinct Roots.

If $R(z) = P(z)/Q(z)$, *where* $Q(z) = q_0(1-\rho_1 z)\ldots(1-\rho_l z)$ *and the numbers* $(\rho_1, \ldots, \rho_l)$ *are distinct, and if* $P(z)$ *is a polynomial of degree less than* l, *then*

$$[z^n]\,R(z) \;=\; a_1\rho_1^n + \cdots + a_l\rho_l^n\,, \qquad \text{where} \quad a_k = \frac{-\rho_k P(1/\rho_k)}{Q'(1/\rho_k)}. \tag{7.29}$$

Proof: Let $a_1, \ldots, a_l$ be the stated constants. Formula (7.29) holds if $R(z) = P(z)/Q(z)$ is equal to

$$S(z) \;=\; \frac{a_1}{1-\rho_1 z} + \cdots + \frac{a_l}{1-\rho_l z}\,.$$

Impress your parents by leaving the book open at this page.

And we can prove that $R(z) = S(z)$ by showing that the function $T(z) = R(z) - S(z)$ is not infinite as $z \to 1/\rho_k$. For this will show that the rational function $T(z)$ is never infinite; hence $T(z)$ must be a polynomial. We also can show that $T(z) \to 0$ as $z \to \infty$; hence $T(z)$ must be zero.

Let $\alpha_k = 1/\rho_k$. To prove that $\lim_{z\to\alpha_k} T(z) \neq \infty$, it suffices to show that $\lim_{z\to\alpha_k}(z-\alpha_k)T(z) = 0$, because $T(z)$ is a rational function of z. Thus we want to show that

$$\lim_{z\to\alpha_k}(z-\alpha_k)R(z) \;=\; \lim_{z\to\alpha_k}(z-\alpha_k)S(z)\,.$$

The right-hand limit equals $\lim_{z\to\alpha_k} a_k(z-\alpha_k)/(1-\rho_k z) = -a_k/\rho_k$, because $(1-\rho_k z) = -\rho_k(z-\alpha_k)$ and $(z-\alpha_k)/(1-\rho_j z) \to 0$ for $j \neq k$. The left-hand limit is

$$\lim_{z\to\alpha_k}(z-\alpha_k)\frac{P(z)}{Q(z)} \;=\; P(\alpha_k)\lim_{z\to\alpha_k}\frac{z-\alpha_k}{Q(z)} \;=\; \frac{P(\alpha_k)}{Q'(\alpha_k)}\,,$$

by L'Hospital's rule. Thus the theorem is proved.

Returning to the Fibonacci example, we have $P(z) = z$ and $Q(z) = 1 - z - z^2 = (1-\phi z)(1-\hat{\phi}z)$; hence $Q'(z) = -1-2z$, and

$$\frac{-\rho P(1/\rho)}{Q'(1/\rho)} \;=\; \frac{-1}{-1-2/\rho} \;=\; \frac{\rho}{\rho+2}\,.$$

According to (7.29), the coefficient of ϕ^n in $[z^n]\,R(z)$ is therefore $\phi/(\phi+2) = 1/\sqrt{5}$; the coefficient of $\hat{\phi}^n$ is $\hat{\phi}/(\hat{\phi}+2) = -1/\sqrt{5}$. So the theorem tells us that $F_n = (\phi^n - \hat{\phi}^n)/\sqrt{5}$, as in (6.123).

When $Q(z)$ has repeated roots, the calculations become more difficult, but we can beef up the proof of the theorem and prove the following more general result:

General Expansion Theorem for Rational Generating Functions.

If $R(z) = P(z)/Q(z)$, *where* $Q(z) = q_0(1-\rho_1 z)^{d_1} \ldots (1-\rho_l z)^{d_l}$ *and the numbers* $(\rho_1, \ldots, \rho_l)$ *are distinct, and if* $P(z)$ *is a polynomial of degree less than* $d_1 + \cdots + d_l$, *then*

$$[z^n]\,R(z) \;=\; f_1(n)\rho_1^n + \cdots + f_l(n)\rho_l^n \qquad \text{for all } n \geqslant 0, \tag{7.30}$$

where each $f_k(n)$ *is a polynomial of degree* $d_k - 1$ *with leading coefficient*

$$\begin{aligned} a_k &= \frac{(-\rho_k)^{d_k} P(1/\rho_k) d_k}{Q^{(d_k)}(1/\rho_k)} \\ &= \frac{P(1/\rho_k)}{(d_k-1)!\, q_0 \prod_{j\neq k}(1-\rho_j/\rho_k)^{d_j}}\,. \end{aligned} \tag{7.31}$$

This can be proved by induction on $\max(d_1, \ldots, d_l)$, using the fact that

$$R(z) - \frac{a_1(d_1-1)!}{(1-\rho_1 z)^{d_1}} - \cdots - \frac{a_l(d_l-1)!}{(1-\rho_l z)^{d_l}}$$

is a rational function whose denominator polynomial is not divisible by $(1-\rho_k z)^{d_k}$ for any k.

Example 2: A more-or-less random recurrence.

Now that we've seen some general methods, we're ready to tackle new problems. Let's try to find a closed form for the recurrence

$$\begin{aligned} g_0 &= g_1 = 1\,; \\ g_n &= g_{n-1} + 2g_{n-2} + (-1)^n\,, \qquad \text{for } n \geqslant 2. \end{aligned} \tag{7.32}$$

It's always a good idea to make a table of small cases first, and the recurrence lets us do that easily:

n	0	1	2	3	4	5	6	7
$(-1)^n$	1	−1	1	−1	1	−1	1	−1
g_n	1	1	4	5	14	23	52	97

No closed form is evident, and this sequence isn't even listed in Sloane's *Handbook* [330]; so we need to go through the four-step process if we want to discover the solution.

Step 1 is easy, since we merely need to insert fudge factors to fix things when $n < 2$: The equation

$$g_n = g_{n-1} + 2g_{n-2} + (-1)^n[n \geqslant 0] + [n = 1]$$

holds for all integers n. Now we can carry out Step 2:

N.B.: The upper index on $\sum_{n=1} z^n$ is not missing!

$$\begin{aligned} G(z) &= \sum_n g_n z^n = \sum_n g_{n-1} z^n + 2\sum_n g_{n-2} z^n + \sum_{n\geqslant 0} (-1)^n z^n + \sum_{n=1} z^n \\ &= zG(z) + 2z^2 G(z) + \frac{1}{1+z} + z\,. \end{aligned}$$

(Incidentally, we could also have used $\binom{-1}{n}$ instead of $(-1)^n[n \geqslant 0]$, thereby getting $\sum_n \binom{-1}{n} z^n = (1+z)^{-1}$ by the binomial theorem.) Step 3 is elementary algebra, which yields

$$G(z) = \frac{1 + z(1+z)}{(1+z)(1-z-2z^2)} = \frac{1+z+z^2}{(1-2z)(1+z)^2}\,.$$

And that leaves us with Step 4.

The squared factor in the denominator is a bit troublesome, since we know that repeated roots are more complicated than distinct roots; but there it is. We have two roots, $\rho_1 = 2$ and $\rho_2 = -1$; the general expansion theorem (7.30) tells us that

$$g_n = a_1 2^n + (a_2 n + c)(-1)^n$$

for some constant c, where

$$a_1 = \frac{1 + 1/2 + 1/4}{(1+1/2)^2} = \frac{7}{9}; \qquad a_2 = \frac{1-1+1}{1-2/(-1)} = \frac{1}{3}\,.$$

(The second formula for a_k in (7.31) is easier to use than the first one when the denominator has nice factors. We simply substitute $z = 1/\rho_k$ everywhere in $R(z)$, except in the factor where this gives zero, and divide by $(d_k - 1)!$; this gives the coefficient of $n^{d_k-1}\rho_k^n$.) Plugging in $n = 0$ tells us that the value of the remaining constant c had better be $\frac{2}{9}$; hence our answer is

$$g_n = \tfrac{7}{9}2^n + \left(\tfrac{1}{3}n + \tfrac{2}{9}\right)(-1)^n\,. \tag{7.33}$$

It doesn't hurt to check the cases $n = 1$ and 2, just to be sure that we didn't foul up. Maybe we should even try $n = 3$, since this formula looks weird. But it's correct, all right.

Could we have discovered (7.33) by guesswork? Perhaps after tabulating a few more values we may have observed that $g_{n+1} \approx 2g_n$ when n is large.

And with chutzpah and luck we might even have been able to smoke out the constant $\frac{7}{9}$. But it sure is simpler and more reliable to have generating functions as a tool.

Example 3: Mutually recursive sequences.

Sometimes we have two or more recurrences that depend on each other. Then we can form generating functions for both of them, and solve both by a simple extension of our four-step method.

For example, let's return to the problem of $3 \times n$ domino tilings that we explored earlier this chapter. If we want to know only the total number of ways, U_n, to cover a $3 \times n$ rectangle with dominoes, without breaking this number down into vertical dominoes versus horizontal dominoes, we needn't go into as much detail as we did before. We can merely set up the recurrences

$$\begin{aligned} &U_0 = 1\,, \qquad U_1 = 0\,; \qquad V_0 = 0\,, \qquad V_1 = 1\,; \\ &U_n = 2V_{n-1} + U_{n-2}\,, \qquad V_n = U_{n-1} + V_{n-2}\,, \qquad \text{for } n \geqslant 2. \end{aligned}$$

Here V_n is the number of ways to cover a $3 \times n$ rectangle-minus-corner, using $(3n-1)/2$ dominoes. These recurrences are easy to discover, if we consider the possible domino configurations at the rectangle's left edge, as before. Here are the values of U_n and V_n for small n:

n	0	1	2	3	4	5	6	7
U_n	1	0	3	0	11	0	41	0
V_n	0	1	0	4	0	15	0	56

(7.34)

Let's find closed forms, in four steps. First (Step 1), we have

$$U_n \;=\; 2V_{n-1} + U_{n-2} + [n{=}0]\,, \qquad V_n \;=\; U_{n-1} + V_{n-2}\,,$$

for all n. Hence (Step 2),

$$U(z) \;=\; 2zV(z) + z^2U(z) + 1\,, \qquad V(z) \;=\; zU(z) + z^2V(z)\,.$$

Now (Step 3) we must solve two equations in two unknowns; but these are easy, since the second equation yields $V(z) = zU(z)/(1-z^2)$; we find

$$U(z) \;=\; \frac{1-z^2}{1-4z^2+z^4}\,; \qquad V(z) \;=\; \frac{z}{1-4z^2+z^4}\,. \tag{7.35}$$

(We had this formula for $U(z)$ in (7.10), but with z^3 instead of z^2. In that derivation, n was the number of dominoes; now it's the width of the rectangle.)

The denominator $1-4z^2+z^4$ is a function of z^2; this is what makes $U_{2n+1} = 0$ and $V_{2n} = 0$, as they should be. We can take advantage of this

nice property of z^2 by retaining z^2 when we factor the denominator: We need not take $1-4z^2+z^4$ all the way to a product of four factors $(1-\rho_k z)$, since two factors of the form $(1-\rho_k z^2)$ will be enough to tell us the coefficients. In other words if we consider the generating function

$$W(z) \;=\; \frac{1}{1-4z+z^2} \;=\; W_0 + W_1\,z + W_2\,z^2 + \cdots\,, \tag{7.36}$$

we will have $V(z) = zW(z^2)$ and $U(z) = (1-z^2)W(z^2)$; hence $V_{2n+1} = W_n$ and $U_{2n} = W_n - W_{n-1}$. We save time and energy by working with the simpler function $W(z)$.

The factors of $1-4z+z^2$ are $(z-2-\sqrt{3}\,)$ and $(z-2+\sqrt{3}\,)$, and they can also be written $\bigl(1-(2+\sqrt{3}\,)z\bigr)$ and $\bigl(1-(2-\sqrt{3}\,)z\bigr)$ because this polynomial is its own reflection. Thus it turns out that we have

$$\begin{aligned} V_{2n+1} \;&=\; W_n \;=\; \frac{3+2\sqrt{3}}{6}(2+\sqrt{3}\,)^n + \frac{3-2\sqrt{3}}{6}(2-\sqrt{3}\,)^n\,; \\ U_{2n} \;&=\; W_n - W_{n-1} \;=\; \frac{3+\sqrt{3}}{6}(2+\sqrt{3}\,)^n + \frac{3-\sqrt{3}}{6}(2-\sqrt{3}\,)^n \\ &=\; \frac{(2+\sqrt{3}\,)^n}{3-\sqrt{3}} + \frac{(2-\sqrt{3}\,)^n}{3+\sqrt{3}}\,. \end{aligned} \tag{7.37}$$

This is the desired closed form for the number of $3 \times n$ domino tilings.

Incidentally, we can simplify the formula for U_{2n} by realizing that the second term always lies between 0 and 1. The number U_{2n} is an integer, so we have

$$U_{2n} \;=\; \left\lceil \frac{(2+\sqrt{3}\,)^n}{3-\sqrt{3}} \right\rceil, \qquad \text{for } n \geqslant 0. \tag{7.38}$$

In fact, the other term $(2-\sqrt{3}\,)^n/(3+\sqrt{3}\,)$ is extremely small when n is large, because $2-\sqrt{3} \approx 0.268$. This needs to be taken into account if we try to use formula (7.38) in numerical calculations. For example, a fairly expensive name-brand hand calculator comes up with 413403.0005 when asked to compute $(2+\sqrt{3})^{10}/(3-\sqrt{3})$. This is correct to nine significant figures; but the true value is slightly *less* than 413403, not slightly greater. Therefore it would be a mistake to take the ceiling of 413403.0005; the correct answer, $U_{20} = 413403$, is obtained by *rounding* to the nearest integer. Ceilings can be hazardous.

I've known slippery floors too.

Example 4: A closed form for change.

When we left the problem of making change, we had just calculated the number of ways to pay 50¢. Let's try now to count the number of ways there are to change a dollar, or a million dollars — still using only pennies, nickels, dimes, quarters, and halves.

The generating function derived earlier is

$$C(z) = \frac{1}{1-z}\,\frac{1}{1-z^5}\,\frac{1}{1-z^{10}}\,\frac{1}{1-z^{25}}\,\frac{1}{1-z^{50}}\,;$$

this is a rational function of z with a denominator of degree 91. Therefore we can decompose the denominator into 91 factors and come up with a 91-term "closed form" for C_n, the number of ways to give n cents in change. But that's too horrible to contemplate. Can't we do better than the general method suggests, in this particular case?

One ray of hope suggests itself immediately, when we notice that the denominator is almost a function of z^5. The trick we just used to simplify the calculations by noting that $1-4z^2+z^4$ is a function of z^2 can be applied to $C(z)$, if we replace $1/(1-z)$ by $(1+z+z^2+z^3+z^4)/(1-z^5)$:

$$\begin{aligned} C(z) &= \frac{1+z+z^2+z^3+z^4}{1-z^5}\,\frac{1}{1-z^5}\,\frac{1}{1-z^{10}}\,\frac{1}{1-z^{25}}\,\frac{1}{1-z^{50}} \\ &= (1+z+z^2+z^3+z^4)\check{C}(z^5)\,, \\ \check{C}(z) &= \frac{1}{1-z}\,\frac{1}{1-z}\,\frac{1}{1-z^2}\,\frac{1}{1-z^5}\,\frac{1}{1-z^{10}}\,. \end{aligned}$$

The compressed function $\check{C}(z)$ has a denominator whose degree is only 19, so it's much more tractable than the original. This new expression for $C(z)$ shows us, incidentally, that $C_{5n} = C_{5n+1} = C_{5n+2} = C_{5n+3} = C_{5n+4}$; and indeed, this set of equations is obvious in retrospect: The number of ways to leave a 53¢ tip is the same as the number of ways to leave a 50¢ tip, because the number of pennies is predetermined modulo 5.

But $\check{C}(z)$ still doesn't have a really simple closed form based on the roots of the denominator. The easiest way to compute the coefficients of $\check{C}(z)$ is probably to recognize that each of the denominator factors is a divisor of $1-z^{10}$. Hence we can write

Now we're also getting compressed reasoning.

$$\check{C}(z) = \frac{A(z)}{(1-z^{10})^5}\,, \quad \text{where } A(z) = A_0 + A_1 z + \cdots + A_{31} z^{31}. \tag{7·39}$$

The actual value of $A(z)$, for the curious, is

$$\begin{aligned} &(1+z+\cdots+z^9)^2(1+z^2+\cdots+z^8)(1+z^5) \\ &\quad = 1+2z+4z^2+6z^3+9z^4+13z^5+18z^6+24z^7 \\ &\qquad +31z^8+39z^9+45z^{10}+52z^{11}+57z^{12}+63z^{13}+67z^{14}+69z^{15} \\ &\qquad +69z^{16}+67z^{17}+63z^{18}+57z^{19}+52z^{20}+45z^{21}+39z^{22}+31z^{23} \\ &\qquad +24z^{24}+18z^{25}+13z^{26}+9z^{27}+6z^{28}+4z^{29}+2z^{30}+z^{31}\,. \end{aligned}$$

Finally, since $1/(1-z^{10})^5 = \sum_{k\geqslant 0}\binom{k+4}{4}z^{10k}$, we can determine the coefficient $\check{C}_n = [z^n]\,\check{C}(z)$ as follows, when $n = 10q + r$ and $0 \leqslant r < 10$:

$$\begin{aligned}\check{C}_{10q+r} &= \sum_{j,k} A_j\binom{k+4}{4}[10q+r=10k+j]\\ &= A_r\binom{q+4}{4} + A_{r+10}\binom{q+3}{4} + A_{r+20}\binom{q+2}{4} + A_{r+30}\binom{q+1}{4}\,. \qquad (7.40)\end{aligned}$$

This gives ten cases, one for each value of r; but it's a pretty good closed form, compared with alternatives that involve powers of complex numbers.

For example, we can use this expression to deduce the value of $C_{50q} = \check{C}_{10q}$. Then $r = 0$ and we have

$$C_{50q} = \binom{q+4}{4} + 45\binom{q+3}{4} + 52\binom{q+2}{4} + 2\binom{q+1}{4}\,.$$

The number of ways to change 50¢ is $\binom{5}{4} + 45\binom{4}{4} = 50$; the number of ways to change \$1 is $\binom{6}{4} + 45\binom{5}{4} + 52\binom{4}{4} = 292$; and the number of ways to change \$1,000,000 is

$$\begin{aligned}&\binom{2000004}{4} + 45\binom{2000003}{4} + 52\binom{2000002}{4} + 2\binom{2000001}{4}\\ &\qquad = 66666793333412666685000001\,.\end{aligned}$$

Example 5: A divergent series.

Now let's try to get a closed form for the numbers g_n defined by

$$\begin{aligned}g_0 &= 1\,;\\ g_n &= n g_{n-1}\,, \qquad \text{for } n > 0.\end{aligned}$$

Nowadays people are talking femto seconds.

After staring at this for a few nanoseconds we realize that g_n is just $n!$; in fact, the method of summation factors described in Chapter 2 suggests this answer immediately. But let's try to solve the recurrence with generating functions, just to see what happens. (A powerful technique should be able to handle easy recurrences like this, as well as others that have answers we can't guess so easily.)

The equation

$$g_n = n g_{n-1} + [n=0]$$

holds for all n, and it leads to

$$G(z) = \sum_n g_n z^n = \sum_n n g_{n-1}\, z^n + \sum_{n=0} z^n\,.$$

To complete Step 2, we want to express $\sum_n n g_{n-1}\, z^n$ in terms of $G(z)$, and the basic maneuvers in Table 335 suggest that the derivative $G'(z) =$

$\sum_n n g_n z^{n-1}$ is somehow involved. So we steer toward that kind of sum:

$$\begin{aligned} G(z) &= 1 + \sum_n (n+1) g_n z^{n+1} \\ &= 1 + \sum_n n g_n z^{n+1} + \sum_n g_n z^{n+1} \\ &= 1 + z^2 G'(z) + zG(z) . \end{aligned}$$

Let's check this equation, using the values of g_n for small n. Since

$$\begin{aligned} G &= 1 + z + 2z^2 + 6z^3 + 24z^4 + \cdots , \\ G' &= \quad 1 + 4z + 18z^2 + 96z^3 + \cdots , \end{aligned}$$

we have

$$\begin{aligned} z^2 G' &= \quad z^2 + 4z^3 + 18z^4 + 96z^5 + \cdots , \\ zG &= \quad z + z^2 + 2z^3 + 6z^4 + 24z^5 + \cdots , \\ 1 &= 1 . \end{aligned}$$

These three lines add up to G, so we're fine so far. Incidentally, we often find it convenient to write 'G' instead of '$G(z)$'; the extra '(z)' just clutters up the formula when we aren't changing z.

Step 3 is next, and it's different from what we've done before because we have a differential equation to solve. But this is a differential equation that we can handle with the hypergeometric series techniques of Section 5.6; those techniques aren't too bad. (Readers who are unfamiliar with hypergeometrics needn't worry—this will be quick.)

"This will be quick." That's what the doctor said just before he stuck me with that needle. Come to think of it, "hypergeometric" sounds a lot like "hypodermic."

First we must get rid of the constant '1', so we take the derivative of both sides:

$$\begin{aligned} G' = (z^2 G' + zG + 1)' &= (2zG' + z^2 G'') + (G + zG') \\ &= z^2 G'' + 3zG' + G . \end{aligned}$$

The theory in Chapter 5 tells us to rewrite this using the ϑ operator, and we know from exercise 6.13 that

$$\vartheta G = zG' , \qquad \vartheta^2 G = z^2 G'' + zG' .$$

Therefore the desired form of the differential equation is

$$\vartheta G = z\vartheta^2 G + 2z\vartheta G + zG = z(\vartheta + 1)^2 G .$$

According to (5.109), the solution with $g_0 = 1$ is the hypergeometric series $F(1, 1; ; z)$.

Step 3 was more than we bargained for; but now that we know what the function G is, Step 4 is easy—the hypergeometric definition (5.76) gives us the power series expansion:

$$G(z) = F\left(\begin{matrix}1,1\\ \\ \end{matrix}\middle| z\right) = \sum_{n\geqslant 0} \frac{1^{\overline{n}}\,1^{\overline{n}}\,z^n}{n!} = \sum_{n\geqslant 0} n!\,z^n\,.$$

We've confirmed the closed form we knew all along, $g_n = n!$.

Notice that the technique gave the right answer even though $G(z)$ diverges for all nonzero z. The sequence $n!$ grows so fast, the terms $|n!\,z^n|$ approach ∞ as $n \to \infty$, unless $z = 0$. This shows that formal power series can be manipulated algebraically without worrying about convergence.

Example 6: A recurrence that goes all the way back.

Let's close this section by applying generating functions to a problem in graph theory. A *fan* of order n is a graph on the vertices $\{0, 1, \ldots, n\}$ with $2n-1$ edges defined as follows: Vertex 0 is connected by an edge to each of the other n vertices, and vertex k is connected by an edge to vertex $k+1$, for $1 \leqslant k < n$. Here, for example, is the fan of order 4, which has five vertices and seven edges.

The problem of interest: How many spanning trees f_n are in such a graph? A *spanning tree* is a subgraph containing all the vertices, and containing enough edges to make the subgraph connected yet not so many that it has a cycle. It turns out that every spanning tree of a graph on $n+1$ vertices has exactly n edges. With fewer than n edges the subgraph wouldn't be connected, and with more than n it would have a cycle; graph theory books prove this.

There are $\binom{2n-1}{n}$ ways to choose n edges from among the $2n-1$ present in a fan of order n, but these choices don't always yield a spanning tree. For instance the subgraph

has four edges but is not a spanning tree; it has a cycle from 0 to 4 to 3 to 0, and it has no connection between $\{1, 2\}$ and the other vertices. We want to count how many of the $\binom{2n-1}{n}$ choices actually do yield spanning trees.

Let's look at some small cases. It's pretty easy to enumerate the spanning trees for $n = 1$, 2, and 3:

$f_1 = 1$ $\qquad$ $f_2 = 3$ $\qquad$ $f_3 = 8$

(We need not show the labels on the vertices, if we always draw vertex 0 at the left.) What about the case $n = 0$? At first it seems reasonable to set $f_0 = 1$; but we'll take $f_0 = 0$, because the existence of a fan of order 0 (which should have $2n - 1 = -1$ edges) is dubious.

Our four-step procedure tells us to find a recurrence for f_n that holds for all n. We can get a recurrence by observing how the topmost vertex (vertex n) is connected to the rest of the spanning tree. If it's not connected to vertex 0, it must be connected to vertex $n-1$, since it must be connected to the rest of the graph. In this case, any of the f_{n-1} spanning trees for the remaining fan (on the vertices 0 through $n-1$) will complete a spanning tree for the whole graph. Otherwise vertex n *is* connected to 0, and there's some number $k \leqslant n$ such that vertices n, $n-1$, ..., k are connected directly but the edge between k and $k-1$ is not present. Then there can't be any edges between 0 and $\{n-1, \ldots, k\}$, or there would be a cycle. If $k = 1$, the spanning tree is therefore determined completely. And if $k > 1$, any of the f_{k-1} ways to produce a spanning tree on $\{0, 1, \ldots, k-1\}$ will yield a spanning tree on the whole graph. For example, here's what this analysis produces when $n = 4$:

The general equation, valid for $n \geqslant 1$, is

$$f_n \;=\; f_{n-1} + f_{n-1} + f_{n-2} + f_{n-3} + \cdots + f_1 + 1\,.$$

(It almost seems as though the '1' on the end is f_0 and we should have chosen $f_0 = 1$; but we will doggedly stick with our choice.) A few changes suffice to make the equation valid for all integers n:

$$f_n \;=\; f_{n-1} \;+\; \sum_{k<n} f_k \;+\; [n>0]\,. \tag{7.41}$$

This is a recurrence that "goes all the way back" from f_{n-1} through all previous values, so it's different from the other recurrences we've seen so far in this chapter. We used a special method to get rid of a similar right-side sum in Chapter 2, when we solved the quicksort recurrence (2.12); namely, we subtracted one instance of the recurrence from another ($f_{n+1} - f_n$). This trick would get rid of the $\sum$ now, as it did then; but we'll see that generating functions allow us to work directly with such sums. (And it's a good thing that they do, because we will be seeing much more complicated recurrences before long.)

Step 1 is finished; Step 2 is where we need to do a new thing:

$$\begin{aligned}
F(z) \;=\; \sum_n f_n z^n \;&=\; \sum_n f_{n-1} z^n + \sum_{k,n} f_k z^n [k<n] + \sum_n [n>0] z^n \\
&=\; zF(z) \;+\; \sum_k f_k z^k \sum_n [n>k] z^{n-k} \;+\; \frac{z}{1-z} \\
&=\; zF(z) \;+\; F(z) \sum_{m>0} z^m \;+\; \frac{z}{1-z} \\
&=\; zF(z) \;+\; F(z)\,\frac{z}{1-z} \;+\; \frac{z}{1-z}\,.
\end{aligned}$$

The key trick here was to change z^n to $z^k z^{n-k}$; this made it possible to express the value of the double sum in terms of $F(z)$, as required in Step 2.

Now Step 3 is simple algebra, and we find

$$F(z) \;=\; \frac{z}{1-3z+z^2}\,.$$

Those of us with a zest for memorization will recognize this as the generating function (7.24) for the even-numbered Fibonacci numbers. So, we needn't go through Step 4; we have found a somewhat surprising answer to the spans-of-fans problem:

$$f_n \;=\; F_{2n}\,, \qquad \text{for } n \geqslant 0. \tag{7.42}$$

7.4 SPECIAL GENERATING FUNCTIONS

Step 4 of the four-step procedure becomes much easier if we know the coefficients of lots of different power series. The expansions in Table 336 are quite useful, as far as they go, but many other types of closed forms are possible. Therefore we ought to supplement that table with another one, which lists power series that correspond to the "special numbers" considered in Chapter 6.

Table 352 Generating functions for special numbers.

$$\frac{1}{(1-z)^{m+1}}\ln\frac{1}{1-z} = \sum_{n\geqslant 0}(H_{m+n}-H_m)\binom{m+n}{n}z^n \tag{7.43}$$

$$\frac{z}{e^z-1} = \sum_{n\geqslant 0}B_n\frac{z^n}{n!} \tag{7.44}$$

$$\frac{F_m z}{1-(F_{m-1}+F_{m+1})z+(-1)^m z^2} = \sum_{n\geqslant 0}F_{mn}\,z^n \tag{7.45}$$

$$\sum_k\left\{{m \atop k}\right\}\frac{k!\,z^k}{(1-z)^{k+1}} = \sum_{n\geqslant 0}n^m z^n \tag{7.46}$$

$$\left(z^{-1}\right)^{\overline{-m}} = \frac{z^m}{(1-z)(1-2z)\ldots(1-mz)} = \sum_{n\geqslant 0}\left\{{n \atop m}\right\}z^n \tag{7.47}$$

$$z^{\overline{m}} = z(z+1)\ldots(z+m-1) = \sum_{n\geqslant 0}\left[{m \atop n}\right]z^n \tag{7.48}$$

$$\left(e^z-1\right)^m = m!\sum_{n\geqslant 0}\left\{{n \atop m}\right\}\frac{z^n}{n!} \tag{7.49}$$

$$\left(\ln\frac{1}{1-z}\right)^m = m!\sum_{n\geqslant 0}\left[{n \atop m}\right]\frac{z^n}{n!} \tag{7.50}$$

$$\left(\frac{z}{\ln(1+z)}\right)^m = \sum_{n\geqslant 0}\frac{z^n}{n!}\left\{{m \atop m-n}\right\}\Big/\binom{m-1}{n} \tag{7.51}$$

$$\left(\frac{z}{1-e^{-z}}\right)^m = \sum_{n\geqslant 0}\frac{z^n}{n!}\left[{m \atop m-n}\right]\Big/\binom{m-1}{n} \tag{7.52}$$

$$e^{z+wz} = \sum_{m,n\geqslant 0}\binom{n}{m}w^m\frac{z^n}{n!} \tag{7.53}$$

$$e^{w(e^z-1)} = \sum_{m,n\geqslant 0}\left\{{n \atop m}\right\}w^m\frac{z^n}{n!} \tag{7.54}$$

$$\frac{1}{(1-z)^w} = \sum_{m,n\geqslant 0}\left[{n \atop m}\right]w^m\frac{z^n}{n!} \tag{7.55}$$

$$\frac{1-w}{e^{(w-1)z}-w} = \sum_{m,n\geqslant 0}\left\langle{n \atop m}\right\rangle w^m\frac{z^n}{n!} \tag{7.56}$$

Table 352 is the database we need. The identities in this table are not difficult to prove, so we needn't dwell on them; this table is primarily for reference when we meet a new problem. But there's a nice proof of the first formula, (7.43), that deserves mention: We start with the identity

$$\frac{1}{(1-z)^{x+1}} = \sum_n \binom{x+n}{n} z^n$$

and differentiate it with respect to x. On the left, $(1-z)^{-x-1}$ is equal to $e^{(x+1)\ln(1/(1-z))}$, so d/dx contributes a factor of $\ln\bigl(1/(1-z)\bigr)$. On the right, the numerator of $\binom{x+n}{n}$ is $(x+n)\ldots(x+1)$, and d/dx splits this into n terms whose sum is equivalent to multiplying $\binom{x+n}{n}$ by

$$\frac{1}{x+n} + \cdots + \frac{1}{x+1} = H_{x+n} - H_x\,.$$

Replacing x by m gives (7.43). Notice that $H_{x+n} - H_x$ is meaningful even when x is not an integer.

By the way, this method of differentiating a complicated product — leaving it as a product — is usually better than expressing the derivative as a sum. For example the right side of

$$\frac{d}{dx}\bigl((x+n)^n\ldots(x+1)^1\bigr) = (x+n)^n\ldots(x+1)^1\left(\frac{n}{x+n}+\cdots+\frac{1}{x+1}\right)$$

would be a lot messier written out as a sum.

The general identities in Table 352 include many important special cases. For example, (7.43) simplifies to the generating function for H_n when $m=0$:

$$\frac{1}{1-z}\ln\frac{1}{1-z} = \sum_n H_n z^n\,. \tag{7.57}$$

This equation can also be derived in other ways; for example, we can take the power series for $\ln\bigl(1/(1-z)\bigr)$ and divide it by $1-z$ to get cumulative sums.

Identities (7.51) and (7.52) involve the respective ratios $\left\{{m \atop m-n}\right\}\Big/\binom{m-1}{n}$ and $\left[{m \atop m-n}\right]\Big/\binom{m-1}{n}$, which have the undefined form $0/0$ when $n \geqslant m$. However, there is a way to give them a proper meaning using the Stirling polynomials of (6.45), because we have

$$\left\{{m \atop m-n}\right\}\Big/\binom{m-1}{n} = (-1)^{n+1} n!\, m\sigma_n(n-m)\,; \tag{7.58}$$

$$\left[{m \atop m-n}\right]\Big/\binom{m-1}{n} = n!\, m\sigma_n(m)\,. \tag{7.59}$$

Thus, for example, the case $m=1$ of (7.51) should not be regarded as the power series $\sum_{n\geqslant 0}(z^n/n!)\left\{{1 \atop 1-n}\right\}\Big/\binom{0}{n}$, but rather as

$$\frac{z}{\ln(1+z)} = -\sum_{n\geqslant 0}(-z)^n\sigma_n(n-1) = 1 + \tfrac{1}{2}z - \tfrac{1}{12}z^2 + \cdots\,.$$

Identities (7.53), (7.55), (7.54), and (7.56) are "double generating functions" or "super generating functions" because they have the form $G(w,z) = \sum_{m,n} g_{m,n} w^m z^n$. The coefficient of w^m is a generating function in the variable z; the coefficient of z^n is a generating function in the variable w. Equation (7.56) can be put into the more symmetrical form

$$\frac{e^w - e^z}{we^z - ze^w} = \sum_{m,n} \left\langle {m+n+1 \atop m} \right\rangle \frac{w^m z^n}{(m+n+1)!} . \tag{7.60}$$

7.5 CONVOLUTIONS

The *convolution* of two given sequences $\langle f_0, f_1, \ldots \rangle = \langle f_n \rangle$ and $\langle g_0, g_1, \ldots \rangle = \langle g_n \rangle$ is the sequence $\langle f_0 g_0,\ f_0 g_1 + f_1 g_0,\ \ldots \rangle = \langle \sum_k f_k g_{n-k} \rangle$. We have observed in Sections 5.4 and 7.2 that convolution of sequences corresponds to multiplication of their generating functions. This fact makes it easy to evaluate many sums that would otherwise be difficult to handle.

I always thought convolution was what happens to my brain when I try to do a proof.

Example 1: A Fibonacci convolution.

For example, let's try to evaluate $\sum_{k=0}^{n} F_k F_{n-k}$ in closed form. This is the convolution of $\langle F_n \rangle$ with itself, so the sum must be the coefficient of z^n in $F(z)^2$, where $F(z)$ is the generating function for $\langle F_n \rangle$. All we have to do is figure out the value of this coefficient.

The generating function $F(z)$ is $z/(1-z-z^2)$, a quotient of polynomials; so the general expansion theorem for rational functions tells us that the answer can be obtained from a partial fraction representation. We can use the general expansion theorem (7.30) and grind away; or we can use the fact that

$$\begin{aligned} F(z)^2 &= \left(\frac{1}{\sqrt{5}} \left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi} z} \right) \right)^2 \\ &= \frac{1}{5} \left(\frac{1}{(1-\phi z)^2} - \frac{2}{(1-\phi z)(1-\hat{\phi} z)} + \frac{1}{(1-\hat{\phi} z)^2} \right) \\ &= \frac{1}{5} \sum_{n \geqslant 0} (n+1) \phi^n z^n - \frac{2}{5} \sum_{n \geqslant 0} F_{n+1} z^n + \frac{1}{5} \sum_{n \geqslant 0} (n+1) \hat{\phi}^n z^n . \end{aligned}$$

Instead of expressing the answer in terms of ϕ and $\hat{\phi}$, let's try for a closed form in terms of Fibonacci numbers. Recalling that $\phi + \hat{\phi} = 1$, we have

$$\begin{aligned} \phi^n + \hat{\phi}^n &= [z^n] \left(\frac{1}{1-\phi z} + \frac{1}{1-\hat{\phi} z} \right) \\ &= [z^n] \frac{2 - (\phi + \hat{\phi}) z}{(1-\phi z)(1-\hat{\phi} z)} = [z^n] \frac{2-z}{1-z-z^2} = 2F_{n+1} - F_n . \end{aligned}$$

Hence

$$F(z)^2 = \frac{1}{5}\sum_{n\geqslant 0}(n+1)(2F_{n+1}-F_n)z^n - \frac{2}{5}\sum_{n\geqslant 0}F_{n+1}\,z^n\,,$$

and we have the answer we seek:

$$\sum_{k=0}^{n}F_kF_{n-k} = \frac{2nF_{n+1}-(n+1)F_n}{5}\,. \tag{7.61}$$

For example, when $n = 3$ this formula gives $F_0F_3 + F_1F_2 + F_2F_1 + F_3F_0 = 0+1+1+0 = 2$ on the left and $(6F_4 - 4F_3)/5 = (18-8)/5 = 2$ on the right.

Example 2: Harmonic convolutions.

The efficiency of a certain computer method called "samplesort" depends on the value of the sum

$$T_{m,n} = \sum_{0\leqslant k<n}\binom{k}{m}\frac{1}{n-k}\,, \qquad \text{integers } m,n \geqslant 0.$$

Exercise 5.58 obtains the value of this sum by a somewhat intricate double induction, using summation factors. It's much easier to realize that $T_{m,n}$ is just the nth term in the convolution of $\langle\binom{0}{m},\binom{1}{m},\binom{2}{m},\ldots\rangle$ with $\langle 0,\frac{1}{1},\frac{1}{2},\ldots\rangle$. Both sequences have simple generating functions in Table 336:

$$\sum_{n\geqslant 0}\binom{n}{m}z^n = \frac{z^m}{(1-z)^{m+1}}\,; \qquad \sum_{n>0}\frac{z^n}{n} = \ln\frac{1}{1-z}\,.$$

Therefore, by (7.43),

$$\begin{aligned} T_{m,n} &= [z^n]\,\frac{z^m}{(1-z)^{m+1}}\ln\frac{1}{1-z} = [z^{n-m}]\,\frac{1}{(1-z)^{m+1}}\ln\frac{1}{1-z} \\ &= (H_n - H_m)\binom{n}{n-m}\,. \end{aligned}$$

In fact, there are many more sums that boil down to this same sort of convolution, because we have

$$\frac{1}{(1-z)^{r+1}}\ln\frac{1}{1-z}\cdot\frac{1}{(1-z)^{s+1}} = \frac{1}{(1-z)^{r+s+2}}\ln\frac{1}{1-z}$$

for all r and s. Equating coefficients of z^n gives the general identity

$$\begin{aligned} &\sum_k\binom{r+k}{k}\binom{s+n-k}{n-k}(H_{r+k}-H_r) \\ &\qquad = \binom{r+s+n+1}{n}(H_{r+s+n+1}-H_{r+s+1})\,. \end{aligned} \tag{7.62}$$

This seems almost too good to be true. But it checks, at least when $n = 2$:

Because it's so harmonic.

$$\binom{r+1}{1}\binom{s+1}{1}\frac{1}{r+1} + \binom{r+2}{2}\binom{s+0}{0}\left(\frac{1}{r+2} + \frac{1}{r+1}\right)$$
$$= \binom{r+s+3}{2}\left(\frac{1}{r+s+3} + \frac{1}{r+s+2}\right).$$

Special cases like $s = 0$ are as remarkable as the general case.

And there's more. We can use the convolution identity

$$\sum_k \binom{r+k}{k}\binom{s+n-k}{n-k} = \binom{r+s+n+1}{n}$$

to transpose H_r to the other side, since H_r is independent of k:

$$\sum_k \binom{r+k}{k}\binom{s+n-k}{n-k} H_{r+k}$$
$$= \binom{r+s+n+1}{n}(H_{r+s+n+1} - H_{r+s+1} + H_r)\,. \tag{7.63}$$

There's still more: If r and s are nonnegative integers l and m, we can replace $\binom{r+k}{k}$ by $\binom{l+k}{l}$ and $\binom{s+n-k}{n-k}$ by $\binom{m+n-k}{m}$; then we can change k to $k-l$ and n to $n-m-l$, getting

$$\sum_{k=0}^{n} \binom{k}{l}\binom{n-k}{m} H_k = \binom{n+1}{l+m+1}(H_{n+1} - H_{l+m+1} + H_l)\,,$$
$$\text{integers } l, m, n \geqslant 0. \tag{7.64}$$

Even the special case $l = m = 0$ of this identity was difficult for us to handle in Chapter 2! (See (2.36).) We've come a long way.

Example 3: Convolutions of convolutions.

If we form the convolution of $\langle f_n \rangle$ and $\langle g_n \rangle$, then convolve this with a third sequence $\langle h_n \rangle$, we get a sequence whose nth term is

$$\sum_{j+k+l=n} f_j\, g_k\, h_l\,.$$

The generating function of this three-fold convolution is, of course, the three-fold product $F(z)G(z)H(z)$. In a similar way, the m-fold convolution of a sequence $\langle g_n \rangle$ with itself has nth term equal to

$$\sum_{k_1+k_2+\cdots+k_m=n} g_{k_1}\, g_{k_2} \cdots g_{k_m}$$

and its generating function is $G(z)^m$.

We can apply these observations to the spans-of-fans problem considered earlier (Example 6 in Section 7.3). It turns out that there's another way to compute f_n, the number of spanning trees of an n-fan, based on the configurations of tree edges between the vertices $\{1, 2, \ldots, n\}$: The edge between vertex k and vertex $k+1$ may or may not be selected for the tree; and each of the ways to select these edges connects up certain blocks of adjacent vertices. For example, when $n = 10$ we might connect vertices $\{1, 2\}$, $\{3\}$, $\{4, 5, 6, 7\}$, and $\{8, 9, 10\}$:

Concrete blocks.

How many spanning trees can we make, by adding additional edges to vertex 0? We need to connect 0 to each of the four blocks; and there are two ways to join 0 with $\{1, 2\}$, one way to join it with $\{3\}$, four ways with $\{4, 5, 6, 7\}$, and three ways with $\{8, 9, 10\}$, or $2 \cdot 1 \cdot 4 \cdot 3 = 24$ ways altogether. Summing over all possible ways to make blocks gives us the following expression for the total number of spanning trees:

$$f_n = \sum_{m>0} \sum_{\substack{k_1+k_2+\cdots+k_m=n \\ k_1,k_2,\ldots,k_m>0}} k_1 k_2 \ldots k_m . \tag{7.65}$$

For example, $f_4 = 4 + 3\cdot1 + 2\cdot2 + 1\cdot3 + 2\cdot1\cdot1 + 1\cdot2\cdot1 + 1\cdot1\cdot2 + 1\cdot1\cdot1\cdot1 = 21$.

This is the sum of m-fold convolutions of the sequence $\langle 0, 1, 2, 3, \ldots \rangle$, for $m = 1, 2, 3, \ldots$; hence the generating function for $\langle f_n \rangle$ is

$$F(z) = G(z) + G(z)^2 + G(z)^3 + \cdots = \frac{G(z)}{1 - G(z)}$$

where $G(z)$ is the generating function for $\langle 0, 1, 2, 3, \ldots \rangle$, namely $z/(1-z)^2$. Consequently we have

$$F(z) = \frac{z}{(1-z)^2 - z} = \frac{z}{1 - 3z + z^2},$$

as before. This approach to $\langle f_n \rangle$ is more symmetrical and appealing than the complicated recurrence we had earlier.

Example 4: A convoluted recurrence.

Our next example is especially important. In fact, it's the "classic example" of why generating functions are useful in the solution of recurrences.

Suppose we have $n+1$ variables $x_0, x_1, \ldots, x_n$ whose product is to be computed by doing n multiplications. How many ways C_n are there to insert parentheses into the product $x_0 \cdot x_1 \cdot \ldots \cdot x_n$ so that the order of multiplication is completely specified? For example, when $n = 2$ there are two ways, $x_0 \cdot (x_1 \cdot x_2)$ and $(x_0 \cdot x_1) \cdot x_2$. And when $n = 3$ there are five ways,

$$x_0 \cdot (x_1 \cdot (x_2 \cdot x_3)), \quad x_0 \cdot ((x_1 \cdot x_2) \cdot x_3), \quad (x_0 \cdot x_1) \cdot (x_2 \cdot x_3),$$
$$(x_0 \cdot (x_1 \cdot x_2)) \cdot x_3, \quad ((x_0 \cdot x_1) \cdot x_2) \cdot x_3 .$$

Thus $C_2 = 2$, $C_3 = 5$; we also have $C_1 = 1$ and $C_0 = 1$.

Let's use the four-step procedure of Section 7.3. What is a recurrence for the C's? The key observation is that there's exactly one '$\cdot$' operation outside all of the parentheses, when $n > 0$; this is the final multiplication that ties everything together. If this '$\cdot$' occurs between x_k and x_{k+1}, there are C_k ways to fully parenthesize $x_0 \cdot \ldots \cdot x_k$, and there are C_{n-k-1} ways to fully parenthesize $x_{k+1} \cdot \ldots \cdot x_n$; hence

$$C_n = C_0 C_{n-1} + C_1 C_{n-2} + \cdots + C_{n-1} C_0 , \qquad \text{if } n > 0.$$

By now we recognize this expression as a convolution, and we know how to patch the formula so that it holds for all integers n:

$$C_n = \sum_k C_k C_{n-1-k} + [n=0] . \tag{7.66}$$

Step 1 is now complete. Step 2 tells us to multiply by z^n and sum:

$$\begin{aligned} C(z) &= \sum_n C_n z^n \\ &= \sum_{k,n} C_k C_{n-1-k} z^n + \sum_{n=0} z^n \\ &= \sum_k C_k z^k \sum_n C_{n-1-k} z^{n-k} + 1 \\ &= C(z) \cdot z C(z) + 1 . \end{aligned}$$

Lo and behold, the convolution has become a product, in the generating-function world. Life is full of surprises.

The authors jest.

Step 3 is also easy. We solve for $C(z)$ by the quadratic formula:

$$C(z) = \frac{1 \pm \sqrt{1-4z}}{2z}.$$

But should we choose the $+$ sign or the $-$ sign? Both choices yield a function that satisfies $C(z) = zC(z)^2 + 1$, but only one of the choices is suitable for our problem. We might choose the $+$ sign on the grounds that positive thinking is best; but we soon discover that this choice gives $C(0) = \infty$, contrary to the facts. (The correct function $C(z)$ is supposed to have $C(0) = C_0 = 1$.) Therefore we conclude that

$$C(z) = \frac{1 - \sqrt{1-4z}}{2z}.$$

Finally, Step 4. What is $[z^n]\,C(z)$? The binomial theorem tells us that

$$\sqrt{1-4z} = \sum_{k\geqslant 0} \binom{1/2}{k} (-4z)^k = 1 + \sum_{k\geqslant 1} \frac{1}{2k}\binom{-1/2}{k-1}(-4z)^k;$$

hence, using (5.37),

$$\begin{aligned}\frac{1-\sqrt{1-4z}}{2z} &= \sum_{k\geqslant 1} \frac{1}{k}\binom{-1/2}{k-1}(-4z)^{k-1} \\ &= \sum_{n\geqslant 0} \binom{-1/2}{n}\frac{(-4z)^n}{n+1} = \sum_{n\geqslant 0}\binom{2n}{n}\frac{z^n}{n+1}.\end{aligned}$$

The number of ways to parenthesize, C_n, is $\binom{2n}{n}\frac{1}{n+1}$.

So the convoluted recurrence has led us to an oft-recurring convolution.

We anticipated this result in Chapter 5, when we introduced the sequence of *Catalan numbers* $\langle 1, 1, 2, 5, 14, \ldots\rangle = \langle C_n\rangle$. This sequence arises in dozens of problems that seem at first to be unrelated to each other [46], because many situations have a recursive structure that corresponds to the convolution recurrence (7.66).

For example, let's consider the following problem: How many sequences $\langle a_1, a_2 \ldots, a_{2n}\rangle$ of $+1$'s and -1's have the property that

$$a_1 + a_2 + \cdots + a_{2n} = 0$$

and have all their partial sums

$$a_1, \quad a_1 + a_2, \quad \ldots, \quad a_1 + a_2 + \cdots + a_{2n}$$

nonnegative? There must be n occurrences of $+1$ and n occurrences of -1. We can represent this problem graphically by plotting the sequence of partial

sums $s_n = \sum_{k=1}^{n} a_k$ as a function of n: The five solutions for $n = 3$ are

These are "mountain ranges" of width $2n$ that can be drawn with line segments of the forms and . It turns out that there are exactly C_n ways to do this, and the sequences can be related to the parenthesis problem in the following way: Put an extra pair of parentheses around the entire formula, so that there are n pairs of parentheses corresponding to the n multiplications. Now replace each '$\cdot$' by $+1$ and each ')' by -1 and erase everything else. For example, the formula $x_0 \cdot \big((x_1 \cdot x_2) \cdot (x_3 \cdot x_4)\big)$ corresponds to the sequence $\langle +1, +1, -1, +1, +1, -1, -1, -1 \rangle$ by this rule. The five ways to parenthesize $x_0 \cdot x_1 \cdot x_2 \cdot x_3$ correspond to the five mountain ranges for $n = 3$ shown above.

Moreover, a slight reformulation of our sequence-counting problem leads to a surprisingly simple combinatorial solution that avoids the use of generating functions: How many sequences $\langle a_0, a_1, a_2, \ldots, a_{2n} \rangle$ of $+1$'s and -1's have the property that

$$a_0 + a_1 + a_2 + \cdots + a_{2n} \;=\; 1\,,$$

when all the partial sums

$$a_0, \quad a_0 + a_1, \quad a_0 + a_1 + a_2, \quad \ldots, \quad a_0 + a_1 + \cdots + a_{2n}$$

are required to be *positive*? Clearly these are just the sequences of the previous problem, with the additional element $a_0 = +1$ placed in front. But the sequences in the new problem can be enumerated by a simple counting argument, using a remarkable fact discovered by George Raney [302] in 1959: *If $\langle x_1, x_2, \ldots, x_m \rangle$ is any sequence of integers whose sum is $+1$, exactly one of the cyclic shifts*

$$\langle x_1, x_2, \ldots, x_m \rangle, \;\; \langle x_2, \ldots, x_m, x_1 \rangle, \;\; \ldots, \;\; \langle x_m, x_1, \ldots, x_{m-1} \rangle$$

has all of its partial sums positive. For example, consider the sequence $\langle 3, -5, 2, -2, 3, 0 \rangle$. Its cyclic shifts are

$$\begin{array}{ll}
\langle 3, -5, 2, -2, 3, 0 \rangle & \langle -2, 3, 0, 3, -5, 2 \rangle \\
\langle -5, 2, -2, 3, 0, 3 \rangle & \langle 3, 0, 3, -5, 2, -2 \rangle \;\checkmark \\
\langle 2, -2, 3, 0, 3, -5 \rangle & \langle 0, 3, -5, 2, -2, 3 \rangle
\end{array}$$

and only the one that's checked has entirely positive partial sums.

Raney's lemma can be proved by a simple geometric argument. Let's extend the sequence periodically to get an infinite sequence

$$\langle x_1, x_2, \ldots, x_m, x_1, x_2, \ldots, x_m, x_1, x_2, \ldots \rangle;$$

thus we let $x_{m+k} = x_k$ for all $k \geqslant 0$. If we now plot the partial sums $s_n = x_1 + \cdots + x_n$ as a function of n, the graph of s_n has an "average slope" of $1/m$, because $s_{m+n} = s_n + 1$. For example, the graph corresponding to our example sequence $\langle 3, -5, 2, -2, 3, 0, 3, -5, 2, \ldots \rangle$ begins as follows:

Ah, if stock prices would only continue to rise like this.

The entire graph can be contained between two lines of slope $1/m$, as shown; we have $m = 6$ in the illustration. In general these bounding lines touch the graph just once in each cycle of m points, since lines of slope $1/m$ hit points with integer coordinates only once per m units. The unique lower point of intersection is the only place in the cycle from which all partial sums will be positive, because every other point on the curve has an intersection point within m units to its right.

(Attention, computer scientists: The partial sums in this problem represent the stack size as a function of time, when a product of $n+1$ factors is evaluated, because each "push" operation changes the size by $+1$ and each multiplication changes it by -1.)

With Raney's lemma we can easily enumerate the sequences $\langle a_0, \ldots, a_{2n} \rangle$ of $+1$'s and -1's whose partial sums are entirely positive and whose total sum is $+1$. There are $\binom{2n+1}{n}$ sequences with n occurrences of -1 and $n+1$ occurrences of $+1$, and Raney's lemma tells us that exactly $1/(2n+1)$ of these sequences have all partial sums positive. (List all $N = \binom{2n+1}{n}$ of these sequences and all $2n+1$ of their cyclic shifts, in an $N \times (2n+1)$ array. Each row contains exactly one solution. Each solution appears exactly once in each column. So there are $N/(2n+1)$ distinct solutions in the array, each appearing $(2n+1)$ times.) The total number of sequences with positive partial sums is

$$\binom{2n+1}{n}\frac{1}{2n+1} = \binom{2n}{n}\frac{1}{n+1} = C_n\,.$$

Example 5: A recurrence with m-fold convolution.

We can generalize the problem just considered by looking at sequences $\langle a_0, \ldots, a_{mn} \rangle$ of $+1$'s and $(1-m)$'s whose partial sums are all positive and

whose total sum is $+1$. Such sequences can be called m-*Raney sequences*. If there are k occurrences of $(1-m)$ and $mn+1-k$ occurrences of $+1$, we have

$$k(1-m)+(mn+1-k) \;=\; 1\,,$$

hence $k=n$. There are $\binom{mn+1}{n}$ sequences with n occurrences of $(1-m)$ and $mn+1-n$ occurrences of $+1$, and Raney's lemma tells us that the number of such sequences with all partial sums positive is exactly

(Attention, computer scientists: The stack interpretation now applies with respect to an m-ary operation, instead of the binary multiplication considered earlier.)

$$\binom{mn+1}{n}\frac{1}{mn+1} \;=\; \binom{mn}{n}\frac{1}{(m-1)n+1}\,. \tag{7.67}$$

So this is the number of m-Raney sequences. Let's call this a Fuss-Catalan number $C_n^{(m)}$, because the sequence $\langle C_n^{(m)}\rangle$ was first investigated by N. I. Fuss [135] in 1791 (many years before Catalan himself got into the act). The ordinary Catalan numbers are $C_n = C_n^{(2)}$.

Now that we know the answer, (7.67), let's play "Jeopardy" and figure out a question that leads to it. In the case $m=2$ the question was: "What numbers C_n satisfy the recurrence $C_n = \sum_k C_k C_{n-1-k} + [n=0]$?" We will try to find a similar question (a similar recurrence) in the general case.

The trivial sequence $\langle +1\rangle$ of length 1 is clearly an m-Raney sequence. If we put the number $(1-m)$ at the right of any m sequences that are m-Raney, we get an m-Raney sequence; the partial sums stay positive as they increase to $+2$, then $+3$, ..., $+m$, and $+1$. Conversely, we can show that all m-Raney sequences $\langle a_0,\ldots,a_{mn}\rangle$ arise in this way, if $n>0$: The last term a_{mn} must be $(1-m)$. The partial sums $s_j = a_0+\cdots+a_{j-1}$ are positive for $1\leqslant j\leqslant mn$, and $s_{mn}=m$ because $s_{mn}+a_{mn}=1$. Let k_1 be the largest index $\leqslant mn$ such that $s_{k_1}=1$; let k_2 be largest such that $s_{k_2}=2$; and so on. Thus $s_{k_j}=j$ and $s_k>j$, for $k_j<k\leqslant mn$ and $1\leqslant j\leqslant m$. It follows that $k_m=mn$, and we can verify without difficulty that each of the subsequences $\langle a_0,\ldots,a_{k_1-1}\rangle$, $\langle a_{k_1},\ldots,a_{k_2-1}\rangle$, ..., $\langle a_{k_{m-1}},\ldots,a_{k_m-1}\rangle$ is an m-Raney sequence. We must have $k_1 = mn_1+1$, $k_2-k_1 = mn_2+1$, ..., $k_m-k_{m-1} = mn_m+1$, for some nonnegative integers n_1, n_2, ..., n_m.

Therefore $\binom{mn+1}{n}\frac{1}{mn+1}$ is the answer to the following two interesting questions: "What are the numbers $C_n^{(m)}$ defined by the recurrence

$$C_n^{(m)} \;=\; \Bigg(\sum_{n_1+n_2+\cdots+n_m=n-1} C_{n_1}^{(m)}C_{n_2}^{(m)}\ldots C_{n_m}^{(m)}\Bigg) \;+\; [n=0] \tag{7.68}$$

for all integers n?" "If $G(z)$ is a power series that satisfies

$$G(z) \;=\; z\,G(z)^m \;+\; 1\,, \tag{7.69}$$

what is $[z^n]\,G(z)$?"

Notice that these are not easy questions. In the ordinary Catalan case ($m = 2$), we solved (7.69) for $G(z)$ and its coefficients by using the quadratic formula and the binomial theorem; but when $m = 3$, none of the standard techniques gives any clue about how to solve the cubic equation $G = zG^3 + 1$. So it has turned out to be easier to answer this question before asking it.

Now, however, we know enough to ask even harder questions and deduce their answers. How about this one: "What is $[z^n]\,G(z)^l$, if l is a positive integer and if $G(z)$ is the power series defined by (7.69)?" The argument we just gave can be used to show that $[z^n]\,G(z)^l$ is the number of sequences of length $mn + l$ with the following three properties:

- Each element is either $+1$ or $(1 - m)$.
- The partial sums are all positive.
- The total sum is l.

For we get all such sequences in a unique way by putting together l sequences that have the m-Raney property. The number of ways to do this is

$$\sum_{n_1+n_2+\cdots+n_l=n} C_{n_1}^{(m)} C_{n_2}^{(m)} \ldots C_{n_l}^{(m)} \;=\; [z^n]\,G(z)^l\,.$$

Raney proved a generalization of his lemma that tells us how to count such sequences: *If $\langle x_1, x_2, \ldots, x_m\rangle$ is any sequence of integers with $x_j \leqslant 1$ for all j, and with $x_1 + x_2 + \cdots + x_m = l > 0$, then exactly l of the cyclic shifts*

$$\langle x_1, x_2, \ldots, x_m\rangle,\quad \langle x_2, \ldots, x_m, x_1\rangle,\quad \ldots,\quad \langle x_m, x_1, \ldots, x_{m-1}\rangle$$

have all positive partial sums.

For example, we can check this statement on the sequence $\langle -2, 1, -1, 0, 1, 1, -1, 1, 1, 1\rangle$. The cyclic shifts are

$\langle -2, 1, -1, 0, 1, 1, -1, 1, 1, 1\rangle$	$\langle 1, -1, 1, 1, 1, -2, 1, -1, 0, 1\rangle$
$\langle 1, -1, 0, 1, 1, -1, 1, 1, 1, -2\rangle$	$\langle -1, 1, 1, 1, -2, 1, -1, 0, 1, 1\rangle$
$\langle -1, 0, 1, 1, -1, 1, 1, 1, -2, 1\rangle$	$\langle 1, 1, 1, -2, 1, -1, 0, 1, 1, -1\rangle$ ✓
$\langle 0, 1, 1, -1, 1, 1, 1, -2, 1, -1\rangle$	$\langle 1, 1, -2, 1, -1, 0, 1, 1, -1, 1\rangle$
$\langle 1, 1, -1, 1, 1, 1, -2, 1, -1, 0\rangle$ ✓	$\langle 1, -2, 1, -1, 0, 1, 1, -1, 1, 1\rangle$

and only the two examples marked '✓' have all partial sums positive. This generalized lemma is proved in exercise 13.

A sequence of $+1$'s and $(1 - m)$'s that has length $mn + l$ and total sum l must have exactly n occurrences of $(1 - m)$. The generalized lemma tells us that $l/(mn + l)$ of these $\binom{mn+l}{n}$ sequences have all partial sums positive;

hence our tough question has a surprisingly simple answer:

$$[z^n]\,G(z)^l \;=\; \binom{mn+l}{n}\frac{l}{mn+l}\,, \tag{7.70}$$

for all integers $l > 0$.

Readers who haven't forgotten Chapter 5 might well be experiencing *déjà vu*: "That formula looks familiar; haven't we seen it before?" Yes, indeed; Lambert's equation (5.60) says that

$$[z^n]\,\mathcal{B}_t(z)^r \;=\; \binom{tn+r}{n}\frac{r}{tn+r}\,.$$

Therefore the generating function $G(z)$ in (7.69) must actually be the generalized binomial series $\mathcal{B}_m(z)$. Sure enough, equation (5.59) says

$$\mathcal{B}_m(z)^{1-m} - \mathcal{B}_m(z)^{-m} \;=\; z\,,$$

which is the same as

$$\mathcal{B}_m(z) - 1 \;=\; z\mathcal{B}_m(z)^m\,.$$

Let's switch to the notation of Chapter 5, now that we know we're dealing with generalized binomials. Chapter 5 stated a bunch of identities without proof. We have now closed part of the gap by proving that the power series $\mathcal{B}_t(z)$ defined by

$$\mathcal{B}_t(z) \;=\; \sum_n \binom{tn+1}{n}\frac{z^n}{tn+1}$$

has the remarkable property that

$$\mathcal{B}_t(z)^r \;=\; \sum_n \binom{tn+r}{n}\frac{r\,z^n}{tn+r}\,,$$

whenever t and r are positive integers.

Can we extend these results to *arbitrary* values of t and r? Yes; because the coefficients $\binom{tn+r}{n}\frac{r}{tn+r}$ are polynomials in t and r. The general rth power defined by

$$\mathcal{B}_t(z)^r \;=\; e^{r\ln\mathcal{B}_t(z)} \;=\; \sum_{n\geqslant 0}\frac{\bigl(r\ln\mathcal{B}_t(z)\bigr)^n}{n!} \;=\; \sum_{n\geqslant 0}\frac{r^n}{n!}\biggl(-\sum_{m\geqslant 1}\frac{\bigl(1-\mathcal{B}_t(z)\bigr)^m}{m}\biggr)^n$$

has coefficients that are polynomials in t and r; and those polynomials are equal to $\binom{tn+r}{n}\frac{r}{tn+r}$ for infinitely many values of t and r. So the two sequences of polynomials must be identically equal.

Chapter 5 also mentions the generalized exponential series

$$\mathcal{E}_t(z) \;=\; \sum_{n\geqslant 0} \frac{(tn+1)^{n-1}}{n!} z^n\,,$$

which is said in (5.60) to have an equally remarkable property:

$$[z^n]\,\mathcal{E}_t(z)^r \;=\; \frac{r(tn+r)^{n-1}}{n!}\,. \tag{7.71}$$

We can prove this as a limiting case of the formulas for $\mathcal{B}_t(z)$, because it is not difficult to show that

$$\mathcal{E}_t(z)^r \;=\; \lim_{x\to\infty} \mathcal{B}_{xt}(z/x)^{xr}\,.$$

7.6 EXPONENTIAL GF'S

Sometimes a sequence $\langle g_n\rangle$ has a generating function whose properties are quite complicated, while the related sequence $\langle g_n/n!\rangle$ has a generating function that's quite simple. In such cases we naturally prefer to work with $\langle g_n/n!\rangle$ and then multiply by $n!$ at the end. This trick works sufficiently often that we have a special name for it: We call the power series

$$\widehat{G}(z) \;=\; \sum_{n\geqslant 0} g_n \frac{z^n}{n!} \tag{7.72}$$

the *exponential generating function* or "egf" of the sequence $\langle g_0, g_1, g_2, \ldots\rangle$. This name arises because the exponential function e^z is the egf of $\langle 1, 1, 1, \ldots\rangle$.

Many of the generating functions in Table 352 are actually egf's. For example, equation (7.50) says that $\left(\ln\frac{1}{1-z}\right)^m/m!$ is the egf for the sequence $\langle \left[{0 \atop m}\right], \left[{1 \atop m}\right], \left[{2 \atop m}\right], \ldots\rangle$. The ordinary generating function for this sequence is much more complicated (and also divergent).

Exponential generating functions have their own basic maneuvers, analogous to the operations we learned in Section 7.2. For example, if we multiply the egf of $\langle g_n\rangle$ by z, we get

$$\sum_{n\geqslant 0} g_n \frac{z^{n+1}}{n!} \;=\; \sum_{n\geqslant 1} g_{n-1} \frac{z^n}{(n-1)!} \;=\; \sum_{n\geqslant 0} n g_{n-1} \frac{z^n}{n!}\,;$$

this is the egf of $\langle 0, g_0, 2g_1, \ldots\rangle = \langle n g_{n-1}\rangle$.

Are we having fun yet?

Differentiating the egf of $\langle g_0, g_1, g_2, \ldots\rangle$ with respect to z gives

$$\sum_{n\geqslant 0} n g_n \frac{z^{n-1}}{n!} \;=\; \sum_{n\geqslant 1} g_n \frac{z^{n-1}}{(n-1)!} \;=\; \sum_{n\geqslant 0} g_{n+1} \frac{z^n}{n!}\,; \tag{7.73}$$

this is the egf of $\langle g_1, g_2, \ldots \rangle$. Thus differentiation on egf's corresponds to the left-shift operation $\big(G(z) - g_0\big)/z$ on ordinary gf's. (We used this left-shift property of egf's when we studied hypergeometric series, (5.106).) Integration of an egf gives

$$\int_0^z \sum_{n \geqslant 0} g_n \frac{t^n}{n!}\, dt \;=\; \sum_{n \geqslant 0} g_n \frac{z^{n+1}}{(n+1)!} \;=\; \sum_{n \geqslant 1} g_{n-1} \frac{z^n}{n!}\,; \tag{7.74}$$

this is a right shift, the egf of $\langle 0, g_0, g_1, \ldots \rangle$.

The most interesting operation on egf's, as on ordinary gf's, is multiplication. If $\widehat{F}(z)$ and $\widehat{G}(z)$ are egf's for $\langle f_n \rangle$ and $\langle g_n \rangle$, then $\widehat{F}(z)\widehat{G}(z) = \widehat{H}(z)$ is the egf for a sequence $\langle h_n \rangle$ called the *binomial convolution* of $\langle f_n \rangle$ and $\langle g_n \rangle$:

$$h_n \;=\; \sum_k \binom{n}{k} f_k\, g_{n-k}\,. \tag{7.75}$$

Binomial coefficients appear here because $\binom{n}{k} = n!/k!\,(n-k)!$, hence

$$\frac{h_n}{n!} \;=\; \sum_{k=0}^{n} \frac{f_k}{k!} \frac{g_{n-k}}{(n-k)!}\,;$$

in other words, $\langle h_n/n! \rangle$ is the ordinary convolution of $\langle f_n/n! \rangle$ and $\langle g_n/n! \rangle$.

Binomial convolutions occur frequently in applications. For example, we defined the Bernoulli numbers in (6.79) by the implicit recurrence

$$\sum_{j=0}^{m} \binom{m+1}{j} B_j \;=\; [m=0]\,, \qquad \text{for all } m \geqslant 0;$$

this can be rewritten as a binomial convolution, if we substitute n for $m+1$ and add the term B_n to both sides:

$$\sum_k \binom{n}{k} B_k \;=\; B_n + [n=1]\,, \qquad \text{for all } n \geqslant 0. \tag{7.76}$$

We can now relate this recurrence to power series (as promised in Chapter 6) by introducing the egf for Bernoulli numbers, $\widehat{B}(z) = \sum_{n \geqslant 0} B_n z^n/n!$. The left-hand side of (7.76) is the binomial convolution of $\langle B_n \rangle$ with the constant sequence $\langle 1, 1, 1, \ldots \rangle$; hence the egf of the left-hand side is $\widehat{B}(z)e^z$. The egf of the right-hand side is $\sum_{n \geqslant 0} \big(B_n + [n=1]\big) z^n/n! = \widehat{B}(z) + z$. Therefore we must have $\widehat{B}(z) = z/(e^z - 1)$; we have proved equation (6.81), which appears also in Table 352 as equation (7.44).

Now let's look again at a sum that has been popping up frequently in this book,

$$S_m(n) \;=\; 0^m + 1^m + 2^m + \cdots + (n-1)^m \;=\; \sum_{0\leqslant k<n} k^m\,.$$

This time we will try to analyze the problem with generating functions, in hopes that it will suddenly become simpler. We will consider n to be fixed and m variable; thus our goal is to understand the coefficients of the power series

$$S(z) \;=\; S_0(n) + S_1(n)\,z + S_2(n)\,z^2 + \cdots \;=\; \sum_{m\geqslant 0} S_m(n)\,z^m\,.$$

We know that the generating function for $\langle 1, k, k^2, \ldots\rangle$ is

$$\frac{1}{1-kz} \;=\; \sum_{m\geqslant 0} k^m\,z^m\,,$$

hence

$$S(z) \;=\; \sum_{m\geqslant 0}\sum_{0\leqslant k<n} k^m\,z^m \;=\; \sum_{0\leqslant k<n} \frac{1}{1-kz}$$

by interchanging the order of summation. We can put this sum in closed form,

$$\begin{aligned} S(z) \;&=\; \frac{1}{z}\left(\frac{1}{z^{-1}-0} + \frac{1}{z^{-1}-1} + \cdots + \frac{1}{z^{-1}-n+1}\right) \\ &=\; \frac{1}{z}\left(H_{z^{-1}} - H_{z^{-1}-n}\right); \end{aligned} \tag{7.77}$$

but we know nothing about expanding such a closed form in powers of z.

Exponential generating functions come to the rescue. The egf of our sequence $\langle S_0(n), S_1(n), S_2(n), \ldots\rangle$ is

$$\widehat{S}(z,n) \;=\; S_0(n) + S_1(n)\,\frac{z}{1!} + S_2(n)\,\frac{z^2}{2!} + \cdots \;=\; \sum_{m\geqslant 0} S_m(n)\,\frac{z^m}{m!}\,.$$

To get these coefficients $S_m(n)$ we can use the egf for $\langle 1, k, k^2, \ldots\rangle$, namely

$$e^{kz} \;=\; \sum_{m\geqslant 0} k^m\,\frac{z^m}{m!}\,,$$

and we have

$$\widehat{S}(z,n) \;=\; \sum_{m\geqslant 0}\sum_{0\leqslant k<n} k^m\,\frac{z^m}{m!} \;=\; \sum_{0\leqslant k<n} e^{kz}\,.$$

And the latter sum is a geometric progression, so there's a closed form

$$\widehat{S}(z,n) \;=\; \frac{e^{nz}-1}{e^z-1}\,. \tag{7.78}$$

Eureka! All we need to do is figure out the coefficients of this relatively simple function, and we'll know $S_m(n)$, because $S_m(n) = m!\,[z^m]\widehat{S}(z,n)$.

Here's where Bernoulli numbers come into the picture. We observed a moment ago that the egf for Bernoulli numbers is

$$\widehat{B}(z) \;=\; \sum_{k\geqslant 0} B_k\,\frac{z^k}{k!} \;=\; \frac{z}{e^z-1}\,;$$

hence we can write

$$\begin{aligned}\widehat{S}(z,n) \;&=\; \widehat{B}(z)\,\frac{e^{nz}-1}{z}\\ &=\; \Bigl(B_0\frac{z^0}{0!}+B_1\frac{z^1}{1!}+B_2\frac{z^2}{2!}+\cdots\Bigr)\Bigl(n\frac{z^0}{1!}+n^2\frac{z^1}{2!}+n^3\frac{z^2}{3!}+\cdots\Bigr)\,.\end{aligned}$$

The sum $S_m(n)$ is $m!$ times the coefficient of z^m in this product. For example,

$$\begin{aligned}S_0(n) \;&=\; 0!\,\Bigl(B_0\frac{n}{1!\,0!}\Bigr) &&=\; n\,;\\ S_1(n) \;&=\; 1!\,\Bigl(B_0\frac{n^2}{2!\,0!}+B_1\frac{n}{1!\,1!}\Bigr) &&=\; \tfrac12 n^2-\tfrac12 n\,;\\ S_2(n) \;&=\; 2!\,\Bigl(B_0\frac{n^3}{3!\,0!}+B_1\frac{n^2}{2!\,1!}+B_2\frac{n}{1!\,2!}\Bigr) &&=\; \tfrac13 n^3-\tfrac12 n^2+\tfrac16 n\,.\end{aligned}$$

We have therefore derived the formula $\square_n = S_2(n) = \frac13 n(n-\frac12)(n-1)$ for the umpteenth time, and this was the simplest derivation of all: In a few lines we have found the general behavior of $S_m(n)$ for all m.

The general formula can be written

$$S_{m-1}(n) \;=\; \frac{1}{m}\bigl(B_m(n)-B_m(0)\bigr)\,, \tag{7.79}$$

where $B_m(x)$ is the *Bernoulli polynomial* defined by

$$B_m(x) \;=\; \sum_k \binom{m}{k} B_k\,x^{m-k}\,. \tag{7.80}$$

Here's why: The Bernoulli polynomial is the binomial convolution of the sequence $\langle B_0, B_1, B_2, \ldots\rangle$ with $\langle 1, x, x^2, \ldots\rangle$; hence the exponential generating

function for $\langle B_0(x), B_1(x), B_2(x), \ldots \rangle$ is the product of their egf's,

$$\widehat{B}(z,x) \;=\; \sum_{m\geqslant 0} B_m(x)\frac{z^m}{m!} \;=\; \frac{z}{e^z-1}\sum_{m\geqslant 0} x^m\frac{z^m}{m!} \;=\; \frac{ze^{xz}}{e^z-1}\,. \tag{7.81}$$

Equation (7.79) follows because the egf for $\langle 0, S_0(n), 2S_1(n), \ldots \rangle$ is, by (7.78),

$$z\,\frac{e^{nz}-1}{e^z-1} \;=\; \widehat{B}(z,n)-\widehat{B}(z,0)\,.$$

Let's turn now to another problem for which egf's are just the thing: How many spanning trees are possible in the *complete graph* on n vertices $\{1,2,\ldots,n\}$? Let's call this number t_n. The complete graph has $\frac{1}{2}n(n-1)$ edges, one edge joining each pair of distinct vertices; so we're essentially looking for the total number of ways to connect up n given things by drawing $n-1$ lines between them.

We have $t_1 = t_2 = 1$. Also $t_3 = 3$, because a complete graph on three vertices is a fan of order 2; we know that $f_2 = 3$. And there are sixteen spanning trees when $n = 4$:

(7.82)

Hence $t_4 = 16$.

Our experience with the analogous problem for fans suggests that the best way to tackle this problem is to single out one vertex, and to look at the blocks or components that the spanning tree joins together when we ignore all edges that touch the special vertex. If the non-special vertices form m components of sizes k_1, k_2, $\ldots$, k_m, then we can connect them to the special vertex in $k_1k_2\ldots k_m$ ways. For example, in the case $n = 4$, we can consider the lower left vertex to be special. The top row of (7.82) shows $3t_3$ cases where the other three vertices are joined among themselves in t_3 ways and then connected to the lower left in 3 ways. The bottom row shows $2{\cdot}1\times t_2t_1\times\binom{3}{2}$ solutions where the other three vertices are divided into components of sizes 2 and 1 in $\binom{3}{2}$ ways; there's also the case where the other three vertices are completely unconnected among themselves.

This line of reasoning leads to the recurrence

$$t_n \;=\; \sum_{m>0}\frac{1}{m!}\sum_{k_1+\cdots+k_m=n-1}\binom{n-1}{k_1,k_2,\ldots,k_m}k_1k_2\ldots k_m\,t_{k_1}t_{k_2}\ldots t_{k_m}$$

for all $n > 1$. Here's why: There are $\binom{n-1}{k_1, k_2, \ldots, k_m}$ ways to assign $n-1$ elements to a sequence of m components of respective sizes $k_1, k_2, \ldots, k_m$; there are $t_{k_1} t_{k_2} \ldots t_{k_m}$ ways to connect up those individual components with spanning trees; there are $k_1 k_2 \ldots k_m$ ways to connect vertex n to those components; and we divide by $m!$ because we want to disregard the order of the components. For example, when $n = 4$ the recurrence says that

$$t_4 \;=\; 3t_3 + \tfrac{1}{2}\big(\tbinom{3}{1,2}2t_1t_2 + \tbinom{3}{2,1}2t_2t_1\big) + \tfrac{1}{6}\big(\tbinom{3}{1,1,1}t_1^3\big) \;=\; 3t_3 + 6t_2t_1 + t_1^3\,.$$

The recurrence for t_n looks formidable at first, possibly even frightening; but it really isn't bad, only convoluted. We can define

$$u_n \;=\; n\,t_n$$

and then everything simplifies considerably:

$$\frac{u_n}{n!} \;=\; \sum_{m>0} \frac{1}{m!} \sum_{k_1+k_2+\cdots+k_m=n-1} \frac{u_{k_1}}{k_1!}\frac{u_{k_2}}{k_2!}\cdots\frac{u_{k_m}}{k_m!}\,, \qquad \text{if } n > 1. \tag{7.83}$$

The inner sum is the coefficient of z^{n-1} in the egf $\widehat{U}(z)$, raised to the mth power; and we obtain the correct formula also when $n = 1$, if we add in the term $\widehat{U}(z)^0$ that corresponds to the case $m = 0$. So

$$\frac{u_n}{n!} \;=\; [z^{n-1}] \sum_{m\geqslant 0} \frac{1}{m!}\,\widehat{U}(z)^m \;=\; [z^{n-1}]\,e^{\widehat{U}(z)} \;=\; [z^n]\,z e^{\widehat{U}(z)}$$

for all $n > 0$, and we have the equation

$$\widehat{U}(z) \;=\; z\,e^{\widehat{U}(z)}\,. \tag{7.84}$$

Progress! Equation (7.84) is almost like

$$\mathcal{E}(z) \;=\; e^{z\,\mathcal{E}(z)}\,,$$

which defines the generalized exponential series $\mathcal{E}(z) = \mathcal{E}_1(z)$ in (5.59) and (7.71); indeed, we have

$$\widehat{U}(z) \;=\; z\,\mathcal{E}(z)\,.$$

So we can read off the answer to our problem:

$$t_n \;=\; \frac{u_n}{n} \;=\; \frac{n!}{n}\,[z^n]\,\widehat{U}(z) \;=\; (n-1)!\,[z^{n-1}]\,\mathcal{E}(z) \;=\; n^{n-2}\,. \tag{7.85}$$

The complete graph on $\{1, 2, \ldots, n\}$ has exactly n^{n-2} spanning trees, for all $n > 0$.

7.7 DIRICHLET GENERATING FUNCTIONS

There are many other possible ways to generate a sequence from a series; any system of "kernel" functions $K_n(z)$ such that

$$\sum_n g_n K_n(z) = 0 \implies g_n = 0 \text{ for all } n$$

can be used, at least in principle. Ordinary generating functions use $K_n(z) = z^n$, and exponential generating functions use $K_n(z) = z^n/n!$; we could also try falling factorial powers $z^{\underline{n}}$, or binomial coefficients $z^{\underline{n}}/n! = \binom{z}{n}$.

The most important alternative to gf's and egf's uses the kernel functions $1/n^z$; it is intended for sequences $\langle g_1, g_2, \ldots\rangle$ that begin with $n = 1$ instead of $n = 0$:

$$\widetilde{G}(z) = \sum_{n\geqslant 1} \frac{g_n}{n^z}. \tag{7.86}$$

This is called a *Dirichlet generating function* (dgf), because the German mathematician Gustav Lejeune Dirichlet (1805–1859) made much of it.

For example, the dgf of the constant sequence $\langle 1, 1, 1, \ldots\rangle$ is

$$\sum_{n\geqslant 1} \frac{1}{n^z} = \zeta(z). \tag{7.87}$$

This is Riemann's *zeta function*, which we have also called the generalized harmonic number $H_\infty^{(z)}$ when $z > 1$.

The product of Dirichlet generating functions corresponds to a special kind of convolution:

$$\widetilde{F}(z)\widetilde{G}(z) = \sum_{l,m\geqslant 1} \frac{f_l}{l^z}\frac{g_m}{m^z} = \sum_{n\geqslant 1} \frac{1}{n^z} \sum_{l,m\geqslant 1} f_l\, g_m\, [l\cdot m = n].$$

Thus $\widetilde{F}(z)\widetilde{G}(z) = \widetilde{H}(z)$ is the dgf of the sequence

$$h_n = \sum_{d\backslash n} f_d\, g_{n/d}. \tag{7.88}$$

For example, we know from (4.55) that $\sum_{d\backslash n} \mu(d) = [n=1]$; this is the Dirichlet convolution of the Möbius sequence $\langle \mu(1), \mu(2), \mu(3), \ldots\rangle$ with $\langle 1, 1, 1, \ldots\rangle$, hence

$$\widetilde{M}(z)\zeta(z) = \sum_{n\geqslant 1} \frac{[n=1]}{n^z} = 1. \tag{7.89}$$

In other words, the dgf of $\langle \mu(1), \mu(2), \mu(3), \ldots\rangle$ is $\zeta(z)^{-1}$.

Dirichlet generating functions are particularly valuable when the sequence $\langle g_1, g_2, \ldots \rangle$ is a *multiplicative function*, namely when

$$g_{mn} = g_m g_n \qquad \text{for } m \perp n.$$

In such cases the values of g_n for all n are determined by the values of g_n when n is a power of a prime, and we can factor the dgf into a product over primes:

$$\widetilde{G}(z) = \prod_{p \text{ prime}} \left(1 + \frac{g_p}{p^z} + \frac{g_{p^2}}{p^{2z}} + \frac{g_{p^3}}{p^{3z}} + \cdots\right). \tag{7.90}$$

If, for instance, we set $g_n = 1$ for all n, we obtain a product representation of Riemann's zeta function:

$$\zeta(z) = \prod_{p \text{ prime}} \left(\frac{1}{1 - p^{-z}}\right). \tag{7.91}$$

The Möbius function has $\mu(p) = -1$ and $\mu(p^k) = 0$ for $k > 1$, hence its dgf is

$$\widetilde{M}(z) = \prod_{p \text{ prime}} (1 - p^{-z}); \tag{7.92}$$

this agrees, of course, with (7.89) and (7.91). Euler's φ function has $\varphi(p^k) = p^k - p^{k-1}$, hence its dgf has the factored form

$$\widetilde{\Phi}(z) = \prod_{p \text{ prime}} \left(1 + \frac{p-1}{p^z - p}\right) = \prod_{p \text{ prime}} \frac{1 - p^{-z}}{1 - p^{1-z}}. \tag{7.93}$$

We conclude that $\widetilde{\Phi}(z) = \zeta(z-1)/\zeta(z)$.

Exercises

Warmups

1 An eccentric collector of $2 \times n$ domino tilings pays \$4 for each vertical domino and \$1 for each horizontal domino. How many tilings are worth exactly \$$m$ by this criterion? For example, when $m = 6$ there are three solutions: , , and .

2 Give the generating function and the exponential generating function for the sequence $\langle 2, 5, 13, 35, \ldots \rangle = \langle 2^n + 3^n \rangle$ in closed form.

3 What is $\sum_{n \geqslant 0} H_n/10^n$?

4 The general expansion theorem for rational functions $P(z)/Q(z)$ is not completely general, because it restricts the degree of P to be less than the degree of Q. What happens if P has a larger degree than this?

5 Find a generating function $S(z)$ such that

$$[z^n]\,S(z) \;=\; \sum_k \binom{r}{k}\binom{r}{n-2k}.$$

Basics

6 Show that the recurrence (7.32) can be solved by the repertoire method, without using generating functions.

7 Solve the recurrence

$$\begin{aligned} g_0 &= 1; \\ g_n &= g_{n-1} + 2g_{n-2} + \cdots + ng_0\,, \qquad \text{for } n > 0. \end{aligned}$$

8 What is $[z^n]\,\big(\ln(1-z)\big)^2/(1-z)^{m+1}$?

9 Use the result of the previous exercise to evaluate $\sum_{k=0}^{n} H_k H_{n-k}$.

I deduce that Clark Kent is really Superman.

10 Set $r = s = -1/2$ in identity (7.62) and then remove all occurrences of $1/2$ by using tricks like (5.36). What amazing identity do you deduce?

11 This problem, whose three parts are independent, gives practice in the manipulation of generating functions. We assume that $A(z) = \sum_n a_n z^n$, $B(z) = \sum_n b_n z^n$, $C(z) = \sum_n c_n z^n$, and that the coefficients are zero for negative n.

a If $c_n = \sum_{j+2k\leqslant n} a_j b_k$, express C in terms of A and B.

b If $nb_n = \sum_{k=0}^{n} 2^k a_k/(n-k)!$, express A in terms of B.

c If r is a real number and if $a_n = \sum_{k=0}^{n} \binom{r+k}{k} b_{n-k}$, express A in terms of B; then use your formula to find coefficients $f_k(r)$ such that $b_n = \sum_{k=0}^{n} f_k(r)\,a_{n-k}$.

12 How many ways are there to put the numbers $\{1, 2, \ldots, 2n\}$ into a $2 \times n$ array so that rows and columns are in increasing order from left to right and from top to bottom? For example, one solution when $n = 5$ is

$$\begin{pmatrix} 1 & 2 & 4 & 5 & 8 \\ 3 & 6 & 7 & 9 & 10 \end{pmatrix}.$$

13 Prove Raney's generalized lemma, which is stated just before (7.70).

14 Solve the recurrence

$$\begin{aligned} g_0 &= 0\,, \qquad g_1 = 1\,, \\ g_n &= -2ng_{n-1} + \sum_k \binom{n}{k} g_k g_{n-k}\,, \qquad \text{for } n > 1, \end{aligned}$$

by using an exponential generating function.

15 The *Bell number* ϖ_n is the number of ways to partition n things into subsets. For example, $\varpi_3 = 5$ because we can partition $\{1,2,3\}$ in the following ways:

$$\{1,2,3\}; \quad \{1,2\}\cup\{3\}; \quad \{1,3\}\cup\{2\}; \quad \{1\}\cup\{2,3\}; \quad \{1\}\cup\{2\}\cup\{3\}.$$

Prove that $\varpi_{n+1} = \sum_k \binom{n}{k}\varpi_{n-k}$, and use this recurrence to find a closed form for the exponential generating function $P(z) = \sum_n \varpi_n z^n/n!$.

16 Two sequences $\langle a_n\rangle$ and $\langle b_n\rangle$ are related by the convolution formula

$$b_n = \sum_{k_1+2k_2+\cdots nk_n=n} \binom{a_1+k_1-1}{k_1}\binom{a_2+k_2-1}{k_2}\cdots\binom{a_n+k_n-1}{k_n};$$

also $a_0 = 0$ and $b_0 = 1$. Prove that the corresponding generating functions satisfy $\ln B(z) = A(z) + \frac{1}{2}A(z^2) + \frac{1}{3}A(z^3) + \cdots$.

17 Show that the exponential generating function $\widehat{G}(z)$ of a sequence is related to the ordinary generating function $G(z)$ by the formula

$$\int_0^\infty \widehat{G}(zt)e^{-t}\,dt = G(z),$$

if the integral exists.

18 Find the Dirichlet generating functions for the sequences

a $g_n = \sqrt{n}$;
b $g_n = \ln n$;
c $g_n = [n \text{ is squarefree}]$.

Express your answers in terms of the zeta function. (Squarefreeness is defined in exercise 4.13.)

19 Every power series $F(z) = \sum_{n\geqslant 0} f_n z^n$ with $f_0 = 1$ defines a sequence of polynomials $f_n(x)$ by the rule

$$F(z)^x = \sum_{n\geqslant 0} f_n(x)z^n,$$

where $f_n(1) = f_n$ and $f_n(0) = [n=0]$. In general, $f_n(x)$ has degree n. Show that such polynomials always satisfy the convolution formulas

What do you mean, "in general"? If $f_1 = f_2 = \cdots = f_{m-1} = 0$, the degree of $f_n(x)$ is at most $\lfloor n/m\rfloor$.

$$\sum_{k=0}^{n} f_k(x)f_{n-k}(y) = f_n(x+y);$$

$$(x+y)\sum_{k=0}^{n} kf_k(x)f_{n-k}(y) = xnf_n(x+y).$$

(The identities in Tables 202 and 272 are special cases of this trick.)

20 A power series $G(z)$ is called *differentiably finite* if there exist finitely many polynomials $P_0(z), \ldots, P_m(z)$, not all zero, such that

$$P_0(z)G(z) + P_1(z)G'(z) + \cdots + P_m(z)G^{(m)}(z) \;=\; 0\,.$$

A sequence of numbers $\langle g_0, g_1, g_2, \ldots\rangle$ is called *polynomially recursive* if there exist finitely many polynomials $p_0(z), \ldots, p_m(z)$, not all zero, such that

$$p_0(n)g_n + p_1(n)g_{n+1} + \cdots + p_m(n)g_{n+m} \;=\; 0$$

for all integers $n \geqslant 0$. Prove that a generating function is differentiably finite if and only if its sequence of coefficients is polynomially recursive.

Homework exercises

Will he settle for $2 \times n$ domino tilings?

21 A robber holds up a bank and demands \$500 in tens and twenties. He also demands to know the number of ways in which the cashier can give him the money. Find a generating function $G(z)$ for which this number is $[z^{500}]\,G(z)$, and a more compact generating function $\check{G}(z)$ for which this number is $[z^{50}]\,\check{G}(z)$. Determine the required number of ways by (a) using partial fractions; (b) using a method like (7.39).

22 Let P be the sum of all ways to "triangulate" polygons:

(The first term represents a degenerate polygon with only two vertices; every other term shows a polygon that has been divided into triangles. For example, a pentagon can be triangulated in five ways.) Define a "multiplication" operation $A \triangle B$ on triangulated polygons A and B so that the equation

$$P \;=\; __ \;+\; P \triangle P$$

is valid. Then replace each triangle by 'z'; what does this tell you about the number of ways to decompose an n-gon into triangles?

At union rates, as many as you can afford, plus a few.

23 In how many ways can a $2 \times 2 \times n$ pillar be built out of $2 \times 1 \times 1$ bricks?

24 How many spanning trees are in an n-wheel (a graph with n "outer" vertices in a cycle, each connected to an $(n+1)$st "hub" vertex), when $n \geqslant 3$?

25 Let $m \geqslant 2$ be an integer. What is a closed form for the generating function of the sequence $\langle n \bmod m \rangle$, as a function of z and m? Use this generating function to express '$n \bmod m$' in terms of the complex number $\omega = e^{2\pi i/m}$. (For example, when $m = 2$ we have $\omega = -1$ and $n \bmod 2 = \frac{1}{2} - \frac{1}{2}(-1)^n$.)

26 The second-order Fibonacci numbers $\langle \mathfrak{F}_n \rangle$ are defined by the recurrence

$$\begin{aligned} \mathfrak{F}_0 &= 0; \qquad \mathfrak{F}_1 = 1; \\ \mathfrak{F}_n &= \mathfrak{F}_{n-1} + \mathfrak{F}_{n-2} + F_n, \qquad \text{for } n > 1. \end{aligned}$$

Express $\mathfrak{F}_n$ in terms of the usual Fibonacci numbers F_n and F_{n+1}.

27 A $2 \times n$ domino tiling can also be regarded as a way to draw n disjoint lines in a $2 \times n$ array of points:

If we superimpose two such patterns, we get a set of cycles, since every point is touched by two lines. For example, if the lines above are combined with the lines

,

the result is

.

The same set of cycles is also obtained by combining

with
.

But we get a unique way to reconstruct the original patterns from the superimposed ones if we assign orientations to the vertical lines by using arrows that go alternately up/down/up/down/$\cdots$ in the first pattern and alternately down/up/down/up/$\cdots$ in the second. For example,

.

The number of such oriented cycle patterns must therefore be $T_n^2 = F_{n+1}^2$, and we should be able to prove this via algebra. Let Q_n be the number of oriented $2 \times n$ cycle patterns. Find a recurrence for Q_n, solve it with generating functions, and deduce algebraically that $Q_n = F_{n+1}^2$.

28 The coefficients of $A(z)$ in (7.39) satisfy $A_r + A_{r+10} + A_{r+20} + A_{r+30} = 100$ for $0 \leqslant r < 10$. Find a "simple" explanation for this.

29 What is the sum of Fibonacci products

$$\sum_{m>0} \sum_{\substack{k_1+k_2+\cdots+k_m=n \\ k_1,k_2,\ldots,k_m>0}} F_{k_1} F_{k_2} \ldots F_{k_m} ?$$

30 If the generating function $G(z) = 1/(1-\alpha z)(1-\beta z)$ has the partial fraction decomposition $a/(1-\alpha z)+b/(1-\beta z)$, what is the partial fraction decomposition of $G(z)^n$?

31 What function $g(n)$ of the positive integer n satisfies the recurrence

$$\sum_{d\backslash n} g(d)\,\varphi(n/d) \;=\; 1\,,$$

where φ is Euler's totient function?

32 An *arithmetic progression* is an infinite set of integers

$$\{an+b\} \;=\; \{b, a+b, 2a+b, 3a+b, \ldots\}\,.$$

A set of arithmetic progressions $\{a_1 n+b_1\}, \ldots, \{a_m n+b_m\}$ is called an *exact cover* if every nonnegative integer occurs in one and only one of the progressions. For example, the three progressions $\{2n\}$, $\{4n+1\}$, $\{4n+3\}$ constitute an exact cover. Show that if $\{a_1 n+b_1\}, \ldots, \{a_m n+b_m\}$ is an exact cover such that $2 \leqslant a_1 \leqslant \cdots \leqslant a_m$, then $a_{m-1} = a_m$. *Hint:* Use generating functions.

Exam problems

33 What is $[w^m z^n]\,\big(\ln(1+z)\big)/(1-wz)$?

34 Find a closed form for the generating function $\sum_{n\geqslant 0} G_n(z) w^n$, if

$$G_n(z) \;=\; \sum_{k\leqslant n/m} \binom{n-mk}{k} z^{mk}\,.$$

(Here m is a fixed positive integer.)

35 Evaluate the sum $\sum_{0<k<n} 1/k(n-k)$ in two ways:

a Expand the summand in partial fractions.

b Treat the sum as a convolution and use generating functions.

36 Let $A(z)$ be the generating function for $\langle a_0, a_1, a_2, a_3, \ldots\rangle$. Express $\sum_n a_{\lfloor n/m\rfloor} z^n$ in terms of A, z, and m.

37 Let a_n be the number of ways to write the positive integer n as a sum of powers of 2, disregarding order. For example, $a_4 = 4$, since $4 = 2+2 = 2+1+1 = 1+1+1+1$. By convention we let $a_0 = 1$. Let $b_n = \sum_{k=0}^{n} a_k$ be the cumulative sum of the first a's.

a Make a table of the a's and b's up through $n = 10$. What amazing relation do you observe in your table? (Don't prove it yet.)

b Express the generating function $A(z)$ as an infinite product.

c Use the expression from part (b) to prove the result of part (a).

38 Find a closed form for the double generating function

$$M(w,z) \;=\; \sum_{m,n\geqslant 0} \min(m,n)\, w^m z^n\,.$$

Generalize your answer to obtain, for fixed $m \geqslant 2$, a closed form for

$$M(z_1,\ldots,z_m) \;=\; \sum_{n_1,\ldots,n_m\geqslant 0} \min(n_1,\ldots,n_m)\, z_1^{n_1}\ldots z_m^{n_m}\,.$$

39 Given positive integers m and n, find closed forms for

$$\sum_{1\leqslant k_1<k_2<\cdots<k_m\leqslant n} k_1k_2\ldots k_m \quad\text{and}\quad \sum_{1\leqslant k_1\leqslant k_2\leqslant\cdots\leqslant k_m\leqslant n} k_1k_2\ldots k_m\,.$$

(For example, when $m = 2$ and $n = 3$ the sums are $1\cdot 2 + 1\cdot 3 + 2\cdot 3$ and $1\cdot 1+1\cdot 2+1\cdot 3+2\cdot 2+2\cdot 3+3\cdot 3$.) *Hint:* What are the coefficients of z^m in the generating functions $(1+a_1z)\ldots(1+a_nz)$ and $1/(1-a_1z)\ldots(1-a_nz)$?

40 Express $\sum_k \binom{n}{k}(kF_{k-1} - F_k)(n-k)_{¡}$ in closed form.

41 An *up-down permutation* of order n is an arrangement $a_1a_2\ldots a_n$ of the integers $\{1,2,\ldots,n\}$ that goes alternately up and down:

$$a_1 \;<\; a_2 \;>\; a_3 \;<\; a_4 \;>\; \cdots\,.$$

For example, 35142 is an up-down permutation of order 5. If A_n denotes the number of up-down permutations of order n, show that the exponential generating function of $\langle A_n\rangle$ is $(1+\sin z)/\cos z$.

42 A space probe has discovered that organic material on Mars has DNA composed of five symbols, denoted by (a,b,c,d,e), instead of the four components in earthling DNA. The four pairs cd, ce, ed, and ee never occur consecutively in a string of Martian DNA, but any string without forbidden pairs is possible. (Thus $bbcda$ is forbidden but $bbdca$ is OK.) How many Martian DNA strings of length n are possible? (When $n = 2$ the answer is 21, because the left and right ends of a string are distinguishable.)

43 The *Newtonian generating function* of a sequence $\langle g_n \rangle$ is defined to be

$$\dot{G}(z) \;=\; \sum_n g_n \binom{z}{n}.$$

Find a convolution formula that defines the relation between sequences $\langle f_n \rangle$, $\langle g_n \rangle$, and $\langle h_n \rangle$ whose Newtonian generating functions are related by the equation $\dot{F}(z)\dot{G}(z) = \dot{H}(z)$. Try to make your formula as simple and symmetric as possible.

44 Let q_n be the number of possible outcomes when n numbers $\{x_1, \ldots, x_n\}$ are compared with each other. For example, $q_3 = 13$ because the possibilities are

$$\begin{array}{llll}
x_1 < x_2 < x_3\,; & x_1 < x_2 = x_3\,; & x_1 < x_3 < x_2\,; & x_1 = x_2 < x_3\,; \\
x_1 = x_2 = x_3\,; & x_1 = x_3 < x_2\,; & x_2 < x_1 < x_3\,; & \\
x_2 < x_1 = x_3\,; & x_2 < x_3 < x_1\,; & x_2 = x_3 < x_1\,; & \\
x_3 < x_1 < x_2\,; & x_3 < x_1 = x_2\,; & x_3 < x_2 < x_1\,. &
\end{array}$$

Find a closed form for the egf $\widehat{Q}(z) = \sum_n q_n z^n/n!$. Also find sequences $\langle a_n \rangle$, $\langle b_n \rangle$, $\langle c_n \rangle$ such that

$$q_n \;=\; \sum_{k \geqslant 0} k^n a_k \;=\; \sum_k \left\{ {n \atop k} \right\} b_k \;=\; \sum_k \left\langle {n \atop k} \right\rangle c_k\,, \quad \text{for all } n > 0.$$

45 Evaluate $\sum_{m,n>0} [m \perp n]/m^2 n^2$.

46 Evaluate

$$\sum_{0 \leqslant k \leqslant n/2} \binom{n-2k}{k} \left(\frac{-4}{27}\right)^k$$

in closed form. *Hint:* $z^3 - z^2 + \frac{4}{27} = (z + \frac{1}{3})(z - \frac{2}{3})^2$.

47 Show that the numbers U_n and V_n of $3 \times n$ domino tilings, as given in (7.34), are closely related to the fractions in the Stern–Brocot tree that converge to $\sqrt{3}$.

48 A certain sequence $\langle g_n \rangle$ satisfies the recurrence

$$a g_n + b g_{n+1} + c g_{n+2} + d \;=\; 0\,, \qquad \text{integer } n \geqslant 0,$$

for some integers (a, b, c, d) with $\gcd(a, b, c, d) = 1$. It also has the closed form

$$g_n \;=\; \lfloor \alpha(1 + \sqrt{2}\,)^n \rfloor\,, \qquad \text{integer } n \geqslant 0,$$

for some real number α between 0 and 1. Find a, b, c, d, and α.

49 This is a problem about powers and parity. *Kissinger, take note.*

a Consider the sequence $\langle a_0, a_1, a_2, \ldots\rangle = \langle 2, 2, 6, \ldots\rangle$ defined by the formula

$$a_n \;=\; (1+\sqrt{2})^n + (1-\sqrt{2})^n\,.$$

Find a simple recurrence relation that is satisfied by this sequence.

b Prove that $\lceil (1+\sqrt{2})^n \rceil \equiv n \pmod 2$ for all integers $n > 0$.

c Find a number α of the form $(p+\sqrt{q})/2$, where p and q are positive integers, such that $\lfloor \alpha^n \rfloor \equiv n \pmod 2$ for all integers $n > 0$.

Bonus problems

50 Continuing exercise 22, consider the sum of all ways to decompose polygons into polygons:

Find a symbolic equation for Q and use it to find a generating function for the number of ways to draw nonintersecting diagonals inside a convex n-gon. (Give a closed form for the generating function as a function of z; you need not find a closed form for the coefficients.)

51 Prove that the product

$$2^{mn/2} \prod_{\substack{1\leqslant j\leqslant m\\ 1\leqslant k\leqslant n}} \left(\left(\cos^2 \frac{j\pi}{m+1}\right)\square^2 + \left(\cos^2 \frac{k\pi}{n+1}\right)\square^2 \right)^{1/4}$$

is the generating function for tilings of an $m\times n$ rectangle with dominoes. (There are mn factors, which we can imagine are written in the mn cells of the rectangle. If mn is odd, the middle factor is zero. The coefficient of $\square^j\square^k$ is the number of ways to do the tiling with j vertical and k horizontal dominoes.) *Hint:* This is a difficult problem, really beyond the scope of this book. You may wish to simply verify the formula in the case $m = 3$, $n = 4$. *Is this a hint or a warning?*

52 Prove that the polynomials defined by the recurrence

$$p_n(y) \;=\; \left(y-\frac{1}{4}\right)^n - \sum_{k=0}^{n-1} \binom{2n}{2k} \left(\frac{-1}{4}\right)^{n-k} p_k(y)\,, \quad \text{integer } n \geqslant 0,$$

have the form $p_n(y) = \sum_{m=0}^{n} \left|{n \atop m}\right| y^n$, where $\left|{n \atop m}\right|$ is a positive integer for $1 \leqslant m \leqslant n$. *Hint:* This exercise is very instructive but not very easy.

53 The sequence of *pentagonal numbers* $\langle 1, 5, 12, 22, \ldots \rangle$ generalizes the triangular and square numbers in an obvious way:

Let the nth triangular number be $T_n = n(n+1)/2$; let the nth pentagonal number be $P_n = n(3n-1)/2$; and let U_n be the $3 \times n$ domino-tiling number defined in (7.38). Prove that the triangular number $T_{(U_{4n+2}-1)/2}$ is also a pentagonal number. *Hint:* $3U_{2n}^2 = (V_{2n-1} + V_{2n+1})^2 + 2$.

54 Consider the following curious construction:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
1	2	3	4		6	7	8	9		11	12	13	14		16	...
1	3	6	10		16	23	31	40		51	63	76	90		106	...
1	3	6			16	23	31			51	63	76			106	...
1	4	10			26	49	80			131	194	270			376	...
1	4				26	49				131	194				376	...
1	5				31	80				211	405				781	...
1					31					211					781	...
1					32					243					1024	...

(Start with a row containing all the positive integers. Then delete every mth column; here $m = 5$. Then replace the remaining entries by partial sums. Then delete every $(m-1)$st column. Then replace with partial sums again, and so on.) Use generating functions to show that the final result is the sequence of mth powers. For example, when $m = 5$ we get $\langle 1^5, 2^5, 3^5, 4^5, \ldots \rangle$ as shown.

55 Prove that if the power series $F(z)$ and $G(z)$ are differentiably finite (as defined in exercise 20), then so are $F(z) + G(z)$ and $F(z)G(z)$.

Research problems

56 Prove that there is no "simple closed form" for the coefficient of z^n in $(1 + z + z^2)^n$, as a function of n, in some large class of "simple closed forms."

57 Prove or disprove: If all the coefficients of $G(z)$ are either 0 or 1, and if all the coefficients of $G(z)^2$ are less than some constant M, then infinitely many of the coefficients of $G(z)^2$ are zero.

8

Discrete Probability

THE ELEMENT OF CHANCE enters into many of our attempts to understand the world we live in. A mathematical *theory of probability* allows us to calculate the likelihood of complex events if we assume that the events are governed by appropriate axioms. This theory has significant applications in all branches of science, and it has strong connections with the techniques we have studied in previous chapters.

Probabilities are called "discrete" if we can compute the probabilities of all events by summation instead of by integration. We are getting pretty good at sums, so it should come as no great surprise that we are ready to apply our knowledge to some interesting calculations of probabilities and averages.

8.1 DEFINITIONS

(Readers unfamiliar with probability theory will, with high probability, benefit from a perusal of Feller's classic introduction to the subject [120].)

Probability theory starts with the idea of a *probability space*, which is a set Ω of all things that can happen in a given problem together with a rule that assigns a probability $\Pr(\omega)$ to each elementary event $\omega \in \Omega$. The probability $\Pr(\omega)$ must be a nonnegative real number, and the condition

$$\sum_{\omega \in \Omega} \Pr(\omega) \;=\; 1 \tag{8.1}$$

must hold in every discrete probability space. Thus, each value $\Pr(\omega)$ must lie in the interval $[0\,..\,1]$. We speak of Pr as a *probability distribution*, because it distributes a total probability of 1 among the events ω.

Here's an example: If we're rolling a pair of dice, the set Ω of elementary events is $D^2 = \{$ ⚀⚀, ⚀⚁, ..., ⚅⚅ $\}$, where

$$D \;=\; \{ ⚀, ⚁, ⚂, ⚃, ⚄, ⚅ \}$$

Never say die.

is the set of all six ways that a given die can land. Two rolls such as ⚀⚁ and ⚁⚀ are considered to be distinct; hence this probability space has a total of $6^2 = 36$ elements.

We usually assume that dice are "fair" — that each of the six possibilities for a particular die has probability $\frac{1}{6}$, and that each of the 36 possible rolls in Ω has probability $\frac{1}{36}$. But we can also consider "loaded" dice in which there is a different distribution of probabilities. For example, let

Careful: They might go off.

$$\begin{aligned}
\Pr_1(\boxed{\cdot}) &= \Pr_1(\boxed{\vdots}) = \tfrac{1}{4}\,; \\
\Pr_1(\boxed{\cdot\,\cdot}) &= \Pr_1(\boxed{\cdot\,\cdot\,\cdot}) = \Pr_1(\boxed{\vdots}) = \Pr_1(\boxed{\vdots\,\cdot}) = \tfrac{1}{8}\,.
\end{aligned}$$

Then $\sum_{d\in D} \Pr_1(d) = 1$, so $\Pr_1$ is a probability distribution on the set D, and we can assign probabilities to the elements of $\Omega = D^2$ by the rule

$$\Pr_{11}(d\,d') = \Pr_1(d)\,\Pr_1(d')\,. \tag{8.2}$$

For example, $\Pr_{11}(\boxed{\vdots}\boxed{\cdot\,\cdot\,\cdot}) = \frac{1}{4}\cdot\frac{1}{8} = \frac{1}{32}$. This is a valid distribution because

$$\begin{aligned}
\sum_{\omega\in\Omega} \Pr_{11}(\omega) &= \sum_{d\,d'\in D^2} \Pr_{11}(d\,d') = \sum_{d,d'\in D} \Pr_1(d)\,\Pr_1(d') \\
&= \sum_{d\in D} \Pr_1(d) \sum_{d'\in D} \Pr_1(d') = 1\cdot 1 = 1\,.
\end{aligned}$$

We can also consider the case of one fair die and one loaded die,

$$\Pr_{01}(d\,d') = \Pr_0(d)\,\Pr_1(d')\,, \qquad \text{where } \Pr_0(d) = \tfrac{1}{6}, \tag{8.3}$$

in which case $\Pr_{01}(\boxed{\vdots}\boxed{\cdot\,\cdot\,\cdot}) = \frac{1}{6}\cdot\frac{1}{8} = \frac{1}{48}$. Dice in the "real world" can't really be expected to turn up equally often on each side, because they aren't perfectly symmetrical; but $\frac{1}{6}$ is usually pretty close to the truth.

If all sides of a cube were identical, how could we tell which side is face up?

An *event* is a subset of Ω. In dice games, for example, the set

$$\{\boxed{\cdot}\boxed{\cdot},\ \boxed{\cdot\,\cdot}\boxed{\cdot\,\cdot},\ \boxed{\cdot\,\cdot\,\cdot}\boxed{\cdot\,\cdot\,\cdot},\ \boxed{\vdots}\boxed{\vdots},\ \boxed{\vdots\,\cdot}\boxed{\vdots\,\cdot},\ \boxed{\vdots}\boxed{\vdots}\}$$

is the event that "doubles are thrown." The individual elements ω of Ω are called *elementary events* because they cannot be decomposed into smaller subsets; we can think of ω as a one-element event $\{\omega\}$.

The probability of an event A is defined by the formula

$$\Pr(\omega\in A) = \sum_{\omega\in A} \Pr(\omega)\,; \tag{8.4}$$

and in general if $R(\omega)$ is any statement about ω, we write '$\Pr\big(R(\omega)\big)$' for the sum of all $\Pr(\omega)$ such that $R(\omega)$ is true. Thus, for example, the probability of doubles with fair dice is $\frac{1}{36}+\frac{1}{36}+\frac{1}{36}+\frac{1}{36}+\frac{1}{36}+\frac{1}{36} = \frac{1}{6}$; but when both dice are loaded with probability distribution $\Pr_1$ it is $\frac{1}{16}+\frac{1}{64}+\frac{1}{64}+\frac{1}{64}+\frac{1}{64}+\frac{1}{16} = \frac{3}{16} > \frac{1}{6}$. Loading the dice makes the event "doubles are thrown" more probable.

(We have been using $\sum$-notation in a more general sense here than defined in Chapter 2: The sums in (8.1) and (8.4) occur over all elements ω of an arbitrary set, not over integers only. However, this new development is not really alarming; we can agree to use special notation under a $\sum$ whenever nonintegers are intended, so there will be no confusion with our ordinary conventions. The other definitions in Chapter 2 are still valid; in particular, the definition of infinite sums in that chapter gives the appropriate interpretation to our sums when the set Ω is infinite. Each probability is nonnegative, and the sum of all probabilities is bounded, so the probability of event A in (8.4) is well defined for all subsets $A \subseteq \Omega$.)

A *random variable* is a function defined on the elementary events ω of a probability space. For example, if $\Omega = D^2$ we can define $S(\omega)$ to be the sum of the spots on the dice roll ω, so that $S(\boxed{\vdots}\boxed{\iddots}) = 6+3 = 9$. The probability that the spots total seven is the probability of the event $S(\omega) = 7$, namely

$$\begin{aligned}&\Pr(\boxed{\cdot}\boxed{\vdots}) + \Pr(\boxed{\cdot}\boxed{\because}) + \Pr(\boxed{\iddots}\boxed{\because})\\ &\qquad + \Pr(\boxed{\because}\boxed{\iddots}) + \Pr(\boxed{\because}\boxed{\cdot}) + \Pr(\boxed{\vdots}\boxed{\cdot})\,.\end{aligned}$$

With fair dice ($\Pr = \Pr_{00}$), this happens with probability $\frac{1}{6}$; but with loaded dice ($\Pr = \Pr_{11}$), it happens with probability $\frac{1}{16}+\frac{1}{64}+\frac{1}{64}+\frac{1}{64}+\frac{1}{64}+\frac{1}{16} = \frac{3}{16}$, the same as we observed for doubles.

It's customary to drop the '(ω)' when we talk about random variables, because there's usually only one probability space involved when we're working on any particular problem. Thus we say simply '$S = 7$' for the event that a 7 was rolled, and '$S = 4$' for the event $\{\boxed{\cdot}\boxed{\iddots}, \boxed{\cdot}\boxed{\cdot}, \boxed{\iddots}\boxed{\cdot}\}$.

A random variable can be characterized by the probability distribution of its values. Thus, for example, S takes on eleven possible values $\{2, 3, \ldots, 12\}$, and we can tabulate the probability that $S = s$ for each s in this set:

s	2	3	4	5	6	7	8	9	10	11	12
$\Pr_{00}(S=s)$	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$
$\Pr_{11}(S=s)$	$\frac{4}{64}$	$\frac{4}{64}$	$\frac{5}{64}$	$\frac{6}{64}$	$\frac{7}{64}$	$\frac{12}{64}$	$\frac{7}{64}$	$\frac{6}{64}$	$\frac{5}{64}$	$\frac{4}{64}$	$\frac{4}{64}$

If we're working on a problem that involves only the random variable S and no other properties of dice, we can compute the answer from these probabilities alone, without regard to the details of the set $\Omega = D^2$. In fact, we could define the probability space to be the smaller set $\Omega = \{2, 3, \ldots, 12\}$, with whatever probability distribution $\Pr(s)$ is desired. Then '$S = 4$' would be an elementary event. Thus we can often ignore the underlying probability space Ω and work directly with random variables and their distributions.

If two random variables X and Y are defined over the same probability space Ω, we can characterize their behavior without knowing everything

about Ω if we know the "joint distribution" *Just Say No.*

$$\Pr(X = x \text{ and } Y = y)$$

for each x in the range of X and each y in the range of Y. We say that X and Y are *independent* random variables if

$$\Pr(X = x \text{ and } Y = y) \;=\; \Pr(X = x) \cdot \Pr(Y = y) \tag{8.5}$$

for all x and y. Intuitively, this means that the value of X has no effect on the value of Y.

For example, if Ω is the set of dice rolls D^2, we can let S_1 be the number of spots on the first die and S_2 the number of spots on the second. Then the random variables S_1 and S_2 are independent with respect to each of the probability distributions $\Pr_{00}$, $\Pr_{11}$, and $\Pr_{01}$ discussed earlier, because we defined the dice probability for each elementary event dd' as a product of a probability for $S_1 = d$ multiplied by a probability for $S_2 = d'$. We could have defined probabilities differently so that, say,

$$\Pr(\boxed{\cdot}\,\boxed{\begin{smallmatrix}\cdot&&\cdot\\&\cdot&\\\cdot&&\cdot\end{smallmatrix}}) \,/\, \Pr(\boxed{\cdot}\,\boxed{\begin{smallmatrix}\cdot&\cdot&\cdot\\\cdot&\cdot&\cdot\end{smallmatrix}}) \;\neq\; \Pr(\boxed{\begin{smallmatrix}&\cdot\\\cdot&\\&\end{smallmatrix}}\,\boxed{\begin{smallmatrix}\cdot&&\cdot\\&\cdot&\\\cdot&&\cdot\end{smallmatrix}}) \,/\, \Pr(\boxed{\begin{smallmatrix}&\cdot\\\cdot&\\&\end{smallmatrix}}\,\boxed{\begin{smallmatrix}\cdot&\cdot&\cdot\\\cdot&\cdot&\cdot\end{smallmatrix}})\,; \tag{}$$ *A dicey inequality.*

but we didn't do that, because different dice aren't supposed to influence each other. With our definitions, both of these ratios are $\Pr(S_2 = 5)/\Pr(S_2 = 6)$.

We have defined S to be the sum of the two spot values, $S_1 + S_2$. Let's consider another random variable P, the product $S_1 S_2$. Are S and P independent? Informally, no; if we are told that $S = 2$, we know that P must be 1. Formally, no again, because the independence condition (8.5) fails spectacularly (at least in the case of fair dice): For all legal values of s and p, we have $0 < \Pr_{00}(S = s) \cdot \Pr_{00}(P = p) \leqslant \frac{1}{6} \cdot \frac{1}{9}$; this can't equal $\Pr_{00}(S = s \text{ and } P = p)$, which is a multiple of $\frac{1}{36}$.

If we want to understand the typical behavior of a given random variable, we often ask about its "average" value. But the notion of "average" is ambiguous; people generally speak about three different kinds of averages when a sequence of numbers is given:

- the *mean* (which is the sum of all values, divided by the number of values);
- the *median* (which is the middle value, numerically);
- the *mode* (which is the value that occurs most often).

For example, the mean of $(3, 1, 4, 1, 5)$ is $\frac{3+1+4+1+5}{5} = 2.8$; the median is 3; the mode is 1.

But probability theorists usually work with random variables instead of with sequences of numbers, so we want to define the notion of an "average" for random variables too. Suppose we repeat an experiment over and over again,

making independent trials in such a way that each value of X occurs with a frequency approximately proportional to its probability. (For example, we might roll a pair of dice many times, observing the values of S and/or P.) We'd like to define the average value of a random variable so that such experiments will usually produce a sequence of numbers whose mean, median, or mode is approximately the same as the mean, median, or mode of X, according to our definitions.

Here's how it can be done: The *mean* of a random real-valued variable X on a probability space Ω is defined to be

$$\sum_{x \in X(\Omega)} x \cdot \Pr(X = x) \tag{8.6}$$

if this potentially infinite sum exists. (Here $X(\Omega)$ stands for the set of all values that X can assume.) The *median* of X is defined to be the set of all x such that

$$\Pr(X \leqslant x) \;\geqslant\; \frac{1}{2} \quad \text{and} \quad \Pr(X \geqslant x) \;\geqslant\; \frac{1}{2}\,. \tag{8.7}$$

And the *mode* of X is defined to be the set of all x such that

$$\Pr(X = x) \;\geqslant\; \Pr(X = x') \qquad \text{for all } x' \in X(\Omega). \tag{8.8}$$

In our dice-throwing example, the mean of S turns out to be $2 \cdot \frac{1}{36} + 3 \cdot \frac{2}{36} + \cdots + 12 \cdot \frac{1}{36} = 7$ in distribution $\Pr_{00}$, and it also turns out to be 7 in distribution $\Pr_{11}$. The median and mode both turn out to be $\{7\}$ as well, in both distributions. So S has the same average under all three definitions. On the other hand the P in distribution $\Pr_{00}$ turns out to have a mean value of $\frac{49}{4} = 12.25$; its median is $\{10\}$, and its mode is $\{6, 12\}$. The mean of P is unchanged if we load the dice with distribution $\Pr_{11}$, but the median drops to $\{8\}$ and the mode becomes $\{6\}$ alone.

Probability theorists have a special name and notation for the mean of a random variable: They call it the *expected value*, and write

$$\mathrm{E}X \;=\; \sum_{\omega \in \Omega} X(\omega) \Pr(\omega)\,. \tag{8.9}$$

In our dice-throwing example, this sum has 36 terms (one for each element of Ω), while (8.6) is a sum of only eleven terms. But both sums have the same value, because they're both equal to

$$\sum_{\substack{\omega \in \Omega \\ x \in X(\Omega)}} x \Pr(\omega) \big[x = X(\omega)\big]\,.$$

The mean of a random variable turns out to be more meaningful in applications than the other kinds of averages, so we shall largely forget about medians and modes from now on. We will use the terms "expected value," "mean," and "average" almost interchangeably in the rest of this chapter.

I get it: On average, "average" means "mean."

If X and Y are any two random variables defined on the same probability space, then $X+Y$ is also a random variable on that space. By formula (8.9), the average of their sum is the sum of their averages:

$$E(X+Y) \;=\; \sum_{\omega\in\Omega} \bigl(X(\omega)+Y(\omega)\bigr)\Pr(\omega) \;=\; EX+EY\,. \tag{8.10}$$

Similarly, if α is any constant we have the simple rule

$$E(\alpha X) \;=\; \alpha EX\,. \tag{8.11}$$

But the corresponding rule for multiplication of random variables is more complicated in general; the expected value is defined as a sum over elementary events, and sums of products don't often have a simple form. In spite of this difficulty, there is a very nice formula for the mean of a product in the special case that the random variables are independent:

$$E(XY) \;=\; (EX)(EY), \qquad \text{if } X \text{ and } Y \text{ are independent.} \tag{8.12}$$

We can prove this by the distributive law for products,

$$\begin{aligned}
E(XY) \;&=\; \sum_{\omega\in\Omega} X(\omega)Y(\omega)\cdot\Pr(\omega)\\
&=\; \sum_{\substack{x\in X(\Omega)\\ y\in Y(\Omega)}} xy\cdot\Pr(X=x \text{ and } Y=y)\\
&=\; \sum_{\substack{x\in X(\Omega)\\ y\in Y(\Omega)}} xy\cdot\Pr(X=x)\Pr(Y=y)\\
&=\; \sum_{x\in X(\Omega)} x\Pr(X=x) \;\cdot\; \sum_{y\in Y(\Omega)} y\Pr(Y=y) \;=\; (EX)(EY)\,.
\end{aligned}$$

For example, we know that $S=S_1+S_2$ and $P=S_1S_2$, when S_1 and S_2 are the numbers of spots on the first and second of a pair of random dice. We have $ES_1=ES_2=\frac{7}{2}$, hence $ES=7$; furthermore S_1 and S_2 are independent, so $EP=\frac{7}{2}\cdot\frac{7}{2}=\frac{49}{4}$, as claimed earlier. We also have $E(S+P)=ES+EP=7+\frac{49}{4}$. But S and P are not independent, so we cannot assert that $E(SP)=7\cdot\frac{49}{4}=\frac{343}{4}$. In fact, the expected value of SP turns out to equal $\frac{637}{6}$ in distribution $\Pr_{00}$, while it equals 112 (exactly) in distribution $\Pr_{11}$.

8.2 MEAN AND VARIANCE

The next most important property of a random variable, after we know its expected value, is its *variance*, defined as the mean square deviation from the mean:

$$VX = E\big((X - EX)^2\big)\,. \tag{8.13}$$

If we denote EX by μ, the variance VX is the expected value of $(X-\mu)^2$. This measures the "spread" of X's distribution.

As a simple example of variance computation, let's suppose we have just been made an offer we can't refuse: Someone has given us two gift certificates for a certain lottery. The lottery organizers sell 100 tickets for each weekly drawing. One of these tickets is selected by a uniformly random process — that is, each ticket is equally likely to be chosen — and the lucky ticket holder wins a hundred million dollars. The other 99 ticket holders win nothing.

(Slightly subtle point: There are two probability spaces, depending on what strategy we use; but EX_1 and EX_2 are the same in both.)

We can use our gift in two ways: Either we buy two tickets in the same lottery, or we buy one ticket in each of two lotteries. Which is a better strategy? Let's try to analyze this by letting X_1 and X_2 be random variables that represent the amount we win on our first and second ticket. The expected value of X_1, in millions, is

$$EX_1 = \tfrac{99}{100}\cdot 0 + \tfrac{1}{100}\cdot 100 = 1\,,$$

and the same holds for X_2. Expected values are additive, so our average total winnings will be

$$E(X_1 + X_2) = EX_1 + EX_2 = 2 \text{ million dollars},$$

regardless of which strategy we adopt.

Still, the two strategies seem different. Let's look beyond expected values and study the exact probability distribution of $X_1 + X_2$:

	winnings (millions)		
	0	100	200
same drawing	.9800	.0200	
different drawings	.9801	.0198	.0001

If we buy two tickets in the same lottery we have a 98% chance of winning nothing and a 2% chance of winning \$100 million. If we buy them in different lotteries we have a 98.01% chance of winning nothing, so this is slightly more likely than before; and we have a 0.01% chance of winning \$200 million, also slightly more likely than before; and our chances of winning \$100 million are now 1.98%. So the distribution of $X_1 + X_2$ in this second situation is slightly

more spread out; the middle value, \$100 million, is slightly less likely, but the extreme values are slightly more likely.

It's this notion of the spread of a random variable that the variance is intended to capture. We measure the spread in terms of the squared deviation of the random variable from its mean. In case 1, the variance is therefore

$$.98(0M-2M)^2 + .02(100M-2M)^2 = 196M^2;$$

in case 2 it is

$$\begin{aligned} .9801(0M-2M)^2 + .0198(100M-2M)^2 + .0001(200M-2M)^2 \\ = 198M^2. \end{aligned}$$

As we expected, the latter variance is slightly larger, because the distribution of case 2 is slightly more spread out.

When we work with variances, everything is squared, so the numbers can get pretty big. (The factor M^2 is one trillion, which is somewhat imposing even for high-stakes gamblers.) To convert the numbers back to the more meaningful original scale, we often take the square root of the variance. The resulting number is called the *standard deviation*, and it is usually denoted by the Greek letter σ:

Interesting: The variance of a dollar amount is expressed in units of square dollars.

$$\sigma = \sqrt{VX}. \tag{8.14}$$

The standard deviations of the random variables $X_1 + X_2$ in our two lottery strategies are $\sqrt{196M^2} = 14.00M$ and $\sqrt{198M^2} \approx 14.071247M$. In some sense the second alternative is about \$71,247 riskier.

How does the variance help us choose a strategy? It's not clear. The strategy with higher variance is a little riskier; but do we get the most for our money by taking more risks or by playing it safe? Suppose we had the chance to buy 100 tickets instead of only two. Then we could have a guaranteed victory in a single lottery (and the variance would be zero); or we could gamble on a hundred different lotteries, with a $.99^{100} \approx .366$ chance of winning nothing but also with a nonzero probability of winning up to \$10,000,000,000. To decide between these alternatives is beyond the scope of this book; all we can do here is explain how to do the calculations.

Another way to reduce risk might be to bribe the lottery officials. I guess that's where probability becomes indiscreet.

In fact, there is a simpler way to calculate the variance, instead of using the definition (8.13). (We suspect that there must be something going on in the mathematics behind the scenes, because the variances in the lottery example magically came out to be integer multiples of M^2.) We have

(N.B.: Opinions expressed in these margins do not necessarily represent the opinions of the management.)

$$\begin{aligned} E\big((X-EX)^2\big) &= E\big(X^2 - 2X(EX) + (EX)^2\big) \\ &= E(X^2) - 2(EX)(EX) + (EX)^2, \end{aligned}$$

since (EX) is a constant; hence

$$VX = E(X^2) - (EX)^2 . \tag{8.15}$$

"The variance is the mean of the square minus the square of the mean."

For example, the mean of $(X_1 + X_2)^2$ comes to $.98(0M)^2 + .02(100M)^2 = 200M^2$ or to $.9801(0M)^2 + .0198(100M)^2 + .0001(200M)^2 = 202M^2$ in the lottery problem. Subtracting $4M^2$ (the square of the mean) gives the results we obtained the hard way.

There's an even easier formula yet, if we want to calculate $V(X+Y)$ when X and Y are independent: We have

$$\begin{aligned} E\big((X+Y)^2\big) &= E(X^2 + 2XY + Y^2) \\ &= E(X^2) + 2(EX)(EY) + E(Y^2) , \end{aligned}$$

since we know that $E(XY) = (EX)(EY)$ in the independent case. Therefore

$$\begin{aligned} V(X+Y) &= E\big((X+Y)^2\big) - (EX + EY)^2 \\ &= E(X^2) + 2(EX)(EY) + E(Y^2) \\ &\quad - (EX)^2 - 2(EX)(EY) - (EY)^2 \\ &= E(X^2) - (EX)^2 + E(Y^2) - (EY)^2 \\ &= VX + VY . \end{aligned} \tag{8.16}$$

"The variance of a sum of independent random variables is the sum of their variances." For example, the variance of the amount we can win with a single lottery ticket is

$$E(X_1^2) - (EX_1)^2 = .99(0M)^2 + .01(100M)^2 - (1M)^2 = 99M^2 .$$

Therefore the variance of the total winnings of two lottery tickets in two separate (independent) lotteries is $2 \times 99M^2 = 198M^2$. And the corresponding variance for n independent lottery tickets is $n \times 99M^2$.

The variance of the dice-roll sum S drops out of this same formula, since $S = S_1 + S_2$ is the sum of two independent random variables. We have

$$VS_1 = \frac{1}{6}(1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2) - \left(\frac{7}{2}\right)^2 = \frac{35}{12}$$

when the dice are fair; hence $VS = \frac{35}{12} + \frac{35}{12} = \frac{35}{6}$. The loaded die has

$$VS_1 = \frac{1}{8}(2 \cdot 1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 2 \cdot 6^2) - \left(\frac{7}{2}\right)^2 = \frac{45}{12} ;$$

hence $VS = \frac{45}{6} = 7.5$ when both dice are loaded. Notice that the loaded dice give S a larger variance, although S actually assumes its average value 7 more often than it would with fair dice. If our goal is to shoot lots of lucky 7's, the variance is not our best indicator of success.

OK, we have learned how to compute variances. But we haven't really seen a good reason why the variance is a natural thing to compute. Everybody does it, but why? The main reason is *Chebyshev's inequality* ([29] and [57]), which states that the variance has a significant property:

If he proved it in 1867, it's a classic '67 Chebyshev.

$$\Pr\bigl((X-EX)^2 \geqslant \alpha\bigr) \;\leqslant\; VX/\alpha\,, \qquad \text{for all } \alpha > 0. \tag{8.17}$$

(This is different from the monotonic inequalities of Chebyshev that we encountered in Chapter 2.) Very roughly, (8.17) tells us that a random variable X will rarely be far from its mean EX if its variance VX is small. The proof is amazingly simple. We have

$$\begin{aligned} VX &= \sum_{\omega\in\Omega} \bigl(X(\omega)-EX\bigr)^2 \Pr(\omega) \\ &\geqslant \sum_{\substack{\omega\in\Omega \\ (X(\omega)-EX)^2\geqslant\alpha}} \bigl(X(\omega)-EX\bigr)^2 \Pr(\omega) \\ &\geqslant \sum_{\substack{\omega\in\Omega \\ (X(\omega)-EX)^2\geqslant\alpha}} \alpha \Pr(\omega) \quad = \quad \alpha\cdot\Pr\bigl((X-EX)^2\geqslant\alpha\bigr)\,; \end{aligned}$$

dividing by α finishes the proof.

If we write μ for the mean and σ for the standard deviation, and if we replace α by c^2VX in (8.17), the condition $(X-EX)^2 \geqslant c^2VX$ is the same as $(X-\mu)^2 \geqslant (c\sigma)^2$; hence (8.17) says that

$$\Pr\bigl(|X-\mu| \geqslant c\sigma\bigr) \;\leqslant\; 1/c^2\,. \tag{8.18}$$

Thus, X will lie within c standard deviations of its mean value except with probability at most $1/c^2$. A random variable will lie within 2σ of μ at least 75% of the time; it will lie between $\mu - 10\sigma$ and $\mu + 10\sigma$ at least 99% of the time. These are the cases $\alpha = 4VX$ and $\alpha = 100VX$ of Chebyshev's inequality.

If we roll a pair of fair dice n times, the total value of the n rolls will almost always be near $7n$, for large n. Here's why: The variance of n independent rolls is $\frac{35}{6}n$. A variance of $\frac{35}{6}n$ means a standard deviation of only

$$\sqrt{\tfrac{35}{6}n}\,.$$

So Chebyshev's inequality tells us that the final sum will lie between

$$7n - 10\sqrt{\tfrac{35}{6}n} \quad \text{and} \quad 7n + 10\sqrt{\tfrac{35}{6}n}$$

in at least 99% of all experiments when n fair dice are rolled. For example, the odds are better than 99 to 1 that the total value of a million rolls will be between 6.976 million and 7.024 million.

In general, let X be *any* random variable over a probability space Ω, having finite mean μ and finite standard deviation σ. Then we can consider the probability space Ω^n whose elementary events are n-tuples $(\omega_1, \omega_2, \ldots, \omega_n)$ with each $\omega_k \in \Omega$, and whose probabilities are

$$\Pr(\omega_1, \omega_2, \ldots, \omega_n) \;=\; \Pr(\omega_1)\Pr(\omega_2)\ldots\Pr(\omega_n)\,.$$

If we now define random variables X_k by the formula

$$X_k(\omega_1, \omega_2, \ldots, \omega_n) \;=\; X(\omega_k)\,,$$

the quantity

$$X_1 + X_2 + \cdots + X_n$$

is a sum of n independent random variables, which corresponds to taking n independent "samples" of X on Ω and adding them together. The mean of $X_1 + X_2 + \cdots + X_n$ is $n\mu$, and the standard deviation is $\sqrt{n}\,\sigma$; hence the average of the n samples,

$$\frac{1}{n}(X_1 + X_2 + \cdots + X_n)\,,$$

(That is, the average will fall between the stated limits in at least 99% of all cases when we look at a set of n independent samples, for any fixed value of n. Don't misunderstand this as a statement about the averages of an infinite sequence $X_1, X_2, X_3, \ldots$ as n varies.)

will lie between $\mu - 10\sigma/\sqrt{n}$ and $\mu + 10\sigma/\sqrt{n}$ at least 99% of the time. In other words, if we choose a large enough value of n, the average of n independent samples will almost always be very near the expected value EX. (An even stronger theorem called the Strong Law of Large Numbers is proved in textbooks of probability theory; but the simple consequence of Chebyshev's inequality that we have just derived is enough for our purposes.)

Sometimes we don't know the characteristics of a probability space, and we want to estimate the mean of a random variable X by sampling its value repeatedly. (For example, we might want to know the average temperature at noon on a January day in San Francisco; or we may wish to know the mean life expectancy of insurance agents.) If we have obtained independent empirical observations $X_1, X_2, \ldots, X_n$, we can guess that the true mean is approximately

$$\widehat{E}X \;=\; \frac{X_1 + X_2 + \cdots + X_n}{n}\,. \tag{8.19}$$

And we can also make an estimate of the variance, using the formula

$$\widehat{V}X \;=\; \frac{X_1^2 + X_2^2 + \cdots + X_n^2}{n-1} - \frac{(X_1 + X_2 + \cdots + X_n)^2}{n(n-1)}\,. \tag{8.20}$$

The $(n-1)$'s in this formula look like typographic errors; it seems they should be n's, as in (8.19), because the true variance VX is defined by expected values in (8.15). Yet we get a better estimate with $n-1$ instead of n here, because definition (8.20) implies that

$$E(\widehat{V}X) \;=\; VX\,. \tag{8.21}$$

Here's why:

$$\begin{aligned}
E(\widehat{V}X) &= \frac{1}{n-1}E\Big(\sum_{k=1}^{n} X_k^2 - \frac{1}{n}\sum_{j=1}^{n}\sum_{k=1}^{n} X_jX_k\Big)\\
&= \frac{1}{n-1}\Big(\sum_{k=1}^{n} E(X_k^2) - \frac{1}{n}\sum_{j=1}^{n}\sum_{k=1}^{n} E(X_jX_k)\Big)\\
&= \frac{1}{n-1}\Big(\sum_{k=1}^{n} E(X^2) - \frac{1}{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\big(E(X)^2[j\neq k] + E(X^2)[j=k]\big)\Big)\\
&= \frac{1}{n-1}\Big(nE(X^2) - \frac{1}{n}\big(nE(X^2) + n(n-1)E(X)^2\big)\Big)\\
&= E(X^2) - E(X)^2 \;=\; VX\,.
\end{aligned}$$

(This derivation uses the independence of the observations when it replaces $E(X_jX_k)$ by $(EX)^2[j\neq k] + E(X^2)[j=k]$.)

In practice, experimental results about a random variable X are usually obtained by calculating a sample mean $\hat{\mu} = \widehat{E}X$ and a sample standard deviation $\hat{\sigma} = \sqrt{\widehat{V}X}$, and presenting the answer in the form '$\hat{\mu} \pm \hat{\sigma}/\sqrt{n}$'. For example, here are ten rolls of two supposedly fair dice:

The sample mean of the spot sum S is

$$\hat{\mu} \;=\; (7+11+8+5+4+6+10+8+8+7)/10 \;=\; 7.4\,;$$

the sample variance is

$$(7^2 + 11^2 + 8^2 + 5^2 + 4^2 + 6^2 + 10^2 + 8^2 + 8^2 + 7^2 - 10\hat{\mu}^2)/9 \;\approx\; 2.1^2\,.$$

We estimate the average spot sum of these dice to be $7.4 \pm 2.1/\sqrt{10} = 7.4 \pm 0.7$, on the basis of these experiments.

Let's work one more example of means and variances, in order to show how they can be calculated theoretically instead of empirically. One of the questions we considered in Chapter 5 was the "football victory problem," where n hats are thrown into the air and the result is a random permutation of hats. We showed in equation (5.51) that there's a probability of $n_{\downarrow}/n! \approx 1/e$ that nobody gets the right hat back. We also derived the formula

$$P(n,k) \;=\; \frac{1}{n!}\binom{n}{k}(n-k)_{\downarrow} \;=\; \frac{1}{k!}\frac{(n-k)_{\downarrow}}{(n-k)!} \tag{8.22}$$

for the probability that exactly k people end up with their own hats.

Restating these results in the formalism just learned, we can consider the probability space Π_n of all $n!$ permutations π of $\{1, 2, \ldots, n\}$, where $\Pr(\pi) = 1/n!$ for all $\pi \in \Pi_n$. The random variable

Not to be confused with a Fibonacci number.

$$F_n(\pi) \;=\; \text{number of "fixed points" of } \pi\,, \qquad \text{for } \pi \in \Pi_n,$$

measures the number of correct hat-falls in the football victory problem. Equation (8.22) gives $\Pr(F_n = k)$, but let's pretend that we don't know any such formula; we merely want to study the average value of F_n, and its standard deviation.

The average value is, in fact, extremely easy to calculate, avoiding all the complexities of Chapter 5. We simply observe that

$$\begin{aligned} F_n(\pi) &= F_{n,1}(\pi) + F_{n,2}(\pi) + \cdots + F_{n,n}(\pi)\,, \\ F_{n,k}(\pi) &= [\text{position } k \text{ of } \pi \text{ is a fixed point}]\,, \qquad \text{for } \pi \in \Pi_n. \end{aligned}$$

Hence

$$EF_n \;=\; EF_{n,1} + EF_{n,2} + \cdots + EF_{n,n}\,.$$

And the expected value of $F_{n,k}$ is simply the probability that $F_{n,k} = 1$, which is $1/n$ because exactly $(n-1)!$ of the $n!$ permutations $\pi = \pi_1\pi_2\ldots\pi_n \in \Pi_n$ have $\pi_k = k$. Therefore

$$EF_n \;=\; n/n \;=\; 1\,, \qquad \text{for } n > 0. \tag{8.23}$$

One the average.

On the average, one hat will be in its correct place. "A random permutation has one fixed point, on the average."

Now what's the standard deviation? This question is more difficult, because the $F_{n,k}$'s are not independent of each other. But we can calculate the

variance by analyzing the mutual dependencies among them:

$$\begin{aligned} E(F_n^2) &= E\Bigl(\Bigl(\sum_{k=1}^n F_{n,k}\Bigr)^2\Bigr) = E\Bigl(\sum_{j=1}^n \sum_{k=1}^n F_{n,j}\,F_{n,k}\Bigr) \\ &= \sum_{j=1}^n \sum_{k=1}^n E(F_{n,j}\,F_{n,k}) = \sum_{1\leqslant k\leqslant n} E(F_{n,k}^2) + 2\sum_{1\leqslant j<k\leqslant n} E(F_{n,j}\,F_{n,k})\,. \end{aligned}$$

(We used a similar trick when we derived (2.33) in Chapter 2.) Now $F_{n,k}^2 = F_{n,k}$, since $F_{n,k}$ is either 0 or 1; hence $E(F_{n,k}^2) = EF_{n,k} = 1/n$ as before. And if $j < k$ we have $E(F_{n,j}\,F_{n,k}) = \Pr(\pi$ has both j and k as fixed points$) = (n-2)!/n! = 1/n(n-1)$. Therefore

$$E(F_n^2) = \frac{n}{n} + \binom{n}{2}\frac{2}{n(n-1)} = 2\,, \qquad \text{for } n \geqslant 2. \tag{8.24}$$

(As a check when $n = 3$, we have $\frac{2}{6}0^2 + \frac{3}{6}1^2 + \frac{0}{6}2^2 + \frac{1}{6}3^2 = 2$.) The variance is $E(F_n^2) - (EF_n)^2 = 1$, so the standard deviation (like the mean) is 1. "A random permutation of $n \geqslant 2$ elements has 1 ± 1 fixed points."

8.3 PROBABILITY GENERATING FUNCTIONS

If X is a random variable that takes only nonnegative integer values, we can capture its probability distribution nicely by using the techniques of Chapter 7. The *probability generating function* or pgf of X is

$$G_X(z) = \sum_{k\geqslant 0} \Pr(X=k)\,z^k\,. \tag{8.25}$$

This power series in z contains all the information about the random variable X. We can also express it in two other ways:

$$G_X(z) = \sum_{\omega\in\Omega} \Pr(\omega)\,z^{X(\omega)} = E(z^X)\,. \tag{8.26}$$

The coefficients of $G_X(z)$ are nonnegative, and they sum to 1; the latter condition can be written

$$G_X(1) = 1\,. \tag{8.27}$$

Conversely, any power series $G(z)$ with nonnegative coefficients and with $G(1) = 1$ is the pgf of some random variable.

The nicest thing about pgf's is that they usually simplify the computation of means and variances. For example, the mean is easily expressed:

$$\begin{aligned} EX &= \sum_{k\geqslant 0} k\cdot \Pr(X=k) \\ &= \sum_{k\geqslant 0} \Pr(X=k)\cdot kz^{k-1}\big|_{z=1} \\ &= G'_X(1)\,. \end{aligned} \tag{8.28}$$

We simply differentiate the pgf with respect to z and set $z=1$.

The variance is only slightly more complicated:

$$\begin{aligned} E(X^2) &= \sum_{k\geqslant 0} k^2\cdot \Pr(X=k) \\ &= \sum_{k\geqslant 0} \Pr(X=k)\cdot \big(k(k-1)z^{k-2}+kz^{k-1}\big)\big|_{z=1} \;=\; G''_X(1)+G'_X(1)\,. \end{aligned}$$

Therefore

$$VX \;=\; G''_X(1)+G'_X(1)-G'_X(1)^2\,. \tag{8.29}$$

Equations (8.28) and (8.29) tell us that we can compute the mean and variance if we can compute the values of two derivatives, $G'_X(1)$ and $G''_X(1)$. We don't have to know a closed form for the probabilities; we don't even have to know a closed form for $G_X(z)$ itself.

It is convenient to write

$$\mathrm{Mean}(G) \;=\; G'(1)\,, \tag{8.30}$$

$$\mathrm{Var}(G) \;=\; G''(1)+G'(1)-G'(1)^2\,, \tag{8.31}$$

when G is any function, since we frequently want to compute these combinations of derivatives.

The second-nicest thing about pgf's is that they are comparatively simple functions of z, in many important cases. For example, let's look at the *uniform distribution* of order n, in which the random variable takes on each of the values $\{0,1,\ldots,n-1\}$ with probability $1/n$. The pgf in this case is

$$U_n(z) \;=\; \frac{1}{n}(1+z+\cdots+z^{n-1}) \;=\; \frac{1}{n}\frac{1-z^n}{1-z}\,, \qquad \text{for } n\geqslant 1. \tag{8.32}$$

We have a closed form for $U_n(z)$ because this is a geometric series.

But this closed form proves to be somewhat embarrassing: When we plug in $z=1$ (the value of z that's most critical for the pgf), we get the undefined ratio $0/0$, even though $U_n(z)$ is a polynomial that is perfectly well defined at any value of z. The value $U_n(1)=1$ is obvious from the non-closed form

$(1+z+\cdots+z^{n-1})/n$, yet it seems that we must resort to L'Hospital's rule to find $\lim_{z\to 1} U_n(z)$ if we want to determine $U_n(1)$ from the closed form. The determination of $U_n'(1)$ by L'Hospital's rule will be even harder, because there will be a factor of $(z-1)^2$ in the denominator; $U_n''(1)$ will be harder still.

Luckily there's a nice way out of this dilemma. If $G(z)=\sum_{n\geqslant 0} g_n z^n$ is any power series that converges for at least one value of z with $|z|>1$, the power series $G'(z)=\sum_{n\geqslant 0} n g_n z^{n-1}$ will also have this property, and so will $G''(z)$, $G'''(z)$, etc. Therefore by Taylor's theorem we can write

$$G(1+t) \;=\; G(1)+\frac{G'(1)}{1!}t+\frac{G''(1)}{2!}t^2+\frac{G'''(1)}{3!}t^3+\cdots; \tag{8.33}$$

all derivatives of $G(z)$ at $z=1$ will appear as coefficients, when $G(1+t)$ is expanded in powers of t.

For example, the derivatives of the uniform pgf $U_n(z)$ are easily found in this way:

$$\begin{aligned} U_n(1+t) &= \frac{1}{n}\frac{(1+t)^n-1}{t} \\ &= \frac{1}{n}\binom{n}{1}+\frac{1}{n}\binom{n}{2}t+\frac{1}{n}\binom{n}{3}t^2+\cdots+\frac{1}{n}\binom{n}{n}t^{n-1}. \end{aligned}$$

Comparing this to (8.33) gives

$$U_n(1) \;=\; 1; \quad U_n'(1) \;=\; \frac{n-1}{2}; \quad U_n''(1) \;=\; \frac{(n-1)(n-2)}{3}; \tag{8.34}$$

and in general $U_n^{(m)}(1)=(n-1)^{\underline{m}}/(m+1)$, although we need only the cases $m=1$ and $m=2$ to compute the mean and the variance. The mean of the uniform distribution is

$$U_n'(1) \;=\; \frac{n-1}{2}, \tag{8.35}$$

and the variance is

$$\begin{aligned} U_n''(1)+U_n'(1)-U_n'(1)^2 &= 4\,\frac{(n-1)(n-2)}{12}+6\,\frac{(n-1)}{12}-3\,\frac{(n-1)^2}{12} \\ &= \frac{n^2-1}{12}. \end{aligned} \tag{8.36}$$

The third-nicest thing about pgf's is that the product of pgf's corresponds to the sum of independent random variables. We learned in Chapters 5 and 7 that the product of generating functions corresponds to the convolution of sequences; but it's even more important in applications to know that the convolution of probabilities corresponds to the sum of independent random

variables. Indeed, if X and Y are random variables that take on nothing but integer values, the probability that $X+Y=n$ is

$$\Pr(X+Y=n) \;=\; \sum_k \Pr(X=k \text{ and } Y=n-k)\,.$$

If X and Y are independent, we now have

$$\Pr(X+Y=n) \;=\; \sum_k \Pr(X=k)\,\Pr(Y=n-k)\,,$$

a convolution. Therefore — and this is the punch line —

$$G_{X+Y}(z) \;=\; G_X(z)\,G_Y(z)\,, \qquad \text{if } X \text{ and } Y \text{ are independent.} \tag{8.37}$$

Earlier this chapter we observed that $V(X+Y) = VX + VY$ when X and Y are independent. Let $F(z)$ and $G(z)$ be the pgf's for X and Y, and let $H(z)$ be the pgf for $X+Y$. Then

$$H(z) \;=\; F(z)G(z)\,,$$

and our formulas (8.28) through (8.31) for mean and variance tell us that we must have

$$\text{Mean}(H) \;=\; \text{Mean}(F) + \text{Mean}(G)\,; \tag{8.38}$$

$$\text{Var}(H) \;=\; \text{Var}(F) + \text{Var}(G)\,. \tag{8.39}$$

These formulas, which are properties of the derivatives $\text{Mean}(H) = H'(1)$ and $\text{Var}(H) = H''(1) + H'(1) - H'(1)^2$, aren't valid for arbitrary function products $H(z) = F(z)G(z)$; we have

$$\begin{aligned} H'(z) &= F'(z)G(z) + F(z)G'(z)\,, \\ H''(z) &= F''(z)G(z) + 2F'(z)G'(z) + F(z)G''(z)\,. \end{aligned}$$

But if we set $z=1$, we can see that (8.38) and (8.39) will be valid in general provided only that

$$F(1) \;=\; G(1) \;=\; 1 \tag{8.40}$$

and that the derivatives exist. The "probabilities" don't have to be in $[0\,..\,1]$ for these formulas to hold. We can normalize the functions $F(z)$ and $G(z)$ by dividing through by $F(1)$ and $G(1)$ in order to make this condition valid, whenever $F(1)$ and $G(1)$ are nonzero.

I'll graduate magna cum ulant.

Mean and variance aren't the whole story. They are merely two of an infinite series of so-called *cumulant* statistics introduced by the Danish astronomer Thorvald Nicolai Thiele [351] in 1903. The first two cumulants

κ_1 and κ_2 of a random variable are what we have called the mean and the variance; there also are higher-order cumulants that express more subtle properties of a distribution. The general formula

$$\ln G(e^t) \;=\; \frac{\kappa_1}{1!}t + \frac{\kappa_2}{2!}t^2 + \frac{\kappa_3}{3!}t^3 + \frac{\kappa_4}{4!}t^4 + \cdots \tag{8.41}$$

defines the cumulants of all orders, when $G(z)$ is the pgf of a random variable.

Let's look at cumulants more closely. If $G(z)$ is the pgf for X, we have

$$\begin{aligned} G(e^t) \;&=\; \sum_{k\geqslant 0} \Pr(X{=}k)e^{kt} \;=\; \sum_{k,m\geqslant 0} \Pr(X{=}k)\frac{k^m t^m}{m!} \\ &=\; 1 + \frac{\mu_1}{1!}t + \frac{\mu_2}{2!}t^2 + \frac{\mu_3}{3!}t^3 + \cdots\,, \end{aligned} \tag{8.42}$$

where

$$\mu_m \;=\; \sum_{k\geqslant 0} k^m \Pr(X{=}k) \;=\; E(X^m)\,. \tag{8.43}$$

This quantity μ_m is called the "mth moment" of X. We can take exponentials on both sides of (8.41), obtaining another formula for $G(e^t)$:

$$\begin{aligned} G(e^t) \;&=\; 1 + \frac{(\kappa_1 t + \frac{1}{2}\kappa_2 t^2 + \cdots)}{1!} + \frac{(\kappa_1 t + \frac{1}{2}\kappa_2 t^2 + \cdots)^2}{2!} + \cdots \\ &=\; 1 + \kappa_1 t + \tfrac{1}{2}(\kappa_2 + \kappa_1^2)t^2 + \cdots\,. \end{aligned}$$

Equating coefficients of powers of t leads to a series of formulas

$$\kappa_1 \;=\; \mu_1\,, \tag{8.44}$$
$$\kappa_2 \;=\; \mu_2 - \mu_1^2\,, \tag{8.45}$$
$$\kappa_3 \;=\; \mu_3 - 3\mu_1\mu_2 + 2\mu_1^3\,, \tag{8.46}$$
$$\kappa_4 \;=\; \mu_4 - 4\mu_1\mu_3 + 12\mu_1^2\mu_2 - 3\mu_2^2 - 6\mu_1^4\,, \tag{8.47}$$
$$\begin{aligned} \kappa_5 \;=\; \mu_5 - 5\mu_1\mu_4 + 20\mu_1^2\mu_3 - 10\mu_2\mu_3 \\ + 30\mu_1\mu_2^2 - 60\mu_1^3\mu_2 + 24\mu_1^5\,, \end{aligned} \tag{8.48}$$
$$\vdots$$

defining the cumulants in terms of the moments. Notice that κ_2 is indeed the variance, $E(X^2) - (EX)^2$, as claimed.

Equation (8.41) makes it clear that the cumulants defined by the product $F(z)G(z)$ of two pgf's will be the sums of the corresponding cumulants of $F(z)$ and $G(z)$, because logarithms of products are sums. Therefore all cumulants of the sum of independent random variables are additive, just as the mean and variance are. This property makes cumulants more important than moments.

"For these higher half-invariants we shall propose no special names."
— T. N. Thiele [351]

If we take a slightly different tack, writing

$$G(1+t) \;=\; 1 + \frac{\alpha_1}{1!}t + \frac{\alpha_2}{2!}t^2 + \frac{\alpha_3}{3!}t^3 + \cdots\,,$$

equation (8.33) tells us that the α's are the "factorial moments"

$$\begin{aligned}
\alpha_m &= G^{(m)}(1) \\
&= \sum_{k\geqslant 0} \Pr(X=k)k^{\underline{m}}z^{k-m}\Big|_{z=1} \\
&= \sum_{k\geqslant 0} k^{\underline{m}}\Pr(X=k) \\
&= E(X^{\underline{m}})\,. \qquad (8.49)
\end{aligned}$$

It follows that

$$\begin{aligned}
G(e^t) &= 1 + \frac{\alpha_1}{1!}(e^t-1) + \frac{\alpha_2}{2!}(e^t-1)^2 + \cdots \\
&= 1 + \frac{\alpha_1}{1!}(t+\tfrac{1}{2}t^2+\cdots) + \frac{\alpha_2}{2!}(t^2+t^3+\cdots) + \cdots \\
&= 1 + \alpha_1 t + \tfrac{1}{2}(\alpha_2+\alpha_1)t^2 + \cdots\,,
\end{aligned}$$

and we can express the cumulants in terms of the derivatives $G^{(m)}(1)$:

$$\begin{aligned}
\kappa_1 &= \alpha_1\,, && (8.50)\\
\kappa_2 &= \alpha_2 + \alpha_1 - \alpha_1^2\,, && (8.51)\\
\kappa_3 &= \alpha_3 + 3\alpha_2 + \alpha_1 - 3\alpha_2\alpha_1 - 3\alpha_1^2 + 2\alpha_1^3\,, && (8.52)\\
&\;\;\vdots
\end{aligned}$$

This sequence of formulas yields "additive" identities that extend (8.38) and (8.39) to all the cumulants.

Let's get back down to earth and apply these ideas to simple examples. The simplest case of a random variable is a "random constant," where X has a certain fixed value x with probability 1. In this case $G_X(z) = z^x$, and $\ln G_X(e^t) = xt$; hence the mean is x and all other cumulants are zero. It follows that the operation of multiplying any pgf by z^x increases the mean by x but leaves the variance and all other cumulants unchanged.

How do probability generating functions apply to dice? The distribution of spots on one fair die has the pgf

$$G(z) \;=\; \frac{z+z^2+z^3+z^4+z^5+z^6}{6} \;=\; zU_6(z)\,,$$

where U_6 is the pgf for the uniform distribution of order 6. The factor 'z' adds 1 to the mean, so the mean is 3.5 instead of $\frac{n-1}{2} = 2.5$ as given in (8.35); but an extra 'z' does not affect the variance (8.36), which equals $\frac{35}{12}$.

The pgf for total spots on two independent dice is the square of the pgf for spots on one die,

$$\begin{aligned} G_S(z) &= \frac{z^2 + 2z^3 + 3z^4 + 4z^5 + 5z^6 + 6z^7 + 5z^8 + 4z^9 + 3z^{10} + 2z^{11} + z^{12}}{36} \\ &= z^2 U_6(z)^2 . \end{aligned}$$

If we roll a pair of fair dice n times, the probability that we get a total of k spots overall is, similarly,

$$\begin{aligned} [z^k]\, G_S(z)^n &= [z^k]\, z^{2n} U_6(z)^{2n} \\ &= [z^{k-2n}]\, U_6(z)^{2n} . \end{aligned}$$

In the hats-off-to-football-victory problem considered earlier, otherwise known as the problem of enumerating the fixed points of a random permutation, we know from (5.49) that the pgf is

Hat distribution is a different kind of uniform distribution.

$$F_n(z) = \sum_{0 \leqslant k \leqslant n} \frac{(n-k)_{\downarrow}}{(n-k)!} \frac{z^k}{k!} , \qquad \text{for } n \geqslant 0. \tag{8.53}$$

Therefore

$$\begin{aligned} F_n'(z) &= \sum_{1 \leqslant k \leqslant n} \frac{(n-k)_{\downarrow}}{(n-k)!} \frac{z^{k-1}}{(k-1)!} \\ &= \sum_{0 \leqslant k \leqslant n-1} \frac{(n-1-k)_{\downarrow}}{(n-1-k)!} \frac{z^k}{k!} \\ &= F_{n-1}(z) . \end{aligned}$$

Without knowing the details of the coefficients, we can conclude from this recurrence $F_n'(z) = F_{n-1}(z)$ that $F_n^{(m)}(z) = F_{n-m}(z)$; hence

$$F_n^{(m)}(1) = F_{n-m}(1) = [n \geqslant m] . \tag{8.54}$$

This formula makes it easy to calculate the mean and variance; we find as before (but more quickly) that they are both equal to 1 when $n \geqslant 2$.

In fact, we can now show that the mth cumulant κ_m of this random variable is equal to 1 whenever $n \geqslant m$. For the mth cumulant depends only on $F_n'(1)$, $F_n''(1)$, ..., $F_n^{(m)}(1)$, and these are all equal to 1; hence we obtain

the same answer for the mth cumulant as we do when we replace $F_n(z)$ by the limiting pgf

$$F_\infty(z) \;=\; e^{z-1}\,, \tag{8.55}$$

which has $F_\infty^{(m)}(1) = 1$ for derivatives of all orders. The cumulants of F_∞ are identically equal to 1, because

$$\ln F_\infty(e^t) \;=\; \ln e^{e^t-1} \;=\; e^t - 1 \;=\; \frac{t}{1!} + \frac{t^2}{2!} + \frac{t^3}{3!} + \cdots\,.$$

8.4 FLIPPING COINS

Con artists know that $p \approx 0.1$ when you spin a newly minted U.S. penny on a smooth table. (The weight distribution makes Lincoln's head fall downward.)

Now let's turn to processes that have just two outcomes. If we flip a coin, there's probability p that it comes up heads and probability q that it comes up tails, where

$$p + q \;=\; 1\,.$$

(We assume that the coin doesn't come to rest on its edge, or fall into a hole, etc.) Throughout this section, the numbers p and q will always sum to 1. If the coin is *fair*, we have $p = q = \frac{1}{2}$; otherwise the coin is said to be *biased*.

The probability generating function for the number of heads after one toss of a coin is

$$H(z) \;=\; q + pz\,. \tag{8.56}$$

If we toss the coin n times, always assuming that different coin tosses are independent, the number of heads is generated by

$$H(z)^n \;=\; (q+pz)^n \;=\; \sum_{k\geqslant 0} \binom{n}{k} p^k q^{n-k} z^k\,, \tag{8.57}$$

according to the binomial theorem. Thus, the chance that we obtain exactly k heads in n tosses is $\binom{n}{k}p^k q^{n-k}$. This sequence of probabilities is called the *binomial distribution*.

Suppose we toss a coin repeatedly until heads first turns up. What is the probability that exactly k tosses will be required? We have $k = 1$ with probability p (since this is the probability of heads on the first flip); we have $k = 2$ with probability qp (since this is the probability of tails first, then heads); and for general k the probability is $q^{k-1}p$. So the generating function is

$$pz + qpz^2 + q^2pz^3 + \cdots \;=\; \frac{pz}{1-qz}\,. \tag{8.58}$$

Repeating the process until n heads are obtained gives the pgf

$$\begin{aligned}\left(\frac{pz}{1-qz}\right)^n &= p^n z^n \sum_k \binom{n+k-1}{k}(qz)^k \\ &= \sum_k \binom{k-1}{k-n} p^n q^{k-n} z^k . \qquad (8.59)\end{aligned}$$

This, incidentally, is z^n times

$$\left(\frac{p}{1-qz}\right)^n = \sum_k \binom{n+k-1}{k} p^n q^k z^k , \qquad (8.60)$$

the generating function for the *negative binomial distribution*.

The probability space in example (8.59), where we flip a coin until n heads have appeared, is different from the probability spaces we've seen earlier in this chapter, because it contains infinitely many elements. Each element is a finite sequence of heads and/or tails, containing precisely n heads in all, and ending with heads; the probability of such a sequence is $p^n q^{k-n}$, where $k-n$ is the number of tails. Thus, for example, if $n = 3$ and if we write H for heads and T for tails, the sequence THTTTHH is an element of the probability space, and its probability is $qpqqqpp = p^3 q^4$.

Heads I win, tails you lose.

No? OK; tails you lose, heads I win.

No? Well, then, heads you lose, tails I win.

Let X be a random variable with the binomial distribution (8.57), and let Y be a random variable with the negative binomial distribution (8.60). These distributions depend on n and p. The mean of X is $nH'(1) = np$, since its pgf is $H(z)^n$; the variance is

$$n\big(H''(1) + H'(1) - H'(1)^2\big) = n(0 + p - p^2) = npq . \qquad (8.61)$$

Thus the standard deviation is $\sqrt{npq}$: If we toss a coin n times, we expect to get heads about $np \pm \sqrt{npq}$ times. The mean and variance of Y can be found in a similar way: If we let

$$G(z) = \frac{p}{1-qz} ,$$

we have

$$\begin{aligned} G'(z) &= \frac{pq}{(1-qz)^2} , \\ G''(z) &= \frac{2pq^2}{(1-qz)^3} ; \end{aligned}$$

hence $G'(1) = pq/p^2 = q/p$ and $G''(1) = 2pq^2/p^3 = 2q^2/p^2$. It follows that the mean of Y is nq/p and the variance is nq/p^2.

A simpler way to derive the mean and variance of Y is to use the reciprocal generating function

$$F(z) = \frac{1-qz}{p} = \frac{1}{p} - \frac{q}{p}z, \tag{8.62}$$

and to write

$$G(z)^n = F(z)^{-n}. \tag{8.63}$$

This polynomial $F(z)$ is not a probability generating function, because it has a negative coefficient. But it does satisfy the crucial condition $F(1) = 1$. Thus $F(z)$ is formally a binomial that corresponds to a coin for which we get heads with "probability" equal to $-q/p$; and $G(z)$ is formally equivalent to flipping such a coin -1 times(!). The negative binomial distribution with parameters (n, p) can therefore be regarded as the ordinary binomial distribution with parameters $(n', p') = (-n, -q/p)$. Proceeding formally, the mean must be $n'p' = (-n)(-q/p) = nq/p$, and the variance must be $n'p'q' = (-n)(-q/p)(1 + q/p) = nq/p^2$. This formal derivation involving negative probabilities is valid, because our derivation for ordinary binomials was based on identities between formal power series in which the assumption $0 \leqslant p \leqslant 1$ was never used.

The probability is negative that I'm getting younger.

Oh? Then it's >1 that you're getting older, or staying the same.

Let's move on to another example: How many times do we have to flip a coin until we get heads twice in a row? The probability space now consists of all sequences of H's and T's that end with HH but have no consecutive H's until the final position:

$$\Omega = \{\mathtt{HH}, \mathtt{THH}, \mathtt{TTHH}, \mathtt{HTHH}, \mathtt{TTTHH}, \mathtt{THTHH}, \mathtt{HTTHH}, \ldots\}.$$

The probability of any given sequence is obtained by replacing H by p and T by q; for example, the sequence THTHH will occur with probability

$$\Pr(\mathtt{THTHH}) = qpqpp = p^3q^2.$$

We can now play with generating functions as we did at the beginning of Chapter 7, letting S be the infinite sum

$$S = \mathtt{HH} + \mathtt{THH} + \mathtt{TTHH} + \mathtt{HTHH} + \mathtt{TTTHH} + \mathtt{THTHH} + \mathtt{HTTHH} + \cdots$$

of all the elements of Ω. If we replace each H by pz and each T by qz, we get the probability generating function for the number of flips needed until two consecutive heads turn up.

There's a curious relation between S and the sum of domino tilings

$$T = \mathrm{I} + \square + \square\square + \square\!\!\square + \square\square\square + \square\square\!\!\square + \square\!\!\square\square + \cdots$$

in equation (7.1). Indeed, we obtain S from T if we replace each $\square$ by $\mathtt{T}$ and each $\square\!\!\square$ by $\mathtt{HT}$, then tack on an $\mathtt{HH}$ at the end. This correspondence is easy to prove because each element of Ω has the form $(\mathtt{T}+\mathtt{HT})^n\mathtt{HH}$ for some $n \geqslant 0$, and each term of T has the form $(\square + \square\!\!\square)^n$. Therefore by (7.4) we have

$$S = (1 - \mathtt{T} - \mathtt{HT})^{-1}\mathtt{HH}\,,$$

and the probability generating function for our problem is

$$\begin{aligned} G(z) &= \big(1 - qz - (pz)(qz)\big)^{-1}(pz)^2 \\ &= \frac{p^2z^2}{1 - qz - pqz^2}\,. \end{aligned} \tag{8.64}$$

Our experience with the negative binomial distribution gives us a clue that we can most easily calculate the mean and variance of (8.64) by writing

$$G(z) = \frac{z^2}{F(z)}\,,$$

where

$$F(z) = \frac{1 - qz - pqz^2}{p^2}\,,$$

and by calculating the "mean" and "variance" of this pseudo-pgf $F(z)$. (Once again we've introduced a function with $F(1) = 1$.) We have

$$\begin{aligned} F'(1) &= (-q - 2pq)/p^2 = 2 - p^{-1} - p^{-2}\,; \\ F''(1) &= -2pq/p^2 = 2 - 2p^{-1}\,. \end{aligned}$$

Therefore, since $z^2 = F(z)G(z)$, $\mathrm{Mean}(z^2) = 2$, and $\mathrm{Var}(z^2) = 0$, the mean and variance of distribution $G(z)$ are

$$\mathrm{Mean}(G) = 2 - \mathrm{Mean}(F) = p^{-2} + p^{-1}\,; \tag{8.65}$$

$$\mathrm{Var}(G) = -\mathrm{Var}(F) = p^{-4} + 2p^{-3} - 2p^{-2} - p^{-1}\,. \tag{8.66}$$

When $p = \frac{1}{2}$ the mean and variance are 6 and 22, respectively. (Exercise 4 discusses the calculation of means and variances by subtraction.)

Now let's try a more intricate experiment: We will flip coins until the pattern THTTH is first obtained. The sum of winning positions is now

$$\begin{aligned} S &= \mathtt{THTTH} + \mathtt{HTHTTH} + \mathtt{TTHTTH} \\ &\quad + \mathtt{HHTHTTH} + \mathtt{HTTHTTH} + \mathtt{THTHTTH} + \mathtt{TTTHTTH} + \cdots ; \end{aligned}$$

this sum is more difficult to describe than the previous one. If we go back to the method by which we solved the domino problems in Chapter 7, we can obtain a formula for S by considering it as a "finite state language" defined by the following "automaton":

"'You really are an automaton — a calculating machine,' I cried. 'There is something positively inhuman in you at times.'"
— J. H. Watson [83]

The elementary events in the probability space are the sequences of H's and T's that lead from state 0 to state 5. Suppose, for example, that we have just seen THT; then we are in state 3. Flipping tails now takes us to state 4; flipping heads in state 3 would take us to state 2 (not all the way back to state 0, since the TH we've just seen may be followed by TTH).

In this formulation, we can let S_k be the sum of all sequences of H's and T's that lead to state k; it follows that

$$\begin{aligned} S_0 &= 1 + S_0\,\mathtt{H} + S_2\,\mathtt{H}, \\ S_1 &= S_0\,\mathtt{T} + S_1\,\mathtt{T} + S_4\,\mathtt{T}, \\ S_2 &= S_1\,\mathtt{H} + S_3\,\mathtt{H}, \\ S_3 &= S_2\,\mathtt{T}, \\ S_4 &= S_3\,\mathtt{T}, \\ S_5 &= S_4\,\mathtt{H}. \end{aligned}$$

Now the sum S in our problem is S_5; we can obtain it by solving these six equations in the six unknowns $S_0, S_1, \ldots, S_5$. Replacing H by pz and T by qz gives generating functions where the coefficient of z^n in S_k is the probability that we are in state k after n flips.

In the same way, any diagram of transitions between states, where the transition from state j to state k occurs with given probability $p_{j,k}$, leads to a set of simultaneous linear equations whose solutions are generating functions for the state probabilities after n transitions have occurred. Systems of this kind are called *Markov processes*, and the theory of their behavior is intimately related to the theory of linear equations.

But the coin-flipping problem can be solved in a much simpler way, without the complexities of the general finite-state approach. Instead of six equations in six unknowns S_0, S_1, ..., S_5, we can characterize S with only two equations in two unknowns. The trick is to consider the auxiliary sum $N = S_0 + S_1 + S_2 + S_3 + S_4$ of all flip sequences that don't contain any occurrences of the given pattern THTTH:

$$N = 1 + \mathtt{H} + \mathtt{T} + \mathtt{HH} + \cdots + \mathtt{THTHT} + \mathtt{THTTT} + \cdots .$$

We have

$$1 + N(\mathtt{H} + \mathtt{T}) = N + S , \tag{8.67}$$

because every term on the left either ends with THTTH (and belongs to S) or doesn't (and belongs to N); conversely, every term on the right is either empty or belongs to $N\,\mathtt{H}$ or $N\,\mathtt{T}$. And we also have the important additional equation

$$N\,\mathtt{THTTH} = S + S\,\mathtt{TTH} , \tag{8.68}$$

because every term on the left completes a term of S after either the first H or the second H, and because every term on the right belongs to the left.

The solution to these two simultaneous equations is easily obtained: We have $N = (1 - S)(1 - \mathtt{H} - \mathtt{T})^{-1}$ from (8.67), hence

$$(1 - S)(1 - \mathtt{T} - \mathtt{H})^{-1}\,\mathtt{THTTH} = S(1 + \mathtt{TTH}) .$$

As before, we get the probability generating function $G(z)$ for the number of flips if we replace H by pz and T by qz. A bit of simplification occurs since $p + q = 1$, and we find

$$\frac{\big(1 - G(z)\big)\,p^2q^3z^5}{1 - z} = G(z)(1 + pq^2z^3) ;$$

hence the solution is

$$G(z) = \frac{p^2q^3z^5}{p^2q^3z^5 + (1 + pq^2z^3)(1 - z)} . \tag{8.69}$$

Notice that $G(1) = 1$, if $pq \neq 0$; we do eventually encounter the pattern THTTH, with probability 1, unless the coin is rigged so that it always comes up heads or always tails.

To get the mean and variance of the distribution (8.69), we invert $G(z)$ as we did in the previous problem, writing $G(z) = z^5/F(z)$ where F is a polynomial:

$$F(z) = \frac{p^2q^3z^5 + (1 + pq^2z^3)(1 - z)}{p^2q^3} . \tag{8.70}$$

The relevant derivatives are

$$\begin{aligned} F'(1) &= 5-(1+pq^2)/p^2q^3\,, \\ F''(1) &= 20-6pq^2/p^2q^3\,; \end{aligned}$$

and if X is the number of flips we get

$$EX = \mathrm{Mean}(G) = 5-\mathrm{Mean}(F) = p^{-2}q^{-3}+p^{-1}q^{-1}\,; \tag{8.71}$$

$$\begin{aligned} VX = \mathrm{Var}(G) &= -\mathrm{Var}(F) \\ &= -25+p^{-2}q^{-3}+7p^{-1}q^{-1}+\mathrm{Mean}(F)^2 \\ &= (EX)^2-9p^{-2}q^{-3}-3p^{-1}q^{-1}\,. \end{aligned} \tag{8.72}$$

When $p=\frac{1}{2}$, the mean and variance are 36 and 996.

Let's get general: The problem we have just solved was "random" enough to show us how to analyze the case that we are waiting for the first appearance of an *arbitrary* pattern A of heads and tails. Again we let S be the sum of all winning sequences of H's and T's, and we let N be the sum of all sequences that haven't encountered the pattern A yet. Equation (8.67) will remain the same; equation (8.68) will become

$$\begin{aligned} NA = S\big(1 &+ A^{(1)}\,[A^{(m-1)}=A_{(m-1)}]+A^{(2)}\,[A^{(m-2)}=A_{(m-2)}] \\ &+\cdots+A^{(m-1)}\,[A^{(1)}=A_{(1)}]\big)\,, \end{aligned} \tag{8.73}$$

where m is the length of A, and where $A^{(k)}$ and $A_{(k)}$ denote respectively the last k characters and the first k characters of A. For example, if A is the pattern THTTH we just studied, we have

$$\begin{aligned} A^{(1)} &= \mathtt{H}, \quad A^{(2)} = \mathtt{TH}, \quad A^{(3)} = \mathtt{TTH}, \quad A^{(4)} = \mathtt{HTTH}; \\ A_{(1)} &= \mathtt{T}, \quad A_{(2)} = \mathtt{TH}, \quad A_{(3)} = \mathtt{THT}, \quad A_{(4)} = \mathtt{THTT}. \end{aligned}$$

Since the only perfect match is $A^{(2)}=A_{(2)}$, equation (8.73) reduces to (8.68).

Let $\widetilde{A}$ be the result of substituting p^{-1} for H and q^{-1} for T in the pattern A. Then it is not difficult to generalize our derivation of (8.71) and (8.72) to conclude (exercise 20) that the general mean and variance are

$$EX = \sum_{k=1}^{m} \widetilde{A}_{(k)}\,[A^{(k)}=A_{(k)}]\,; \tag{8.74}$$

$$VX = (EX)^2-\sum_{k=1}^{m}(2k-1)\widetilde{A}_{(k)}\,[A^{(k)}=A_{(k)}]\,. \tag{8.75}$$

In the special case $p = \frac{1}{2}$ we can interpret these formulas in a particularly simple way. Given a pattern A of m heads and tails, let

$$A{:}A \;=\; \sum_{k=1}^{m} 2^{k-1}\,[A^{(k)} = A_{(k)}]\,. \tag{8.76}$$

We can easily find the binary representation of this number by placing a '1' under each position such that the string matches itself perfectly when it is superimposed on a copy of itself that has been shifted to start in this position:

```
  A =  HTHTHHTHTH
A:A = (1000010101)_2 = 512 + 16 + 4 + 1 = 533
        HTHTHHTHTH     √
         HTHTHHTHTH
          HTHTHHTHTH
           HTHTHHTHTH
            HTHTHHTHTH
             HTHTHHTHTH     √
              HTHTHHTHTH
               HTHTHHTHTH     √
                HTHTHHTHTH
                 HTHTHHTHTH     √
```

Equation (8.74) now tells us that the expected number of flips until pattern A appears is exactly $2(A{:}A)$, if we use a fair coin, because $\widetilde{A}_{(k)} = 2^k$ when $p = q = \frac{1}{2}$. This result, first discovered by the Soviet mathematician A. D. Solov'ev in 1966 [331], seems paradoxical at first glance: Patterns with no self-overlaps occur sooner than overlapping patterns do! It takes almost twice as long to encounter HHHHH as it does to encounter HHHHT or THHHH.

"Chem bol'she periodov u nashego slova, tem pozzhe ono poi͡avli͡aetsi͡a."
— A. D. Solov'ev

Now let's consider an amusing game that was invented by (of all people) Walter Penney [289] in 1969. Alice and Bill flip a coin until either HHT or HTT occurs; Alice wins if the pattern HHT comes first, Bill wins if HTT comes first. This game — now called "Penney ante" — certainly seems to be fair, if played with a fair coin, because both patterns HHT and HTT have the same characteristics if we look at them in isolation: The probability generating function for the waiting time until HHT first occurs is

$$G(z) \;=\; \frac{z^3}{z^3 - 8(z-1)}\,,$$

and the same is true for HTT. Therefore neither Alice nor Bill has an advantage, if they play solitaire.

Of course not! Who could they have an advantage over?

But there's an interesting interplay between the patterns when both are considered simultaneously. Let S_A be the sum of Alice's winning configurations, and let S_B be the sum of Bill's:

$$\begin{aligned} S_A &= \mathtt{HHT} + \mathtt{HHHT} + \mathtt{THHT} + \mathtt{HHHHT} + \mathtt{HTHHT} + \mathtt{THHHT} + \cdots ; \\ S_B &= \mathtt{HTT} + \mathtt{THTT} + \mathtt{HTHTT} + \mathtt{TTHTT} + \mathtt{THTHTT} + \mathtt{TTTHTT} + \cdots . \end{aligned}$$

Also — taking our cue from the trick that worked when only one pattern was involved — let us denote by N the sum of all sequences in which neither player has won so far:

$$N = 1 + \mathtt{H} + \mathtt{T} + \mathtt{HH} + \mathtt{HT} + \mathtt{TH} + \mathtt{TT} + \mathtt{HHH} + \mathtt{HTH} + \mathtt{THH} + \cdots . \tag{8.77}$$

Then we can easily verify the following set of equations:

$$\begin{aligned} 1 + N(\mathtt{H}+\mathtt{T}) &= N + S_A + S_B ; \\ N\,\mathtt{HHT} &= S_A ; \\ N\,\mathtt{HTT} &= S_A\,\mathtt{T} + S_B . \end{aligned} \tag{8.78}$$

If we now set $\mathtt{H} = \mathtt{T} = \frac{1}{2}$, the resulting value of S_A becomes the probability that Alice wins, and S_B becomes the probability that Bill wins. The three equations reduce to

$$1 + N = N + S_A + S_B ; \qquad \tfrac{1}{8}N = S_A ; \qquad \tfrac{1}{8}N = \tfrac{1}{2}S_A + S_B ;$$

and we find $S_A = \frac{2}{3}$, $S_B = \frac{1}{3}$. Alice will win about twice as often as Bill!

In a generalization of this game, Alice and Bill choose patterns A and B of heads and tails, and they flip coins until either A or B appears. The two patterns need not have the same length, but we assume that A doesn't occur within B, nor does B occur within A. (Otherwise the game would be degenerate. For example, if $A = \mathtt{HT}$ and $B = \mathtt{THTH}$, poor Bill could never win; and if $A = \mathtt{HTH}$ and $B = \mathtt{TH}$, both players might claim victory simultaneously.) Then we can write three equations analogous to (8.73) and (8.78):

$$\begin{aligned} 1 + N(\mathtt{H}+\mathtt{T}) &= N + S_A + S_B ; \\ NA &= S_A \sum_{k=1}^{l} A^{(l-k)}\,[A^{(k)} = A_{(k)}] \;+\; S_B \sum_{k=1}^{\min(l,m)} A^{(l-k)}\,[B^{(k)} = A_{(k)}] ; \\ NB &= S_A \sum_{k=1}^{\min(l,m)} B^{(m-k)}\,[A^{(k)} = B_{(k)}] \;+\; S_B \sum_{k=1}^{m} B^{(m-k)}\,[B^{(k)} = B_{(k)}] . \end{aligned} \tag{8.79}$$

Here l is the length of A and m is the length of B. For example, if we have $A = \mathtt{HTTHTHTH}$ and $B = \mathtt{THTHTTH}$, the two pattern-dependent equations are

$$\begin{aligned} N\,\mathtt{HTTHTHTH} &= S_A\,\mathtt{TTHTHTH} + S_A + S_B\,\mathtt{TTHTHTH} + S_B\,\mathtt{THTH}\,; \\ N\,\mathtt{THTHTTH} &= S_A\,\mathtt{THTTH} + S_A\,\mathtt{TTH} + S_B\,\mathtt{THTTH} + S_B\,. \end{aligned}$$

We obtain the victory probabilities by setting $H = T = \frac{1}{2}$, if we assume that a fair coin is being used; this reduces the two crucial equations to

$$\begin{aligned} N &= S_A \sum_{k=1}^{l} 2^k\,[A^{(k)} = A_{(k)}] + S_B \sum_{k=1}^{\min(l,m)} 2^k\,[B^{(k)} = A_{(k)}]\,; \\ N &= S_A \sum_{k=1}^{\min(l,m)} 2^k\,[A^{(k)} = B_{(k)}] + S_B \sum_{k=1}^{m} 2^k\,[B^{(k)} = B_{(k)}]\,. \end{aligned} \tag{8.80}$$

We can see what's going on if we generalize the $A{:}A$ operation of (8.76) to a function of two independent strings A and B:

$$A{:}B \;=\; \sum_{k=1}^{\min(l,m)} 2^{k-1}\,[A^{(k)} = B_{(k)}]\,. \tag{8.81}$$

Equations (8.80) now become simply

$$S_A(A{:}A) + S_B(B{:}A) \;=\; S_A(A{:}B) + S_B(B{:}B)\,;$$

the odds in Alice's favor are

$$\frac{S_A}{S_B} \;=\; \frac{B{:}B - B{:}A}{A{:}A - A{:}B}\,. \tag{8.82}$$

(This beautiful formula was discovered by John Horton Conway [137].)

For example, if $A = \mathtt{HTTHTHTH}$ and $B = \mathtt{THTHTTH}$ as above, we have $A{:}A = (10000001)_2 = 129$, $A{:}B = (0001010)_2 = 10$, $B{:}A = (0001001)_2 = 9$, and $B{:}B = (1000010)_2 = 66$; so the ratio S_A/S_B is $(66-9)/(129-10) = 57/119$. Alice will win this one only 57 times out of every 176, on the average.

Strange things can happen in Penney's game. For example, the pattern HHTH wins over the pattern HTHH with 3/2 odds, and HTHH wins over THHH with 7/5 odds. So HHTH ought to be much better than THHH. Yet THHH actually wins over HHTH, with 7/5 odds! The relation between patterns is not transitive. In fact, exercise 57 proves that if Alice chooses any pattern $\tau_1\tau_2\ldots\tau_l$ of length $l \geqslant 3$, Bill can always ensure better than even chances of winning if he chooses the pattern $\overline{\tau}_2\tau_1\tau_2\ldots\tau_{l-1}$, where $\overline{\tau}_2$ is the heads/tails opposite of τ_2.

Odd, odd.

8.5 HASHING

"Somehow the verb 'to hash' magically became standard terminology for key transformation during the mid-1960s, yet nobody was rash enough to use such an undignified word publicly until 1967." —D. E. Knuth [209]

Let's conclude this chapter by applying probability theory to computer programming. Several important algorithms for storing and retrieving information inside a computer are based on a technique called "hashing." The general problem is to maintain a set of records that each contain a "key" value, K, and some data $D(K)$ about that key; we want to be able to find $D(K)$ quickly when K is given. For example, each key might be the name of a student, and the associated data might be that student's homework grades.

In practice, computers don't have enough capacity to set aside one memory cell for every possible key; billions of keys are possible, but comparatively few keys are actually present in any one application. One solution to the problem is to maintain two tables $\texttt{KEY}[j]$ and $\texttt{DATA}[j]$ for $1 \leqslant j \leqslant N$, where N is the total number of records that can be accommodated; another variable n tells how many records are actually present. Then we can search for a given key K by going through the table sequentially in an obvious way:

S1 Set $j := 1$. (We've searched through all positions $< j$.)
S2 If $j > n$, stop. (The search was unsuccessful.)
S3 If $\texttt{KEY}[j] = K$, stop. (The search was successful.)
S4 Increase j by 1 and return to step S2. (We'll try again.)

After a successful search, the desired data entry $D(K)$ appears in $\texttt{DATA}[j]$. After an unsuccessful search, we can insert K and $D(K)$ into the table by setting

$$n := j, \quad \texttt{KEY}[n] := K, \quad \texttt{DATA}[n] := D(K),$$

assuming that the table was not already filled to capacity.

This method works, but it can be dreadfully slow; we need to repeat step S2 a total of $n + 1$ times whenever an unsuccessful search is made, and n can be quite large.

Hashing was invented to speed things up. The basic idea, in one of its popular forms, is to use m separate lists instead of one giant list. A "hash function" transforms every possible key K into a list number $h(K)$ between 1 and m. An auxiliary table $\texttt{FIRST}[i]$ for $1 \leqslant i \leqslant m$ points to the first record in list i; another auxiliary table $\texttt{NEXT}[j]$ for $1 \leqslant j \leqslant N$ points to the record following record j in its list. We assume that

$$\begin{aligned} \texttt{FIRST}[i] &= -1, && \text{if list } i \text{ is empty;} \\ \texttt{NEXT}[j] &= 0, && \text{if record } j \text{ is the last in its list.} \end{aligned}$$

As before, there's a variable n that tells how many records have been stored altogether.

For example, suppose the keys are names, and suppose that there are $m = 4$ lists based on the first letter of a name:

$$h(\text{name}) = \begin{cases} 1, & \text{for A–F;} \\ 2, & \text{for G–L;} \\ 3, & \text{for M–R;} \\ 4, & \text{for S–Z.} \end{cases}$$

We start with four empty lists and with $n = 0$. If, say, the first record has Nora as its key, we have $h(\text{Nora}) = 3$, so Nora becomes the key of the first item in list 3. If the next two names are Glenn and Jim, they both go into list 2. Now the tables in memory look like this:

$$\begin{array}{llll} \texttt{FIRST[1]} = -1, & \texttt{FIRST[2]} = 2, & \texttt{FIRST[3]} = 1, & \texttt{FIRST[4]} = -1. \\ \texttt{KEY[1]} = \text{Nora}, & \texttt{NEXT[1]} = 0; & & \\ \texttt{KEY[2]} = \text{Glenn}, & \texttt{NEXT[2]} = 3; & & \\ \texttt{KEY[3]} = \text{Jim}, & \texttt{NEXT[3]} = 0; & n = 3. & \end{array}$$

(The values of `DATA[1]`, `DATA[2]`, and `DATA[3]` are confidential and will not be shown.) After 18 records have been inserted, the lists might contain the names

Let's hear it for the Concrete Math students who sat in the front rows and lent their names to this experiment.

list 1	list 2	list 3	list 4
Dianne	Glenn	Nora	Scott
Ari	Jim	Mike	Tina
Brian	Jennifer	Michael	
Fran	Joan	Ray	
Doug	Jerry	Paula	
	Jean		

and these names would appear intermixed in the `KEY` array with `NEXT` entries to keep the lists effectively separate. If we now want to search for John, we have to scan through the six names in list 2 (which happens to be the longest list); but that's not nearly as bad as looking at all 18 names.

Here's a precise specification of the algorithm that searches for key K in accordance with this scheme:

H1 Set $i := h(K)$ and $j :=$ `FIRST[i]`.
H2 If $j \leqslant 0$, stop. (The search was unsuccessful.)
H3 If `KEY[j]` $= K$, stop. (The search was successful.)
H4 Set $i := j$, then set $j :=$ `NEXT[i]` and return to step H2. (We'll try again.)

For example, to search for Jennifer in the example given, step H1 would set $i := 2$ and $j := 2$; step H3 would find that Glenn $\neq$ Jennifer; step H4 would set $j := 3$; and step H3 would find Jim $\neq$ Jennifer. One more iteration of steps H4 and H3 would locate Jennifer in the table.

I bet their parents are glad about that.

After a successful search, the desired data $D(K)$ appears in `DATA[j]`, as in the previous algorithm. After an unsuccessful search, we can enter K and $D(K)$ in the table by doing the following operations:

$$
\begin{aligned}
&n := n+1;\\
&\textbf{if } j<0 \textbf{ then } \texttt{FIRST[i]} := n \textbf{ else } \texttt{NEXT[i]} := n;\\
&\texttt{KEY[n]} := K;\quad \texttt{DATA[n]} := D(K);\quad \texttt{NEXT[n]} := 0.
\end{aligned}
\tag{8.83}
$$

Now the table will once again be up to date.

We hope to get lists of roughly equal length, because this will make the task of searching about m times faster. The value of m is usually much greater than 4, so a factor of $1/m$ will be a significant improvement.

We don't know in advance what keys will be present, but it is generally possible to choose the hash function h so that we can consider $h(K)$ to be a random variable that is uniformly distributed between 1 and m, independent of the hash values of other keys that are present. In such cases computing the hash function is like rolling a die that has m faces. There's a chance that all the records will fall into the same list, just as there's a chance that a die will always turn up ⚅; but probability theory tells us that the lists will *almost always* be pretty evenly balanced.

Analysis of Hashing: Introduction.

"Algorithmic analysis" is a branch of computer science that derives quantitative information about the efficiency of computer methods. "Probabilistic analysis of an algorithm" is the study of an algorithm's running time, considered as a random variable that depends on assumed characteristics of the input data. Hashing is an especially good candidate for probabilistic analysis, because it is an extremely efficient method on the average, even though its worst case is too horrible to contemplate. (The worst case occurs when all keys have the same hash value.) Indeed, a computer programmer who uses hashing had better be a believer in probability theory.

Let P be the number of times step H3 is performed when the algorithm above is used to carry out a search. (Each execution of H3 is called a "probe" in the table.) If we know P, we know how often each step is performed, depending on whether the search is successful or unsuccessful:

Step	Unsuccessful search	Successful search
H1	1 time	1 time
H2	$P+1$ times	P times
H3	P times	P times
H4	P times	$P-1$ times

Thus the main quantity that governs the running time of the search procedure is the number of probes, P.

We can get a good mental picture of the algorithm by imagining that we are keeping an address book that is organized in a special way, with room for only one entry per page. On the cover of the book we note down the page number for the first entry in each of m lists; each name K determines the list $h(K)$ that it belongs to. Every page inside the book refers to the successor page in its list. The number of probes needed to find an address in such a book is the number of pages we must consult.

If n items have been inserted, their positions in the table depend only on their respective hash values, $\langle h_1, h_2, \ldots, h_n\rangle$. Each of the m^n possible sequences $\langle h_1, h_2, \ldots, h_n\rangle$ is considered to be equally likely, and P is a random variable depending on such a sequence.

Case 1: The key is not present.

Check under the doormat.

Let's consider first the behavior of P in an unsuccessful search, assuming that n records have previously been inserted into the hash table. In this case the relevant probability space consists of m^{n+1} elementary events

$$\omega \;=\; (h_1, h_2, \ldots, h_n, h_{n+1})$$

where h_j is the hash value of the jth key inserted, and where h_{n+1} is the hash value of the key for which the search is unsuccessful. We assume that the hash function h has been chosen properly so that $\Pr(\omega) = 1/m^{n+1}$ for every such ω.

For example, if $m = n = 2$, there are eight equally likely possibilities:

h_1	h_2	h_3:	P
1	1	1 :	2
1	1	2 :	0
1	2	1 :	1
1	2	2 :	1
2	1	1 :	1
2	1	2 :	1
2	2	1 :	0
2	2	2 :	2

If $h_1 = h_2 = h_3$ we make two unsuccessful probes before concluding that the new key K is not present; if $h_1 = h_2 \neq h_3$ we make none; and so on. This list of all possibilities shows that P has a probability distribution given by the pgf $(\frac{2}{8} + \frac{4}{8}z + \frac{2}{8}z^2) = (\frac{1}{2} + \frac{1}{2}z)^2$, when $m = n = 2$.

An unsuccessful search makes one probe for every item in list number h_{n+1}, so we have the general formula

$$P \;=\; [h_1 = h_{n+1}] + [h_2 = h_{n+1}] + \cdots + [h_n = h_{n+1}]\,. \tag{8.84}$$

The probability that $h_j = h_{n+1}$ is $1/m$, for $1 \leqslant j \leqslant n$; so it follows that

$$EP = E[h_1 = h_{n+1}] + E[h_2 = h_{n+1}] + \cdots + E[h_n = h_{n+1}] = \frac{n}{m}.$$

Maybe we should do that more slowly: Let X_j be the random variable

$$X_j = X_j(\omega) = [h_j = h_{n+1}].$$

Then $P = X_1 + \cdots + X_n$, and $EX_j = 1/m$ for all $j \leqslant n$; hence

$$EP = EX_1 + \cdots + EX_n = n/m.$$

Good: As we had hoped, the average number of probes is $1/m$ times what it was without hashing. Furthermore the random variables X_j are independent, and they each have the same probability generating function

$$X_j(z) = \frac{m-1+z}{m};$$

therefore the pgf for the total number of probes in an unsuccessful search is

$$P(z) = X_1(z)\ldots X_n(z) = \left(\frac{m-1+z}{m}\right)^n. \tag{8.85}$$

This is a binomial distribution, with $p = 1/m$ and $q = (m-1)/m$; in other words, the number of probes in an unsuccessful search behaves just like the number of heads when we toss a biased coin whose probability of heads is $1/m$ on each toss. Equation (8.61) tells us that the variance of P is therefore

$$npq = \frac{n(m-1)}{m^2}.$$

When m is large, the variance of P is approximately n/m, so the standard deviation is approximately $\sqrt{n/m}$.

Case 2: The key is present.

Now let's look at successful searches. In this case the appropriate probability space is a bit more complicated, depending on our application: We will let Ω be the set of all elementary events

$$\omega = (h_1, \ldots, h_n; k), \tag{8.86}$$

where h_j is the hash value for the jth key as before, and where k is the index of the key being sought (the key whose hash value is h_k). Thus we have $1 \leqslant h_j \leqslant m$ for $1 \leqslant j \leqslant n$, and $1 \leqslant k \leqslant n$; there are $m^n \cdot n$ elementary events ω in all.

Let s_j be the probability that we are searching for the jth key that was inserted into the table. Then

$$\Pr(\omega) \;=\; s_k/m^n \tag{8.87}$$

if ω is the event (8.86). (Some applications search most often for the items that were inserted first, or for the items that were inserted last, so we will not assume that each $s_j = 1/n$.) Notice that $\sum_{\omega\in\Omega}\Pr(\omega) = \sum_{k=1}^{n} s_k = 1$, hence (8.87) defines a legal probability distribution.

The number of probes P in a successful search is p if key K was the pth key to be inserted into its list. Therefore

$$P(h_1,\ldots,h_n;k) \;=\; [h_1{=}h_k] + [h_2{=}h_k] + \cdots + [h_k{=}h_k]\,; \tag{8.88}$$

or, if we let X_j be the random variable $[h_j{=}h_k]$, we have

$$P \;=\; X_1 + X_2 + \cdots + X_k\,. \tag{8.89}$$

Suppose, for example, that we have $m = 10$ and $n = 16$, and that the hash values have the following "random" pattern:

Where have I seen that pattern before?

$$\begin{aligned}(h_1,\ldots,h_{16}) &= 3\;1\;4\;1\;5\;9\;2\;6\;5\;3\;5\;8\;9\;7\;9\;3\;;\\ (P_1,\ldots,P_{16}) &= 1\;1\;1\;2\;1\;1\;1\;1\;2\;2\;3\;1\;2\;1\;3\;3\;.\end{aligned}$$

The number of probes P_j needed to find the jth key is shown below h_j.

Equation (8.89) represents P as a sum of random variables, but we can't simply calculate EP as $EX_1+\cdots+EX_k$ because the quantity k itself is a random variable. What is the probability generating function for P? To answer this question we should digress a moment to talk about *conditional probability*.

Equation (8.43) was also a momentary digression.

If A and B are events in a probability space, we say that the conditional probability of A, given B, is

$$\Pr(\omega\in A \mid \omega\in B) \;=\; \frac{\Pr(\omega\in A\cap B)}{\Pr(\omega\in B)}\,. \tag{8.90}$$

For example, if X and Y are random variables, the conditional probability of the event $X = x$, given that $Y = y$, is

$$\Pr(X{=}x \mid Y{=}y) \;=\; \frac{\Pr(X{=}x \text{ and } Y{=}y)}{\Pr(Y{=}y)}\,. \tag{8.91}$$

For any fixed y in the range of Y, the sum of these conditional probabilities over all x in the range of X is $\Pr(Y{=}y)/\Pr(Y{=}y) = 1$; therefore (8.91) defines a probability distribution, and we can define a new random variable '$X|y$' such that $\Pr\big((X|y){=}x\big) = \Pr(X{=}x \mid Y{=}y)$.

If X and Y are independent, the random variable $X|y$ will be essentially the same as X, regardless of the value of y, because $\Pr(X=x \mid Y=y)$ is equal to $\Pr(X=x)$ by (8.5); that's what independence means. But if X and Y are dependent, the random variables $X|y$ and $X|y'$ need not resemble each other in any way when $y \neq y'$.

If X takes only nonnegative integer values, we can decompose its pgf into a sum of conditional pgf's with respect to any other random variable Y:

$$G_X(z) \;=\; \sum_{y \in Y(\Omega)} \Pr(Y=y)\, G_{X|y}(z)\,. \tag{8.92}$$

This holds because the coefficient of z^x on the left side is $\Pr(X=x)$, for all $x \in X(\Omega)$, and on the right it is

$$\begin{aligned}\sum_{y \in Y(\Omega)} \Pr(Y=y)\Pr(X=x \mid Y=y) \;&=\; \sum_{y \in Y(\Omega)} \Pr(X=x \text{ and } Y=y)\\ &=\; \Pr(X=x)\,.\end{aligned}$$

For example, if X is the product of the spots on two fair dice and if Y is the sum of the spots, the pgf for $X|6$ is

$$G_{X|6}(z) \;=\; \tfrac{2}{5}z^5 + \tfrac{2}{5}z^8 + \tfrac{1}{5}z^9$$

because the conditional probabilities for $Y = 6$ consist of five equally probable events {⚀⚄, ⚁⚃, ⚂⚂, ⚃⚁, ⚄⚀}. Equation (8.92) in this case reduces to

$$\begin{aligned}G_X(z) \;=\; &\tfrac{1}{36}G_{X|2}(z) + \tfrac{2}{36}G_{X|3}(z) + \tfrac{3}{36}G_{X|4}(z) + \tfrac{4}{36}G_{X|5}(z)\\ &\tfrac{5}{36}G_{X|6}(z) + \tfrac{6}{36}G_{X|7}(z) + \tfrac{5}{36}G_{X|8}(z) + \tfrac{4}{36}G_{X|9}(z)\\ &\tfrac{3}{36}G_{X|10}(z) + \tfrac{2}{36}G_{X|11}(z) + \tfrac{1}{36}G_{X|12}(z)\,,\end{aligned}$$

a formula that is obvious once you understand it. (End of digression.)

Oh, now I understand what mathematicians mean when they say something is "obvious," "clear," or "trivial."

In the case of hashing, (8.92) tells us how to write down the pgf for probes in a successful search, if we let $X = P$ and $Y = K$. For any fixed k between 1 and n, the random variable $P|k$ is defined as a sum of independent random variables $X_1 + \cdots + X_k$; this is (8.89). So it has the pgf

$$G_{P|k}(z) \;=\; \Big(\frac{m-1+z}{m}\Big)^{k-1} z\,.$$

Therefore the pgf for P itself is clearly

"By clearly, I mean a good freshman should be able to do it, although it's not completely trivial."
—Paul Erdős [94].

$$\begin{aligned} G_P(z) &= \sum_{k=1}^{n} s_k G_{P|k}(z) \\ &= \sum_{k=1}^{n} s_k \Bigl(\frac{m-1+z}{m}\Bigr)^{k-1} z \\ &= z\,S\Bigl(\frac{m-1+z}{m}\Bigr), \end{aligned} \tag{8.93}$$

where

$$S(z) = s_1 + s_2 z + s_3 z^2 + \cdots + s_n z^{n-1} \tag{8.94}$$

is the pgf for the search probabilities s_k (divided by z for convenience).

Good. We have a probability generating function for P; we can now find the mean and variance by differentiation. It's somewhat easier to remove the z factor first, as we've done before, thus finding the mean and variance of $P-1$ instead:

$$\begin{aligned} F(z) &= G_P(z)/z = S\Bigl(\frac{m-1+z}{m}\Bigr); \\ F'(z) &= \frac{1}{m} S'\Bigl(\frac{m-1+z}{m}\Bigr); \\ F''(z) &= \frac{1}{m^2} S''\Bigl(\frac{m-1+z}{m}\Bigr). \end{aligned}$$

Therefore

$$EP = 1 + \mathrm{Mean}(F) = 1 + F'(1) = 1 + m^{-1}\,\mathrm{Mean}(S); \tag{8.95}$$

$$\begin{aligned} VP = \mathrm{Var}(F) &= F''(1) + F'(1) - F'(1)^2 \\ &= m^{-2} S''(1) + m^{-1} S'(1) - m^{-2} S'(1)^2 \\ &= m^{-2}\,\mathrm{Var}(S) + (m^{-1} - m^{-2})\,\mathrm{Mean}(S). \end{aligned} \tag{8.96}$$

These are general formulas expressing the mean and variance of the number of probes P in terms of the mean and variance of the assumed search distribution S.

For example, suppose we have $s_k = 1/n$ for $1 \leqslant k \leqslant n$. This means we are doing a purely "random" successful search, with all keys in the table equally likely. Then $S(z)$ is the uniform probability distribution $U_n(z)$ in

(8.32), and we have $\mathrm{Mean}(S) = (n-1)/2$, $\mathrm{Var}(S) = (n^2-1)/12$. Hence

$$\mathrm{EP} = \frac{n-1}{2m} + 1\,; \tag{8.97}$$

$$\mathrm{VP} = \frac{n^2-1}{12m^2} + \frac{(m-1)(n-1)}{2m^2} = \frac{(n-1)(6m+n-5)}{12m^2}\,. \tag{8.98}$$

Once again we have gained the desired speedup factor of $1/m$. If $m = n/\ln n$ and $n \to \infty$, the average number of probes per successful search in this case is about $\frac{1}{2}\ln n$, and the standard deviation is asymptotically $(\ln n)/\sqrt{12}$.

On the other hand, we might suppose that $s_k = (kH_n)^{-1}$ for $1 \leqslant k \leqslant n$; this distribution is called "Zipf's law." Then $\mathrm{Mean}(G) = n/H_n$ and $\mathrm{Var}(G) = \frac{1}{2}n(n+1)/H_n - n^2/H_n^2$. The average number of probes for $m = n/\ln n$ as $n \to \infty$ is approximately 2, with standard deviation asymptotic to $\sqrt{\ln n}/\sqrt{2}$.

In both cases the analysis allows the cautious souls among us, who fear the worst case, to rest easily: Chebyshev's inequality tells us that the lists will be nice and short, except in extremely rare cases.

Case 2, continued: Variants of the variance.

We have just computed the variance of the number of probes in a successful search, by considering P to be a random variable over a probability space with $m^n \cdot n$ elements $(h_1, \ldots, h_n; k)$. But we could have adopted another point of view: Each pattern $(h_1, \ldots, h_n)$ of hash values defines a random variable $P|(h_1, \ldots, h_n)$, representing the probes we make in a successful search of a particular hash table on n given keys. The average value of $P|(h_1, \ldots, h_n)$,

OK, gang, time to put on your skim suits again.
—Friendly TA

$$A(h_1, \ldots, h_n) = \sum_{p=1}^{n} p \cdot \Pr\bigl(\bigl(P|(h_1, \ldots, h_n)\bigr) = p\bigr)\,, \tag{8.99}$$

can be said to represent the running time of a successful search. This quantity $A(h_1, \ldots, h_n)$ is a random variable that depends only on $(h_1, \ldots, h_n)$, not on the final component k. We can write it in the form

$$A(h_1, \ldots, h_n) = \sum_{k=1}^{n} s_k\, P(h_1, \ldots, h_n; k)\,,$$

where $P(h_1, \ldots, h_n; k)$ is defined in (8.88), since $P|(h_1, \ldots, h_n) = p$ with probability

$$\frac{\sum_{k=1}^{n} \Pr\bigl(P(h_1, \ldots, h_n; k) = p\bigr)}{\sum_{k=1}^{n} \Pr(h_1, \ldots, h_n; k)} = \frac{\sum_{k=1}^{n} m^{-n} s_k \bigl[P(h_1, \ldots, h_n; k) = p\bigr]}{\sum_{k=1}^{n} m^{-n} s_k}$$

$$= \sum_{k=1}^{n} s_k \bigl[P(h_1, \ldots, h_n; k) = p\bigr]\,.$$

The mean value of $A(h_1,\ldots,h_n)$, obtained by summing over all m^n possibilities $(h_1,\ldots,h_n)$ and dividing by m^n, will be the same as the mean value we obtained before in (8.95). But the *variance* of $A(h_1,\ldots,h_n)$ is something different; this is a variance of m^n averages, not a variance of $m^n\cdot n$ probe counts. For example, if $m=1$ (so that there is only one list), the "average" value $A(h_1,\ldots,h_n)=A(1,\ldots,1)$ is actually constant, so its variance VA is zero; but the number of probes in a successful search is not constant, so the variance VP is nonzero.

But the VP is nonzero only in an election year.

We can illustrate this difference between variances by carrying out the calculations for general m and n in the simplest case, when $s_k = 1/n$ for $1 \leqslant k \leqslant n$. In other words, we will assume temporarily that there is a uniform distribution of search keys. Any given sequence of hash values $(h_1,\ldots,h_n)$ defines m lists that contain respectively $(n_1,n_2,\ldots,n_m)$ entries for some numbers n_j, where

$$n_1+n_2+\cdots+n_m \;=\; n\,.$$

A successful search in which each of the n keys in the table is equally likely will have an average running time of

$$\begin{aligned} A(h_1,\ldots,h_n) \;&=\; \frac{(1+\cdots+n_1)+(1+\cdots+n_2)+\cdots+(1+\cdots+n_m)}{n} \\ &=\; \frac{n_1(n_1+1)+n_2(n_2+1)+\cdots+n_m(n_m+1)}{2n} \\ &=\; \frac{n_1^2+n_2^2+\cdots+n_m^2+n}{2n} \end{aligned}$$

probes. Our goal is to calculate the variance of this quantity $A(h_1,\ldots,h_n)$, over the probability space consisting of all m^n sequences $(h_1,\ldots,h_n)$.

The calculations will be simpler, it turns out, if we compute the variance of a slightly different quantity,

$$B(h_1,\ldots,h_n) \;=\; \binom{n_1}{2}+\binom{n_2}{2}+\cdots+\binom{n_m}{2}\,.$$

We have

$$A(h_1,\ldots,h_n) \;=\; 1+B(h_1,\ldots,h_n)/n\,,$$

hence the mean and variance of A satisfy

$$EA \;=\; 1+\frac{EB}{n}\,; \qquad VA \;=\; \frac{VB}{n^2}\,. \tag{8.100}$$

The probability that the list sizes will be $n_1, n_2, \ldots, n_m$ is the multinomial coefficient

$$\binom{n}{n_1, n_2, \ldots, n_m} = \frac{n!}{n_1!\, n_2! \ldots n_m!}$$

divided by m^n; hence the pgf for $B(h_1, \ldots, h_n)$ is

$$B_n(z) = \sum_{\substack{n_1, n_2, \ldots, n_m \geqslant 0 \\ n_1 + n_2 + \cdots + n_m = n}} \binom{n}{n_1, n_2, \ldots, n_m} z^{\binom{n_1}{2} + \binom{n_2}{2} + \cdots + \binom{n_m}{2}} m^{-n} .$$

This sum looks a bit scary to inexperienced eyes, but our experiences in Chapter 7 have taught us to recognize it as an m-fold convolution. Indeed, if we consider the exponential super-generating function

$$G(w, z) = \sum_{n \geqslant 0} B_n(z) \frac{m^n w^n}{n!} ,$$

we can readily verify that $G(w, z)$ is simply an mth power:

$$G(w, z) = \left(\sum_{k \geqslant 0} z^{\binom{k}{2}} \frac{w^k}{k!} \right)^m .$$

As a check, we can try setting $z = 1$; we get $G(w, 1) = (e^w)^m$, so the coefficient of $m^n w^n / n!$ is $B_n(1) = 1$.

If we knew the values of $B_n'(1)$ and $B_n''(1)$, we would be able to calculate $\mathrm{Var}(B_n)$. So we take partial derivatives of $G(w, z)$ with respect to z:

$$\begin{aligned}
\frac{\partial}{\partial z} G(w, z) &= \sum_{n \geqslant 0} B_n'(z) \frac{m^n w^n}{n!} \\
&= m \left(\sum_{k \geqslant 0} z^{\binom{k}{2}} \frac{w^k}{k!} \right)^{m-1} \sum_{k \geqslant 0} \binom{k}{2} z^{\binom{k}{2} - 1} \frac{w^k}{k!} ; \\
\frac{\partial^2}{\partial z^2} G(w, z) &= \sum_{n \geqslant 0} B_n''(z) \frac{m^n w^n}{n!} \\
&= m(m-1) \left(\sum_{k \geqslant 0} z^{\binom{k}{2}} \frac{w^k}{k!} \right)^{m-2} \left(\sum_{k \geqslant 0} \binom{k}{2} z^{\binom{k}{2} - 1} \frac{w^k}{k!} \right)^2 \\
&\quad + m \left(\sum_{k \geqslant 0} z^{\binom{k}{2}} \frac{w^k}{k!} \right)^{m-1} \sum_{k \geqslant 0} \binom{k}{2} \left(\binom{k}{2} - 1 \right) z^{\binom{k}{2} - 2} \frac{w^k}{k!} .
\end{aligned}$$

Complicated, yes; but everything simplifies greatly when we set $z = 1$. For example, we have

$$\begin{aligned}
\sum_{n\geqslant 0} B'_n(1)\frac{m^n w^n}{n!} &= me^{(m-1)w}\sum_{k\geqslant 2}\frac{w^k}{2(k-2)!} \\
&= me^{(m-1)w}\sum_{k\geqslant 0}\frac{w^{k+2}}{2k!} \\
&= \frac{mw^2e^{(m-1)w}}{2}e^w = \sum_{n\geqslant 0}\frac{(mw)^{n+2}}{2m\,n!} = \sum_{n\geqslant 0}\frac{n(n-1)m^n w^n}{2m\,n!}\,,
\end{aligned}$$

and it follows that

$$B'_n(1) = \binom{n}{2}\frac{1}{m}\,. \tag{8.101}$$

The expression for EA in (8.100) now gives $EA = 1+(n-1)/2m$, in agreement with (8.97).

The formula for $B''_n(1)$ involves the similar sum

$$\begin{aligned}
\sum_{k\geqslant 0}\binom{k}{2}\left(\binom{k}{2}-1\right)\frac{w^k}{k!} &= \frac{1}{4}\sum_{k\geqslant 0}\frac{(k+1)k(k-1)(k-2)w^k}{k!} \\
&= \frac{1}{4}\sum_{k\geqslant 3}\frac{(k+1)w^k}{(k-3)!} = \frac{1}{4}\sum_{k\geqslant 0}\frac{(k+4)w^{k+3}}{k!} = \left(\tfrac{1}{4}w^4+w^3\right)e^w\,;
\end{aligned}$$

hence we find that

$$\begin{aligned}
\sum_{n\geqslant 0} B''_n(1)\frac{m^n w^n}{n!} &= m(m-1)e^{w(m-2)}\left(\tfrac{1}{2}w^2e^w\right)^2+me^{w(m-1)}\left(\tfrac{1}{4}w^4+w^3\right)e^w \\
&= me^{wm}\left(\tfrac{1}{4}mw^4+w^3\right)\,; \\
B''_n(1) &= \binom{n}{2}\left(\binom{n}{2}-1\right)\frac{1}{m^2}\,.
\end{aligned} \tag{8.102}$$

Now we can put all the pieces together and evaluate the desired variance VA. Massive cancellation occurs, and the result is surprisingly simple:

$$\begin{aligned}
VA = \frac{VB}{n^2} &= \frac{B''_n(1)+B'_n(1)-B'_n(1)^2}{n^2} \\
&= \frac{n(n-1)}{m^2n^2}\left(\frac{(n+1)(n-2)}{4}+\frac{m}{2}-\frac{n(n-1)}{4}\right) \\
&= \frac{(m-1)(n-1)}{2m^2n}\,.
\end{aligned} \tag{8.103}$$

When such "coincidences" occur, we suspect that there's a mathematical reason; there might be another way to attack the problem, explaining why the answer has such a simple form. And indeed, there is another approach (in exercise 61), which shows that the variance of the average successful search has the general form

$$VA = \frac{m-1}{m^2} \sum_{k=1}^{n} s_k^2 (k-1) \tag{8.104}$$

when s_k is the probability that the kth-inserted element is being sought. Equation (8.103) is the special case $s_k = 1/n$ for $1 \leqslant k \leqslant n$.

Besides the variance of the average, we might also consider the average of the variance. In other words, each sequence $(h_1, \ldots, h_n)$ that defines a hash table also defines a probability distribution for successful searching, and the variance of this probability distribution tells how spread out the number of probes will be in different successful searches. For example, let's go back to the case where we inserted $n = 16$ things into $m = 10$ lists:

Where have I seen that pattern before?

Where have I seen that graffito before?

$I\eta\nu P_\pi$.

$$\begin{aligned}(h_1, \ldots, h_{16}) &= 3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3\ 5\ 8\ 9\ 7\ 9\ 3\\ (P_1, \ldots, P_{16}) &= 1\ 1\ 1\ 2\ 1\ 1\ 1\ 1\ 2\ 2\ 3\ 1\ 2\ 1\ 3\ 3\end{aligned}$$

A successful search in the resulting hash table has the pgf

$$\begin{aligned}G(3,1,4,1,\ldots,3) &= \sum_{k=1}^{16} s_k z^{P(3,1,4,1,\ldots,3;k)}\\ &= s_1 z + s_2 z + s_3 z + s_4 z^2 + \cdots + s_{16} z^3 .\end{aligned}$$

We have just considered the average number of probes in a successful search of this table, namely $A(3,1,4,1,\ldots,3) = \mathrm{Mean}\big(G(3,1,4,1,\ldots,3)\big)$. We can also consider the variance,

$$\begin{aligned}&s_1 \cdot 1^2 + s_2 \cdot 1^2 + s_3 \cdot 1^2 + s_4 \cdot 2^2 + \cdots + s_{16} \cdot 3^2\\ &\quad - (s_1 \cdot 1 + s_2 \cdot 1 + s_3 \cdot 1 + s_4 \cdot 2 + \cdots + s_{16} \cdot 3)^2 .\end{aligned}$$

This variance is a random variable, depending on $(h_1, \ldots, h_n)$, so it is natural to consider its average value.

In other words, there are three natural kinds of variance that we may wish to know, in order to understand the behavior of a successful search: The *overall variance* of the number of probes, taken over all $(h_1, \ldots, h_n)$ and k; the *variance of the average* number of probes, where the average is taken over all k and the variance is then taken over all $(h_1, \ldots, h_n)$; and the *average of the variance* of the number of the probes, where the variance is taken over

all k and the average is then taken over all $(h_1, \ldots, h_n)$. In symbols, the overall variance is

$$\begin{aligned} VP &= \sum_{1 \leqslant h_1, \ldots, h_n \leqslant m} \sum_{k=1}^{n} \frac{s_k}{m^n} P(h_1, \ldots, h_n; k)^2 \\ &\quad - \Bigg(\sum_{1 \leqslant h_1, \ldots, h_n \leqslant m} \sum_{k=1}^{n} \frac{s_k}{m^n} P(h_1, \ldots, h_n; k) \Bigg)^2 ; \end{aligned}$$

the variance of the average is

$$\begin{aligned} VA &= \sum_{1 \leqslant h_1, \ldots, h_n \leqslant m} \frac{1}{m^n} \Bigg(\sum_{k=1}^{n} s_k P(h_1, \ldots, h_n; k) \Bigg)^2 \\ &\quad - \Bigg(\sum_{1 \leqslant h_1, \ldots, h_n \leqslant m} \frac{1}{m^n} \sum_{k=1}^{n} s_k P(h_1, \ldots, h_n; k) \Bigg)^2 ; \end{aligned}$$

and the average of the variance is

$$\begin{aligned} AV &= \sum_{1 \leqslant h_1, \ldots, h_n \leqslant m} \frac{1}{m^n} \Bigg(\sum_{k=1}^{n} s_k P(h_1, \ldots, h_n; k)^2 \\ &\qquad\qquad - \Bigg(\sum_{k=1}^{n} s_k P(h_1, \ldots, h_n; k) \Bigg)^2 \Bigg) . \end{aligned}$$

It turns out that these three quantities are interrelated in a simple way:

$$VP = VA + AV . \tag{8.105}$$

In fact, conditional probability distributions always satisfy the identity

$$VX = V\big(E(X|Y)\big) + E\big(V(X|Y)\big) \tag{8.106}$$

if X and Y are random variables in any probability space and if X takes real values. (This identity is proved in exercise 22.) Equation (8.105) is the special case where X is the number of probes in a successful search and Y is the sequence of hash values $(h_1, \ldots, h_n)$.

The general equation (8.106) needs to be understood carefully, because the notation tends to conceal the different random variables and probability spaces in which expectations and variances are being calculated. For each y in the range of Y, we have defined the random variable $X|y$ in (8.91), and this random variable has an expected value $E(X|y)$ depending on y. Now $E(X|Y)$ denotes the random variable whose values are $E(X|y)$ as y ranges over all

(Now is a good time to do warmup exercise 6.)

possible values of Y, and $V\big(E(X|Y)\big)$ is the variance of this random variable with respect to the probability distribution of Y. Similarly, $E\big(V(X|Y)\big)$ is the average of the random variables $V(X|y)$ as y varies. On the left of (8.106) is VX, the unconditional variance of X. Since variances are nonnegative, we always have

$$VX \geqslant V\big(E(X|Y)\big) \quad \text{and} \quad VX \geqslant E\big(V(X|Y)\big)\,. \tag{8.107}$$

Case 1, again: Unsuccessful search revisited.

Let's bring our microscopic examination of hashing to a close by doing one more calculation typical of algorithmic analysis. This time we'll look more closely at the *total running time* associated with an unsuccessful search, assuming that the computer will insert the previously unknown key into its memory.

P is still the number of probes.

The insertion process in (8.83) has two cases, depending on whether j is negative or zero. We have $j < 0$ if and only if $P = 0$, since a negative value comes from the FIRST entry of an empty list. Thus, if the list was previously empty, we have $P = 0$ and we must set $\texttt{FIRST}[h_{n+1}] := n + 1$. (The new record will be inserted into position $n + 1$.) Otherwise we have $P > 0$ and we must set a LINK entry to $n + 1$. These two cases may take different amounts of time; therefore the total running time for an unsuccessful search has the form

$$T \;=\; \alpha + \beta P + \delta[P = 0]\,, \tag{8.108}$$

where α, β, and δ are constants that depend on the computer being used and on the way in which hashing is encoded in that machine's internal language. It would be nice to know the mean and variance of T, since such information is more relevant in practice than the mean and variance of P.

So far we have used probability generating functions only in connection with random variables that take nonnegative integer values. But it turns out that we can deal in essentially the same way with

$$G_X(z) \;=\; \sum_{\omega\in\Omega} \Pr(\omega) z^{X(\omega)}$$

when X is any real-valued random variable, because the essential characteristics of X depend only on the behavior of G_X near $z = 1$, where powers of z are well defined. For example, the running time (8.108) of an unsuccessful search is a random variable, defined on the probability space of equally likely hash values $(h_1, \ldots, h_n, h_{n+1})$ with $1 \leqslant h_j \leqslant m$; we can consider the series

$$G_T(z) \;=\; \frac{1}{m^{n+1}} \sum_{h_1=1}^{m} \cdots \sum_{h_n=1}^{m} \sum_{h_{n+1}=1}^{m} z^{\alpha+\beta P(h_1,\ldots,h_{n+1})+\delta[P(h_1,\ldots,h_{n+1})=0]}$$

to be a pgf even when α, β, and δ are not integers. (In fact, the parameters α, β, δ are physical quantities that have dimensions of time; they aren't even pure numbers! Yet we can use them in the exponent of z.) We can still calculate the mean and variance of T, by evaluating $G_T'(1)$ and $G_T''(1)$ and combining these values in the usual way.

The generating function for P instead of T is

$$P(z) = \Bigl(\frac{m-1+z}{m}\Bigr)^n = \sum_{p\geqslant 0} \Pr(P=p)z^p\,.$$

Therefore we have

$$\begin{aligned} G_T(z) &= \sum_{p\geqslant 0} \Pr(P=p)z^{\alpha+\beta p+\delta[p=0]} \\ &= z^\alpha\Bigl((z^\delta-1)\Pr(P=0) + \sum_{p\geqslant 0}\Pr(P=p)z^{\beta p}\Bigr) \\ &= z^\alpha\biggl((z^\delta-1)\Bigl(\frac{m-1}{m}\Bigr)^n + \Bigl(\frac{m-1+z^\beta}{m}\Bigr)^n\biggr)\,. \end{aligned}$$

The determination of $\mathrm{Mean}(G_T)$ and $\mathrm{Var}(G_T)$ is now routine:

$$\mathrm{Mean}(G_T) = G_T'(1) = \alpha + \beta\frac{n}{m} + \delta\Bigl(\frac{m-1}{m}\Bigr)^n; \tag{8.109}$$

$$\begin{aligned} G_T''(1) &= \alpha(\alpha-1) + 2\alpha\beta\frac{n}{m} + \beta(\beta-1)\frac{n}{m} + \beta^2\frac{n(n-1)}{m^2} \\ &\qquad + 2\alpha\delta\Bigl(\frac{m-1}{m}\Bigr)^n + \delta(\delta-1)\Bigl(\frac{m-1}{m}\Bigr)^n; \\ \mathrm{Var}(G_T) &= G_T''(1) + G_T'(1) - G_T'(1)^2 \\ &= \beta^2\frac{n(m-1)}{m^2} - 2\beta\delta\Bigl(\frac{m-1}{m}\Bigr)^n\frac{n}{m} \\ &\qquad + \delta^2\biggl(\Bigl(\frac{m-1}{m}\Bigr)^n - \Bigl(\frac{m-1}{m}\Bigr)^{2n}\biggr)\,. \end{aligned} \tag{8.110}$$

In Chapter 9 we will learn how to estimate quantities like this when m and n are large. If, for example, $m = n$ and $n \to \infty$, the techniques of Chapter 9 will show that the mean and variance of T are respectively $\alpha + \beta + \delta e^{-1} + O(n^{-1})$ and $\beta^2 - 2\beta\delta e^{-1} + \delta^2(e^{-1} - e^{-2}) + O(n^{-1})$. If $m = n/\ln n$ and $n \to \infty$ the corresponding results are

$$\begin{aligned} \mathrm{Mean}(G_T) &= \beta\ln n + \alpha + \delta/n + O\bigl((\log n)^2/n^2\bigr)\,; \\ \mathrm{Var}(G_T) &= \beta^2\ln n - \bigl((\beta\ln n)^2 + 2\beta\delta\ln n - \delta^2\bigr)/n + O\bigl((\log n)^3/n^2\bigr)\,. \end{aligned}$$

Exercises

Warmups

1 What's the probability of doubles in the probability distribution $\Pr_{01}$ of (8.3), when one die is fair and the other is loaded? What's the probability that $S = 7$ is rolled?

2 What's the probability that the top and bottom cards of a randomly shuffled deck are both aces? (All 52! permutations have probability 1/52!.)

Why only ten numbers?

The other students either weren't empiricists or they were just too flipped out.

3 Stanford's Concrete Math students were asked in 1979 to flip coins until they got heads twice in succession, and to report the number of flips required. The answers were

$$3, 2, 3, 5, 10, 2, 6, 6, 9, 2\,.$$

Princeton's Concrete Math students were asked in 1987 to do a similar thing, with the following results:

$$10, 2, 10, 7, 5, 2, 10, 6, 10, 2\,.$$

Estimate the mean and variance, based on (a) the Stanford sample; (b) the Princeton sample.

4 Let $H(z) = F(z)/G(z)$, where $F(1) = G(1) = 1$. Prove that

$$\begin{aligned} \text{Mean}(H) &= \text{Mean}(F) - \text{Mean}(G)\,, \\ \text{Var}(H) &= \text{Var}(F) - \text{Var}(G)\,, \end{aligned}$$

in analogy with (8.38) and (8.39), if the indicated derivatives exist at $z = 1$.

5 Suppose Alice and Bill play the game (8.78) with a biased coin that comes up heads with probability p. Is there a value of p for which the game becomes fair?

6 What does the conditional variance law (8.106) reduce to, when X and Y are independent random variables?

Basics

7 Show that if two dice are loaded with the same probability distribution, the probability of doubles is always at least $\frac{1}{6}$.

8 Let A and B be events such that $A \cup B = \Omega$. Prove that

$$\Pr(\omega \in A \cap B) \;=\; \Pr(\omega \in A)\Pr(\omega \in B) - \Pr(\omega \notin A)\Pr(\omega \notin B)\,.$$

9 Prove or disprove: If X and Y are independent random variables, then so are $F(X)$ and $G(Y)$, when F and G are any functions.

10 What's the maximum number of elements that can be medians of a random variable X, according to definition (8.7)?

11 Construct a random variable that has finite mean and infinite variance.

12 **a** If $P(z)$ is the pgf for the random variable X, prove that

$$\begin{aligned} \Pr(X \leqslant r) &\leqslant x^{-r}P(x) \qquad \text{for } 0 < x \leqslant 1; \\ \Pr(X \geqslant r) &\leqslant x^{-r}P(x) \qquad \text{for } x \geqslant 1. \end{aligned}$$

(These important relations are called the *tail inequalities*.)

b In the special case $P(z) = (1+z)^n/2^n$, use the first tail inequality to prove that $\sum_{k \leqslant \alpha n} \binom{n}{k} \leqslant 1/\alpha^{\alpha n}(1-\alpha)^{(1-\alpha)n}$ when $0 < \alpha < \frac{1}{2}$.

13 If $X_1, \ldots, X_{2n}$ are independent random variables with the same distribution, and if α is any real number whatsoever, prove that

$$\Pr\left(\left|\frac{X_1 + \cdots + X_{2n}}{2n} - \alpha\right| \leqslant \left|\frac{X_1 + \cdots + X_n}{n} - \alpha\right|\right) \geqslant \frac{1}{2}.$$

14 Let $F(z)$ and $G(z)$ be probability generating functions, and let

$$H(z) = p\,F(z) + q\,G(z)$$

where $p + q = 1$. (This is called a *mixture* of F and G; it corresponds to flipping a coin and choosing probability distribution F or G depending on whether the coin comes up heads or tails.) Find the mean and variance of H in terms of p, q, and the mean and variance of F and G.

15 If $F(z)$ and $G(z)$ are probability generating functions, we can define another pgf $H(z)$ by "composition":

$$H(z) = F\big(G(z)\big).$$

Express $\mathrm{Mean}(H)$ and $\mathrm{Var}(H)$ in terms of $\mathrm{Mean}(F)$, $\mathrm{Var}(F)$, $\mathrm{Mean}(G)$, and $\mathrm{Var}(G)$. (Equation (8.93) is a special case.)

16 Find a closed form for the super generating function $\sum_{n \geqslant 0} F_n(z) w^n$, when $F_n(z)$ is the football-fixation generating function defined in (8.53).

17 Let $X_{n,p}$ and $Y_{n,p}$ have the binomial and negative binomial distributions, respectively, with parameters (n, p). (These distributions are defined in (8.57) and (8.60).) Prove that $\Pr(Y_{n,p} \leqslant m) = \Pr(X_{m+n,p} \geqslant n)$. What identity in binomial coefficients does this imply?

18 A random variable X is said to have the *Poisson distribution* with mean μ if $\Pr(X = k) = e^{-\mu}\mu^k/k!$ for all $k \geqslant 0$.

The distribution of fish per unit volume of water.

a What is the pgf of such a random variable?

b What are its mean, variance, and other cumulants?

19 Continuing the previous exercise, let X_1 be a random Poisson variable with mean μ_1, and let X_2 be a random Poisson variable with mean μ_2, independent of X_1.

a What is the probability that $X_1 + X_2 = n$?

b What are the mean, variance, and other cumulants of $2X_1 + 3X_2$?

20 Prove (8.74) and (8.75), the general formulas for mean and variance of the time needed to wait for a given pattern of heads and tails.

21 What does the value of N represent, if H and T are both set equal to $\frac{1}{2}$ in (8.77)?

22 Prove (8.106), the law of conditional expectations and variances.

Homework exercises

23 Let $\Pr_{00}$ be the probability distribution of two fair dice, and let $\Pr_{11}$ be the probability distribution of two loaded dice as given in (8.2). Find all events A such that $\Pr_{00}(A) = \Pr_{11}(A)$. Which of these events depend only on the random variable S? (A probability space with $\Omega = D^2$ has 2^{36} events; only 2^{11} of those events depend on S alone.)

24 Player J rolls $2n+1$ fair dice and removes those that come up ⚅. Player K then calls a number between 1 and 6, rolls the remaining dice, and removes those that show the number called. This process is repeated until no dice remain. The player who has removed the most total dice ($n+1$ or more) is the winner.

a What are the mean and variance of the total number of dice that J removes? *Hint:* The dice are independent.

b What's the probability that J wins, when $n = 2$?

25 Consider a gambling game in which you stake a given amount A and you roll a fair die. If k spots turn up, you multiply your stake by $2(k-1)/5$. (In particular, you double the stake whenever you roll ⚅, but you lose everything if you roll ⚀.) You can stop at any time and reclaim the current stake. What are the mean and variance of your stake after n rolls? (Ignore any effects of rounding to integer amounts of currency.)

26 Find the mean and variance of the number of l-cycles in a random permutation of n elements. (The football victory problem discussed in (8.23), (8.24), and (8.53) is the special case $l = 1$.)

27 Let $X_1, X_2, \ldots, X_n$ be independent samples of the random variable X. Equations (8.19) and (8.20) explain how to estimate the mean and variance of X on the basis of these observations; give an analogous formula for estimating the third cumulant κ_3. (Your formula should be an "unbiased" estimate, in the sense that its expected value should be κ_3.)

28 What is the average length of the coin-flipping game (8.78),
a given that Alice wins?
b given that Bill wins?

29 Alice, Bill, and Computer flip a fair coin until one of the respective patterns $A = \texttt{HHTH}$, $B = \texttt{HTHH}$, or $C = \texttt{THHH}$ appears for the first time. (If only two of these patterns were involved, we know from (8.82) that A would probably beat B, that B would probably beat C, and that C would probably beat A; but all three patterns are simultaneously in the game.) What are each player's chances of winning?

30 The text considers three kinds of variances associated with successful search in a hash table. Actually there are two more: We can consider the average (over k) of the variances (over $h_1, \ldots, h_n$) of $P(h_1, \ldots, h_n; k)$; and we can consider the variance (over k) of the averages (over $h_1, \ldots, h_n$). Evaluate these quantities.

31 An apple is located at vertex A of pentagon ABCDE, and a worm is located two vertices away, at C. Every day the worm crawls with equal probability to one of the two adjacent vertices. Thus after one day the worm is at vertex B or vertex D, each with probability $\frac{1}{2}$. After two days, the worm might be back at C again, because it has no memory of previous positions. When it reaches vertex A, it stops to dine.

Schrödinger's worm.

a What are the mean and variance of the number of days until dinner?
b Let p be the probability that the number of days is 100 or more. What does Chebyshev's inequality say about p?
c What do the tail inequalities (exercise 12) tell us about p?

32 Alice and Bill are in the military, stationed in one of the five states Kansas, Nebraska, Missouri, Oklahoma, or Colorado. Initially Alice is in Nebraska and Bill is in Oklahoma. Every month each person is reassigned to an adjacent state, each adjacent state being equally likely. (Here's a diagram of the adjacencies:

The initial states are circled.) For example, Alice is restationed after the first month to Colorado, Kansas, or Missouri, each with probability 1/3. Find the mean and variance of the number of months it takes Alice and Bill to find each other. (You may wish to enlist a computer's help.)

Definitely a finite-state situation.

33 Are the random variables X_1 and X_2 in (8.89) independent?

34 Gina is a golfer who has probability $p = .05$ on each stroke of making a "supershot" that gains a stroke over par, probability $q = .91$ of making an ordinary shot, and probability $r = .04$ of making a "subshot" that costs her a stroke with respect to par. (Non-golfers: At each turn she advances 2, 1, or 0 steps toward her goal, with probability p, q, or r, respectively. On a par-m hole, her score is the minimum n such that she has advanced m or more steps after taking n turns. A low score is better than a high score.)

(Use a calculator for the numerical work on this problem.)

a Show that Gina wins a par-4 hole more often than she loses, when she plays against a player who shoots par. (In other words, the probability that her score is less than 4 is greater than the probability that her score is greater than 4.)

b Show that her average score on a par-4 hole is greater than 4. (Therefore she tends to lose against a "steady" player on total points, although she would tend to win in match play by holes.)

Exam problems

35 A die has been loaded with the probability distribution

$$\Pr(\boxed{\cdot}) = p_1\,; \qquad \Pr(\boxed{\cdot\cdot}) = p_2\,; \qquad \ldots\,; \qquad \Pr(\boxed{\cdot\cdot\cdot}) = p_6\,.$$

Let S_n be the sum of the spots after this die has been rolled n times. Find a necessary and sufficient condition on the "loading distribution" such that the two random variables $S_n \bmod 2$ and $S_n \bmod 3$ are independent of each other, for all n.

36 The six faces of a certain die contain the spot patterns

instead of the usual $\boxed{\cdot}$ through $\boxed{\cdot\cdot\cdot}$.

a Show that there is a way to assign spots to the six faces of another die so that, when these two dice are thrown, the sum of spots has the same probability distribution as the sum of spots on two ordinary dice. (Assume that all 36 face pairs are equally likely.)

b Generalizing, find all ways to assign spots to the $6n$ faces of n dice so that the distribution of spot sums will be the same as the distribution of spot sums on n ordinary dice. (Each face should receive a positive integer number of spots.)

37 Let p_n be the probability that exactly n tosses of a fair coin are needed before heads are seen twice in a row, and let $q_n = \sum_{k \geqslant n} p_k$. Find closed forms for both p_n and q_n in terms of Fibonacci numbers.

38 What is the probability generating function for the number of times you need to roll a fair die until all six faces have turned up? Generalize to m-sided fair dice: Give closed forms for the mean and variance of the number of rolls needed to see l of the m faces. What is the probability that this number will be exactly n?

39 A *Dirichlet probability generating function* has the form

$$P(z) = \sum_{n\geqslant 1} \frac{p_n}{n^z}.$$

Thus $P(0) = 1$. If X is a random variable with $\Pr(X=n) = p_n$, express $E(X)$, $V(X)$, and $E(\ln X)$ in terms of $P(z)$ and its derivatives.

40 The mth cumulant κ_m of the binomial distribution (8.57) has the form $nf_m(p)$, where f_m is a polynomial of degree m. (For example, $f_1(p) = p$ and $f_2(p) = p - p^2$, because the mean and variance are np and npq.)

a Find a closed form for the coefficient of p^k in $f_m(p)$.

b Prove that $f_m(\frac{1}{2}) = (2^m - 1)B_m/m + [m=1]$, where B_m is the mth Bernoulli number.

41 Let the random variable X_n be the number of flips of a fair coin until heads have turned up a total of n times. Show that $E(X_{n+1}^{-1}) = (-1)^n(\ln 2 + H_{\lfloor n/2\rfloor} - H_n)$. Use the methods of Chapter 9 to estimate this value with an absolute error of $O(n^{-3})$.

42 A certain man has a problem finding work. If he is unemployed on any given morning, there's constant probability p_h (independent of past history) that he will be hired before that evening; but if he's got a job when the day begins, there's constant probability p_f that he'll be laid off by nightfall. Find the average number of evenings on which he will have a job lined up, assuming that he is initially employed and that this process goes on for n days. (For example, if $n = 1$ the answer is $1 - p_f$.)

Does TEX choose optimal line breaks?

43 Find a closed form for the pgf $G_n(z) = \sum_{k\geqslant 0} p_{k,n} z^k$, where $p_{k,n}$ is the probability that a random permutation of n objects has exactly k cycles. What are the mean and standard deviation of the number of cycles?

44 The athletic department runs an intramural "knockout tournament" for 2^n tennis players as follows. In the first round, the players are paired off randomly, with each pairing equally likely, and 2^{n-1} matches are played. The winners advance to the second round, where the same process produces 2^{n-2} winners. And so on; the kth round has 2^{n-k} randomly chosen matches between the 2^{n-k+1} players who are still undefeated. The nth round produces the champion. Unbeknownst to the tournament organizers, there is actually an ordering among the players, so that x_1 is best, x_2

A peculiar set of tennis players.

is second best, ..., x_{2^n} is worst. When x_j plays x_k and $j < k$, the winner is x_j with probability p and x_k with probability $1 - p$, independent of the other matches. We assume that the same probability p applies to all j and k.

a What's the probability that x_1 wins the tournament?

b What's the probability that the nth round (the final match) is between the top two players, x_1 and x_2?

c What's the probability that the best 2^k players are the competitors in the kth-to-last round? (The previous questions were the cases $k = 0$ and $k = 1$.)

d Let $N(n)$ be the number of essentially different tournament results; two tournaments are essentially the same if the matches take place between the same players and have the same winners. Prove that $N(n) = 2^n!$.

e What's the probability that x_2 wins the tournament?

f Prove that if $\frac{1}{2} < p < 1$, the probability that x_j wins is strictly greater than the probability that x_{j+1} wins, for $1 \leqslant j < 2^n$.

"A fast arithmetic computation shows that the sherry is always at least three years old. Taking computation further gives the vertigo."
—Revue du vin de France (Nov 1984)

45 True sherry is made in Spain according to a multistage system called "Solera." For simplicity we'll assume that the winemaker has only three barrels, called A, B, and C. Every year a third of the wine from barrel C is bottled and replaced by wine from B; then B is topped off with a third of the wine from A; finally A is topped off with new wine. Let $A(z)$, $B(z)$, $C(z)$ be probability generating functions, where the coefficient of z^n is the fraction of n-year-old wine in the corresponding barrel just after the transfers have been made.

a Assume that the operation has been going on since time immemorial, so that we have a steady state in which $A(z)$, $B(z)$, and $C(z)$ are the same at the beginning of each year. Find closed forms for these generating functions.

b Find the mean and standard deviation of the age of the wine in each barrel, under the same assumptions. What is the average age of the sherry when it is bottled? How much of it is exactly 25 years old?

c Now take the finiteness of time into account: Suppose that all three barrels contained new wine at the beginning of year 0. What is the average age of the sherry that is bottled at the beginning of year n?

46 Stefan Banach used to carry two boxes of matches, each containing n matches initially. Whenever he needed a light he chose a box at random, each with probability $\frac{1}{2}$, independent of his previous choices. After taking out a match he'd put the box back in its pocket (even if the box became empty — all famous mathematicians used to do this). When his chosen box was empty he'd throw it away and reach for the other box.

a Once he found that the other box was empty too. What's the probability that this occurs? (For $n = 1$ it happens half the time and for $n = 2$ it happens $3/8$ of the time.) To answer this part, find a closed form for the generating function $P(w, z) = \sum_{m,n} p_{m,n} w^m z^n$, where $p_{m,n}$ is the probability that, starting with m matches in one box and n in the other, both boxes are empty when an empty box is first chosen. Then find a closed form for $p_{n,n}$.

b Generalizing your answer to part (a), find a closed form for the probability that exactly k matches are in the other box when an empty one is first thrown away.

c Find a closed form for the average number of matches in that other box.

And for the number in the empty box.

47 Some physicians, collaborating with some physicists, recently discovered a pair of microbes that reproduce in a peculiar way. The male microbe, called a *diphage*, has two receptors on its surface; the female microbe, called a *triphage*, has three:

When a culture of diphages and triphages is irradiated with a psi-particle, exactly one of the receptors on one of the phages absorbs the particle; each receptor is equally likely. If it was a diphage receptor, that diphage changes to a triphage; if it was a triphage receptor, that triphage splits into two diphages. Thus if an experiment starts with one diphage, the first psi-particle changes it to a triphage, the second particle splits the triphage into two diphages, and the third particle changes one of the diphages to a triphage. The fourth particle hits either the diphage or the triphage; then there are either two triphages (probability $\frac{2}{5}$) or three diphages (probability $\frac{3}{5}$). Find a closed form for the average number of diphages present, if we begin with a single diphage and irradiate the culture n times with single psi-particles.

48 Five people stand at the vertices of a pentagon, throwing frisbees to each other.

Or, if this pentagon is in Arlington, throwing missiles at each other.

They have two frisbees, initially at adjacent vertices as shown. In each time interval, each frisbee is thrown either to the left or to the right (along an edge of the pentagon) with equal probability. This process continues until one person is the target of two frisbees simultaneously; then the game stops. (All throws are independent of past history.)

Frisbee is a trademark of Wham-O Manufacturing Company.

a Find the mean and variance of the number of pairs of throws.

b Find a closed form for the probability that the game lasts more than 100 steps, in terms of Fibonacci numbers.

49 Luke Snowwalker spends winter vacations at his mountain cabin. The front porch has m pairs of boots and the back porch has n pairs. Every time he goes for a walk he flips a (fair) coin to decide whether to leave from the front porch or the back, and he puts on a pair of boots at that porch and heads off. There's a 50/50 chance that he returns to each porch, independent of his starting point, and he leaves the boots at the porch he returns to. Thus after one walk there will be $m + [-1, 0, \text{or} +1]$ pairs on the front porch and $n - [-1, 0, \text{or} +1]$ pairs on the back porch. If all the boots pile up on one porch and if he decides to leave from the other, he goes without boots and gets frostbite, ending his vacation. Assuming that he continues his walks until the bitter end, let $P_N(m, n)$ be the probability that he completes exactly N nonfrostbitten trips, starting with m pairs on the front porch and n on the back. Thus, if both m and n are positive,

$$\begin{aligned} P_N(m,n) \;=\; & \tfrac{1}{4}P_{N-1}(m-1,n+1) + \tfrac{1}{2}P_{N-1}(m,n) \\ & + \tfrac{1}{4}P_{N-1}(m+1,n-1)\,; \end{aligned}$$

this follows because this first trip is either front/back, front/front, back/back, or back/front, each with probability $\frac{1}{4}$, and $N-1$ trips remain.

a Complete the recurrence for $P_N(m, n)$ by finding formulas that hold when $m = 0$ or $n = 0$. Use the recurrence to obtain equations that hold among the probability generating functions

$$g_{m,n}(z) \;=\; \sum_{N \geqslant 0} P_N(m,n) z^N\,.$$

b Differentiate your equations and set $z = 1$, thereby obtaining relations among the quantities $g'_{m,n}(1)$. Solve these equations, thereby determining the mean number of trips before frostbite.

c Show that $g_{m,n}$ has a closed form if we substitute $z = 1/\cos^2\theta$:

$$g_{m,n}\Big(\frac{1}{\cos^2\theta}\Big) \;=\; \frac{\sin(2m+1)\theta + \sin(2n+1)\theta}{\sin(2m+2n+2)\theta}\,\cos\theta\,.$$

50 Consider the function

$$H(z) \;=\; 1 + \frac{1-z}{2z}\bigl(z - 3 + \sqrt{(1-z)(9-z)}\,\bigr)\,.$$

The purpose of this problem is to prove that $H(z) = \sum_{k\geqslant 0} h_k z^k$ is a probability generating function, and to obtain some basic facts about it.

a Let $(1-z)^{3/2}(9-z)^{1/2} = \sum_{k\geqslant 0} c_k z^k$. Prove that $c_0 = 3$, $c_1 = -14/3$, $c_2 = 37/27$, and $c_{3+l} = 3\sum_k \binom{l}{k}\binom{1/2}{3+k}\left(\frac{8}{9}\right)^{k+3}$ for all $l \geqslant 0$. *Hint:* Use the identity

$$(9-z)^{1/2} \;=\; 3(1-z)^{1/2}\bigl(1 + \tfrac{8}{9}z/(1-z)\bigr)^{1/2}$$

and expand the last factor in powers of $z/(1-z)$.

b Use part (a) and exercise 5.81 to show that the coefficients of $H(z)$ are all positive.

c Prove the amazing identity

$$\sqrt{\frac{9-H(z)}{1-H(z)}} \;=\; \sqrt{\frac{9-z}{1-z}} + 2\,.$$

d What are the mean and variance of H?

51 The state lottery in El Dorado uses the payoff distribution H defined in the previous problem. Each lottery ticket costs 1 doubloon, and the payoff is k doubloons with probability h_k. Your chance of winning with each ticket is completely independent of your chance with other tickets; in other words, winning or losing with one ticket does not affect your probability of winning with any other ticket you might have purchased in the same lottery.

a Suppose you start with one doubloon and play this game. If you win k doubloons, you buy k tickets in the second game; then you take the total winnings in the second game and apply all of them to the third; and so on. If none of your tickets is a winner, you're broke and you have to stop gambling. Prove that the pgf of your current holdings after n rounds of such play is

$$1 - \frac{4}{\sqrt{(9-z)/(1-z)} + 2n - 1} + \frac{4}{\sqrt{(9-z)/(1-z)} + 2n + 1}\,.$$

b Let g_n be the probability that you lose all your money for the first time on the nth game, and let $G(z) = g_1 z + g_2 z^2 + \cdots$. Prove that $G(1) = 1$. (This means that you're bound to lose sooner or later, with probability 1, although you might have fun playing in the meantime.) What are the mean and the variance of G?

c What is the average total number of tickets you buy, if you continue to play until going broke?

d What is the average number of games until you lose everything if you start with two doubloons instead of just one?

A doubledoubloon.

Bonus problems

52 Show that the text's definitions of median and mode for random variables correspond in some meaningful sense to the definitions of median and mode for sequences, when the probability space is finite.

53 Prove or disprove: If X, Y, and Z are random variables with the property that all three pairs (X, Y), (X, Z) and (Y, Z) are independent, then $X + Y$ is independent of Z.

54 Equation (8.20) proves that the average value of $\widehat{V}X$ is VX. What is the *variance* of $\widehat{V}X$?

55 A normal deck of playing cards contains 52 cards, four each with face values in the set $\{\mathtt{A}, 2, 3, 4, 5, 6, 7, 8, 9, 10, \mathtt{J}, \mathtt{Q}, \mathtt{K}\}$. Let X and Y denote the respective face values of the top and bottom cards, and consider the following algorithm for shuffling:

S1 Permute the deck randomly so that each arrangement occurs with probability $1/52!$.

S2 If $X \neq Y$, flip a biased coin that comes up heads with probability p, and go back to step S1 if heads turns up. Otherwise stop.

Each coin flip and each permutation is assumed to be independent of all the other randomizations. What value of p will make X and Y independent random variables after this procedure stops?

56 Generalize the frisbee problem of exercise 48 from a pentagon to an m-gon. What are the mean and variance of the number of collision-free throws in general, when the frisbees are initially at adjacent vertices? Show that, if m is odd, the pgf for the number of throws can be written as a product of coin-flipping distributions:

$$G_m(z) = \prod_{k=1}^{(m-1)/2} \frac{p_k z}{1 - q_k z},$$

$$\text{where } p_k = \sin^2 \frac{(2k-1)\pi}{2m}, \quad q_k = \cos^2 \frac{(2k-1)\pi}{2m}.$$

Hint: Try the substitution $z = 1/\cos^2 \theta$.

57 Prove that the Penney-ante pattern $\tau_1\tau_2\ldots\tau_{l-1}\tau_l$ is always inferior to the pattern $\overline{\tau}_2\tau_1\tau_2\ldots\tau_{l-1}$ when a fair coin is flipped, if $l \geqslant 3$.

58 Is there any sequence $A = \tau_1\tau_2\ldots\tau_{l-1}\tau_l$ of $l \geqslant 3$ heads and tails such that the sequences $H\tau_1\tau_2\ldots\tau_{l-1}$ and $T\tau_1\tau_2\ldots\tau_{l-1}$ both perform equally well against A in the game of Penney ante?

59 Are there patterns A and B of heads and tails such that A is longer than B, yet A appears before B more than half the time when a fair coin is being flipped?

60 Let k and n be fixed positive integers with $k < n$.

a Find a closed form for the probability generating function

$$G(w,z) = \frac{1}{m^n}\sum_{h_1=1}^{m}\cdots\sum_{h_n=1}^{m} w^{P(h_1,\ldots,h_n;k)} z^{P(h_1,\ldots,h_n;n)}$$

for the joint distribution of the numbers of probes needed to find the kth and nth items that have been inserted into a hash table with m lists.

b Although the random variables $P(h_1,\ldots,h_n;k)$ and $P(h_1,\ldots,h_n;n)$ are dependent, show that they are somewhat independent:

$$\begin{aligned}&E\big(P(h_1,\ldots,h_n;k)P(h_1,\ldots,h_n;n)\big)\\ &\qquad = \big(EP(h_1,\ldots,h_n;k)\big)\big(EP(h_1,\ldots,h_n;n)\big)\,.\end{aligned}$$

61 Use the result of the previous exercise to prove (8.104).

62 Continuing exercise 47, find the *variance* of the number of diphages after n irradiations.

Research problem

63 The *normal distribution* is a non-discrete probability distribution characterized by having all its cumulants zero except the mean and the variance. Is there an easy way to tell if a given sequence of cumulants $\langle\kappa_1,\kappa_2,\kappa_3,\ldots\rangle$ comes from a *discrete* distribution? (All the probabilities must be "atomic" in a discrete distribution.)

9

Asymptotics

EXACT ANSWERS are great when we can find them; there's something very satisfying about complete knowledge. But there's also a time when approximations are in order. If we run into a sum or a recurrence whose solution doesn't have a closed form (as far as we can tell), we still would like to know something about the answer; we don't have to insist on all or nothing. And even if we do have a closed form, our knowledge might be imperfect, since we might not know how to compare it with other closed forms.

For example, there is (apparently) no closed form for the sum

$$S_n = \sum_{k=0}^{n} \binom{3n}{k}.$$

But it is nice to know that

$$S_n \sim 2\binom{3n}{n}, \qquad \text{as } n \to \infty;$$

Uh oh ... here comes that A-word.

we say that the sum is "asymptotic to" $2\binom{3n}{n}$. It's even nicer to have more detailed information, like

$$S_n = \binom{3n}{n}\left(2 - \frac{4}{n} + O\left(\frac{1}{n^2}\right)\right), \tag{9.1}$$

which gives us a "relative error of order $1/n^2$." But even this isn't enough to tell us how big S_n is, compared with other quantities. Which is larger, S_n or the Fibonacci number F_{4n}? Answer: We have $S_2 = 22 > F_8 = 21$ when $n = 2$; but F_{4n} is eventually larger, because $F_{4n} \sim \phi^{4n}/\sqrt{5}$ and $\phi^4 \approx 6.8541$, while

$$S_n = \sqrt{\frac{3}{\pi n}}(6.75)^n\left(1 - \frac{151}{72n} + O\left(\frac{1}{n^2}\right)\right). \tag{9.2}$$

Our goal in this chapter is to learn how to understand and to derive results like this without great pain.

The word *asymptotic* stems from a Greek root meaning "not falling together." When ancient Greek mathematicians studied conic sections, they considered hyperbolas like the graph of $y = \sqrt{1 + x^2}$,

Other words like 'symptom' and 'ptomaine' also come from this root.

which has the lines $y = x$ and $y = -x$ as "asymptotes." The curve approaches but never quite touches these asymptotes, when $x \to \infty$. Nowadays we use "asymptotic" in a broader sense to mean any approximate value that gets closer and closer to the truth, when some parameter approaches a limiting value. For us, asymptotics means "almost falling together."

Some asymptotic formulas are very difficult to derive, well beyond the scope of this book. We will content ourselves with an introduction to the subject; we hope to acquire a suitable foundation on which further techniques can be built. We will be particularly interested in understanding the definitions of '$\sim$' and 'O' and similar symbols, and we'll study basic ways to manipulate asymptotic quantities.

9.1 A HIERARCHY

Functions of n that occur in practice usually have different "asymptotic growth ratios"; one of them will approach infinity faster than another. We formalize this by saying that

$$f(n) \prec g(n) \qquad \Longleftrightarrow \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\,. \tag{9.3}$$

This relation is transitive: If $f(n) \prec g(n)$ and $g(n) \prec h(n)$ then $f(n) \prec h(n)$. We also may write $g(n) \succ f(n)$ if $f(n) \prec g(n)$. This notation was introduced in 1871 by Paul du Bois-Reymond [85].

All functions great and small.

For example, $n \prec n^2$; informally we say that n grows more slowly than n^2. In fact,

$$n^\alpha \prec n^\beta \quad \Longleftrightarrow \quad \alpha < \beta\,, \tag{9.4}$$

when α and β are arbitrary real numbers.

There are, of course, many functions of n besides powers of n. We can use the $\prec$ relation to rank lots of functions into an asymptotic pecking order

that includes entries like this:

$$1 \prec \log\log n \prec \log n \prec n^{\epsilon} \prec n^{c} \prec n^{\log n} \prec c^{n} \prec n^{n} \prec c^{c^{n}}.$$

(Here ϵ and c are arbitrary constants with $0 < \epsilon < 1 < c$.)

All functions listed here, except 1, go to infinity as n goes to infinity. Thus when we try to place a new function in this hierarchy, we're not trying to determine *whether* it becomes infinite but rather *how fast*.

It helps to cultivate an expansive attitude when we're doing asymptotic analysis: We should THINK BIG, when imagining a variable that approaches infinity. For example, the hierarchy says that $\log n \prec n^{0.0001}$; this might seem wrong if we limit our horizons to teeny-tiny numbers like one googol, $n = 10^{100}$. For in that case, $\log n = 100$, while $n^{0.0001}$ is only $10^{0.01} \approx 1.0233$. But if we go up to a googolplex, $n = 10^{10^{100}}$, then $\log n = 10^{100}$ pales in comparison with $n^{0.0001} = 10^{10^{96}}$.

Even if ϵ is extremely small (smaller than, say, $1/10^{10^{100}}$), the value of $\log n$ will be much smaller than the value of n^{ϵ}, if n is large enough. For if we set $n = 10^{10^{2k}}$, where k is so large that $\epsilon \geqslant 10^{-k}$, we have $\log n = 10^{2k}$ but $n^{\epsilon} \geqslant 10^{10^{k}}$. The ratio $(\log n)/n^{\epsilon}$ therefore approaches zero as $n \to \infty$.

The hierarchy shown above deals with functions that go to infinity. Often, however, we're interested in functions that go to zero, so it's useful to have a similar hierarchy for those functions. We get one by taking reciprocals, because when $f(n)$ and $g(n)$ are never zero we have

A loerarchy?

$$f(n) \prec g(n) \iff \frac{1}{g(n)} \prec \frac{1}{f(n)}. \tag{9.5}$$

Thus, for example, the following functions (except 1) all go to zero:

$$\frac{1}{c^{c^{n}}} \prec \frac{1}{n^{n}} \prec \frac{1}{c^{n}} \prec \frac{1}{n^{\log n}} \prec \frac{1}{n^{c}} \prec \frac{1}{n^{\epsilon}} \prec \frac{1}{\log n} \prec \frac{1}{\log\log n} \prec 1.$$

Let's look at a few other functions to see where they fit in. The number $\pi(n)$ of primes less than or equal to n is known to be approximately $n/\ln n$. Since $1/n^{\epsilon} \prec 1/\ln n \prec 1$, multiplying by n tells us that

$$n^{1-\epsilon} \prec \pi(n) \prec n.$$

We can in fact generalize (9.4) by noticing, for example, that

$$\begin{aligned} n^{\alpha_1}(\log n)^{\alpha_2}(\log\log n)^{\alpha_3} &\prec n^{\beta_1}(\log n)^{\beta_2}(\log\log n)^{\beta_3} \\ &\iff (\alpha_1, \alpha_2, \alpha_3) < (\beta_1, \beta_2, \beta_3). \end{aligned} \tag{9.6}$$

Here '$(\alpha_1, \alpha_2, \alpha_3) < (\beta_1, \beta_2, \beta_3)$' means lexicographic order (dictionary order); in other words, either $\alpha_1 < \beta_1$, or $\alpha_1 = \beta_1$ and $\alpha_2 < \beta_2$, or $\alpha_1 = \beta_1$ and $\alpha_2 = \beta_2$ and $\alpha_3 < \beta_3$.

How about the function $e^{\sqrt{\log n}}$; where does it live in the hierarchy? We can answer questions like this by using the rule

$$e^{f(n)} \prec e^{g(n)} \qquad \Longleftrightarrow \qquad \lim_{n\to\infty}\bigl(f(n)-g(n)\bigr) = -\infty\,, \tag{9.7}$$

which follows in two steps from definition (9.3) by taking logarithms. Consequently

$$1 \prec f(n) \prec g(n) \qquad \Longrightarrow \qquad e^{|f(n)|} \prec e^{|g(n)|}\,.$$

And since $1 \prec \log\log n \prec \sqrt{\log n} \prec \epsilon \log n$, we have $\log n \prec e^{\sqrt{\log n}} \prec n^{\epsilon}$.

When two functions $f(n)$ and $g(n)$ have the *same* rate of growth, we write '$f(n) \asymp g(n)$'. The official definition is:

$$f(n) \asymp g(n) \quad \Longleftrightarrow \quad \bigl|f(n)\bigr| \leqslant C\bigl|g(n)\bigr| \quad\text{and}\quad \bigl|g(n)\bigr| \leqslant C\bigl|f(n)\bigr|\,, \quad \text{for some } C \text{ and for all sufficiently large } n. \tag{9.8}$$

This holds, for example, if $f(n)$ is constant and $g(n) = \cos n + \arctan n$. We will prove shortly that it holds whenever $f(n)$ and $g(n)$ are polynomials of the same degree. There's also a stronger relation, defined by the rule

$$f(n) \sim g(n) \quad \Longleftrightarrow \quad \lim_{n\to\infty}\frac{f(n)}{g(n)} = 1\,. \tag{9.9}$$

In this case we say that "$f(n)$ is asymptotic to $g(n)$."

G. H. Hardy [179] introduced an interesting and important concept called the class of *logarithmico-exponential functions*, defined recursively as the smallest family $\mathfrak{L}$ of functions satisfying the following properties:

- The constant function $f(n) = \alpha$ is in $\mathfrak{L}$, for all real α.
- The identity function $f(n) = n$ is in $\mathfrak{L}$.
- If $f(n)$ and $g(n)$ are in $\mathfrak{L}$, so is $f(n) - g(n)$.
- If $f(n)$ is in $\mathfrak{L}$, so is $e^{f(n)}$.
- If $f(n)$ is in $\mathfrak{L}$ and is "eventually positive," then $\ln f(n)$ is in $\mathfrak{L}$.

A function $f(n)$ is called "eventually positive" if there is an integer n_0 such that $f(n) > 0$ whenever $n \geqslant n_0$.

We can use these rules to show, for example, that $f(n) + g(n)$ is in $\mathfrak{L}$ whenever $f(n)$ and $g(n)$ are, because $f(n) + g(n) = f(n) - \bigl(0 - g(n)\bigr)$. If $f(n)$ and $g(n)$ are eventually positive members of $\mathfrak{L}$, their product $f(n)g(n) = e^{\ln f(n) + \ln g(n)}$ and quotient $f(n)/g(n) = e^{\ln f(n) - \ln g(n)}$ are in $\mathfrak{L}$; so are functions like $\sqrt{f(n)} = e^{\frac{1}{2}\ln f(n)}$, etc. Hardy proved that every logarithmico-exponential function is eventually positive, eventually negative, or identically zero. Therefore the product and quotient of any two $\mathfrak{L}$-functions is in $\mathfrak{L}$, except that we cannot divide by a function that's identically zero.

Hardy's main theorem about logarithmico-exponential functions is that they form an asymptotic hierarchy: If $f(n)$ and $g(n)$ are any functions in $\mathfrak{L}$, then either $f(n) \prec g(n)$, or $f(n) \succ g(n)$, or $f(n) \asymp g(n)$. In the last case there is, in fact, a constant α such that

$$f(n) \sim \alpha g(n).$$

The proof of Hardy's theorem is beyond the scope of this book; but it's nice to know that the theorem exists, because almost every function we ever need to deal with is in $\mathfrak{L}$. In practice, we can generally fit a given function into a given hierarchy without great difficulty.

9.2 O NOTATION

A wonderful notational convention for asymptotic analysis was introduced by Paul Bachmann in 1894 and popularized in subsequent years by Edmund Landau and others. We have seen it in formulas like

"... wir durch das Zeichen $O(n)$ eine Größe ausdrücken, deren Ordnung in Bezug auf n die Ordnung von n nicht überschreitet; ob sie wirklich Glieder von der Ordnung n in sich enthält, bleibt bei dem bisherigen Schlußverfahren dahingestellt."
—P. Bachmann [17]

$$H_n = \ln n + \gamma + O(1/n), \tag{9.10}$$

which tells us that the nth harmonic number is equal to the natural logarithm of n plus Euler's constant, plus a quantity that is "Big Oh of 1 over n." This last quantity isn't specified exactly; but whatever it is, the notation claims that its absolute value is no more than a constant times $1/n$.

The beauty of O-notation is that it suppresses unimportant detail and lets us concentrate on salient features: The quantity $O(1/n)$ is negligibly small, if constant multiples of $1/n$ are unimportant.

Furthermore we get to use O right in the middle of a formula. If we want to express (9.10) in terms of the notations in Section 9.1, we must transpose '$\ln n + \gamma$' to the left side and specify a weaker result like

$$H_n - \ln n - \gamma \prec \frac{\log\log n}{n}$$

or a stronger result like

$$H_n - \ln n - \gamma \asymp \frac{1}{n}.$$

The Big Oh notation allows us to specify an appropriate amount of detail in place, without transposition.

The idea of imprecisely specified quantities can be made clearer if we consider some additional examples. We occasionally use the notation '± 1' to stand for something that is either $+1$ or -1; we don't know (or perhaps we don't care) which it is, yet we can manipulate it in formulas.

N. G. de Bruijn begins his book *Asymptotic Methods in Analysis* [74] by considering a Big Ell notation that helps us understand Big Oh. If we write $L(5)$ for a number whose absolute value is less than 5 (but we don't say what the number is), then we can perform certain calculations without knowing the full truth. For example, we can deduce formulas such as $1 + L(5) = L(6)$; $L(2) + L(3) = L(5)$; $L(2)L(3) = L(6)$; $e^{L(5)} = L(e^5)$; and so on. But we cannot conclude that $L(5) - L(3) = L(2)$, since the left side might be $4 - 0$. In fact, the most we can say is $L(5) - L(3) = L(8)$.

Bachmann's O-notation is similar to L-notation but it's even less precise: $O(\alpha)$ stands for a number whose absolute value is at most a constant times $|\alpha|$. We don't say what the number is and we don't even say what the constant is. Of course the notion of a "constant" is nonsense if there is nothing variable in the picture, so we use O-notation only in contexts when there's at least one quantity (say n) whose value is varying. The formula

It's not nonsense, but it is pointless.

$$f(n) \;=\; O\big(g(n)\big) \qquad \text{for all } n \tag{9.11}$$

means in this context that there is a constant C such that

$$\big|f(n)\big| \;\leqslant\; C\big|g(n)\big| \qquad \text{for all } n; \tag{9.12}$$

and when $O\big(g(n)\big)$ stands in the middle of a formula it represents a function $f(n)$ that satisfies (9.12). The values of $f(n)$ are unknown, but we do know that they aren't too large. Similarly, de Bruijn's '$L(n)$' represents an unspecified function $f(n)$ whose values satisfy $\big|f(n)\big| < |n|$. The main difference between L and O is that O-notation involves an unspecified constant C; each appearance of O might involve a different C, but each C is independent of n.

I've got a little list — I've got a little list,
Of annoying terms and details that might well be under ground,
And that never would be missed — that never would be missed.

For example, we know that the sum of the first n squares is

$$\square_n \;=\; \tfrac{1}{3}n(n+\tfrac{1}{2})(n+1) \;=\; \tfrac{1}{3}n^3 + \tfrac{1}{2}n^2 + \tfrac{1}{6}n\,.$$

We can write

$$\square_n \;=\; O(n^3)$$

because $|\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n| \leqslant \frac{1}{3}|n|^3 + \frac{1}{2}|n|^2 + \frac{1}{6}|n| \leqslant \frac{1}{3}|n^3| + \frac{1}{2}|n^3| + \frac{1}{6}|n^3| = |n^3|$ for all integers n. Similarly, we have the more specific formula

$$\square_n \;=\; \tfrac{1}{3}n^3 + O(n^2)\,;$$

we can also be sloppy and throw away information, saying that

$$\square_n \;=\; O(n^{10})\,.$$

Nothing in the definition of O requires us to give a best possible bound.

But wait a minute. What if the variable n isn't an integer? What if we have a formula like $S(x) = \frac{1}{3}x^3 + \frac{1}{2}x^2 + \frac{1}{6}x$, where x is a real number? Then we cannot say that $S(x) = O(x^3)$, because the ratio $S(x)/x^3 = \frac{1}{3} + \frac{1}{2}x^{-1} + \frac{1}{6}x^{-2}$ becomes unbounded when $x \to 0$. And we cannot say that $S(x) = O(x)$, because the ratio $S(x)/x = \frac{1}{3}x^2 + \frac{1}{2}x + \frac{1}{6}$ becomes unbounded when $x \to \infty$. So we apparently can't use O-notation with $S(x)$.

The answer to this dilemma is that variables used with O are generally subject to side conditions. For example, if we stipulate that $|x| \geqslant 1$, or that $x \geqslant \epsilon$ where ϵ is any positive constant, or that x is an integer, then we can write $S(x) = O(x^3)$. If we stipulate that $|x| \leqslant 1$, or that $|x| \leqslant c$ where c is any positive constant, then we can write $S(x) = O(x)$. The O-notation is governed by its environment, by constraints on the variables involved.

These constraints are often specified by a limiting relation. For example, we might say that

$$f(n) \;=\; O\big(g(n)\big) \qquad \text{as } n \to \infty. \tag{9.13}$$

This means that the O-condition is supposed to hold when n is "near" ∞; we don't care what happens unless n is quite large. Moreover, we don't even specify exactly what "near" means; in such cases each appearance of O implicitly asserts the existence of *two* constants C and n_0, such that

$$\big|f(n)\big| \;\leqslant\; C\big|g(n)\big| \qquad \text{whenever } n \geqslant n_0. \tag{9.14}$$

The values of C and n_0 might be different for each O, but they do not depend on n. Similarly, the notation

You are the fairest of your sex,
Let me be your hero;
I love you as one over x,
As x approaches zero.
Positively.

$$f(x) \;=\; O\big(g(x)\big) \qquad \text{as } x \to 0$$

means that there exist two constants C and ϵ such that

$$\big|f(x)\big| \;\leqslant\; C\big|g(x)\big| \qquad \text{whenever } |x| \leqslant \epsilon. \tag{9.15}$$

The limiting value does not have to be ∞ or 0; we can write

$$\ln z \;=\; z - 1 + O\big((z-1)^2\big) \qquad \text{as } z \to 1,$$

because it can be proved that $|\ln z - z + 1| \leqslant |z-1|^2$ when $|z-1| \leqslant \frac{1}{2}$.

Our definition of O has gradually developed, over a few pages, from something that seemed pretty obvious to something that seems rather complex; we now have O representing an undefined function and either one or two unspecified constants, depending on the environment. This may seem complicated enough for any reasonable notation, but it's still not the whole story! Another

subtle consideration lurks in the background. Namely, we need to realize that it's fine to write

$$\tfrac{1}{3}n^3 + \tfrac{1}{2}n^2 + \tfrac{1}{6}n \;=\; O(n^3)\,,$$

but we should *never* write this equality with the sides reversed. Otherwise we could deduce ridiculous things like $n = n^2$ from the identities $n = O(n^2)$ and $n^2 = O(n^2)$. When we work with O-notation and any other formulas that involve imprecisely specified quantities, we are dealing with *one-way equalities*. The right side of an equation does not give more information than the left side, and it may give less; the right is a "crudification" of the left.

"And to auoide the tediouse repetition of these woordes: is equalle to: I will sette as I doe often in woorke use, a paire of paralleles, or Gemowe lines of one lengthe, thus: ═══, bicause noe .2. thynges, can be moare equalle."
—R. Recorde [305]

From a strictly formal point of view, the notation $O\big(g(n)\big)$ does not stand for a single function $f(n)$, but for the *set* of all functions $f(n)$ such that $\big|f(n)\big| \leqslant C\big|g(n)\big|$ for some constant C. An ordinary formula $g(n)$ that doesn't involve O-notation stands for the set containing a single function $f(n) = g(n)$. If S and T are sets of functions of n, the notation $S + T$ stands for the set of all functions of the form $f(n) + g(n)$, where $f(n) \in S$ and $g(n) \in T$; other notations like $S-T$, ST, S/T, $\sqrt{S}$, e^S, $\ln S$ are defined similarly. Then an "equation" between such sets of functions is, strictly speaking, a *set inclusion*; the '$=$' sign really means '$\subseteq$'. These formal definitions put all of our O manipulations on firm logical ground.

For example, the "equation"

$$\tfrac{1}{3}n^3 + O(n^2) \;=\; O(n^3)$$

means that $S_1 \subseteq S_2$, where S_1 is the set of all functions of the form $\frac{1}{3}n^3 + f_1(n)$ such that there exists a constant C_1 with $\big|f_1(n)\big| \leqslant C_1|n^2|$, and where S_2 is the set of all functions $f_2(n)$ such that there exists a constant C_2 with $\big|f_2(n)\big| \leqslant C_2|n^3|$. We can formally prove this "equation" by taking an arbitrary element of the left-hand side and showing that it belongs to the right-hand side: Given $\frac{1}{3}n^3 + f_1(n)$ such that $\big|f_1(n)\big| \leqslant C_1|n^2|$, we must prove that there's a constant C_2 such that $|\frac{1}{3}n^3 + f_1(n)| \leqslant C_2|n^3|$. The constant $C_2 = \frac{1}{3} + C_1$ does the trick, since $n^2 \leqslant |n^3|$ for all integers n.

If '$=$' really means '$\subseteq$', why don't we use '$\subseteq$' instead of abusing the equals sign? There are four reasons.

First, tradition. Number theorists started using the equals sign with O-notation and the practice stuck. It's sufficiently well established by now that we cannot hope to get the mathematical community to change.

Second, tradition. Computer people are quite used to seeing equals signs abused—for years FORTRAN and BASIC programmers have been writing assignment statements like '$N = N + 1$'. One more abuse isn't much.

Third, tradition. We often read '$=$' as the word 'is'. For instance we verbalize the formula $H_n = O(\log n)$ by saying "H sub n is Big Oh of log n."

And in English, this 'is' is one-way. We say that a bird is an animal, but we don't say that an animal is a bird; "animal" is a crudification of "bird."

"It is obvious that the sign = is really the wrong sign for such relations, because it suggests symmetry, and there is no such symmetry. ... Once this warning has been given, there is, however, not much harm in using the sign =, and we shall maintain it, for no other reason than that it is customary."
—N. G. de Bruijn [74]

Fourth, for our purposes it's natural. If we limited our use of O-notation to situations where it occupies the whole right side of a formula—as in the harmonic number approximation $H_n = O(\log n)$, or as in the description of a sorting algorithm's running time $T(n) = O(n \log n)$—it wouldn't matter whether we used '=' or something else. But when we use O-notation in the middle of an expression, as we usually do in asymptotic calculations, our intuition is well satisfied if we think of the equals sign as an equality, and if we think of something like $O(1/n)$ as a very small quantity.

So we'll continue to use '=', and we'll continue to regard $O\big(g(n)\big)$ as an incompletely specified function, knowing that we can always fall back on the set-theoretic definition if we must.

But we ought to mention one more technicality while we're picking nits about definitions: If there are several variables in the environment, O-notation formally represents sets of functions of two or more variables, not just one. The domain of each function is every variable that is currently "free" to vary.

This concept can be a bit subtle, because a variable might be defined only in parts of an expression, when it's controlled by a $\sum$ or something similar. For example, let's look closely at the equation

$$\sum_{k=0}^{n} \big(k^2 + O(k)\big) \;=\; \tfrac{1}{3}n^3 + O(n^2)\,, \qquad \text{integer } n \geqslant 0. \tag{9.16}$$

The expression $k^2 + O(k)$ on the left stands for the set of all two-variable functions of the form $k^2 + f(k,n)$ such that there exists a constant C with $\big|f(k,n)\big| \leqslant Ck$ for $0 \leqslant k \leqslant n$. The sum of this set of functions, for $0 \leqslant k \leqslant n$, is the set of all functions $g(n)$ of the form

$$\sum_{k=0}^{n} \big(k^2 + f(k,n)\big) \;=\; \tfrac{1}{3}n^3 + \tfrac{1}{2}n^2 + \tfrac{1}{6}n + f(0,n) + f(1,n) + \cdots + f(n,n)\,,$$

where f has the stated property. Since we have

$$\begin{aligned}
&\big|\tfrac{1}{2}n^2 + \tfrac{1}{6}n + f(0,n) + f(1,n) + \cdots + f(n,n)\big| \\
&\qquad \leqslant\; \tfrac{1}{2}n^2 + \tfrac{1}{6}n^2 + C\cdot 0 + C\cdot 1 + \cdots + C\cdot n \\
&\qquad <\; n^2 + C(n^2+n)/2 \;<\; (C+1)n^2\,,
\end{aligned}$$

(Now is a good time to do warmup exercises 3 and 4.)

all such functions $g(n)$ belong to the right-hand side of (9.16); therefore (9.16) is true.

People sometimes abuse O-notation by assuming that it gives an exact order of growth; they use it as if it specifies a lower bound as well as an upper bound. For example, an algorithm to sort n numbers might be called

inefficient "because its running time is $O(n^2)$." But a running time of $O(n^2)$ does not imply that the running time is not also $O(n)$. There's another notation, Big Omega, for lower bounds:

$$f(n) = \Omega\big(g(n)\big) \iff \big|f(n)\big| \geqslant C\big|g(n)\big| \qquad \text{for some } C > 0. \tag{9.17}$$

We have $f(n) = \Omega\big(g(n)\big)$ if and only if $g(n) = O\big(f(n)\big)$. A sorting algorithm whose running time is $\Omega(n^2)$ is inefficient compared with one whose running time is $O(n \log n)$, if n is large enough.

Finally there's Big Theta, which specifies an exact order of growth:

Since Ω and Θ are uppercase Greek letters, the O in O-notation must be a capital Greek Omicron. After all, Greeks invented asymptotics.

$$f(n) = \Theta\big(g(n)\big) \iff \begin{array}{c} f(n) = O\big(g(n)\big) \\ \text{and} \quad f(n) = \Omega\big(g(n)\big). \end{array} \tag{9.18}$$

We have $f(n) = \Theta\big(g(n)\big)$ if and only if $f(n) \asymp g(n)$ in the notation we saw previously, equation (9.8).

Edmund Landau [238] invented a "little oh" notation,

$$\begin{aligned} f(n) &= o\big(g(n)\big) \\ &\iff \big|f(n)\big| \leqslant \epsilon\big|g(n)\big| \qquad \begin{array}{c} \text{for all } n \geqslant n_0(\epsilon) \text{ and} \\ \text{for all constants } \epsilon > 0. \end{array} \end{aligned} \tag{9.19}$$

This is essentially the relation $f(n) \prec g(n)$ of (9.3). We also have

$$f(n) \sim g(n) \iff f(n) = g(n) + o\big(g(n)\big). \tag{9.20}$$

Many authors use 'o' in asymptotic formulas, but a more explicit 'O' expression is almost always preferable. For example, the average running time of a computer method called "bubblesort" depends on the asymptotic value of the sum $P(n) = \sum_{k=0}^{n} k^{n-k} k!/n!$. Elementary asymptotic methods suffice to prove the formula $P(n) \sim \sqrt{\pi n/2}$, which means that the ratio $P(n)/\sqrt{\pi n/2}$ approaches 1 as $n \to \infty$. However, the true behavior of $P(n)$ is best understood by considering the *difference*, $P(n) - \sqrt{\pi n/2}$, not the ratio:

n	$P(n)/\sqrt{\pi n/2}$	$P(n) - \sqrt{\pi n/2}$
1	0.798	−0.253
10	0.878	−0.484
20	0.904	−0.538
30	0.918	−0.561
40	0.927	−0.575
50	0.934	−0.585

The numerical evidence in the middle column is not very compelling; it certainly is far from a dramatic proof that $P(n)/\sqrt{\pi n/2}$ approaches 1 rapidly,

if at all. But the right-hand column shows that $P(n)$ is very close indeed to $\sqrt{\pi n/2}$. Thus we can characterize the behavior of $P(n)$ much better if we can derive formulas of the form

$$P(n) \;=\; \sqrt{\pi n/2} + O(1)\,,$$

or even sharper estimates like

$$P(n) \;=\; \sqrt{\pi n/2} - \tfrac{2}{3} + O(1/\sqrt{n}\,)\,.$$

Stronger methods of asymptotic analysis are needed to prove O-results, but the additional effort required to learn these stronger methods is amply compensated by the improved understanding that comes with O-bounds.

Moreover, many sorting algorithms have running times of the form

$$T(n) \;=\; A\,n\lg n \;+\; B\,n \;+\; O(\log n)$$

for some constants A and B. Analyses that stop at $T(n) \sim A\,n\lg n$ don't tell the whole story, and it turns out to be a bad strategy to choose a sorting algorithm based just on its A value. Algorithms with a good 'A' often achieve this at the expense of a bad 'B'. Since $n\lg n$ grows only slightly faster than n, the algorithm that's faster asymptotically (the one with a slightly smaller A value) might be faster only for values of n that never actually arise in practice. Thus, asymptotic methods that allow us to go past the first term and evaluate B are necessary if we are to make the right choice of method.

Also lD, the Duraflame logarithm.

Before we go on to study O, let's talk about one more small aspect of mathematical style. Three different notations for logarithms have been used in this chapter: lg, ln, and log. We often use 'lg' in connection with computer methods, because binary logarithms are often relevant in such cases; and we often use 'ln' in purely mathematical calculations, since the formulas for natural logarithms are nice and simple. But what about 'log'? Isn't this the "common" base-10 logarithm that students learn in high school—the "common" logarithm that turns out to be very uncommon in mathematics and computer science? Yes; and many mathematicians confuse the issue by using 'log' to stand for natural logarithms or binary logarithms. There is no universal agreement here. But we can usually breathe a sigh of relief when a logarithm appears inside O-notation, because O ignores multiplicative constants. There is no difference between $O(\lg n)$, $O(\ln n)$, and $O(\log n)$, as $n \to \infty$; similarly, there is no difference between $O(\lg\lg n)$, $O(\ln\ln n)$, and $O(\log\log n)$. We get to choose whichever we please; and the one with 'log' seems friendlier because it is more pronounceable. Therefore we generally use 'log' in all contexts where it improves readability without introducing ambiguity.

Notice that $\log\log\log n$ *is undefined when* $n \leqslant 10$.

9.3 O MANIPULATION

Like any mathematical formalism, the O-notation has rules of manipulation that free us from the grungy details of its definition. Once we prove that the rules are correct, using the definition, we can henceforth work on a higher plane and forget about actually verifying that one set of functions is contained in another. We don't even need to calculate the constants C that are implied by each O, as long as we follow rules that guarantee the existence of such constants.

The secret of being a bore is to tell everything.
—Voltaire

For example, we can prove once and for all that

$$n^m = O(n^{m'}), \qquad \text{when } m \leqslant m'; \tag{9.21}$$

$$O\big(f(n)\big) + O\big(g(n)\big) = O\big(|f(n)| + |g(n)|\big)\,. \tag{9.22}$$

Then we can say immediately that $\frac{1}{3}n^3+\frac{1}{2}n^2+\frac{1}{6}n = O(n^3)+O(n^3)+O(n^3) = O(n^3)$, without the laborious calculations in the previous section.

Here are some more rules that follow easily from the definition:

$$f(n) = O\big(f(n)\big)\,; \tag{9.23}$$

$$c \cdot O\big(f(n)\big) = O\big(f(n)\big)\,, \qquad \text{if } c \text{ is constant;} \tag{9.24}$$

$$O\big(O\big(f(n)\big)\big) = O\big(f(n)\big)\,; \tag{9.25}$$

$$O\big(f(n)\big)O\big(g(n)\big) = O\big(f(n)\,g(n)\big)\,; \tag{9.26}$$

$$O\big(f(n)\,g(n)\big) = f(n)O\big(g(n)\big)\,. \tag{9.27}$$

Exercise 9 proves (9.22), and the proofs of the others are similar. We can always replace something of the form on the left by what's on the right, regardless of the side conditions on the variable n.

Equations (9.27) and (9.23) allow us to derive the identity $O\big(f(n)^2\big) = O\big(f(n)\big)^2$. This sometimes helps avoid parentheses, since we can write

$$O(\log n)^2 \qquad \text{instead of} \qquad O\big((\log n)^2\big)\,.$$

Both of these are preferable to '$O(\log^2 n)$', which is ambiguous because some authors use it to mean '$O(\log\log n)$'.

(Note: The formula $O(f(n))^2$ does not denote the set of all functions $g(n)^2$ where $g(n)$ is in $O(f(n))$; such functions $g(n)^2$ cannot be negative, but the set $O(f(n))^2$ includes negative functions. In general, when S is a set, the notation S^2 stands for the set of all products $s_1 s_2$ with s_1 and s_2 in S, not for the set of all squares s^2 with $s \in S$.)

Can we also write

$$O(\log n)^{-1} \qquad \text{instead of} \qquad O\big((\log n)^{-1}\big)\,?$$

No! This is an abuse of notation, since the set of functions $1/O(\log n)$ is neither a subset nor a superset of $O(1/\log n)$. We could legitimately substitute $\Omega(\log n)^{-1}$ for $O\big((\log n)^{-1}\big)$, but this would be awkward. So we'll restrict our use of "exponents outside the O" to constant, positive integer exponents.

Power series give us some of the most useful operations of all. If the sum

$$S(z) \;=\; \sum_{n\geqslant 0} a_n\, z^n$$

converges absolutely for some complex number $z = z_0$, then

$$S(z) \;=\; O(1)\,, \qquad \text{for all } |z| \leqslant |z_0|.$$

This is obvious, because

$$|S(z)| \;\leqslant\; \sum_{n\geqslant 0} |a_n|\,|z|^n \;\leqslant\; \sum_{n\geqslant 0} |a_n|\,|z_0|^n \;=\; C \;<\; \infty\,.$$

In particular, $S(z) = O(1)$ as $z \to 0$, and $S(1/n) = O(1)$ as $n \to \infty$, provided only that $S(z)$ converges for at least one nonzero value of z. We can use this principle to truncate a power series at any convenient point and estimate the remainder with O. For example, not only is $S(z) = O(1)$, but

$$\begin{aligned} S(z) \;&=\; a_0 + O(z)\,, \\ S(z) \;&=\; a_0 + a_1 z + O(z^2)\,, \end{aligned}$$

and so on, because

$$S(z) \;=\; \sum_{0\leqslant k<m} a_k z^k + z^m \sum_{n\geqslant m} a_n z^{n-m}$$

and the latter sum, like $S(z)$ itself, converges absolutely for $z = z_0$ and is $O(1)$. Table 452 lists some of the most useful asymptotic formulas, half of which are simply based on truncation of power series according to this rule.

Dirichlet series, which are sums of the form $\sum_{k\geqslant 1} a_k/k^z$, can be truncated in a similar way: If a Dirichlet series converges absolutely when $z = z_0$, we can truncate it at any term and get the approximation

$$\sum_{1\leqslant k<m} a_k/k^z + O(m^{-z})\,,$$

Remember that $\Re$ stands for "real part."

valid for $\Re z \geqslant \Re z_0$. The asymptotic formula for Bernoulli numbers B_n in Table 452 illustrates this principle.

On the other hand, the asymptotic formulas for H_n, $n!$, and $\pi(n)$ in Table 452 are not truncations of convergent series; if we extended them indefinitely they would diverge for all values of n. This is particularly easy to see in the case of $\pi(n)$, since we have already observed in Section 7.3, Example 5, that the power series $\sum_{k\geqslant 0} k!/(\ln n)^k$ is everywhere divergent. Yet these truncations of divergent series turn out to be useful approximations.

Table 452 Asymptotic approximations, valid as $n \to \infty$ and $z \to 0$.

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} + O\Bigl(\frac{1}{n^6}\Bigr). \tag{9.28}$$

$$n! = \sqrt{2\pi n}\,\Bigl(\frac{n}{e}\Bigr)^n \Bigl(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{51840n^3} + O\Bigl(\frac{1}{n^4}\Bigr)\Bigr). \tag{9.29}$$

$$B_n = 2[n \text{ even}](-1)^{n/2-1}\frac{n!}{(2\pi)^n}\bigl(1 + 2^{-n} + 3^{-n} + O(4^{-n})\bigr). \tag{9.30}$$

$$\pi(n) = \frac{n}{\ln n} + \frac{n}{(\ln n)^2} + \frac{2!\,n}{(\ln n)^3} + \frac{3!\,n}{(\ln n)^4} + O\Bigl(\frac{n}{(\log n)^5}\Bigr). \tag{9.31}$$

$$e^z = 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \frac{z^4}{4!} + O(z^5). \tag{9.32}$$

$$\ln(1+z) = z - \frac{z^2}{2} + \frac{z^3}{3} - \frac{z^4}{4} + O(z^5). \tag{9.33}$$

$$\frac{1}{1-z} = 1 + z + z^2 + z^3 + z^4 + O(z^5). \tag{9.34}$$

$$(1+z)^\alpha = 1 + \alpha z + \binom{\alpha}{2}z^2 + \binom{\alpha}{3}z^3 + \binom{\alpha}{4}z^4 + O(z^5). \tag{9.35}$$

An asymptotic approximation is said to have *absolute error* $O\bigl(g(n)\bigr)$ if it has the form $f(n) + O\bigl(g(n)\bigr)$ where $f(n)$ doesn't involve O. The approximation has *relative error* $O\bigl(g(n)\bigr)$ if it has the form $f(n)\bigl(1 + O\bigl(g(n)\bigr)\bigr)$ where $f(n)$ doesn't involve O. For example, the approximation for H_n in Table 452 has absolute error $O(n^{-6})$; the approximation for $n!$ has relative error $O(n^{-4})$. (The right-hand side of (9.29) doesn't actually have the required form $f(n)\bigl(1 + O(n^{-4})\bigr)$, but we could rewrite it

$$\sqrt{2\pi n}\,\Bigl(\frac{n}{e}\Bigr)^n \Bigl(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{51840n^3}\Bigr)\bigl(1 + O(n^{-4})\bigr)$$

if we wanted to; a similar calculation is the subject of exercise 12.) The absolute error of this approximation is $O(n^{n-3.5}e^{-n})$. Absolute error is related to the number of correct decimal digits to the right of the decimal point if the O term is ignored; relative error corresponds to the number of correct "significant figures."

(Relative error is nice for taking reciprocals, because $1/(1 + O(\epsilon)) = 1 + O(\epsilon)$.)

We can use truncation of power series to prove the general laws

$$\ln\bigl(1 + O(f(n))\bigr) = O\bigl(f(n)\bigr), \qquad \text{if } f(n) \prec 1; \tag{9.36}$$

$$e^{O(f(n))} = 1 + O\bigl(f(n)\bigr), \qquad \text{if } f(n) = O(1). \tag{9.37}$$

(Here we assume that $n \to \infty$; similar formulas hold for $\ln\bigl(1 + O(f(x))\bigr)$ and $e^{O(f(x))}$ as $x \to 0$.) For example, let $\ln\bigl(1 + g(n)\bigr)$ be any function belonging to the left side of (9.36). Then there are constants C, n_0, and c such that

$$\bigl|g(n)\bigr| \;\leqslant\; C\bigl|f(n)\bigr| \;\leqslant\; c \;<\; 1\,, \qquad \text{for all } n \geqslant n_0.$$

It follows that the infinite sum

$$\ln\bigl(1 + g(n)\bigr) \;=\; g(n) \cdot \bigl(1 - \tfrac{1}{2}g(n) + \tfrac{1}{3}g(n)^2 - \cdots\bigr)$$

converges for all $n \geqslant n_0$, and the parenthesized series is bounded by the constant $1 + \frac{1}{2}c + \frac{1}{3}c^2 + \cdots$. This proves (9.36), and the proof of (9.37) is similar. Equations (9.36) and (9.37) combine to give the useful formula

$$\bigl(1 + O(f(n))\bigr)^{O(g(n))} \;=\; 1 + O\bigl(f(n)g(n)\bigr)\,, \qquad \begin{array}{c}\text{if } f(n) \prec 1 \text{ and}\\ f(n)g(n) = O(1).\end{array} \tag{9.38}$$

Problem 1: Return to the Wheel of Fortune.

Let's try our luck now at a few asymptotic problems. In Chapter 3 we derived equation (3.13) for the number of winning positions in a certain game:

$$W \;=\; \lfloor N/K \rfloor + \tfrac{1}{2}K^2 + \tfrac{5}{2}K - 3\,, \qquad K = \lfloor \sqrt[3]{N} \rfloor.$$

And we promised that an asymptotic version of W would be derived in Chapter 9. Well, here we are in Chapter 9; let's try to estimate W, as $N \to \infty$.

The main idea here is to remove the floor brackets, replacing K by $N^{1/3} + O(1)$. Then we can go further and write

$$K \;=\; N^{1/3}\bigl(1 + O(N^{-1/3})\bigr)\,;$$

this is called "pulling out the large part." (We will be using this trick a lot.) Now we have

$$\begin{aligned} K^2 \;&=\; N^{2/3}\bigl(1 + O(N^{-1/3})\bigr)^2 \\ &=\; N^{2/3}\bigl(1 + O(N^{-1/3})\bigr) \;=\; N^{2/3} + O(N^{1/3}) \end{aligned}$$

by (9.38) and (9.26). Similarly

$$\begin{aligned} \lfloor N/K \rfloor \;&=\; N^{1-1/3}\bigl(1 + O(N^{-1/3})\bigr)^{-1} + O(1) \\ &=\; N^{2/3}\bigl(1 + O(N^{-1/3})\bigr) + O(1) \;=\; N^{2/3} + O(N^{1/3})\,. \end{aligned}$$

It follows that the number of winning positions is

$$\begin{aligned} W \;&=\; N^{2/3} + O(N^{1/3}) + \tfrac{1}{2}\bigl(N^{2/3} + O(N^{1/3})\bigr) + O(N^{1/3}) + O(1) \\ &=\; \tfrac{3}{2}N^{2/3} + O(N^{1/3})\,. \end{aligned} \tag{9.39}$$

Notice how the O terms absorb one another until only one remains; this is typical, and it illustrates why O-notation is useful in the middle of a formula.

Problem 2: Perturbation of Stirling's formula.

Stirling's approximation for $n!$ is undoubtedly the most famous asymptotic formula of all. We will prove it later in this chapter; for now, let's just try to get better acquainted with its properties. We can write one version of the approximation in the form

$$n! \;=\; \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\frac{a}{n}+\frac{b}{n^2}+O(n^{-3})\right), \qquad \text{as } n\to\infty, \tag{9.40}$$

for certain constants a and b. Since this holds for all large n, it must also be asymptotically true when n is replaced by $n-1$:

$$\begin{aligned}(n-1)! \;&=\; \sqrt{2\pi(n-1)}\left(\frac{n-1}{e}\right)^{n-1}\\ &\quad\times\left(1+\frac{a}{n-1}+\frac{b}{(n-1)^2}+O\big((n-1)^{-3}\big)\right).\end{aligned} \tag{9.41}$$

We know, of course, that $(n-1)! = n!/n$; hence the right-hand side of this formula must simplify to the right-hand side of (9.40), divided by n.

Let us therefore try to simplify (9.41). The first factor becomes tractable if we pull out the large part:

$$\begin{aligned}\sqrt{2\pi(n-1)} \;&=\; \sqrt{2\pi n}\,(1-n^{-1})^{1/2}\\ &=\; \sqrt{2\pi n}\left(1-\frac{1}{2n}-\frac{1}{8n^2}+O(n^{-3})\right).\end{aligned}$$

Equation (9.35) has been used here.

Similarly we have

$$\begin{aligned}\frac{a}{n-1} \;&=\; \frac{a}{n}(1-n^{-1})^{-1} \;=\; \frac{a}{n}+\frac{a}{n^2}+O(n^{-3});\\ \frac{b}{(n-1)^2} \;&=\; \frac{b}{n^2}(1-n^{-1})^{-2} \;=\; \frac{b}{n^2}+O(n^{-3});\\ O\big((n-1)^{-3}\big) \;&=\; O\big(n^{-3}(1-n^{-1})^{-3}\big) \;=\; O(n^{-3}).\end{aligned}$$

The only thing in (9.41) that's slightly tricky to deal with is the factor $(n-1)^{n-1}$, which equals

$$n^{n-1}(1-n^{-1})^{n-1} \;=\; n^{n-1}(1-n^{-1})^n\big(1+n^{-1}+n^{-2}+O(n^{-3})\big).$$

(We are expanding everything out until we get a relative error of $O(n^{-3})$, because the relative error of a product is the sum of the relative errors of the individual factors. All of the $O(n^{-3})$ terms will coalesce.)

In order to expand $(1-n^{-1})^n$, we first compute $\ln(1-n^{-1})$ and then form the exponential, $e^{n\ln(1-n^{-1})}$:

$$\begin{aligned}
(1-n^{-1})^n &= \exp\bigl(n\ln(1-n^{-1})\bigr)\\
&= \exp\bigl(n\bigl(-n^{-1}-\tfrac{1}{2}n^{-2}-\tfrac{1}{3}n^{-3}+O(n^{-4})\bigr)\bigr)\\
&= \exp\bigl(-1-\tfrac{1}{2}n^{-1}-\tfrac{1}{3}n^{-2}+O(n^{-3})\bigr)\\
&= \exp(-1)\cdot\exp(-\tfrac{1}{2}n^{-1})\cdot\exp(-\tfrac{1}{3}n^{-2})\cdot\exp\bigl(O(n^{-3})\bigr)\\
&= \exp(-1)\cdot\bigl(1-\tfrac{1}{2}n^{-1}+\tfrac{1}{8}n^{-2}+O(n^{-3})\bigr)\\
&\qquad\cdot\bigl(1-\tfrac{1}{3}n^{-2}+O(n^{-4})\bigr)\cdot\bigl(1+O(n^{-3})\bigr)\\
&= e^{-1}\bigl(1-\tfrac{1}{2}n^{-1}-\tfrac{5}{24}n^{-2}+O(n^{-3})\bigr)\,.
\end{aligned}$$

Here we use the notation $\exp z$ instead of e^z, since it allows us to work with a complicated exponent on the main line of the formula instead of in the superscript position. We must expand $\ln(1-n^{-1})$ with absolute error $O(n^{-4})$ in order to end with a relative error of $O(n^{-3})$, because the logarithm is being multiplied by n.

The right-hand side of (9.41) has now been reduced to $\sqrt{2\pi n}$ times n^{n-1}/e^n times a product of several factors:

$$\begin{aligned}
&\bigl(1-\tfrac{1}{2}n^{-1}-\tfrac{1}{8}n^{-2}+O(n^{-3})\bigr)\\
&\quad\cdot\bigl(1+n^{-1}+n^{-2}+O(n^{-3})\bigr)\\
&\quad\cdot\bigl(1-\tfrac{1}{2}n^{-1}-\tfrac{5}{24}n^{-2}+O(n^{-3})\bigr)\\
&\quad\cdot\bigl(1+an^{-1}+(a+b)n^{-2}+O(n^{-3})\bigr)\,.
\end{aligned}$$

Multiplying these out and absorbing all asymptotic terms into one $O(n^{-3})$ yields

$$1+an^{-1}+(a+b-\tfrac{1}{12})n^{-2}+O(n^{-3})\,.$$

Hmmm; we were hoping to get $1+an^{-1}+bn^{-2}+O(n^{-3})$, since that's what we need to match the right-hand side of (9.40). Has something gone awry? No, everything is fine, provided that $a+b-\frac{1}{12}=b$.

This perturbation argument doesn't prove the validity of Stirling's approximation, but it does prove something: It proves that formula (9.40) cannot be valid unless $a=\frac{1}{12}$. If we had replaced the $O(n^{-3})$ in (9.40) by $cn^{-3}+O(n^{-4})$ and carried out our calculations to a relative error of $O(n^{-4})$, we could have deduced that b must be $\frac{1}{288}$, as claimed in Table 452. (This is not the easiest way to determine the values of a and b, but it works.)

Problem 3: The nth prime number.

Equation (9.31) is an asymptotic formula for $\pi(n)$, the number of primes that do not exceed n. If we replace n by $p = P_n$, the nth prime number, we have $\pi(p) = n$; hence

$$n \;=\; \frac{p}{\ln p} + O\Bigl(\frac{p}{(\log p)^2}\Bigr) \tag{9.42}$$

as $n \to \infty$. Let us try to "solve" this equation for p; then we will know the approximate size of the nth prime.

The first step is to simplify the O term. If we divide both sides by $p/\ln p$, we find that $n \ln p/p \to 1$; hence $p/\ln p = O(n)$ and

$$O\Bigl(\frac{p}{(\log p)^2}\Bigr) \;=\; O\Bigl(\frac{n}{\log p}\Bigr) \;=\; O\Bigl(\frac{n}{\log n}\Bigr)\,.$$

(We have $(\log p)^{-1} \leqslant (\log n)^{-1}$ because $p \geqslant n$.)

The second step is to transpose the two sides of (9.42), except for the O term. This is legal because of the general rule

$$a_n \;=\; b_n + O\bigl(f(n)\bigr) \quad\Longleftrightarrow\quad b_n \;=\; a_n + O\bigl(f(n)\bigr)\,. \tag{9.43}$$

(Each of these equations follows from the other if we multiply both sides by -1 and then add $a_n + b_n$ to both sides.) Hence

$$\frac{p}{\ln p} \;=\; n + O\Bigl(\frac{n}{\log n}\Bigr) \;=\; n\bigl(1 + O(1/\log n)\bigr)\,,$$

and we have

$$p \;=\; n \ln p\bigl(1 + O(1/\log n)\bigr)\,. \tag{9.44}$$

This is an "approximate recurrence" for $p = P_n$ in terms of itself. Our goal is to change it into an "approximate closed form," and we can do this by unfolding the recurrence asymptotically. So let's try to unfold (9.44).

By taking logarithms of both sides we deduce that

$$\ln p \;=\; \ln n + \ln\ln p + O(1/\log n)\,. \tag{9.45}$$

This value can be substituted for $\ln p$ in (9.44), but we would like to get rid of all p's on the right before making the substitution. Somewhere along the line, that last p must disappear; we can't get rid of it in the normal way for recurrences, because (9.44) doesn't specify initial conditions for small p.

One way to do the job is to start by proving the weaker result $p = O(n^2)$. This follows if we square (9.44) and divide by pn^2,

$$\frac{p}{n^2} \;=\; \frac{(\ln p)^2}{p}\bigl(1 + O(1/\log n)\bigr)\,,$$

since the right side approaches zero as $n \to \infty$. OK, we know that $p = O(n^2)$; therefore $\log p = O(\log n)$ and $\log\log p = O(\log\log n)$. We can now conclude from (9.45) that

$$\ln p \;=\; \ln n + O(\log\log n)\,;$$

in fact, with this new estimate in hand we can conclude that $\ln\ln p = \ln\ln n + O(\log\log n/\log n)$, and (9.45) now yields

$$\ln p \;=\; \ln n + \ln\ln n + O(\log\log n/\log n)\,.$$

And we can plug this into the right-hand side of (9.44), obtaining

$$p \;=\; n\ln n + n\ln\ln n + O(n)\,.$$

This is the approximate size of the nth prime.

We can refine this estimate by using a better approximation of $\pi(n)$ in place of (9.42). The next term of (9.31) tells us that

$$n \;=\; \frac{p}{\ln p} + \frac{p}{(\ln p)^2} + O\Bigl(\frac{p}{(\log p)^3}\Bigr)\,; \tag{9.46}$$

Get out the scratch paper again, gang.

proceeding as before, we obtain the recurrence

Boo, Hiss.

$$p \;=\; n\ln p\,\bigl(1 + (\ln p)^{-1}\bigr)^{-1}\bigl(1 + O(1/\log n)^2\bigr)\,, \tag{9.47}$$

which has a relative error of $O(1/\log n)^2$ instead of $O(1/\log n)$. Taking logarithms and retaining proper accuracy (but not too much) now yields

$$\begin{aligned}
\ln p \;&=\; \ln n + \ln\ln p + O(1/\log n)\\
&=\; \ln n\Bigl(1 + \frac{\ln\ln p}{\ln n} + O(1/\log n)^2\Bigr)\,;\\
\ln\ln p \;&=\; \ln\ln n + \frac{\ln\ln n}{\ln n} + O\Bigl(\frac{\log\log n}{\log n}\Bigr)^2\,.
\end{aligned}$$

Finally we substitute these results into (9.47) and our answer finds its way out:

$$P_n \;=\; n\ln n + n\ln\ln n - n + n\frac{\ln\ln n}{\ln n} + O\Bigl(\frac{n}{\log n}\Bigr)\,. \tag{9.48}$$

For example, when $n = 10^6$ this estimate comes to $15631363.8 + O(n/\log n)$; the millionth prime is actually 15485863. Exercise 21 shows that a still more accurate approximation to P_n results if we begin with a still more accurate approximation to $\pi(n)$ in place of (9.46).

Problem 4: A sum from an old final exam.

When Concrete Mathematics was first taught at Stanford University during the 1970–1971 term, students were asked for the asymptotic value of the sum

$$S_n = \frac{1}{n^2+1} + \frac{1}{n^2+2} + \cdots + \frac{1}{n^2+n}, \tag{9.49}$$

with an absolute error of $O(n^{-7})$. Let's imagine that we've just been given this problem on a (take-home) final; what is our first instinctive reaction?

No, we don't panic. Our first reaction is to THINK BIG. If we set $n = 10^{100}$, say, and look at the sum, we see that it consists of n terms, each of which is slightly less than $1/n^2$; hence the sum is slightly less than $1/n$. In general, we can usually get a decent start on an asymptotic problem by taking stock of the situation and getting a ballpark estimate of the answer.

Let's try to improve the rough estimate by pulling out the largest part of each term. We have

$$\frac{1}{n^2+k} = \frac{1}{n^2(1+k/n^2)} = \frac{1}{n^2}\left(1 - \frac{k}{n^2} + \frac{k^2}{n^4} - \frac{k^3}{n^6} + O\left(\frac{k^4}{n^8}\right)\right),$$

and so it's natural to try summing all these approximations:

$$\begin{aligned}
\frac{1}{n^2+1} &= \frac{1}{n^2} - \frac{1}{n^4} + \frac{1^2}{n^6} - \frac{1^3}{n^8} + O\left(\frac{1^4}{n^{10}}\right)\\
\frac{1}{n^2+2} &= \frac{1}{n^2} - \frac{2}{n^4} + \frac{2^2}{n^6} - \frac{2^3}{n^8} + O\left(\frac{2^4}{n^{10}}\right)\\
&\vdots\\
\frac{1}{n^2+n} &= \frac{1}{n^2} - \frac{n}{n^4} + \frac{n^2}{n^6} - \frac{n^3}{n^8} + O\left(\frac{n^4}{n^{10}}\right)\\
\hline
S_n &= \frac{n}{n^2} - \frac{n(n+1)}{2n^4} + \cdots .
\end{aligned}$$

It looks as if we're getting $S_n = n^{-1} - \frac{1}{2}n^{-2} + O(n^{-3})$, based on the sums of the first two columns; but the calculations are getting hairy.

If we persevere in this approach, we will ultimately reach the goal; but we won't bother to sum the other columns, for two reasons: First, the last column is going to give us terms that are $O(n^{-6})$, when $n/2 \leqslant k \leqslant n$, so we will have an error of $O(n^{-5})$; that's too big, and we will have to include yet another column in the expansion. Could the exam-giver have been so sadistic? We suspect that there must be a better way. Second, there is indeed a much better way, staring us right in the face.

Do pajamas have buttons?

Namely, we know a closed form for S_n: It's just $H_{n^2+n} - H_{n^2}$. And we know a good approximation for harmonic numbers, so we just apply it twice:

$$\begin{aligned} H_{n^2+n} &= \ln(n^2+n) + \gamma + \frac{1}{2(n^2+n)} - \frac{1}{12(n^2+n)^2} + O\Bigl(\frac{1}{n^8}\Bigr); \\ H_{n^2} &= \ln n^2 + \gamma + \frac{1}{2n^2} - \frac{1}{12n^4} + O\Bigl(\frac{1}{n^8}\Bigr). \end{aligned}$$

Now we can pull out large terms and simplify, as we did when looking at Stirling's approximation. We have

$$\begin{aligned} \ln(n^2+n) &= \ln n^2 + \ln\Bigl(1+\frac{1}{n}\Bigr) = \ln n^2 + \frac{1}{n} - \frac{1}{2n^2} + \frac{1}{3n^3} - \cdots; \\ \frac{1}{n^2+n} &= \frac{1}{n^2} - \frac{1}{n^3} + \frac{1}{n^4} - \cdots; \\ \frac{1}{(n^2+n)^2} &= \frac{1}{n^4} - \frac{2}{n^5} + \frac{3}{n^6} - \cdots. \end{aligned}$$

So there's lots of helpful cancellation, and we find

$$\begin{aligned} S_n = n^{-1} &- \tfrac12 n^{-2} + \tfrac13 n^{-3} - \tfrac14 n^{-4} + \tfrac15 n^{-5} - \tfrac16 n^{-6} \\ &- \tfrac12 n^{-3} + \tfrac12 n^{-4} - \tfrac12 n^{-5} + \tfrac12 n^{-6} \\ &+ \tfrac16 n^{-5} - \tfrac14 n^{-6} \end{aligned}$$

plus terms that are $O(n^{-7})$. A bit of arithmetic and we're home free:

$$S_n = n^{-1} - \tfrac12 n^{-2} - \tfrac16 n^{-3} + \tfrac14 n^{-4} - \tfrac{2}{15} n^{-5} + \tfrac{1}{12} n^{-6} + O(n^{-7}). \quad (9.50)$$

It would be nice if we could check this answer numerically, as we did when we derived exact results in earlier chapters. Asymptotic formulas are harder to verify; an arbitrarily large constant may be hiding in a O term, so any numerical test is inconclusive. But in practice, we have no reason to believe that an adversary is trying to trap us, so we can assume that the unknown O-constants are reasonably small. With a pocket calculator we find that $S_4 = \frac{1}{17} + \frac{1}{18} + \frac{1}{19} + \frac{1}{20} = 0.2170107$; and our asymptotic estimate when $n = 4$ comes to

$$\tfrac14\bigl(1 + \tfrac14(-\tfrac12 + \tfrac14(-\tfrac16 + \tfrac14(\tfrac14 + \tfrac14(-\tfrac{2}{15} + \tfrac14\cdot\tfrac{1}{12}))))\bigr) = 0.2170125.$$

If we had made an error of, say, $\frac{1}{12}$ in the term for n^{-6}, a difference of $\frac{1}{12}\frac{1}{4096}$ would have shown up in the fifth decimal place; so our asymptotic answer is probably correct.

Problem 5: An infinite sum.

We turn now to an asymptotic question posed by Solomon Golomb [152]: What is the approximate value of

$$S_n = \sum_{k\geqslant 1} \frac{1}{kN_n(k)^2}\,, \tag{9.51}$$

where $N_n(k)$ is the number of digits required to write k in radix n notation?

First let's try again for a ballpark estimate. The number of digits, $N_n(k)$, is approximately $\log_n k = \log k/\log n$; so the terms of this sum are roughly $(\log n)^2/k(\log k)^2$. Summing on k gives $\approx (\log n)^2 \sum_{k\geqslant 2} 1/k(\log k)^2$, and this sum converges to a constant value because it can be compared to the integral

$$\int_2^\infty \frac{dx}{x(\ln x)^2} = -\frac{1}{\ln x}\bigg|_2^\infty = \frac{1}{\ln 2}\,.$$

Therefore we expect S_n to be about $C(\log n)^2$, for some constant C.

Hand-wavy analyses like this are useful for orientation, but we need better estimates to solve the problem. One idea is to express $N_n(k)$ exactly:

$$N_n(k) = \lfloor \log_n k \rfloor + 1\,. \tag{9.52}$$

Thus, for example, k has three radix n digits when $n^2 \leqslant k < n^3$, and this happens precisely when $\lfloor \log_n k \rfloor = 2$. It follows that $N_n(k) > \log_n k$, hence $S_n = \sum_{k\geqslant 1} 1/kN_n(k)^2 < 1 + (\log n)^2 \sum_{k\geqslant 2} 1/k(\log k)^2$.

Proceeding as in Problem 1, we can try to write $N_n(k) = \log_n k + O(1)$ and substitute this into the formula for S_n. The term represented here by $O(1)$ is always between 0 and 1, and it is about $\frac{1}{2}$ on the average, so it seems rather well-behaved. But still, this isn't a good enough approximation to tell us about S_n; it gives us zero significant figures (that is, high relative error) when k is small, and these are the terms that contribute the most to the sum. We need a different idea.

The key (as in Problem 4) is to use our manipulative skills to put the sum into a more tractable form, before we resort to asymptotic estimates. We can introduce a new variable of summation, $m = N_n(k)$:

$$\begin{aligned} S_n &= \sum_{k,m\geqslant 1} \frac{[m = N_n(k)]}{km^2} \\ &= \sum_{k,m\geqslant 1} \frac{[n^{m-1} \leqslant k < n^m]}{km^2} \\ &= \sum_{m\geqslant 1} \frac{1}{m^2}\big(H_{n^m-1} - H_{n^{m-1}-1}\big)\,. \end{aligned}$$

This may look worse than the sum we began with, but it's actually a step forward, because we have very good approximations for the harmonic numbers.

Still, we hold back and try to simplify some more. No need to rush into asymptotics. Summation by parts allows us to group the terms for each value of H_{n^m-1} that we need to approximate:

$$S_n = \sum_{k\geqslant 1} H_{n^k-1}\Big(\frac{1}{k^2} - \frac{1}{(k+1)^2}\Big).$$

For example, H_{n^2-1} is multiplied by $1/2^2$ and then by $-1/3^2$. (We have used the fact that $H_{n^0-1} = H_0 = 0$.)

Now we're ready to expand the harmonic numbers. Our experience with estimating $(n-1)!$ has taught us that it will be easier to estimate H_{n^k} than H_{n^k-1}, since the (n^k-1)'s will be messy; therefore we write

$$\begin{aligned} H_{n^k-1} &= H_{n^k} - \frac{1}{n^k} = \ln n^k + \gamma + \frac{1}{2n^k} + O\Big(\frac{1}{n^{2k}}\Big) - \frac{1}{n^k} \\ &= k\ln n + \gamma - \frac{1}{2n^k} + O\Big(\frac{1}{n^{2k}}\Big). \end{aligned}$$

Our sum now reduces to

$$\begin{aligned} S_n &= \sum_{k\geqslant 1}\Big(k\ln n + \gamma - \frac{1}{2n^k} + O\Big(\frac{1}{n^{2k}}\Big)\Big)\Big(\frac{1}{k^2} - \frac{1}{(k+1)^2}\Big) \\ &= (\ln n)\Sigma_1 + \gamma\Sigma_2 - \tfrac{1}{2}\Sigma_3(n) + O\big(\Sigma_3(n^2)\big). \end{aligned} \tag{9.53}$$

There are four easy pieces left: Σ_1, Σ_2, $\Sigma_3(n)$, and $\Sigma_3(n^2)$.

Let's do the Σ_3's first, since $\Sigma_3(n^2)$ is the O term; then we'll see what sort of error we're getting. (There's no sense carrying out other calculations with perfect accuracy if they will be absorbed into a O anyway.) This sum is simply a power series,

Into a Big Oh.

$$\Sigma_3(x) = \sum_{k\geqslant 1}\Big(\frac{1}{k^2} - \frac{1}{(k+1)^2}\Big)x^{-k},$$

and the series converges when $x \geqslant 1$ so we can truncate it at any desired point. If we stop $\Sigma_3(n^2)$ at the term for $k = 1$, we get $\Sigma_3(n^2) = O(n^{-2})$; hence (9.53) has an absolute error of $O(n^{-2})$. (To decrease this absolute error, we could use a better approximation to H_{n^k}; but $O(n^{-2})$ is good enough for now.) If we truncate $\Sigma_3(n)$ at the term for $k = 2$, we get

$$\Sigma_3(n) = \tfrac{3}{4}n^{-1} + O(n^{-2});$$

this is all the accuracy we need.

We might as well do Σ_2 now, since it is so easy:

$$\Sigma_2 \;=\; \sum_{k\geqslant 1}\Big(\frac{1}{k^2}-\frac{1}{(k+1)^2}\Big)\,.$$

This is the telescoping series $(1-\frac14)+(\frac14-\frac19)+(\frac19-\frac{1}{16})+\cdots=1$.

Finally, Σ_1 gives us the leading term of S_n, the coefficient of $\ln n$ in (9.53):

$$\Sigma_1 \;=\; \sum_{k\geqslant 1} k\Big(\frac{1}{k^2}-\frac{1}{(k+1)^2}\Big)\,.$$

This is $(1-\frac14)+(\frac24-\frac29)+(\frac39-\frac{3}{16})+\cdots=\frac11+\frac14+\frac19+\cdots=H_\infty^{(2)}=\pi^2/6$. (If we hadn't applied summation by parts earlier, we would have seen directly that $S_n\sim\sum_{k\geqslant 1}(\ln n)/k^2$, because $H_{n^k-1}-H_{n^{k-1}-1}\sim\ln n$; so summation by parts didn't help us to evaluate the leading term, although it did make some of our other work easier.)

Now we have evaluated each of the Σ's in (9.53), so we can put everything together and get the answer to Golomb's problem:

$$S_n \;=\; \frac{\pi^2}{6}\ln n+\gamma-\frac{3}{8n}+O\Big(\frac{1}{n^2}\Big)\,. \tag{9.54}$$

Notice that this grows more slowly than our original hand-wavy estimate of $C(\log n)^2$. Sometimes a discrete sum fails to obey a continuous intuition.

Problem 6: Big Phi.

Near the end of Chapter 4, we observed that the number of fractions in the Farey series $\mathcal{F}_n$ is $1+\Phi(n)$, where

$$\Phi(n) \;=\; \varphi(1)+\varphi(2)+\cdots+\varphi(n)\,;$$

and we showed in (4.62) that

$$\Phi(n) \;=\; \frac12\sum_{k\geqslant 1}\mu(k)\lfloor n/k\rfloor\lfloor 1+n/k\rfloor\,. \tag{9.55}$$

Let us now try to estimate $\Phi(n)$ when n is large. (It was sums like this that led Bachmann to invent O-notation in the first place.)

Thinking BIG tells us that $\Phi(n)$ will probably be proportional to n^2. For if the final factor were just $\lfloor n/k\rfloor$ instead of $\lfloor 1+n/k\rfloor$, we would have $\big|\Phi(n)\big|\leqslant\frac12\sum_{k\geqslant 1}\lfloor n/k\rfloor^2\leqslant\frac12\sum_{k\geqslant 1}(n/k)^2=\frac{\pi^2}{12}n^2$, because the Möbius function $\mu(k)$ is either -1, 0, or $+1$. The additional '$1+$' in that final factor adds $\sum_{k\geqslant 1}\mu(k)\lfloor n/k\rfloor$; but this is zero for $k>n$, so it cannot be more than $nH_n=O(n\log n)$ in absolute value.

This preliminary analysis indicates that we'll find it advantageous to write

$$\begin{aligned}\Phi(n) &= \frac{1}{2}\sum_{k=1}^{n}\mu(k)\Big(\Big(\frac{n}{k}\Big)+O(1)\Big)^2 = \frac{1}{2}\sum_{k=1}^{n}\mu(k)\Big(\Big(\frac{n}{k}\Big)^2+O\Big(\frac{n}{k}\Big)\Big)\\ &= \frac{1}{2}\sum_{k=1}^{n}\mu(k)\Big(\frac{n}{k}\Big)^2+\sum_{k=1}^{n}O\Big(\frac{n}{k}\Big)\\ &= \frac{1}{2}\sum_{k=1}^{n}\mu(k)\Big(\frac{n}{k}\Big)^2 + O(n\log n)\,.\end{aligned}$$

This removes the floors; the remaining problem is to evaluate the unfloored sum $\frac{1}{2}\sum_{k=1}^{n}\mu(k)n^2/k^2$ with an accuracy of $O(n\log n)$; in other words, we want to evaluate $\sum_{k=1}^{n}\mu(k)1/k^2$ with an accuracy of $O(n^{-1}\log n)$. But that's easy; we can simply run the sum all the way up to $k=\infty$, because the newly added terms are

$$\begin{aligned}\sum_{k>n}\frac{\mu(k)}{k^2} &= O\Big(\sum_{k>n}\frac{1}{k^2}\Big) = O\Big(\sum_{k>n}\frac{1}{k(k-1)}\Big)\\ &= O\Big(\sum_{k>n}\Big(\frac{1}{k-1}-\frac{1}{k}\Big)\Big) = O\Big(\frac{1}{n}\Big)\,.\end{aligned}$$

We proved in (7.89) that $\sum_{k\geqslant 1}\mu(k)/k^z = 1/\zeta(z)$. Hence $\sum_{k\geqslant 1}\mu(k)/k^2 = 1/\big(\sum_{k\geqslant 1}1/k^2\big) = 6/\pi^2$, and we have our answer:

(The error term was shown to be at most $O(n(\log n)^{2/3}\times(\log\log n)^{1+\epsilon})$ by Saltykov in 1960 [316]. On the other hand, it is not as small as $o(n(\log\log n)^{1/2})$, according to Montgomery [275].)

$$\Phi(n) = \frac{3}{\pi^2}n^2 + O(n\log n)\,. \tag{9.56}$$

9.4 TWO ASYMPTOTIC TRICKS

Now that we have some facility with O manipulations, let's look at what we've done from a slightly higher perspective. Then we'll have some important weapons in our asymptotic arsenal, when we need to do battle with tougher problems.

Trick 1: Bootstrapping.

When we estimated the nth prime P_n in Problem 3 of Section 9.3, we solved an asymptotic recurrence of the form

$$P_n = n\ln P_n\big(1+O(1/\log n)\big)\,.$$

We proved that $P_n = n\ln n + O(n)$ by first using the recurrence to show the weaker result $O(n^2)$. This is a special case of a general method called *bootstrapping*, in which we solve a recurrence asymptotically by starting with

a rough estimate and plugging it into the recurrence; in this way we can often derive better and better estimates, "pulling ourselves up by our bootstraps."

Here's another problem that illustrates bootstrapping nicely: What is the asymptotic value of the coefficient $g_n = [z^n]\,G(z)$ in the generating function

$$G(z) \;=\; \exp\Bigl(\sum_{k\geqslant 1}\frac{z^k}{k^2}\Bigr)\,, \tag{9.57}$$

as $n \to \infty$? If we differentiate this equation with respect to z, we find

$$G'(z) \;=\; \sum_{n=0}^{\infty} n g_n z^{n-1} \;=\; \Bigl(\sum_{k\geqslant 1}\frac{z^{k-1}}{k}\Bigr)\,G(z)\,;$$

equating coefficients of z^{n-1} on both sides gives the recurrence

$$n g_n \;=\; \sum_{0\leqslant k<n}\frac{g_k}{n-k}\,. \tag{9.58}$$

Our problem is equivalent to finding an asymptotic formula for the solution to (9.58), with the initial condition $g_0 = 1$. The first few values

n	0	1	2	3	4	5	6
g_n	1	1	$\frac{3}{4}$	$\frac{19}{36}$	$\frac{107}{288}$	$\frac{641}{2400}$	$\frac{51103}{259200}$

don't reveal much of a pattern, and the integer sequence $\langle n!^2 g_n\rangle$ doesn't appear in Sloane's *Handbook* [330]; therefore a closed form for g_n seems out of the question, and asymptotic information is probably the best we can hope to derive.

Our first handle on this problem is the observation that $0 < g_n \leqslant 1$ for all $n \geqslant 0$; this is easy to prove by induction. So we have a start:

$$g_n \;=\; O(1)\,.$$

This equation can, in fact, be used to "prime the pump" for a bootstrapping operation: Plugging it in on the right of (9.58) yields

$$n g_n \;=\; \sum_{0\leqslant k<n}\frac{O(1)}{n-k} \;=\; H_n O(1) \;=\; O(\log n)\,;$$

hence we have

$$g_n \;=\; O\Bigl(\frac{\log n}{n}\Bigr)\,, \qquad \text{for } n > 1.$$

And we can bootstrap yet again:

$$\begin{aligned} ng_n &= \frac{1}{n} + \sum_{0<k<n} \frac{O\big((1+\log k)/k\big)}{n-k} \\ &= \frac{1}{n} + \sum_{0<k<n} \frac{O(\log n)}{k(n-k)} \\ &= \frac{1}{n} + \sum_{0<k<n} \Big(\frac{1}{k} + \frac{1}{n-k}\Big) \frac{O(\log n)}{n} \\ &= \frac{1}{n} + \frac{2}{n} H_{n-1} O(\log n) \;=\; \frac{1}{n} O(\log n)^2 , \end{aligned}$$

obtaining

$$g_n \;=\; O\Big(\frac{\log n}{n}\Big)^2 . \tag{9.59}$$

Will this go on forever? Perhaps we'll have $g_n = O(n^{-1}\log n)^m$ for all m.

Actually no; we have just reached a point of diminishing returns. The next attempt at bootstrapping involves the sum

$$\begin{aligned} \sum_{0<k<n} \frac{1}{k^2(n-k)} &= \sum_{0<k<n} \Big(\frac{1}{nk^2} + \frac{1}{n^2 k} + \frac{1}{n^2(n-k)}\Big) \\ &= \frac{1}{n} H^{(2)}_{n-1} + \frac{2}{n^2} H_{n-1} , \end{aligned}$$

which is $\Omega(n^{-1})$; so we cannot get an estimate for g_n that falls below $\Omega(n^{-2})$.

In fact, we now know enough about g_n to apply our old trick of pulling out the largest part:

$$\begin{aligned} ng_n &= \sum_{0\leqslant k<n} \frac{g_k}{n} + \sum_{0\leqslant k<n} g_k \Big(\frac{1}{n-k} - \frac{1}{n}\Big) \\ &= \frac{1}{n} \sum_{k\geqslant 0} g_k - \frac{1}{n} \sum_{k\geqslant n} g_k + \frac{1}{n} \sum_{0\leqslant k<n} \frac{k g_k}{n-k} . \end{aligned} \tag{9.60}$$

The first sum here is $G(1) = \exp(\frac{1}{1} + \frac{1}{4} + \frac{1}{9} + \cdots) = e^{\pi^2/6}$, because $G(z)$ converges for all $|z| \leqslant 1$. The second sum is the tail of the first; we can get an upper bound by using (9.59):

$$\sum_{k\geqslant n} g_k \;=\; O\Big(\sum_{k\geqslant n} \frac{(\log k)^2}{k^2}\Big) \;=\; O\Big(\frac{(\log n)^2}{n}\Big) .$$

This last estimate follows because, for example,

$$\sum_{k>n} \frac{(\log k)^2}{k^2} < \sum_{m\geqslant 1} \sum_{n^m<k\leqslant n^{m+1}} \frac{(\log n^{m+1})^2}{k(k-1)} < \sum_{m\geqslant 1} \frac{(m+1)^2(\log n)^2}{n^m}.$$

(Exercise 54 discusses a more general way to estimate such tails.)

The third sum in (9.60) is

$$O\Bigl(\sum_{0\leqslant k<n} \frac{(\log n)^2}{k(n-k)}\Bigr) = O\Bigl(\frac{(\log n)^3}{n}\Bigr),$$

by an argument that's already familiar. So (9.60) proves that

$$g_n = \frac{e^{\pi^2/6}}{n^2} + O\bigl(\log n/n\bigr)^3. \tag{9.61}$$

Finally, we can feed this formula back into the recurrence, bootstrapping once more; the result is

$$g_n = \frac{e^{\pi^2/6}}{n^2} + O(\log n/n^3). \tag{9.62}$$

(Exercise 23 peeks inside the remaining O term.)

Trick 2: Trading tails.

We derived (9.62) in somewhat the same way we derived the asymptotic value (9.56) of $\Phi(n)$: In both cases we started with a finite sum but got an asymptotic value by considering an infinite sum. We couldn't simply get the infinite sum by introducing O into the summand; we had to be careful to use one approach when k was small and another when k was large.

Those derivations were special cases of an important three-step asymptotic summation method we will now discuss in greater generality. Whenever we want to estimate the value of $\sum_k a_k(n)$, we can try the following approach:

(This important method was pioneered by Laplace [240].)

1 First break the sum into two disjoint ranges, D_n and T_n. The summation over D_n should be the "dominant" part, in the sense that it includes enough terms to determine the significant digits of the sum, when n is large. The summation over the other range T_n should be just the "tail" end, which contributes little to the overall total.

2 Find an asymptotic estimate

$$a_k(n) = b_k(n) + O\bigl(c_k(n)\bigr)$$

that is valid when $k \in D_n$. The O bound need not hold when $k \in T_n$.

3 Now prove that each of the following three sums is small:

$$\Sigma_a(n) = \sum_{k\in T_n} a_k(n)\,; \qquad \Sigma_b(n) = \sum_{k\in T_n} b_k(n)\,;$$
$$\Sigma_c(n) = \sum_{k\in D_n} \bigl|c_k(n)\bigr|\,. \tag{9.63}$$

If all three steps can be completed successfully, we have a good estimate:

$$\sum_{k\in D_n\cup T_n} a_k(n) = \sum_{k\in D_n\cup T_n} b_k(n) + O\bigl(\Sigma_a(n)\bigr) + O\bigl(\Sigma_b(n)\bigr) + O\bigl(\Sigma_c(n)\bigr)\,.$$

Here's why. We can "chop off" the tail of the given sum, getting a good estimate in the range D_n where a good estimate is necessary:

$$\sum_{k\in D_n} a_k(n) = \sum_{k\in D_n} \bigl(b_k(n) + O(c_k(n))\bigr) = \sum_{k\in D_n} b_k(n) + O\bigl(\Sigma_c(n)\bigr)\,.$$

And we can replace the tail with another one, even though the new tail might be a terrible approximation to the old, because the tails don't really matter:

$$\begin{aligned}\sum_{k\in T_n} a_k(n) &= \sum_{k\in T_n} \bigl(b_k(n) - b_k(n) + a_k(n)\bigr)\\ &= \sum_{k\in T_n} b_k(n) + O\bigl(\Sigma_b(n)\bigr) + O\bigl(\Sigma_a(n)\bigr)\,.\end{aligned}$$

Asymptotics is the art of knowing where to be sloppy and where to be precise.

When we evaluated the sum in (9.60), for example, we had

$$\begin{aligned}a_k(n) &= [0\leqslant k<n]g_k/(n-k)\,,\\ b_k(n) &= g_k/n\,,\\ c_k(n) &= kg_k/n(n-k)\,;\end{aligned}$$

the ranges of summation were

$$D_n = \{0,1,\ldots,n-1\}, \qquad T_n = \{n,n+1,\ldots\};$$

and we found that

$$\Sigma_a(n) = 0\,, \quad \Sigma_b(n) = O\bigl((\log n)^2/n^2\bigr)\,, \quad \Sigma_c(n) = O\bigl((\log n)^3/n^2\bigr)\,.$$

This led to (9.61).

Similarly, when we estimated $\Phi(n)$ in (9.55) we had

$$\begin{aligned}&a_k(n) = \mu(k)\lfloor n/k\rfloor\lfloor 1+n/k\rfloor\,, \quad b_k(n) = \mu(k)n^2/k^2\,, \quad c_k(n) = n/k;\\ &D_n = \{1,2,\ldots,n\}, \qquad T_n = \{n+1,n+2,\ldots\}.\end{aligned}$$

We derived (9.56) by observing that $\Sigma_a(n)=0$, $\Sigma_b(n)=O(n)$, and $\Sigma_c(n)=O(n\log n)$.

Also, horses switch their tails when feeding time approaches.

Here's another example where tail switching is effective. (Unlike our previous examples, this one illustrates the trick in its full generality, with $\Sigma_a(n) \neq 0$.) We seek the asymptotic value of

$$L_n = \sum_{k\geqslant 0} \frac{\ln(n+2^k)}{k!}\,.$$

The big contributions to this sum occur when k is small, because of the $k!$ in the denominator. In this range we have

$$\ln(n+2^k) = \ln n + \frac{2^k}{n} - \frac{2^{2k}}{2n^2} + O\Bigl(\frac{2^{3k}}{n^3}\Bigr)\,. \tag{9.64}$$

We can prove that this estimate holds for $0 \leqslant k < \lfloor \lg n \rfloor$, since the original terms that have been truncated with O are bounded by the convergent series

$$\sum_{m\geqslant 3} \frac{2^{km}}{mn^m} \leqslant \frac{2^{3k}}{n^3} \sum_{m\geqslant 3} \frac{2^{k(m-3)}}{n^{m-3}} \leqslant \frac{2^{3k}}{n^3}\Bigl(1+\frac{1}{2}+\frac{1}{4}+\cdots\Bigr) = \frac{2^{3k}}{n^3}\cdot 2\,.$$

(In this range, $2^k/n \leqslant 2^{\lfloor \lg n\rfloor - 1}/n \leqslant \frac{1}{2}$.)

Therefore we can apply the three-step method just described, with

$$\begin{aligned} a_k(n) &= \ln(n+2^k)/k!\,,\\ b_k(n) &= (\ln n + 2^k/n - 4^k/2n^2)/k!\,,\\ c_k(n) &= 8^k/n^3k!\,;\\ D_n &= \{0,1,\ldots,\lfloor \lg n\rfloor - 1\}\,,\\ T_n &= \{\lfloor \lg n\rfloor, \lfloor \lg n\rfloor + 1, \ldots\}\,. \end{aligned}$$

All we have to do is find good bounds on the three Σ's in (9.63), and we'll know that $\sum_{k\geqslant 0} a_k(n) \approx \sum_{k\geqslant 0} b_k(n)$.

The error we have committed in the dominant part of the sum, $\Sigma_c(n) = \sum_{k\in D_n} 8^k/n^3k!$, is obviously bounded by $\sum_{k\geqslant 0} 8^k/n^3k! = e^8/n^3$, so it can be replaced by $O(n^{-3})$. The new tail error is

$$\begin{aligned} \bigl|\Sigma_b(n)\bigr| &= \Bigl|\sum_{k\geqslant \lfloor \lg n\rfloor} b_k(n)\Bigr| \\ &< \sum_{k\geqslant \lfloor \lg n\rfloor} \frac{\ln n + 2^k + 4^k}{k!} \\ &< \frac{\ln n + 2^{\lfloor \lg n\rfloor} + 4^{\lfloor \lg n\rfloor}}{\lfloor \lg n\rfloor!} \sum_{k\geqslant 0} \frac{4^k}{k!} = O\Bigl(\frac{n^2}{\lfloor \lg n\rfloor!}\Bigr)\,. \end{aligned}$$

"We may not be big, but we're small."

Since $\lfloor \lg n \rfloor !$ grows faster than any power of n, this minuscule error is overwhelmed by $\Sigma_c(n) = O(n^{-3})$. The error that comes from the original tail,

$$\Sigma_a(n) \;=\; \sum_{k \geqslant \lfloor \lg n \rfloor} a_k(n) \;<\; \sum_{k \geqslant \lfloor \lg n \rfloor} \frac{k + \ln n}{k!}\,,$$

is smaller yet.

Finally, it's easy to sum $\sum_{k \geqslant 0} b_k(n)$ in closed form, and we have obtained the desired asymptotic formula:

$$\sum_{k \geqslant 0} \frac{\ln(n + 2^k)}{k!} \;=\; e \ln n + \frac{e^2}{n} - \frac{e^4}{2n^2} + O\Bigl(\frac{1}{n^3}\Bigr)\,. \tag{9.65}$$

The method we've used makes it clear that, in fact,

$$\sum_{k \geqslant 0} \frac{\ln(n + 2^k)}{k!} \;=\; e \ln n + \sum_{k=1}^{m-1} (-1)^{k+1} \frac{e^{2^k}}{k n^k} + O\Bigl(\frac{1}{n^m}\Bigr)\,, \tag{9.66}$$

for any fixed $m > 0$. (This is a truncation of a series that diverges for all fixed n if we let $m \to \infty$.)

There's only one flaw in our solution: We were too cautious. We derived (9.64) on the assumption that $k < \lfloor \lg n \rfloor$, but exercise 53 proves that the stated estimate is actually valid for all values of k. If we had known the stronger general result, we wouldn't have had to use the two-tail trick; we could have gone directly to the final formula! But later we'll encounter problems where exchange of tails is the only decent approach available.

9.5 EULER'S SUMMATION FORMULA

And now for our next trick — which is, in fact, the last important technique that will be discussed in this book — we turn to a general method of approximating sums that was first published by Leonhard Euler [101] in 1732. (The idea is sometimes also associated with the name of Colin Maclaurin, a professor of mathematics at Edinburgh who discovered it independently a short time later [263, page 305].)

Here's the formula:

$$\sum_{a \leqslant k < b} f(k) \;=\; \int_a^b f(x)\,dx \;+\; \sum_{k=1}^{m} \frac{B_k}{k!} f^{(k-1)}(x)\Bigg|_a^b \;+\; R_m\,, \tag{9.67}$$

$$\text{where } R_m \;=\; (-1)^{m+1} \int_a^b \frac{B_m(\{x\})}{m!} f^{(m)}(x)\,dx\,, \quad \begin{array}{l}\text{integers } a \leqslant b;\\ \text{integer } m \geqslant 1.\end{array} \tag{9.68}$$

On the left is a typical sum that we might want to evaluate. On the right is another expression for that sum, involving integrals and derivatives. If $f(x)$ is a sufficiently "smooth" function, it will have m derivatives $f'(x), \ldots, f^{(m)}(x)$, and this formula turns out to be an identity. The right-hand side is often an excellent approximation to the sum on the left, in the sense that the remainder R_m is often small. For example, we'll see that Stirling's approximation for $n!$ is a consequence of Euler's summation formula; so is our asymptotic approximation for the harmonic number H_n.

The numbers B_k in (9.67) are the Bernoulli numbers that we met in Chapter 6; the function $B_m(\{x\})$ in (9.68) is the Bernoulli polynomial that we met in Chapter 7. The notation $\{x\}$ stands for the fractional part $x - \lfloor x \rfloor$, as in Chapter 3. Euler's summation formula sort of brings everything together.

Let's recall the values of small Bernoulli numbers, since it's always handy to have them listed near Euler's general formula:

$$\begin{aligned} &B_0 = 1\,,\ B_1 = -\tfrac{1}{2}\,,\ B_2 = \tfrac{1}{6}\,,\ B_4 = -\tfrac{1}{30}\,,\ B_6 = \tfrac{1}{42}\,,\ B_8 = -\tfrac{1}{30}\,; \\ &B_3 = B_5 = B_7 = B_9 = B_{11} = \cdots = 0\,. \end{aligned}$$

Jakob Bernoulli discovered these numbers when studying the sums of powers of integers, and Euler's formula explains why: If we set $f(x) = x^{m-1}$, we have $f^{(m)}(x) = 0$; hence $R_m = 0$, and (9.67) reduces to

$$\begin{aligned} \sum_{a \leqslant k < b} k^{m-1} &= \left.\frac{x^m}{m}\right|_a^b + \sum_{k=1}^{m} \frac{B_k}{k!}\,(m-1)^{\underline{k-1}}\,x^{m-k}\Bigg|_a^b \\ &= \frac{1}{m}\sum_{k=0}^{m}\binom{m}{k}B_k \cdot (b^{m-k} - a^{m-k})\,. \end{aligned}$$

For example, when $m = 3$ we have our favorite example of summation:

$$\sum_{0 \leqslant k < n} k^2 = \frac{1}{3}\left(\binom{3}{0}B_0 n^3 + \binom{3}{1}B_1 n^2 + \binom{3}{2}B_2 n\right) = \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}\,.$$

(This is the last time we shall derive this famous formula in this book.)

All good things must come to an end.

Before we prove Euler's formula, let's look at a high-level reason (due to Lagrange [234]) why such a formula ought to exist. Chapter 2 defines the difference operator Δ and explains that $\sum$ is the inverse of Δ, just as $\int$ is the inverse of the derivative operator D. We can express Δ in terms of D using Taylor's formula as follows:

$$f(x+\epsilon) = f(x) + \frac{f'(x)}{1!}\epsilon + \frac{f''(x)}{2!}\epsilon^2 + \cdots\,.$$

Setting $\epsilon = 1$ tells us that

$$\begin{aligned}\Delta f(x) &= f(x+1) - f(x)\\ &= f'(x)/1! + f''(x)/2! + f'''(x)/3! + \cdots\\ &= (D/1! + D^2/2! + D^3/3! + \cdots)\, f(x) \;=\; (e^D - 1)\, f(x)\,. \qquad (9.69)\end{aligned}$$

Here e^D stands for the differential operation $1 + D/1! + D^2/2! + D^3/3! + \cdots$. Since $\Delta = e^D - 1$, the inverse operator $\Sigma = 1/\Delta$ should be $1/(e^D - 1)$; and we know from Table 352 that $z/(e^z - 1) = \sum_{k\geqslant 0} B_k z^k/k!$ is a power series involving Bernoulli numbers. Thus

$$\sum \;=\; \frac{B_0}{D} + \frac{B_1}{1!} + \frac{B_2}{2!}D + \frac{B_3}{3!}D^2 + \cdots \;=\; \int + \sum_{k\geqslant 1} \frac{B_k}{k!} D^{k-1}\,. \qquad (9.70)$$

Applying this operator equation to $f(x)$ and attaching limits yields

$$\sum\nolimits_a^b f(x)\,\delta x \;=\; \int_a^b f(x)\,dx + \sum_{k\geqslant 1} \frac{B_k}{k!} f^{(k-1)}(x)\Big|_a^b\,, \qquad (9.71)$$

which is exactly Euler's summation formula (9.67) without the remainder term. (Euler did not, in fact, consider the remainder, nor did anybody else until S. D. Poisson [295] published an important memoir about approximate summation in 1823. The remainder term is important, because the infinite sum $\sum_{k\geqslant 1}(B_k/k!)f^{(k-1)}(x)\big|_a^b$ often diverges. Our derivation of (9.71) has been purely formal, without regard to convergence.)

Now let's prove (9.67), with the remainder included. It suffices to prove the case $a = 0$ and $b = 1$, namely

$$f(0) \;=\; \int_0^1 f(x)\,dx + \sum_{k=1}^{m} \frac{B_k}{k!} f^{(k-1)}(x)\Big|_0^1 - (-1)^m \int_0^1 \frac{B_m(x)}{m!} f^{(m)}(x)\,dx\,,$$

because we can then replace $f(x)$ by $f(x+l)$ for any integer l, getting

$$f(l) \;=\; \int_l^{l+1} f(x)\,dx + \sum_{k=1}^{m} \frac{B_k}{k!} f^{(k-1)}(x)\Big|_l^{l+1} - (-1)^m \int_l^{l+1} \frac{B_m\big(\{x\}\big)}{m!} f^{(m)}(x)\,dx\,.$$

The general formula (9.67) is just the sum of this identity over the range $a \leqslant l < b$, because intermediate terms telescope nicely.

The proof when $a = 0$ and $b = 1$ is by induction on m, starting with $m = 1$:

$$f(0) \;=\; \int_0^1 f(x)\,dx - \frac{1}{2}\big(f(1) - f(0)\big) + \int_0^1 (x - \tfrac{1}{2}) f'(x)\,dx\,.$$

(The Bernoulli polynomial $B_m(x)$ is defined by the equation

$$B_m(x) \;=\; \binom{m}{0}B_0x^m + \binom{m}{1}B_1x^{m-1} + \cdots + \binom{m}{m}B_mx^0 \tag{9.72}$$

in general, hence $B_1(x) = x - \frac{1}{2}$ in particular.) In other words, we want to prove that

$$\frac{f(0)+f(1)}{2} \;=\; \int_0^1 f(x)\,dx + \int_0^1 (x-\tfrac{1}{2})f'(x)\,dx\,.$$

But this is just a special case of the formula

$$u(x)v(x)\Big|_0^1 \;=\; \int_0^1 u(x)\,dv(x) \;+\; \int_0^1 v(x)\,du(x) \tag{9.73}$$

for integration by parts, with $u(x) = f(x)$ and $v(x) = x - \frac{1}{2}$. Hence the case $m = 1$ is easy.

To pass from $m-1$ to m and complete the induction when $m > 1$, we need to show that $R_{m-1} = (B_m/m!)f^{(m-1)}(x)\big|_0^1 + R_m$, namely that

$$\begin{aligned}(-1)^m &\int_0^1 \frac{B_{m-1}(x)}{(m-1)!}\,f^{(m-1)}(x)\,dx \\ &= \frac{B_m}{m!}\,f^{(m-1)}(x)\Big|_0^1 - (-1)^m\int_0^1 \frac{B_m(x)}{m!}\,f^{(m)}(x)\,dx\,.\end{aligned}$$

This reduces to the equation

$$(-1)^mB_mf^{(m-1)}(x)\Big|_0^1 \;=\; m\int_0^1 B_{m-1}(x)f^{(m-1)}(x)\,dx + \int_0^1 B_m(x)f^{(m)}(x)\,dx\,.$$

Once again (9.73) applies to these two integrals, with $u(x) = f^{(m-1)}(x)$ and $v(x) = B_m(x)$, because the derivative of the Bernoulli polynomial (9.72) is

Will the authors never get serious?

$$\begin{aligned}\frac{d}{dx}\sum_k \binom{m}{k}B_kx^{m-k} \;&=\; \sum_k \binom{m}{k}(m-k)B_kx^{m-k-1} \\ &=\; m\sum_k \binom{m-1}{k}B_kx^{m-1-k} \;=\; mB_{m-1}(x)\,. \qquad (9.74)\end{aligned}$$

(The absorption identity (5.7) was useful here.) Therefore the required formula will hold if and only if

$$(-1)^mB_mf^{(m-1)}(x)\big|_0^1 \;=\; B_m(x)f^{(m-1)}(x)\big|_0^1\,.$$

In other words, we need to have

$$(-1)^m B_m \;=\; B_m(1) \;=\; B_m(0)\,, \qquad \text{for } m > 1. \tag{9.75}$$

This is a bit embarrassing, because $B_m(0)$ is obviously equal to B_m, not to $(-1)^m B_m$. But there's no problem really, because $m > 1$; we know that B_m is zero when m is odd. (Still, that was a close call.)

To complete the proof of Euler's summation formula we need to show that $B_m(1) = B_m(0)$, which is the same as saying that

$$\sum_k \binom{m}{k} B_k \;=\; B_m\,, \qquad \text{for } m > 1.$$

But this is just the definition of Bernoulli numbers, (6.79), so we're done.

The identity $B'_m(x) = mB_{m-1}(x)$ implies that

$$\int_0^1 B_m(x)\,dx \;=\; \frac{B_{m+1}(1) - B_{m+1}(0)}{m+1}\,,$$

and we know now that this integral is zero when $m \geqslant 1$. Hence the remainder term in Euler's formula,

$$R_m \;=\; \frac{(-1)^{m+1}}{m!} \int_a^b B_m\big(\{x\}\big) f^{(m)}(x)\,dx\,,$$

multiplies $f^{(m)}(x)$ by a function $B_m\big(\{x\}\big)$ whose average value is zero. This means that R_m has a reasonable chance of being small.

Let's look more closely at $B_m(x)$ for $0 \leqslant x \leqslant 1$, since $B_m(x)$ governs the behavior of R_m. Here are the graphs for $B_m(x)$ for the first twelve values of m:

Although $B_3(x)$ through $B_9(x)$ are quite small, the Bernoulli polynomials and numbers ultimately get quite large. Fortunately R_m has a compensating factor $1/m!$, which helps to calm things down.

The graph of $B_m(x)$ begins to look very much like a sine wave when $m \geqslant 3$; exercise 58 proves that $B_m(x)$ can in fact be well approximated by a negative multiple of $\cos(2\pi x - \frac{1}{2}\pi m)$, with relative error $1/2^m$.

In general, $B_{4k+1}(x)$ is negative for $0 < x < \frac{1}{2}$ and positive for $\frac{1}{2} < x < 1$. Therefore its integral, $B_{4k+2}(x)/(4k+2)$, decreases for $0 < x < \frac{1}{2}$ and increases for $\frac{1}{2} < x < 1$. Moreover, we have

$$B_{4k+1}(1-x) \;=\; -B_{4k+1}(x)\,, \qquad \text{for } 0 \leqslant x \leqslant 1,$$

and it follows that

$$B_{4k+2}(1-x) \;=\; B_{4k+2}(x)\,, \qquad \text{for } 0 \leqslant x \leqslant 1.$$

The constant term B_{4k+2} causes the integral $\int_0^1 B_{4k+2}(x)\,dx$ to be zero; hence $B_{4k+2} > 0$. The integral of $B_{4k+2}(x)$ is $B_{4k+3}(x)/(4k+3)$, which must therefore be positive when $0 < x < \frac{1}{2}$ and negative when $\frac{1}{2} < x < 1$; furthermore $B_{4k+3}(1-x) = -B_{4k+3}(x)$, so $B_{4k+3}(x)$ has the properties stated for $B_{4k+1}(x)$, but negated. Therefore $B_{4k+4}(x)$ has the properties stated for $B_{4k+2}(x)$, but negated. Therefore $B_{4k+5}(x)$ has the properties stated for $B_{4k+1}(x)$; we have completed a cycle that establishes the stated properties inductively for all k.

According to this analysis, the maximum value of $B_{2m}(x)$ must occur either at $x = 0$ or at $x = \frac{1}{2}$. Exercise 17 proves that

$$B_{2m}(\tfrac{1}{2}) \;=\; (2^{1-2m} - 1)B_{2m}\,; \tag{9.76}$$

hence we have

$$\bigl|B_{2m}\bigl(\{x\}\bigr)\bigr| \;\leqslant\; |B_{2m}|\,. \tag{9.77}$$

This can be used to establish a useful upper bound on the remainder in Euler's summation formula, because we know from (6.89) that

$$\frac{|B_{2m}|}{(2m)!} \;=\; \frac{2}{(2\pi)^{2m}} \sum_{k\geqslant 1} \frac{1}{k^{2m}} \;=\; O\bigl((2\pi)^{-2m}\bigr)\,, \qquad \text{when } m > 0.$$

Therefore we can rewrite Euler's formula (9.67) as follows:

$$\begin{aligned}\sum_{a\leqslant k<b} f(k) \;=\; \int_a^b f(x)\,dx - \frac{1}{2}f(x)\Big|_a^b + \sum_{k=1}^{m} \frac{B_{2k}}{(2k)!} f^{(2k-1)}(x)\Big|_a^b \\ + O\bigl((2\pi)^{-2m}\bigr) \int_a^b \bigl|f^{(2m)}(x)\bigr|\,dx\,. \end{aligned} \tag{9.78}$$

For example, if $f(x) = e^x$, all derivatives are the same and this formula tells us that $\sum_{a\leqslant k<b} e^k = (e^b - e^a)\bigl(1 - \frac{1}{2} + B_2/2! + B_4/4! + \cdots + B_{2m}/(2m)!\bigr) +$

$O\big((2\pi)^{-2m}\big)$. Of course, we know that this sum is actually a geometric series, equal to $(e^b - e^a)/(e-1) = (e^b - e^a)\sum_{k\geqslant 0} B_k/k!$.

If $f^{(2m)}(x) \geqslant 0$ for $a \leqslant x \leqslant b$, the integral $\int_a^b |f^{(2m)}(x)|\,dx$ is just $f^{(2m-1)}(x)\big|_a^b$, so we have

$$|R_{2m}| \leqslant \left| \frac{B_{2m}}{(2m)!} f^{(2m-1)}(x)\Big|_a^b \right|;$$

in other words, the remainder is bounded by the magnitude of the *final term* (the term just before the remainder), in this case. We can give an even better estimate if we know that

$$f^{(2m+2)}(x) \geqslant 0 \quad \text{and} \quad f^{(2m+4)}(x) \geqslant 0\,, \qquad \text{for } a \leqslant x \leqslant b. \tag{9.79}$$

For it turns out that this implies the relation

$$R_{2m} = \theta_m \frac{B_{2m+2}}{(2m+2)!} f^{(2m+1)}(x)\Big|_a^b\,, \qquad \text{for some } 0 < \theta_m < 1; \tag{9.80}$$

in other words, the remainder will then lie between 0 and the *first discarded term* in (9.78)—the term that would follow the final term if we increased m.

Here's the proof: Euler's summation formula is valid for all m, and $B_{2m+1} = 0$ when $m > 0$; hence $R_{2m} = R_{2m+1}$, and the first discarded term must be

$$R_{2m} - R_{2m+2}\,.$$

We therefore want to show that R_{2m} lies between 0 and $R_{2m} - R_{2m+2}$; and this is true if and only if R_{2m} and R_{2m+2} have opposite signs. We claim that

$$f^{(2m+2)}(x) \geqslant 0 \quad \text{for } a \leqslant x \leqslant b \quad \text{implies} \quad (-1)^m R_{2m} \geqslant 0\,. \tag{9.81}$$

This, together with (9.79), will prove that R_{2m} and R_{2m+2} have opposite signs, so the proof of (9.80) will be complete.

It's not difficult to prove (9.81) if we recall the definition of R_{2m+1} and the facts we proved about the graph of $B_{2m+1}(x)$. Namely, we have

$$R_{2m} = R_{2m+1} = \int_a^b \frac{B_{2m+1}\big(\{x\}\big)}{(2m+1)!} f^{(2m+1)}(x)\,dx\,,$$

and $f^{(2m+1)}(x)$ is increasing because its derivative $f^{(2m+2)}(x)$ is positive. (More precisely, $f^{(2m+1)}(x)$ is nondecreasing because its derivative is nonnegative.) The graph of $B_{2m+1}\big(\{x\}\big)$ looks like $(-1)^{m+1}$ times a sine wave, so it is geometrically obvious that the second half of each sine wave is more influential than the first half when it is multiplied by an increasing function. This makes $(-1)^m R_{2m+1} \geqslant 0$, as desired. Exercise 16 proves the result formally.

9.6 FINAL SUMMATIONS

Now comes the summing up, as we prepare to conclude this book. We will apply Euler's summation formula to some interesting and important examples.

Summation 1: This one is too easy.

But first we will consider an interesting *unimportant* example, namely a sum that we already know how to do. Let's see what Euler's summation formula tells us if we apply it to the telescoping sum

$$S_n = \sum_{1\leqslant k<n} \frac{1}{k(k+1)} = \sum_{1\leqslant k<n} \left(\frac{1}{k} - \frac{1}{k+1}\right) = 1 - \frac{1}{n}\,.$$

It can't hurt to embark on our first serious application of Euler's formula with the asymptotic equivalent of training wheels.

We might as well start by writing the function $f(x) = 1/x(x+1)$ in partial fraction form,

$$f(x) = \frac{1}{x} - \frac{1}{x+1}\,,$$

since this makes it easier to integrate and differentiate. Indeed, we have $f'(x) = -1/x^2 + 1/(x+1)^2$ and $f''(x) = 2/x^3 - 2/(x+1)^3$; in general

$$f^{(k)}(x) = (-1)^k k!\Big(\frac{1}{x^{k+1}} - \frac{1}{(x+1)^{k+1}}\Big)\,, \qquad \text{for } k \geqslant 0.$$

Furthermore

$$\int_1^n f(x)\,dx = \ln x - \ln(x+1)\,\Big|_1^n = \ln\frac{2n}{n+1}\,.$$

Plugging this into the summation formula (9.67) gives

$$S_n = \ln\frac{2n}{n+1} - \sum_{k=1}^{m}(-1)^k\frac{B_k}{k}\left(\frac{1}{n^k} - \frac{1}{(n+1)^k} - 1 + \frac{1}{2^k}\right) + R_m(n)\,,$$

where $R_m(n) = -\displaystyle\int_1^n B_m(\{x\})\Big(\frac{1}{x^{m+1}} - \frac{1}{(x+1)^{m+1}}\Big)\,dx\,.$

For example, the right-hand side when $m = 4$ is

$$\ln\frac{2n}{n+1} - \frac{1}{2}\Big(\frac{1}{n} - \frac{1}{n+1} - \frac{1}{2}\Big) - \frac{1}{12}\Big(\frac{1}{n^2} - \frac{1}{(n+1)^2} - \frac{3}{4}\Big) + \frac{1}{120}\Big(\frac{1}{n^4} - \frac{1}{(n+1)^4} - \frac{15}{16}\Big) + R_4(n)\,.$$

This is kind of a mess; it certainly doesn't look like the real answer $1 - n^{-1}$. But let's keep going anyway, to see what we've got. We know how to expand the right-hand terms in negative powers of n up to, say, $O(n^{-5})$:

$$\begin{aligned}
\ln\frac{n}{n+1} &= -n^{-1} + \tfrac{1}{2}n^{-2} - \tfrac{1}{3}n^{-3} + \tfrac{1}{4}n^{-4} + O(n^{-5})\,; \\
\frac{1}{n+1} &= n^{-1} - n^{-2} + n^{-3} - n^{-4} + O(n^{-5})\,; \\
\frac{1}{(n+1)^2} &= \phantom{-1+{}}n^{-2} - 2n^{-3} + 3n^{-4} + O(n^{-5})\,; \\
\frac{1}{(n+1)^4} &= \phantom{-1+-2+{}}n^{-4} + O(n^{-5})\,.
\end{aligned}$$

Therefore the terms on the right of our approximation add up to

$$\begin{aligned}
&\ln 2 + \tfrac{1}{4} + \tfrac{1}{16} - \tfrac{1}{128} + \bigl(-1 - \tfrac{1}{2} + \tfrac{1}{2}\bigr)n^{-1} + \bigl(\tfrac{1}{2} - \tfrac{1}{2} - \tfrac{1}{12} + \tfrac{1}{12}\bigr)n^{-2} \\
&\qquad + \bigl(-\tfrac{1}{3} + \tfrac{1}{2} - \tfrac{2}{12}\bigr)n^{-3} + \bigl(\tfrac{1}{4} - \tfrac{1}{2} + \tfrac{3}{12} + \tfrac{1}{120} - \tfrac{1}{120}\bigr)n^{-4} + R_4(n) \\
&\quad = \ln 2 + \tfrac{39}{128} - n^{-1} + R_4(n) + O(n^{-5})\,.
\end{aligned}$$

The coefficients of n^{-2}, n^{-3}, and n^{-4} cancel nicely, as they should.

If all were well with the world, we would be able to show that $R_4(n)$ is asymptotically small, maybe $O(n^{-5})$, and we would have an approximation to the sum. But we can't possibly show this, because we happen to know that the correct constant term is 1, not $\ln 2 + \frac{39}{128}$ (which is approximately 0.9978). So $R_4(n)$ is actually equal to $\frac{89}{128} - \ln 2 + O(n^{-4})$, but Euler's summation formula doesn't tell us this.

In other words, we lose.

One way to try fixing things is to notice that the constant terms in the approximation form a pattern, if we let m get larger and larger:

$$\ln 2 - \tfrac{1}{2}B_1 + \tfrac{1}{2}\cdot\tfrac{3}{4}B_2 - \tfrac{1}{3}\cdot\tfrac{7}{8}B_3 + \tfrac{1}{4}\cdot\tfrac{15}{16}B_4 - \tfrac{1}{5}\cdot\tfrac{31}{32}B_5 + \cdots\,.$$

Perhaps we can show that this series approaches 1 as the number of terms becomes infinite? But no; the Bernoulli numbers get very large. For example, $B_{22} = \frac{854513}{138} > 6192$; therefore $\bigl|R_{22}(n)\bigr|$ will be much larger than $\bigl|R_4(n)\bigr|$. We lose totally.

There is a way out, however, and this escape route will turn out to be important in other applications of Euler's formula. The key is to notice that $R_4(n)$ approaches a definite limit as $n \to \infty$:

$$\lim_{n\to\infty} R_4(n) \;=\; -\int_1^\infty B_4\bigl(\{x\}\bigr)\Bigl(\frac{1}{x^5} - \frac{1}{(x+1)^5}\Bigr)\,dx \;=\; R_4(\infty)\,.$$

The integral $\int_1^\infty B_m(\{x\})f^{(m)}(x)\,dx$ will exist whenever $f^{(m)}(x) = O(x^{-2})$ as $x \to \infty$, and in this case $f^{(4)}(x)$ surely qualifies. Moreover, we have

$$\begin{aligned} R_4(n) &= R_4(\infty) + \int_n^\infty B_4(\{x\})\Big(\frac{1}{x^5} - \frac{1}{(x+1)^5}\Big)\,dx \\ &= R_4(\infty) + O\Big(\int_n^\infty x^{-6}\,dx\Big) \;=\; R_4(\infty) + O(n^{-5})\,. \end{aligned}$$

Thus we have used Euler's summation formula to prove that

$$\begin{aligned} \sum_{1\leqslant k<n} \frac{1}{k(k+1)} &= \ln 2 + \tfrac{39}{128} - n^{-1} + R_4(\infty) + O(n^{-5}) \\ &= C - n^{-1} + O(n^{-5}) \end{aligned}$$

for some constant C. We do not know what the constant is — some other method must be used to establish it — but Euler's summation formula is able to let us deduce that the constant exists.

Suppose we had chosen a much larger value of m. Then the same reasoning would tell us that

$$R_m(n) \;=\; R_m(\infty) + O(n^{-m-1})\,,$$

and we would have the formula

$$\sum_{1\leqslant k<n} \frac{1}{k(k+1)} = C - n^{-1} + c_2 n^{-2} + c_3 n^{-3} + \cdots + c_m n^{-m} + O(n^{-m-1})$$

for certain constants c_2, c_3, $\ldots$. We know that the c's happen to be zero in this case; but let's prove it, just to restore some of our confidence (in Euler's formula if not in ourselves). The term $\ln\frac{n}{n+1}$ contributes $(-1)^m/m$ to c_m; the term $(-1)^{m+1}(B_m/m)n^{-m}$ contributes $(-1)^{m+1}B_m/m$; and the term $(-1)^k(B_k/k)(n+1)^{-k}$ contributes $(-1)^m\binom{m-1}{k-1}B_k/k$. Therefore

$$\begin{aligned} (-1)^m c_m &= \frac{1}{m} - \frac{B_m}{m} + \sum_{k=1}^m \binom{m-1}{k-1}\frac{B_k}{k} \\ &= \frac{1}{m} - \frac{B_m}{m} + \frac{1}{m}\sum_{k=1}^m \binom{m}{k} B_k \;=\; \frac{1}{m}\big(1 - B_m + B_m(1) - 1\big)\,. \end{aligned}$$

Sure enough, it's zero, when $m > 1$. We have proved that

$$\sum_{1\leqslant k<n} \frac{1}{k(k+1)} \;=\; C - n^{-1} + O(n^{-m-1})\,, \qquad \text{for all } m \geqslant 1. \tag{9.82}$$

This is not enough to prove that the sum is exactly equal to $C - n^{-1}$; the actual value might be $C - n^{-1} + 2^{-n}$ or something. But Euler's summation

formula does give us the error bound $O(n^{-m-1})$ for arbitrarily large m, even though we haven't evaluated any remainders explicitly.

Summation 1, again: Recapitulation and generalization.

Before we leave our training wheels, let's review what we just did from a somewhat higher perspective. We began with a sum

$$S_n = \sum_{1 \leqslant k < n} f(k)$$

and we used Euler's summation formula to write

$$S_n = F(n) - F(1) + \sum_{k=1}^{m} \bigl(T_k(n) - T_k(1)\bigr) + R_m(n)\,, \tag{9.83}$$

where $F(x)$ was $\int f(x)\,dx$ and where $T_k(x)$ was a certain term involving B_k and $f^{(k-1)}(x)$. We also noticed that there was a constant c such that

$$f^{(m)}(x) = O(x^{c-m}) \quad \text{as } x \to \infty, \qquad \text{for all large } m.$$

(Namely, $f(k)$ was $1/k(k+1)$; $F(x)$ was $\ln\bigl(x/(x+1)\bigr)$; c was -2; and $T_k(x)$ was $(-1)^{k+1}(B_k/k)\bigl(x^{-k} - (x+1)^{-k}\bigr)$.) For all large enough values of m, this implied that the remainders had a small tail,

$$\begin{aligned} R'_m(n) &= R_m(\infty) - R_m(n) \\ &= (-1)^{m+1} \int_n^\infty \frac{B_m\bigl(\{x\}\bigr)}{m!} f^{(m)}(x)\,dx = O(n^{c+1-m})\,. \end{aligned} \tag{9.84}$$

Therefore we were able to conclude that there exists a constant C such that

$$S_n = F(n) + C + \sum_{k=1}^{m} T_k(n) - R'_m(n)\,. \tag{9.85}$$

(Notice that C nicely absorbed the $T_k(1)$ terms, which were a nuisance.)

We can save ourselves unnecessary work in future problems by simply asserting the existence of C whenever $R_m(\infty)$ exists.

Now let's suppose that $f^{(2m+2)}(x) \geqslant 0$ and $f^{(2m+4)}(x) \geqslant 0$ for $1 \leqslant x \leqslant n$. We have proved that this implies a simple bound (9.80) on the remainder,

$$R_{2m}(n) = \theta_{m,n}\bigl(T_{2m+2}(n) - T_{2m+2}(1)\bigr)\,,$$

where $\theta_{m,n}$ lies somewhere between 0 and 1. But we don't really want bounds that involve $R_{2m}(n)$ and $T_{2m+2}(1)$; after all, we got rid of $T_k(1)$ when we introduced the constant C. What we really want is a bound like

$$-R'_{2m}(n) = \phi_{m,n} T_{2m+2}(n)\,,$$

where $0 < \phi_{m,n} < 1$; this will allow us to conclude from (9.85) that

$$S_n \;=\; F(n) + C + T_1(n) + \sum_{k=1}^{m} T_{2k}(n) + \phi_{m,n} T_{2m+2}(n)\,, \tag{9.86}$$

hence the remainder will truly be between zero and the first discarded term.

A slight modification of our previous argument will patch things up perfectly. Let us assume that

$$f^{(2m+2)}(x) \;\geqslant\; 0 \quad\text{and}\quad f^{(2m+4)}(x) \;\geqslant\; 0\,, \qquad \text{as } x \to \infty. \tag{9.87}$$

The right-hand side of (9.85) is just like the negative of the right-hand side of Euler's summation formula (9.67) with $a = n$ and $b = \infty$, as far as remainder terms are concerned, and successive remainders are generated by induction on m. Therefore our previous argument can be applied.

Summation 2: Harmonic numbers harmonized.

Now that we've learned so much from a trivial (but safe) example, we can readily do a nontrivial one. Let us use Euler's summation formula to derive the approximation for H_n that we have been claiming for some time.

In this case, $f(x) = 1/x$. We already know about the integral and derivatives of f, because of Summation 1; also $f^{(m)}(x) = O(x^{-m-1})$ as $x \to \infty$. Therefore we can immediately plug into formula (9.85):

$$\sum_{1\leqslant k<n} \frac{1}{k} \;=\; \ln n + C + B_1 n^{-1} - \sum_{k=1}^{m} \frac{B_{2k}}{2kn^{2k}} - R'_{2m}(n)\,,$$

for some constant C. The sum on the left is H_{n-1}, not H_n; but it's more convenient to work with H_{n-1} and to add $1/n$ later, than to mess around with $(n+1)$'s on the right-hand side. The $B_1 n^{-1}$ will then become $(B_1+1)n^{-1} = 1/(2n)$. Let us call the constant γ instead of C, since Euler's constant γ is, in fact, defined to be $\lim_{n\to\infty}(H_n - \ln n)$.

The remainder term can be estimated nicely by the theory we developed a minute ago, because $f^{(2m)}(x) = (2m)!/x^{2m+1} \geqslant 0$ for all $x > 0$. Therefore (9.86) tells us that

$$H_n \;=\; \ln n + \gamma + \frac{1}{2n} - \sum_{k=1}^{m} \frac{B_{2k}}{2kn^{2k}} + \theta_{m,n} \frac{B_{2m+2}}{(2m+2)n^{2m+2}}\,, \tag{9.88}$$

where $\theta_{m,n}$ is some fraction between 0 and 1. This is the general formula whose first few terms are listed in Table 452. For example, when $m = 2$ we get

$$H_n \;=\; \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \frac{\theta_{2,n}}{252n^6}\,. \tag{9.89}$$

This equation, incidentally, gives us a good approximation to γ even when $n = 2$:

$$\gamma \;=\; H_2 - \ln 2 - \tfrac{1}{4} + \tfrac{1}{48} - \tfrac{1}{1920} + \epsilon \;=\; 0.577165\ldots + \epsilon\,,$$

where ϵ is between zero and $\frac{1}{16128}$. If we take $n = 10^4$ and $m = 250$, we get the value of γ correct to 1271 decimal places, beginning thus [205]:

$$\gamma \;=\; 0.57721\,56649\,01532\,86060\,65120\,90082\,40243\ldots. \tag{9.90}$$

But Euler's constant appears also in other formulas that allow it to be evaluated even more efficiently [345].

Summation 3: Stirling's approximation.

If $f(x) = \ln x$, we have $f'(x) = 1/x$, so we can evaluate the sum of logarithms using almost the same calculations as we did when summing reciprocals. Euler's summation formula yields

$$\sum_{1\leqslant k<n} \ln k \;=\; n\ln n - n + \sigma - \frac{\ln n}{2} + \sum_{k=1}^{m} \frac{B_{2k}}{2k(2k-1)n^{2k-1}} + \varphi_{m,n}\frac{B_{2m+2}}{(2m+2)(2m+1)n^{2m+1}}$$

where σ is a certain constant, "Stirling's constant," and $0 < \varphi_{m,n} < 1$. (In this case $f^{(2m)}(x)$ is negative, not positive; but we can still say that the remainder is governed by the first discarded term, because we could have started with $f(x) = -\ln x$ instead of $f(x) = \ln x$.) Adding $\ln n$ to both sides gives

$$\ln n! \;=\; n\ln n - n + \frac{\ln n}{2} + \sigma + \frac{1}{12n} - \frac{1}{360n^3} + \frac{\varphi_{2,n}}{1260n^5} \tag{9.91}$$

when $m = 2$. And we can get the approximation in Table 452 by taking 'exp' of both sides. (The value of e^σ turns out to be $\sqrt{2\pi}$, but we aren't quite ready to derive that formula. In fact, Stirling didn't discover the closed form for σ until several years after de Moivre [76] had proved that the constant exists.)

If m is fixed and $n \to \infty$, the general formula gives a better and better approximation to $\ln n!$ in the sense of absolute error, hence it gives a better and better approximation to $n!$ in the sense of relative error. But if n is fixed and m increases, the error bound $|B_{2m+2}|/(2m+2)(2m+1)n^{2m+1}$ decreases to a certain point and then begins to increase. Therefore the approximation reaches a point beyond which a sort of uncertainty principle limits the amount by which $n!$ can be approximated.

Heisenberg may have been here.

In Chapter 5, equation (5.83), we generalized factorials to arbitrary real α by using a definition

$$\frac{1}{\alpha!} = \lim_{n\to\infty} \binom{n+\alpha}{n} n^{-\alpha}$$

suggested by Euler. Suppose α is a large number; then

$$\ln \alpha! = \lim_{n\to\infty} \Big(\alpha \ln n + \ln n! - \sum_{k=1}^{n} \ln(\alpha + k) \Big) ,$$

and Euler's summation formula can be used with $f(x) = \ln(x+\alpha)$ to estimate this sum:

$$\begin{aligned}
\sum_{k=1}^{n} \ln(k+\alpha) &= F_m(\alpha, n) - F_m(\alpha, 0) + R_{2m}(\alpha, n) , \\
F_m(\alpha, x) &= (x+\alpha)\ln(x+\alpha) - x + \frac{\ln(x+\alpha)}{2} \\
&\qquad + \sum_{k=1}^{m} \frac{B_{2k}}{2k(2k-1)(x+\alpha)^{2k-1}} , \\
R_{2m}(\alpha, n) &= \int_0^n \frac{B_{2m}(\{x\})}{2m} \frac{dx}{(x+\alpha)^{2m}} .
\end{aligned}$$

(Here we have used (9.67) with $a = 0$ and $b = n$, then added $\ln(n+\alpha) - \ln \alpha$ to both sides.) If we subtract this approximation for $\sum_{k=1}^{n} \ln(k+\alpha)$ from Stirling's approximation for $\ln n!$, then add $\alpha \ln n$ and take the limit as $n \to \infty$, we get

$$\begin{aligned}
\ln \alpha! &= \alpha \ln \alpha - \alpha + \frac{\ln \alpha}{2} + \sigma \\
&\qquad + \sum_{k=1}^{m} \frac{B_{2k}}{(2k)(2k-1)\alpha^{2k-1}} - \int_0^\infty \frac{B_{2m}(\{x\})}{2m} \frac{dx}{(x+\alpha)^{2m}} ,
\end{aligned}$$

because $\alpha \ln n + n \ln n - n + \frac{1}{2} \ln n - (n+\alpha)\ln(n+\alpha) + n - \frac{1}{2}\ln(n+\alpha) \to -\alpha$ and the other terms not shown here tend to zero. Thus Stirling's approximation behaves for generalized factorials (and for the Gamma function $\Gamma(\alpha+1) = \alpha!$) exactly as for ordinary factorials.

Summation 4: A bell-shaped summand.

Let's turn now to a sum that has quite a different flavor:

$$\begin{aligned}
\Theta_n &= \sum_k e^{-k^2/n} \qquad (9.92) \\
&= \cdots + e^{-9/n} + e^{-4/n} + e^{-1/n} + 1 + e^{-1/n} + e^{-4/n} + e^{-9/n} + \cdots .
\end{aligned}$$

This is a doubly infinite sum, whose terms reach their maximum value $e^0 = 1$ when $k = 0$. We call it Θ_n because it is a power series involving the quantity $e^{-1/n}$ raised to the $p(k)$th power, where $p(k)$ is a polynomial of degree 2; such power series are traditionally called "theta functions." If $n = 10^{100}$, we have

$$e^{-k^2/n} = \begin{cases} e^{-.01} \approx 0.99005, & \text{when } k = 10^{49}; \\ e^{-1} \approx 0.36788, & \text{when } k = 10^{50}; \\ e^{-100} < 10^{-43}, & \text{when } k = 10^{51}. \end{cases}$$

So the summand stays very near 1 until k gets up to about $\sqrt{n}$, when it drops off and stays very near zero. We can guess that Θ_n will be proportional to $\sqrt{n}$. Here is a graph of $e^{-k^2/n}$ when $n = 10$:

Larger values of n just stretch the graph horizontally by a factor of $\sqrt{n}$.

We can estimate Θ_n by letting $f(x) = e^{-x^2/n}$ and taking $a = -\infty$, $b = +\infty$ in Euler's summation formula. (If infinities seem too scary, let $a = -A$ and $b = +B$, then take limits as $A, B \to \infty$.) The integral of $f(x)$ is

$$\int_{-\infty}^{+\infty} e^{-x^2/n}\,dx = \sqrt{n}\int_{-\infty}^{+\infty} e^{-u^2}\,du = \sqrt{n}\,C\,,$$

if we replace x by $u\sqrt{n}$. The value of $\int_{-\infty}^{+\infty} e^{-u^2}\,du$ is well known, but we'll call it C for now and come back to it after we have finished plugging into Euler's summation formula.

The next thing we need to know is the sequence of derivatives $f'(x)$, $f''(x)$, ..., and for this purpose it's convenient to set

$$f(x) = g\bigl(x/\sqrt{n}\bigr)\,, \qquad g(x) = e^{-x^2}\,.$$

Then the chain rule of calculus says that

$$\frac{df(x)}{dx} = \frac{dg(y)}{dy}\frac{dy}{dx}\,, \qquad y = \frac{x}{\sqrt{n}}\,;$$

and this is the same as saying that

$$f'(x) = \frac{1}{\sqrt{n}}\,g'\bigl(x/\sqrt{n}\bigr)\,.$$

By induction we have

$$f^{(k)}(x) = n^{-k/2}g^{(k)}\bigl(x/\sqrt{n}\bigr)\,.$$

For example, we have $g'(x) = -2xe^{-x^2}$ and $g''(x) = (4x^2 - 2)e^{-x^2}$; hence

$$f'(x) \;=\; \frac{1}{\sqrt{n}}\left(-2\frac{x}{\sqrt{n}}\right)e^{-x^2/n}\,, \qquad f''(x) \;=\; \frac{1}{n}\left(4\left(\frac{x}{\sqrt{n}}\right)^2 - 2\right)e^{-x^2/n}\,.$$

It's easier to see what's going on if we work with the simpler function $g(x)$.

We don't have to evaluate the derivatives of $g(x)$ exactly, because we're only going to be concerned about the limiting values when $x = \pm\infty$. And for this purpose it suffices to notice that every derivative of $g(x)$ is e^{-x^2} times a polynomial in x:

$$g^{(k)}(x) \;=\; P_k(x)e^{-x^2}\,, \qquad \text{where } P_k \text{ is a polynomial of degree } k.$$

This follows by induction.

The negative exponential e^{-x^2} goes to zero much faster than $P_k(x)$ goes to infinity, when $x \to \pm\infty$, so we have

$$f^{(k)}(+\infty) \;=\; f^{(k)}(-\infty) \;=\; 0$$

for all $k \geqslant 0$. Therefore all of the terms

$$\sum_{k=1}^{m} \frac{B_k}{k!} f^{(k-1)}(x)\Big|_{-\infty}^{+\infty}$$

vanish, and we are left with the term from $\int f(x)\,dx$ and the remainder:

$$\begin{aligned}
\Theta_n \;&=\; C\sqrt{n} + (-1)^{m+1}\int_{-\infty}^{+\infty} \frac{B_m\big(\{x\}\big)}{m!} f^{(m)}(x)\,dx \\
&=\; C\sqrt{n} + \frac{(-1)^{m+1}}{n^{m/2}}\int_{-\infty}^{+\infty} \frac{B_m\big(\{x\}\big)}{m!} g^{(m)}\left(\frac{x}{\sqrt{n}}\right) dx \qquad (x = u\sqrt{n}\,) \\
&=\; C\sqrt{n} + \frac{(-1)^{m+1}}{n^{(m-1)/2}}\int_{-\infty}^{+\infty} \frac{B_m\big(\{u\sqrt{n}\,\}\big)}{m!} P_m(u)e^{-u^2}\,du \\
&=\; C\sqrt{n} + O(n^{(1-m)/2})\,.
\end{aligned}$$

The O estimate here follows since $\big|B_m\big(\{u\sqrt{n}\,\}\big)\big|$ is bounded and the integral $\int_{-\infty}^{+\infty}\big|P(u)\big|e^{-u^2}\,du$ exists whenever P is a polynomial. (The constant implied by this O depends on m.)

We have proved that $\Theta_n = C\sqrt{n} + O(n^{-M})$, for arbitrarily large M; the difference between Θ_n and $C\sqrt{n}$ is "exponentially small." Let us therefore determine the constant C that plays such a big role in the value of Θ_n.

One way to determine C is to look the integral up in a table; but we prefer to know how the value can be derived, so that we can do integrals even

when they haven't been tabulated. Elementary calculus suffices to evaluate C if we are clever enough to look at the double integral

$$C^2 = \int_{-\infty}^{+\infty} e^{-x^2}\,dx \int_{-\infty}^{+\infty} e^{-y^2}\,dy = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} e^{-(x^2+y^2)}\,dx\,dy\,.$$

Converting to polar coordinates gives

$$\begin{aligned} C^2 &= \int_0^{2\pi}\int_0^{\infty} e^{-r^2}\,r\,dr\,d\theta \\ &= \frac{1}{2}\int_0^{2\pi} d\theta \int_0^{\infty} e^{-u}\,du \qquad\qquad (u = r^2) \\ &= \frac{1}{2}\int_0^{2\pi} d\theta = \pi\,. \end{aligned}$$

So $C = \sqrt{\pi}$. The fact that $x^2 + y^2 = r^2$ is the equation of a circle whose circumference is $2\pi r$ somehow explains why π gets into the act.

Another way to evaluate C is to replace x by $\sqrt{t}$ and dx by $\frac{1}{2}t^{-1/2}\,dt$:

$$C = \int_{-\infty}^{+\infty} e^{-x^2}\,dx = 2\int_0^{\infty} e^{-x^2}\,dx = \int_0^{\infty} t^{-1/2}e^{-t}\,dt\,.$$

This integral equals $\Gamma\bigl(\frac{1}{2}\bigr)$, since $\Gamma(\alpha) = \int_0^{\infty} t^{\alpha-1}e^{-t}\,dt$ according to (5.84). Therefore we have demonstrated that $\Gamma\bigl(\frac{1}{2}\bigr) = \sqrt{\pi}$.

Our final formula, then, is

$$\Theta_n = \sum_k e^{-k^2/n} = \sqrt{\pi n} + O(n^{-M})\,, \qquad \text{for all fixed } M. \tag{9.93}$$

The constant in the O depends on M; that's why we say that M is "fixed."

When $n = 2$, for example, the infinite sum Θ_2 is approximately equal to 2.506628288; this is already very close to $\sqrt{2\pi} \approx 2.506628275$, even though n is quite small. The value of Θ_{100} agrees with $10\sqrt{\pi}$ to 427 decimal places! Exercise 59 uses advanced methods to derive a rapidly convergent series for Θ_n; it turns out that

$$\Theta_n/\sqrt{\pi n} = 1 + 2e^{-n\pi^2} + O(e^{-4n\pi^2})\,. \tag{9.94}$$

Summation 5: The clincher.

Now we will do one last sum, which will turn out to tell us the value of Stirling's constant σ. This last sum also illustrates many of the other techniques of this last chapter (and of this whole book), so it will be a fitting way for us to conclude our explorations of Concrete Mathematics.

The final task seems almost absurdly easy: We will try to find the asymptotic value of

$$A_n = \sum_k \binom{2n}{k}$$

by using Euler's summation formula.

This is another case where we already know the answer (right?); but it's always interesting to try new methods on old problems, so that we can compare facts and maybe discover something new.

So we THINK BIG and realize that the main contribution to A_n comes from the middle terms, near $k = n$. It's almost always a good idea to choose notation so that the biggest contribution to a sum occurs near $k = 0$, because we can then use the tail-exchange trick to get rid of terms that have large $|k|$. Therefore we replace k by $n + k$:

$$A_n = \sum_k \binom{2n}{n+k} = \sum_k \frac{(2n)!}{(n+k)!\,(n-k)!}\,.$$

Things are looking reasonably good, since we know how to approximate $(n \pm k)!$ when n is large and k is small.

Now we want to carry out the three-step procedure associated with the tail-exchange trick. Namely, we want to write

$$\frac{(2n)!}{(n+k)!\,(n-k)!} = a_k(n) = b_k(n) + O\big(c_k(n)\big)\,, \qquad \text{for } k \in D_n,$$

so that we can obtain the estimate

$$A_n = \sum_k b_k(n) + O\Big(\sum_{k\notin D_n} a_k(n)\Big) + O\Big(\sum_{k\notin D_n} b_k(n)\Big) + \sum_{k\in D_n} O\big(c_k(n)\big)\,.$$

Let us therefore try to estimate $\binom{2n}{n+k}$ in the region where $|k|$ is small. We could use Stirling's approximation as it appears in Table 452, but it's easier to work with the logarithmic equivalent in (9.91):

$$\begin{aligned}
\ln a_k(n) &= \ln(2n)! - \ln(n+k)! - \ln(n-k)! \\
&= 2n\ln 2n - 2n + \tfrac{1}{2}\ln 2n + \sigma + O(n^{-1}) \\
&\quad - (n{+}k)\ln(n{+}k) + n + k - \tfrac{1}{2}\ln(n{+}k) - \sigma + O\big((n{+}k)^{-1}\big) \\
&\quad - (n{-}k)\ln(n{-}k) + n - k - \tfrac{1}{2}\ln(n{-}k) - \sigma + O\big((n{-}k)^{-1}\big)\,.
\end{aligned} \tag{9.95}$$

We want to convert this to a nice, simple O estimate.

The tail-exchange method allows us to work with estimates that are valid only when k is in the "dominant" set D_n. But how should we define D_n?

Actually I'm not into dominance.

We have to make D_n small enough that we can make a good estimate; for example, we had better not let k get near n, or the term $O\big((n-k)^{-1}\big)$ in (9.95) will blow up. Yet D_n must be large enough that the tail terms (the terms with $k \notin D_n$) are negligibly small compared with the overall sum. Trial and error is usually necessary to find an appropriate set D_n; in this problem the calculations we are about to make will show that it's wise to define things as follows:

$$k \in D_n \quad \Longleftrightarrow \quad |k| \leqslant n^{1/2+\epsilon}\,. \tag{9.96}$$

Here ϵ is a small positive constant that we can choose later, after we get to know the territory. (Our O estimates will depend on the value of ϵ.) Equation (9.95) now reduces to

$$\begin{aligned}\ln a_k(n) \;=\; & (2n+\tfrac{1}{2})\ln 2 - \sigma - \tfrac{1}{2}\ln n + O(n^{-1}) \\ & - (n+k+\tfrac{1}{2})\ln(1+k/n) - (n-k+\tfrac{1}{2})\ln(1-k/n)\,.\end{aligned} \tag{9.97}$$

(We have pulled out the large parts of the logarithms, writing

$$\ln(n \pm k) \;=\; \ln n + \ln(1 \pm k/n)\,,$$

and this has made a lot of $\ln n$ terms cancel out.)

Now we need to expand the terms $\ln(1 \pm k/n)$ asymptotically, until we have an error term that approaches zero as $n \to \infty$. We are multiplying $\ln(1 \pm k/n)$ by $(n \pm k + \frac{1}{2})$, so we should expand the logarithm until we reach $o(n^{-1})$, using the assumption that $|k| \leqslant n^{1/2+\epsilon}$:

$$\ln\Big(1 \pm \frac{k}{n}\Big) \;=\; \pm\frac{k}{n} - \frac{k^2}{2n^2} + O(n^{-3/2+3\epsilon})\,.$$

Multiplication by $n \pm k + \frac{1}{2}$ yields

$$\pm k - \frac{k^2}{2n} + \frac{k^2}{n} + O(n^{-1/2+3\epsilon})\,,$$

plus other terms that are absorbed in the $O(n^{-1/2+3\epsilon})$. So (9.97) becomes

$$\ln a_k(n) \;=\; (2n+\frac{1}{2})\ln 2 - \sigma - \frac{1}{2}\ln n - k^2/n + O(n^{-1/2+3\epsilon})\,.$$

Taking exponentials, we have

$$a_k(n) \;=\; \frac{2^{2n+1/2}}{e^{\sigma}\sqrt{n}}\, e^{-k^2/n}\big(1 + O(n^{-1/2+3\epsilon})\big)\,. \tag{9.98}$$

This is our approximation, with

$$b_k(n) \;=\; \frac{2^{2n+1/2}}{e^\sigma\sqrt{n}}\,e^{-k^2/n}\,, \qquad c_k(n) \;=\; 2^{2n}\,n^{-1+3\epsilon}\,e^{-k^2/n}\,.$$

Notice that k enters $b_k(n)$ and $c_k(n)$ in a very simple way. We're in luck, because we will be summing over k.

The tail-exchange trick tells us that $\sum_k a_k(n)$ will be approximately $\sum_k b_k(n)$ if we have done a good job of estimation. Let us therefore evaluate

$$\begin{aligned}\sum_k b_k(n) \;&=\; \frac{2^{2n+1/2}}{e^\sigma\sqrt{n}}\sum_k e^{-k^2/n}\\ &=\; \frac{2^{2n+1/2}}{e^\sigma\sqrt{n}}\Theta_n \;=\; \frac{2^{2n}\sqrt{2\pi}}{e^\sigma}\big(1+O(n^{-M})\big)\,.\end{aligned}$$

(Another stroke of luck: We get to use the sum Θ_n from the previous example.) This is encouraging, because we know that the original sum is actually

What an amazing coincidence.

$$A_n \;=\; \sum_k \binom{2n}{k} \;=\; (1+1)^{2n} \;=\; 2^{2n}\,.$$

Therefore it looks as if we will have $e^\sigma = \sqrt{2\pi}$, as advertised.

But there's a catch: We still need to prove that our estimates are good enough. So let's look first at the error contributed by $c_k(n)$:

I'm tired of getting to the end of long, hard books and not even getting a word of good wishes from the author. It would be nice to read a "thanks for reading this, hope it comes in handy," instead of just running into a hard, cold, cardboard cover at the end of a long, dry proof. You know?

$$\Sigma_c(n) \;=\; \sum_{|k|\leqslant n^{1/2+\epsilon}} 2^{2n}n^{-1+3\epsilon}e^{-k^2/n} \;\leqslant\; 2^{2n}n^{-1+3\epsilon}\Theta_n \;=\; O(2^{2n}n^{-\frac12+3\epsilon})\,.$$

Good; this is asymptotically smaller than the previous sum, if $3\epsilon < \frac12$.

Next we must check the tails. We have

$$\begin{aligned}\sum_{k>n^{1/2+\epsilon}} e^{-k^2/n} \;&<\; \exp\big(-\lfloor n^{1/2+\epsilon}\rfloor^2/n\big)(1+e^{-1/n}+e^{-2/n}+\cdots)\\ &=\; O(e^{-n^{2\epsilon}})\cdot O(n)\,,\end{aligned}$$

which is $O(n^{-M})$ for all M; so $\sum_{k\notin D_n} b_k(n)$ is asymptotically negligible. (We chose the cutoff at $n^{1/2+\epsilon}$ just so that $e^{-k^2/n}$ would be exponentially small outside of D_n. Other choices like $n^{1/2}\log n$ would have been good enough too, and the resulting estimates would have been slightly sharper, but the formulas would have come out more complicated. We need not make the strongest possible estimates, since our main goal is to establish the value of the constant σ.) Similarly, the other tail

$$\sum_{k>n^{1/2+\epsilon}} \binom{2n}{n+k}$$

is bounded by $2n$ times its largest term, which occurs at the cutoff point $k \approx n^{1/2+\epsilon}$. This term is known to be approximately $b_k(n)$, which is exponentially small compared with A_n; and an exponentially small multiplier wipes out the factor of $2n$.

Thus we have successfully applied the tail-exchange trick to prove the estimate

$$2^{2n} = \sum_k \binom{2n}{k} = \frac{\sqrt{2\pi}}{e^\sigma} 2^{2n} + O(2^{2n} n^{-\frac{1}{2}+3\epsilon}), \quad \text{if } 0 < \epsilon < \tfrac{1}{6}. \tag{9.99}$$

Thanks for reading this, hope it comes in handy.
—The authors

We may choose $\epsilon = \frac{1}{8}$ and conclude that

$$\sigma = \tfrac{1}{2} \ln 2\pi .$$

QED.

Exercises

Warmups

1 Prove or disprove: If $f_1(n) \prec g_1(n)$ and $f_2(n) \prec g_2(n)$, then we have $f_1(n) + f_2(n) \prec g_1(n) + g_2(n)$.

2 Which function grows faster:

a $n^{(\ln n)}$ or $(\ln n)^n$?
b $n^{(\ln \ln \ln n)}$ or $(\ln n)!$?
c $(n!)!$ or $\big((n-1)!\big)!\,(n-1)!^{n!}$?
d $F^2_{\lceil H_n \rceil}$ or H_{F_n}?

3 What's wrong with the following argument? "Since $n = O(n)$ and $2n = O(n)$ and so on, we have $\sum_{k=1}^n kn = \sum_{k=1}^n O(n) = O(n^2)$."

4 Give an example of a valid equation that has O-notation on the left but not on the right. (Do not use the trick of multiplying by zero; that's too easy.) *Hint:* Consider taking limits.

5 Prove or disprove: $O\big(f(n) + g(n)\big) = f(n) + O\big(g(n)\big)$, if $f(n)$ and $g(n)$ are positive for all n. (Compare with (9.27).)

6 Multiply $\big(\ln n + \gamma + O(1/n)\big)$ by $\big(n + O(\sqrt{n}\,)\big)$, and express your answer in O-notation.

7 Estimate $\sum_{k \geqslant 0} e^{-k/n}$ with absolute error $O(n^{-1})$.

Basics

8 Give an example of functions $f(n)$ and $g(n)$ such that none of the three relations $f(n) \prec g(n)$, $f(n) \succ g(n)$, $f(n) \asymp g(n)$ is valid, although $f(n)$ and $g(n)$ both increase monotonically to ∞.

9 Prove (9.22) rigorously by showing that the left side is a subset of the right side, according to the set-of-functions definition of O.

10 Prove or disprove: $\cos O(x) = 1 + O(x^2)$ for all real x.

11 Prove or disprove: $O(x+y)^2 = O(x^2) + O(y^2)$.

12 Prove that

$$1 + \frac{2}{n} + O(n^{-2}) \;=\; \Bigl(1 + \frac{2}{n}\Bigr)\bigl(1 + O(n^{-2})\bigr)\,,$$

as $n \to \infty$.

13 Evaluate $\bigl(n + 2 + O(n^{-1})\bigr)^n$ with relative error $O(n^{-1})$.

14 Prove that $(n+\alpha)^{n+\beta} = n^{n+\beta}e^{\alpha}\bigl(1 + \alpha(\beta - \frac{1}{2}\alpha)n^{-1} + O(n^{-2})\bigr)$.

15 Give an asymptotic formula for the "middle" trinomial coefficient $\binom{3n}{n,n,n}$, correct to relative error $O(n^{-3})$.

16 Show that if $B(1-x) = -B(x) \geqslant 0$ for $0 < x < \frac{1}{2}$, we have

$$\int_a^b B\bigl(\{x\}\bigr) f(x)\,dx \;\geqslant\; 0$$

if we assume also that $f'(x) \geqslant 0$ for $a \leqslant x \leqslant b$.

17 Use generating functions to show that $B_m(\frac{1}{2}) = (2^{1-m} - 1)B_m$, for all $m \geqslant 0$.

18 Find $\sum_k \binom{2n}{k}^{\alpha}$ with relative error $O(n^{-1/4})$, when $\alpha > 0$.

Homework exercises

19 Use a computer to compare the left and right sides of the approximations in Table 452, when $n = 10$, $z = \alpha = 0.1$, and $O\bigl(f(n)\bigr) = O\bigl(f(z)\bigr) = 0$.

20 Prove or disprove the following estimates, as $n \to \infty$:

a $\quad O\biggl(\Bigl(\dfrac{n^2}{\log\log n}\Bigr)^{1/2}\biggr) \;=\; O\bigl(\lfloor\sqrt{n}\rfloor^2\bigr)\,.$

b $\quad e^{(1+O(1/n))^2} \;=\; e + O(1/n)\,.$

c $\quad n! \;=\; O\Bigl(\bigl((1-1/n)^n n\bigr)^n\Bigr)\,.$

21 Equation (9.48) gives the nth prime with relative error $O(\log n)^{-2}$. Improve the relative error to $O(\log n)^{-3}$ by starting with another term of (9.31) in (9.46).

22 Improve (9.54) to $O(n^{-3})$.

23 Push the approximation (9.62) further, getting absolute error $O(n^{-3})$. *Hint:* Let $g_n = c/(n+1)(n+2) + h_n$; what recurrence does h_n satisfy?

24 Suppose $a_n = O\big(f(n)\big)$ and $b_n = O\big(f(n)\big)$. Prove or disprove that the convolution $\sum_{k=0}^{n} a_k b_{n-k}$ is also $O\big(f(n)\big)$, in the following cases:

a $f(n) = n^{-\alpha}$, $\alpha > 1$.
b $f(n) = \alpha^{-n}$, $\alpha > 1$.

25 Prove (9.1) and (9.2), with which we opened this chapter.

26 Equation (9.91) shows how to evaluate $\ln 10!$ with an absolute error $< \frac{1}{126000000}$. Therefore if we take exponentials, we get $10!$ with a relative error that is less than $e^{1/126000000} - 1 < 10^{-8}$. (In fact, the approximation gives 3628799.9714.) If we now round to the nearest integer, knowing that $10!$ is an integer, we get an exact result.

Is it always possible to calculate $n!$ in a similar way, if enough terms of Stirling's approximation are computed? Estimate the value of m that gives the best approximation to $\ln n!$, when n is a fixed (large) integer. Compare the absolute error in this approximation with $n!$ itself.

27 Use Euler's summation formula to find the asymptotic value of $H_n^{(-\alpha)} = \sum_{k=1}^{n} k^{\alpha}$, where α is any fixed real number. (Your answer may involve a constant that you do not know in closed form.)

28 Exercise 5.13 defines the hyperfactorial function $Q_n = 1^1 2^2 \ldots n^n$. Find the asymptotic value of Q_n with relative error $O(n^{-1})$. (Your answer may involve a constant that you do not know in closed form.)

29 Estimate the function $1^{1/1} 2^{1/2} \ldots n^{1/n}$ as in the previous exercise.

30 Find the asymptotic value of $\sum_{k\geqslant 0} k^l e^{-k^2/n}$ with absolute error $O(n^{-3})$, when l is a fixed nonnegative integer.

31 Evaluate $\sum_{k\geqslant 0} 1/(c^k + c^m)$ with absolute error $O(c^{-3m})$, when $c > 1$ and m is a positive integer.

Exam problems

32 Evaluate $e^{H_n + H_n^{(2)}}$ with absolute error $O(n^{-1})$.

33 Evaluate $\sum_{k\geqslant 0} \binom{n}{k}/n^{\overline{k}}$ with absolute error $O(n^{-3})$.

34 Determine values A through F such that $(1 + 1/n)^{nH_n}$ is

$$An + B(\ln n)^2 + C\ln n + D + \frac{E(\ln n)^2}{n} + \frac{F\ln n}{n} + O(n^{-1})\,.$$

35 Evaluate $\sum_{k=1}^{n} 1/kH_k$ with absolute error $O(1)$.

36 Evaluate $S_n = \sum_{k=1}^{n} 1/(n^2 + k^2)$ with absolute error $O(n^{-5})$.

37 Evaluate $\sum_{k=1}^{n} (n \bmod k)$ with absolute error $O(n \log n)$.

38 Evaluate $\sum_{k\geqslant 0} k^k \binom{n}{k}$ with relative error $O(n^{-1})$.

39 Evaluate $\sum_{0\leqslant k<n} \ln(n-k)(\ln n)^k/k!$ with absolute error $O(n^{-1})$. *Hint:* Show that the terms for $k \geqslant 10\ln n$ are negligible.

40 Let m be a (fixed) positive integer. Evaluate $\sum_{k=1}^{n}(-1)^k H_k^m$ with absolute error $O(1)$.

41 Evaluate the "Fibonacci factorial" $\prod_{k=1}^{n} F_k$ with relative error $O(n^{-1})$ or better. Your answer may involve a constant whose value you do not know in closed form.

42 Let α be a constant in the range $0 < \alpha < \frac{1}{2}$. We've seen in previous chapters that there is no general closed form for the sum $\sum_{k\leqslant\alpha n}\binom{n}{k}$. Show that there is, however, an asymptotic formula

$$\sum_{k\leqslant\alpha n}\binom{n}{k} = 2^{nH(\alpha)-\frac{1}{2}\lg n+O(1)},$$

where $H(\alpha) = \alpha\lg\frac{1}{\alpha}+(1-\alpha)\lg(\frac{1}{1-\alpha})$. *Hint:* Show that $\binom{n}{k-1} < \frac{\alpha}{1-\alpha}\binom{n}{k}$ for $0 < k \leqslant \alpha n$.

43 Show that C_n, the number of ways to change n cents (as considered in Chapter 7) is asymptotically $cn^4 + O(n^3)$ for some constant c. What is that constant?

44 Prove that

$$x^{\underline{1/2}} = x^{1/2}\begin{bmatrix}1/2\\1/2\end{bmatrix} - x^{-1/2}\begin{bmatrix}1/2\\-1/2\end{bmatrix} + x^{-3/2}\begin{bmatrix}1/2\\-3/2\end{bmatrix} + O(x^{-5/2})$$

as $x \to \infty$. (Recall the definition $x^{\underline{1/2}} = x!/(x-\frac{1}{2})!$ in (5.88), and the definition of generalized Stirling numbers in Table 272.)

45 Let α be an irrational number between 0 and 1. Chapter 3 discusses the quantity $D(\alpha, n)$, which measures the maximum discrepancy by which the fractional parts $\{k\alpha\}$ for $0 \leqslant k < n$ deviate from a uniform distribution. The recurrence

$$D(\alpha, n) \leqslant D\big(\{\alpha^{-1}\}, \lfloor\alpha n\rfloor\big) + \alpha^{-1} + 2$$

was proved in (3.31); we also have the obvious bounds

$$0 \leqslant D(\alpha, n) \leqslant n.$$

Prove that $\lim_{n\to\infty} D(\alpha, n)/n = 0$. *Hint:* Chapter 6 discusses continued fractions.

46 Show that the Bell number $\varpi_n = e^{-1}\sum_{k\geqslant 0} k^n/k!$ of exercise 7.15 is asymptotically equal to

$$m(n)^n e^{m(n)-n-1/2}/\sqrt{\ln n}\,,$$

where $m(n)\ln m(n) = n - \frac{1}{2}$, and estimate the relative error in this approximation.

47 Let m be an integer $\geqslant 2$. Analyze the two sums

$$\sum_{k=1}^{n} \lfloor \log_m k \rfloor \qquad \text{and} \qquad \sum_{k=1}^{n} \lceil \log_m k \rceil\,;$$

which is asymptotically closer to $\log_m n!$?

48 Consider a table of the harmonic numbers H_k for $1 \leqslant k \leqslant n$ in decimal notation. The kth entry $\widehat{H}_k$ has been correctly rounded to d_k significant digits, where d_k is just large enough to distinguish this value from the values of H_{k-1} and H_{k+1}. For example, here is an extract from the table, showing five entries where H_k passes 10:

k	H_k	$\widehat{H}_k$	d_k
12364	9.99980041−	9.9998	5
12365	9.99988128+	9.9999	5
12366	9.99996215−	9.99996	6
12367	10.00004301−	10.0000	6
12368	10.00012386+	10.0001	6

Estimate the total number of digits in the table, $\sum_{k=1}^{n} d_k$, with an absolute error of $O(n)$.

49 In Chapter 6 we considered the tale of a worm that reaches the end of a stretching band after n seconds, where $H_{n-1} < 100 \leqslant H_n$. Prove that if n is a positive integer such that $H_{n-1} \leqslant \alpha \leqslant H_n$, then

$$\lfloor e^{\alpha-\gamma} \rfloor \leqslant n \leqslant \lceil e^{\alpha-\gamma} \rceil\,.$$

50 Venture capitalists in Silicon Valley are being offered a deal giving them a chance for an exponential payoff on their investments: For an n million dollar investment, where $n \geqslant 2$, the GKP consortium promises to pay up to N million dollars after one year, where $N = 10^n$. Of course there's some risk; the actual deal is that GKP pays k million dollars with probability $1/(k^2 H_N^{(2)})$, for each integer k in the range $1 \leqslant k \leqslant N$. (All payments are in megabucks, that is, in exact multiples of \$1,000,000; the payoff is determined by a truly random process.) Notice that an investor always gets at least a million dollars back.

a What is the asymptotic expected return after one year, if n million dollars are invested? (In other words, what is the mean value of the payment?) Your answer should be correct within an absolute error of $O(10^{-n})$ dollars.

I once earned $O(10^{-n})$ *dollars.*

b What is the asymptotic probability that you make a profit, if you invest n million? (In other words, what is the chance that you get back more than you put in?) Your answer here should be correct within an absolute error of $O(n^{-3})$.

Bonus problems

51 Prove or disprove: $\int_n^\infty O(x^{-2})\,dx = O(n^{-1})$ as $n \to \infty$.

52 Show that there exists a power series $A(z) = \sum_{k\geqslant 0} a_n z^n$, convergent for all complex z, such that

$$A(n) \succ n^{n^{n^{\cdot^{\cdot^{\cdot^{n}}}}}}\Big\}n.$$

53 Prove that if $f(x)$ is a function whose derivatives satisfy

$$f'(x) \leqslant 0\,,\quad -f''(x) \leqslant 0\,,\quad f'''(x) \leqslant 0\,,\quad \ldots,\quad (-1)^m f^{(m+1)}(x) \leqslant 0$$

for all $x \geqslant 0$, then we have

$$f(x) \;=\; f(0) + \frac{f'(0)}{1!}x + \cdots + \frac{f^{(m-1)}(0)}{(m-1)!}x^{m-1} + O(x^m)\,,\quad \text{for } x \geqslant 0.$$

In particular, the case $f(x) = -\ln(1+x)$ proves (9.64) for all $k, n > 0$.

54 Let $f(x)$ be a positive, differentiable function such that $xf'(x) \prec f(x)$ as $x \to \infty$. Prove that

$$\sum_{k\geqslant n} \frac{f(k)}{k^{1+\alpha}} \;=\; O\left(\frac{f(n)}{n^\alpha}\right), \qquad \text{if } \alpha > 0.$$

Hint: Consider the quantity $f(k-\frac{1}{2})/(k-\frac{1}{2})^\alpha - f(k+\frac{1}{2})/(k+\frac{1}{2})^\alpha$.

55 Improve (9.99) to relative error $O(n^{-3/2+5\epsilon})$.

56 The quantity $Q(n) = 1 + \frac{n-1}{n} + \frac{n-1}{n}\frac{n-2}{n} + \cdots = \sum_{k\geqslant 1} n^{\underline{k}}/n^k$ occurs in the analysis of many algorithms. Find its asymptotic value, with absolute error $o(1)$.

57 An asymptotic formula for Golomb's sum $\sum_{k\geqslant 1} 1/k\lfloor 1+\log_n k\rfloor^2$ is derived in (9.54). Find an asymptotic formula for the analogous sum without floor brackets, $\sum_{k\geqslant 1} 1/k(1+\log_n k)^2$. *Hint:* Consider the integral $\int_0^\infty ue^{-u}k^{-tu}\,du = 1/(1+t\ln k)^2$.

58 Prove that

$$B_m(\{x\}) \;=\; -2\frac{m!}{(2\pi)^m}\sum_{k\geqslant 1}\frac{\cos(2\pi kx-\frac{1}{2}\pi m)}{k^m}\,, \qquad \text{for } m\geqslant 2,$$

by using residue calculus, integrating

$$\frac{1}{2\pi i}\oint \frac{2\pi i\, e^{2\pi i z\theta}}{e^{2\pi i z}-1}\frac{dz}{z^m}$$

on the square contour $z = x+iy$, where $\max\bigl(|x|,|y|\bigr) = M+\frac{1}{2}$, and letting the integer M tend to ∞.

59 Let $\Theta_n(t) = \sum_k e^{-(k+t)^2/n}$, a periodic function of t. Show that the expansion of $\Theta_n(t)$ as a Fourier series is

$$\Theta_n(t) \;=\; \sqrt{\pi n}\bigl(1+2e^{-\pi^2 n}(\cos 2\pi t)+2e^{-4\pi^2 n}(\cos 4\pi t) \\ +2e^{-9\pi^2 n}(\cos 6\pi t)+\cdots\bigr)\,.$$

(This formula gives a rapidly convergent series for the sum $\Theta_n = \Theta_n(0)$ in equation (9.93).)

60 Explain why the coefficients in the asymptotic expansion

$$\binom{2n}{n} \;=\; \frac{4^n}{\sqrt{\pi n}}\left(1-\frac{1}{8n}+\frac{1}{128n^2}+\frac{5}{1024n^3}-\frac{21}{32768n^4}+O(n^{-5})\right)$$

all have denominators that are powers of 2.

61 Exercise 45 proves that the discrepancy $D(\alpha,n)$ is $o(n)$ for all irrational numbers α. Exhibit an irrational α such that $D(\alpha,n)$ is *not* $O(n^{1-\epsilon})$ for any $\epsilon > 0$.

62 Given n, let $\left\{{n \atop m(n)}\right\} = \max_k \left\{{n \atop k}\right\}$ be the largest entry in row n of Stirling's subset triangle. Show that for all sufficiently large n, we have $m(n) = \lfloor\overline{m}(n)\rfloor$ or $m(n) = \lceil\overline{m}(n)\rceil$, where

$$\overline{m}(n)(\overline{m}(n)+2)\ln(\overline{m}(n)+2) \;=\; n(\overline{m}(n)+1)\,.$$

Hint: This is difficult.

63 Prove that S. W. Golomb's self-describing sequence of exercise 2.36 satisfies $f(n) = \phi^{2-\phi}n^{\phi-1}+O(n^{\phi-1}/\log n)$.

64 Find a proof of the identity

$$\sum_{n\geqslant 1}\frac{\cos 2n\pi x}{n^2} \;=\; \pi^2\bigl(x^2-x+\tfrac{1}{6}\bigr) \qquad \text{for } 0\leqslant x\leqslant 1,$$

that uses only "Eulerian" (eighteenth-century) mathematics.

65 What are the coefficients of the asymptotic series

$$1+\frac{1}{n-1}+\frac{1}{(n-1)(n-2)}+\cdots+\frac{1}{(n-1)!} \;=\; a_0+\frac{a_1}{n}+\frac{a_2}{n^2}+\cdots\ ?$$

Research problems

66 Find a "combinatorial" proof of Stirling's approximation. (Note that n^n is the number of mappings of $\{1,2,\ldots,n\}$ into itself, and $n!$ is the number of mappings of $\{1,2,\ldots,n\}$ onto itself.)

67 Consider an $n \times n$ array of dots, $n \geqslant 3$, in which each dot has four neighbors. (At the edges we "wrap around" modulo n.) Let χ_n be the number of ways to assign the colors red, white, and blue to these dots in such a way that no neighboring dots have the same color. (Thus $\chi_3 = 12$.) Prove that

$$\chi_n \;\sim\; \left(\tfrac{4}{3}\right)^{3n^2/2} e^{-\pi/6}\,.$$

68 Let Q_n be the least integer m such that $H_m > n$. Find the smallest integer n such that $Q_n \neq \lfloor e^{n-\gamma} + \frac{1}{2} \rfloor$, or prove that no such n exist.

Th-th-th-that's all, folks!

A

Answers to Exercises

EVERY EXERCISE is answered here (at least briefly), and some of these answers go beyond what was asked. Readers will learn best if they make a serious attempt to find their own answers BEFORE PEEKING at this appendix.

The authors will be interested to learn of any solutions (or partial solutions) to the research problems, or of any simpler (or more correct) ways to solve the non-research ones.

(The first finder of every error in this book will receive a reward of $2.56.)

Does that mean I have to find every error?

(We meant to say "any error.")

Does that mean only one person gets a reward?

(Hmmm. Try it and see.)

1.1 The proof is fine except when $n = 2$. If all sets of two horses have horses of the same color, the statement is true for any number of horses.

1.2 If X_n is the number of moves, we have $X_0 = 0$ and $X_n = X_{n-1} + 1 + X_{n-1} + 1 + X_{n-1}$ when $n > 0$. It follows (for example by adding 1 to both sides) that $X_n = 3^n - 1$. (After $\frac{1}{2}X_n$ moves, it turns out that the entire tower will be on the middle peg, halfway home!)

1.3 There are 3^n possible arrangements, since each disk can be on any of the pegs. We must hit them all, since the shortest solution takes $3^n - 1$ moves. (This construction is equivalent to a "ternary Gray code," which runs through all numbers from $(0\ldots0)_3$ to $(2\ldots2)_3$, changing only one digit at a time.)

1.4 No. If the largest disk doesn't have to move, $2^{n-1} - 1$ moves will suffice (by induction); otherwise $(2^{n-1} - 1) + 1 + (2^{n-1} - 1)$ will suffice (again by induction).

The number of intersection points turns out to give the whole story; convexity was a red herring.

1.5 No; different circles can intersect in at most two points, so the fourth circle can increase the number of regions to at most 14. However, it is possible to do the job with ovals:

Venn [359] claimed that there is no way to do the five-set case with ellipses, but a five-set construction with ellipses was found by Grünbaum [167].

1.6 If the nth line intersects the previous lines in $k > 0$ distinct points, we get $k-1$ new bounded regions (assuming that none of the previous lines were mutually parallel) and two new infinite regions. Hence the maximum number of bounded regions is $(n-2)+(n-3)+\cdots = S_{n-2} = (n-1)(n-2)/2 = L_n - 2n$.

This answer assumes that $n > 0$.

1.7 The basis is unproved; and in fact, $H(1) \neq 2$.

1.8 $Q_2 = (1+\beta)/\alpha$; $Q_3 = (1+\alpha+\beta)/\alpha\beta$; $Q_4 = (1+\alpha)/\beta$; $Q_5 = \alpha$; $Q_6 = \beta$. So the sequence is periodic!

1.9 (a) We get $P(n-1)$ from the inequality

$$x_1 \ldots x_{n-1} \left(\frac{x_1 + \cdots + x_{n-1}}{n-1} \right) \leqslant \left(\frac{x_1 + \cdots + x_{n-1}}{n-1} \right)^n .$$

(b) $x_1 \ldots x_n x_{n+1} \ldots x_{2n} \leqslant \big(((x_1 + \cdots + x_n)/n)((x_{n+1} + \cdots + x_{2n})/n)\big)^n$ by $P(n)$; the product inside is $\leqslant \big((x_1 + \cdots + x_{2n})/2n\big)^2$ by $P(2)$. (c) For example, $P(5)$ follows from $P(6)$ from $P(3)$ from $P(4)$ from $P(2)$.

1.10 First show that $R_n = R_{n-1} + 1 + Q_{n-1} + 1 + R_{n-1}$, when $n > 0$. Incidentally, the methods of Chapter 7 will tell us that $Q_n = \big((1+\sqrt{3}\,)^{n+1} - (1-\sqrt{3}\,)^{n+1}\big)/\big(2\sqrt{3}\big) - 1$.

1.11 (a) We cannot do better than to move a double $(n-1)$-tower, then move (and invert the order of) the two largest disks, then move the double $(n-1)$-tower again; hence $A_n = 2A_{n-1} + 2$ and $A_n = 2T_n = 2^{n+1} - 2$. This solution interchanges the two largest disks but returns the other $2n-2$ to their original order.

(b) Let B_n be the minimum number of moves. Then $B_1 = 3$, and it can be shown that no strategy does better than $B_n = A_{n-1} + 2 + A_{n-1} + 2 + B_{n-1}$ when $n > 1$. Hence $B_n = 2^{n+2} - 5$, for all $n > 0$. Curiously this is just $2A_n - 1$, and we also have $B_n = A_{n-1} + 1 + A_{n-1} + 1 + A_{n-1} + 1 + A_{n-1}$.

1.12 If all $m_k > 0$, then $A(m_1, \ldots, m_n) = 2A(m_1, \ldots, m_{n-1}) + m_n$. This is an equation of the "generalized Josephus" type, with solution $(m_1 \ldots m_n)_2 = 2^{n-1}m_1 + \cdots + 2m_{n-1} + m_n$.

Incidentally, the corresponding generalization of exercise 11b appears to satisfy the recurrence

$$B(m_1, \ldots, m_n) = \begin{cases} A(m_1, \ldots, m_n), & \text{if } m_n = 1; \\ 2m_n - 1, & \text{if } n = 1; \\ 2A(m_1, \ldots, m_{n-1}) + 2m_n & \\ \quad + B(m_1, \ldots, m_{n-1}), & \text{if } n > 1 \text{ and } m_n > 1. \end{cases}$$

1.13 Given n straight lines that define L_n regions, we can replace them by extremely narrow zig-zags with segments sufficiently long that there are nine intersections between each pair of zig-zags. This shows that $ZZ_n = ZZ_{n-1}+9n-8$, for all $n > 0$; consequently $ZZ_n = 9S_n-8n+1 = \frac{9}{2}n^2-\frac{7}{2}n+1$.

1.14 The number of new 3-dimensional regions defined by each new cut is the number of 2-dimensional regions defined in the new plane by its intersections with the previous planes. Hence $P_n = P_{n-1} + L_{n-1}$, and it turns out that $P_5 = 26$. (Six cuts in a cubical piece of cheese can make 27 cubelets, or up to $P_6 = 42$ cuts of weirder shapes.)

Incidentally, the solution to this recurrence fits into a nice pattern if we express it in terms of binomial coefficients (see Chapter 5):

$$\begin{aligned} X_n &= \binom{n}{0} + \binom{n}{1}; \\ L_n &= \binom{n}{0} + \binom{n}{1} + \binom{n}{2}; \\ P_n &= \binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \binom{n}{3}. \end{aligned}$$

I bet I know what happens in four dimensions!

Here X_n is the maximum number of 1-dimensional regions definable by n points on a line.

1.15 The function I satisfies the same recurrence as J when $n > 1$, but $I(1)$ is undefined. Since $I(2) = 2$ and $I(3) = 1$, there's no value of $I(1) = \alpha$ that will allow us to use our general method; the "end game" of unfolding depends on the two leading bits in n's binary representation.

If $n = 2^m + 2^{m-1} + k$, where $0 \leqslant k < 2^{m+1} + 2^m - (2^m + 2^{m-1}) = 2^m+2^{m-1}$, the solution is $I(n) = 2k+1$ for all $n > 2$. Another way to express this, in terms of the representation $n = 2^m + l$, is to say that

$$I(n) = \begin{cases} J(n) + 2^{m-1}, & \text{if } 0 \leqslant l < 2^{m-1}; \\ J(n) - 2^m, & \text{if } 2^{m-1} \leqslant l < 2^m. \end{cases}$$

1.16 Let $g(n) = a(n)\alpha + b(n)\beta_0 + c(n)\beta_1 + d(n)\gamma$. We know from (1.18) that $a(n)\alpha + b(n)\beta_0 + c(n)\beta_1 = (\alpha\,\beta_{b_{m-1}}\,\beta_{b_{m-2}} \ldots \beta_{b_1}\,\beta_{b_0})_3$ when $n = (1\,b_{m-1} \ldots b_1\,b_0)_2$; this defines $a(n)$, $b(n)$, and $c(n)$. Setting $g(n) = n$ in the recurrence implies that $a(n) + c(n) - d(n) = n$; hence we know everything. [Setting $g(n) = 1$ gives the additional identity $a(n) - 2b(n) - 2c(n) = 1$, which can be used to define $b(n)$ in terms of the simpler functions $a(n)$ and $a(n) + c(n)$.]

1.17 In general we have $W_m \leqslant 2W_{m-k} + T_k$, for $0 \leqslant k \leqslant m$. (This relation corresponds to transferring the top $m - k$, then using only three pegs to

move the bottom k, then finishing with the top $m-k$.) The stated relation turns out to be based on the unique value of k that minimizes the right-hand side of this general inequality, when $m = n(n+1)/2$. (However, we cannot conclude that equality holds; many other strategies for transferring the tower are conceivable.) If we set $Y_n = (W_{n(n+1)/2} - 1)/2^n$, we find that $Y_n \leqslant Y_{n-1} + 1$; hence $W_{n(n+1)/2} \leqslant 2^n(n-1) + 1$.

1.18 It suffices to show that both of the lines from $(n^{2j}, 0)$ intersect both of the lines from $(n^{2k}, 0)$, and that all these intersection points are distinct.

A line from $(x_j, 0)$ through $(x_j - a_j, 1)$ intersects a line from $(x_k, 0)$ through $(x_k - a_k, 1)$ at the point $(x_j - ta_j, t)$ where $t = (x_k - x_j)/(a_k - a_j)$. Let $x_j = n^{2j}$ and $a_j = n^j + (0 \text{ or } n^{-n})$. Then the ratio $t = (n^{2k} - n^{2j})/\bigl(n^k - n^j + (-n^{-n} \text{ or } 0 \text{ or } n^{-n})\bigr)$ lies strictly between $n^j + n^k - 1$ and $n^j + n^k + 1$; hence the y coordinate of the intersection point uniquely identifies j and k. Also the four intersections that have the same j and k are distinct.

1.19 Not when $n > 11$. A bent line whose half-lines run at angles θ and $\theta + 30^\circ$ from its apex can intersect four times with another whose half-lines run at angles ϕ and $\phi + 30^\circ$ only if $|\theta - \phi| > 30^\circ$. We can't choose more than 11 angles this far apart from each other. (Is it possible to choose 11?)

1.20 Let $h(n) = a(n)\alpha + b(n)\beta_0 + c(n)\beta_1 + d(n)\gamma_0 + e(n)\gamma_1$. We know from (1.18) that $a(n)\alpha + b(n)\beta_0 + c(n)\beta_1 = (\alpha\,\beta_{b_{m-1}}\,\beta_{b_{m-2}} \ldots \beta_{b_1}\,\beta_{b_0})_4$ when $n = (1\,b_{m-1} \ldots b_1\,b_0)_2$; this defines $a(n)$, $b(n)$, and $c(n)$. Setting $h(n) = n$ in the recurrence implies that $a(n) + c(n) - 2d(n) - 2e(n) = n$; setting $h(n) = n^2$ implies that $a(n) + c(n) + 4e(n) = n^2$. Hence $d(n) = \bigl(3a(n) + 3c(n) - n^2 - 2n\bigr)/4$; $e(n) = \bigl(n^2 - a(n) - c(n)\bigr)/4$.

1.21 We can let m be the least (or any) common multiple of $2n$, $2n-1$, $\ldots$, $n+1$. [A non-rigorous argument suggests that a "random" value of m will succeed with probability

$$\frac{n}{2n}\frac{n-1}{2n-1}\cdots\frac{1}{n+1} = 1\Bigg/\binom{2n}{n} \sim \frac{\sqrt{\pi n}}{4^n},$$

so we might expect to find such an m less than 4^n.]

1.22 Take a regular polygon with 2^n sides and label the sides with the elements of a "de Bruijn cycle" of length 2^n. (This is a cyclic sequence of 0's and 1's in which all n-tuples of adjacent elements are different; see [207, exercise 2.3.4.2–23] and [208, exercise 3.2.2–17].) Attach a very thin convex extension to each side that's labeled 1. The n sets are copies of the resulting polygon, rotated by the length of k sides for $k = 0, 1, \ldots, n-1$.

I once rode a de Bruijn cycle (when visiting at his home in Nuenen, The Netherlands).

1.23 Yes. (We need principles of elementary number theory from Chapter 4.) Let $L(n) = \operatorname{lcm}(1, 2, \ldots, n)$. We can assume that $n > 2$; hence by Bertrand's postulate there is a prime p between $n/2$ and n. We can also

assume that $j > n/2$, since $q' = L(n) + 1 - q$ leaves $j' = n + 1 - j$ if and only if q leaves j. Choose q so that $q \equiv 1 \pmod{L(n)/p}$ and $q \equiv j + 1 - n \pmod{p}$. The people are now removed in order $1, 2, \ldots, n - p, j + 1, j + 2, \ldots, n, n - p + 1, \ldots, j - 1$.

1.24 The only known examples are: $X_n = 1/X_{n-1}$, which has period 2; Gauss's recurrence of period 5 in exercise 8; H. Todd's even more remarkable recurrence $X_n = (1+X_{n-1}+X_{n-2})/X_{n-3}$, which has period 8 (see [261]); and recurrences derived from these when we replace X_n by a constant times X_{mn}. We can assume that the first nonzero coefficient in the denominator is unity, and that the first nonzero coefficient in the numerator (if any) has nonnegative real part. Computer algebra shows easily that there are no further solutions of period $\leqslant 5$ when $k = 2$. A partial theory has been developed by Lyness [261, 262] and by Kurshan and Gopinath [231].

An interesting example of another type, with period 9 when the starting values are real, is the recurrence $X_n = |X_{n-1}| - X_{n-2}$ discovered by Morton Brown [43]. Nonlinear recurrences having any desired period $\geqslant 5$ can be based on continuants [65].

1.25 If $T^{(k)}(n)$ denotes the minimum number of moves needed to transfer n disks with k auxiliary pegs (hence $T^{(1)}(n) = T_n$ and $T^{(2)}(n) = W_n$), we have $T^{(k)}(\binom{n+1}{k}) \leqslant 2T^{(k)}(\binom{n}{k}) + T^{(k-1)}(\binom{n}{k-1})$. No examples (n, k) are known where this inequality fails to be an equality. When k is small compared with n, the formula $2^{n+1-k}\binom{n-1}{k-1}$ gives a convenient (but non-optimum) upper bound on $T^{(k)}(\binom{n}{k})$.

1.26 The execution-order permutation can be computed in $O(n \log n)$ steps for all m and n [209, exercises 5.1.1–2 and 5.1.1–5]. Bjorn Poonen has proved that non-Josephus sets with exactly four "bad guys" exist whenever $n \equiv 0 \pmod 3$ and $n \geqslant 9$; in fact, the number of such sets is at least $\epsilon\binom{n}{4}$ for some $\epsilon > 0$. He also found by extensive computations that the only other $n < 24$ with non-Josephus sets is $n = 20$, which has 236 such sets with $k = 14$ and two with $k = 13$. (One of the latter is $\{1, 2, 3, 4, 5, 6, 7, 8, 11, 14, 15, 16, 17\}$; the other is its reflection with respect to 21.) There is a unique non-Josephus set with $n = 15$ and $k = 9$, namely $\{3, 4, 5, 6, 8, 10, 11, 12, 13\}$.

2.1 There's no agreement about this; three answers are defensible: (1) We can say that $\sum_{k=m}^{n} q_k$ is always equivalent to $\sum_{m \leqslant k \leqslant n} q_k$; then the stated sum is zero. (2) A person might say that the given sum is $q_4 + q_3 + q_2 + q_1 + q_0$, by summing over decreasing values of k. But this conflicts with the generally accepted convention that $\sum_{k=1}^{n} q_k = 0$ when $n = 0$. (3) We can say that $\sum_{k=m}^{n} q_k = \sum_{k \leqslant n} q_k - \sum_{k<m} q_k$; then the stated sum is equal to $-q_1 - q_2 - q_3$. This convention may appear strange, but it obeys the useful law $\sum_{k=a}^{b} + \sum_{k=b+1}^{c} = \sum_{k=a}^{c}$ for all a, b, c.

It's best to use the notation $\sum_{k=m}^{n}$ only when $n-m \geqslant -1$; then both conventions (1) and (3) agree.

2.2 This is $|x|$. Incidentally, the quantity $\big([x>0]-[x<0]\big)$ is often called $\operatorname{sign}(x)$ or $\operatorname{signum}(x)$; it is $+1$ when $x>0$, 0 when $x=0$, and -1 when $x<0$.

2.3 The first sum is, of course, $a_0+a_1+a_2+a_3+a_4+a_5$; the second is $a_4+a_1+a_0+a_1+a_4$, because the sum is over the values $k \in \{-2,-1,0,+1,+2\}$. The commutative law doesn't hold here because the function $p(k)=k^2$ is not a permutation. Some values of n (e.g., $n=3$) have no k such that $p(k)=n$; others (e.g., $n=4$) have two such k.

2.4 (a) $\sum_{i=1}^{4}\sum_{j=i+1}^{4}\sum_{k=j+1}^{4} a_{ijk} = \sum_{i=1}^{2}\sum_{j=i+1}^{3}\sum_{k=j+1}^{4} a_{ijk} = \big((a_{123}+a_{124})+a_{134}\big)+a_{234}$.

(b) $\sum_{k=1}^{4}\sum_{j=1}^{k-1}\sum_{i=1}^{j-1} a_{ijk} = \sum_{k=3}^{4}\sum_{j=2}^{k-1}\sum_{i=1}^{j-1} a_{ijk} = a_{123}+\big(a_{124}+(a_{134}+a_{234})\big)$.

2.5 The same index 'k' is being used for two different index variables, although k is bound in the inner sum. This is a famous mistake in mathematics (and computer programming). The result turns out to be correct if $a_j = a_k$ for all j and k, $1 \leqslant j,k \leqslant n$.

2.6 It's $[1 \leqslant j \leqslant n](n-j+1)$. The first factor is necessary here because we should get zero when $j<1$ or $j>n$.

2.7 $mx^{\overline{m-1}}$. A version of finite calculus based on ∇ instead of Δ would therefore give special prominence to *rising* factorial powers.

2.8 0, if $m \geqslant 1$; $1/|m|!$, if $m \leqslant 0$.

2.9 $x^{\overline{m+n}} = x^{\overline{m}}(x+m)^{\overline{n}}$, for integers m and n. Setting $m=-n$ tells us that $x^{\overline{-n}} = 1/(x-n)^{\overline{n}} = 1/(x-1)^{\underline{n}}$.

2.10 Another possible right-hand side is $Eu\,\Delta v + v\,\Delta u$.

2.11 Break the left-hand side into two sums, and change k to $k+1$ in the second of these.

2.12 If $p(k)=n$ then $n+c = k+\big((-1)^k+1\big)c$ and $\big((-1)^k+1\big)$ is even; hence $(-1)^{n+c}=(-1)^k$ and $k = n-(-1)^{n+c}c$. Conversely, this value of k yields $p(k)=n$.

2.13 Let $R_0=\alpha$, and $R_n = R_{n-1}+(-1)^n(\beta+n\gamma+n^2\delta)$ for $n>0$. Then $R(n)=A(n)\alpha+B(n)\beta+C(n)\gamma+D(n)\delta$. Setting $R_n=1$ yields $A(n)=1$. Setting $R_n=(-1)^n$ yields $A(n)+2B(n)=(-1)^n$. Setting $R_n=(-1)^n n$ yields $-B(n)+2C(n)=(-1)^n n$. Setting $R_n=(-1)^n n^2$ yields $B(n)-2C(n)+2D(n)=(-1)^n n^2$. Therefore $2D(n)=(-1)^n(n^2+n)$; the stated sum is $D(n)$.

2.14 The suggested rewrite is legitimate since we have $k = \sum_{1 \leqslant j \leqslant k} 1$ when $1 \leqslant k \leqslant n$. Sum first on k; the multiple sum reduces to

$$\sum_{1 \leqslant j \leqslant n} (2^{n+1} - 2^j) \;=\; n2^{n+1} - (2^{n+1} - 2)\,.$$

2.15 The first step replaces $k(k+1)$ by $2\sum_{1 \leqslant j \leqslant k} j$. The second step gives $\boxed{}_n + \square_n = \left(\sum_{k=1}^{n} k\right)^2 + \square_n$.

2.16 $x^{\underline{m}}(x-m)^{\underline{n}} = x^{\underline{m+n}} = x^{\underline{n}}(x-n)^{\underline{m}}$, by (2.52).

2.17 Use induction for the first two $=$'s, and (2.52) for the third. The second line follows from the first.

2.18 Use the facts that $(\Re z)^+ \leqslant |z|$, $(\Re z)^- \leqslant |z|$, $(\Im z)^+ \leqslant |z|$, $(\Im z)^- \leqslant |z|$, and $|z| \leqslant (\Re z)^+ + (\Re z)^- + (\Im z)^+ + (\Im z)^-$.

2.19 Multiply both sides by $2^{n-1}/n!$ and let $S_n = 2^n T_n/n! = S_{n-1} + 3 \cdot 2^{n-1} = 3(2^n - 1) + S_0$. The solution is $T_n = 3 \cdot n! + n!/2^{n-1}$. (We'll see in Chapter 4 that T_n is an integer only when n is 0 or a power of 2.)

"It is a profoundly erroneous truism, repeated by all copybooks and by eminent people when they are making speeches, that we should cultivate the habit of thinking of what we are doing. The precise opposite is the case. Civilization advances by extending the number of important operations which we can perform without thinking about them. Operations of thought are like cavalry charges in a battle—they are strictly limited in number, they require fresh horses, and must only be made at decisive moments."
—A. N. Whitehead [370]

2.20 The perturbation method gives

$$S_n + (n+1)H_{n+1} \;=\; S_n + \left(\sum_{0 \leqslant k \leqslant n} H_k\right) + n + 1\,.$$

2.21 Extracting the final term of S_{n+1} gives $S_{n+1} = 1 - S_n$; extracting the first term gives

$$\begin{aligned} S_{n+1} \;&=\; (-1)^{n+1} + \sum_{1 \leqslant k \leqslant n+1} (-1)^{n+1-k} \;=\; (-1)^{n+1} + \sum_{0 \leqslant k \leqslant n} (-1)^{n-k} \\ &=\; (-1)^{n+1} + S_n\,. \end{aligned}$$

Hence $2S_n = 1 + (-1)^n$ and we have $S_n = [n \text{ is even}]$. Similarly, we find

$$T_{n+1} \;=\; n + 1 - T_n \;=\; \sum_{k=0}^{n} (-1)^{n-k}(k+1) \;=\; T_n + S_n\,,$$

hence $2T_n = n + 1 - S_n$ and we have $T_n = \frac{1}{2}\big(n + [n \text{ is odd}]\big)$. Finally, the same approach yields

$$\begin{aligned} U_{n+1} \;=\; (n+1)^2 - U_n \;&=\; U_n + 2T_n + S_n \\ &=\; U_n + n + [n \text{ is odd}] + [n \text{ is even}] \\ &=\; U_n + n + 1\,. \end{aligned}$$

Hence U_n is the triangular number $\frac{1}{2}(n+1)n$.

2.22 Twice the general sum gives a "vanilla" sum over $1 \leqslant j, k \leqslant n$, which splits and yields twice $\bigl(\sum_k a_k A_k\bigr)\bigl(\sum_k b_k B_k\bigr) - \bigl(\sum_k a_k B_k\bigr)\bigl(\sum_k b_k A_k\bigr)$.

2.23 (a) This approach gives four sums that evaluate to $2n + H_n - 2n + (H_n + \frac{1}{n+1} - 1)$. (It would have been easier to replace the summand by $1/k + 1/(k+1)$.) (b) Let $u(x) = 2x + 1$ and $\Delta v(x) = 1/x(x+1) = (x-1)^{\underline{-2}}$; then $\Delta u(x) = 2$ and $v(x) = -(x-1)^{\underline{-1}} = -1/x$. The answer is $2H_n - \frac{n}{n+1}$.

2.24 Summing by parts, $\sum x^{\underline{m}} H_x \, \delta x = x^{\underline{m+1}} H_x/(m+1) - x^{\underline{m+1}}/(m+1)^2 + C$; hence $\sum_{0 \leqslant k < n} k^{\underline{m}} H_k = n^{\underline{m+1}}\bigl(H_n - 1/(m+1)\bigr)/(m+1) + 0^{\underline{m+1}}/(m+1)^2$. In our case $m = -2$, so the sum comes to $1 - (H_n + 1)/(n+1)$.

2.25 Here are some of the basic analogies:

$$\sum_{k\in K} c a_k = c \sum_{k\in K} a_k \qquad \longleftrightarrow \qquad \prod_{k\in K} a_k^c = \Bigl(\prod_{k\in K} a_k\Bigr)^c$$

$$\sum_{k\in K} (a_k + b_k) = \sum_{k\in K} a_k + \sum_{k\in K} b_k \qquad \longleftrightarrow \qquad \prod_{k\in K} a_k b_k = \Bigl(\prod_{k\in K} a_k\Bigr)\Bigl(\prod_{k\in K} b_k\Bigr)$$

$$\sum_{k\in K} a_k = \sum_{p(k)\in K} a_{p(k)} \qquad \longleftrightarrow \qquad \prod_{k\in K} a_k = \prod_{p(k)\in K} a_{p(k)}$$

$$\sum_{\substack{j\in J\\ k\in K}} a_{j,k} = \sum_{j\in J} \sum_{k\in K} a_{j,k} \qquad \longleftrightarrow \qquad \prod_{\substack{j\in J\\ k\in K}} a_{j,k} = \prod_{j\in J} \prod_{k\in K} a_{j,k}$$

$$\sum_{k\in K} a_k = \sum_{k} a_k [k \in K] \qquad \longleftrightarrow \qquad \prod_{k\in K} a_k = \prod_{k} a_k^{[k\in K]}$$

$$\sum_{k\in K} 1 = \#K \qquad \longleftrightarrow \qquad \prod_{k\in K} c = c^{\#K}$$

2.26 $P^2 = \bigl(\prod_{1\leqslant j,k\leqslant n} a_j a_k\bigr)\bigl(\prod_{1\leqslant j=k\leqslant n} a_j a_k\bigr)$. The first factor is equal to $\bigl(\prod_{k=1}^n a_k^n\bigr)^2$; the second factor is $\prod_{k=1}^n a_k^2$. Hence $P = \bigl(\prod_{k=1}^n a_k\bigr)^{n+1}$.

2.27 $\Delta(c^{\underline{x}}) = c^{\underline{x}}(c - x - 1) = c^{\underline{x+2}}/(c - x)$. Setting $c = -2$ and decreasing x by 2 yields $\Delta(-(-2)^{\underline{x-2}}) = (-2)^{\underline{x}}/x$, hence the stated sum is $(-2)^{\underline{-1}} - (-2)^{\underline{n-1}} = (-1)^n n! - 1$.

2.28 The interchange of summation between the second and third lines is not justifiable; the terms of this sum do not converge absolutely. Everything else is perfectly correct, except that the result of $\sum_{k\geqslant 1} [k = j-1] k/j$ should perhaps have been written $[j - 1 \geqslant 1](j-1)/j$ and simplified explicitly.

As opposed to imperfectly correct.

2.29 Use partial fractions to get

$$\frac{k}{4k^2 - 1} = \frac{1}{4}\left(\frac{1}{2k+1} + \frac{1}{2k-1}\right).$$

The $(-1)^k$ factor now makes the two halves of each term cancel with their neighbors. Hence the answer is $-1/4 + (-1)^n/(8n+4)$.

2.30 $\sum_a^b x\,\delta x = \frac{1}{2}(b^{\underline{2}} - a^{\underline{2}}) = \frac{1}{2}(b-a)(b+a-1)$. So we have

$$(b-a)(b+a-1) \;=\; 2100 \;=\; 2^2 \cdot 3 \cdot 5^2 \cdot 7\,.$$

There is one solution for each way to write $2100 = x \cdot y$ where x is even and y is odd; we let $a = \frac{1}{2}|x-y| + \frac{1}{2}$ and $b = \frac{1}{2}(x+y) + \frac{1}{2}$. So the number of solutions is the number of divisors of $3 \cdot 5^2 \cdot 7$, namely 12. In general, there are $\prod_{p>2}(n_p+1)$ ways to represent $\prod_p p^{n_p}$, where the products range over primes.

2.31 $\sum_{j,k\geqslant 2} j^{-k} = \sum_{j\geqslant 2} 1/j^2(1-1/j) = \sum_{j\geqslant 2} 1/j(j-1)$. The second sum is, similarly, $3/4$.

2.32 If $2n \leqslant x < 2n+1$, the sums are $0+\cdots+n+(x-n-1)+\cdots+(x-2n) = n(x-n) = (x-1)+(x-3)+\cdots+(x-2n+1)$. If $2n-1 \leqslant x < 2n$ they are, similarly, both equal to $n(x-n)$. (Looking ahead to Chapter 3, the formula $\lfloor\frac{1}{2}(x+1)\rfloor\big(x - \lfloor\frac{1}{2}(x+1)\rfloor\big)$ covers both cases.)

2.33 If K is empty, $\bigwedge_{k\in K} a_k = \infty$. The basic analogies are:

$$\sum_{k\in K} ca_k \;=\; c\sum_{k\in K} a_k \qquad\longleftrightarrow\qquad \bigwedge_{k\in K}(c+a_k) \;=\; c + \bigwedge_{k\in K} a_k$$

$$\sum_{k\in K}(a_k+b_k) \;=\; \sum_{k\in K} a_k + \sum_{k\in K} b_k \qquad\longleftrightarrow\qquad \bigwedge_{k\in K}\min(a_k,b_k) \;=\; \min\Big(\bigwedge_{k\in K} a_k, \bigwedge_{k\in K} b_k\Big)$$

$$\sum_{k\in K} a_k \;=\; \sum_{p(k)\in K} a_{p(k)} \qquad\longleftrightarrow\qquad \bigwedge_{k\in K} a_k \;=\; \bigwedge_{p(k)\in K} a_{p(k)}$$

$$\sum_{\substack{j\in J\\ k\in K}} a_{j,k} \;=\; \sum_{j\in J}\sum_{k\in K} a_{j,k} \qquad\longleftrightarrow\qquad \bigwedge_{\substack{j\in J\\ k\in K}} a_{j,k} \;=\; \bigwedge_{j\in J}\bigwedge_{k\in K} a_{j,k}$$

$$\sum_{k\in K} a_k \;=\; \sum_k a_k[k\in K] \qquad\longleftrightarrow\qquad \bigwedge_{k\in K} a_k \;=\; \bigwedge_k a_k \cdot \infty^{[k\notin K]}$$

A permutation that consumes terms of one sign faster than those of the other can steer the sum toward any value that it likes.

2.34 Let $K^+ = \{k \mid a_k \geqslant 0\}$ and $K^- = \{k \mid a_k < 0\}$. Then if, for example, n is odd, we choose F_n to be $F_{n-1} \cup E_n$, where $E_n \subseteq K^-$ is sufficiently large that $\sum_{k\in(F_{n-1}\cap K^+)} a_k - \sum_{k\in E_n}(-a_k) < A^-$.

2.35 Goldbach's sum can be shown to equal

$$\sum_{m,n\geqslant 2} m^{-n} \;=\; \sum_{m\geqslant 2}\frac{1}{m(m-1)} \;=\; 1$$

as follows: By unsumming a geometric series, it equals $\sum_{k\in P,\, l\geqslant 1} k^{-l}$; therefore the proof will be complete if we can find a one-to-one correspondence between ordered pairs (m,n) with $m,n \geqslant 2$ and ordered pairs (k,l) with $k\in P$ and $l\geqslant 1$, where $m^n = k^l$ when the pairs correspond. If $m\notin P$ we let $(m,n) \longleftrightarrow (m^n,1)$; but if $m = a^b \in P$, we let $(m,n)\longleftrightarrow(a^n,b)$.

2.36 (a) By definition, $g(n)-g(n-1)=f(n)$. (b) By part (a), $g\bigl(g(n)\bigr)-g\bigl(g(n-1)\bigr)=\sum_k f(k)\bigl[g(n-1)<k\leqslant g(n)\bigr]=n\bigl(g(n)-g(n-1)\bigr)=nf(n)$. (c) By part (a) again, $g\bigl(g(g(n))\bigr)-g\bigl(g(g(n-1))\bigr)$ is

With this self-description, Golomb's sequence wouldn't do too well on the Dating Game.

$$\begin{aligned}
&\sum_k f(k)\bigl[g(g(n-1))<k\leqslant g(g(n))\bigr]\\
&\quad=\sum_{j,k} j\,\bigl[j=f(k)\bigr]\bigl[g(g(n-1))<k\leqslant g(g(n))\bigr]\\
&\quad=\sum_{j,k} j\,\bigl[j=f(k)\bigr]\bigl[g(n-1)<j\leqslant g(n)\bigr]\\
&\quad=\sum_{j} j\,\bigl(g(j)-g(j-1)\bigr)\bigl[g(n-1)<j\leqslant g(n)\bigr]\\
&\quad=\sum_{j} jf(j)\,\bigl[g(n-1)<j\leqslant g(n)\bigr] \;=\; n\sum_j j\,\bigl[g(n-1)<j\leqslant g(n)\bigr].
\end{aligned}$$

Colin Mallows observes that the sequence can also be defined by the recurrence

$$f(1) \;=\; 1; \qquad f(n+1) \;=\; 1+f\bigl(n+1-f(f(n))\bigr), \quad \text{for } n\geqslant 0.$$

2.37 (RLG thinks they probably won't fit; DEK thinks they probably will; OP is not committing himself.)

3.1 $m=\lfloor \lg n\rfloor$; $l=n-2^m=n-2^{\lfloor \lg n\rfloor}$.

3.2 (a) $\lfloor x+.5\rfloor$. (b) $\lceil x-.5\rceil$.

3.3 This is $\lfloor mn-\{m\alpha\}n/\alpha\rfloor = mn-1$, since $0<\{m\alpha\}<1$.

3.4 Something where no proof is required, only a lucky guess (I guess).

3.5 We have $\lfloor nx\rfloor = \bigl\lfloor n\lfloor x\rfloor + n\{x\}\bigr\rfloor = n\lfloor x\rfloor + \bigl\lfloor n\{x\}\bigr\rfloor$ by (3.8) and (3.6). Therefore $\lfloor nx\rfloor = n\lfloor x\rfloor \iff \bigl\lfloor n\{x\}\bigr\rfloor = 0 \iff 0\leqslant n\{x\}<1 \iff \{x\}<1/n$, assuming that n is a positive integer. (Notice that $n\lfloor x\rfloor \leqslant \lfloor nx\rfloor$ for all x in this case.)

3.6 $\lfloor f(x)\rfloor = \lfloor f(\lceil x\rceil)\rfloor$.

3.7 $\lfloor n/m\rfloor + n \bmod m$.

3.8 If all boxes contain $<\lceil n/m\rceil$ objects, then $n\leqslant \bigl(\lceil n/m\rceil-1\bigr)m$, so $n/m+1\leqslant\lceil n/m\rceil$, contradicting (3.5). The other proof is similar.

3.9 We have $m/n - 1/q = (n \text{ mumble } m)/qn$. The process must terminate, because $0 \leqslant n \text{ mumble } m < m$. The denominators of the representation are strictly increasing, hence distinct, because $qn/(n \text{ mumble } m) > q$.

3.10 $\lceil x + \frac{1}{2} \rceil - \big[(2x+1)/4 \text{ is not an integer}\big]$ is the nearest integer to x, if $\{x\} \neq \frac{1}{2}$; otherwise it's the nearest even integer. (See exercise 2.) Thus the formula gives an "unbiased" way to round.

3.11 If n is an integer, $\alpha < n < \beta \iff \lfloor \alpha \rfloor < n < \lceil \beta \rceil$. The number of integers satisfying $a < n < b$ when a and b are integers is $(b - a - 1)[b > a]$. We would therefore get the wrong answer if $\alpha = \beta =$ integer.

3.12 Subtract $\lfloor n/m \rfloor$ from both sides, by (3.6), getting $\lceil (n \bmod m)/m \rceil = \lfloor (n \bmod m + m - 1)/m \rfloor$. Both sides are now equal to $[n \bmod m > 0]$, since $0 \leqslant n \bmod m < m$.

A shorter but less direct proof simply observes that the first term in (3.24) must equal the last term in (3.25).

3.13 If they form a partition, the text's formula for $N(\alpha, n)$ implies that $1/\alpha + 1/\beta = 1$, because the coefficients of n in the equation $N(\alpha, n) + N(\beta, n) = n$ must agree if the equation is to hold for large n. Hence α and β are both rational or both irrational. If both are irrational, we do get a partition, as shown in the text. If both can be written with numerator m, the value $m - 1$ occurs in neither spectrum, and m occurs in both. (However, Golomb [151] has observed that the sets $\{\lfloor n\alpha \rfloor \mid n \geqslant 1\}$ and $\{\lceil n\beta \rceil - 1 \mid n \geqslant 1\}$ always do form a partition, when $1/\alpha + 1/\beta = 1$.)

3.14 It's obvious by (3.22) if $ny = 0$, otherwise true by (3.21) and (3.6).

3.15 Plug in $\lceil mx \rceil$ for n in (3.24): $\lceil mx \rceil = \lceil x \rceil + \lceil x - \frac{1}{m} \rceil + \cdots + \lceil x - \frac{m-1}{m} \rceil$.

3.16 The formula $n \bmod 3 = 1 + \frac{1}{3}\big((\omega - 1)\omega^n - (\omega + 2)\omega^{2n}\big)$ can be verified by checking it when $0 \leqslant n < 3$.

A general formula for $n \bmod m$, when m is any positive integer, appears in exercise 7.25.

3.17 $\sum_{j,k}[0 \leqslant k < m][1 \leqslant j \leqslant x + k/m] = \sum_{j,k}[0 \leqslant k < m][1 \leqslant j \leqslant \lceil x \rceil] \times \big[k \geqslant m(j - x)\big] = \sum_{1 \leqslant j \leqslant \lceil x \rceil} \sum_k [0 \leqslant k < m] - \sum_{j = \lceil x \rceil} \sum_k \big[0 \leqslant k < m(j - x)\big] = m\lceil x \rceil - \big\lceil m(\lceil x \rceil - x) \big\rceil = -\lceil -mx \rceil = \lfloor mx \rfloor$.

3.18 We have

$$S = \sum_{0 \leqslant j < \lceil n\alpha \rceil} \sum_{k \geqslant n} \big[j\alpha^{-1} \leqslant k < (j + \nu)\alpha^{-1} \big] .$$

If $j \leqslant n\alpha - 1 \leqslant n\alpha - \nu$, there is no contribution, because $(j + \nu)\alpha^{-1} \leqslant n$. Hence $j = \lfloor n\alpha \rfloor$ is the only case that matters, and the value in that case equals $\big\lceil (\lfloor n\alpha \rfloor + \nu)\alpha^{-1} \big\rceil - n \leqslant \lceil \nu\alpha^{-1} \rceil$.

3.19 If and only if b is an integer. (If b is an integer, $\log_b x$ is a continuous, increasing function that takes integer values only at integer points. If b is not an integer, the condition fails when $x = b$.)

3.20 We have $\sum_k kx[\alpha \leqslant kx \leqslant \beta] = x \sum_k k\big[\lceil \alpha/x \rceil \leqslant k \leqslant \lfloor \beta/x \rfloor\big]$, which sums to $\frac{1}{2}x\big(\lfloor \beta/x \rfloor \lfloor \beta/x + 1 \rfloor - \lceil \alpha/x \rceil \lceil \alpha/x - 1 \rceil\big)$.

3.21 If $10^n \leqslant 2^M < 10^{n+1}$, there are exactly $n+1$ such powers of 2, because there's exactly one such k-digit power of 2 for each k. Therefore the answer is $1 + \lfloor M \log 2 \rfloor$.

Note: The number of powers of 2 with leading digit l is more difficult, when $l > 1$; it's $\sum_{0 \leqslant n \leqslant M} \big(\lfloor n \log 2 - \log l \rfloor - \lfloor n \log 2 - \log(l+1) \rfloor\big)$.

3.22 All terms are the same for n and $n-1$ except the kth, where $n = 2^{k-1}q$ and q is odd; we have $S_n = S_{n-1} + 1$ and $T_n = T_{n-1} + 2^k q$. Hence $S_n = n$ and $T_n = n(n+1)$.

3.23 $X_n = m \iff \frac{1}{2}m(m-1) < n \leqslant \frac{1}{2}m(m+1) \iff m^2 - m + \frac{1}{4} < 2n < m^2 + m + \frac{1}{4} \iff m - \frac{1}{2} < \sqrt{2n} < m + \frac{1}{2}$.

3.24 Let $\beta = \alpha/(\alpha+1)$. Then the number of times the nonnegative integer m occurs in $\mathrm{Spec}(\beta)$ is exactly one more than the number of times it occurs in $\mathrm{Spec}(\alpha)$. Why? Because $N(\beta, n) = N(\alpha, n) + n + 1$.

3.25 Continuing the development in the text, if we could find a value of m such that $K_m \leqslant m$, we could violate the stated inequality at $n+1$ when $n = 2m+1$. (Also when $n = 3m+1$ and $n = 3m+2$.) But the existence of such an $m = n' + 1$ requires that $2K_{\lfloor n'/2 \rfloor} \leqslant n'$ or $3K_{\lfloor n'/3 \rfloor} \leqslant n'$, i.e., that

$$K_{\lfloor n'/2 \rfloor} \leqslant \lfloor n'/2 \rfloor \qquad \text{or} \qquad K_{\lfloor n'/3 \rfloor} \leqslant \lfloor n'/3 \rfloor .$$

Aha. This goes down further and further, implying that $K_0 \leqslant 0$; but $K_0 = 1$.

What we really want to prove is that K_n is strictly *greater* than n, for all $n > 0$. In fact, it's easy to prove this by induction, although it's a stronger result than the one we couldn't prove!

(This exercise teaches an important lesson. It's more an exercise about the nature of induction than about properties of the floor function.)

"In trying to devise a proof by mathematical induction, you may fail for two opposite reasons. You may fail because you try to prove too much: Your $P(n)$ is too heavy a burden. Yet you may also fail because you try to prove too little: Your $P(n)$ is too weak a support. In general, you have to balance the statement of your theorem so that the support is just enough for the burden."
—G. Pólya [297]

3.26 Induction, using the stronger hypothesis

$$D_n^{(q)} \leqslant (q-1)\left(\left(\frac{q}{q-1}\right)^{n+1} - 1\right), \qquad \text{for } n \geqslant 0.$$

3.27 If $D_n^{(3)} = 2^m b - a$, where a is 0 or 1, then $D_{n+m}^{(3)} = 3^m b - a$.

3.28 The key observation is that $a_n = m^2$ implies $a_{n+2k+1} = (m+k)^2 + m - k$ and $a_{n+2k+2} = (m+k)^2 + 2m$, for $0 \leqslant k \leqslant m$; hence $a_{n+2m+1} = (2m)^2$. The solution can be written in a nice form discovered by Carl Witty:

$$a_{n-1} \;=\; 2^l + \left\lfloor \left(\frac{n-l}{2}\right)^2 \right\rfloor, \qquad \text{when } 2^l + l \leqslant n < 2^{l+1} + l + 1.$$

3.29 $D(\alpha', \lfloor \alpha n \rfloor)$ is at most the maximum of the right-hand side of

$$s(\alpha', \lfloor n\alpha \rfloor, \nu') \;=\; -s(\alpha, n, \nu) + S - \epsilon - \{0 \text{ or } 1\} - \nu' + \{0 \text{ or } 1\}.$$

3.30 $X_n = \alpha^{2^n} + \alpha^{-2^n}$, by induction; and X_n is an integer.

This logic is seriously floored.

3.31 Here's an "elegant," "impressive" proof that gives no clue about how it was discovered:

$$\begin{aligned} \lfloor x \rfloor + \lfloor y \rfloor + \lfloor x+y \rfloor \;&=\; \lfloor x + \lfloor y \rfloor \rfloor + \lfloor x+y \rfloor \\ &\leqslant\; \lfloor x + \tfrac{1}{2}\lfloor 2y \rfloor \rfloor + \lfloor x + \tfrac{1}{2}\lfloor 2y \rfloor + \tfrac{1}{2} \rfloor \\ &=\; \lfloor 2x + \lfloor 2y \rfloor \rfloor \;=\; \lfloor 2x \rfloor + \lfloor 2y \rfloor . \end{aligned}$$

But there's also a simple, graphical proof based on the observation that we need to consider only the case $0 \leqslant x, y < 1$. Then the functions look like this in the plane:

A slightly stronger result is possible, namely

$$\lceil x \rceil + \lfloor y \rfloor + \lfloor x+y \rfloor \;\leqslant\; \lceil 2x \rceil + \lfloor 2y \rfloor ;$$

but this is stronger only when $\{x\} = \frac{1}{2}$. If we replace (x, y) by $(-x, x+y)$ in this identity and apply the reflective law (3.4), we get

$$\lfloor y \rfloor + \lfloor x+y \rfloor + \lfloor 2x \rfloor \;\leqslant\; \lfloor x \rfloor + \lfloor 2x+2y \rfloor .$$

3.32 Let $f(x)$ be the sum in question. Since $f(x) = f(-x)$, we may assume that $x \geqslant 0$. The terms are bounded by 2^k as $k \to -\infty$ and by $x^2/2^k$ as $k \to +\infty$, so the sum exists for all real x.

We have $f(2x) = 2\sum_k 2^{k-1}\|x/2^{k-1}\|^2 = 2f(x)$. Let $f(x) = l(x) + r(x)$ where $l(x)$ is the sum for $k \leqslant 0$ and $r(x)$ is the sum for $k > 0$. Then $l(x+1) = l(x)$, and $l(x) \leqslant 1/2$ for all x. When $0 \leqslant x < 1$, we have $r(x) = x^2/2 + x^2/4 + \cdots = x^2$, and $r(x+1) = (x-1)^2/2 + (x+1)^2/4 + (x+1)^2/8 + \cdots = x^2 + 1$. Hence $f(x+1) = f(x) + 1$, when $0 \leqslant x < 1$.

We can now prove by induction that $f(x+n) = f(x)+n$ for all integers $n \geqslant 0$, when $0 \leqslant x < 1$. In particular, $f(n) = n$. Therefore in general, $f(x) = 2^{-m}f(2^m x) = 2^{-m}\lfloor 2^m x\rfloor + 2^{-m}f\bigl(\{2^m x\}\bigr)$. But $f\bigl(\{2^m x\}\bigr) = l\bigl(\{2^m x\}\bigr) + r\bigl(\{2^m x\}\bigr) \leqslant \frac{1}{2} + 1$; so $\bigl|f(x) - x\bigr| \leqslant \bigl|2^{-m}\lfloor 2^m x\rfloor - x\bigr| + 2^{-m}\cdot\frac{3}{2} \leqslant 2^{-m}\cdot\frac{5}{2}$ for all integers m.

The inescapable conclusion is that $f(x) = |x|$ for all real x.

3.33 Let $r = n-\frac{1}{2}$ be the radius of the circle. (a) There are $2n-1$ horizontal lines and $2n-1$ vertical lines between cells of the board, and the circle crosses each of these lines twice. Since r^2 is not an integer, the Pythagorean theorem tells us that the circle doesn't pass through the corner of any cell. Hence the circle passes through as many cells as there are crossing points, namely $8n-4 = 8r$. (The same formula gives the number of cells at the edge of the board.) (b) $f(n,k) = 4\lfloor\sqrt{r^2-k^2}\rfloor$.

It follows from (a) and (b) that

$$\tfrac{1}{4}\pi r^2 - 2r \;\leqslant\; \sum_{0<k<r} \lfloor\sqrt{r^2-k^2}\rfloor \;\leqslant\; \tfrac{1}{4}\pi r^2\,, \qquad r = n-\tfrac{1}{2}.$$

The task of obtaining more precise estimates of this sum is a famous problem in number theory, investigated by Gauss and many others; see Dickson [78, volume 2, chapter 6].

3.34 (a) Let $m = \lceil \lg n\rceil$. We can add $2^m - n$ terms to simplify the calculations at the boundary:

$$\begin{aligned} f(n) + (2^m - n)m \;=\; \sum_{k=1}^{2^m} \lceil \lg k\rceil \;&=\; \sum_{j,k} j[j = \lceil \lg k\rceil][1 \leqslant k \leqslant 2^m] \\ &=\; \sum_{j,k} j[2^{j-1} < k \leqslant 2^j][1 \leqslant j \leqslant m] \\ &=\; \sum_{j=1}^{m} j\,2^{j-1} \;=\; 2^m(m-1)+1\,. \end{aligned}$$

Consequently $f(n) = nm - 2^m + 1$.

(b) We have $\lceil n/2\rceil = \lfloor (n+1)/2\rfloor$, and it follows that the solution to the general recurrence $g(n) = a(n) + g\bigl(\lceil n/2\rceil\bigr) + g\bigl(\lfloor n/2\rfloor\bigr)$ must satisfy $\Delta g(n) = \Delta a(n) + \Delta g\bigl(\lfloor n/2\rfloor\bigr)$. In particular, when $a(n) = n-1$, $\Delta f(n) = 1 + \Delta f\bigl(\lfloor n/2\rfloor\bigr)$ is satisfied by the number of bits in the binary representation of n, namely $\lceil \lg(n+1)\rceil$. Now convert from Δ to Σ.

A more direct solution can be based on the identities $\lceil \lg 2j\rceil = \lceil \lg j\rceil + 1$ and $\lceil \lg(2j-1)\rceil = \lceil \lg j\rceil + [j>1]$, for $j \geqslant 1$.

3.35 $(n+1)^2 n!\,e = A_n + (n+1)^2 + (n+1) + B_n$, where

$$A_n = \frac{(n+1)^2 n!}{0!} + \frac{(n+1)^2 n!}{1!} + \cdots + \frac{(n+1)^2 n!}{(n-1)!}$$

is a multiple of n and

$$\begin{aligned} B_n &= \frac{(n+1)^2 n!}{(n+2)!} + \frac{(n+1)^2 n!}{(n+3)!} + \cdots \\ &= \frac{n+1}{n+2}\left(1 + \frac{1}{n+3} + \frac{1}{(n+3)(n+4)} + \cdots\right) \\ &< \frac{n+1}{n+2}\left(1 + \frac{1}{n+3} + \frac{1}{(n+3)(n+3)} + \cdots\right) \\ &= \frac{(n+1)(n+3)}{(n+2)^2} \end{aligned}$$

is less than 1. Hence the answer is 2 mod n.

3.36 The sum is

$$\begin{aligned} &\sum_{k,l,m} 2^{-l} 4^{-m} \big[m = \lfloor \lg l \rfloor\big]\big[l = \lfloor \lg k \rfloor\big][1 < k < 2^{2^n}] \\ &\quad = \sum_{k,l,m} 2^{-l} 4^{-m} [2^m \leqslant l < 2^{m+1}][2^l \leqslant k < 2^{l+1}][0 \leqslant m < n] \\ &\quad = \sum_{l,m} 4^{-m} [2^m \leqslant l < 2^{m+1}][0 \leqslant m < n] \\ &\quad = \sum_{m} 2^{-m} [0 \leqslant m < n] \;=\; 2(1 - 2^{-n})\,. \end{aligned}$$

3.37 First consider the case $m < n$, which breaks into subcases based on whether $m < \frac{1}{2}n$; then show that both sides change in the same way when m is increased by n.

This is really only a level 4 problem, in spite of the way it's stated.

3.38 At most one x_k can be noninteger. Discard all integer x_k, and suppose that n are left. When $\{x\} \neq 0$, the average of $\{mx\}$ as $m \to \infty$ lies between $\frac{1}{4}$ and $\frac{1}{2}$; hence $\{mx_1\} + \cdots + \{mx_n\} - \{mx_1 + \cdots + mx_n\}$ cannot have average value zero when $n > 1$.

But the argument just given relies on a difficult theorem about uniform distribution. An elementary proof is possible, sketched here for $n = 2$: Let P_m be the point $\big(\{mx\},\{my\}\big)$. Divide the unit square $0 \leqslant x, y < 1$ into triangular regions A and B according as $x + y < 1$ or $x + y \geqslant 1$. We want to show that $P_m \in B$ for some m, if $\{x\}$ and $\{y\}$ are nonzero. If $P_1 \in B$, we're done. Otherwise there is a disk D of radius $\epsilon > 0$ centered at P_1 such that

$D \subseteq A$. By Dirichlet's box principle, the sequence $P_1, \ldots, P_N$ must contain two points with $|P_k - P_j| < \epsilon$ and $k > j$, if N is large enough.

It follows that P_{k-j-1} is within ϵ of $(1,1) - P_1$; hence $P_{k-j-1} \in B$.

3.39 Replace j by $b - j$ and add the term $j = 0$ to the sum, so that exercise 15 can be used for the sum on j. The result,

$$\lceil x/b^k \rceil - \lceil x/b^{k+1} \rceil + b - 1\,,$$

telescopes when summed on k.

3.40 Let $\lfloor 2\sqrt{n} \rfloor = 4k + r$ where $-2 \leqslant r < 2$, and let $m = \lfloor \sqrt{n} \rfloor$. Then the following relationships can be proved by induction:

segment	r	m	x	y	if and only if
W_k	-2	$2k-1$	$m(m+1) - n - k$	k	$(2k-1)(2k-1) \leqslant n \leqslant (2k-1)(2k)$
S_k	-1	$2k-1$	$-k$	$m(m+1) - n + k$	$(2k-1)(2k) < n < (2k)(2k)$
E_k	0	$2k$	$n - m(m+1) + k$	$-k$	$(2k)(2k) \leqslant n \leqslant (2k)(2k+1)$
N_k	1	$2k$	k	$n - m(m+1) - k$	$(2k)(2k+1) < n < (2k+1)(2k+1)$

Thus, when $k \geqslant 1$, W_k is a segment of length $2k$ where the path travels west and $y(n) = k$; S_k is a segment of length $2k - 2$ where the path travels south and $x(n) = -k$; etc. (a) The desired formula is therefore

$$y(n) \;=\; (-1)^m \Big(\big(n - m(m+1)\big) \cdot \big[\lfloor 2\sqrt{n} \rfloor \text{ is odd}\big] - \lceil \tfrac{1}{2} m \rceil \Big)\,.$$

(b) On all segments, $k = \max\big(|x(n)|, |y(n)|\big)$. On segments W_k and S_k we have $x < y$ and $n + x + y = m(m+1) = (2k)^2 - 2k$; on segments E_k and N_k we have $x \geqslant y$ and $n - x - y = m(m+1) = (2k)^2 + 2k$. Hence the sign is $(-1)^{(x(n) < y(n))}$.

3.41 Since $1/\phi + 1/\phi^2 = 1$, the stated sequences do partition the positive integers. Since the condition $g(n) = f\big(f(n)\big) + 1$ determines f and g uniquely, we need only show that $\big\lfloor \lfloor n\phi \rfloor \phi \big\rfloor + 1 = \lfloor n\phi^2 \rfloor$ for all $n > 0$. This follows from exercise 3, with $\alpha = \phi$ and $n = 1$.

3.42 No; an argument like the analysis of the two-spectrum case in the text and in exercise 13 shows that a tripartition occurs if and only if $1/\alpha + 1/\beta + 1/\gamma = 1$ and

$$\left\{\frac{n+1}{\alpha}\right\} + \left\{\frac{n+1}{\beta}\right\} + \left\{\frac{n+1}{\gamma}\right\} = 1,$$

for all $n > 0$. But the average value of $\{(n+1)/\alpha\}$ is $1/2$ if α is irrational, by the theorem on uniform distribution. The parameters can't all be rational, and if $\gamma = m/n$ the average is $3/2 - 1/(2n)$. Hence γ must be an integer, but this doesn't work either. (There's also a proof of impossibility that uses only simple principles, without the theorem on uniform distribution; see [155].)

3.43 One step of unfolding the recurrence for K_n gives the minimum of the four numbers $1 + a + a \cdot b \cdot K_{\lfloor (n-1-a)/(a \cdot b) \rfloor}$, where a and b are each 2 or 3. (This simplification involves an application of (3.11) to remove floors within floors, together with the identity $x + \min(y, z) = \min(x + y, x + z)$. We must omit terms with negative subscripts; i.e., with $n - 1 - a < 0$.)

Continuing along such lines now leads to the following interpretation: K_n is the least number $> n$ in the multiset S of all numbers of the form

$$1 + a_1 + a_1 a_2 + a_1 a_2 a_3 + \cdots + a_1 a_2 a_3 \ldots a_m,$$

where $m \geqslant 0$ and each a_k is 2 or 3. Thus,

$$S = \{1, 3, 4, 7, 9, 10, 13, 15, 19, 21, 22, 27, 28, 31, 31, \ldots\};$$

the number 31 is in S "twice" because it has two representations $1 + 2 + 4 + 8 + 16 = 1 + 3 + 9 + 18$. (Incidentally, Michael Fredman [134] has shown that $\lim_{n\to\infty} K_n/n = 1$, i.e., that S has no enormous gaps.)

3.44 Let $d_n^{(q)} = D_{n-1}^{(q)} \text{mumble}(q-1)$, so that $D_n^{(q)} = (qD_{n-1}^{(q)} + d_n^{(q)})/(q-1)$ and $a_n^{(q)} = \lceil D_{n-1}^{(q)}/(q-1) \rceil$. Now $D_{k-1}^{(q)} \leqslant (q-1)n \iff a_k^{(q)} \leqslant n$, and the results follow. (This is the solution found by Euler [116], who determined the a's and d's sequentially without realizing that a single sequence $D_n^{(q)}$ would suffice.)

Too easy. **3.45** Let $\alpha > 1$ satisfy $\alpha + 1/\alpha = 2m$. Then we find $2Y_n = \alpha^{2^n} + \alpha^{-2^n}$, and it follows that $Y_n = \lceil \alpha^{2^n}/2 \rceil$.

3.46 The hint follows from (3.9), since $2n(n+1) = \lfloor 2(n+\frac{1}{2})^2 \rfloor$. Let $n + \theta = (\sqrt{2}^l + \sqrt{2}^{l-1})m$ and $n' + \theta' = (\sqrt{2}^{l+1} + \sqrt{2}^l)m$, where $0 \leqslant \theta, \theta' < 1$. Then $\theta' = 2\theta \bmod 1 = 2\theta - d$, where d is 0 or 1. We want to prove that $n' = \lfloor \sqrt{2}(n + \frac{1}{2}) \rfloor$; this equality holds if and only if

$$0 \leqslant \theta'(2 - \sqrt{2}) + \sqrt{2}(1 - d) < 2.$$

To solve the recurrence, note that $\mathrm{Spec}(1+1/\sqrt{2}\,)$ and $\mathrm{Spec}(1+\sqrt{2}\,)$ partition the positive integers; hence any positive integer a can be written uniquely in the form $a = \lfloor(\sqrt{2}^{\,l} + \sqrt{2}^{\,l-1})m\rfloor$, where l and m are integers with m odd and $l \geqslant 0$. It follows that $L_n = \lfloor(\sqrt{2}^{\,l+n} + \sqrt{2}^{\,l+n-1})m\rfloor$.

3.47 (a) $c = -\frac{1}{2}$. (b) c is an integer. (c) $c = 0$. (d) c is arbitrary. See the answer to exercise 1.2.4–40 in [207] for more general results.

3.48 Let $x^{:0} = 1$ and $x^{:(k+1)} = x\lfloor x^{:k}\rfloor$; also let $a_k = \{x^{:k}\}$ and $b_k = \lfloor x^{:k}\rfloor$, so that the stated identity reads $x^3 = 3x^{:3} + 3a_1a_2 + a_1^3 - 3b_1b_2 + b_1^3$. Since $a_k + b_k = x^{:k} = xb_{k-1}$ for $k \geqslant 0$, we have $(1-xz)(1+b_1z+b_2z^2+\cdots) = 1 - a_1z - a_2z^2 - \cdots$; thus

$$\frac{1}{1-xz} = \frac{1+b_1z+b_2z^2+\cdots}{1-a_1z-a_2z^2-\cdots}.$$

Take the logarithm of both sides, to separate the a's from the b's. Then differentiate with respect to z, obtaining

$$\frac{x}{1-xz} = \frac{a_1+2a_2z+3a_3z^2+\cdots}{1-a_1z-a_2z^2-\cdots} + \frac{b_1+2b_2z+3b_3z^2+\cdots}{1+b_1z+b_2z^2+\cdots}.$$

The coefficient of z^{n-1} on the left is x^n; on the right it is a formula that matches the given identity when $n = 3$.

Similar identities for the more general product $x_0x_1\ldots x_{n-1}$ can also be derived [170].

3.49 (Solution by Heinrich Rolletschek.) We can replace (α,β) by $(\{\beta\}, \alpha+\lfloor\beta\rfloor)$ without changing $\lfloor n\alpha\rfloor + \lfloor n\beta\rfloor$. Hence the condition $\alpha = \{\beta\}$ is necessary. It is also sufficient: Let $m = \lfloor\beta\rfloor$ be the least element of the given multiset, and let S be the multiset obtained from the given one by subtracting mn from the nth smallest element, for all n. If $\alpha = \{\beta\}$, consecutive elements of S differ by either 0 or 2, hence the multiset $\frac{1}{2}S = \mathrm{Spec}(\alpha)$ determines α.

A more interesting (still unsolved) problem: Restrict both α and β to be < 1, and ask when the given multiset determines the unordered pair $\{\alpha,\beta\}$.

3.50 According to unpublished notes of William A. Veech, it is sufficient to have $\alpha\beta$, β, and 1 linearly independent over the rationals.

3.51 H. S. Wilf observes that the functional equation $f(x^2-1) = f(x)^2$ would determine $f(x)$ for all $x \geqslant \phi$ if we knew $f(x)$ on any interval $(\phi\,..\,\phi+\epsilon)$.

3.52 There are infinitely many ways to partition the positive integers into three or more generalized spectra with *irrational* α_k; for example,

$$\mathrm{Spec}(2\alpha;0) \cup \mathrm{Spec}(4\alpha;-\alpha) \cup \mathrm{Spec}(4\alpha;-3\alpha) \cup \mathrm{Spec}(\beta;0)$$

works. But there's a precise sense in which all such partitions arise by "expanding" a basic one, $\mathrm{Spec}(\alpha) \cup \mathrm{Spec}(\beta)$; see [158]. The only known rational

examples, e.g.,

$$\mathrm{Spec}(7;-3)\cup\mathrm{Spec}(\tfrac{7}{2};-1)\cup\mathrm{Spec}(\tfrac{7}{4};0)\,,$$

are based on parameters like those in the stated conjecture, which is due to A. S. Fraenkel [128].

3.53 Partial results are discussed in [95, pages 30–31]. The greedy algorithm probably does not terminate.

4.1 1, 2, 4, 6, 16, 12.

4.2 Note that $m_p + n_p = \min(m_p, n_p) + \max(m_p, n_p)$. The recurrence $\mathrm{lcm}(m,n) = \bigl(n/(n \bmod m)\bigr)\,\mathrm{lcm}(n \bmod m, m)$ is valid but not really advisable for computing lcm's; the best way known to compute $\mathrm{lcm}(m,n)$ is to compute $\gcd(m,n)$ first and then to divide mn by the gcd.

4.3 This holds if x is an integer, but $\pi(x)$ is defined for all real x. The correct formula,

$$\pi(x) - \pi(x-1) \;=\; \bigl[\lfloor x\rfloor \text{ is prime}\bigr]\,,$$

is easy to verify.

4.4 Between $\frac{1}{0}$ and $\frac{0}{-1}$ we'd have a left-right reflected Stern–Brocot tree with all denominators negated, etc. So the result is *all* fractions m/n with $m \perp n$. The condition $m'n - mn' = 1$ still holds throughout the construction. (This is called the *Stern–Brocot wreath*, because we can conveniently regard the final $\frac{0}{1}$ as identical to the first $\frac{0}{1}$, thereby joining the trees in a cycle at the top. The Stern–Brocot wreath has interesting applications to computer graphics because it represents all rational directions in the plane.)

4.5 $L^k = \left(\begin{smallmatrix}1 & k\\ 0 & 1\end{smallmatrix}\right)$ and $R^k = \left(\begin{smallmatrix}1 & 0\\ k & 1\end{smallmatrix}\right)$; this holds even when $k < 0$. (We will find a general formula for any product of L's and R's in Chapter 6.)

After all, 'mod y' sort of means "pretend y is zero." So if it already is, there's nothing to pretend.

4.6 $a = b$. (Chapter 3 defined $x \bmod 0 = x$, primarily so that this would be true.)

4.7 We need $m \bmod 10 = 0$, $m \bmod 9 = k$, and $m \bmod 8 = 1$. But m can't be both even and odd.

4.8 We want $10x + 6y \equiv 10x + y \pmod{15}$; hence $5y \equiv 0 \pmod{15}$; hence $y \equiv 0 \pmod 3$. We must have $y = 0$ or 3, and $x = 0$ or 1.

4.9 $3^{2k+1} \bmod 4 = 3$, so $(3^{2k+1} - 1)/2$ is odd. The stated number is divisible by $(3^7 - 1)/2$ and $(3^{11} - 1)/2$ (and by other numbers).

4.10 $999(1 - \frac{1}{3})(1 - \frac{1}{37}) = 648$.

4.11 $\sigma(0) = 1$; $\sigma(1) = -1$; $\sigma(n) = 0$ for $n > 1$. (Generalized Möbius functions defined on arbitrary partially ordered structures have interesting and important properties, first explored by Weisner [366] and developed by many other people, notably Gian-Carlo Rota [313].)

4.12 $\sum_{d\backslash m}\sum_{k\backslash d}\mu(d/k)\,g(k) = \sum_{k\backslash m}\sum_{d\backslash(m/k)}\mu(d)\,g(k) = \sum_{k\backslash m} g(k)\times [m/k=1] = g(m)$, by (4.7) and (4.9).

4.13 (a) $n_p \leqslant 1$ for all p; (b) $\mu(n) \neq 0$.

4.14 True when $k > 0$. Use (4.12), (4.14), and (4.15).

4.15 No. For example, $e_n \bmod 5 = [2 \text{ or } 3]$; $e_n \bmod 11 = [2, 3, 7, \text{or } 10]$.

4.16 $1/e_1 + 1/e_2 + \cdots + 1/e_n = 1 - 1\big/\big(e_n(e_n - 1)\big) = 1 - 1/(e_{n+1} - 1)$.

4.17 We have $f_n \bmod f_m = 2$; hence $\gcd(f_n, f_m) = \gcd(2, f_m) = 1$. (Incidentally, the relation $f_n = f_0 f_1 \ldots f_{n-1} + 2$ is very similar to the recurrence that defines the Euclid numbers e_n.)

4.18 If $n = qm$ and q is odd, $2^n+1 = (2^m+1)(2^{n-m}-2^{n-2m}+\cdots-2^m+1)$.

4.19 The first sum is $\pi(n)$, since the summand is $[k+1 \text{ is prime}]$. The inner sum in the second is $\sum_{1\leqslant k<m}[k\backslash m]$, so it is greater than 1 if and only if m is composite; again we get $\pi(n)$. Finally $\big\lceil\{m/n\}\big\rceil = [n\backslash\!\!\backslash m]$, so the third sum is an application of Wilson's theorem. To evaluate $\pi(n)$ by any of these formulas is, of course, sheer lunacy.

4.20 Let $p_1 = 2$ and let p_n be the smallest prime greater than $2^{p_{n-1}}$. Then $2^{p_{n-1}} < p_n < 2^{p_{n-1}+1}$, and it follows that we can take $b = \lim_{n\to\infty} \lg^{(n)} p_n$ where $\lg^{(n)}$ is the function lg iterated n times. The stated numerical value comes from $p_2 = 5$, $p_3 = 37$. It turns out that $p_4 = 2^{37} + 9$, and this gives the more precise value

$$b \approx 1.2516475977905$$

(but no clue about p_5).

4.21 By Bertrand's postulate, $P_n < 10^n$. Let

$$K = \sum_{k\geqslant 1} 10^{-k^2} P_k = .200300005\ldots .$$

Then $10^{n^2} K \equiv P_n + \text{fraction} \pmod{10^{2n-1}}$.

4.22 $(b^{mn} - 1)/(b - 1) = \big((b^m - 1)/(b - 1)\big)(b^{mn-m} + \cdots + 1)$. [The only prime numbers of the form $(10^p - 1)/9$ for $p < 20000$ occur when $p = 2$, 19, 23, 317, 1031.] Numbers of this form are called "repunits."

4.23 $\rho(2k+1) = 0$; $\rho(2k) = \rho(k) + 1$, for $k \geqslant 1$. By induction we can show that $\rho(n) = \rho(n-2^m)$, if $n > 2^m$ and $m > \rho(n)$. The kth Hanoi move is disk $\rho(k)$, if we number the disks $0, 1, \ldots, n-1$. This is clear if k is a power of 2. And if $2^m < k < 2^{m+1}$, we have $\rho(k) < m$; moves k and $k - 2^m$ correspond in the sequence that transfers $m+1$ disks in $T_m + 1 + T_m$ steps.

4.24 The digit that contributes dp^m to n contributes $dp^{m-1} + \cdots + d = d(p^m - 1)/(p-1)$ to $\epsilon_p(n!)$, hence $\epsilon_p(n!) = \bigl(n - \nu_p(n)\bigr)/(p-1)$.

4.25 $m \backslash\backslash n \iff m_p = 0$ or $m_p = n_p$, for all p. It follows that (a) is true. But (b) fails, in our favorite example $m = 12$, $n = 18$. (This is a common fallacy.)

4.26 Yes, since $\mathcal{G}_N$ defines a subtree of the Stern–Brocot tree.

4.27 Extend the shorter string with M's (since M lies alphabetically between L and R) until both strings are the same length, then use dictionary order. For example, the topmost levels of the tree are LL < LM < LR < MM < RL < RM < RR. (Another solution is to append the infinite string RL^∞ to both inputs, and to keep comparing until finding L < R.)

4.28 We need to use only the first part of the representation:

$$\begin{array}{cccccccccccccccc}
 & \mathrm{R} & \mathrm{R} & \mathrm{R} & \mathrm{L} & \mathrm{L} & \mathrm{L} & \mathrm{L} & \mathrm{L} & \mathrm{L} & \mathrm{L} & \mathrm{R} & \mathrm{R} & \mathrm{R} & \mathrm{R} & \mathrm{R} & \mathrm{R} \\
\frac{1}{1}, & \frac{2}{1}, & \frac{3}{1}, & \frac{4}{1}, & \frac{7}{2}, & \frac{10}{3}, & \frac{13}{4}, & \frac{16}{5}, & \frac{19}{6}, & \frac{22}{7}, & \frac{25}{8}, & \frac{47}{15}, & \frac{69}{22}, & \frac{91}{29}, & \frac{113}{36}, & \frac{135}{43}, & \ldots .
\end{array}$$

The fraction $\frac{4}{1}$ appears because it's a better upper bound than $\frac{1}{0}$, not because it's closer than $\frac{3}{1}$. Similarly, $\frac{25}{8}$ is a better lower bound than $\frac{3}{1}$. The simplest upper bounds and the simplest lower bounds all appear, but the next really good approximation doesn't occur until just before the string of R's switches back to L.

4.29 $1/\alpha$. To get $1-x$ from x in binary notation, we interchange 0 and 1; to get $1/\alpha$ from α in Stern–Brocot notation, we interchange L and R. (The finite cases must also be considered, but they must work since the correspondence is order preserving.)

4.30 The m integers $x \in [A \mathinner{..} A+m)$ are different mod m; so their residues $(x \bmod m_1, \ldots, x \bmod m_r)$ run through all $m_1 \ldots m_r = m$ possible values, one of which must be equal to $(a_1 \bmod m_1, \ldots, a_r \bmod m_r)$ by the pigeonhole principle.

4.31 A number in radix b notation is divisible by d if and only if the sum of its digits is divisible by d, whenever $b \equiv 1 \pmod{d}$. This follows because $(a_m \ldots a_0)_b = a_m b^m + \cdots + a_0 b^0 \equiv a_m + \cdots + a_0$.

4.32 The $\varphi(m)$ numbers $\{\, kn \bmod m \mid k \perp m \text{ and } 0 \leqslant k < m \,\}$ are the numbers $\{\, k \mid k \perp m \text{ and } 0 \leqslant k < m \,\}$ in some order. Multiply them together and divide by $\prod_{0 \leqslant k < m,\, k \perp m} k$.

4.33 Obviously $h(1) = 1$. If $m \perp n$ then $h(mn) = \sum_{d\backslash mn} f(d)\, g(mn/d) = \sum_{c\backslash m, d\backslash n} f(cd)\, g\big((m/c)(n/d)\big) = \sum_{c\backslash m} \sum_{d\backslash n} f(c)\, g(m/c)\, f(d)\, g(n/d)$; this is $h(m)\, h(n)$, since $c \perp d$ for every term in the sum.

4.34 $g(m) = \sum_{d\backslash m} f(d) = \sum_{d\backslash m} f(m/d) = \sum_{d\geqslant 1} f(m/d)$ if $f(x)$ is zero when x is not an integer.

4.35 The base cases are

$$I(0,n) \;=\; 0\,; \qquad I(m,0) \;=\; 1\,.$$

When $m, n > 0$, there are two rules, where the first is trivial if $m > n$ and the second is trivial if $m < n$:

$$\begin{aligned} I(m,n) \;&=\; I(m, n \bmod m) - \lfloor n/m \rfloor I(n \bmod m, m)\,; \\ I(m,n) \;&=\; I(m \bmod n, n)\,. \end{aligned}$$

4.36 A factorization of any of the given quantities into nonunits must have $m^2 - 10n^2 = \pm 2$ or ± 3, but this is impossible mod 10.

4.37 Let $a_n = 2^{-n} \ln(e_n - \frac{1}{2})$ and $b_n = 2^{-n} \ln(e_n + \frac{1}{2})$. Then

$$e_n \;=\; \lfloor E^{2^n} + \tfrac{1}{2} \rfloor \quad\Longleftrightarrow\quad a_n \;\leqslant\; \ln E \;<\; b_n\,.$$

And $a_{n-1} < a_n < b_n < b_{n-1}$, so we can take $E = \lim_{n\to\infty} e^{a_n}$. In fact, it turns out that

$$E^2 \;=\; \frac{3}{2} \prod_{n\geqslant 1} \left(1 + \frac{1}{(2e_n - 1)^2}\right)^{1/2^n}\,,$$

a product that converges rapidly to $(1.26408473530530111\ldots)^2$. But these observations don't tell us what e_n is, unless we can find another expression for E that doesn't depend on Euclid numbers.

4.38 Let $r = n \bmod m$. Then $a^n - b^n = (a^m - b^m)(a^{n-m}b^0 + a^{n-2m}b^m + \cdots + a^r b^{n-m-r}) + b^{m\lfloor n/m \rfloor}(a^r - b^r)$.

4.39 If $a_1 \ldots a_t$ and $b_1 \ldots b_u$ are perfect squares, so is

$$a_1 \ldots a_t b_1 \ldots b_u / c_1^2 \ldots c_v^2\,,$$

where $\{a_1, \ldots, a_t\} \cap \{b_1, \ldots, b_u\} = \{c_1, \ldots, c_v\}$. (It can be shown, in fact, that the sequence $\langle S(1), S(2), S(3), \ldots, \rangle$ contains every nonprime positive integer exactly once.)

4.40 Let $f(n) = \prod_{1\leqslant k\leqslant n,\, p\nmid k} k = n!/p^{\lfloor n/p\rfloor}\lfloor n/p\rfloor!$ and $g(n) = n!/p^{\epsilon_p(n!)}$. Then

$$g(n) \;=\; f(n)f\bigl(\lfloor n/p\rfloor\bigr)f\bigl(\lfloor n/p^2\rfloor\bigr)\ldots \;=\; f(n)g\bigl(\lfloor n/p\rfloor\bigr)\,.$$

Also $f(n) \equiv a_0!(p-1)!^{\lfloor n/p\rfloor} \equiv a_0!(-1)^{\lfloor n/p\rfloor} \pmod p$, and $\epsilon_p(n!) = \lfloor n/p\rfloor + \epsilon_p\bigl(\lfloor n/p\rfloor!\bigr)$. These recurrences make it easy to prove the result by induction. (Several other solutions are possible.)

4.41 (a) If $n^2 \equiv -1 \pmod p$ then $(n^2)^{(p-1)/2} \equiv -1$; but Fermat says it's $+1$. (b) Let $n = \bigl((p-1)/2\bigr)!$; we have $n \equiv (-1)^{(p-1)/2}\prod_{1\leqslant k<p/2}(p-k) = (p-1)!/n$, hence $n^2 \equiv (p-1)!$.

4.42 First we observe that $k \perp l \iff k \perp l + ak$ for any integer a, since $\gcd(k,l) = \gcd(k, l+ak)$ by Euclid's algorithm. Now

$$\begin{aligned} m \perp n \quad\text{and}\quad n' \perp n \quad &\iff \quad mn' \perp n \\ &\iff \quad mn' + nm' \perp n\,. \end{aligned}$$

Similarly

$$m' \perp n' \quad\text{and}\quad n \perp n' \quad\iff\quad mn' + nm' \perp n'.$$

Hence

$$m \perp n \quad\text{and}\quad m' \perp n' \quad\text{and}\quad n \perp n' \quad\iff\quad mn'+nm' \perp nn'.$$

4.43 We want to multiply by $L^{-1}R$, then by $R^{-1}L^{-1}RL$, then $L^{-1}R$, then $R^{-2}L^{-1}RL^2$, etc.; the nth multiplier is $R^{-\rho(n)}L^{-1}RL^{\rho(n)}$, since we must cancel $\rho(n)$ R's. And $R^{-m}L^{-1}RL^m = \left(\begin{smallmatrix}0 & -1\\ 1 & 2m+1\end{smallmatrix}\right)$.

4.44 We can find the simplest rational number that lies in

$$[0.3155\,..\,0.3165) \;=\; \bigl[\tfrac{631}{2000}\,..\,\tfrac{633}{2000}\bigr)$$

John .316

— banner displayed during the 1993 World Series, when John Kruk came to bat.

by looking at the Stern–Brocot representations of $\frac{631}{2000}$ and $\frac{633}{2000}$ and stopping just before the former has L where the latter has R:

$$\begin{aligned} &(m_1, n_1, m_2, n_2) \;:=\; (631, 2000, 633, 2000);\\ &\textbf{while}\;\; m_1 > n_1 \;\;\textbf{or}\;\; m_2 < n_2 \;\;\textbf{do}\\ &\quad\textbf{if}\; m_2 \;<\; n_2 \;\textbf{then}\; \bigl(\text{output}(L);\; (n_1,n_2) \;:=\; (n_1,n_2) - (m_1,m_2)\bigr)\\ &\qquad\qquad\textbf{else}\;\; \bigl(\text{output}(R);\; (m_1,m_2) \;:=\; (m_1,m_2) - (n_1,n_2)\bigr)\,. \end{aligned}$$

The output is LLLRRRRR $= \frac{6}{19} \approx .3158$. Incidentally, an average of .334 implies at least 287 at bats.

4.45 $x^2 \equiv x \pmod{10^n} \iff x(x-1) \equiv 0 \pmod{2^n}$ and $x(x-1) \equiv 0 \pmod{5^n} \iff x \bmod 2^n = [0 \text{ or } 1]$ and $x \bmod 5^n = [0 \text{ or } 1]$. (The last step is justified because $x(x-1) \bmod 5 = 0$ implies that either x or $x-1$ is a multiple of 5, in which case the other factor is relatively prime to 5^n and can be divided from the congruence.)

So there are at most four solutions, of which two ($x = 0$ and $x = 1$) don't qualify for the title "n-digit number" unless $n = 1$. The other two solutions have the forms x and $10^n + 1 - x$, and at least one of these numbers is $\geqslant 10^{n-1}$. When $n = 4$ the other solution, $10001 - 9376 = 625$, is not a four-digit number. We expect to get two n-digit solutions for about 90% of all n, but this conjecture has not been proved.

(Such self-reproducing numbers have been called "automorphic.")

4.46 (a) If $j'j - k'k = \gcd(j,k)$, we have $n^{k'k} n^{\gcd(j,k)} = n^{j'j} \equiv 1$ and $n^{k'k} \equiv 1$. (b) Let $n = pq$, where p is the smallest prime divisor of n. If $2^n \equiv 1 \pmod{n}$ then $2^n \equiv 1 \pmod{p}$. Also $2^{p-1} \equiv 1 \pmod{p}$; hence $2^{\gcd(p-1,n)} \equiv 1 \pmod{p}$. But $\gcd(p-1,n) = 1$ by the definition of p.

4.47 If $n^{m-1} \equiv 1 \pmod{m}$ we must have $n \perp m$. If $n^k \equiv n^j$ for some $1 \leqslant j < k < m$, then $n^{k-j} \equiv 1$ because we can divide by n^j. Therefore if the numbers $n^1 \bmod m, \ldots, n^{m-1} \bmod m$ are not distinct, there is a $k < m-1$ with $n^k \equiv 1$. The least such k divides $m-1$, by exercise 46(a). But then $kq = (m-1)/p$ for some prime p and some positive integer q; this is impossible, since $n^{kq} \not\equiv 1$. Therefore the numbers $n^1 \bmod m, \ldots, n^{m-1} \bmod m$ are distinct and relatively prime to m. Therefore the numbers $1, \ldots, m-1$ are relatively prime to m, and m must be prime.

4.48 By pairing numbers up with their inverses, we can reduce the product $\pmod{m}$ to $\prod_{1 \leqslant n < m,\, n^2 \bmod m = 1} n$. Now we can use our knowledge of the solutions to $n^2 \bmod m = 1$. By residue arithmetic we find that the result is $m-1$ if $m = 4$, p^k, or $2p^k$ ($p > 2$); otherwise it's $+1$.

4.49 (a) Either $m < n$ ($\Phi(N-1)$ cases) or $m = n$ (one case) or $m > n$ ($\Phi(N-1)$ again). Hence $R(N) = 2\Phi(N-1) + 1$. (b) From (4.62) we get

$$2\Phi(N-1) + 1 \;=\; 1 + \sum_{d \geqslant 1} \mu(d) \lfloor N/d \rfloor \lfloor N/d - 1 \rfloor ;$$

hence the stated result holds if and only if

$$\sum_{d \geqslant 1} \mu(d) \lfloor N/d \rfloor \;=\; 1 , \qquad \text{for } N \geqslant 1.$$

And this is a special case of (4.61) if we set $f(x) = [x \geqslant 1]$.

4.50 (a) If f is any function,

$$\begin{aligned}\sum_{0\leqslant k<m} f(k) &= \sum_{d\backslash m}\sum_{0\leqslant k<m} f(k)\big[d=\gcd(k,m)\big]\\ &= \sum_{d\backslash m}\sum_{0\leqslant k<m} f(k)\big[k/d\perp m/d\big]\\ &= \sum_{d\backslash m}\sum_{0\leqslant k<m/d} f(kd)\big[k\perp m/d\big]\\ &= \sum_{d\backslash m}\sum_{0\leqslant k<d} f(km/d)\big[k\perp d\big]\,;\end{aligned}$$

we saw a special case of this in the derivation of (4.63). An analogous derivation holds for $\prod$ instead of $\sum$. Thus we have

$$z^m-1 = \prod_{0\leqslant k<m}(z-\omega^k) = \prod_{d\backslash m}\prod_{\substack{0\leqslant k<d\\ k\perp d}}(z-\omega^{km/d}) = \prod_{d\backslash m}\Psi_d(z)$$

because $\omega^{m/d}=e^{2\pi i/d}$.

Part (b) follows from part (a) by the analog of (4.56) for products instead of sums. Incidentally, this formula shows that $\Psi_m(z)$ has integer coefficients, since $\Psi_m(z)$ is obtained by multiplying and dividing polynomials whose leading coefficient is 1.

4.51 $(x_1+\cdots+x_n)^p = \sum_{k_1+\cdots+k_n=p} p!/(k_1!\ldots k_n!)x_1^{k_1}\ldots x_n^{k_n}$, and the coefficient is divisible by p unless some $k_j=p$. Hence $(x_1+\cdots+x_n)^p \equiv x_1^p+\cdots+x_n^p \pmod p$. Now we can set all the x's to 1, obtaining $n^p\equiv n$.

4.52 If $p>n$ there is nothing to prove. Otherwise $x\perp p$, so $x^{k(p-1)}\equiv 1 \pmod p$; this means that at least $\lfloor (n-1)/(p-1)\rfloor$ of the given numbers are multiples of p. And $(n-1)/(p-1)\geqslant n/p$ since $n\geqslant p$.

"Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk."
—L. Kronecker [365]

4.53 First show that if $m\geqslant 6$ and m is not prime then $(m-2)!\equiv 0 \pmod m$. (If $m=p^2$, the product for $(m-2)!$ includes p and $2p$; otherwise it includes d and m/d where $d<m/d$.) Next consider cases:

Case 0, $n<5$. The condition holds for $n=1$ only.

Case 1, $n\geqslant 5$ and n is prime. Then $(n-1)!/(n+1)$ is an integer and it can't be a multiple of n.

Case 2, $n\geqslant 5$, n is composite, and $n+1$ is composite. Then n and $n+1$ divide $(n-1)!$, and $n\perp n+1$; hence $n(n+1)\backslash(n-1)!$.

Case 3, $n\geqslant 5$, n is composite, and $n+1$ is prime. Then $(n-1)!\equiv 1 \pmod{n+1}$ by Wilson's theorem, and

$$\lfloor (n-1)!/(n+1)\rfloor = \big((n-1)!+n\big)/(n+1)\,;$$

this is divisible by n.

Therefore the answer is: Either $n = 1$ or $n \neq 4$ is composite.

4.54 $\epsilon_2(1000!) > 500$ and $\epsilon_5(1000!) = 249$, hence $1000! = a \cdot 10^{249}$ for some even integer a. Since $1000 = (1300)_5$, exercise 40 tells us that $a \cdot 2^{249} = 1000!/5^{249} \equiv -1 \pmod 5$. Also $2^{249} \equiv 2$, hence $a \equiv 2$, hence $a \bmod 10 = 2$ or 7; hence the answer is $2 \cdot 10^{249}$.

4.55 One way is to prove by induction that $P_{2n}/P_n^4(n+1)$ is an integer; this stronger result helps the induction go through. Another way is based on showing that each prime p divides the numerator at least as often as it divides the denominator. This reduces to proving the inequality

$$\sum_{k=1}^{2n} \lfloor k/m \rfloor \;\geqslant\; 4 \sum_{k=1}^{n} \lfloor k/m \rfloor ,$$

which follows from

$$\lfloor (2n-1)/m \rfloor + \lfloor 2n/m \rfloor \;\geqslant\; \lfloor n/m \rfloor .$$

The latter is true when $0 \leqslant n < m$, and both sides increase by 4 when n is increased by m.

4.56 Let $f(m) = \sum_{k=1}^{2n-1} \min(k, 2n-k)[m \backslash k]$, $g(m) = \sum_{k=1}^{n-1} (2n-2k-1) \times \big[m \backslash (2k+1)\big]$. The number of times p divides the numerator of the stated product is $f(p) + f(p^2) + f(p^3) + \cdots$, and the number of times p divides the denominator is $g(p) + g(p^2) + g(p^3) + \cdots$. But $f(m) = g(m)$ whenever m is odd, by exercise 2.32. The stated product therefore reduces to $2^{n(n-1)}$, by exercise 3.22.

4.57 The hint suggests a standard interchange of summation, since

$$\sum_{1 \leqslant m \leqslant n} [d \backslash m] \;=\; \sum_{0 < k \leqslant n/d} [m = dk] \;=\; \lfloor n/d \rfloor .$$

Calling the hinted sum $\Sigma(n)$, we have

$$\Sigma(m+n) - \Sigma(m) - \Sigma(n) \;=\; \sum_{d \in S(m,n)} \varphi(d) .$$

On the other hand, we know from (4.54) that $\Sigma(n) = \frac{1}{2}n(n+1)$. Hence $\Sigma(m+n) - \Sigma(m) - \Sigma(n) = mn$.

4.58 The function $f(m)$ is multiplicative, and when $m = p^k$ it equals $1 + p + \cdots + p^k$. This is a power of 2 if and only if p is a Mersenne prime and $k = 1$. For k must be odd, and in that case the sum is

$$(1+p)(1+p^2+p^4+\cdots+p^{k-1})$$

and $(k-1)/2$ must be odd, etc. The necessary and sufficient condition is that m be a product of distinct Mersenne primes.

4.59 Proof of the hint: If $n = 1$ we have $x_1 = \alpha = 2$, so there's no problem. If $n > 1$ we can assume that $x_1 \leqslant \cdots \leqslant x_n$. Case 1: $x_1^{-1} + \cdots + x_{n-1}^{-1} + (x_n - 1)^{-1} \geqslant 1$ and $x_n > x_{n-1}$. Then we can find $\beta \geqslant x_n - 1 \geqslant x_{n-1}$ such that $x_1^{-1} + \cdots + x_{n-1}^{-1} + \beta^{-1} = 1$; hence $x_n \leqslant \beta + 1 \leqslant e_n$ and $x_1 \ldots x_n \leqslant x_1 \ldots x_{n-1}(\beta + 1) \leqslant e_1 \ldots e_n$, by induction. There is a positive integer m such that $\alpha = x_1 \ldots x_n/m$; hence $\alpha \leqslant e_1 \ldots e_n = e_{n+1} - 1$, and we have $x_1 \ldots x_n(\alpha + 1) \leqslant e_1 \ldots e_n e_{n+1}$. Case 2: $x_1^{-1} + \cdots + x_{n-1}^{-1} + (x_n - 1)^{-1} \geqslant 1$ and $x_n = x_{n-1}$. Let $a = x_n$ and $a^{-1} + (a-1)^{-1} = (a-2)^{-1} + \zeta^{-1}$. Then we can show that $a \geqslant 4$ and $(a-2)(\zeta + 1) \geqslant a^2$. So there's a $\beta \geqslant \zeta$ such that $x_1^{-1} + \cdots + x_{n-2}^{-1} + (a-2)^{-1} + \beta^{-1} = 1$; it follows by induction that $x_1 \ldots x_n \leqslant x_1 \ldots x_{n-2}(a-2)(\zeta + 1) \leqslant x_1 \ldots x_{n-2}(a-2)(\beta + 1) \leqslant e_1 \ldots e_n$, and we can finish as before. Case 3: $x_1^{-1} + \cdots + x_{n-1}^{-1} + (x_n - 1)^{-1} < 1$. Let $a = x_n$, and let $a^{-1} + \alpha^{-1} = (a-1)^{-1} + \beta^{-1}$. It can be shown that $(a-1)(\beta + 1) > a(\alpha + 1)$, because this identity is equivalent to

$$a\alpha^2 - a^2\alpha + a\alpha - a^2 + \alpha + a \;>\; 0\,,$$

which is a consequence of $a\alpha(\alpha - a) + (1 + a)\alpha \geqslant (1 + a)\alpha > a^2 - a$. Hence we can replace x_n and α by $a - 1$ and β, repeating this transformation until cases 1 or 2 apply.

Another consequence of the hint is that $1/x_1 + \cdots + 1/x_n < 1$ implies $1/x_1 + \cdots + 1/x_n \leqslant 1/e_1 + \cdots + 1/e_n$; see exercise 16.

4.60 The main point is that $\theta < \frac{2}{3}$. Then we can take p_1 sufficiently large (to meet the conditions below) and p_n to be the least prime greater than p_{n-1}^3. With this definition let $a_n = 3^{-n} \ln p_n$ and $b_n = 3^{-n} \ln(p_n + 1)$. If we can show that $a_{n-1} \leqslant a_n < b_n \leqslant b_{n-1}$, we can take $P = \lim_{n\to\infty} e^{a_n}$ as in exercise 37. But this hypothesis is equivalent to $p_{n-1}^3 \leqslant p_n < (p_{n-1} + 1)^3$. If there's no prime p_n in this range, there must be a prime $p < p_{n-1}^3$ such that $p + cp^\theta > (p_{n-1} + 1)^3$. But this implies that $cp^\theta > 3p^{2/3}$, which is impossible when p is sufficiently large.

"Man made the integers: All else is Dieudonné."
—R. K. Guy

We can almost certainly take $p_1 = 2$, since all available evidence indicates that the known bounds on gaps between primes are much weaker than the truth (see exercise 69). Then $p_2 = 11$, $p_3 = 1361$, $p_4 = 2521008887$, and $1.306377883863 < P < 1.306377883869$.

4.61 Let $\hat{m}$ and $\hat{n}$ be the right-hand sides; observe that $\hat{m}n' - m'\hat{n} = 1$, hence $\hat{m} \perp \hat{n}$. Also $\hat{m}/\hat{n} > m'/n'$ and $N = \big((n+N)/n'\big)n' - n \geqslant \hat{n} > \big((n+N)/n' - 1\big)n' - n = N - n' \geqslant 0$. So we have $\hat{m}/\hat{n} \geqslant m''/n''$. If equality doesn't hold, we have $n'' = (\hat{m}n' - m'\hat{n})n'' = n'(\hat{m}n'' - m''\hat{n}) + \hat{n}(m''n' - m'n'') \geqslant n' + \hat{n} > N$, a contradiction.

Incidentally, this exercise implies that $(m+m'')/(n+n'') = m'/n'$, although the former fraction is not always reduced.

4.62 $2^{-1}+2^{-2}+2^{-3}-2^{-6}-2^{-7}+2^{-12}+2^{-13}-2^{-20}-2^{-21}+2^{-30}+2^{-31}-2^{-42}-2^{-43}+\cdots$ can be written

$$\frac{1}{2}+3\sum_{k\geqslant 0}\left(2^{-4k^2-6k-3}-2^{-4k^2-10k-7}\right).$$

This sum, incidentally, can be expressed in closed form using the "theta function" $\theta(z,\lambda)=\sum_k e^{-\pi\lambda k^2+2izk}$; we have

$$e \quad\leftrightarrow\quad \tfrac{1}{2}+\tfrac{3}{8}\theta(\tfrac{4}{\pi}\ln 2,\,3i\ln 2)-\tfrac{3}{128}\theta(\tfrac{4}{\pi}\ln 2,\,5i\ln 2)\,.$$

4.63 Any $n>2$ either has a prime divisor d or is divisible by $d=4$. In either case, a solution with exponent n implies a solution $(a^{n/d})^d+(b^{n/d})^d=(c^{n/d})^d$ with exponent d. Since $d=4$ has no solutions, d must be prime.

I have discovered a wonderful proof of Fermat's Last Theorem, but there's no room for it here.

The hint follows from the binomial theorem, since $(a^p+(x-a)^p)/x\equiv pa^{p-1} \pmod{x}$ when p is odd. The smallest counterexample, if (4.46) fails, has $a\perp x$. If x is not divisible by p then x is relatively prime to c^p/x; this means that whenever q is prime and $q^e\backslash\backslash x$ and $q^f\backslash\backslash c$, we have $e=fp$. Hence $x=m^p$ for some m. On the other hand if x is divisible by p, then c^p/x is divisible by p but not by p^2, and c^p has no other factors in common with x.

4.64 Equal fractions in $\mathcal{P}_N$ appear in "organ-pipe order":

$$\frac{2m}{2n},\frac{4m}{4n},\ldots,\frac{rm}{rn},\ldots,\frac{3m}{3n},\frac{m}{n}\,.$$

Suppose that $\mathcal{P}_N$ is correct; we want to prove that $\mathcal{P}_{N+1}$ is correct. This means that if kN is odd, we want to show that

$$\frac{k-1}{N+1} \;=\; \mathcal{P}_{N,kN}\,;$$

if kN is even, we want to show that

$$\mathcal{P}_{N,kN-1}\;\;\mathcal{P}_{N,kN}\;\;\frac{k-1}{N+1}\;\;\mathcal{P}_{N,kN}\;\;\mathcal{P}_{N,kN+1}\,.$$

In both cases it will be helpful to know the number of fractions that are strictly less than $(k-1)/(N+1)$ in $\mathcal{P}_N$; this is

$$\begin{aligned}\sum_{n=1}^{N}\sum_{m}\left[0\leqslant\frac{m}{n}<\frac{k-1}{N+1}\right] &= \sum_{n=1}^{N}\left\lceil\frac{(k-1)n}{N+1}\right\rceil = \sum_{n=0}^{N}\left\lfloor\frac{(k-1)n+N}{N+1}\right\rfloor\\ &= \frac{(k-2)N}{2}+\frac{d-1}{2}+d\left\lfloor\frac{N}{d}\right\rfloor\end{aligned}$$

by (3.32), where $d = \gcd(k-1, N+1)$. And this reduces to $\frac{1}{2}(kN - d + 1)$, since $N \bmod d = d - 1$.

Furthermore, the number of fractions equal to $(k-1)/(N+1)$ in $\mathcal{P}_N$ that should precede it in $\mathcal{P}_{N+1}$ is $\frac{1}{2}\big(d - 1 - [d \text{ even}]\big)$, by the nature of organ-pipe order.

If kN is odd, then d is even and $(k-1)/(N+1)$ is preceded by $\frac{1}{2}(kN-1)$ elements of $\mathcal{P}_N$; this is just the correct number to make things work. If kN is even, then d is odd and $(k-1)/(N+1)$ is preceded by $\frac{1}{2}(kN)$ elements of $\mathcal{P}_N$. If $d = 1$, none of these equals $(k-1)/(N+1)$ and $\mathcal{P}_{N,kN}$ is '$<$'; otherwise $(k-1)/(N+1)$ falls between two equal elements and $\mathcal{P}_{N,kN}$ is '$=$'. (C. S. Peirce [288] independently discovered the Stern–Brocot tree at about the same time as he discovered $\mathcal{P}_N$.)

"No square less than 25×10^{14} divides a Euclid number."
—Ilan Vardi

4.65 The analogous question for the (analogous) Fermat numbers f_n is a famous unsolved problem. This one might be easier or harder.

4.66 It is known that no square less than 36×10^{18} divides a Mersenne number or Fermat number. But there has still been no proof of Schinzel's conjecture that there exist infinitely many squarefree Mersenne numbers. It is not even known if there are infinitely many p such that $p \backslash\backslash (a \pm b)$, where all prime factors of a and b are $\leqslant 31$.

4.67 M. Szegedy has proved this conjecture for all large n; see [348], [95, pp. 78–79], and [55].

4.68 This is a much weaker conjecture than the result in the following exercise.

4.69 Cramér [66] showed that this conjecture is plausible on probabilistic grounds, and computational experience bears this out: Brent [37] has shown that $P_{n+1} - P_n \leqslant 602$ for $P_{n+1} < 2.686 \times 10^{12}$. But the much weaker bounds in exercise 60 are the best that have been published so far [255]. Exercise 68 has a "yes" answer if $P_{n+1} - P_n < 2P_n^{1/2}$ for all sufficiently large n. According to Guy [169, problem A8], Paul Erdős offers \$10,000 for proof that there are infinitely many n such that

$$P_{n+1} - P_n \;>\; \frac{c \ln n \, \ln\ln n \, \ln\ln\ln\ln n}{(\ln\ln\ln n)^2}$$

for all $c > 0$.

4.70 This holds if and only if $\nu_2(n) = \nu_3(n)$, according to exercise 24. The methods of [96] may help to crack this conjecture.

4.71 When $k = 3$ the smallest solution is $n = 4700063497 = 19 \cdot 47 \cdot 5263229$; no other solutions are known in this case.

4.72 This is known to be true for infinitely many values of a, including -1 (of course) and 0 (not so obviously). Lehmer [244] has a famous conjecture that $\varphi(n)\backslash(n-1)$ if and only if n is prime.

4.73 This is known to be equivalent to the Riemann hypothesis (that the complex zeta function $\zeta(z)$ is nonzero when the real part of z is greater than $1/2$).

4.74 Experimental evidence suggests that there are about $p(1-1/e)$ distinct values, just as if the factorials were randomly distributed modulo p.

5.1 $(11)_r^4 = (14641)_r$, in any number system of radix $r \geqslant 7$, because of the binomial theorem. *What's 11^4 in radix 11?*

5.2 The ratio $\binom{n}{k+1}/\binom{n}{k} = (n-k)/(k+1)$ is $\leqslant 1$ when $k \geqslant \lfloor n/2 \rfloor$ and $\geqslant 1$ when $k < \lceil n/2 \rceil$, so the maximum occurs when $k = \lfloor n/2 \rfloor$ and $k = \lceil n/2 \rceil$.

5.3 Expand into factorials. Both products are equal to $f(n)/f(n-k)f(k)$, where $f(n) = (n+1)!\,n!\,(n-1)!$.

5.4 $\binom{-1}{k} = (-1)^k\binom{k+1-1}{k} = (-1)^k\binom{k}{k} = (-1)^k[k \geqslant 0]$.

5.5 If $0 < k < p$, there's a p in the numerator of $\binom{p}{k}$ with nothing to cancel it in the denominator. Since $\binom{p}{k} = \binom{p-1}{k} + \binom{p-1}{k-1}$, we must have $\binom{p-1}{k} \equiv (-1)^k \pmod{p}$, for $0 \leqslant k < p$.

5.6 The crucial step (after second down) should be

$$\begin{aligned}
&\frac{1}{n+1}\sum_k \binom{n+k}{k}\binom{n+1}{k+1}(-1)^k \\
&\quad = \frac{1}{n+1}\sum_{k\geqslant 0} \binom{n+k}{n}\binom{n+1}{k+1}(-1)^k \\
&\quad = \frac{1}{n+1}\sum_k \binom{n+k}{n}\binom{n+1}{k+1}(-1)^k \\
&\qquad - \frac{1}{n+1}\binom{n-1}{n}\binom{n+1}{0}(-1)^{-1}.
\end{aligned}$$

The original derivation forgot to include this extra term, which is $[n=0]$.

5.7 Yes, because $r^{\underline{-k}} = (-1)^k/(-r-1)^{\underline{k}}$. We also have

$$r^{\overline{k}}(r+\tfrac{1}{2})^{\overline{k}} \;=\; (2r)^{\overline{2k}}/2^{2k}.$$

5.8 $f(k) = (k/n-1)^n$ is a polynomial of degree n whose leading coefficient is n^{-n}. By (5.40), the sum is $n!/n^n$. When n is large, Stirling's approximation says that this is approximately $\sqrt{2\pi n}/e^n$. (This is quite different from $(1-1/e)$, which is what we get if we use the approximation $(1-k/n)^n \sim e^{-k}$, valid for fixed k as $n \to \infty$.)

5.9 $\mathcal{E}_t(z)^t = \sum_{k\geqslant 0} t(tk+t)^{k-1}z^k/k! = \sum_{k\geqslant 0}(k+1)^{k-1}(tz)^k/k! = \mathcal{E}_1(tz)$, by (5.60).

5.10 $\sum_{k\geqslant 0} 2z^k/(k+2) = F(2,1;3;z)$, since $t_{k+1}/t_k = (k+2)z/(k+3)$.

But not Imbesselian.

5.11 The first is Besselian and the second is Gaussian:

$$\begin{aligned} z^{-1}\sin z &= \textstyle\sum_{k\geqslant 0}(-1)^k z^{2k}/(2k+1)! \;=\; F(1;1,\tfrac{3}{2};-z^2/4)\,; \\ z^{-1}\arcsin z &= \textstyle\sum_{k\geqslant 0} z^{2k}(\tfrac{1}{2})^{\overline{k}}/(2k+1)k! \;=\; F(\tfrac{1}{2},\tfrac{1}{2};\tfrac{3}{2};z^2)\,. \end{aligned}$$

5.12 (a) Yes, if $n \neq 0$, since the term ratio is n. (b) Yes, when n is an integer; the term ratio is $(k+1)^n/k^n$. Notice that we get this term from (5.115) by setting $m = n+1$, $a_1 = \cdots = a_m = 1$, $b_1 = \cdots = b_n = 0$, $z = 1$, and multiplying by 0^n. (c) Yes, the term ratio is $(k+1)(k+3)/(k+2)$. (d) No, the term ratio is $1 + 1/(k+1)H_k$; and $H_k \sim \ln k$ isn't a rational function. (e) Yes, the reciprocal of any hypergeometric term is a hypergeometric term. The fact that $t(k) = \infty$ when $k < 0$ or $k > n$ does not exclude $t(k)$ from hypergeometric termhood. (f) Of course. (g) Not when, say, $t(k) = 2^k$ and $T(k) = 1$. (h) Yes; the term ratio $t(n-1-k)/t(n-1-(k+1))$ is a rational function (the reciprocal of the term ratio for t, with k replaced by $n-1-k$), for arbitrary n. (i) Yes; the term ratio can be written

Each value of a hypergeometric term $t(k)$ can be written $0^{e(k)}\nu(k)$, where $e(k)$ is an integer and $\nu(k) \neq 0$. Suppose the term ratio $t(k+1)/t(k)$ is $p(k)/q(k)$, and that p and q have been completely factored over the complex numbers. Then, for each k, $e(k+1)$ is $e(k)$ plus the number of zero factors of $p(k)$ minus the number of zero factors of $q(k)$, and $\nu(k+1)$ is $\nu(k)$ times the product of the nonzero factors of $p(k)$ divided by the product of the nonzero factors of $q(k)$.

$$\frac{a\,t(k{+}1)/t(k) + b\,t(k{+}2)/t(k) + c\,t(k{+}3)/t(k)}{a + b\,t(k{+}1)/t(k) + c\,t(k{+}2)/t(k)}\,,$$

and $t(k+m)/t(k) = \big(t(k+m)/t(k+m-1)\big)\ldots\big(t(k+1)/t(k)\big)$ is a rational function of k. (j) No. Whenever two rational functions $p_1(k)/q_1(k)$ and $p_2(k)/q_2(k)$ are equal for infinitely many k, they are equal for all k, because $p_1(k)q_2(k) = q_1(k)p_2(k)$ is a polynomial identity. Therefore the term ratio $\lceil(k+1)/2\rceil/\lceil k/2\rceil$ would have to equal 1 if it were a rational function. (k) No. The term ratio would have to be $(k+1)/k$, since it is $(k+1)/k$ for all $k > 0$; but then $t(-1)$ can be zero only if $t(0)$ is a multiple of 0^2, while $t(1)$ can be 1 only if $t(0) = 0^1$.

5.13 $R_n = n!^{n+1}/P_n^2 = Q_n/P_n = Q_n^2/n!^{n+1}$.

5.14 The first factor in (5.25) is $\binom{l-k}{l-k-m}$ when $k \leqslant l$, so it's $(-1)^{l-k-m} \times \binom{-m-1}{l-k-m}$. The sum for $k \leqslant l$ is the sum over all k, since $m \geqslant 0$. (The condition $n \geqslant 0$ isn't really needed, although k must assume negative values if $n < 0$.)

To go from (5.25) to (5.26), first replace s by $-1-n-q$.

5.15 If n is odd, the sum is zero, since we can replace k by $n-k$. If $n = 2m$, the sum is $(-1)^m(3m)!/m!^3$, by (5.29) with $a = b = c = m$.

5.16 This is just $(2a)!\,(2b)!\,(2c)!/(a+b)!\,(b+c)!\,(c+a)!$ times (5.29), if we write the summands in terms of factorials.

5.17 The formulas $\binom{2n-1/2}{n} = \binom{4n}{2n}/2^{2n}$ and $\binom{2n-1/2}{2n} = \binom{4n}{2n}/2^{4n}$ yield $\binom{2n-1/2}{n} = 2^{2n}\binom{2n-1/2}{2n}$.

5.18 $\binom{3r}{3k}\binom{3k}{k,k,k}/3^{3k}$.

5.19 $\mathcal{B}_{1-t}(-z)^{-1} = \sum_{k\geqslant 0}\binom{k-tk-1}{k}\big(-1/(k-tk-1)\big)(-z)^k$, by (5.60), and this is $\sum_{k\geqslant 0}\binom{tk}{k}\big(1/(tk-k+1)\big)z^k = \mathcal{B}_t(z)$.

5.20 It equals $F(-a_1,\ldots,-a_m;-b_1,\ldots,-b_n;(-1)^{m+n}z)$; see exercise 2.17.

5.21 $\lim_{n\to\infty}(n+m)^{\underline{m}}/n^m = 1$.

5.22 Multiplying and dividing instances of (5.83) gives

$$\begin{aligned}\frac{(-1/2)!}{x!\,(x-1/2)!} &= \lim_{n\to\infty}\binom{n+x}{n}\binom{n+x-1/2}{n}n^{-2x}\bigg/\binom{n-1/2}{n}\\ &= \lim_{n\to\infty}\binom{2n+2x}{2n}n^{-2x},\end{aligned}$$

by (5.34) and (5.36). Also

$$1/(2x)! = \lim_{n\to\infty}\binom{2n+2x}{2n}(2n)^{-2x}.$$

Hence, etc. The Gamma function equivalent, incidentally, is

$$\Gamma(x)\,\Gamma(x+\tfrac{1}{2}) = \Gamma(2x)\,\Gamma(\tfrac{1}{2})/2^{2x-1}.$$

5.23 $(-1)^n n_|$, see (5.50).

5.24 This sum is $\binom{n}{m}F\left(\begin{matrix}m-n,-m\\1/2\end{matrix}\middle|1\right) = \binom{2n}{2m}$, by (5.35) and (5.93).

5.25 This is equivalent to the easily proved identity

$$(a-b)\frac{a^{\overline{k}}}{(b+1)^{\overline{k}}} = a\,\frac{(a+1)^{\overline{k}}}{(b+1)^{\overline{k}}} - b\,\frac{a^{\overline{k}}}{b^{\overline{k}}}$$

as well as to the operator formula $a-b = (\vartheta+a)-(\vartheta+b)$.

Similarly, we have

$$\begin{aligned}&(a_1-a_2)\,F\left(\begin{matrix}a_1,a_2,a_3,\ldots,a_m\\b_1,\ldots,b_n\end{matrix}\middle|z\right)\\ &\quad= a_1\,F\left(\begin{matrix}a_1{+}1,a_2,a_3,\ldots,a_m\\b_1,\ldots,b_n\end{matrix}\middle|z\right) - a_2\,F\left(\begin{matrix}a_1,a_2{+}1,a_3,\ldots,a_m\\b_1,\ldots,b_n\end{matrix}\middle|z\right),\end{aligned}$$

because $a_1 - a_2 = (a_1 + k) - (a_2 + k)$. If $a_1 - b_1$ is a nonnegative integer d, this second identity allows us to express $F(a_1, \ldots, a_m; b_1, \ldots, b_n; z)$ as a linear combination of $F(a_2 + j, a_3, \ldots, a_m; b_2, \ldots, b_n; z)$ for $0 \leqslant j \leqslant d$, thereby eliminating an upper parameter and a lower parameter. Thus, for example, we get closed forms for $F(a, b; a - 1; z)$, $F(a, b; a - 2; z)$, etc.

Gauss [143, §7] derived analogous relations between $F(a, b; c; z)$ and any two "contiguous" hypergeometrics in which a parameter has been changed by ± 1. Rainville [301] generalized this to cases with more parameters.

5.26 If the term ratio in the original hypergeometric series is $t_{k+1}/t_k = r(k)$, the term ratio in the new one is $t_{k+2}/t_{k+1} = r(k+1)$. Hence

$$F\left(\begin{matrix} a_1, \ldots, a_m \\ b_1, \ldots, b_n \end{matrix}\middle| z\right) = 1 + \frac{a_1 \ldots a_m z}{b_1 \ldots b_n} F\left(\begin{matrix} a_1+1, \ldots, a_m+1, 1 \\ b_1+1, \ldots, b_n+1, 2 \end{matrix}\middle| z\right).$$

5.27 This is the sum of the even terms of $F(2a_1, \ldots, 2a_m; 2b_1, \ldots, 2b_m; z)$. We have $(2a)^{\overline{2k+2}}/(2a)^{\overline{2k}} = 4(k+a)(k+a+\frac{1}{2})$, etc.

Equating coefficients of z^n gives the Pfaff-Saalschütz formula (5.97).

5.28 $F\left(\begin{smallmatrix} a, b \\ c \end{smallmatrix}\middle| z\right) = (1-z)^{-a} F\left(\begin{smallmatrix} a, c-b \\ c \end{smallmatrix}\middle| \frac{-z}{1-z}\right) = (1-z)^{-a} F\left(\begin{smallmatrix} c-b, a \\ c \end{smallmatrix}\middle| \frac{-z}{1-z}\right) = (1-z)^{c-a-b} F\left(\begin{smallmatrix} c-a, c-b \\ c \end{smallmatrix}\middle| z\right)$. (Euler proved the identity by showing that both sides satisfy the same differential equation. The reflection law is often attributed to Euler, but it does not seem to appear in his published papers.)

5.29 The coefficients of z^n are equal, by Vandermonde's convolution. (Kummer's original proof was different: He considered $\lim_{m\to\infty} F(m, b-a; b; z/m)$ in the reflection law (5.101).)

5.30 Differentiate again to get $z(1-z)F''(z) + (2-3z)F'(z) - F(z) = 0$. Therefore $F(z) = F(1, 1; 2; z)$ by (5.108).

5.31 The condition $f(k) = T(k+1) - T(k)$ implies that $f(k+1)/f(k) = \big(T(k+2)/T(k+1) - 1\big)/\big(1 - T(k)/T(k+1)\big)$ is a rational function of k.

5.32 When summing a polynomial in k, Gosper's method reduces to the "method of undetermined coefficients." We have $q(k) = r(k) = 1$, and we try to solve $p(k) = s(k+1) - s(k)$. The method suggests letting $s(k)$ be a polynomial whose degree is $d = \deg(p) + 1$.

5.33 The solution to $k = (k-1)s(k+1) - (k+1)s(k)$ is $s(k) = -k + \frac{1}{2}$; hence the answer is $(1-2k)/2k(k-1) + C$.

5.34 The limiting relation holds because all terms for $k > c$ vanish, and $\epsilon - c$ cancels with $-c$ in the limit of the other terms. Therefore the second partial sum is $\lim_{\epsilon\to 0} F(-m, -n; \epsilon - m; 1) = \lim_{\epsilon\to 0} (\epsilon + n - m)^{\overline{m}}/(\epsilon - m)^{\overline{m}} = (-1)^m \binom{n-1}{m}$.

5.35 (a) $2^{-n}3^n[n \geqslant 0]$. (b) $(1-\frac{1}{2})^{-k-1}[k \geqslant 0] = 2^{k+1}[k \geqslant 0]$.

5.36 The sum of the digits of $m+n$ is the sum of the digits of m plus the sum of the digits of n, minus $p-1$ times the number of carries, because each carry decreases the digit sum by $p-1$. [See [226] for extensions of this result to generalized binomial coefficients.]

5.37 Dividing the first identity by $n!$ yields $\binom{x+y}{n} = \sum_k \binom{x}{k}\binom{y}{n-k}$, Vandermonde's convolution. The second identity follows, for example, from the formula $x^{\overline{k}} = (-1)^k(-x)^{\underline{k}}$ if we negate both x and y.

5.38 Choose c as large as possible such that $\binom{c}{3} \leqslant n$. Then $0 \leqslant n - \binom{c}{3} < \binom{c+1}{3} - \binom{c}{3} = \binom{c}{2}$; replace n by $n - \binom{c}{3}$ and continue in the same fashion. Conversely, any such representation is obtained in this way. (We can do the same thing with

$$n = \binom{a_1}{1} + \binom{a_2}{2} + \cdots + \binom{a_m}{m}, \qquad 0 \leqslant a_1 < a_2 < \cdots < a_m$$

for any fixed m.)

5.39 $x^m y^n = \sum_{k=1}^m \binom{m+n-1-k}{n-1} a^n b^{m-k} x^k + \sum_{k=1}^n \binom{m+n-1-k}{m-1} a^{n-k} b^m y^k$ for all $mn > 0$, by induction on $m+n$.

5.40 $(-1)^{m+1} \sum_{k=1}^n \sum_{j=1}^m \binom{r}{j}\binom{m-rk-s-1}{m-j} = (-1)^m \sum_{k=1}^n \Bigl(\binom{m-rk-s-1}{m} - \binom{m-r(k-1)-s-1}{m}\Bigr) = (-1)^m \Bigl(\binom{m-rn-s-1}{m} - \binom{m-s-1}{m}\Bigr) = \binom{rn+s}{m} - \binom{s}{m}$.

5.41 $\sum_{k\geqslant 0} n!/(n-k)!\,(n+k+1)! = \bigl(n!/(2n+1)!\bigr) \sum_{k>n} \binom{2n+1}{k}$, which is $2^{2n} n!/(2n+1)!$.

5.42 We treat n as an indeterminate real variable. Gosper's method with $q(k) = k+1$ and $r(k) = k-1-n$ has the solution $s(k) = 1/(n+2)$; hence the desired indefinite sum is $(-1)^{x-1} \frac{n+1}{n+2} / \binom{n+1}{x}$. And

$$\sum_{k=0}^n (-1)^k \Big/ \binom{n}{k} = (-1)^{x-1} \frac{n+1}{n+2} \Big/ \binom{n+1}{x}\Bigg|_0^{n+1} = 2\,\frac{n+1}{n+2}\,[n \text{ even}]\,.$$

This exercise, incidentally, implies the formula

$$\frac{1}{n\binom{n-1}{k}} = \frac{1}{(n+1)\binom{n}{k+1}} + \frac{1}{(n+1)\binom{n}{k}}\,,$$

a "dual" to the basic recurrence (5.8).

5.43 After the hinted first step we can apply (5.21) and sum on k. Then (5.21) applies again and Vandermonde's convolution finishes the job. (A combinatorial proof of this identity has been given by Andrews [10]. There's a quick way to go from this identity to a proof of (5.29), explained in [207, exercise 1.2.6–62].)

5.44 Cancellation of factorials shows that

$$\binom{m}{j}\binom{n}{k}\binom{m+n}{m} = \binom{m+n-j-k}{m-j}\binom{j+k}{j}\binom{m+n}{j+k},$$

so the second sum is $1/\binom{m+n}{m}$ times the first. And the first is just the special case $l=0$, $n=b$, $r=a$, $s=m+n-b$ of (5.32), so it is $\binom{a+b}{a}\binom{m+n-a-b}{n-a}$.

5.45 According to (5.9), $\sum_{k\leqslant n}\binom{k-1/2}{k} = \binom{n+1/2}{n}$. If this form of the answer isn't "closed" enough, we can apply (5.35) and get $(2n+1)\binom{2n}{n}4^{-n}$.

5.46 By (5.69), this convolution is the negative of the coefficient of z^{2n} in $\mathcal{B}_{-1}(z)\mathcal{B}_{-1}(-z)$. Now $(2\mathcal{B}_{-1}(z)-1)(2\mathcal{B}_{-1}(-z)-1) = \sqrt{1-16z^2}$; hence $\mathcal{B}_{-1}(z)\mathcal{B}_{-1}(-z) = \frac{1}{4}\sqrt{1-16z^2}+\frac{1}{2}\mathcal{B}_{-1}(z)+\frac{1}{2}\mathcal{B}_{-1}(-z)-\frac{1}{4}$. By the binomial theorem,

$$(1-16z^2)^{1/2} = \sum_n \binom{1/2}{n}(-16)^n z^{2n} = -\sum_n \binom{2n}{n}\frac{4^n z^{2n}}{2n-1},$$

so the answer is $\binom{2n}{n}4^{n-1}/(2n-1)+\binom{4n-1}{2n}/(4n-1)$.

The boxed sentence on the other side of this page is true.

5.47 It's the coefficient of z^n in $\big(\mathcal{B}_r(z)^s/Q_r(z)\big)\big(\mathcal{B}_r(z)^{-s}/Q_r(z)\big) = Q_r(z)^{-2}$, where $Q_r(z) = 1-r+r\mathcal{B}_r(z)^{-1}$, by (5.61).

5.48 $F(2n+2,1;n+2;\frac{1}{2}) = 2^{2n+1}/\binom{2n+1}{n+1}$, a special case of (5.111).

5.49 Saalschütz's identity (5.97) yields

$$\binom{x+n}{n}\frac{y}{y+n}\,F\left(\begin{matrix}-x,\ -n,\ -n-y\\ -x-n,\ 1-n-y\end{matrix}\middle|\,1\right) = \frac{(y-x)^{\overline{n}}}{(y+1)^{\overline{n}}}.$$

5.50 The left-hand side is

$$\sum_{k\geqslant 0}\frac{a^{\overline{k}}\,b^{\overline{k}}}{c^{\overline{k}}}\frac{(-z)^k}{k!}\sum_{m\geqslant 0}\binom{k+a+m-1}{m}z^m$$
$$= \sum_{n\geqslant 0} z^n \sum_{k\geqslant 0}\frac{a^{\overline{k}}\,b^{\overline{k}}}{c^{\overline{k}}\,k!}(-1)^k\binom{n+a-1}{n-k}$$

and the coefficient of z^n is

$$\binom{n+a-1}{n}F\left(\begin{matrix}a,b,-n\\ c,a\end{matrix}\middle|\,1\right)\frac{a^{\overline{n}}}{n!} = \frac{(c-b)^{\overline{n}}}{c^{\overline{n}}}$$

by Vandermonde's convolution (5.92).

5.51 (a) Reflection gives $F(a,-n;2a;2) = (-1)^n F(a,-n;2a;2)$. (Incidentally, this formula implies the remarkable identity $\Delta^{2m+1}f(0) = 0$, when $f(n) = 2^n x^{\underline{n}}/(2x)^{\underline{n}}$.)

(b) The term-by-term limit is $\sum_{0\leqslant k\leqslant m}\binom{m}{k}\frac{2m+1}{2m+1-k}(-2)^k$ plus an additional term for $k=2m-1$. The additional term is

$$\frac{(-m)\ldots(-1)\,(1)\ldots(m)\,(-2m+1)\ldots(-1)\,2^{2m+1}}{(-2m)\ldots(-1)\,(2m-1)!}$$
$$= (-1)^{m+1}\frac{m!\,m!\,2^{2m+1}}{(2m)!} = \frac{-2}{\binom{-1/2}{m}};$$

hence, by (5.104), this limit is $-1/\binom{-1/2}{m}$, the negative of what we had.

5.52 The terms of both series are zero for $k>N$. This identity corresponds to replacing k by $N-k$. Notice that

$$\begin{aligned} a^{\overline{N}} &= a^{\overline{N-k}}(a+N-k)^{\overline{k}} \\ &= a^{\overline{N-k}}(a+N-1)^{\underline{k}} = a^{\overline{N-k}}(1-a-N)^{\overline{k}}(-1)^k. \end{aligned}$$

5.53 When $b=-\frac{1}{2}$, the left side of (5.110) is $1-2z$ and the right side is $(1-4z+4z^2)^{1/2}$, independent of a. The right side is the formal power series

$$1+\binom{1/2}{1}4z(z-1)+\binom{1/2}{2}16z^2(z-1)^2+\cdots,$$

which can be expanded and rearranged to give $1-2z+0z^2+0z^3+\cdots$; but the rearrangement involves divergent series in its intermediate steps when $z=1$, so it is not legitimate.

The boxed sentence on the other side of this page is false.

5.54 If $m+n$ is odd, say $2N-1$, we want to show that

$$\lim_{\epsilon\to 0} F\left(\begin{matrix} N-m-\frac{1}{2},\ -N+\epsilon \\ -m+\epsilon \end{matrix}\middle|\, 1\right) = 0.$$

Equation (5.92) applies, since $-m+\epsilon > -m-\frac{1}{2}+\epsilon$, and the denominator factor $\Gamma(c-b)=\Gamma(N-m)$ is infinite since $N\leqslant m$; the other factors are finite. Otherwise $m+n$ is even; setting $n=m-2N$ we have

$$\lim_{\epsilon\to 0} F\left(\begin{matrix} -N,\ N-m-\frac{1}{2}+\epsilon \\ -m+\epsilon \end{matrix}\middle|\, 1\right) = \frac{(N-1/2)^{\underline{N}}}{m^{\underline{N}}}$$

by (5.93). The remaining job is to show that

$$\binom{m}{m-2N}\frac{(N-1/2)!}{(-1/2)!}\frac{(m-N)!}{m!} = \binom{m-N}{m-2N}2^{-2N},$$

and this is the case $x=N$ of exercise 22.

5.55 Let $Q(k) = (k + A_1) \ldots (k + A_M)Z$ and $R(k) = (k + B_1) \ldots (k + B_N)$. Then $t(k+1)/t(k) = P(k)Q(k-1)/P(k-1)R(k)$, where $P(k) = Q(k) - R(k)$ is a nonzero polynomial.

5.56 The solution to $-(k+1)(k+2) = s(k+1)+s(k)$ is $s(k) = -\frac{1}{2}k^2-k-\frac{1}{4}$; hence $\sum \binom{-3}{k} \delta k = \frac{1}{8}(-1)^{k-1}(2k^2 + 4k + 1) + C$. Also

$$\begin{aligned}
&(-1)^{k-1} \left\lfloor \frac{k+1}{2} \right\rfloor \left\lfloor \frac{k+2}{2} \right\rfloor \\
&\quad = \frac{(-1)^{k-1}}{4} \left(k + 1 - \frac{1+(-1)^k}{2} \right) \left(k + 2 - \frac{1-(-1)^k}{2} \right) \\
&\quad = \frac{(-1)^{k-1}}{8} (2k^2 + 4k + 1) + \frac{1}{8} .
\end{aligned}$$

5.57 We have $t(k+1)/t(k) = (k-n)(k+1+\theta)(-z)/(k+1)(k+\theta)$. Therefore we let $p(k) = k + \theta$, $q(k) = (k - n)(-z)$, $r(k) = k$. The secret function $s(k)$ must be a constant α_0, and we have

$$k + \theta \;=\; \big(-z(k - n) - k\big)\, \alpha_0 \,;$$

hence $\alpha_0 = -1/(1 + z)$ and $\theta = -nz/(1 + z)$. The sum is

$$\sum \binom{n}{k} z^k \left(k - \frac{nz}{1+z} \right) \delta k \;=\; -\frac{n}{1+z} \binom{n-1}{k-1} z^k + C \,.$$

(The special case $z = 1$ was mentioned in (5.18).)

5.58 If $m > 0$ we can replace $\binom{k}{m}$ by $\frac{k}{m}\binom{k-1}{m-1}$ and derive the formula $T_{m,n} = \frac{n}{m} T_{m-1,n-1} - \frac{1}{m}\binom{n-1}{m}$. The summation factor $\binom{n}{m}^{-1}$ is therefore appropriate:

$$\frac{T_{m,n}}{\binom{n}{m}} \;=\; \frac{T_{m-1,n-1}}{\binom{n-1}{m-1}} - \frac{1}{m} + \frac{1}{n} \,.$$

We can unfold this to get

$$\frac{T_{m,n}}{\binom{n}{m}} \;=\; T_{0,n-m} - H_m + H_n - H_{n-m} \,.$$

Finally $T_{0,n-m} = H_{n-m}$, so $T_{m,n} = \binom{n}{m}(H_n - H_m)$. (It's also possible to derive this result by using generating functions; see Example 2 in Section 7.5.)

5.59 $\sum_{j \geqslant 0,\, k \geqslant 1} \binom{n}{j} \big[j = \lfloor \log_m k \rfloor \big] = \sum_{j \geqslant 0,\, k \geqslant 1} \binom{n}{j} [m^j \leqslant k < m^{j+1}]$, which is $\sum_{j \geqslant 0} \binom{n}{j} (m^{j+1} - m^j) = (m - 1) \sum_{j \geqslant 0} \binom{n}{j} m^j = (m - 1)(m + 1)^n$.

5.60 $\binom{2n}{n} \approx 4^n/\sqrt{\pi n}$ is the case $m = n$ of

$$\binom{m+n}{n} \approx \sqrt{\frac{1}{2\pi}\Big(\frac{1}{m}+\frac{1}{n}\Big)}\Big(1+\frac{m}{n}\Big)^n\Big(1+\frac{n}{m}\Big)^m .$$

5.61 Let $\lfloor n/p \rfloor = q$ and $n \bmod p = r$. The polynomial identity $(x+1)^p \equiv x^p + 1 \pmod p$ implies that

$$(x+1)^{pq+r} \equiv (x+1)^r (x^p+1)^q \pmod p .$$

The coefficient of x^m on the left is $\binom{n}{m}$. On the right it's $\sum_k \binom{r}{m-pk}\binom{q}{k}$, which is just $\binom{r}{m \bmod p}\binom{q}{\lfloor m/p \rfloor}$ because $0 \leqslant r < p$.

5.62 $\binom{np}{mp} = \sum_{k_1+\cdots+k_n=mp} \binom{p}{k_1}\cdots\binom{p}{k_n} \equiv \binom{n}{m} \pmod{p^2}$, because all terms of the sum are multiples of p^2 except for the $\binom{n}{m}$ terms in which exactly m of the k's are equal to p. (Stanley [335, exercise 1.6(d)] shows that the congruence actually holds modulo p^3 when $p > 3$.)

5.63 This is $S_n = \sum_{k=0}^n (-4)^k \binom{n+k}{n-k} = \sum_{k=0}^n (-4)^{n-k}\binom{2n-k}{k}$. The denominator of (5.74) is zero when $z = -1/4$, so we can't simply plug into that formula. The recurrence $S_n = -2S_{n-1} - S_{n-2}$ leads to the solution $S_n = (-1)^n(2n+1)$.

5.64 $\sum_{k\geqslant 0}\big(\binom{n}{2k} + \binom{n}{2k+1}\big)/(k+1) = \sum_{k\geqslant 0}\binom{n+1}{2k+1}/(k+1)$, which is

$$\frac{2}{n+2}\sum_{k\geqslant 0}\binom{n+2}{2k+2} = \frac{2^{n+2}-2}{n+2} .$$

5.65 Multiply both sides by n^{n-1} and replace k by $n-1-k$ to get

$$\begin{aligned}\sum_k \binom{n-1}{k} n^k (n-k)! &= (n-1)! \sum_{k=0}^{n-1}\big(n^{k+1}/k! - n^k/(k-1)!\big)\\ &= (n-1)!\, n^n/(n-1)! .\end{aligned}$$

(The partial sums can, in fact, be found by Gosper's algorithm.) Alternatively, $\binom{n}{k} k n^{n-1-k} k!$ can be interpreted as the number of mappings of $\{1,\ldots,n\}$ into itself with $f(1),\ldots,f(k)$ distinct but $f(k+1) \in \{f(1),\ldots,f(k)\}$; summing on k must give n^n.

5.66 This is a walk-the-garden-path problem where there's only one "obvious" way to proceed at every step. First replace $k-j$ by l, then replace $\lfloor \sqrt{l} \rfloor$ by k, getting

$$\sum_{j,k\geqslant 0}\binom{-1}{j-k}\binom{j}{m}\frac{2k+1}{2^j} .$$

The infinite series converges because the terms for fixed j are dominated by a polynomial in j divided by 2^j. Now sum over k, getting

$$\sum_{j\geqslant 0}\binom{j}{m}\frac{j+1}{2^j}\,.$$

Absorb the $j+1$ and apply (5.57) to get the answer, $4(m+1)$.

5.67 $3\binom{2n+2}{n+5}$ by (5.26), because

$$\binom{\binom{k}{2}}{2} = 3\binom{k+1}{4}\,.$$

5.68 Using the fact that

$$\sum_{k\leqslant n/2}\binom{n}{k} = 2^{n-1}+\frac{1}{2}\binom{n}{n/2}[n \text{ is even}]\,,$$

we get $n\bigl(2^{n-1}-\binom{n-1}{\lfloor n/2\rfloor}\bigr)$.

The boxed sentence on the other side of this page is not a sentence.

5.69 Since $\binom{k+1}{2}+\binom{l-1}{2}\leqslant\binom{k}{2}+\binom{l}{2} \iff k<l$, the minimum occurs when the k's are as equal as possible. Hence, by the equipartition formula of Chapter 3, the minimum is

$$\begin{aligned}&(n \bmod m)\binom{\lceil n/m\rceil}{2}+(n-(n \bmod m))\binom{\lfloor n/m\rfloor}{2}\\ &\quad= n\binom{\lfloor n/m\rfloor}{2}+(n \bmod m)\left\lfloor\frac{n}{m}\right\rfloor\,.\end{aligned}$$

A similar result holds for any lower index in place of 2.

5.70 This is $F(-n,\frac{1}{2};1;2)$; but it's also $(-2)^{-n}\binom{2n}{n}F(-n,-n;\frac{1}{2}-n;\frac{1}{2})$ if we replace k by $n-k$. Now $F(-n,-n;\frac{1}{2}-n;\frac{1}{2})=F(-\frac{n}{2},-\frac{n}{2};\frac{1}{2}-n;1)$ by Gauss's identity (5.111). (Alternatively, $F(-n,-n;\frac{1}{2}-n;\frac{1}{2})=2^{-n}F(-n,\frac{1}{2};\frac{1}{2}-n;-1)$ by the reflection law (5.101), and Kummer's formula (5.94) relates this to (5.55).) The answer is 0 when n is odd, $2^{-n}\binom{n}{n/2}$ when n is even. (See [164, §1.2] for another derivation. This sum arises in the study of a simple search algorithm [195].)

5.71 (a) Observe that

$$S(z) = \sum_{k\geqslant 0}a_k\frac{z^{m+k}}{(1-z)^{m+2k+1}} = \frac{z^m}{(1-z)^{m+1}}A\bigl(z/(1-z)^2\bigr)\,.$$

(b) Here $A(z)=\sum_{k\geqslant 0}\binom{2k}{k}(-z)^k/(k+1)=\bigl(\sqrt{1+4z}-1\bigr)/2z$, so we have $A\bigl(z/(1-z)^2\bigr)=1-z$. Thus $S_n=[z^n]\,\bigl(z/(1-z)\bigr)^m=\binom{n-1}{n-m}$.

5.72 The stated quantity is $m(m-n)\ldots\bigl(m-(k-1)n\bigr)n^{k-\nu(k)}/k!$. Any prime divisor p of n divides the numerator at least $k-\nu(k)$ times and divides the denominator at most $k-\nu(k)$ times, since this is the number of times 2 divides $k!$. A prime p that does not divide n must divide the product $m(m-n)\ldots\bigl(m-(k-1)n\bigr)$ at least as often as it divides $k!$, because $m(m-n)\ldots\bigl(m-(p^r-1)n\bigr)$ is a multiple of p^r for all $r \geqslant 1$ and all m.

5.73 Plugging in $X_n = n!$ yields $\alpha = \beta = 1$; plugging in $X_n = n_{\underline{j}}$ yields $\alpha = 1$, $\beta = 0$. Therefore the general solution is $X_n = \alpha n_{\underline{j}} + \beta(n! - n_{\underline{j}})$.

5.74 $\binom{n+1}{k} - \binom{n-1}{k-1}$, for $1 \leqslant k \leqslant n$.

5.75 The recurrence $S_k(n+1) = S_k(n) + S_{(k-1) \bmod 3}(n)$ makes it possible to verify inductively that two of the S's are equal and that $S_{(-n) \bmod 3}(n)$ differs from them by $(-1)^n$. These three values split their sum $S_0(n)+S_1(n)+S_2(n) = 2^n$ as equally as possible, so there must be $2^n \bmod 3$ occurrences of $\lceil 2^n/3 \rceil$ and $3 - (2^n \bmod 3)$ occurrences of $\lfloor 2^n/3 \rfloor$.

5.76 $Q_{n,k} = (n+1)\binom{n}{k} - \binom{n}{k+1}$.

5.77 The terms are zero unless $k_1 \leqslant \cdots \leqslant k_m$, when the product is the multinomial coefficient

$$\binom{k_m}{k_1,\ k_2-k_1,\ \ldots,\ k_m-k_{m-1}}.$$

Therefore the sum over $k_1, \ldots, k_{m-1}$ is m^{k_m}, and the final sum over k_m yields $(m^{n+1}-1)/(m-1)$.

The boxed sentence on the other side of this page is not boxed.

5.78 Extend the sum to $k = 2m^2 + m - 1$; the new terms are $\binom{1}{4} + \binom{2}{6} + \cdots + \binom{m-1}{2m} = 0$. Since $m \perp (2m+1)$, the pairs $\bigl(k \bmod m, k \bmod (2m+1)\bigr)$ are distinct. Furthermore, the numbers $(2j+1) \bmod (2m+1)$ as j varies from 0 to $2m$ are the numbers $0, 1, \ldots, 2m$ in some order. Hence the sum is

$$\sum_{\substack{0 \leqslant k < m \\ 0 \leqslant j < 2m+1}} \binom{k}{j} = \sum_{0 \leqslant k < m} 2^k = 2^m - 1.$$

5.79 (a) The sum is 2^{2n-1}, so the gcd must be a power of 2. If $n = 2^k q$ where q is odd, $\binom{2n}{1}$ is divisible by 2^{k+1} and not by 2^{k+2}. Each $\binom{2n}{2j+1}$ is divisible by 2^{k+1} (see exercise 36), so this must be the gcd. (b) If $p^r \leqslant n+1 < p^{r+1}$, we get the most radix p carries by adding k to $n-k$ when $k = p^r - 1$. The number of carries in this case is $r - \epsilon_p(n+1)$, and $r = \epsilon_p\bigl(L(n+1)\bigr)$.

5.80 First prove by induction that $k! \geqslant (k/e)^k$.

5.81 Let $f_{l,m,n}(x)$ be the left-hand side. It is sufficient to show that we have $f_{l,m,n}(1) > 0$ and that $f'_{l,m,n}(x) < 0$ for $0 \leqslant x \leqslant 1$. The value of $f_{l,m,n}(1)$ is $(-1)^{n-m-1}\binom{l+m+\theta}{l+n}$ by (5.23), and this is positive because the binomial coefficient has exactly $n-m-1$ negative factors. The inequality is true when $l = 0$, for the same reason. If $l > 0$, we have $f'_{l,m,n}(x) = -l\, f_{l-1,m,n+1}(x)$, which is negative by induction.

5.82 Let $\epsilon_p(a)$ be the exponent by which the prime p divides a, and let $m = n - k$. The identity to be proved reduces to

$$\begin{aligned}&\min\bigl(\epsilon_p(m)-\epsilon_p(m+k), \epsilon_p(m+k+1)-\epsilon_p(k+1), \epsilon_p(k)-\epsilon_p(m+1)\bigr)\\ &\quad = \min\bigl(\epsilon_p(k)-\epsilon_p(m+k), \epsilon_p(m)-\epsilon_p(k+1), \epsilon_p(m+k+1)-\epsilon_p(m+1)\bigr)\,.\end{aligned}$$

For brevity let's write this as $\min(x_1, y_1, z_1) = \min(x_2, y_2, z_2)$. Notice that $x_1 + y_1 + z_1 = x_2 + y_2 + z_2$. The general relation

$$\epsilon_p(a) \;<\; \epsilon_p(b) \quad\Longrightarrow\quad \epsilon_p(a) \;=\; \epsilon_p\bigl(|a \pm b|\bigr)$$

allows us to conclude that $x_1 \neq x_2 \Longrightarrow \min(x_1, x_2) = 0$; the same holds also for (y_1, y_2) and (z_1, z_2). It's now a simple matter to complete the proof.

5.83 (Solution by P. Paule.) Let r be a nonnegative integer. The given sum is the coefficient of $x^l y^m$ in

$$\begin{aligned}&\sum_{j,k}(-1)^{j+k}\frac{(1+x)^{j+k}}{x^k}\binom{r}{j}\binom{n}{k}(1+y)^{s+n-j-k}y^j\\ &\quad= \left(1-\frac{(1+x)y}{1+y}\right)^r\left(1-\frac{1+x}{(1+y)x}\right)^n(1+y)^{s+n}\\ &\quad= (-1)^n(1-xy)^{n+r}(1+y)^{s-r}/x^n\,,\end{aligned}$$

so it is clearly $(-1)^l\binom{n+r}{n+l}\binom{s-r}{m-n-l}$. (See also exercise 106.)

5.84 Following the hint, we get

$$z\mathcal{B}_t(z)^{r-1}\mathcal{B}'_t(z) \;=\; \sum_{k\geqslant 0}\binom{tk+r}{k}\frac{kz^k}{tk+r}\,,$$

and a similar formula for $\mathcal{E}_t(z)$. Thus the formulas $\bigl(zt\mathcal{B}_t^{-1}(z)\mathcal{B}'_t(z)+1\bigr)\mathcal{B}_t(z)^r$ and $\bigl(zt\mathcal{E}_t^{-1}(z)\mathcal{E}'_t(z)+1\bigr)\mathcal{E}_t(z)^r$ give the respective right-hand sides of (5.61). We must therefore prove that

$$\begin{aligned}\bigl(zt\mathcal{B}_t^{-1}(z)\mathcal{B}'_t(z)+1\bigr)\mathcal{B}_t(z)^r &= \frac{1}{1-t+t\mathcal{B}_t(z)^{-1}}\,,\\ \bigl(zt\mathcal{E}_t^{-1}(z)\mathcal{E}'_t(z)+1\bigr)\mathcal{E}_t(z)^r &= \frac{1}{1-zt\mathcal{E}(z)^t}\,,\end{aligned}$$

and these follow from (5.59).

5.85 If $f(x) = a_n x^n + \cdots + a_1 x + a_0$ is any polynomial of degree $\leqslant n$, we can prove inductively that

$$\sum_{0\leqslant \epsilon_1,\ldots,\epsilon_n\leqslant 1} (-1)^{\epsilon_1+\cdots+\epsilon_n} f(\epsilon_1 x_1+\cdots+\epsilon_n x_n) \;=\; (-1)^n n!\, a_n x_1 \ldots x_n\,.$$

The stated identity is the special case where $a_n = 1/n!$ and $x_k = k^3$.

5.86 (a) First expand with $n(n-1)$ index variables l_{ij} for all $i \neq j$. Setting $k_{ij} = l_{ij} - l_{ji}$ for $1 \leqslant i < j < n$ and using the constraints $\sum_{i\neq j}(l_{ij} - l_{ji}) = 0$ for all $i < n$ allows us to carry out the sums on l_{jn} for $1 \leqslant j < n$ and then on l_{ji} for $1 \leqslant i < j < n$ by Vandermonde's convolution. (b) $f(z) - 1$ is a polynomial of degree $< n$ that has n roots, so it must be zero. (c) Consider the constant terms in

$$\prod_{\substack{1\leqslant i,j\leqslant n\\ i\neq j}} \left(1-\frac{z_i}{z_j}\right)^{a_i} \;=\; \sum_{k=1}^{n} \prod_{\substack{1\leqslant i,j\leqslant n\\ i\neq j}} \left(1-\frac{z_i}{z_j}\right)^{a_i-[i=k]}\,.$$

5.87 The first term is $\sum_k \binom{n-k}{k} z^{mk}$, by (5.61). The summands in the second term are

$$\begin{aligned}\frac{1}{m}\sum_{k\geqslant 0}&\binom{(n+1)/m+(1{+}1/m)k}{k}(\zeta z)^{k+n+1}\\ &= \frac{1}{m}\sum_{k>n}\binom{(1{+}1/m)k-n-1}{k-n-1}(\zeta z)^k\,.\end{aligned}$$

Since $\sum_{0\leqslant j<m}(\zeta^{2j+1})^k = m(-1)^l[k{=}ml]$, these terms sum to

$$\begin{aligned}\sum_{k>n/m}&\binom{(1{+}1/m)mk-n-1}{mk-n-1}(-z^m)^k\\ &= \sum_{k>n/m}\binom{(m{+}1)k-n-1}{k}(-z^m)^k \;=\; \sum_{k>n/m}\binom{n-mk}{k}z^{mk}\,.\end{aligned}$$

Incidentally, the functions $\mathcal{B}_m(z^m)$ and $\zeta^{2j+1} z\,\mathcal{B}_{1+1/m}(\zeta^{2j+1}z)^{1/m}$ are the $m+1$ complex roots of the equation $w^{m+1} - w^m = z^m$.

5.88 Use the facts that $\int_0^\infty (e^{-t} - e^{-nt})\,dt/t = \ln n$ and $(1-e^{-t})/t \leqslant 1$. (We have $\binom{x}{k} = O(k^{-x-1})$ as $k \to \infty$, by (5.83); so this bound implies that Stirling's series $\sum_k s_k \binom{x}{k}$ converges when $x > -1$. Hermite [186] showed that the sum is $\ln\Gamma(1+x)$.)

5.89 Adding this to (5.19) gives $y^{-r}(x+y)^{m+r}$ on both sides, by the binomial theorem. Differentiation gives

$$\sum_{k>m}\binom{m+r}{k}\binom{m-k}{n}x^k y^{m-k-n} = \sum_{k>m}\binom{-r}{k}\binom{m-k}{n}(-x)^k(x+y)^{m-k-n},$$

and we can replace k by $k+m+1$ and apply (5.15) to get

$$\sum_{k\geqslant 0}\binom{m+r}{m+1+k}\binom{-n-1}{k}(-x)^{m+1+k}y^{-1-k-n} = \sum_{k\geqslant 0}\binom{-r}{m+1+k}\binom{-n-1}{k}x^{m+1+k}(x+y)^{-1-k-n}.$$

In hypergeometric form, this reduces to

$$F\left(\begin{matrix}1-r,\ n+1\\ m+2\end{matrix}\middle|\,\frac{-x}{y}\right) = \left(1+\frac{x}{y}\right)^{-n-1} F\left(\begin{matrix}m+1+r,\ n+1\\ m+2\end{matrix}\middle|\,\frac{x}{x+y}\right),$$

which is the special case $(a,b,c,z) = (n+1, m+1+r, m+2, -x/y)$ of the reflection law (5.101). (Thus (5.105) is related to reflection and to the formula in exercise 52.)

The boxed sentence on the other side of this page is self-referential.

5.90 If r is a nonnegative integer, the sum is finite, and the derivation in the text is valid as long as none of the terms of the sum for $0 \leqslant k \leqslant r$ has zero in the denominator. Otherwise the sum is infinite, and the kth term $\binom{k-r-1}{k}/\binom{k-s-1}{k}$ is approximately $k^{s-r}(-s-1)!/(-r-1)!$ by (5.83). So we need $r > s+1$ if the infinite series is going to converge. (If r and s are complex, the condition is $\Re r > \Re s + 1$, because $|k^z| = k^{\Re z}$.) The sum is

$$F\left(\begin{matrix}-r,\ 1\\ -s\end{matrix}\middle|\,1\right) = \frac{\Gamma(r-s-1)\Gamma(-s)}{\Gamma(r-s)\Gamma(-s-1)} = \frac{s+1}{s+1-r}$$

by (5.92); this is the same formula we found when r and s were integers.

5.91 (It's best to have computer help for this.) Incidentally, when $c = (a+1)/2$, this reduces to an identity that's equivalent to Gauss's identity (5.110), in view of Pfaff's reflection law. For if $w = -z/(1-z)$ we have $4w(1-w) = -4z/(1-z)^2$, and

$$F\left(\begin{matrix}\frac{1}{2}a,\ \frac{1}{2}a+\frac{1}{2}-b\\ 1+a-b\end{matrix}\middle|\,4w(1-w)\right) = F\left(\begin{matrix}a,\ a+1-2b\\ 1+a-b\end{matrix}\middle|\,\frac{-z}{1-z}\right) = (1-z)^a F\left(\begin{matrix}a,\ b\\ 1+a-b\end{matrix}\middle|\,z\right).$$

5.92 The identities can be proved, as Clausen proved them more than 150 years ago, by showing that both sides satisfy the same differential equation. One way to write the resulting equations between coefficients of z^n is in terms of binomial coefficients:

$$\sum_k \frac{\binom{r}{k}\binom{s}{k}\binom{r}{n-k}\binom{s}{n-k}}{\binom{r+s-1/2}{k}\binom{r+s-1/2}{n-k}} = \frac{\binom{2r}{n}\binom{r+s}{n}\binom{2s}{n}}{\binom{2r+2s}{n}\binom{r+s-1/2}{n}};$$

$$\sum_k \frac{\binom{-1/4+r}{k}\binom{-1/4+s}{k}\binom{-1/4-r}{n-k}\binom{-1/4-s}{n-k}}{\binom{-1+r+s}{k}\binom{-1-r-s}{n-k}}$$

$$= \frac{\binom{-1/2}{n}\binom{-1/2+r-s}{n}\binom{-1/2-r+s}{n}}{\binom{-1+r+s}{n}\binom{-1-r-s}{n}}.$$

Another way is in terms of hypergeometrics:

$$F\left(\begin{matrix} a, b, \frac{1}{2}-a-b-n, -n \\ \frac{1}{2}+a+b, 1-a-n, 1-b-n \end{matrix}\middle| 1\right) = \frac{(2a)^{\overline{n}}\,(a+b)^{\overline{n}}\,(2b)^{\overline{n}}}{(2a+2b)^{\overline{n}}\,a^{\overline{n}}\,b^{\overline{n}}};$$

$$F\left(\begin{matrix} \frac{1}{4}+a, \frac{1}{4}+b, a+b-n, -n \\ 1+a+b, \frac{3}{4}+a-n, \frac{3}{4}+b-n \end{matrix}\middle| 1\right)$$

$$= \frac{(1/2)^{\overline{n}}\,(1/2+a-b)^{\overline{n}}\,(1/2-a+b)^{\overline{n}}}{(1+a+b)^{\overline{n}}\,(1/4-a)^{\overline{n}}\,(1/4-b)^{\overline{n}}}.$$

The boxed sentence on the other side of this page is not self-referential.

5.93 $\alpha^{-1}\prod_{j=1}^{k}\big(f(j)+\alpha\big)/f(j)$.

5.94 Gosper's algorithm finds the answer $-\binom{a-1}{k-1}\binom{-a-1}{n-k}a/n + C$. Consequently, when $m \geqslant 0$ is an integer, we have

$$\sum \binom{a}{k}\binom{m-a}{n-k}\,\delta k = \sum_j \binom{m}{j}\frac{-a}{n-j}\binom{a-1}{k-1}\binom{-a-1}{n-j-k} + C.$$

5.95 The leading coefficients of p and r should be unity, and p should have no factors in common with q or r. It is easy to fulfill these additional conditions by shuffling factors around.

Now suppose $p(k+1)q(k)/p(k)r(k+1) = P(k+1)Q(k)/P(k)R(k+1)$, where the polynomials (p, q, r) and (P, Q, R) both satisfy the new criteria. Let

$p_0(k) = p(k)/g(k)$ and $P_0(k) = P(k)/g(k)$, where $g(k) = \gcd\bigl(p(k), P(k)\bigr)$ is the product of all common factors of p and P. Then

$$p_0(k+1)\,q(k)P_0(k)R(k+1) \;=\; p_0(k)r(k+1)P_0(k+1)Q(k)\,.$$

Suppose $p_0(k) \neq 1$. Then there is a complex number α such that $p_0(\alpha) = 0$; this implies $q(\alpha) \neq 0$, $r(\alpha) \neq 0$, and $P_0(\alpha) \neq 0$. Hence we must have $p_0(\alpha+1)R(\alpha+1) = 0$ and $p_0(\alpha-1)Q(\alpha-1) = 0$. Let N be a positive integer such that $p_0(\alpha+N) \neq 0$ and $p_0(\alpha-N) \neq 0$. Repeating the argument N times, we find $R(\alpha+1)\ldots R(\alpha+N) = 0 = Q(\alpha-1)\ldots Q(\alpha-N)$, contradicting (5.118). Therefore $p_0(k) = 1$. Similarly $P_0(k) = 1$, so $p(k) = P(k)$. Now $q(\alpha) = 0$ implies $r(\alpha+1) \neq 0$, by (5.118), hence $q(k)\backslash Q(k)$. Similarly $Q(k)\backslash q(k)$, so $q(k) = Q(k)$ since they have the same leading coefficient. That leaves $r(k) = R(k)$.

5.96 If $r(k)$ is a nonzero rational function and $T(k)$ is a hypergeometric term, then $r(k)T(k)$ is a hypergeometric term, which is called *similar* to $T(k)$. (We allow $r(k)$ to be ∞ and $T(k)$ to be 0, or vice versa, for finitely many values of k.) In particular, $T(k+1)$ is always similar to $T(k)$. If $T_1(k)$ and $T_2(k)$ are similar hypergeometric terms, then $T_1(k) + T_2(k)$ is a hypergeometric term. If $T_1(k), \ldots, T_m(k)$ are mutually dissimilar, and $m > 1$, then $T_1(k) + \cdots + T_m(k)$ cannot be zero for all but finitely many k. For if it could, consider a counterexample for which m is minimum, and let $r_j(k) = T_j(k+1)/T_j(k)$. Since $T_1(k) + \cdots + T_m(k) = 0$, we have $r_m(k)T_1(k) + \cdots + r_m(k)T_m(k) = 0$ and $r_1(k)T_1(k) + \cdots + r_m(k)T_m(k) = T_1(k+1) + \cdots + T_m(k+1) = 0$; hence $\bigl(r_m(k) - r_1(k)\bigr)T_1(k) + \cdots + \bigl(r_m(k) - r_{m-1}(k)\bigr)T_{m-1}(k) = 0$. We cannot have $r_m(k) - r_j(k) = 0$, for any $j < m$, since T_j and T_m are dissimilar. But m was minimum, so this cannot be a counterexample; it follows that $m = 2$. But then $T_1(k)$ and $T_2(k)$ must be similar, since they are both zero for all but finitely many k.

Burma-Shave

Now let $t(k)$ be any hypergeometric term with $t(k+1)/t(k) = r(k)$, and suppose that $t(k) = \bigl(T_1(k+1) + \cdots + T_m(k+1)\bigr) - \bigl(T_1(k) + \cdots + T_m(k)\bigr)$, where m is minimal. Then $T_1, \ldots, T_m$ must be mutually dissimilar. Let $r_j(k)$ be the rational function such that

$$r(k)\bigl(T_j(k+1) - T_j(k)\bigr) - \bigl(T_j(k+2) - T_j(k+1)\bigr) \;=\; r_j(k)T_j(k)\,.$$

Suppose $m > 1$. Since $0 = r(k)t(k) - t(k+1) = r_1(k)T_1(k) + \cdots + r_m(k)T_m(k)$, we must have $r_j(k) = 0$ for all but at most one value of j. If $r_j(k) = 0$, the function $\bar{t}(k) = T_j(k+1) - T_j(k)$ satisfies $\bar{t}(k+1)/\bar{t}(k) = t(k+1)/t(k)$. So Gosper's algorithm will find a solution.

5.97 Suppose first that z is not equal to $-d - 1/d$ for any integer $d > 0$. Then in Gosper's algorithm we have $p(k) = 1$, $q(k) = (k+1)^2$, $r(k) =$

$k^2 + kz + 1$. Since $\deg(Q) < \deg(R)$ and $\deg(p) - \deg(R) + 1 = -1$, the only possibility is $z = d + 2$ where d is a nonnegative integer. Trying $s(k) = \alpha_d k^d + \cdots + \alpha_0$ fails when $d = 0$ but succeeds whenever $d > 0$. (The linear equations obtained by equating coefficients of k^d, k^{d-1}, ..., k^1 in (5.122) express $\alpha_{d-1}, \ldots, \alpha_0$ as positive multiples of α_d, and the remaining equation $1 = \alpha_d + \cdots + \alpha_1$ then defines α_d.) For example, when $z = 3$ the indefinite sum is $(k+2)k!^2 / \prod_{j=1}^{k-1}(j^2 + 3j + 1) + C$.

If $z = -d - 1/d$, on the other hand, the stated terms $t(k)$ are infinite for $k \geqslant d$. There are two reasonable ways to proceed: We can cancel the zero in the denominator by redefining

$$t(k) \;=\; \frac{k!^2}{\prod_{j=d+1}^{k}\bigl(j^2 - j(d+1/d) + 1\bigr)} \;=\; \frac{(d-1/d)!\,k!^2}{(k-1/d)!\,(k-d)!}\,,$$

thereby making $t(k) = 0$ for $0 \leqslant k < d$ and positive for $k \geqslant d$. Then Gosper's algorithm gives $p(k) = k^{\underline{d}}$, $q(k) = k + 1$, $r(k) = k - 1/d$, and we can solve (5.122) for $s(k)$ because the coefficient of k^j on the right is $(j + 1 + 1/d)\alpha^j$ plus multiples of $\{\alpha_{j+1}, \ldots, \alpha_d\}$. For example, when $d = 2$ the indefinite sum is $(3/2)!\,k!\,(\frac{2}{7}k^2 - \frac{26}{35}k + \frac{32}{105})/(k - 3/2)! + C$.

Alternatively, we can try to sum the original terms, but only in the range $0 \leqslant k < d$. Then we can replace $p(k) = k^{\underline{d}}$ by

Look, any finite sequence is trivially summable, because we can find a polynomial that matches $t(k)$ for $0 \leqslant k < d$.

$$p'(k) \;=\; \sum_{j=1}^{d} (-1)^{d-j} j \begin{bmatrix} d \\ j \end{bmatrix} k^{j-1}\,.$$

This is justified since (5.117) still holds for $0 \leqslant k < d - 1$; we have $p'(k) = \lim_{\epsilon\to 0}\bigl((k+\epsilon)^{\underline{d}} - k^{\underline{d}}\bigr)/\epsilon = \lim_{\epsilon\to 0}(k+\epsilon)^{\underline{d}}/\epsilon$, so this trick essentially cancels a 0 from the numerator and denominator of (5.117) as in L'Hospital's rule. Gosper's method now yields an indefinite sum.

5.98 $nS_{n+1} = 2nS_n$. (Beware: This gives no information about S_1/S_0.)

5.99 Let $p(n,k) = (n+1+k)\beta_0(n) + (n+1+a+b+c+k)\beta_1(n) = \hat{p}(n,k)$, $\bar{t}(n,k) = t(n,k)/(n+1+k)$, $q(n,k) = (n+1+a+b+c+k)(a-k)(b-k)$, $r(n,k) = (n+1+k)(c+k)k$. Then (5.129) is solved by $\beta_0(n) = (n+1+a+b+c)(n+1+a+b)$, $\beta_1(n) = -(n+1+a)(n+1+b)$, $\alpha_0(n) = s(n,k) = -1$. We discover (5.134) by observing that it is true when $n = -a$ and using induction on n.

5.100 The Gosper-Zeilberger algorithm discovers easily that

$$\frac{n+2}{\binom{n}{k}} - \frac{2n+2}{\binom{n+1}{k}} \;=\; \frac{n-k}{\binom{n}{k+1}} - \frac{n+1-k}{\binom{n}{k}}\,, \qquad 0 \leqslant k < n.$$

Summing from $k=0$ to $n-1$ yields $(n+2)(S_n-1)-(2n+2)(S_{n+1}-1-\frac{1}{n+1})=-n$. Hence $(2n+2)S_{n+1}=(n+2)S_n+2n+2$. Applying a summation factor now leads to the expression $S_n=(n+1)2^{-n-1}\sum_{k=1}^{n+1}2^k/k$.

5.101 (a) If we hold m fixed, the Gosper-Zeilberger algorithm discovers that $(n+2)S_{m,n+2}(z)=(z-1)(n+1)S_{m,n}(z)+(2n+3-z(n-m+1))S_{m,n+1}(z)$. We can also apply the method to the term

$$\beta_0(m,n)t(m,n,k)+\beta_1(m,n)t(m+1,n,k)+\beta_2(m,n)t(m,n+1,k)\,,$$

in which case we get a simpler recurrence,

$$(m+1)S_{m+1,n}(z)-(n+1)S_{m,n+1}(z) \;=\; (1-z)(m-n)S_{m,n}(z)\,.$$

(b) Now we must work a little harder, with five equations in six unknowns. The algorithm finds

$$\begin{aligned}(n+1)(z-1)^2\binom{n}{k}^2 z^k-(2n+3)(z+1)\binom{n+1}{k}^2 z^k\\ +(n+2)\binom{n+2}{k}^2 z^k \;=\; T(n,k+1)-T(n,k)\,,\\ T(n,k) \;=\; \binom{n+1}{k-1}^2\frac{s(n,k)}{n+1}z^k\,,\\ s(n,k) \;=\; (z-1)k^2-2((n+2)z-2n-3)k+(n+2)((n+2)z-4n-5)\,.\end{aligned}$$

Therefore $(n+1)(z-1)^2S_n(z)-(2n+3)(z+1)S_{n+1}(z)+(n+2)S_{n+2}(z)=0$. Incidentally, this recurrence holds also for negative n, and we have $S_{-n-1}(z)=S_n(z)/(1-z)^{2n+1}$.

The sum $S_n(z)$ can be regarded as a modified form of the Legendre polynomial $P_n(z)=\sum_k\binom{n}{k}^2(z-1)^{n-k}(z+1)^k/2^n$, since we can write $S_n(z)=(1-z)^nP_n\big(\frac{1+z}{1-z}\big)$. Similarly, $S_{m,n}(z)=(1-z)^nP_n^{(0,m-n)}\big(\frac{1+z}{1-z}\big)$ is a modified Jacobi polynomial.

How about $z=0$?

5.102 The sum is $F(a-\frac{1}{3}n,-n;b-\frac{4}{3}n;-z)$, so we need not consider the case $z=-1$. Let $n=3m$. We seek solutions to (5.129) when

$$\begin{aligned}p(m,k) &= (3m+3-k)^{\underline{3}}(m+1-k)\beta_0+(4m+4-b-k)^{\underline{4}}\beta_1\,,\\ q(m,k) &= (3m+3-k)(m+1-a-k)z\,,\\ r(m,k) &= k(4m+1-b-k)\,,\\ s(m,k) &= \alpha_2k^2+\alpha_1k+\alpha_0\,.\end{aligned}$$

The resulting five homogeneous equations have a nonzero solution $(\alpha_0,\alpha_1,\alpha_2,\beta_0,\beta_1)$ if and only if the determinant of coefficients is zero; and this determinant, a polynomial in m, vanishes only in eight cases. One of those cases

is, of course, (5.113); but we can now evaluate the sum for all nonnegative integers n, not just $n \not\equiv 2 \pmod 3$:

$$\sum_k \binom{n}{k}\binom{\frac{1}{3}n-\frac{1}{6}}{k} 8^k \Big/ \binom{\frac{4}{3}n-\frac{2}{3}}{k} \;=\; [1,1,-\tfrac{1}{2}]\binom{2n}{n}\Big/\binom{\frac{4}{3}n-\frac{2}{3}}{n}.$$

Here the notation $[c_0, c_1, c_2]$ stands for the single value $c_{n \bmod 3}$. Another case, $(a, b, z) = (\frac{1}{2}, 0, 8)$, yields the identity

$$\sum_k \binom{n}{k}\binom{\frac{1}{3}n-\frac{1}{2}}{k} 8^k \Big/ \binom{\frac{4}{3}n}{k} \;=\; [1,0,0]\,16^{n/3}\binom{\frac{2}{3}n}{\frac{1}{3}n}\Big/\binom{\frac{4}{3}n}{n}.$$

(This sum, amazingly, is zero unless n is a multiple of 3; and then the identity can be written

$$\sum_k \binom{3m}{k}\binom{2m}{2k}\binom{2k}{k} 2^k \Big/ \binom{4m}{k}\binom{m}{k} \;=\; 16^m \frac{(3m)!\,(2m)!}{(4m)!\,m!},$$

which might even be useful.) The remaining six cases generate even weirder sums

$$\sum_k \binom{n}{k}\binom{\frac{1}{3}n-a}{k} z^k \Big/ \binom{\frac{4}{3}n-b}{k}$$
$$= [c_0, c_1, c_2]\frac{\binom{\frac{1}{3}n-a}{\lfloor n/3\rfloor}\binom{\frac{1}{3}n-a'}{\lfloor n/3\rfloor} x^{\lfloor n/3\rfloor}}{\binom{\frac{4}{3}n-b}{n}\binom{\frac{1}{3}n-b}{\lfloor n/3\rfloor}\binom{\frac{1}{3}n-b'}{\lfloor n/3\rfloor}}$$

where the respective values of $(a, b, z, c_0, c_1, c_2, a', b', x)$ are

$$\begin{array}{ll}
(\tfrac{7}{12}, \tfrac{1}{3}, \;\; 8, 1, -1, \;\; 0, \tfrac{1}{4}, 0, \;\; 64); & (\tfrac{1}{4}, 0, \;\; 8, 1, 2, \;\; 0, \tfrac{7}{12}, \tfrac{1}{3}, \;\; 64); \\
(\tfrac{5}{12}, \tfrac{2}{3}, \;\; 8, 1, \;\; 0, -3, \tfrac{3}{4}, 0, \;\; 64); & (\tfrac{1}{12}, \tfrac{1}{3}, \;\; 8, 1, 3, \;\; 0, \tfrac{3}{4}, 0, \;\; 64); \\
(\tfrac{1}{2}, 0, -4, 1, \;\; 2, \;\; 0, \tfrac{1}{6}, \tfrac{1}{3}, -16); & (\tfrac{1}{6}, \tfrac{2}{3}, -4, 1, 0, -3, \;\; \tfrac{5}{6}, 0, -16).
\end{array}$$

5.103 We assume that each a_i' and b_i' is nonzero, since the corresponding factors would otherwise have no influence on the degrees in k. Let $\hat{t}(n,k) = \hat{p}(n,k)\bar{t}(n,k)$ where

$$\bar{t}(n,k) \;=\; \frac{\prod_{i=1}^{p}(a_i n + a_i' k + a_i l[a_i<0] + a_i'')!}{\prod_{i=1}^{q}(b_i n + b_i' k + b_i l[b_i>0] + b_i'')!} z^k.$$

Then we have $\deg(\hat{p}) = \deg(f) + \max\bigl(\sum_{i=1}^{q} b_i[b_i>0] - \sum_{i=1}^{p} a_i[a_i<0], \sum_{i=1}^{p} a_i[a_i>0] - \sum_{i=1}^{q} b_i[b_i<0]\bigr) \geqslant \deg(f) + \frac{1}{2}l\bigl(|a_1| + \cdots + |a_p| + |b_1| +$

$\cdots + |b_q|\big)$, except in unusual cases where cancellation occurs in the leading coefficient. And $\deg(q) = \sum_{i=1}^{p} a_i'[a_i' > 0] - \sum_{i=1}^{q} b_i'[b_i' < 0]$, $\deg(r) = \sum_{i=1}^{q} b_i'[b_i' > 0] - \sum_{i=1}^{p} a_i'[a_i' < 0]$, again except in unusual cases.

(These estimates can be used to show directly that, as l increases, the degree of $\hat{p}$ eventually becomes large enough to make a polynomial $s(n,k)$ possible, and the number of unknown α_j and β_j eventually becomes larger than the number of homogeneous linear equations to be solved. So we obtain another proof that the Gosper-Zeilberger algorithm succeeds, if we argue as in the text that there must be a solution with $\beta_0(n), \ldots, \beta_l(n)$ not all zero.)

5.104 Let $t(n,k) = (-1)^k (r-s-k)!\,(r-2k)!/\big((r-s-2k)!\,(r-n-k+1)!\,(n-k)!\,k!\big)$. Then $\beta_0(n)t(n,k) + \beta_1(n)t(n+1,k)$ is not summable in hypergeometric terms, because $\deg(\hat{p}) = 1$, $\deg(q-r) = 3$, $\deg(q+r) = 4$, $\lambda = -8$, $\lambda' = -4$; but $\beta_0(n)t(n,k) + \beta_1(n)t(n+1,k) + \beta_2(n)t(n+2,k)$ is — basically because $\lambda' = 0$ when $q(n,k) = -(r-s-2k)(r-s-2k-1)(n+2-k)(r-n-k+1)$ and $r(k) = (r-s-k+1)(r-2k+2)(r-2k+1)k$. The solution is

$$\begin{aligned}
\beta_0(n) &= (s-n)(r-n+1)(r-2n+1)\,,\\
\beta_1(n) &= (rs - s^2 - 2rn + 2n^2 - 2r + 2n)(r-2n-1)\,,\\
\beta_2(n) &= (s-r+n+1)(n+2)(r-2n-3)\,,\\
\alpha_0(n) &= r-2n-1\,,
\end{aligned}$$

and we may conclude that $\beta_0(n)S_n + \beta_1(n)S_{n+1} + \beta_2(n)S_{n+2} = 0$ when S_n denotes the stated sum. This suffices to prove the identity by induction, after verifying the cases $n = 0$ and $n = 1$.

But S_n also satisfies the simpler recurrence $\bar{\beta}_0(n)S_n + \bar{\beta}_1(n)S_{n+1} = 0$, where $\bar{\beta}_0(n) = (s-n)(r-2n+1)$ and $\bar{\beta}_1(n) = -(n+1)(r-2n-1)$. Why didn't the method discover this? Well, nobody ever said that such a recurrence necessarily forces the terms $\bar{\beta}_0(n)t(n,k) + \bar{\beta}_1(n)t(n+1,k)$ to be indefinitely summable. The surprising thing is that the Gosper-Zeilberger method actually does find the simplest recurrence in so many other cases.

Notice that the second-order recurrence we found can be factored:

$$\begin{aligned}
&\beta_0(n) + \beta_1(n)N + \beta_2(n)N^2\\
&\qquad = \big((r-n+1)N + (r-s-n-1)\big)\ \big(\bar{\beta}_0(n) + \bar{\beta}_1(n)N\big)\,,
\end{aligned}$$

where N is the shift operator in (5.145).

5.105 Set $a = 1$ and compare the coefficients of z^{3n} on both sides of Henrici's "friendly monster" identity,

$$\begin{aligned}
&f(a,z)\,f(a,\omega z)\,f(a,\omega^2 z)\\
&\quad = F\left(\begin{matrix} \frac{1}{2}a-\frac{1}{4},\ \frac{1}{2}a+\frac{1}{4} \\ \frac{1}{3}a,\ \frac{1}{3}a+\frac{1}{3},\ \frac{1}{3}a+\frac{2}{3},\ \frac{2}{3}a-\frac{1}{3},\ \frac{2}{3}a,\ \frac{2}{3}a+\frac{1}{3},\ a \end{matrix}\ \middle|\ \left(\frac{4z}{9}\right)^3\right),
\end{aligned}$$

where $f(a,z) = F(1;a,1;z)$. The identity can be proved by showing that both sides satisfy the same differential equation.

Peter Paule has found another interesting way to evaluate the sum:

$$\begin{aligned}
\sum_{k,l}\binom{N}{k,l,N-k-l}^2\omega^{k+2l} &= \sum_{k,l}\binom{N}{k-l,l,N-k}^2\omega^{k+l}\\
&= \sum_{k,l}\binom{N}{k}^2\binom{k}{l}^2\omega^{k+l}\\
&= \sum_{k}\binom{N}{k}^2\omega^k\,[z^k]\,\big((1+z)(\omega+z)\big)^k\\
&= [z^0]\sum_{k}\binom{N}{k}^2\left(\frac{\omega(1+z)(\omega+z)}{z}\right)^k\\
&= [z^0]\sum_{k,j}\binom{N}{k}^2\binom{k}{j}\left(\frac{\omega(1+z)(\omega+z)}{z}-1\right)^j\\
&= [z^0]\sum_{k,j}\binom{N}{k}\binom{N-j}{N-k}\binom{N}{j}\left(\frac{(\omega z-1)^2}{\omega z}\right)^j\\
&= \sum_{j}\binom{2N-j}{N}\binom{N}{j}\,[z^j]\,(z-1)^{2j}\\
&= \sum_{j}\binom{2N-j}{N}\binom{N}{j}\binom{2j}{j}(-1)^j\,,
\end{aligned}$$

using the binomial theorem, Vandermonde's convolution, and the fact that $[z^0]g(az) = [z^0]g(z)$. We can now set $N = 3n$ and apply the Gosper-Zeilberger algorithm to this sum S_n, miraculously obtaining the first-order recurrence $(n+1)^2S_{n+1} = 4(4n+1)(4n+3)S_n$; the result follows by induction.

If $3n$ is replaced by $3n+1$ or $3n+2$, the stated sum is zero. Indeed, $\sum_{k+l+m=N} t(k,l,m)\omega^{l-m}$ is always zero when $N \bmod 3 \neq 0$ and $t(k,l,m) = t(l,m,k)$.

5.106 (Solution by Shalosh B Ekhad.) Let

$$\begin{aligned}
T(r,j,k) &= \frac{((1{+}n{+}s)(1{+}r)-(1{+}n{+}r)j+(s{-}r)k)(j{-}l)j}{(l-m+n-r+s)(n+r+1)(j-r-1)(j+k)}t(r,j,k)\,;\\
U(r,j,k) &= \frac{(s+n+1)(k+l)k}{(l-m+n-r+s)(n+r+1)(j+k)}t(r,j,k)\,.
\end{aligned}$$

The stated equality is routinely verifiable, and (5.32) follows by summing with respect to j and k. (We sum $T(r,j+1,k)-T(r,j,k)$ first with respect to j, then with respect to k; we sum the other terms $U(r,j,k+1)-U(r,j,k)$ first with respect to k, then with respect to j.)

Well, we also need to verify (5.32) when $r = 0$. In that case it reduces via trinomial revision to $\sum_k (-1)^k \binom{n}{n+l}\binom{n+l}{k+l}\binom{s+n-k}{m} = (-1)^l \binom{n}{n+l}\binom{s}{m-n-l}$. We are assuming that l, m, and n are integers and $n \geqslant 0$. Both sides are clearly zero unless $n + l \geqslant 0$. Otherwise we can replace k by $n - k$ and use (5.24).

Noticee that $1/nk$ *is proper, since it's* $(n-1)!(k-1)!/n!\,k!$. *Also* $1/(n^2 - k^2)$ *is proper. But* $1/(n^2 + k^2)$ *isn't.*

5.107 If it were proper, there would be a linear difference operator that annihilates it. In other words, we would have a finite summation identity

$$\sum_{i=0}^{I}\sum_{j=0}^{J} \alpha_{i,j}(n)/\big((n+i)(k+j)+1\big) \;=\; 0\,,$$

where the α's are polynomials in n, not all zero. Choose integers i, j, and n such that $n > 1$ and $\alpha_{i,j}(n) \neq 0$. Then when $k = -1/(n+i) - j$, the (i,j) term in the sum is infinite but the other terms are finite.

5.108 Replace k by $m - k$ in the double sum, then use (5.28) to sum on k, getting

$$A_{m,n} \;=\; \sum_j \binom{m}{j}^2 \binom{m+n-j}{m}^2;$$

trinomial revision (5.21) then yields one of the desired formulas.

It appears to be difficult to find a direct proof that the two symmetrical sums for $A_{m,n}$ are equal. We can, however, prove the equation indirectly with the Gosper-Zeilberger algorithm, by showing that both sums satisfy the recurrence

$$(n+1)^3 A_{m,n} - f(m,n)A_{m,n+1} + (n+2)^3 A_{m,n+2} \;=\; 0\,,$$

where $f(m,n) = (2n+3)(n^2+3n+2m^2+2m+3)$. Setting $t_1(n,k) = \binom{m}{k}\binom{n}{k}\binom{m+k}{k}\binom{n+k}{k}$ and $t_2(n,k) = \binom{m+n-k}{k}^2\binom{m+n-2k}{m-k}^2$, we find

$$\begin{aligned}(n+1)^2 t_j(n,k) - f(m,n)t_j(n+1,k) + (n+2)^2 t_j(n+2,k)\\ \;=\; T_j(n,k+1) - T_j(n,k)\,,\end{aligned}$$

where $T_1(n,k) = -2(2n+3)k^4 t_1(n,k)/(n+1-k)(n+2-k)$ and $T_2(n,k) = -\big((n+2)(4mn+n+3m^2+8m+2) - 2(3mn+n+m^2+6m+2)k + (2m+1)k^2\big)k^2(m+n+1-k)^2 t_2(n,k)/(n+2-k)^2$. This proves the recurrence, so we need only verify equality when $n = 0$ and $n = 1$. (We could also have used the simpler recurrence

$$m^3 A_{m,n-1} - n^3 A_{m-1,n} \;=\; (m-n)(m^2+n^2-mn)A_{m-1,n-1}\,,$$

which can be discovered by the method of exercise 101.)

The fact that the first formula for $A_{m,n}$ equals the third implies a remarkable identity between the generating functions $\sum_{m,n} A_{m,n} w^m z^n$:

$$\sum_k \frac{w^k S_k(z)^2}{(1-z)^{2k+1}} = \sum_k \binom{2k}{k}^2 \frac{w^k}{(1-w)^{2k+1}} \frac{z^k}{(1-z)^{2k+1}},$$

where $S_k(z) = \sum_j \binom{k}{j}^2 z^j$. It turns out, in fact, that

$$\sum_k \frac{w^k S_k(x) S_k(y)}{(1-x)^k (1-y)^k} = \sum_k \binom{2k}{k} \frac{w^k}{(1-w)^{2k+1}} \frac{\sum_j \binom{k}{j}^2 x^j y^{k-j}}{(1-x)^k (1-y)^k};$$

this is a special case of an identity discovered by Bailey [19].

5.109 Let $X_n = \sum_k \binom{n}{k}^{a_0} \binom{n+k}{k}^{a_1} \ldots \binom{n+lk}{k}^{a_l} x^k$ for any positive integers $a_0, a_1, \ldots, a_l$, and any integer x. Then if $0 \leqslant m < p$ we have

$$X_{m+pn} = \sum_{j=0}^{p-1} \sum_k \binom{m+pn}{j+pk}^{a_0} \cdots \binom{m+pn+l(j+pk)}{j+pk}^{a_l} x^{j+pk},$$

$$X_m X_n = \sum_{j=0}^{p-1} \sum_k \binom{m}{j}^{a_0} \binom{n}{k}^{a_0} \cdots \binom{m+lj}{j}^{a_l} \binom{n+lk}{k}^{a_l} x^{j+k}.$$

And corresponding terms are congruent (mod p), because exercise 36 implies that they are multiples of p when $lj + m \geqslant p$, exercise 61 implies that the binomials are congruent when $lj + m < p$, and (4.48) implies that $x^p \equiv x$.

5.110 The congruence surely holds if $2n+1$ is prime. Steven Skiena has also found the example $n = 2953$, when $2n + 1 = 3 \cdot 11 \cdot 179$.

Ilan Vardi notes that the condition holds for $2n + 1 = p^2$, where p is prime, if and only if $2^{p-1} \bmod p^2 = 1$. This yields two more examples: $n = (1093^2-1)/2$; $n = (3511^2-1)/2$.

5.111 See [96] for partial results. The computer experiments were done by V. A. Vyssotsky.

5.112 If n is not a power of 2, $\binom{2n}{n}$ is a multiple of 4 because of exercise 36. Otherwise the stated phenomenon was verified for $n \leqslant 2^{22000}$ by A. Granville and O. Ramaré, who also sharpened a theorem of Sárközy [317] by showing that $\binom{2n}{n}$ is divisible by the square of a prime for all $n > 2^{22000}$. This established a long-standing conjecture that $\binom{2n}{n}$ is never squarefree when $n > 4$.

The analogous conjectures for cubes are that $\binom{2n}{n}$ is divisible by the cube of a prime for all $n > 1056$, and by either 2^3 or 3^3 for all $n > 2^{29} + 2^{23}$. This has been verified for all $n < 2^{10000}$. Paul Erdős conjectures that, in fact, $\max_p \epsilon_p\left(\binom{2n}{n}\right)$ tends to infinity as $n \to \infty$; this might be true even if we restrict p to the values 2 and 3.

5.113 The theorem about generating functions in exercise 7.20 may help resolve this conjecture.

5.114 Strehl [344] has shown that $c_n^{(2)} = \sum_k \binom{n}{k}^3 = \sum_k \binom{n}{k}^2 \binom{2k}{n}$ is a so-called Franel number [132], and that $c_n^{(3)} = \sum_k \binom{n}{k}^2 \binom{2k}{k}^2 \binom{2k}{n-k}$. In another direction, H. S. Wilf has shown that $c_n^{(m)}$ is an integer for all m when $n \leqslant 9$.

6.1 2314, 2431, 3241, 1342, 3124, 4132, 4213, 1423, 2143, 3412, 4321.

6.2 $\left\{ {n \atop k} \right\} m^{\underline{k}}$, because every such function partitions its domain into k nonempty subsets, and there are $m^{\underline{k}}$ ways to assign function values for each partition. (Summing over k gives a combinatorial proof of (6.10).)

6.3 Now $d_{k+1} \leqslant (\text{center of gravity}) - \epsilon = 1 - \epsilon + (d_1 + \cdots + d_k)/k$. This recurrence is like (6.55) but with $1 - \epsilon$ in place of 1; hence the optimum solution is $d_{k+1} = (1 - \epsilon)H_k$. This is unbounded as long as $\epsilon < 1$.

6.4 $H_{2n+1} - \frac{1}{2}H_n$. (Similarly $\sum_{k=1}^{2n} (-1)^{k-1}/k = H_{2n} - H_n$.)

6.5 $U_n(x, y)$ is equal to

$$x \sum_{k\geqslant 1} \binom{n}{k} (-1)^{k-1} k^{-1} (x+ky)^{n-1} + y \sum_{k\geqslant 1} \binom{n}{k} (-1)^{k-1} (x+ky)^{n-1} ,$$

and the first sum is

$$U_{n-1}(x, y) + \sum_{k\geqslant 1} \binom{n-1}{k-1} (-1)^{k-1} k^{-1} (x + ky)^{n-1} .$$

The remaining k^{-1} can be absorbed, and we have

$$\begin{aligned} \sum_{k\geqslant 1} \binom{n}{k} (-1)^{k-1} (x+ky)^{n-1} &= x^{n-1} + \sum_{k\geqslant 0} \binom{n}{k} (-1)^{k-1} (x+ky)^{n-1} \\ &= x^{n-1} . \end{aligned}$$

This proves (6.75). Let $R_n(x, y) = x^{-n} U_n(x, y)$; then $R_0(x, y) = 0$ and $R_n(x, y) = R_{n-1}(x, y) + 1/n + y/x$, hence $R_n(x, y) = H_n + ny/x$. (Incidentally, the original sum $U_n = U_n(n, -1)$ doesn't lead to a recurrence such as this; therefore the more general sum, which detaches x from its dependence on n, is easier to solve inductively than its special case. This is another instructive example where a strong induction hypothesis makes the difference between success and failure.)

The Fibonacci recurrence is additive, but the rabbits are multiplying.

6.6 Each pair of babies bb present at the end of a month becomes a pair of adults **aa** at the end of the next month; and each pair **aa** becomes an **aa** and a bb. Thus each bb behaves like a drone in the bee tree and each **aa** behaves like a queen, except that the bee tree goes backward in time while the

rabbits are going forward. There are F_{n+1} pairs of rabbits after n months; F_n of them are adults and F_{n-1} are babies. (This is the context in which Fibonacci originally introduced his numbers.)

6.7 (a) Set $k = 1 - n$ and apply (6.107). (b) Set $m = 1$ and $k = n - 1$ and apply (6.128).

6.8 $55 + 8 + 2$ becomes $89 + 13 + 3 = 105$; the true value is 104.607361.

That "true value" is the length of 65 international miles, but the international mile is actually only .999998 as big as a U.S. statute mile. There are exactly 6336 kilometers in 3937 U.S. statute miles; the Fibonacci method converts 3937 to 6370.

6.9 21. (We go from F_n to F_{n+2} when the units are squared. The true answer is about 20.72.)

6.10 The partial quotients a_0, a_1, a_2, ... are all equal to 1, because $\phi = 1 + 1/\phi$. (The Stern–Brocot representation is therefore RLRLRLRLRL... .)

6.11 $(-1)^{\overline{n}} = [n=0] - [n=1]$; see (6.11).

6.12 This is a consequence of (6.31) and its dual in Table 264.

6.13 The two formulas are equivalent, by exercise 12. We can use induction. Or we can observe that $z^n D^n$ applied to $f(z) = z^x$ gives $x^{\underline{n}} z^x$ while ϑ^n applied to the same function gives $x^n z^x$; therefore the sequence $\langle \vartheta^0, \vartheta^1, \vartheta^2, \ldots \rangle$ must relate to $\langle z^0 D^0, z^1 D^1, z^2 D^2, \ldots \rangle$ as $\langle x^0, x^1, x^2, \ldots \rangle$ relates to $\langle x^{\underline{0}}, x^{\underline{1}}, x^{\underline{2}}, \ldots \rangle$.

6.14 We have

$$x\binom{x+k}{n} = (k+1)\binom{x+k}{n+1} + (n-k)\binom{x+k+1}{n+1},$$

because $(n+1)x = (k+1)(x+k-n) + (n-k)(x+k+1)$. (It suffices to verify the latter identity when $k = 0$, $k = -1$, and $k = n$.)

6.15 Since $\Delta\big(\binom{x+k}{n}\big) = \binom{x+k}{n-1}$, we have the general formula

$$\sum_k \left\langle {n \atop k} \right\rangle \binom{x+k}{n-m} = \Delta^m(x^n) = \sum_j \binom{m}{j} (-1)^{m-j} (x+j)^n .$$

Set $x = 0$ and appeal to (6.19).

6.16 $A_{n,k} = \sum_{j \geqslant 0} a_j \left\{ {n-j \atop k} \right\}$; this sum is always finite.

6.17 (a) $\left| {n \atop k} \right| = \left[{n+1 \atop n+1-k} \right]$. (b) $\left| {n \atop k} \right| = n^{\underline{n-k}} = n!\,[n \geqslant k]/k!$. (c) $\left| {n \atop k} \right| = k! \left\{ {n \atop k} \right\}$.

6.18 This is equivalent to (6.3) or (6.8).

6.19 Use Table 272.

6.20 $\sum_{1 \leqslant j \leqslant k \leqslant n} 1/j^2 = \sum_{1 \leqslant j \leqslant n} (n+1-j)/j^2 = (n+1)H_n^{(2)} - H_n$.

6.21 The hinted number is a sum of fractions with odd denominators, so it has the form a/b with a and b odd. (Incidentally, Bertrand's postulate implies that b_n is also divisible by at least one odd prime, whenever $n > 2$.)

6.22 $\bigl|z/k(k+z)\bigr| \leqslant 2|z|/k^2$ when $k > 2|z|$, so the sum is well defined when the denominators are not zero. If $z = n$ we have $\sum_{k=1}^{m}\bigl(1/k - 1/(k+n)\bigr) = H_m - H_{m+n} + H_n$, which approaches H_n as $m \to \infty$. (The quantity $H_{z-1} - \gamma$ is often called the psi function $\psi(z)$.)

6.23 $z/(e^z+1) = z/(e^z-1) - 2z/(e^{2z}-1) = \sum_{n\geqslant 0}(1-2^n)B_n z^n/n!$.

6.24 When n is odd, $T_n(x)$ is a polynomial in x^2, hence its coefficients are multiplied by even numbers when we form the derivative and compute $T_{n+1}(x)$ by (6.95). (In fact we can prove more: The Bernoulli number B_{2n} always has 2 to the first power in its denominator, by exercise 54; hence $2^{2n-k} \backslash\backslash T_{2n+1} \iff 2^k \backslash\backslash (n+1)$. The odd positive integers $(n+1)T_{2n+1}/2^{2n}$ are called Genocchi numbers $\langle 1, 1, 3, 17, 155, 2073, \ldots\rangle$, after Genocchi [145].)

6.25 $100n - nH_n < 100(n-1) - (n-1)H_{n-1} \iff H_{n-1} > 99$. (The least such n is approximately $e^{99-\gamma}$, while he finishes at $N \approx e^{100-\gamma}$, about e times as long. So he is getting closer during the final 63% of his journey.)

6.26 Let $u(k) = H_{k-1}$ and $\Delta v(k) = 1/k$, so that $u(k) = v(k)$. Then we have $S_n - H_n^{(2)} = \sum_{k=1}^{n} H_{k-1}/k = H_{k-1}^2\big|_1^{n+1} - S_n = H_n^2 - S_n$.

6.27 Observe that when $m > n$ we have $\gcd(F_m, F_n) = \gcd(F_{m-n}, F_n)$ by (6.108). This yields a proof by induction.

6.28 (a) $Q_n = \alpha(L_n - F_n)/2 + \beta F_n$. (The solution can also be written $Q_n = \alpha F_{n-1} + \beta F_n$.) (b) $L_n = \phi^n + \hat{\phi}^n$.

6.29 When $k = 0$ the identity is (6.133). When $k = 1$ it is, essentially,

$$\begin{aligned} K(x_1, \ldots, x_n)x_m \;=\; & K(x_1, \ldots, x_m)\,K(x_m, \ldots, x_n) \\ & - K(x_1, \ldots, x_{m-2})\,K(x_{m+2}, \ldots, x_n)\,; \end{aligned}$$

in Morse code terms, the second product on the right subtracts out the cases where the first product has intersecting dashes. When $k > 1$, an induction on k suffices, using both (6.127) and (6.132). (The identity is also true when one or more of the subscripts on K become -1, if we adopt the convention that $K_{-1} = 0$. When multiplication is not commutative, Euler's identity remains valid if we write it in the form

$$\begin{aligned} & K_{m+n}(x_1, \ldots, x_{m+n})\,K_k(x_{m+k}, \ldots, x_{m+1}) \\ & \quad = K_{m+k}(x_1, \ldots, x_{m+k})\,K_n(x_{m+n}, \ldots, x_{m+1}) \\ & \qquad + (-1)^k K_{m-1}(x_1, \ldots, x_{m-1})\,K_{n-k-1}(x_{m+n}, \ldots, x_{m+k+2})\,. \end{aligned}$$

For example, we obtain the somewhat surprising noncommutative factorizations

$$(abc + a + c)(1 + ba) \;=\; (ab + 1)(cba + a + c)$$

from the case $k = 2$, $m = 0$, $n = 3$.)

6.30 The derivative of $K(x_1,\ldots,x_n)$ with respect to x_m is

$$K(x_1,\ldots,x_{m-1})\,K(x_{m+1},\ldots,x_n)\,,$$

and the second derivative is zero; hence the answer is

$$K(x_1,\ldots,x_n)+K(x_1,\ldots,x_{m-1})\,K(x_{m+1},\ldots,x_n)\,y\,.$$

6.31 Since $x^{\overline{n}} = (x+n-1)^{\underline{n}} = \sum_k \binom{n}{k} x^{\underline{k}}(n-1)^{\underline{n-k}}$, we have $\left|{n \atop k}\right| = \binom{n}{k}(n-1)^{\underline{n-k}}$. These coefficients, incidentally, satisfy the recurrence ${n \atop k} = {-k \atop -n}$.

$$\left|{n \atop k}\right| = (n-1+k)\left|{n-1 \atop k}\right| + \left|{n-1 \atop k-1}\right|, \qquad \text{integers } n,k>0.$$

6.32 $\sum_{k\leqslant m} k\left\{{n+k \atop k}\right\} = \left\{{m+n+1 \atop m}\right\}$ and $\sum_{0\leqslant k\leqslant n}\left\{{k \atop m}\right\}(m+1)^{n-k} = \left\{{n+1 \atop m+1}\right\}$, both of which appear in Table 265.

6.33 If $n>0$, we have $\left[{n \atop 3}\right] = \frac{1}{2}(n-1)!\,(H_{n-1}^2 - H_{n-1}^{(2)})$, by (6.71); $\left\{{n \atop 3}\right\} = \frac{1}{6}(3^n - 3\cdot 2^n + 3)$, by (6.19).

6.34 We have $\left\langle{-1 \atop k}\right\rangle = 1/(k+1)$, $\left\langle{-2 \atop k}\right\rangle = H_{k+1}^{(2)}$, and in general $\left\langle{n \atop k}\right\rangle$ is given by (6.38) for all integers n.

6.35 Let n be the least integer $> 1/\epsilon$ such that $\lfloor H_n\rfloor > \lfloor H_{n-1}\rfloor$.

6.36 Now $d_{k+1} = \bigl(100+(1+d_1)+\cdots+(1+d_k)\bigr)/(100+k)$, and the solution is $d_{k+1} = H_{k+100} - H_{101} + 1$ for $k \geqslant 1$. This exceeds 2 when $k \geqslant 176$.

6.37 The sum (by parts) is $H_{mn} - \bigl(\frac{m}{m} + \frac{m}{2m} + \cdots + \frac{m}{mn}\bigr) = H_{mn} - H_n$. The infinite sum is therefore $\ln m$. (It follows that

$$\sum_{k\geqslant 1} \frac{\nu_m(k)}{k(k+1)} = \frac{m}{m-1}\ln m\,,$$

because $\nu_m(k) = (m-1)\sum_{j\geqslant 1}(k \bmod m^j)/m^j$.)

6.38 $(-1)^k\bigl(\binom{r-1}{k}r^{-1} - \binom{r-1}{k-1}H_k\bigr) + C$. (By parts, using (5.16).)

6.39 Write it as $\sum_{1\leqslant j\leqslant n} j^{-1}\sum_{j\leqslant k\leqslant n} H_k$ and sum first on k via (6.67), to get

$$(n+1)H_n^2 - (2n+1)H_n + 2n\,.$$

6.40 If $6n-1$ is prime, the numerator of

$$\sum_{k=1}^{4n-1} \frac{(-1)^{k-1}}{k} = H_{4n-1} - H_{2n-1}$$

is divisible by $6n-1$, because the sum is

$$\sum_{k=2n}^{4n-1}\frac{1}{k} = \sum_{k=2n}^{3n-1}\left(\frac{1}{k}+\frac{1}{6n-1-k}\right) = \sum_{k=2n}^{3n-1}\frac{6n-1}{k(6n-1-k)}\,.$$

Similarly if $6n+1$ is prime, the numerator of $\sum_{k=1}^{4n}(-1)^{k-1}/k = H_{4n}-H_{2n}$ is a multiple of $6n+1$. For 1987 we sum up to $k=1324$.

6.41 $S_{n+1} = \sum_k \binom{\lfloor (n+1+k)/2\rfloor}{k} = \sum_k \binom{\lfloor (n+k)/2\rfloor}{k-1}$, hence we have $S_{n+1} + S_n = \sum_k \binom{\lfloor (n+k)/2+1\rfloor}{k} = S_{n+2}$. The answer is F_{n+2}.

6.42 F_n.

6.43 Set $z=\frac{1}{10}$ in $\sum_{n\geqslant 0}F_nz^n = z/(1-z-z^2)$ to get $\frac{10}{89}$. The sum is a repeating decimal with period length 44:

$$0.11235\,95505\,61797\,75280\,89887\,64044\,94382\,02247\,19101\,12359\,55+\,.$$

6.44 Replace (m,k) by $(-m,-k)$ or $(k,-m)$ or $(-k,m)$, if necessary, so that $m\geqslant k\geqslant 0$. The result is clear if $m=k$. If $m>k$, we can replace (m,k) by $(m-k,m)$ and use induction.

6.45 $X_n = A(n)\alpha+B(n)\beta+C(n)\gamma+D(n)\delta$, where $B(n)=F_n$, $A(n)=F_{n-1}$, $A(n)+B(n)-D(n)=1$, and $B(n)-C(n)+3D(n)=n$.

6.46 $\phi/2$ and $\phi^{-1}/2$. Let $u=\cos 72^\circ$ and $v=\cos 36^\circ$; then $u=2v^2-1$ and $v=1-2\sin^2 18^\circ = 1-2u^2$. Hence $u+v=2(u+v)(v-u)$, and $4v^2-2v-1=0$. We can pursue this investigation to find the five complex fifth roots of unity:

$$1\,,\quad \frac{\phi^{-1}\pm i\sqrt{2+\phi}}{2}\,,\quad \frac{-\phi\pm i\sqrt{3-\phi}}{2}\,.$$

"Let p be any old prime." (See [171], p. 419.)

6.47 $2^n\sqrt{5}\,F_n = (1+\sqrt{5})^n-(1-\sqrt{5})^n$, and the even powers of $\sqrt{5}$ cancel out. Now let p be an odd prime. Then $\binom{p}{2k+1}\equiv 0$ except when $k=(p-1)/2$, and $\binom{p+1}{2k+1}\equiv 0$ except when $k=0$ or $k=(p-1)/2$; hence $F_p\equiv 5^{(p-1)/2}$ and $2F_{p+1}\equiv 1+5^{(p-1)/2} \pmod p$. It can be shown that $5^{(p-1)/2}\equiv 1$ when p has the form $10k\pm 1$, and $5^{(p-1)/2}\equiv -1$ when p has the form $10k\pm 3$.

6.48 Let $K_{i,j} = K_{j-i+1}(x_i,\ldots,x_j)$. Using (6.133) repeatedly, both sides expand to $(K_{1,m-2}(x_{m-1}+x_{m+1})+K_{1,m-3})K_{m+2,n}+K_{1,m-2}K_{m+3,n}$.

6.49 Set $z=\frac{1}{2}$ in (6.146); the partial quotients are $0,\ 2^{F_0},\ 2^{F_1},\ 2^{F_2},\ \ldots$. (Knuth [206] noted that this number is transcendental.)

6.50 (a) $f(n)$ is even $\iff 3\backslash n$. (b) If the binary representation of n is $(1^{a_1}0^{a_2}\ldots 1^{a_{m-1}}0^{a_m})_2$, where m is even, we have $f(n)=K(a_1,a_2,\ldots,a_{m-1})$.

6.51 (a) Combinatorial proof: The arrangements of $\{1, 2, \ldots, p\}$ into k subsets or cycles are divided into "orbits" of 1 or p arrangements each, if we add 1 to each element modulo p. For example,

$$\begin{aligned}&\{1,2,4\}\cup\{3,5\} \to \{2,3,5\}\cup\{4,1\} \to \{3,4,1\}\cup\{5,2\}\\ &\quad\to \{4,5,2\}\cup\{1,3\} \to \{5,1,3\}\cup\{2,4\} \to \{1,2,4\}\cup\{3,5\}.\end{aligned}$$

We get an orbit of size 1 only when this transformation takes an arrangement into itself; but then $k = 1$ or $k = p$. Alternatively, there's an algebraic proof: We have $x^p \equiv x^{\underline{p}} + x^{\underline{1}}$ and $x^{\underline{p}} \equiv x^p - x \pmod p$, since Fermat's theorem tells us that $x^p - x$ is divisible by $(x-0)(x-1)\ldots\bigl(x-(p-1)\bigr)$.

(b) This result follows from (a) and Wilson's theorem; or we can use $x^{\overline{p-1}} \equiv x^{\overline{p}}/(x-1) \equiv (x^p - x)/(x-1) = x^{p-1} + x^{p-2} + \cdots + x$.

(c) We have $\left\{{p+1 \atop k}\right\} \equiv \left[{p+1 \atop k}\right] \equiv 0$ for $3 \leqslant k \leqslant p$, then $\left\{{p+2 \atop k}\right\} \equiv \left[{p+2 \atop k}\right] \equiv 0$ for $4 \leqslant k \leqslant p$, etc. (Similarly, we have $\left[{2p-1 \atop p}\right] \equiv -\left\{{2p-1 \atop p}\right\} \equiv 1$.)

(d) $p! = p^{\underline{p}} = \sum_k (-1)^{p-k} p^k \left[{p \atop k}\right] = p^p \left[{p \atop p}\right] - p^{p-1}\left[{p \atop p-1}\right] + \cdots + p^3\left[{p \atop 3}\right] - p^2\left[{p \atop 2}\right] + p\left[{p \atop 1}\right]$. But $p\left[{p \atop 1}\right] = p!$, so

$$\begin{bmatrix}p\\2\end{bmatrix} = p\begin{bmatrix}p\\3\end{bmatrix} - p^2\begin{bmatrix}p\\4\end{bmatrix} + \cdots + p^{p-2}\begin{bmatrix}p\\p\end{bmatrix}$$

is a multiple of p^2. (This is called Wolstenholme's theorem.)

6.52 (a) Observe that $H_n = H_n^* + H_{\lfloor n/p\rfloor}/p$, where $H_n^* = \sum_{k=1}^n [k \perp p]/k$. (b) Working mod 5 we have $H_r = \langle 0, 1, 4, 1, 0\rangle$ for $0 \leqslant r \leqslant 4$. Thus the first solution is $n = 4$. By part (a) we know that $5 \backslash a_n \Longrightarrow 5 \backslash a_{\lfloor n/5\rfloor}$; so the next possible range is $n = 20 + r$, $0 \leqslant r \leqslant 4$, when we have $H_n = H_n^* + \frac{1}{5}H_4 = H_{20}^* + \frac{1}{5}H_4 + H_r + \sum_{k=1}^r 20/k(20+k)$. The numerator of H_{20}^*, like the numerator of H_4, is divisible by 25. Hence the only solutions in this range are $n = 20$ and $n = 24$. The next possible range is $n = 100 + r$; now $H_n = H_n^* + \frac{1}{5}H_{20}$, which is $\frac{1}{5}H_{20} + H_r$ plus a fraction whose numerator is a multiple of 5. If $\frac{1}{5}H_{20} \equiv m \pmod 5$, where m is an integer, the harmonic number H_{100+r} will have a numerator divisible by 5 if and only if $m + H_r \equiv 0 \pmod 5$; hence m must be $\equiv 0$, 1, or 4. Working modulo 5 we find $\frac{1}{5}H_{20} = \frac{1}{5}H_{20}^* + \frac{1}{25}H_4 \equiv \frac{1}{25}H_4 = \frac{1}{12} \equiv 3$; hence there are no solutions for $100 \leqslant n \leqslant 104$. Similarly there are none for $120 \leqslant n \leqslant 124$; we have found all three solutions.

(By exercise 6.51(d), we always have $p^2 \backslash a_{p-1}$, $p \backslash a_{p^2-p}$, and $p \backslash a_{p^2-1}$, if p is any prime $\geqslant 5$. The argument just given shows that these are the only solutions to $p \backslash a_n$ if and only if there are no solutions to $p^{-2}H_{p-1} + H_r \equiv 0 \pmod p$ for $0 \leqslant r < p$. The latter condition holds not only for $p = 5$ but also for $p = 13$, 17, 23, 41, and 67—perhaps for infinitely many primes. The numerator of H_n is divisible by 3 only when $n = 2$, 7, and 22; it is divisible

(Attention, computer programmers: Here's an interesting condition to test, for as many primes as you can.)

by 7 only when $n = 6$, 42, 48, 295, 299, 337, 341, 2096, 2390, 14675, 16731, 16735, and 102728.)

6.53 Summation by parts yields

$$\frac{n+1}{(n+2)^2}\left(\frac{(-1)^m}{\binom{n+1}{m+1}}\big((n+2)H_{m+1}-1\big)-1\right).$$

6.54 (a) If $m \geqslant p$ we have $S_m(p) \equiv S_{m-(p-1)}(p) \pmod p$, since $k^{p-1} \equiv 1$ when $1 \leqslant k < p$. Also $S_{p-1}(p) \equiv p-1 \equiv -1$. If $0 < m < p-1$, we can write

$$S_m(p) \;=\; \sum_{j=0}^{m}\begin{bmatrix} m \\ j \end{bmatrix}(-1)^{m-j}\sum_{k=0}^{p-1}k^{\underline{j}} \;=\; \sum_{j=0}^{m}\begin{bmatrix} m \\ j \end{bmatrix}(-1)^{m-j}\frac{p^{\underline{j+1}}}{j+1} \;\equiv\; 0\,.$$

(The numerators of Bernoulli numbers played an important role in early studies of Fermat's Last Theorem; see Ribenboim [308].)

(b) The condition in the hint implies that the denominator of I_{2n} is not divisible by any prime p; hence I_{2n} must be an integer. To prove the hint, we may assume that $n>1$. Then

$$B_{2n} + \frac{\big[(p-1)\backslash(2n)\big]}{p} + \sum_{k=0}^{2n-2}\binom{2n+1}{k}B_k\frac{p^{2n-k}}{2n+1}$$

is an integer, by (6.78), (6.84), and part (a). So we want to verify that none of the fractions $\binom{2n+1}{k}B_k p^{2n-k}/(2n+1) = \binom{2n}{k}B_k p^{2n-k}/(2n-k+1)$ has a denominator divisible by p. The denominator of $\binom{2n}{k}B_k p$ isn't divisible by p, since B_k has no p^2 in its denominator (by induction); and the denominator of $p^{2n-k-1}/(2n-k+1)$ isn't divisible by p, since $2n-k+1 < p^{2n-k}$ when $k \leqslant 2n-2$; QED. (The numbers I_{2n} are tabulated in [224]. Hermite calculated them through I_{18} in 1875 [184]. It turns out that $I_2 = I_4 = I_6 = I_8 = I_{10} = I_{12} = 1$; hence there *is* actually a "simple" pattern to the Bernoulli numbers displayed in the text, including $\frac{-691}{2730}$(!). But the numbers I_{2n} don't seem to have any memorable features when $2n > 12$. For example, $B_{24} = -86579 - \frac{1}{2} - \frac{1}{3} - \frac{1}{5} - \frac{1}{7} - \frac{1}{13}$, and 86579 is prime.)

(c) The numbers $2-1$ and $3-1$ always divide $2n$. If n is prime, the only divisors of $2n$ are 1, 2, n, and $2n$, so the denominator of B_{2n} for prime $n > 2$ will be 6 unless $2n+1$ is also prime. In the latter case we can try $4n+3$, $8n+7$, $\ldots$, until we eventually hit a nonprime (since n divides $2^{n-1}n + 2^{n-1} - 1$). (This proof does not need the more difficult, but true, theorem that there are infinitely many primes of the form $6k+1$.) The denominator of B_{2n} can be 6 also when n has nonprime values, such as 49.

6.55 The stated sum is $\frac{m+1}{x+m+1}\binom{x+n}{n}\binom{n}{m+1}$, by Vandermonde's convolution. To get (6.70), differentiate and set $x = 0$.

6.56 First replace k^{n+1} by $\big((k-m)+m\big)^{n+1}$ and expand in powers of $k-m$; simplifications occur as in the derivation of (6.72). If $m > n$ or $m < 0$, the answer is $(-1)^n n! - m^n/\binom{n-m}{n}$. Otherwise we need to take the limit of (5.41) minus the term for $k = m$, as $x \to -m$; the answer comes to $(-1)^n n! + (-1)^{m+1}\binom{n}{m} m^n (n + 1 + mH_{n-m} - mH_m)$.

6.57 First prove by induction that the nth row contains at most three distinct values $A_n \geqslant B_n \geqslant C_n$; if n is even they occur in the cyclic order $[C_n, B_n, A_n, B_n, C_n]$, while if n is odd they occur in the cyclic order $[C_n, B_n, A_n, A_n, B_n]$. Also

$$\begin{aligned} A_{2n+1} &= A_{2n} + B_{2n}\,; & A_{2n} &= 2A_{2n-1}\,; \\ B_{2n+1} &= B_{2n} + C_{2n}\,; & B_{2n} &= A_{2n-1} + B_{2n-1}\,; \\ C_{2n+1} &= 2C_{2n}\,; & C_{2n} &= B_{2n-1} + C_{2n-1}\,. \end{aligned}$$

It follows that $Q_n = A_n - C_n = F_{n+1}$. (See exercise 5.75 for wraparound binomial coefficients of order 3.)

6.58 (a) $\sum_{n\geqslant 0} F_n^2 z^n = z(1-z)/(1+z)(1-3z+z^2) = \frac{1}{5}\big((2-3z)/(1-3z+z^2) - 2/(1+z)\big)$. (Square Binet's formula (6.123) and sum on n, then combine terms so that ϕ and $\hat\phi$ disappear.) (b) Similarly,

$$\sum_{n\geqslant 0} F_n^3 z^n = \frac{z(1-2z-z^2)}{(1-4z-z^2)(1+z-z^2)} = \frac{1}{5}\left(\frac{2z}{1-4z-z^2} + \frac{3z}{1+z-z^2}\right).$$

It follows that $F_{n+1}^3 - 4F_n^3 - F_{n-1}^3 = 3(-1)^n F_n$. (The corresponding recurrence for mth powers involves the Fibonomial coefficients of exercise 86; it was discovered by Jarden and Motzkin [194].)

6.59 Let m be fixed. We can prove by induction on n that it is, in fact, possible to find such an x with the additional condition $x \not\equiv 2 \pmod 4$. If x is such a solution, we can move up to a solution modulo 3^{n+1} because

$$F_{8\cdot 3^{n-1}} \equiv 3^n\,, \qquad F_{8\cdot 3^{n-1}-1} \equiv 3^n + 1 \pmod{3^{n+1}}\,;$$

either x or $x + 8\cdot 3^{n-1}$ or $x + 16\cdot 3^{n-1}$ will do the job.

6.60 F_1+1, F_2+1, F_3+1, F_4-1, and F_6-1 are the only cases. Otherwise the Lucas numbers of exercise 28 arise in the factorizations

$$\begin{aligned} F_{2m} + (-1)^m &= L_{m+1} F_{m-1}\,; & F_{2m+1} + (-1)^m &= L_m F_{m+1}\,; \\ F_{2m} - (-1)^m &= L_{m-1} F_{m+1}\,; & F_{2m+1} - (-1)^m &= L_{m+1} F_m\,. \end{aligned}$$

(We have $F_{m+n} - (-1)^n F_{m-n} = L_m F_n$ in general.)

6.61 $1/F_{2m} = F_{m-1}/F_m - F_{2m-1}/F_{2m}$ when m is even and positive. The second sum is $5/4 - F_{3\cdot 2^n-1}/F_{3\cdot 2^n}$, for $n \geqslant 1$.

6.62 (a) $A_n = \sqrt{5}\,A_{n-1} - A_{n-2}$ and $B_n = \sqrt{5}\,B_{n-1} - B_{n-2}$. Incidentally, we also have $\sqrt{5}\,A_n + B_n = 2A_{n+1}$ and $\sqrt{5}\,B_n - A_n = 2B_{n-1}$. (b) A table of small values reveals that

$$A_n = \begin{cases} L_n, & n \text{ even;} \\ \sqrt{5}\,F_n, & n \text{ odd;} \end{cases} \qquad B_n = \begin{cases} \sqrt{5}\,F_n, & n \text{ even;} \\ L_n, & n \text{ odd.} \end{cases}$$

(c) $B_n/A_{n+1} - B_{n-1}/A_n = 1/(F_{2n+1}+1)$ because $B_nA_n - B_{n-1}A_{n+1} = \sqrt{5}$ and $A_nA_{n+1} = \sqrt{5}\,(F_{2n+1}+1)$. Notice that $B_n/A_{n+1} = (F_n/F_{n+1})[n \text{ even}] + (L_n/L_{n+1})[n \text{ odd}]$. (d) Similarly, $\sum_{k=1}^{n} 1/(F_{2k+1}-1) = (A_0/B_1 - A_1/B_2) + \cdots + (A_{n-1}/B_n - A_n/B_{n+1}) = 2 - A_n/B_{n+1}$. This quantity can also be expressed as $(5F_n/L_{n+1})[n \text{ even}] + (L_n/F_{n+1})[n \text{ odd}]$.

6.63 (a) $\left[{n \atop k}\right]$. There are $\left[{n-1 \atop k-1}\right]$ with $\pi_n = n$ and $(n-1)\left[{n-1 \atop k}\right]$ with $\pi_n < n$. (b) $\left\langle{n \atop k}\right\rangle$. Each permutation $\rho_1 \ldots \rho_{n-1}$ of $\{1, \ldots, n-1\}$ leads to n permutations $\pi_1\pi_2\ldots\pi_n = \rho_1 \ldots \rho_{j-1}\, n\, \rho_{j+1} \ldots \rho_{n-1}\rho_j$. If $\rho_1 \ldots \rho_{n-1}$ has k excedances, there are $k+1$ values of j that yield k excedances in $\pi_1\pi_2\ldots\pi_n$; the remaining $n-1-k$ values yield $k+1$. Hence the total number of ways to get k excedances in $\pi_1\pi_2\ldots\pi_n$ is $(k+1)\left\langle{n-1 \atop k}\right\rangle + \big((n-1)-(k-1)\big)\left\langle{n-1 \atop k-1}\right\rangle = \left\langle{n \atop k}\right\rangle$.

6.64 The denominator of $\binom{1/2}{2n}$ is $2^{4n-\nu_2(n)}$, by the proof in exercise 5.72. The denominator of $\left[{1/2 \atop 1/2-n}\right]$ is the same, by (6.44), because $\left\langle\!\left\langle{n \atop 0}\right\rangle\!\right\rangle = 1$ and $\left\langle\!\left\langle{n \atop k}\right\rangle\!\right\rangle$ is even for $k > 0$.

6.65 This is equivalent to saying that $\left\langle{n \atop k}\right\rangle/n!$ is the probability that we have $\lfloor x_1 + \cdots + x_n \rfloor = k$, when $x_1, \ldots, x_n$ are independent random numbers uniformly distributed between 0 and 1. Let $y_j = (x_1 + \cdots + x_j) \bmod 1$. Then $y_1, \ldots, y_n$ are independently and uniformly distributed, and $\lfloor x_1 + \cdots + x_n \rfloor$ is the number of descents in the y's. The permutation of the y's is random, and the probability of k descents is the same as the probability of k ascents.

6.66 $2^{n+1}(2^{n+1}-1)B_{n+1}/(n+1)$, if $n > 0$. (See (7.56) and (6.92); the desired numbers are essentially the coefficients of $1 - \tanh z$.)

6.67 It is $\sum_k \left(\left\{{n \atop k+1}\right\}(k+1)! + \left\{{n \atop k}\right\}k!\right)\binom{n-k}{n-m}(-1)^{m-k} = \sum_k \left\{{n \atop k}\right\}k!(-1)^{m-k} \times \left(\binom{n-k}{n-m} - \binom{n+1-k}{n-m}\right) = \sum_k \left\{{n \atop k}\right\}k!(-1)^{m+1-k}\binom{n-k}{n-m-1} = \left\langle{n \atop n-m-1}\right\rangle$ by (6.3) and (6.40). Now use (6.34). (This identity has a combinatorial interpretation [59].)

6.68 We have the general formula

$$\left\langle\!\left\langle{n \atop m}\right\rangle\!\right\rangle = \sum_{k=0}^{m} \binom{2n+1}{k} \left\{{n+m+1-k \atop m+1-k}\right\}(-1)^k, \qquad \text{for } n > m \geqslant 0,$$

analogous to (6.38). When $m = 2$ this equals

$$\left\langle\!\left\langle {n \atop 2} \right\rangle\!\right\rangle = \left\{ {n+3 \atop 3} \right\} - (2n+1)\left\{ {n+2 \atop 2} \right\} + \binom{2n+1}{2}\left\{ {n+1 \atop 1} \right\}$$
$$= \tfrac{1}{2}3^{n+2} - (2n+3)2^{n+1} + \tfrac{1}{2}(4n^2+6n+3)\,.$$

6.69 $\frac{1}{3}n(n+\frac{1}{2})(n+1)(2H_{2n}-H_n) - \frac{1}{36}n(10n^2+9n-1)$. (It would be nice to automate the derivation of formulas such as this.)

6.70 $1/k - 1/(k+z) = z/k^2 - z^2/k^3 + \cdots$, which converges when $|z| < 1$.

6.71 Note that $\prod_{k=1}^{n}(1+z/k)e^{-z/k} = \binom{n+z}{n}n^{-z}e^{(\ln n - H_n)z}$. If $f(z) = \frac{d}{dz}(z!)$ we find $f(z)/z! + \gamma = H_z$.

6.72 For $\tan z$, we can use $\tan z = \cot z - 2\cot 2z$ (which is equivalent to the identity of exercise 23). Also $z/\sin z = z\cot z + z\tan\frac{1}{2}z$ has the power series $\sum_{n\geqslant 0}(-1)^{n-1}(4^n-2)B_{2n}z^{2n}/(2n)!$; and

$$\begin{aligned}
\ln\frac{\tan z}{z} &= \ln\frac{\sin z}{z} - \ln\cos z \\
&= \sum_{n\geqslant 1}(-1)^n\frac{4^nB_{2n}z^{2n}}{(2n)(2n)!} - \sum_{n\geqslant 1}(-1)^n\frac{4^n(4^n-1)B_{2n}z^{2n}}{(2n)(2n)!} \\
&= \sum_{n\geqslant 1}(-1)^{n-1}\,\frac{4^n(4^n-2)B_{2n}z^{2n}}{(2n)(2n)!}\,,
\end{aligned}$$

because $\frac{d}{dz}\ln\sin z = \cot z$ and $\frac{d}{dz}\ln\cos z = -\tan z$.

6.73 $\cot(z+\pi) = \cot z$ and $\cot(z+\frac{1}{2}\pi) = -\tan z$; hence the identity is equivalent to

$$\cot z = \frac{1}{2^n}\sum_{k=0}^{2^n-1}\cot\frac{z+k\pi}{2^n}\,,$$

which follows by induction from the case $n = 1$. The stated limit follows since $z\cot z \to 1$ as $z \to 0$. It can be shown that term-by-term passage to the limit is justified, hence (6.88) is valid. (Incidentally, the general formula

$$\cot z = \frac{1}{n}\sum_{k=0}^{n-1}\cot\frac{z+k\pi}{n}$$

is also true. It can be proved from (6.88), or from

$$\frac{1}{e^{nz}-1} = \frac{1}{n}\sum_{k=0}^{n-1}\frac{1}{e^{z+2k\pi i/n}-1},$$

which is equivalent to the partial fraction expansion of $1/(z^n-1)$.)

6.74 Since $\tan 2z + \sec 2z = (\sin z + \cos z)/(\cos z - \sin z)$, setting $x = 1$ in (6.94) gives $T_n(1) = 2^n T_n$ when n is odd, $T_n(1) = 2^n|E_n|$ when n is even, where $1/\cos z = \sum_{n\geqslant 0}|E_{2n}|z^{2n}/(2n)!$. (The coefficients $|E_n|$ are called *secant numbers*; with alternating signs they are called *Euler numbers*, not to be confused with the Eulerian numbers $\left\langle {n \atop k} \right\rangle$. We have $\langle E_0, E_2, E_4, \ldots\rangle = \langle 1, -1, 5, -61, 1385, -50521, 2702765, \ldots\rangle$.)

6.75 Let $G(w,z) = \sin z/\cos(w+z)$ and $H(w,z) = \cos z/\cos(w+z)$, and let $G(w,z) + H(w,z) = \sum_{m,n} A_{m,n} w^m z^n/m!\,n!$. Then the equations $G(w,0) = 0$ and $\left(\frac{\partial}{\partial z} - \frac{\partial}{\partial w}\right)G(w,z) = H(w,z)$ imply that $A_{m,0} = 0$ when m is odd, $A_{m,n+1} = A_{m+1,n} + A_{m,n}$ when $m+n$ is even; the equations $H(0,z) = 1$ and $\left(\frac{\partial}{\partial w} - \frac{\partial}{\partial z}\right)H(w,z) = G(w,z)$ imply that $A_{0,n} = [n=0]$ when n is even, $A_{m+1,n} = A_{m,n+1} + A_{m,n}$ when $m+n$ is odd. Consequently the nth row below the apex of the triangle contains the numbers $A_{n,0}$, $A_{n-1,1}$, $\ldots$, $A_{0,n}$. At the left, $A_{n,0}$ is the secant number $|E_n|$; at the right, $A_{0,n} = T_n + [n=0]$.

6.76 Let A_n denote the sum. Looking ahead to equation (7.49), we see that $\sum_n A_n z^n/n! = \sum_{n,k}(-1)^k {n \brace k} 2^{n-k} k!\, z^n/n! = \sum_k (-1)^k 2^{-k}(e^{2z}-1)^k = 2/(e^{2z}+1) = 1 - \tanh z$. Therefore, by exercise 23 or 72,

$$A_n \;=\; (2^{n+1} - 4^{n+1})B_{n+1}/(n+1) \;=\; (-1)^{(n+1)/2}T_n + [n=0]\,.$$

6.77 This follows by induction on m, using the recurrence in exercise 18. It can also be proved from (6.50), using the fact that

$$\begin{aligned}\frac{(-1)^{m-1}(m-1)!}{(e^z-1)^m} &= (D+1)^{\overline{m-1}}\frac{1}{e^z-1}\\ &= \sum_{k=0}^{m-1}\begin{bmatrix} m \\ m-k\end{bmatrix}\frac{d^{m-k-1}}{dz^{m-k-1}}\frac{1}{e^z-1}\,, \qquad \text{integer } m > 0.\end{aligned}$$

The latter equation, incidentally, is equivalent to

$$\frac{d^m}{dz^m}\frac{1}{e^z-1} \;=\; (-1)^m\sum_k \left\{ {m+1 \atop k} \right\}\frac{(k-1)!}{(e^z-1)^k}\,, \qquad \text{integer } m \geqslant 0.$$

6.78 If $p(x)$ is any polynomial of degree $\leqslant n$, we have

$$p(x) \;=\; \sum_k p(-k)\binom{-x}{k}\binom{x+n}{n-k}\,,$$

because this equation holds for $x = 0, -1, \ldots, -n$. The stated identity is the special case where $p(x) = x\sigma_n(x)$ and $x = 1$. Incidentally, we obtain a simpler expression for Bernoulli numbers in terms of Stirling numbers by

setting $k = 1$ in (6.99):

$$\sum_{k\geqslant 0} \left\{ {m \atop k} \right\} (-1)^k \frac{k!}{k+1} \;=\; B_m\,.$$

6.79 Sam Loyd [256, pages 288 and 378] gave the construction

and claimed to have invented (but not published) the $64 = 65$ arrangement in 1858. (Similar paradoxes go back at least to the eighteenth century, but Loyd found better ways to present them.)

6.80 We expect $A_m/A_{m-1} \approx \phi$, so we try $A_{m-1} = 618034 + r$ and $A_{m-2} = 381966 - r$. Then $A_{m-3} = 236068 + 2r$, etc., and we find $A_{m-18} = 144 - 2584r$, $A_{m-19} = 154 + 4181r$. Hence $r = 0$, $x = 154$, $y = 144$, $m = 20$.

6.81 If $P(F_{n+1}, F_n) = 0$ for infinitely many *even* values of n, then $P(x, y)$ is divisible by $U(x, y) - 1$, where $U(x, y) = x^2 - xy - y^2$. For if t is the total degree of P, we can write

$$P(x,y) \;=\; \sum_{k=0}^{t} q_k x^k y^{t-k} + \sum_{j+k<t} r_{j,k} x^j y^k \;=\; Q(x,y) + R(x,y)\,.$$

Then

$$\frac{P(F_{n+1}, F_n)}{F_n^t} \;=\; \sum_{k=0}^{t} q_k \left(\frac{F_{n+1}}{F_n}\right)^k + O(1/F_n)$$

and we have $\sum_{k=0}^{t} q_k \phi^k = 0$ by taking the limit as $n \to \infty$. Hence $Q(x, y)$ is a multiple of $U(x, y)$, say $A(x, y)U(x, y)$. But $U(F_{n+1}, F_n) = (-1)^n$ and n is even, so $P_0(x, y) = P(x, y) - \big(U(x, y) - 1\big)A(x, y)$ is another polynomial such that $P_0(F_{n+1}, F_n) = 0$. The total degree of P_0 is less than t, so P_0 is a multiple of $U - 1$ by induction on t.

Similarly, $P(x, y)$ is divisible by $U(x, y) + 1$ if $P(F_{n+1}, F_n) = 0$ for infinitely many *odd* values of n. A combination of these two facts gives the desired necessary and sufficient condition: $P(x, y)$ is divisible by $U(x, y)^2 - 1$.

6.82 First add the digits without carrying, getting digits 0, 1, and 2. Then use the two carry rules

$$\begin{aligned} 0\,(d{+}1)\,(e{+}1) &\to 1\,d\,e\,, \\ 0\,(d{+}2)\,0\,e &\to 1\,d\,0\,(e+1)\,, \end{aligned}$$

always applying the leftmost applicable carry. This process terminates because the binary value obtained by reading $(b_m \ldots b_2)_F$ as $(b_m \ldots b_2)_2$ increases whenever a carry is performed. But a carry might propagate to the right of the "Fibonacci point"; for example, $(1)_F{+}(1)_F$ becomes $(10.01)_F$. Such rightward propagation extends at most two positions; and those two digit positions can be zeroed again by using the text's "add 1" algorithm if necessary.

Incidentally, there's a corresponding "multiplication" operation on nonnegative integers: If $m = F_{j_1} + \cdots + F_{j_q}$ and $n = F_{k_1} + \cdots + F_{k_r}$ in the Fibonacci number system, let $m \circ n = \sum_{b=1}^{q} \sum_{c=1}^{r} F_{j_b + k_c}$, by analogy with multiplication of binary numbers. (This definition implies that $m \circ n \approx \sqrt{5}\,mn$ when m and n are large, although $1 \circ n \approx \phi^2 n$.) Fibonacci addition leads to a proof of the associative law $l \circ (m \circ n) = (l \circ m) \circ n$.

Exercise: $m \circ n = mn + \lfloor (m{+}1)/\phi \rfloor n + m \lfloor (n{+}1)/\phi \rfloor$.

6.83 Yes; for example, we can take

$$\begin{aligned} A_0 &= 331635635998274737472200656430763\,; \\ A_1 &= 151002891108840197118959030549878 5\,. \end{aligned}$$

The resulting sequence has the property that A_n is divisible by (but unequal to) p_k when $n \bmod m_k = r_k$, where the numbers (p_k, m_k, r_k) have the following 18 respective values:

$(3,4,1)$	$(2,3,2)$	$(5,5,1)$
$(7,8,3)$	$(17,9,4)$	$(11,10,2)$
$(47,16,7)$	$(19,18,10)$	$(61,15,3)$
$(2207,32,15)$	$(53,27,16)$	$(31,30,24)$
$(1087,64,31)$	$(109,27,7)$	$(41,20,10)$
$(4481,64,63)$	$(5779,54,52)$	$(2521,60,60)$

One of these triples applies to every integer n; for example, the six triples in the first column cover every odd value of n, and the middle column covers all even n that are not divisible by 6. The remainder of the proof is based on the fact that $A_{m+n} = A_m F_{n-1} + A_{m+1} F_n$, together with the congruences

$$\begin{aligned} A_0 &\equiv F_{m_k - r_k} \bmod p_k\,, \\ A_1 &\equiv F_{m_k - r_k + 1} \bmod p_k\,, \end{aligned}$$

for each of the triples (p_k, m_k, r_k). (An improved solution, in which A_0 and A_1 are numbers of "only" 17 digits each, is also possible [218].)

6.84 The sequences of exercise 62 satisfy $A_{-m} = A_m$, $B_{-m} = -B_m$, and

$$\begin{aligned} A_m A_n &= A_{m+n} + A_{m-n}\,; \\ A_m B_n &= B_{m+n} - B_{m-n}\,; \\ B_m B_n &= A_{m+n} - A_{m-n}\,. \end{aligned}$$

Let $f_k = B_{mk}/A_{mk+l}$ and $g_k = A_{mk}/B_{mk+l}$, where $l = \frac{1}{2}(n-m)$. Then $f_{k+1} - f_k = A_l B_m/(A_{2mk+n} + A_m)$ and $g_k - g_{k+1} = A_l B_m/(A_{2mk+n} - A_m)$; hence we have

$$\begin{aligned} S^+_{m,n} &= \frac{\sqrt{5}}{A_l B_m} \lim_{k\to\infty} (f_k - f_0) &&= \frac{\sqrt{5}}{\phi^l A_l L_m}\,; \\ S^-_{m,n} &= \frac{\sqrt{5}}{A_l B_m} \lim_{k\to\infty} (g_0 - g_k) &&= \frac{\sqrt{5}}{A_l L_m}\left(\frac{2}{B_l} - \frac{1}{\phi^l}\right) \\ & &&= \frac{2}{F_l L_l L_m} - S^+_{m,n}\,. \end{aligned}$$

6.85 The property holds if and only if N has one of the seven forms 5^k, $2\cdot5^k$, $4\cdot5^k$, $3^j\cdot5^k$, $6\cdot5^k$, $7\cdot5^k$, $14\cdot5^k$.

6.86 For any positive integer m, let $r(m)$ be the smallest index j such that C_j is divisible by m; if no such j exists, let $r(m) = \infty$. Then C_n is divisible by m if and only if $\gcd(C_n, C_{r(m)})$ is divisible by m if and only if $C_{\gcd(n,r(m))}$ is divisible by m if and only if $\gcd(n, r(m)) = r(m)$ if and only if n is divisible by $r(m)$.

(Conversely, the gcd condition is easily seen to be implied by the condition that the sequence C_1, C_2, ... has a function $r(m)$, possibly infinite, such that C_n is divisible by m if and only if n is divisible by $r(m)$.)

Now let $\Pi(n) = C_1 C_2 \ldots C_n$, so that

$$\binom{m+n}{m}_{\mathcal{C}} = \frac{\Pi(m+n)}{\Pi(m)\,\Pi(n)}\,.$$

If p is prime, the number of times p divides $\Pi(n)$ is $f_p(n) = \sum_{k\geqslant1} \lfloor n/r(p^k) \rfloor$, since $\lfloor n/p^k \rfloor$ is the number of elements $\{C_1, \ldots, C_n\}$ that are divisible by p^k. Therefore $f_p(m+n) \geqslant f_p(m) + f_p(n)$ for all p, and $\binom{m+n}{m}_{\mathcal{C}}$ is an integer.

6.87 The matrix product is

$$\begin{pmatrix} K_{n-2}(x_2, \ldots, x_{n-1}) & K_{n-1}(x_2, \ldots, x_{n-1}, x_n) \\ K_{n-1}(x_1, x_2, \ldots, x_{n-1}) & K_n(x_1, x_2, \ldots, x_{n-1}, x_n) \end{pmatrix}.$$

This relates to products of L and R as in (6.137), because we have

$$R^a \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & a \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} L^a .$$

The determinant is $K_n(x_1, \ldots, x_n)$; the more general tridiagonal determinant

$$\det \begin{pmatrix} x_1 & 1 & 0 & \ldots & 0 \\ y_2 & x_2 & 1 & & 0 \\ 0 & y_3 & x_3 & 1 & \vdots \\ \vdots & & & \ddots & 1 \\ 0 & 0 & \ldots & y_n & x_n \end{pmatrix}$$

satisfies the recurrence $D_n = x_n D_{n-1} - y_n D_{n-2}$.

6.88 Let $\alpha^{-1} = a_0 + 1/\bigl(a_1 + 1/(a_2 + \cdots)\bigr)$ be the continued fraction representation of α^{-1}. Then we have

$$\frac{a_0}{z} + \cfrac{1}{A_0(z) + \cfrac{1}{A_1(z) + \cfrac{1}{A_2(z) + \cfrac{1}{\ddots}}}} = \frac{1-z}{z} \sum_{n \geqslant 1} z^{\lfloor n\alpha \rfloor} ,$$

where

$$A_m(z) = \frac{z^{-q_{m+1}} - z^{-q_{m-1}}}{z^{-q_m} - 1} , \qquad q_m = K_m(a_1, \ldots, a_m) .$$

A proof analogous to the text's proof of (6.146) uses a generalization of Zeckendorf's theorem (Fraenkel [129, §4]). If $z = 1/b$, where b is an integer $\geqslant 2$, this gives the continued fraction representation of the transcendental number $(b-1)\sum_{n \geqslant 1} b^{-\lfloor n\alpha \rfloor}$, as in exercise 49.

6.89 Let $p = K(0, a_1, a_2, \ldots, a_m)$, so that p/n is the mth convergent to the continued fraction. Then $\alpha = p/n + (-1)^m/nq$, where $q = K(a_1, \ldots, a_m, \beta)$ and $\beta > 1$. The points $\{k\alpha\}$ for $0 \leqslant k < n$ can therefore be written

$$\frac{0}{n}, \ \frac{1}{n} + \frac{(-1)^m \pi_1}{nq}, \ \ldots, \ \frac{n-1}{n} + \frac{(-1)^m \pi_{n-1}}{nq} ,$$

where $\pi_1 \ldots \pi_{n-1}$ is a permutation of $\{1, \ldots, n-1\}$. Let $f(\nu)$ be the number of such points $< \nu$; then $f(\nu)$ and νn both increase by 1 when ν increases from k/n to $(k+1)/n$, except when $k = 0$ or $k = n-1$, so they never differ by 2 or more.

6.90 By (6.139) and (6.136), we want to maximize $K(a_1, \ldots, a_m)$ over all sequences of positive integers whose sum is $\leqslant n+1$. The maximum occurs when all the a's are 1, for if $j \geqslant 1$ and $a \geqslant 1$ we have

$$\begin{aligned} &K_{j+k+1}(1, \ldots, 1, a+1, b_1, \ldots, b_k) \\ &\quad = K_{j+k+1}(1, \ldots, 1, a, b_1, \ldots, b_k) + K_j(1, \ldots, 1)\, K_k(b_1, \ldots, b_k) \\ &\quad \leqslant K_{j+k+1}(1, \ldots, 1, a, b_1, \ldots, b_k) + K_{j+k}(1, \ldots, 1, a, b_1, \ldots, b_k) \\ &\quad = K_{j+k+2}(1, \ldots, 1, a, b_1, \ldots, b_k)\,. \end{aligned}$$

(Motzkin and Straus [278] show how to solve more general maximization problems on continuants.)

6.91 A candidate for the case $n \bmod 1 = \frac{1}{2}$ appears in [213, §6], although it may be best to multiply the integers discussed there by some constant involving $\sqrt{\pi}$. Alternatively, Renzo Sprugnoli observes that we can define $\left\{ {n \atop m} \right\} = \sum_k \binom{m}{k} k^n (-1)^{(} m-k)/m!$ for integer $m \geqslant 0$ and arbitrary $n \geqslant 0$; then (6.3) holds for all $n \geqslant 1$.

6.92 (a) If there are only finitely many solutions, it is natural to conjecture that the same holds for all primes. (b) The behavior of b_n is quite strange: We have $b_n = \operatorname{lcm}(1, \ldots, n)$ for $968 \leqslant n \leqslant 1066$; on the other hand, $b_{600} = \operatorname{lcm}(1, \ldots, 600)/(3^3 \cdot 5^2 \cdot 43)$. Andrew Odlyzko observes that p divides $\operatorname{lcm}(1, \ldots, n)/b_n$ if and only if $kp^m \leqslant n < (k+1)p^m$ for some $m \geqslant 1$ and some $k < p$ such that p divides the numerator of H_k. Therefore infinitely many such n exist if it can be shown, for example, that almost all primes have only one such value of k (namely $k = p-1$).

Another reason to remember 1066?

6.93 (Brent [38] found the surprisingly large partial quotient 1568705 in e^γ, but this seems to be just a coincidence. For example, Gosper has found even larger partial quotients in π: The 453,294th is 12996958 and the 11,504,931st is 878783625.)

6.94 Consider the generating function $\sum_{m,n \geqslant 0} \left| {m+n \atop m} \right| w^m z^n$, which has the form $\sum_n \big(wF(a,b,c) + zF(a',b',c')\big)^n$, where $F(a,b,c)$ is the differential operator $a + b\vartheta_w + c\vartheta_z$.

6.95 Complete success might be difficult or impossible, because Stirling numbers are not "holonomic" in the sense of [382].

7.1 Substitute z^4 for $\square$ and z for $\square$ in the generating function, getting $1/(1-z^4-z^2)$. This is like the generating function for T, but with z replaced by z^2. Therefore the answer is zero if m is odd, otherwise $F_{m/2+1}$.

7.2 $G(z) = 1/(1-2z) + 1/(1-3z)$; $\widehat{G}(z) = e^{2z} + e^{3z}$.

7.3 Set $z = 1/10$ in the generating function, getting $\frac{10}{9} \ln \frac{10}{9}$.

7.4 Divide $P(z)$ by $Q(z)$, getting a quotient $T(z)$ and a remainder $P_0(z)$ whose degree is less than the degree of Q. The coefficients of $T(z)$ must be added to the coefficients $[z^n]\,P_0(z)/Q(z)$ for small n. (This is the polynomial $T(z)$ in (7.28).)

7.5 This is the convolution of $(1+z^2)^r$ with $(1+z)^r$, so

$$S(z) \;=\; (1+z+z^2+z^3)^r\,.$$

Incidentally, no simple form is known for the coefficients of this generating function; hence the stated sum probably has no simple closed form. (We can use generating functions to obtain negative results as well as positive ones.)

7.6 Let the solution to $g_0=\alpha$, $g_1=\beta$, $g_n=g_{n-1}+2g_{n-2}+(-1)^n\gamma$ be $g_n=A(n)\alpha+B(n)\beta+C(n)\gamma$. The function 2^n works when $\alpha=1$, $\beta=2$, $\gamma=0$; the function $(-1)^n$ works when $\alpha=1$, $\beta=-1$, $\gamma=0$; the function $(-1)^n n$ works when $\alpha=0$, $\beta=-1$, $\gamma=3$. Hence $A(n)+2B(n)=2^n$, $A(n)-B(n)=(-1)^n$, and $-B(n)+3C(n)=(-1)^n n$.

7.7 $G(z)=\bigl(z/(1-z)^2\bigr)G(z)+1$, hence

$$G(z) \;=\; \frac{1-2z+z^2}{1-3z+z^2} \;=\; 1+\frac{z}{1-3z+z^2}\,;$$

I bet that the controversial "fan of order zero" does have one spanning tree.

we have $g_n=F_{2n}+[n=0]$.

7.8 Differentiate $(1-z)^{-x-1}$ twice with respect to x, obtaining

$$\binom{x+n}{n}\bigl((H_{x+n}-H_x)^2-(H^{(2)}_{x+n}-H^{(2)}_x)\bigr)\,.$$

Now set $x=m$.

7.9 $(n+1)(H_n^2-H_n^{(2)})-2n(H_n-1)$.

7.10 The identity $H_{k-1/2}-H_{-1/2}=\frac{2}{2k-1}+\cdots+\frac{2}{1}=2H_{2k}-H_k$ implies that $\sum_k\binom{2k}{k}\binom{2n-2k}{n-k}(2H_{2k}-H_k)=4^nH_n$.

7.11 (a) $C(z)=A(z)B(z^2)/(1-z)$. (b) $zB'(z)=A(2z)e^z$, hence $A(z)=\frac{z}{2}e^{-z/2}B'(\frac{z}{2})$. (c) $A(z)=B(z)/(1-z)^{r+1}$, hence $B(z)=(1-z)^{r+1}A(z)$ and we have $f_k(r)=\binom{r+1}{k}(-1)^k$.

7.12 C_n. The numbers in the upper row correspond to the positions of $+1$'s in a sequence of $+1$'s and -1's that defines a "mountain range"; the numbers in the lower row correspond to the positions of -1's. For example, the given array corresponds to

7.13 Extend the sequence periodically (let $x_{m+k} = x_k$) and define $s_n = x_1 + \cdots + x_n$. We have $s_m = l$, $s_{2m} = 2l$, etc. There must be a largest index k_j such that $s_{k_j} = j$, $s_{k_j+m} = l+j$, etc. These indices $k_1, \ldots, k_l$ (modulo m) specify the cyclic shifts in question.

For example, in the sequence $\langle -2, 1, -1, 0, 1, 1, -1, 1, 1, 1\rangle$ with $m = 10$ and $l = 2$ we have $k_1 = 17$, $k_2 = 24$.

7.14 $\widehat{G}(z) = -2z\widehat{G}(z) + \widehat{G}(z)^2 + z$ (be careful about the final term!) leads via the quadratic formula to

$$\widehat{G}(z) \;=\; \frac{1 + 2z - \sqrt{1 + 4z^2}}{2}\,.$$

Hence $g_{2n+1} = 0$ and $g_{2n} = (-1)^n (2n)!\, C_{n-1}$, for all $n > 0$.

7.15 There are $\binom{n}{k}\varpi_{n-k}$ partitions with k other objects in the subset containing $n+1$. Hence $\widehat{P}'(z) = e^z \widehat{P}(z)$. The solution to this differential equation is $\widehat{P}(z) = e^{e^z + c}$, and $c = -1$ since $\widehat{P}(0) = 1$. (We can also get this result by summing (7.49) on m, since $\varpi_n = \sum_m \left\{ {n \atop m} \right\}$.)

7.16 One way is to take the logarithm of

$$B(z) \;=\; 1\Big/\big((1-z)^{a_1}(1-z^2)^{a_2}(1-z^3)^{a_3}(1-z^4)^{a_4}\ldots\big),$$

then use the formula for $\ln \frac{1}{1-z}$ and interchange the order of summation.

7.17 This follows since $\int_0^\infty t^n e^{-t}\,dt = n!$. There's also a formula that goes in the other direction:

$$\widehat{G}(z) \;=\; \frac{1}{2\pi}\int_{-\pi}^{+\pi} G(ze^{-i\theta})\, e^{e^{i\theta}}\, d\theta\,.$$

7.18 (a) $\zeta(z - \frac{1}{2})$; (b) $-\zeta'(z)$; (c) $\zeta(z)/\zeta(2z)$. Every positive integer is uniquely representable as $m^2 q$, where q is squarefree.

7.19 If $n > 0$, the coefficient $[z^n]\,\exp\big(x \ln F(z)\big)$ is a polynomial of degree n in x that's a multiple of x. The first convolution formula comes from equating coefficients of z^n in $F(z)^x F(z)^y = F(z)^{x+y}$. The second comes from equating coefficients of z^{n-1} in $F'(z)F(z)^{x-1}F(z)^y = F'(z)F(z)^{x+y-1}$, because we have

$$F'(z)F(z)^{x-1} \;=\; x^{-1}\frac{\partial}{\partial z}\big(F(z)^x\big) \;=\; x^{-1}\sum_{n\geqslant 0} n f_n(x) z^{n-1}\,.$$

(Further convolutions follow by taking $\partial/\partial x$, as in (7.43).)

Still more is true, as shown in [221]: We have

$$\sum_{k=0}^{n} \frac{x f_k(x+tk)}{x+tk}\,\frac{y f_{n-k}(y+t(n-k))}{y+t(n-k)} \;=\; \frac{(x+y) f_n(x+y+tn)}{x+y+tn}\,,$$

for arbitrary x, y, and t. In fact, $xf_n(x+tn)/(x+tn)$ is the sequence of polynomials for the coefficients of $\mathcal{F}_t(z)^x$, where

$$\mathcal{F}_t(z) \;=\; F\bigl(z\mathcal{F}_t(z)^t\bigr)\,.$$

(We saw special cases in (5.59) and (6.52).)

7.20 Let $G(z) = \sum_{n\geqslant 0} g_n z^n$. Then

$$z^l G^{(k)}(z) \;=\; \sum_{n\geqslant 0} n^{\underline{k}} g_n z^{n-k+l} \;=\; \sum_{n\geqslant 0} (n+k-l)^{\underline{k}} g_{n+k-l} z^n$$

for all $k, l \geqslant 0$, if we regard $g_n = 0$ for $n < 0$. Hence if $P_0(z), \ldots, P_m(z)$ are polynomials, not all zero, having maximum degree d, then there are polynomials $p_0(n), \ldots, p_{m+d}(n)$ such that

$$P_0(z)G(z) + \cdots + P_m(z)G^{(m)}(z) \;=\; \sum_{n\geqslant 0}\sum_{j=0}^{m+d} p_j(n) g_{n+j-d} z^n\,.$$

Therefore a differentiably finite $G(z)$ implies that

$$\sum_{j=0}^{m+d} p_j(n+d) g_{n+j} \;=\; 0\,, \qquad \text{for all } n \geqslant 0.$$

The converse is similar. (One consequence is that $G(z)$ is differentiably finite if and only if the corresponding egf, $\widehat{G}(z)$, is differentiably finite.)

This slow method of finding the answer is just the cashier's way of stalling until the police come.

The USA has two-cent pieces, but they haven't been minted since 1873.

7.21 This is the problem of giving change with denominations 10 and 20, so $G(z) = 1/(1-z^{10})(1-z^{20}) = \check{G}(z^{10})$, where $\check{G}(z) = 1/(1-z)(1-z^2)$. (a) The partial fraction decomposition of $\check{G}(z)$ is $\frac{1}{2}(1-z)^{-2} + \frac{1}{4}(1-z)^{-1} + \frac{1}{4}(1+z)^{-1}$, so $[z^n]\,\check{G}(z) = \frac{1}{4}\bigl(2n+3+(-1)^n\bigr)$. Setting $n = 50$ yields 26 ways to make the payment. (b) $\check{G}(z) = (1+z)/(1-z^2)^2 = (1+z)(1+2z^2+3z^4+\cdots)$, so $[z^n]\,\check{G}(z) = \lfloor n/2 \rfloor + 1$. (Compare this with the value $N_n = \lfloor n/5 \rfloor + 1$ in the text's coin-changing problem. The bank robber's problem is equivalent to the problem of making change with pennies and tuppences.)

7.22 Each polygon has a "base" (the line segment at the bottom). If A and B are triangulated polygons, let $A \triangle B$ be the result of pasting the base of A to the upper left diagonal of $\triangle$, and pasting the base of B to the upper right diagonal. Thus, for example,

(The polygons might need to be warped a bit and/or banged into shape.) Every triangulation arises in this way, because the base line is part of a unique triangle and there are triangulated polygons A and B at its left and right.

Replacing each triangle by z gives a power series in which the coefficient of z^n is the number of triangulations with n triangles, namely the number of ways to decompose an $(n+2)$-gon into triangles. Since $P = 1 + zP^2$, this is the generating function for Catalan numbers $C_0 + C_1 z + C_2 z^2 + \cdots$; the number of ways to triangulate an n-gon is $C_{n-2} = \binom{2n-4}{n-2}/(n-1)$.

7.23 Let a_n be the stated number, and b_n the number of ways with a $2\times1\times1$ notch missing at the top. By considering the possible patterns visible on the top surface, we have

$$\begin{aligned} a_n &= 2a_{n-1} + 4b_{n-1} + a_{n-2} + [n{=}0]\,; \\ b_n &= a_{n-1} + b_{n-1}\,. \end{aligned}$$

Hence the generating functions satisfy $A = 2zA + 4zB + z^2A + 1$, $B = zA + zB$, and we have

$$A(z) = \frac{1-z}{(1+z)(1-4z+z^2)}\,.$$

This formula relates to the problem of $3 \times n$ domino tilings; we have $a_n = \frac{1}{3}\bigl(U_{2n} + V_{2n+1} + (-1)^n\bigr) = \frac{1}{6}(2+\sqrt{3}\,)^{n+1} + \frac{1}{6}(2-\sqrt{3}\,)^{n+1} + \frac{1}{3}(-1)^n$, which is $(2+\sqrt{3}\,)^{n+1}/6$ rounded to the nearest integer.

7.24 $n\sum_{k_1+\cdots+k_m=n} k_1 \cdot \ldots \cdot k_m/m = F_{2n+1} + F_{2n-1} - 2$. (Consider the coefficient $[z^{n-1}]\,\frac{d}{dz}\ln\bigl(1/(1-G(z))\bigr)$, where $G(z) = z/(1-z)^2$.)

7.25 The generating function is $P(z)/(1-z^m)$, where $P(z) = z + 2z^2 + \cdots + (m-1)z^{m-1} = \bigl((m-1)z^{m+1} - mz^m + z\bigr)/(1-z)^2$. The denominator is $Q(z) = 1 - z^m = (1-\omega^0 z)(1-\omega^1 z)\ldots(1-\omega^{m-1}z)$. By the rational expansion theorem for distinct roots, we obtain

$$n \bmod m = \frac{m-1}{2} + \sum_{k=1}^{m-1} \frac{\omega^{-kn}}{\omega^k - 1}\,.$$

7.26 $(1-z-z^2)\mathfrak{F}(z) = F(z)$ leads to $\mathfrak{F}_n = \bigl(2(n+1)F_n + nF_{n+1}\bigr)/5$ as in equation (7.61).

7.27 Each oriented cycle pattern begins with or or a $2 \times k$ cycle (for some $k \geqslant 2$) oriented in one of two ways. Hence

$$Q_n = Q_{n-1} + Q_{n-2} + 2Q_{n-2} + 2Q_{n-3} + \cdots + 2Q_0$$

for $n \geqslant 2$; $Q_0 = Q_1 = 1$. The generating function is therefore

$$\begin{aligned} Q(z) &= zQ(z) + z^2Q(z) + 2z^2Q(z)/(1-z) + 1 \\ &= 1/\bigl(1 - z - z^2 - 2z^2/(1-z)\bigr) \end{aligned}$$

$$= \frac{(1-z)}{(1-2z-2z^2+z^3)}$$
$$= \frac{\phi^2/5}{1-\phi^2 z} + \frac{\phi^{-2}/5}{1-\phi^{-2}z} + \frac{2/5}{1+z},$$

and $Q_n = \bigl(\phi^{2n+2} + \phi^{-2n-2} + 2(-1)^n\bigr)/5 = \bigl((\phi^{n+1} - \hat\phi^{n+1})/\sqrt5\bigr)^2 = F_{n+1}^2$.

7.28 In general if $A(z) = (1+z+\cdots+z^{m-1})B(z)$, we have $A_r + A_{r+m} + A_{r+2m} + \cdots = B(1)$ for $0 \leqslant r < m$. In this case $m = 10$ and $B(z) = (1+z+\cdots+z^9)(1+z^2+z^4+z^6+z^8)(1+z^5)$.

7.29 $F(z) + F(z)^2 + F(z)^3 + \cdots = z/(1-z-z^2-z) = \bigl(1/(1-(1+\sqrt2)z) - (1/(1-(1-\sqrt2)z)\bigr)/\sqrt8$, so the answer is $\bigl((1+\sqrt2)^n - (1-\sqrt2)^n\bigr)/\sqrt8$.

7.30 $\sum_{k=1}^n \binom{2n-1-k}{n-1}\bigl(a^n b^{n-k}/(1-\alpha z)^k + a^{n-k}b^n/(1-\beta z)^k\bigr)$, by exercise 5.39.

7.31 The dgf is $\zeta(z)^2/\zeta(z-1)$; hence we find $g(n)$ is the product of $(k+1-kp)$ over all prime powers p^k that exactly divide n.

7.32 We may assume that each $b_k \geqslant 0$. A set of arithmetic progressions forms an exact cover if and only if

$$\frac{1}{1-z} = \frac{z^{b_1}}{1-z^{a_1}} + \cdots + \frac{z^{b_m}}{1-z^{a_m}}.$$

Subtract $z^{b_m}/(1-z^{a_m})$ from both sides and set $z = e^{2\pi i/a_m}$. The left side is infinite, and the right side will be finite unless $a_{m-1} = a_m$.

7.33 $(-1)^{n-m+1}[n>m]/(n-m)$.

7.34 We can also write $G_n(z) = \sum_{k_1+(m+1)k_{m+1}=n} \binom{k_1+k_{m+1}}{k_{m+1}}(z^m)^{k_{m+1}}$. In general, if

$$G_n = \sum_{k_1+2k_2+\cdots+rk_r=n} \binom{k_1+k_2+\cdots+k_r}{k_1,k_2,\ldots,k_r} z_1^{k_1} z_2^{k_2}\ldots z_r^{k_r},$$

we have $G_n = z_1G_{n-1} + z_2G_{n-2} + \cdots + z_rG_{n-r} + [n=0]$, and the generating function is $1/(1-z_1w-z_2w^2-\cdots-z_rw^r)$. In the stated special case the answer is $1/(1-w-z^mw^{m+1})$. (See (5.74) for the case $m=1$.)

7.35 (a) $\frac1n\sum_{0<k<n}\bigl(1/k + 1/(n-k)\bigr) = \frac2n H_{n-1}$. (b) $[z^n]\bigl(\ln\frac{1}{1-z}\bigr)^2 = \frac{2!}{n!}\left[{n \atop 2}\right] = \frac2n H_{n-1}$ by (7.50) and (6.58). Another way to do part (b) is to use the rule $[z^n]\,F(z) = \frac1n[z^{n-1}]\,F'(z)$ with $F(z) = \bigl(\ln\frac{1}{1-z}\bigr)^2$.

7.36 $\frac{1-z^m}{1-z}A(z^m)$.

7.37 (a) The amazing identity $a_{2n} = a_{2n+1} = b_n$ holds in the table

n	0	1	2	3	4	5	6	7	8	9	10
a_n	1	1	2	2	4	4	6	6	10	10	14
b_n	1	2	4	6	10	14	20	26	36	46	60

(b) $A(z) = 1/\bigl((1-z)(1-z^2)(1-z^4)(1-z^8)\dots\bigr)$. (c) $B(z) = A(z)/(1-z)$, and we want to show that $A(z) = (1+z)B(z^2)$. This follows from $A(z) = A(z^2)/(1-z)$.

7.38 $(1-wz)M(w,z) = \sum_{m,n\geqslant 1}\bigl(\min(m,n) - \min(m-1,n-1)\bigr)w^m z^n = \sum_{m,n\geqslant 1} w^m z^n = wz/(1-w)(1-z)$. In general,

$$M(z_1,\dots,z_m) \;=\; \frac{z_1\dots z_m}{(1-z_1)\dots(1-z_m)(1-z_1\dots z_m)}\,.$$

7.39 The answers to the hint are

$$\sum_{1\leqslant k_1<k_2<\cdots<k_m\leqslant n} a_{k_1}a_{k_2}\dots a_{k_m} \quad\text{and}\quad \sum_{1\leqslant k_1\leqslant k_2\leqslant\cdots\leqslant k_m\leqslant n} a_{k_1}a_{k_2}\dots a_{k_m}\,,$$

respectively. Therefore: (a) We want the coefficient of z^m in the product $(1+z)(1+2z)\dots(1+nz)$. This is the reflection of $(z+1)^{\overline{n}}$, so it is $\left[{n+1 \atop n+1}\right] + \left[{n+1 \atop n}\right]z + \dots + \left[{n+1 \atop 1}\right]z^n$ and the answer is $\left[{n+1 \atop n+1-m}\right]$. (b) The coefficient of z^m in $1/\bigl((1-z)(1-2z)\dots(1-nz)\bigr)$ is $\left\{{m+n \atop n}\right\}$ by (7.47).

7.40 The egf for $\langle nF_{n-1} - F_n\rangle$ is $(z-1)\widehat{F}(z)$ where $\widehat{F}(z) = \sum_{n\geqslant 0} F_n z^n/n! = (e^{\phi z} - e^{\hat\phi z})/\sqrt{5}$. The egf for $\langle n_{\downarrow}\rangle$ is $e^{-z}/(1-z)$. The product is

$$5^{-1/2}\bigl(e^{(\hat\phi-1)z} - e^{(\phi-1)z}\bigr) \;=\; 5^{-1/2}(e^{-\phi z} - e^{-\hat\phi z})\,.$$

We have $\widehat{F}(z)e^{-z} = -\widehat{F}(-z)$. So the answer is $(-1)^n F_n$.

7.41 The number of up-down permutations with the largest element n in position $2k$ is $\binom{n-1}{2k-1}A_{2k-1}A_{n-2k}$. Similarly, the number of up-down permutations with the smallest element 1 in position $2k+1$ is $\binom{n-1}{2k}A_{2k}A_{n-2k-1}$, because down-up permutations and up-down permutations are equally numerous. Summing over all possibilities gives

$$2A_n \;=\; \sum_k \binom{n-1}{k} A_k\,A_{n-1-k} + 2[n=0] + [n=1]\,.$$

The egf $\widehat{A}$ therefore satisfies $2\widehat{A}'(z) = \widehat{A}(z)^2 + 1$ and $\widehat{A}(0) = 1$; the given function solves this differential equation. (Consequently $A_n = |E_n| + T_n$ is a secant number when n is even, a tangent number when n is odd.)

7.42 Let a_n be the number of Martian DNA strings that don't end with c or e; let b_n be the number that do. Then

$$\begin{aligned} a_n &= 3a_{n-1} + 2b_{n-1} + [n=0]\,, & b_n &= 2a_{n-1} + b_{n-1}\,; \\ A(z) &= 3zA(z) + 2zB(z) + 1\,, & B(z) &= 2zA(z) + zB(z)\,; \\ A(z) &= \frac{1-z}{1-4z-z^2}\,, & B(z) &= \frac{2z}{1-4z-z^2}\,; \end{aligned}$$

and the total number is $[z^n](1+z)/(1-4z-z^2) = F_{3n+2}$.

7.43 By (5.45), $g_n = \Delta^n \dot{G}(0)$. The nth difference of a product can be written

$$\Delta^n A(z)B(z) = \sum_k \binom{n}{k} \big(\Delta^k E^{n-k} A(z)\big)\big(\Delta^{n-k} B(z)\big)\,,$$

and $E^{n-k} = (1+\Delta)^{n-k} = \sum_j \binom{n-k}{j}\Delta^j$. Therefore we find

$$h_n = \sum_{j,k} \binom{n}{k}\binom{n-k}{j} f_{j+k}\, g_{n-k}\,.$$

This is a sum over all trinomial coefficients; it can be put into the more symmetric form

$$h_n = \sum_{j+k+l=n} \binom{n}{j,k,l} f_{j+k}\, g_{k+l}\,.$$

The empty set is pointless.

7.44 Each partition into k nonempty subsets can be ordered in $k!$ ways, so $b_k = k!$. Thus $\widehat{Q}(z) = \sum_{n,k\geqslant 0} \left\{ {n \atop k} \right\} k!\, z^n/n! = \sum_{k\geqslant 0}(e^z-1)^k = 1/(2-e^z)$. And this is the geometric series $\sum_{k\geqslant 0} e^{kz}/2^{k+1}$, hence $a_k = 1/2^{k+1}$. Finally, $c_k = 2^k$; consider all permutations when the x's are distinct, change each '$>$' between subscripts to '$<$' and allow each '$<$' between subscripts to become either '$<$' or '$=$'. (For example, the permutation $x_1x_3x_2$ produces $x_1 < x_3 < x_2$ and $x_1 = x_3 < x_2$, because $1 < 3 > 2$.)

7.45 This sum is $\sum_{n\geqslant 1} r(n)/n^2$, where $r(n)$ is the number of ways to write n as a product of two relatively prime factors. If n is divisible by t distinct primes, $r(n) = 2^t$. Hence $r(n)/n^2$ is multiplicative and the sum is

$$\begin{aligned} \prod_p \left(1 + \frac{2}{p^2} + \frac{2}{p^4} \cdots\right) &= \prod_p \left(1 + \frac{2}{p^2-1}\right) \\ &= \prod_p \left(\frac{p^2+1}{p^2-1}\right) = \zeta(2)^2/\zeta(4) = \frac{5}{2}\,. \end{aligned}$$

7.46 Let $S_n = \sum_{0\leqslant k\leqslant n/2}\binom{n-2k}{k}\alpha^k$. Then $S_n = S_{n-1} + \alpha S_{n-3} + [n=0]$, and the generating function is $1/(1-z-\alpha z^3)$. When $\alpha = -\frac{4}{27}$, the hint tells us that this has a nice factorization $1/(1+\frac{1}{3}z)(1-\frac{2}{3}z)^2$. The general expansion theorem now tells us that $S_n = (\frac{2}{3}n + c)(\frac{2}{3})^n + \frac{1}{9}(-\frac{1}{3})^n$, and the remaining constant c turns out to be $\frac{8}{9}$.

7.47 The Stern–Brocot representation of $\sqrt{3}$ is $R(LR^2)^\infty$, because

$$\sqrt{3}+1 \;=\; 2+\cfrac{1}{1+\cfrac{1}{\sqrt{3}+1}}\,.$$

The fractions are $\frac{1}{1}$, $\frac{2}{1}$, $\frac{3}{2}$, $\frac{5}{3}$, $\frac{7}{4}$, $\frac{12}{7}$, $\frac{19}{11}$, $\frac{26}{15}$, $\ldots$; they eventually have the cyclic pattern

$$\frac{V_{2n-1}+V_{2n+1}}{U_{2n}},\ \frac{U_{2n}+V_{2n+1}}{V_{2n+1}},\ \frac{U_{2n+2}+V_{2n-1}}{U_{2n}+V_{2n+1}},\ \frac{V_{2n+1}+V_{2n+3}}{U_{2n+2}},\ \ldots.$$

7.48 We have $g_0 = 0$, and if $g_1 = m$ the generating function satisfies

$$aG(z) + bz^{-1}G(z) + cz^{-2}\bigl(G(z) - mz\bigr) + \frac{d}{1-z} \;=\; 0\,.$$

Hence $G(z) = P(z)/(az^2+bz+c)(1-z)$ for some polynomial $P(z)$. Let ρ_1 and ρ_2 be the roots of cz^2+bz+a, with $|\rho_1| \geqslant |\rho_2|$. If $b^2-4ac \leqslant 0$ then $|\rho_1|^2 = \rho_1\rho_2 = a/c$ is rational, contradicting the fact that $\sqrt[n]{g_n}$ approaches $1+\sqrt{2}$. Hence $\rho_1 = (-b+\sqrt{b^2-4ca})/2c = 1+\sqrt{2}$; and this implies that $a=-c$, $b=-2c$, $\rho_2 = 1-\sqrt{2}$. The generating function now takes the form

$$\begin{aligned} G(z) \;&=\; \frac{z\bigl(m-(r+m)z\bigr)}{(1-2z-z^2)(1-z)} \\ &=\; \frac{-r+(2m+r)z}{2(1-2z-z^2)} + \frac{r}{2(1-z)} \;=\; mz + (2m-r)z^2 + \cdots\,, \end{aligned}$$

where $r = d/c$. Since g_2 is an integer, r is an integer. We also have

$$g_n \;=\; \alpha(1+\sqrt{2}\,)^n + \hat{\alpha}(1-\sqrt{2}\,)^n + \tfrac{1}{2}r \;=\; \bigl\lfloor \alpha(1+\sqrt{2}\,)^n \bigr\rfloor\,,$$

and this can hold only if $r=-1$, because $(1-\sqrt{2}\,)^n$ alternates in sign as it approaches zero. Hence $(a,b,c,d) = \pm(1,2,-1,1)$. Now we find $\alpha = \frac{1}{4}(1+\sqrt{2}\,m)$, which is between 0 and 1 only if $0 \leqslant m \leqslant 2$. Each of these values actually gives a solution; the sequences $\langle g_n\rangle$ are $\langle 0,0,1,3,8,\ldots\rangle$, $\langle 0,1,3,8,20,\ldots\rangle$, and $\langle 0,2,5,13,32,\ldots\rangle$.

7.49 (a) The denominator of $\bigl(1/(1-(1+\sqrt{2})z)+1/(1-(1-\sqrt{2})z)\bigr)$ is $1-2z-z^2$; hence $a_n = 2a_{n-1}+a_{n-2}$ for $n \geqslant 2$. (b) True because a_n is even and $-1 < 1-\sqrt{2} < 0$. (c) Let

$$b_n \;=\; \left(\frac{p+\sqrt{q}}{2}\right)^n + \left(\frac{p-\sqrt{q}}{2}\right)^n .$$

We would like b_n to be odd for all $n > 0$, and $-1 < (p-\sqrt{q})/2 < 0$. Working as in part (a), we find $b_0 = 2$, $b_1 = p$, and $b_n = pb_{n-1} + \frac{1}{4}(q-p^2)b_{n-2}$ for $n \geqslant 2$. One satisfactory solution has $p = 3$ and $q = 17$.

7.50 Extending the multiplication idea of exercise 22, we have

Replace each n-gon by z^{n-2}. This substitution behaves properly under multiplication, because the pasting operation takes an m-gon and an n-gon into an $(m+n-2)$-gon. Thus the generating function is

$$Q \;=\; 1 + zQ^2 + z^2Q^3 + z^3Q^4 + \cdots \;=\; 1 + \frac{zQ^2}{1-zQ}$$

and the quadratic formula gives $Q = \bigl(1+z-\sqrt{1-6z+z^2}\bigr)/4z$. The coefficient of z^{n-2} in this power series is the number of ways to put nonoverlapping diagonals into a convex n-gon. These coefficients apparently have no closed form in terms of other quantities that we have discussed in this book, but their asymptotic behavior is known [207, exercise 2.2.1–12].

Give me Legendre polynomials and I'll give you a closed form.

Incidentally, if each n-gon in Q is replaced by wz^{n-2} we get

$$Q \;=\; \frac{1+z-\sqrt{1-(4w+2)z+z^2}}{2(1+w)z},$$

a formula in which the coefficient of $w^m z^{n-2}$ is the number of ways to divide an n-gon into m polygons by nonintersecting diagonals.

7.51 The key first step is to observe that the square of the number of ways is the number of cycle patterns of a certain kind, generalizing exercise 27. These can be enumerated by evaluating the determinant of a matrix whose eigenvalues are not difficult to determine. When $m = 3$ and $n = 4$, the fact that $\cos 36^\circ = \phi/2$ is helpful (exercise 6.46).

7.52 The first few cases are $p_0(y) = 1$, $p_1(y) = y$, $p_2(y) = y^2+y$, $p_3(y) = y^3+3y^2+3y$. Let $p_n(y) = q_{2n}(x)$ where $y = x(1-x)$; we seek a generating function that defines $q_{2n+1}(x)$ in a convenient way. One such function is $\sum_n q_n(x)z^n/n! = 2e^{ixz}/(e^{iz}+1)$, from which it follows

that $q_n(x) = i^n E_n(x)$, where $E_n(x)$ is called an Euler polynomial. We have $\sum(-1)^x x^n\,\delta x = \frac{1}{2}(-1)^{x+1}E_n(x)$, so Euler polynomials are analogous to Bernoulli polynomials, and they have factors analogous to those in (6.98). By exercise 6.23 we have $nE_{n-1}(x) = \sum_{k=0}^{n}\binom{n}{k}B_k x^{n-k}(2-2^{k+1})$; this polynomial has integer coefficients by exercise 6.54. Hence $q_{2n}(x)$, whose coefficients have denominators that are powers of 2, must have integer coefficients. Hence $p_n(y)$ has integer coefficients. Finally, the relation $(4y-1)p_n''(y)+2p_n'(y) = 2n(2n-1)p_{n-1}(y)$ shows that

$$2m(2m-1)\left|{n \atop m}\right| \;=\; m(m+1)\left|{n \atop m+1}\right| + 2n(2n-1)\left|{n-1 \atop m-1}\right|,$$

and it follows that the $\left|{n \atop m}\right|$'s are positive. (A similar proof shows that the related quantity $(-1)^n(2n+2)E_{2n+1}(x)/(2x-1)$ has positive integer coefficients, when expressed as an nth degree polynomial in y.) It can be shown that $\left|{n \atop 1}\right|$ is the Genocchi number $(-1)^{n-1}(2^{2n+1}-2)B_{2n}$ (see exercise 6.24), and that $\left|{n \atop n-1}\right| = \binom{n}{2}$, $\left|{n \atop n-2}\right| = 2\binom{n+1}{4}+3\binom{n}{4}$, etc.

7.53 It is $P_{(1+V_{4n+1}+V_{4n+3})/6}$. Thus, for example, $T_{20} = P_{12} = 210$; $T_{285} = P_{165} = 40755$.

7.54 Let E_k be the operation on power series that sets all coefficients to zero except those of z^n where $n \bmod m = k$. The stated construction is equivalent to the operation

$$E_0\,S\,E_0\,S\,(E_0+E_1)\,S\,\ldots\,S\,(E_0+E_1+\cdots+E_{m-1})$$

applied to $1/(1-z)$, where S means "multiply by $1/(1-z)$." There are $m!$ terms

$$E_0\,S\,E_{k_1}\,S\,E_{k_2}\,S\,\ldots\,S\,E_{k_m}$$

where $0 \leqslant k_j < j$, and every such term evaluates to $z^{rm}/(1-z^m)^{m+1}$ if r is the number of places where $k_j < k_{j+1}$. Exactly $\left\langle {m \atop r} \right\rangle$ terms have a given value of r, so the coefficient of z^{mn} is $\sum_{r=0}^{m-1}\left\langle {m \atop r} \right\rangle\binom{n+m-r}{m} = (n+1)^m$ by (6.37). (The fact that operation E_k can be expressed with complex roots of unity seems to be of no help in this problem.)

7.55 Suppose that $P_0(z)F(z)+\cdots+P_m(z)F^{(m)}(z) = Q_0(z)G(z)+\cdots+Q_n(z)G^{(n)}(z) = 0$, where $P_m(z)$ and $Q_n(z)$ are nonzero. (a) Let $H(z) = F(z)+G(z)$. Then there are rational functions $R_{k,l}(z)$ for $0 \leqslant l < m+n$ such that $H^{(k)}(z) = R_{k,0}(z)F^{(0)}(z)+\cdots+R_{k,m-1}(z)F^{(m-1)}(z)+R_{k,m}(z)G^{(0)}(z)+\cdots+R_{k,m+n-1}(z)G^{(n-1)}(z)$. The $m+n+1$ vectors $\bigl(R_{k,0}(z),\ldots,R_{k,m+n-1}(z)\bigr)$ are linearly dependent in the $(m+n)$-dimensional vector space whose components are rational functions; hence there are rational functions $S_l(z)$, not

all zero, such that $S_0(z)H^{(0)}(z) + \cdots + S_{m+n}(z)H^{(m+n)}(z) = 0$. (b) Similarly, let $H(z) = F(z)G(z)$. There are rational $R_{k,l}(z)$ for $0 \leqslant l < mn$ with $H^{(k)}(z) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} R_{k,ni+j}(z)F^{(i)}(z)G^{(j)}(z)$, hence $S_0(z)H^{(0)}(z) + \cdots + S_{mn}(z)H^{(mn)}(z) = 0$ for some rational $S_l(z)$, not all zero. (A similar proof shows that if $\langle f_n\rangle$ and $\langle g_n\rangle$ are polynomially recursive, so are $\langle f_n + g_n\rangle$ and $\langle f_n g_n\rangle$. Incidentally, there is no similar result for quotients; for example, $\cos z$ is differentiably finite, but $1/\cos z$ is not.)

7.56 Euler [113] showed that this number is also $[z^n]\,1/\sqrt{1-2z-3z^2}$, and he gave the formula $t_n = \sum_{k\geqslant 0} n^{\underline{2k}}/k!^2 = \sum_k \binom{n}{k}\binom{n-k}{k}$. He also discovered a "memorable failure of induction" while examining these numbers: Although $3t_n - t_{n+1}$ is equal to $F_{n-1}(F_{n-1}+1)$ for $0 \leqslant n \leqslant 8$, this empirical law mysteriously breaks down when n is 9 or more! George Andrews [12] has explained the mystery by showing that the sum $\sum_k [z^{n+10k}]\,(1+z+z^2)^n$ can be expressed as a closed form in terms of Fibonacci numbers.

H. S. Wilf observes that $[z^n]\,(a+bz+cz^2)^n = [z^n]\,1/f(z)$, where $f(z) = \sqrt{1-2bz+(b^2-4ac)z^2}$ (see [373, page 159]), and it follows that the coefficients satisfy

$$(n+1)A_{n+1} - (2n+1)bA_n + n(b^2-4ac)A_{n-1} \;=\; 0\,.$$

Give me Legendre polynomials and I'll give you a closed form.

The algorithm of Petkovšek [291] can be used to prove that this recurrence has a closed form solution as a finite sum of hypergeometric terms if and only if $abc(b^2-4ac) = 0$. Therefore in particular, the middle trinomial coefficients have no such closed form. The next step is presumably to extend this result to a larger class of closed forms (including harmonic numbers and/or Stirling numbers, for example).

7.57 (Paul Erdős currently offers \$500 for a solution.)

8.1 $\frac{1}{24}+\frac{1}{48}+\frac{1}{48}+\frac{1}{48}+\frac{1}{48}+\frac{1}{24} = \frac{1}{6}$. (In fact, we *always* get doubles with probability $\frac{1}{6}$ when at least one of the dice is fair.) Any two faces whose sum is 7 have the same probability in distribution $\Pr_1$, so $S = 7$ has the same probability as doubles.

8.2 There are 12 ways to specify the top and bottom cards and 50! ways to arrange the others; so the probability is $12\cdot 50!/52! = 12/(51\cdot 52) = \frac{1}{17\cdot 13} = \frac{1}{221}$.

8.3 $\frac{1}{10}(3+2+\cdots+9+2) = 4.8$; $\frac{1}{9}(3^2+2^2+\cdots+9^2+2^2-10(4.8)^2) = \frac{388}{45}$, which is approximately 8.6. The true mean and variance with a fair coin are 6 and 22, so Stanford had an unusually heads-up class. The corresponding Princeton figures are 6.4 and $\frac{562}{45} \approx 12.5$. (This distribution has $\kappa_4 = 2974$, which is rather large. Hence the standard deviation of this variance estimate when $n = 10$ is also rather large, $\sqrt{2974/10 + 2(22)^2/9} \approx 20.1$ according to exercise 54. One cannot complain that the students cheated.)

8.4 This follows from (8.38) and (8.39), because $F(z) = G(z)H(z)$. (A similar formula holds for all the cumulants, even though $F(z)$ and $G(z)$ may have negative coefficients.)

8.5 Replace H by p and T by $q = 1-p$. If $S_A = S_B = \frac{1}{2}$ we have $p^2qN = \frac{1}{2}$ and $pq^2N = \frac{1}{2}q + \frac{1}{2}$; the solution is $p = 1/\phi^2$, $q = 1/\phi$.

8.6 In this case $X|y$ has the same distribution as X, for all y, hence $E(X|Y) = EX$ is constant and $V\bigl(E(X|Y)\bigr) = 0$. Also $V(X|Y)$ is constant and equal to its expected value.

8.7 We have $1 = (p_1+p_2+\cdots+p_6)^2 \leqslant 6(p_1^2+p_2^2+\cdots+p_6^2)$ by Chebyshev's monotonic inequality of Chapter 2.

8.8 Let $p = \Pr(\omega \in A \cap B)$, $q = \Pr(\omega \notin A)$, and $r = \Pr(\omega \notin B)$. Then $p+q+r = 1$, and the identity to be proved is $p = (p+r)(p+q) - qr$.

8.9 This is true (subject to the obvious proviso that F and G are defined on the respective ranges of X and Y), because

$$\begin{aligned}
\Pr\bigl(F(X)=f \text{ and } G(Y)=g\bigr) &= \sum_{\substack{x\in F^{-1}(f)\\ y\in G^{-1}(g)}} \Pr(X=x \text{ and } Y=y) \\
&= \sum_{\substack{x\in F^{-1}(f)\\ y\in G^{-1}(g)}} \Pr(X=x)\cdot\Pr(Y=y) \\
&= \Pr\bigl(F(X)=f\bigr)\cdot\Pr\bigl(G(y)=g\bigr)\,.
\end{aligned}$$

8.10 Two. Let $x_1 < x_2$ be medians; then $1 \leqslant \Pr(X \leqslant x_1) + \Pr(X \geqslant x_2) \leqslant 1$, hence equality holds. (Some discrete distributions have no median elements. For example, let Ω be the set of all fractions of the form $\pm 1/n$, with $\Pr(+1/n) = \Pr(-1/n) = \frac{\pi^2}{12}n^{-2}$.)

8.11 For example, let $K = k$ with probability $4/(k+1)(k+2)(k+3)$, for all integers $k \geqslant 0$. Then $EK = 1$, but $E(K^2) = \infty$. (Similarly we can construct random variables with finite cumulants through κ_m but with $\kappa_{m+1} = \infty$.)

8.12 (a) Let $p_k = \Pr(X=k)$. If $0 < x \leqslant 1$, we have $\Pr(X \leqslant r) = \sum_{k\leqslant r} p_k \leqslant \sum_{k\leqslant r} x^{k-r}p_k \leqslant \sum_k x^{k-r}p_k = x^{-r}P(x)$. The other inequality has a similar proof. (b) Let $x = \alpha/(1-\alpha)$ to minimize the right-hand side. (A more precise estimate for the given sum is obtained in exercise 9.42.)

8.13 (Solution by Boris Pittel.) Let us set $Y = (X_1 + \cdots + X_n)/n$ and $Z = (X_{n+1} + \cdots + X_{2n})/n$. Then

$$\Pr\biggl(\Bigl|\frac{Y+Z}{2} - \alpha\Bigr| \;\leqslant\; \bigl|Y-\alpha\bigr|\biggr)$$

$$\begin{aligned} &\geqslant \Pr\Bigl(\Bigl|\frac{Y-\alpha}{2}\Bigr| + \Bigl|\frac{Z-\alpha}{2}\Bigr| \leqslant |Y-\alpha|\Bigr) \\ &= \Pr\bigl(|Z-\alpha| \leqslant |Y-\alpha|\bigr) \quad \geqslant \quad \tfrac{1}{2}\,. \end{aligned}$$

The last inequality is, in fact, '>' in any discrete probability distribution, because $\Pr(Y=Z) > 0$.

8.14 $\mathrm{Mean}(H) = p\,\mathrm{Mean}(F) + q\,\mathrm{Mean}(G)$; $\mathrm{Var}(H) = p\,\mathrm{Var}(F) + q\,\mathrm{Var}(G) + pq\bigl(\mathrm{Mean}(F) - \mathrm{Mean}(G)\bigr)^2$. (A mixture is actually a special case of conditional probabilities: Let Y be the coin, let $X|H$ be generated by $F(z)$, and let $X|T$ be generated by $G(z)$. Then $VX = EV(X|Y) + VE(X|Y)$, where $EV(X|Y) = pV(X|H) + qV(X|T)$ and $VE(X|Y)$ is the variance of $pz^{\mathrm{Mean}(F)} + qz^{\mathrm{Mean}(G)}$.)

8.15 By the chain rule, $H'(z) = G'(z)F'\bigl(G(z)\bigr)$; $H''(z) = G''(z)F'\bigl(G(z)\bigr) + G'(z)^2F''\bigl(G(z)\bigr)$. Hence

$$\begin{aligned} \mathrm{Mean}(H) &= \mathrm{Mean}(F)\,\mathrm{Mean}(G)\,; \\ \mathrm{Var}(H) &= \mathrm{Var}(F)\,\mathrm{Mean}(G)^2 + \mathrm{Mean}(F)\,\mathrm{Var}(G)\,. \end{aligned}$$

(The random variable corresponding to probability distribution H can be understood as follows: Determine a nonnegative integer n by distribution F; then add the values of n independent random variables that have distribution G. The identity for variance in this exercise is a special case of (8.106), when X has distribution H and Y has distribution F.)

8.16 $e^{w(z-1)}/(1-w)$.

8.17 $\Pr(Y_{n,p} \leqslant m) = \Pr(Y_{n,p} + n \leqslant m+n) =$ probability that we need $\leqslant m+n$ tosses to obtain n heads $=$ probability that $m+n$ tosses yield $\geqslant n$ heads $= \Pr(X_{m+n,p} \geqslant n)$. Thus

$$\begin{aligned} \sum_{k\leqslant m} \binom{n+k-1}{k} p^n q^k &= \sum_{k\geqslant n} \binom{m+n}{k} p^k q^{m+n-k} \\ &= \sum_{k\leqslant m} \binom{m+n}{k} p^{m+n-k} q^k\,; \end{aligned}$$

and this is (5.19) with $n = r$, $x = q$, $y = p$.

8.18 (a) $G_X(z) = e^{\mu(z-1)}$. (b) The mth cumulant is μ, for all $m \geqslant 1$. (The case $\mu = 1$ is called F_∞ in (8.55).)

8.19 (a) $G_{X_1+X_2}(z) = G_{X_1}(z)G_{X_2}(z) = e^{(\mu_1+\mu_2)(z-1)}$. Hence the probability is $e^{-\mu_1-\mu_2}(\mu_1+\mu_2)^n/n!$; the sum of independent Poisson variables is Poisson. (b) In general, if K_mX denotes the mth cumulant of a random variable X, we have $K_m(aX_1 + bX_2) = a^m(K_mX_1) + b^m(K_mX_2)$, when $a, b \geqslant 0$. Hence the answer is $2^m\mu_1 + 3^m\mu_2$.

8.20 The general pgf will be $G(z) = z^m/F(z)$, where

$$\begin{aligned} F(z) &= z^m + (1-z)\sum_{k=1}^{m} \widetilde{A}_{(k)}[A^{(k)} = A_{(k)}]z^{m-k}\,, \\ F'(1) &= m - \sum_{k=1}^{m} \widetilde{A}_{(k)}[A^{(k)} = A_{(k)}]\,, \\ F''(1) &= m(m-1) - 2\sum_{k=1}^{m}(m-k)\widetilde{A}_{(k)}[A^{(k)} = A_{(k)}]\,. \end{aligned}$$

8.21 This is $\sum_{n\geqslant 0} q_n$, where q_n is the probability that the game between Alice and Bill is still incomplete after n flips. Let p_n be the probability that the game ends at the nth flip; then $p_n + q_n = q_{n-1}$. Hence the average time to play the game is $\sum_{n\geqslant 1} np_n = (q_0 - q_1) + 2(q_1 - q_2) + 3(q_2 - q_3) + \cdots = q_0 + q_1 + q_2 + \cdots = N$, since $\lim_{n\to\infty} nq_n = 0$.

Another way to establish this answer is to replace H and T by $\frac{1}{2}z$. Then the derivative of the first equation in (8.78) tells us that $N(1) + N'(1) = N'(1) + S'_A(1) + S'_B(1)$.

By the way, $N = \frac{16}{3}$.

8.22 By definition we have $V(X|Y) = E(X^2|Y) - \bigl(E(X|Y)\bigr)^2$ and $V\bigl(E(X|Y)\bigr) = E\bigl((E(X|Y))^2\bigr) - \bigl(E\bigl(E(X|Y)\bigr)\bigr)^2$; hence $E\bigl(V(X|Y)\bigr) + V\bigl(E(X|Y)\bigr) = E\bigl(E(X^2|Y)\bigr) - \bigl(E\bigl(E(X|Y)\bigr)\bigr)^2$. But $E\bigl(E(X|Y)\bigr) = \sum_y \Pr(Y=y)E(x|y) = \sum_{x,y} \Pr(Y=y) \times \Pr\bigl((X|y)=x\bigr) = EX$ and $E\bigl(E(X^2|Y)\bigr) = E(X^2)$, so the result is just VX.

8.23 Let $\Omega_0 = \{\boxed{\cdot}, \boxed{\vdots}\}^2$ and $\Omega_1 = \{\boxed{\cdot\cdot}, \boxed{\cdot\cdot}, \boxed{\cdot\cdot}, \boxed{\cdot\cdot}\}^2$; and let Ω_2 be the other 16 elements of Ω. Then $\Pr_{11}(\omega) - \Pr_{00}(\omega) = \frac{+20}{576}, \frac{-7}{576}, \frac{+2}{576}$ according as $\omega \in \Omega_0, \Omega_1, \Omega_2$. The events A must therefore be chosen with k_j elements from Ω_j, where (k_0, k_1, k_2) is one of the following: $(0,0,0)$, $(0,2,7)$, $(0,4,14)$, $(1,4,4)$, $(1,6,11)$, $(2,6,1)$, $(2,8,8)$, $(2,10,15)$, $(3,10,5)$, $(3,12,12)$, $(4,12,2)$, $(4,14,9)$, $(4,16,16)$. For example, there are $\binom{4}{2}\binom{16}{6}\binom{16}{1}$ events of type $(2,6,1)$. The total number of such events is $[z^0](1+z^{20})^4(1+z^{-7})^{16}(1+z^2)^{16}$, which turns out to be 1304872090. If we restrict ourselves to events that depend on S only, we get 40 solutions $S \in A$, where $A = \emptyset$, $\{{2 \atop 12}, {4 \atop 10}, {6 \atop 8}\}$, $\{{2 \atop 12}, 5, 9\}$, $\{2, 12, {4 \atop 10}, {6 \atop 8}, 5, 9\}$, $\{2,4,6,8,10,12\}$, $\{{3 \atop 11}, 7, {5 \atop 9}, 4, 10\}$, and the complements of these sets. (Here the notation '${2 \atop 12}$' means either 2 or 12 but not both.)

8.24 (a) Any one of the dice ends up in J's possession with probability $p = \frac{1}{6} + \bigl(\frac{5}{6}\bigr)^2 p$; hence $p = \frac{6}{11}$. Let $q = \frac{5}{11}$. Then the pgf for J's total holdings is $(q + pz)^{2n+1}$, with mean $(2n+1)p$ and variance $(2n+1)pq$, by (8.61).
(b) $\binom{5}{3}p^3q^2 + \binom{5}{4}p^4q + \binom{5}{5}p^5 = \frac{94176}{161051} \approx .585$.

8.25 The pgf for the current stake after n rolls is $G_n(z)$, where

$$\begin{aligned} G_0(z) &= z^A\,; \\ G_n(z) &= \textstyle\sum_{k=1}^{6} G_{n-1}(z^{2(k-1)/5})/6\,, \qquad \text{for } n>0. \end{aligned}$$

This problem can perhaps be solved more easily without generating functions than with them.

(The noninteger exponents cause no trouble.) It follows that $\mathrm{Mean}(G_n) = \mathrm{Mean}(G_{n-1})$, and $\mathrm{Var}(G_n)+\mathrm{Mean}(G_n)^2 = \frac{22}{15}(\mathrm{Var}(G_{n-1})+\mathrm{Mean}(G_{n-1})^2)$. So the mean is always A, but the variance grows to $\big((\frac{22}{15})^n-1\big)A^2$.

8.26 The pgf $F_{l,n}(z)$ satisfies $F'_{l,n}(z) = F_{l,n-l}(z)/l$; hence $\mathrm{Mean}(F_{l,n}) = F'_{l,n}(1) = [n\geqslant l]/l$ and $F''_{l,n}(1) = [n\geqslant 2l]/l^2$; the variance is easily computed. (In fact, we have

$$F_{l,n}(z) = \sum_{0\leqslant k\leqslant n/l} \frac{1}{k!}\Big(\frac{z-1}{l}\Big)^k,$$

which approaches a Poisson distribution with mean $1/l$ as $n\to\infty$.)

8.27 $(n^2\Sigma_3 - 3n\Sigma_2\Sigma_1 + 2\Sigma_1^3)/n(n-1)(n-2)$ has the desired mean, where $\Sigma_k = X_1^k+\cdots+X_n^k$. This follows from the identities

$$\begin{aligned} E\Sigma_3 &= n\mu_3\,; \\ E(\Sigma_2\Sigma_1) &= n\mu_3 + n(n-1)\mu_2\mu_1\,; \\ E(\Sigma_1^3) &= n\mu_3 + 3n(n-1)\mu_2\mu_1 + n(n-1)(n-2)\mu_1^3\,. \end{aligned}$$

Incidentally, the third cumulant is $\kappa_3 = E\big((X-EX)^3\big)$, but the fourth cumulant does not have such a simple expression; we have $\kappa_4 = E\big((X-EX)^4\big) - 3(VX)^2$.

8.28 (The exercise implicitly calls for $p=q=\frac12$, but the general answer is given here for completeness.) Replace H by pz and T by qz, getting $S_A(z) = p^2qz^3/(1-pz)(1-qz)(1-pqz^2)$ and $S_B(z) = pq^2z^3/(1-qz)(1-pqz^2)$. The pgf for the conditional probability that Alice wins at the nth flip, given that she wins the game, is

$$\frac{S_A(z)}{S_A(1)} = z^3\cdot\frac{q}{1-pz}\cdot\frac{p}{1-qz}\cdot\frac{1-pq}{1-pqz^2}\,.$$

This is a product of pseudo-pgf's, whose mean is $3+p/q+q/p+2pq/(1-pq)$. The formulas for Bill are the same but without the factor $q/(1-pz)$, so Bill's mean is $3+q/p+2pq/(1-pq)$. When $p=q=\frac12$, the answer in case (a) is $\frac{17}{3}$; in case (b) it is $\frac{14}{3}$. Bill wins only half as often, but when he does win he tends to win sooner. The overall average number of flips is $\frac23\cdot\frac{17}{3}+\frac13\cdot\frac{14}{3} = \frac{16}{3}$, agreeing with exercise 21. The solitaire game for each pattern has a waiting time of 8.

8.29 Set $\mathtt{H} = \mathtt{T} = \frac{1}{2}$ in

$$\begin{aligned}
1 + N(\mathtt{H}+\mathtt{T}) &= N + S_A + S_B + S_C \\
N\,\mathtt{HHTH} &= S_A(\mathtt{HTH}+1) + S_B(\mathtt{HTH}+\mathtt{TH}) + S_C(\mathtt{HTH}+\mathtt{TH}) \\
N\,\mathtt{HTHH} &= S_A(\mathtt{THH}+\mathtt{H}) + S_B(\mathtt{THH}+1) + S_C(\mathtt{THH}) \\
N\,\mathtt{THHH} &= S_A(\mathtt{HH}) + S_B(\mathtt{H}) + S_C
\end{aligned}$$

to get the winning probabilities. In general we will have $S_A + S_B + S_C = 1$ and

$$\begin{aligned}
S_A(A{:}A) + S_B(B{:}A) + S_C(C{:}A) &= S_A(A{:}B) + S_B(B{:}B) + S_C(C{:}B) \\
&= S_A(A{:}B) + S_B(B{:}C) + S_C(C{:}C)\,.
\end{aligned}$$

In particular, the equations $9S_A + 3S_B + 3S_C = 5S_A + 9S_B + S_C = 2S_A + 4S_B + 8S_C$ imply that $S_A = \frac{16}{52}$, $S_B = \frac{17}{52}$, $S_C = \frac{19}{52}$.

8.30 The variance of $P(h_1, \ldots, h_n; k)|k$ is the variance of the shifted binomial distribution $\bigl((m-1+z)/m\bigr)^{k-1}z$, which is $(k-1)(\frac{1}{m})(1-\frac{1}{m})$ by (8.61). Hence the average of the variance is $\mathrm{Mean}(S)(m-1)/m^2$. The variance of the average is the variance of $(k-1)/m$, namely $\mathrm{Var}(S)/m^2$. According to (8.106), the sum of these two quantities should be VP, and it is. Indeed, we have just replayed the derivation of (8.96) in slight disguise. (See exercise 15.)

8.31 (a) A brute force solution would set up five equations in five unknowns:

$$\begin{aligned}
A &= \tfrac{1}{2}zB + \tfrac{1}{2}zE\,; \quad B = \tfrac{1}{2}zC\,; \quad C = 1 + \tfrac{1}{2}zB + \tfrac{1}{2}zD\,; \\
D &= \tfrac{1}{2}zC + \tfrac{1}{2}zE\,; \quad E = \tfrac{1}{2}zD\,.
\end{aligned}$$

But positions C and D are equidistant from the goal, as are B and E, so we can lump them together. If $X = B + E$ and $Y = C + D$, there are now three equations:

$$A = \tfrac{1}{2}zX\,; \quad X = \tfrac{1}{2}zY\,; \quad Y = 1 + \tfrac{1}{2}zX + \tfrac{1}{2}zY\,.$$

Hence $A = z^2/(4 - 2z - z^2)$; we have $\mathrm{Mean}(A) = 6$ and $\mathrm{Var}(A) = 22$. (Rings a bell? In fact, this problem is equivalent to flipping a fair coin until getting heads twice in a row: Heads means "advance toward the apple" and tails means "go back.") (b) Chebyshev's inequality says that $\Pr(S \geqslant 100) = \Pr\bigl((S-6)^2 \geqslant 94^2\bigr) \leqslant 22/94^2 \approx .0025$. (c) The second tail inequality says that $\Pr(S \geqslant 100) \leqslant 1/x^{98}(4 - 2x - x^2)$ for all $x \geqslant 1$, and we get the upper bound 0.00000005 when $x = (\sqrt{49001} - 99)/100$. (The actual probability is approximately 0.0000000009, according to exercise 37.)

"Toto, I have a feeling we're not in Kansas anymore."
—Dorothy

8.32 By symmetry, we can reduce each month's situation to one of four possibilities:

D, the states are diagonally opposite;
A, the states are adjacent and not Kansas;
K, the states are Kansas and one other;
S, the states are the same.

Considering the Markovian transitions, we get four equations

$$\begin{aligned} D &= 1 + z(\tfrac{2}{9}D + \tfrac{2}{12}K) \\ A &= z(\tfrac{4}{9}A + \tfrac{4}{12}K) \\ K &= z(\tfrac{4}{9}D + \tfrac{4}{9}A + \tfrac{4}{12}K) \\ S &= z(\tfrac{3}{9}D + \tfrac{1}{9}A + \tfrac{2}{12}K) \end{aligned}$$

whose sum is $D + K + A + S = 1 + z(D + A + K)$. The solution is

$$S = \frac{81z - 45z^2 - 4z^3}{243 - 243z + 24z^2 + 8z^3},$$

but the simplest way to find the mean and variance may be to write $z = 1 + w$ and expand in powers of w, ignoring multiples of w^2:

$$\begin{aligned} D &= \tfrac{27}{16} + \tfrac{1593}{512}w + \cdots; \\ A &= \tfrac{9}{8} + \tfrac{2115}{256}w + \cdots; \\ K &= \tfrac{15}{8} + \tfrac{2661}{256}w + \cdots. \end{aligned}$$

Now $S'(1) = \frac{27}{16} + \frac{9}{8} + \frac{15}{8} = \frac{75}{16}$, and $\frac{1}{2}S''(1) = \frac{1593}{512} + \frac{2115}{256} + \frac{2661}{256} = \frac{11145}{512}$. The mean is $\frac{75}{16}$ and the variance is $\frac{105}{4}$. (Is there a simpler way?)

8.33 First answer: Clearly yes, because the hash values $h_1, \ldots, h_n$ are independent. Second answer: Certainly no, even though the hash values $h_1, \ldots, h_n$ are independent. We have $\Pr(X_j = 0) = \sum_{k=1}^{n} s_k\big([j \neq k](m-1)/m\big) = (1 - s_j)(m-1)/m$, but $\Pr(X_1 = X_2 = 0) = \sum_{k=1}^{n} s_k[k > 2](m-1)^2/m^2 = (1 - s_1 - s_2)(m-1)^2/m^2 \neq \Pr(X_1 = 0)\Pr(X_2 = 0)$.

8.34 Let $[z^n]\,S_m(z)$ be the probability that Gina has advanced $< m$ steps after taking n turns. Then $S_m(1)$ is her average score on a par-m hole; $[z^m]\,S_m(z)$ is the probability that she loses such a hole against a steady player; and $1 - [z^{m-1}]\,S_m(z)$ is the probability that she wins it. We have the recurrence

$$\begin{aligned} S_0(z) &= 0; \\ S_m(z) &= \big(1 + pzS_{m-2}(z) + qzS_{m-1}(z)\big)/(1 - rz), \qquad \text{for } m > 0. \end{aligned}$$

To solve part (a), it suffices to compute the coefficients for $m, n \leqslant 4$; it is convenient to replace z by $100w$ so that the computations involve nothing but integers. We obtain the following tableau of coefficients:

S_0	0	0	0	0	0
S_1	1	4	16	64	256
S_2	1	95	744	4432	23552
S_3	1	100	9065	104044	819808
S_4	1	100	9975	868535	12964304

Therefore Gina wins with probability $1 - .868535 = .131465$; she loses with probability .12964304. (b) To find the mean number of strokes, we compute

$$S_1(1) = \tfrac{25}{24}\,;\quad S_2(1) = \tfrac{4675}{2304}\,;\quad S_3(1) = \tfrac{667825}{221184}\,;\quad S_4(1) = \tfrac{85134475}{21233664}\,.$$

(Incidentally, $S_5(1) \approx 4.9995$; she wins with respect to both holes and strokes on a par-5 hole, but loses either way when par is 3.)

8.35 The condition will be true for all n if and only if it is true for $n = 1$, by the Chinese remainder theorem. One necessary and sufficient condition is the polynomial identity

$$\begin{aligned}&\big(p_2{+}p_4{+}p_6 + (p_1{+}p_3{+}p_5)w\big)\big(p_3{+}p_6 + (p_1{+}p_4)z + (p_2{+}p_5)z^2\big)\\ &\qquad = (p_1wz + p_2z^2 + p_3w + p_4z + p_5wz^2 + p_6)\,,\end{aligned}$$

but that just more-or-less restates the problem. A simpler characterization is

$$(p_2 + p_4 + p_6)(p_3 + p_6) = p_6\,,\qquad (p_1 + p_3 + p_5)(p_2 + p_5) = p_5\,,$$

which checks only two of the coefficients in the former product. The general solution has three degrees of freedom: Let $a_0 + a_1 = b_0 + b_1 + b_2 = 1$, and put $p_1 = a_1b_1$, $p_2 = a_0b_2$, $p_3 = a_1b_0$, $p_4 = a_0b_1$, $p_5 = a_1b_2$, $p_6 = a_0b_0$.

8.36 (a) ⚀ ⚁ ⚂ ⚃ ⚄ ⚅. (b) If the kth die has faces with $s_1, \ldots, s_6$ spots, let $p_k(z) = z^{s_1} + \cdots + z^{s_6}$. We want to find such polynomials with $p_1(z)\ldots p_n(z) = (z + z^2 + z^3 + z^4 + z^5 + z^6)^n$. The irreducible factors of this polynomial with rational coefficients are $z^n(z+1)^n \times (z^2+z+1)^n(z^2-z+1)^n$; hence $p_k(z)$ must be of the form $z^{a_k}(z+1)^{b_k} \times (z^2+z+1)^{c_k}(z^2-z+1)^{d_k}$. We must have $a_k \geqslant 1$, since $p_k(0) = 0$; and in fact $a_k = 1$, since $a_1 + \cdots + a_n = n$. Furthermore the condition $p_k(1) = 6$ implies that $b_k = c_k = 1$. It is now easy to see that $0 \leqslant d_k \leqslant 2$, since $d_k > 2$ gives negative coefficients. When $d = 0$ and $d = 2$, we get the two dice in part (a); therefore the only solutions have k pairs of dice as in (a), plus $n - 2k$ ordinary dice, for some $k \leqslant \frac{1}{2}n$.

8.37 The number of coin-toss sequences of length n is F_{n-1}, for all $n > 0$, because of the relation between domino tilings and coin flips. Therefore the probability that exactly n tosses are needed is $F_{n-1}/2^n$, when the coin is fair. Also $q_n = F_{n+1}/2^{n-1}$, since $\sum_{k\geqslant n} F_k z^k = (F_n z^n + F_{n-1} z^{n+1})/(1-z-z^2)$. (A systematic solution via generating functions is, of course, also possible.)

8.38 When k faces have been seen, the task of rolling a new one is equivalent to flipping coins with success probability $p_k = (m-k)/m$. Hence the pgf is $\prod_{k=0}^{l-1} p_k z/(1-q_k z) = \prod_{k=0}^{l-1}(m-k)z/(m-kz)$. The mean is $\sum_{k=0}^{l-1} p_k^{-1} = m(H_m - H_{m-l})$; the variance is $m^2\bigl(H_m^{(2)} - H_{m-l}^{(2)}\bigr) - m(H_m - H_{m-l})$; and equation (7.47) provides a closed form for the requested probability, namely $m^{-n} m! \left\{ {n-1 \atop l-1} \right\} / (m-l)!$. (The problem discussed in this exercise is traditionally called "coupon collecting.")

8.39 $E(X) = P(-1)$; $V(X) = P(-2) - P(-1)^2$; $E(\ln X) = -P'(0)$.

8.40 (a) We have $\kappa_m = n\bigl(0!\left\{ {m \atop 1} \right\} p - 1!\left\{ {m \atop 2} \right\} p^2 + 2!\left\{ {m \atop 3} \right\} p^3 - \cdots\bigr)$, by (7.49). Incidentally, the third cumulant is $npq(q-p)$ and the fourth is $npq(1-6pq)$. The identity $q+pe^t = (p+qe^{-t})e^t$ shows that $f_m(p) = (-1)^m f_m(q) + [m=1]$; hence we can write $f_m(p) = g_m(pq)(q-p)^{[m \text{ odd}]}$, where g_m is a polynomial of degree $\lfloor m/2 \rfloor$, whenever $m > 1$. (b) Let $p = \frac{1}{2}$ and $F(t) = \ln(\frac{1}{2} + \frac{1}{2}e^t)$. Then $\sum_{m\geqslant 1} \kappa_m t^{m-1}/(m-1)! = F'(t) = 1 - 1/(e^t+1)$, and we can use exercise 6.23.

8.41 If $G(z)$ is the pgf for a random variable X that assumes only positive integer values, then $\int_0^1 G(z)\,dz/z = \sum_{k\geqslant 1} \Pr(X=k)/k = E(X^{-1})$. If X is the distribution of the number of flips to obtain $n+1$ heads, we have $G(z) = \bigl(pz/(1-qz)\bigr)^{n+1}$ by (8.59), and the integral is

$$\int_0^1 \left(\frac{pz}{1-qz}\right)^{n+1} \frac{dz}{z} = \int_0^1 \frac{w^n\,dw}{1+(q/p)w}$$

if we substitute $w = pz/(1-qz)$. When $p = q$ the integrand can be written $(-1)^n\bigl((1+w)^{-1} - 1 + w - w^2 + \cdots + (-1)^n w^{n-1}\bigr)$, so the integral is $(-1)^n\bigl(\ln 2 - 1 + \frac{1}{2} - \frac{1}{3} + \cdots + (-1)^n/n\bigr)$. We have $H_{2n} - H_n = \ln 2 - \frac{1}{4}n^{-1} + \frac{1}{16}n^{-2} + O(n^{-4})$ by (9.28), and it follows that $E(X_{n+1}^{-1}) = \frac{1}{2}n^{-1} - \frac{1}{4}n^{-2} + O(n^{-4})$.

8.42 Let $F_n(z)$ and $G_n(z)$ be pgf's for the number of employed evenings, if the man is initially unemployed or employed, respectively. Let $q_h = 1 - p_h$ and $q_f = 1 - p_f$. Then $F_0(z) = G_0(z) = 1$, and

$$\begin{aligned} F_n(z) &= p_h z G_{n-1}(z) + q_h F_{n-1}(z)\,; \\ G_n(z) &= p_f F_{n-1}(z) + q_f z G_{n-1}(z)\,. \end{aligned}$$

The solution is given by the super generating function

$$G(w,z) \;=\; \sum_{n\geqslant 0} G_n(z)w^n \;=\; A(w)/\bigl(1-zB(w)\bigr)\,,$$

where $B(w) = w\bigl(q_f-(q_f-p_h)w\bigr)/(1-q_hw)$ and $A(w) = \bigl(1-B(w)\bigr)/(1-w)$. Now $\sum_{n\geqslant 0} G_n'(1)w^n = \alpha w/(1-w)^2+\beta/(1-w)-\beta/\bigl(1-(q_f-p_h)w\bigr)$ where

$$\alpha \;=\; \frac{p_h}{p_h+p_f}\,, \qquad \beta \;=\; \frac{p_f(q_f-p_h)}{(p_h+p_f)^2}\,;$$

hence $G_n'(1) = \alpha n+\beta\bigl(1-(q_f-p_h)^n\bigr)$. (Similarly $G_n''(1) = \alpha^2n^2+O(n)$, so the variance is $O(n)$.)

8.43 $G_n(z) = \sum_{k\geqslant 0}\left[{n \atop k}\right]z^k/n! = z^{\overline{n}}/n!$, by (6.11). This is a product of binomial pgf's, $\prod_{k=1}^{n}\bigl((k-1+z)/k\bigr)$, where the kth has mean $1/k$ and variance $(k-1)/k^2$; hence $\mathrm{Mean}(G_n) = H_n$ and $\mathrm{Var}(G_n) = H_n - H_n^{(2)}$.

8.44 (a) The champion must be undefeated in n rounds, so the answer is p^n. (b,c) Players $x_1, \ldots, x_{2^k}$ must be "seeded" (by chance) in distinct subtournaments and they must win all $2^k(n-k)$ of their matches. The 2^n leaves of the tournament tree can be filled in $2^n!$ ways; to seed it we have $2^k!(2^{n-k})^{2^k}$ ways to place the top 2^k players, and $(2^n-2^k)!$ ways to place the others. Hence the probability is $(2p)^{2^k(n-k)}/\binom{2^n}{2^k}$. When $k=1$ this simplifies to $(2p^2)^{n-1}/(2^n-1)$. (d) Each tournament outcome corresponds to a permutation of the players: Let y_1 be the champ; let y_2 be the other finalist; let y_3 and y_4 be the players who lost to y_1 and y_2 in the semifinals; let $(y_5,\ldots,y_8)$ be those who lost respectively to $(y_1,\ldots,y_4)$ in the quarterfinals; etc. (Another proof shows that the first round has $2^n!/2^{n-1}!$ essentially different outcomes; the second round has $2^{n-1}!/2^{n-2}!$; and so on.) (e) Let S_k be the set of 2^{k-1} potential opponents of x_2 in the kth round. The conditional probability that x_2 wins, given that x_1 belongs to S_k, is

$$\begin{aligned}&\Pr(x_1 \text{ plays } x_2)\cdot p^{n-1}(1-p) \;+\; \Pr(x_1 \text{ doesn't play } x_2)\cdot p^n\\ &\qquad = \; p^{k-1}p^{n-1}(1-p) \;+\; (1-p^{k-1})p^n\,.\end{aligned}$$

The chance that $x_1 \in S_k$ is $2^{k-1}/(2^n-1)$; summing on k gives the answer:

$$\sum_{k=1}^{n}\frac{2^{k-1}}{2^n-1}\bigl(p^{k-1}p^{n-1}(1-p)+(1-p^{k-1})p^n\bigr) \;=\; p^n-\frac{(2p)^n-1}{2^n-1}\,p^{n-1}\,.$$

(f) Each of the $2^n!$ tournament outcomes has a certain probability of occurring, and the probability that x_j wins is the sum of these probabilities over all $(2^n-1)!$ tournament outcomes in which x_j is champion. Consider interchanging x_j with x_{j+1} in all those outcomes; this change doesn't affect the

probability if x_j and x_{j+1} never meet, but it multiplies the probability by $(1-p)/p < 1$ if they do meet.

8.45 (a) $A(z) = 1/(3-2z)$; $B(z) = zA(z)^2$; $C(z) = z^2A(z)^3$. The pgf for sherry when it's bottled is $z^3A(z)^3$, which is z^3 times a negative binomial distribution with parameters $n = 3$, $p = \frac{1}{3}$. (b) $\mathrm{Mean}(A) = 2$, $\mathrm{Var}(A) = 6$; $\mathrm{Mean}(B) = 5$, $\mathrm{Var}(B) = 2\,\mathrm{Var}(A) = 12$; $\mathrm{Mean}(C) = 8$, $\mathrm{Var}(C) = 18$. The sherry is nine years old, on the average. The fraction that's 25 years old is $\binom{-3}{22}(-2)^{22}3^{-25} = \binom{24}{22}2^{22}3^{-25} = 23\cdot(\frac{2}{3})^{24} \approx .00137$. (c) Let the coefficient of w^n be the pgf for the beginning of year n. Then

$$\begin{aligned} A &= \big(1+\tfrac{1}{3}w/(1-w)\big)/(1-\tfrac{2}{3}zw)\,; \\ B &= \big(1+\tfrac{1}{3}zwA\big)/(1-\tfrac{2}{3}zw)\,; \\ C &= \big(1+\tfrac{1}{3}zwB\big)/(1-\tfrac{2}{3}zw)\,. \end{aligned}$$

Differentiate with respect to z and set $z = 1$; this makes

$$C' = \frac{8}{1-w} - \frac{1/2}{(1-\frac{2}{3}w)^3} - \frac{3/2}{(1-\frac{2}{3}w)^2} - \frac{6}{1-\frac{2}{3}w}\,.$$

The average age of bottled sherry n years after the process started is 1 greater than the coefficient of w^{n-1}, namely $9-(\frac{2}{3})^n(3n^2+21n+72)/8$. (This already exceeds 8 when $n = 11$.)

8.46 (a) $P(w,z) = 1 + \frac{1}{2}\big(wP(w,z) + zP(w,z)\big) = \big(1-\frac{1}{2}(w+z)\big)^{-1}$, hence $p_{mn} = 2^{-m-n}\binom{m+n}{n}$. (b) $P_k(w,z) = \frac{1}{2}(w^k+z^k)P(w,z)$; hence

$$p_{k,m,n} = 2^{k-1-m-n}\left(\binom{m+n-k}{m} + \binom{m+n-k}{n}\right).$$

(c) $\sum_k kp_{k,n,n} = \sum_{k=0}^n k2^{k-2n}\binom{2n-k}{n} = \sum_{k=0}^n (n-k)2^{-n-k}\binom{n+k}{n}$; this can be summed using (5.20):

$$\begin{aligned} &\sum_{k=0}^n 2^{-n-k}\left((2n+1)\binom{n+k}{n} - (n+1)\binom{n+1+k}{n+1}\right) \\ &\quad = (2n+1) - (n+1)2^{-n}\left(2^{n+1} - 2^{-n-1}\binom{2n+2}{n+1}\right) \\ &\quad = \frac{2n+1}{2^{2n}}\binom{2n}{n} - 1\,. \end{aligned}$$

(The methods of Chapter 9 show that this is $2\sqrt{n/\pi} - 1 + O(n^{-1/2})$.)

8.47 After n irradiations there are $n+2$ equally likely receptors. Let the random variable X_n denote the number of diphages present; then $X_{n+1} =$

$X_n + Y_n$, where $Y_n = -1$ if the $(n+1)$st particle hits a diphage receptor (conditional probability $2X_n/(n+2)$) and $Y_n = +2$ otherwise. Hence

$$EX_{n+1} \;=\; EX_n + EY_n \;=\; EX_n - 2EX_n/(n{+}2) + 2\bigl(1 - 2EX_n/(n{+}2)\bigr)\,.$$

The recurrence $(n{+}2)EX_{n+1} = (n{-}4)EX_n{+}2n{+}4$ can be solved if we multiply both sides by the summation factor $(n+1)^{\underline{5}}$; or we can guess the answer and prove it by induction: $EX_n = (2n+4)/7$ for all $n > 4$. (Incidentally, there are always two diphages and one triphage after five steps, regardless of the configuration after four.)

8.48 (a) The distance between frisbees (measured so as to make it an even number) is either 0, 2, or 4 units, initially 4. The corresponding generating functions A, B, C (where, say, $[z^n]\,C$ is the probability of distance 4 after n throws) satisfy

$$A \;=\; \tfrac14 zB\,, \quad B \;=\; \tfrac12 zB + \tfrac14 zC\,, \quad C \;=\; 1 + \tfrac14 zB + \tfrac34 zC\,.$$

It follows that $A = z^2/(16 - 20z + 5z^2) = z^2/F(z)$, and we have $\mathrm{Mean}(A) = 2 - \mathrm{Mean}(F) = 12$, $\mathrm{Var}(A) = -\mathrm{Var}(F) = 100$. (A more difficult but more amusing solution factors A as follows:

$$A \;=\; \frac{p_1 z}{1-q_1 z}\cdot\frac{p_2 z}{1-q_2 z} \;=\; \frac{p_2}{p_2-p_1}\,\frac{p_1 z}{1-q_1 z} + \frac{p_1}{p_1-p_2}\,\frac{p_2 z}{1-q_2 z}\,,$$

where $p_1 = \phi^2/4 = (3+\sqrt5\,)/8$, $p_2 = \hat\phi^2/4 = (3-\sqrt5\,)/8$, and $p_1 + q_1 = p_2 + q_2 = 1$. Thus, the game is equivalent to having two biased coins whose heads probabilities are p_1 and p_2; flip the coins one at a time until they have both come up heads, and the total number of flips will have the same distribution as the number of frisbee throws. The mean and variance of the waiting times for these two coins are respectively $6 \mp 2\sqrt5$ and $50 \mp 22\sqrt5$, hence the total mean and variance are 12 and 100 as before.)

(b) Expanding the generating function in partial fractions makes it possible to sum the probabilities. (Note that $\sqrt5/(4\phi) + \phi^2/4 = 1$, so the answer can be stated in terms of powers of ϕ.) The game will last more than n steps with probability $5^{(n-1)/2}4^{-n}(\phi^{n+2} - \phi^{-n-2})$; when n is even this is $5^{n/2}4^{-n}F_{n+2}$. So the answer is $5^{50}4^{-100}F_{102} \approx .00006$.

8.49 (a) If $n > 0$, $P_N(0,n) = \frac12[N{=}0] + \frac14 P_{N-1}(0,n) + \frac14 P_{N-1}(1,n{-}1)$; $P_N(m,0)$ is similar; $P_N(0,0) = [N{=}0]$. Hence

$$\begin{aligned} g_{m,n} \;&=\; \tfrac14 z g_{m-1,n+1} + \tfrac12 z g_{m,n} + \tfrac14 z g_{m+1,n-1}\,; \\ g_{0,n} \;&=\; \tfrac12 + \tfrac14 z g_{0,n} + \tfrac14 g_{1,n-1}\,; \quad \text{etc.} \end{aligned}$$

(b) $g'_{m,n} = 1 + \frac14 g'_{m-1,n+1} + \frac12 g'_{m,n} + \frac14 g'_{m+1,n-1}$; $g'_{0,n} = \frac12 + \frac14 g'_{0,n} + \frac14 g'_{1,n-1}$; etc. By induction on m, we have $g'_{m,n} = (2m+1)g'_{0,m+n} - 2m^2$ for all

$m, n \geqslant 0$. And since $g'_{m,0} = g'_{0,m}$, we must have $g'_{m,n} = m + n + 2mn$. (c) The recurrence is satisfied when $mn > 0$, because

$$\sin(2m+1)\theta \;=\; \frac{1}{\cos^2\theta}\biggl(\frac{\sin(2m-1)\theta}{4} + \frac{\sin(2m+1)\theta}{2} + \frac{\sin(2m+3)\theta}{4}\biggr);$$

this is a consequence of the identity $\sin(x-y) + \sin(x+y) = 2\sin x\cos y$. So all that remains is to check the boundary conditions.

8.50 (a) Using the hint, we get

$$\begin{aligned} &3(1-z)^2 \sum_k \binom{1/2}{k}\Bigl(\frac{8}{9}z\Bigr)^k (1-z)^{2-k} \\ &\quad = 3(1-z)^2 \sum_k \binom{1/2}{k}\Bigl(\frac{8}{9}\Bigr)^k \sum_j \binom{k+j-3}{j} z^{j+k}; \end{aligned}$$

now look at the coefficient of z^{3+l}. (b) $H(z) = \frac{2}{3} + \frac{5}{27}z + \frac{1}{2}\sum_{l\geqslant 0} c_{3+l} z^{2+l}$. (c) Let $r = \sqrt{(1-z)(9-z)}$. One can show that $(z-3+r)(z-3-r) = 4z$, and hence that $\bigl(r/(1-z)+2\bigr)^2 = (13-5z+4r)/(1-z) = \bigl(9-H(z)\bigr)/\bigl(1-H(z)\bigr)$. (d) Evaluating the first derivative at $z = 1$ shows that $\mathrm{Mean}(H) = 1$. The second derivative diverges at $z = 1$, so the variance is infinite.

8.51 (a) Let $H_n(z)$ be the pgf for your holdings after n rounds of play, with $H_0(z) = z$. The distribution for n rounds is

$$H_{n+1}(z) \;=\; H_n\bigl(H(z)\bigr),$$

so the result is true by induction (using the amazing identity of the preceding problem). (b) $g_n = H_n(0) - H_{n-1}(0) = 4/n(n+1)(n+2) = 4(n-1)^{\underline{-3}}$. The mean is 2, and the variance is infinite. (c) The expected number of tickets you buy on the nth round is $\mathrm{Mean}(H_n) = 1$, by exercise 15. So the total expected number of tickets is infinite. (Thus, you almost surely lose eventually, and you expect to lose after the second game, yet you also expect to buy an infinite number of tickets.) (d) Now the pgf after n games is $H_n(z)^2$, and the method of part (b) yields a mean of $16 - \frac{4}{3}\pi^2 \approx 2.8$. (The sum $\sum_{k\geqslant 1} 1/k^2 = \pi^2/6$ shows up here.)

8.52 If ω and ω' are events with $\Pr(\omega) > \Pr(\omega')$, then a sequence of n independent experiments will encounter ω more often than ω', with high probability, because ω will occur very nearly $n\Pr(\omega)$ times. Consequently, as $n \to \infty$, the probability approaches 1 that the median or mode of the

values of X in a sequence of independent trials will be a median or mode of the random variable X.

8.53 We can disprove the statement, even in the special case that each variable is 0 or 1. Let $p_0 = \Pr(X{=}Y{=}Z{=}0)$, $p_1 = \Pr(X{=}Y{=}\overline{Z}{=}0)$, $\ldots$, $p_7 = \Pr(\overline{X}{=}\overline{Y}{=}\overline{Z}{=}0)$, where $\overline{X} = 1 - X$. Then $p_0 + p_1 + \cdots + p_7 = 1$, and the variables are independent in pairs if and only if we have

$$\begin{aligned}(p_4+p_5+p_6+p_7)(p_2+p_3+p_6+p_7) &= p_6+p_7\,,\\ (p_4+p_5+p_6+p_7)(p_1+p_3+p_5+p_7) &= p_5+p_7\,,\\ (p_2+p_3+p_6+p_7)(p_1+p_3+p_5+p_7) &= p_3+p_7\,.\end{aligned}$$

But $\Pr(X+Y{=}Z{=}0) \neq \Pr(X+Y{=}0)\Pr(Z{=}0) \iff p_0 \neq (p_0+p_1)(p_0+p_2+p_4+p_6)$. One solution is

$$p_0 = p_3 = p_5 = p_6 = 1/4\,; \qquad p_1 = p_2 = p_4 = p_7 = 0\,.$$

This is equivalent to flipping two fair coins and letting X = (the first coin is heads), Y = (the second coin is heads), Z = (the coins differ). Another example, with all probabilities nonzero, is

$$\begin{aligned}p_0 &= 4/64\,, \quad p_1 = p_2 = p_4 = 5/64\,,\\ p_3 &= p_5 = p_6 = 10/64\,, \quad p_7 = 15/64\,.\end{aligned}$$

For this reason we say that n variables $X_1, \ldots, X_n$ are independent if

$$\Pr(X_1{=}x_1 \text{ and } \cdots \text{ and } X_n{=}x_n) = \Pr(X_1{=}x_1)\ldots\Pr(X_n{=}x_n)\,;$$

pairwise independence isn't enough to guarantee this.

8.54 (See exercise 27 for notation.) We have

$$\begin{aligned}E(\Sigma_2^2) &= n\mu_4 + n(n{-}1)\mu_2^2\,;\\ E(\Sigma_2\Sigma_1^2) &= n\mu_4 + 2n(n{-}1)\mu_3\mu_1 + n(n{-}1)\mu_2^2 + n(n{-}1)(n{-}2)\mu_2\mu_1^2\,;\\ E(\Sigma_1^4) &= n\mu_4 + 4n(n{-}1)\mu_3\mu_1 + 3n(n{-}1)\mu_2^2\\ &\qquad + 6n(n{-}1)(n{-}2)\mu_2\mu_1^2 + n(n{-}1)(n{-}2)(n{-}3)\mu_1^4\,;\end{aligned}$$

it follows that $V(\widehat{V}X) = \kappa_4/n + 2\kappa_2^2/(n-1)$.

8.55 There are $A = \frac{1}{17}\cdot 52!$ permutations with $X = Y$, and $B = \frac{16}{17}\cdot 52!$ permutations with $X \neq Y$. After the stated procedure, each permutation with $X = Y$ occurs with probability $\frac{1}{17}\big/\big((1-\frac{16}{17}p)A\big)$, because we return to step S1 with probability $\frac{16}{17}p$. Similarly, each permutation with $X \neq Y$ occurs with probability $\frac{16}{17}(1-p)\big/\big((1-\frac{16}{17}p)B\big)$. Choosing $p = \frac{1}{4}$ makes $\Pr(X{=}x \text{ and } Y{=}y) = \frac{1}{169}$ for all x and y. (We could therefore make two flips of a fair coin and go back to S1 if both come up heads.)

8.56 If m is even, the frisbees always stay an odd distance apart and the game lasts forever. If $m = 2l + 1$, the relevant generating functions are

$$\begin{aligned} G_m &= \tfrac{1}{4}zA_1\,; \\ A_1 &= \tfrac{1}{2}zA_1 + \tfrac{1}{4}zA_2\,, \\ A_k &= \tfrac{1}{4}zA_{k-1} + \tfrac{1}{2}zA_k + \tfrac{1}{4}zA_{k+1}\,, \qquad \text{for } 1 < k < l, \\ A_l &= \tfrac{1}{4}zA_{l-1} + \tfrac{3}{4}zA_l + 1\,. \end{aligned}$$

(The coefficient $[z^n]\,A_k$ is the probability that the distance between frisbees is $2k$ after n throws.) Taking a clue from the similar equations in exercise 49, we set $z = 1/\cos^2\theta$ and $A_1 = X\sin 2\theta$, where X is to be determined. It follows by induction (not using the equation for A_l) that $A_k = X\sin 2k\theta$. Therefore we want to choose X such that

$$\Bigl(1 - \frac{3}{4\cos^2\theta}\Bigr)\,X\,\sin 2l\theta \;=\; 1 + \frac{1}{4\cos^2\theta}\,X\,\sin(2l-2)\theta\,.$$

It turns out that $X = 2\cos^2\theta/\sin\theta\cos(2l+1)\theta$, hence

$$G_m \;=\; \frac{\cos\theta}{\cos m\theta}\,.$$

The denominator vanishes when θ is an odd multiple of $\pi/(2m)$; thus $1-q_kz$ is a root of the denominator for $1 \leqslant k \leqslant l$, and the stated product representation must hold. To find the mean and variance we can write

Trigonometry wins again. Is there a connection with pitching pennies along the angles of the m-gon?

$$\begin{aligned} G_m &= (1 - \tfrac{1}{2}\theta^2 + \tfrac{1}{24}\theta^4 - \cdots)/(1 - \tfrac{1}{2}m^2\theta^2 + \tfrac{1}{24}m^4\theta^4 - \cdots) \\ &= 1 + \tfrac{1}{2}(m^2-1)\theta^2 + \tfrac{1}{24}(5m^4 - 6m^2 + 1)\theta^4 + \cdots \\ &= 1 + \tfrac{1}{2}(m^2-1)(\tan\theta)^2 + \tfrac{1}{24}(5m^4 - 14m^2 + 9)(\tan\theta)^4 + \cdots \\ &= 1 + G'_m(1)(\tan\theta)^2 + \tfrac{1}{2}G''_m(1)(\tan\theta)^4 + \cdots\,, \end{aligned}$$

because $\tan^2\theta = z - 1$ and $\tan\theta = \theta + \frac{1}{3}\theta^3 + \cdots$. So we have $\mathrm{Mean}(G_m) = \frac{1}{2}(m^2-1)$ and $\mathrm{Var}(G_m) = \frac{1}{6}m^2(m^2-1)$. (Note that this implies the identities

$$\frac{m^2-1}{2} \;=\; \sum_{k=1}^{(m-1)/2} \frac{1}{p_k} \;=\; \sum_{k=1}^{(m-1)/2} \Bigl(1\Big/\sin\frac{(2k-1)\pi}{2m}\Bigr)^2\,;$$

$$\frac{m^2(m^2-1)}{6} \;=\; \sum_{k=1}^{(m-1)/2} \Bigl(\cot\frac{(2k-1)\pi}{2m}\Big/\sin\frac{(2k-1)\pi}{2m}\Bigr)^2\,.$$

The third cumulant of this distribution is $\frac{1}{30}m^2(m^2-1)(4m^2-1)$; but the pattern of nice cumulant factorizations stops there. There's a much simpler

way to derive the mean: We have $G_m + A_1 + \cdots + A_l = z(A_1 + \cdots + A_l) + 1$, hence when $z = 1$ we have $G'_m = A_1 + \cdots + A_l$. Since $G_m = 1$ when $z = 1$, an easy induction shows that $A_k = 4k$.)

8.57 We have $A{:}A \geqslant 2^{l-1}$ and $B{:}B < 2^{l-1} + 2^{l-3}$ and $B{:}A \geqslant 2^{l-2}$, hence $B{:}B - B{:}A \geqslant A{:}A - A{:}B$ is possible only if $A{:}B > 2^{l-3}$. This means that $\overline{\tau}_2 = \tau_3$, $\tau_1 = \tau_4$, $\tau_2 = \tau_5$, $\ldots$, $\tau_{l-3} = \tau_l$. But then $A{:}A \approx 2^{l-1} + 2^{l-4} + \cdots$, $A{:}B \approx 2^{l-3} + 2^{l-6} + \cdots$, $B{:}A \approx 2^{l-2} + 2^{l-5} + \cdots$, and $B{:}B \approx 2^{l-1} + 2^{l-4} + \cdots$; hence $B{:}B - B{:}A$ is less than $A{:}A - A{:}B$ after all. (Sharper results have been obtained by Guibas and Odlyzko [168], who show that Bill's chances are always maximized with one of the two patterns $\mathtt{H}\tau_1 \ldots \tau_{l-1}$ or $\mathtt{T}\tau_1 \ldots \tau_{l-1}$. Bill's winning strategy is, in fact, unique; see the following exercise.)

8.58 (Solution by J. Csirik.) If A is $\mathtt{H}^l$ or $\mathtt{T}^l$, one of the two sequences matches A and cannot be used. Otherwise let $\hat{A} = \tau_1 \ldots \tau_{l-1}$, $H = \mathtt{H}\hat{A}$, and $T = \mathtt{T}\hat{A}$. It is not difficult to verify that $H{:}A = T{:}A = \hat{A}{:}\hat{A}$, $H{:}H + T{:}T = 2^{l-1} + 2(\hat{A}{:}\hat{A}) + 1$, and $A{:}H + A{:}T = 1 + 2(A{:}A) - 2^l$. Therefore the equation

$$\frac{H{:}H - H{:}A}{A{:}A - A{:}H} = \frac{T{:}T - T{:}A}{A{:}A - A{:}T}$$

implies that both fractions equal

$$\frac{H{:}H - H{:}A + T{:}T - T{:}A}{A{:}A - A{:}H + A{:}A - A{:}T} = \frac{2^{l-1} + 1}{2^l - 1}.$$

Then we can rearrange the original fractions to show that

$$\frac{H{:}H - H{:}A}{T{:}T - T{:}A} = \frac{A{:}A - A{:}H}{A{:}A - A{:}T} = \frac{p}{q},$$

where $p \perp q$. And $(p+1)\backslash \gcd(2^{l-1}+1, 2^l - 1) = \gcd(3, 2^l - 1)$; so we may assume that l is even and that $p = 1$, $q = 2$. It follows that $A{:}A - A{:}H = (2^l - 1)/3$ and $A{:}A - A{:}T = (2^{l+1} - 2)/3$, hence $A{:}H - A{:}T = (2^l - 1)/3 \geqslant 2^{l-2}$. We have $A{:}H \geqslant 2^{l-2}$ if and only if $A = (\mathtt{TH})^{l/2}$. But then $H{:}H - H{:}A = A{:}A - A{:}H$, so $2^{l-1} + 1 = 2^l - 1$ and $l = 2$.

(Csirik [69] goes on to show that, when $l \geqslant 4$, Alice can do no better than to play $\mathtt{HT}^{l-3}\mathtt{H}^2$. But even with this strategy, Bill wins with probability nearly $\frac{2}{3}$.)

8.59 According to (8.82), we want $B{:}B - B{:}A > A{:}A - A{:}B$. One solution is $A = \mathtt{TTHH}$, $B = \mathtt{HHH}$.

8.60 (a) Two cases arise depending on whether $h_k \neq h_n$ or $h_k = h_n$:

$$G(w, z) = \frac{m-1}{m}\Big(\frac{m-2+w+z}{m}\Big)^{k-1} w \Big(\frac{m-1+z}{m}\Big)^{n-k-1} z$$

$$+\frac{1}{m}\Big(\frac{m-1+wz}{m}\Big)^{k-1} wz \Big(\frac{m-1+z}{m}\Big)^{n-k-1} z\,.$$

(b) We can either argue algebraically, taking partial derivatives of $G(w,z)$ with respect to w and z and setting $w = z = 1$; or we can argue combinatorially: Whatever the values of $h_1, \ldots, h_{n-1}$, the expected value of $P(h_1,\ldots,h_{n-1},h_n;n)$ is the same (averaged over h_n), because the hash sequence $(h_1,\ldots,h_{n-1})$ determines a sequence of list sizes $(n_1,n_2,\ldots,n_m)$ such that the stated expected value is $\big((n_1+1)+(n_2+1)+\cdots+(n_m+1)\big)/m = (n-1+m)/m$. Therefore the random variable $EP(h_1,\ldots,h_n;n)$ is independent of $(h_1,\ldots,h_{n-1})$, hence independent of $P(h_1,\ldots,h_n;k)$.

8.61 If $1 \leqslant k < l \leqslant n$, the previous exercise shows that the coefficient of $s_k s_l$ in the variance of the average is zero. Therefore we need only consider the coefficient of s_k^2, which is

$$\sum_{1\leqslant h_1,\ldots,h_n\leqslant m} \frac{P(h_1,\ldots,h_n;k)^2}{m^n} - \Bigg(\sum_{1\leqslant h_1,\ldots,h_n\leqslant m} \frac{P(h_1,\ldots,h_n;k)}{m^n}\Bigg)^2,$$

the variance of $\big((m-1+z)/m\big)^{k-1}z$; and this is $(k-1)(m-1)/m^2$ as in exercise 30.

8.62 The pgf $D_n(z)$ satisfies the recurrence

$$\begin{aligned} D_0(z) &= z;\\ D_n(z) &= z^2 D_{n-1}(z) + 2(1-z^3)D'_{n-1}(z)/(n+1)\,, \qquad \text{for } n>0. \end{aligned}$$

We can now derive the recurrence

$$D''_n(1) = (n-11)D''_{n-1}(1)/(n+1) + (8n-2)/7\,,$$

which has the solution $\frac{2}{637}(n+2)(26n+15)$ for all $n \geqslant 11$ (regardless of initial conditions). Hence the variance comes to $\frac{108}{637}(n+2)$ for $n \geqslant 11$.

8.63 (Another question asks if a given sequence of purported cumulants comes from any distribution whatever; for example, κ_2 must be nonnegative, and $\kappa_4 + 3\kappa_2^2 = E\big((X-\mu)^4\big)$ must be at least $\big(E\big((X-\mu)^2\big)\big)^2 = \kappa_2^2$, etc. A necessary and sufficient condition for this other problem was found by Hamburger [6], [175].)

9.1 True if the functions are all positive. But otherwise we might have, say, $f_1(n) = n^3+n^2$, $f_2(n) = -n^3$, $g_1(n) = n^4+n$, $g_2(n) = -n^4$.

9.2 (a) We have $n^{\ln n} \prec c^n \prec (\ln n)^n$, since $(\ln n)^2 \prec n\ln c \prec n\ln\ln n$. (b) $n^{\ln\ln\ln n} \prec (\ln n)! \prec n^{\ln\ln n}$. (c) Take logarithms to show that $(n!)!$ wins. (d) $F^2_{\lceil H_n\rceil} \asymp \phi^{2\ln n} = n^{2\ln\phi}$; $H_{F_n} \sim n\ln\phi$ wins because $\phi^2 = \phi+1 < e$.

9.3 Replacing kn by $O(n)$ requires a different C for each k; but each O stands for a single C. In fact, the context of this O requires it to stand for a set of functions of two variables k and n. It would be correct to write $\sum_{k=1}^{n} kn = \sum_{k=1}^{n} O(n^2) = O(n^3)$.

9.4 For example, $\lim_{n\to\infty} O(1/n) = 0$. On the left, $O(1/n)$ is the set of all functions $f(n)$ such that there are constants C and n_0 with $\bigl|f(n)\bigr| \leqslant C/n$ for all $n \geqslant n_0$. The limit of all functions in that set is 0, so the left-hand side is the singleton set $\{0\}$. On the right, there are no variables; 0 represents $\{0\}$, the (singleton) set of all "functions of no variables, whose value is zero." (Can you see the inherent logic here? If not, come back to it next year; you probably can still manipulate O-notation even if you can't shape your intuitions into rigorous formalisms.)

9.5 Let $f(n) = n^2$ and $g(n) = 1$; then n is in the left set but not in the right, so the statement is false.

9.6 $n \ln n + \gamma n + O(\sqrt{n} \ln n)$.

9.7 $(1 - e^{-1/n})^{-1} = nB_0 - B_1 + B_2 n^{-1}/2! + \cdots = n + \frac{1}{2} + O(n^{-1})$.

9.8 For example, let $f(n) = \lfloor n/2 \rfloor!^2 + n$, $g(n) = \bigl(\lceil n/2 \rceil - 1\bigr)!\,\lceil n/2 \rceil! + n$. These functions, incidentally, satisfy $f(n) = O\bigl(ng(n)\bigr)$ and $g(n) = O\bigl(nf(n)\bigr)$; more extreme examples are clearly possible.

9.9 (For completeness, we assume that there is a side condition $n \to \infty$, so that two constants are implied by each O.) Every function on the left has the form $a(n) + b(n)$, where there exist constants m_0, B, n_0, C such that $\bigl|a(n)\bigr| \leqslant B\bigl|f(n)\bigr|$ for $n \geqslant m_0$ and $\bigl|b(n)\bigr| \leqslant C\bigl|g(n)\bigr|$ for $n \geqslant n_0$. Therefore the left-hand function is at most $\max(B, C)\bigl(\bigl|f(n)\bigr| + \bigl|g(n)\bigr|\bigr)$, for $n \geqslant \max(m_0, n_0)$, so it is a member of the right side.

9.10 If $g(x)$ belongs to the left, so that $g(x) = \cos y$ for some y, where $|y| \leqslant C|x|$ for some C, then $0 \leqslant 1 - g(x) = 2\sin^2(y/2) \leqslant \frac{1}{2}y^2 \leqslant \frac{1}{2}C^2x^2$; hence the set on the left is contained in the set on the right, and the formula is true.

9.11 The proposition is true. For if, say, $|x| \leqslant |y|$, we have $(x+y)^2 \leqslant 4y^2$. Thus $(x+y)^2 = O(x^2) + O(y^2)$. Thus $O(x+y)^2 = O\bigl((x+y)^2\bigr) = O\bigl(O(x^2) + O(y^2)\bigr) = O\bigl(O(x^2)\bigr) + O\bigl(O(y^2)\bigr) = O(x^2) + O(y^2)$.

9.12 $1 + 2/n + O(n^{-2}) = (1 + 2/n)\bigl(1 + O(n^{-2})/(1 + 2/n)\bigr)$ by (9.26), and $1/(1 + 2/n) = O(1)$; now use (9.26).

9.13 $n^n\bigl(1 + 2n^{-1} + O(n^{-2})\bigr)^n = n^n \exp\bigl(n\bigl(2n^{-1} + O(n^{-2})\bigr)\bigr) = e^2 n^n + O(n^{n-1})$.

9.14 It is $n^{n+\beta} \exp\bigl((n + \beta)\bigl(\alpha/n - \frac{1}{2}\alpha^2/n^2 + O(n^{-3})\bigr)\bigr)$.

(It's interesting to compare this formula with the corresponding result for the middle binomial coefficient, exercise 9.60.)

9.15 $\ln\binom{3n}{n,n,n} = 3n\ln 3 - \ln n + \frac{1}{2}\ln 3 - \ln 2\pi + \left(\frac{1}{36} - \frac{1}{4}\right)n^{-1} + O(n^{-3})$, so the answer is

$$\frac{3^{3n+1/2}}{2\pi n}\left(1 - \tfrac{2}{9}n^{-1} + \tfrac{2}{81}n^{-2} + O(n^{-3})\right).$$

9.16 If l is any integer in the range $a \leqslant l < b$ we have

$$\begin{aligned}\int_0^1 B(x)f(l+x)\,dx &= \int_{1/2}^1 B(x)f(l+x)\,dx - \int_0^{1/2} B(1-x)f(l+x)\,dx\\ &= \int_{1/2}^1 B(x)\big(f(l+x) - f(l+1-x)\big)\,dx.\end{aligned}$$

Since $l + x \geqslant l + 1 - x$ when $x \geqslant \frac{1}{2}$, this integral is positive when $f(x)$ is nondecreasing.

9.17 $\sum_{m\geqslant 0} B_m(\frac{1}{2})z^m/m! = ze^{z/2}/(e^z - 1) = z/(e^{z/2} - 1) - z/(e^z - 1)$.

9.18 The text's derivation for the case $\alpha = 1$ generalizes to give

$$b_k(n) = \frac{2^{(2n+1/2)\alpha}}{(2\pi n)^{\alpha/2}}e^{-k^2\alpha/n}, \quad c_k(n) = 2^{2n\alpha}n^{-(1+\alpha)/2+3\epsilon}e^{-k^2\alpha/n};$$

the answer is $2^{2n\alpha}(\pi n)^{(1-\alpha)/2}\alpha^{-1/2}\big(1 + O(n^{-1/2+3\epsilon})\big)$.

9.19 $H_{10} = 2.928968254 \approx 2.928968256$; $10! = 3628800 \approx 3628712.4$; $B_{10} = 0.075757576 \approx 0.075757494$; $\pi(10) = 4 \approx 10.0017845$; $e^{0.1} = 1.10517092 \approx 1.10517083$; $\ln 1.1 = 0.0953102 \approx 0.0953083$; $1.1111111 \approx 1.1111000$; $1.1^{0.1} = 1.00957658 \approx 1.00957643$. (The approximation to $\pi(n)$ gives more significant figures when n is larger; for example, $\pi(10^9) = 50847534 \approx 50840742$.)

9.20 (a) Yes; the left side is $o(n)$ while the right side is equivalent to $O(n)$. (b) Yes; the left side is $e \cdot e^{O(1/n)}$. (c) No; the left side is about $\sqrt{n}$ times the bound on the right.

9.21 We have $P_n = m = n\big(\ln m - 1 - 1/\ln m + O(1/\log n)^2\big)$, where

$$\begin{aligned}\ln m &= \ln n + \ln\ln m - 1/\ln n + \ln\ln n/(\ln n)^2 + O(1/\log n)^2;\\ \ln\ln m &= \ln\ln n + \frac{\ln\ln n}{\ln n} - \frac{(\ln\ln n)^2}{2(\ln n)^2} + \frac{\ln\ln n}{(\ln n)^2} + O(1/\log n)^2.\end{aligned}$$

It follows that

$$\begin{aligned}P_n = n\Bigg(&\ln n + \ln\ln n - 1\\ &+ \frac{\ln\ln n - 2}{\ln n} - \frac{\frac{1}{2}(\ln\ln n)^2 - 3\ln\ln n}{(\ln n)^2} + O(1/\log n)^2\Bigg).\end{aligned}$$

(A slightly better approximation replaces this $O(1/\log n)^2$ by the quantity $-5/(\ln n)^2 + O(\log\log n/\log n)^3$; then we estimate $P_{1000000} \approx 15483612.4$.)

What does a drowning analytic number theorist say?

log log log log ...

9.22 Replace $O(n^{-2k})$ by $-\frac{1}{12}n^{-2k} + O(n^{-4k})$ in the expansion of H_{n^k}; this replaces $O\bigl(\Sigma_3(n^2)\bigr)$ by $-\frac{1}{12}\Sigma_3(n^2) + O\bigl(\Sigma_3(n^4)\bigr)$ in (9.53). We have

$$\Sigma_3(n) \;=\; \tfrac{3}{4}n^{-1} + \tfrac{5}{36}n^{-2} + O(n^{-3})\,,$$

hence the term $O(n^{-2})$ in (9.54) can be replaced by $-\frac{19}{144}n^{-2} + O(n^{-3})$.

9.23 $nh_n = \sum_{0\leqslant k<n} h_k/(n-k) + 2cH_n/(n+1)(n+2)$. Choose $c = e^{\pi^2/6} = \sum_{k\geqslant 0} g_k$ so that $\sum_{k\geqslant 0} h_k = 0$ and $h_n = O(\log n)/n^3$. The expansion of $\sum_{0\leqslant k<n} h_k/(n-k)$ as in (9.60) now yields $nh_n = 2cH_n/(n+1)(n+2) + O(n^{-2})$, hence

$$g_n \;=\; e^{\pi^2/6}\left(\frac{n + 2\ln n + O(1)}{n^3}\right).$$

9.24 (a) If $\sum_{k\geqslant 0}\bigl|f(k)\bigr| < \infty$ and if $f(n-k) = O\bigl(f(n)\bigr)$ when $0 \leqslant k \leqslant n/2$, we have

$$\sum_{k=0}^{n} a_k b_{n-k} \;=\; \sum_{k=0}^{n/2} O\bigl(f(k)\bigr)O\bigl(f(n)\bigr) + \sum_{k=n/2}^{n} O\bigl(f(n)\bigr)O\bigl(f(n-k)\bigr)\,,$$

which is $2O\bigl(f(n)\sum_{k\geqslant 0}\bigl|f(k)\bigr|\bigr)$, so this case is proved. (b) But in this case if $a_n = b_n = \alpha^{-n}$, the convolution $(n+1)\alpha^{-n}$ is not $O(\alpha^{-n})$.

9.25 $S_n/\binom{3n}{n} = \sum_{k=0}^{n} n^{\underline{k}}/(2n+1)^{\overline{k}}$. We may restrict the range of summation to $0 \leqslant k \leqslant (\log n)^2$, say. In this range $n^{\underline{k}} = n^k\bigl(1 - \binom{k}{2}/n + O(k^4/n^2)\bigr)$ and $(2n+1)^{\overline{k}} = (2n)^k\bigl(1 + \binom{k+1}{2}/2n + O(k^4/n^2)\bigr)$, so the summand is

$$\frac{1}{2^k}\left(1 - \frac{3k^2-k}{4n} + O\Bigl(\frac{k^4}{n^2}\Bigr)\right).$$

Hence the sum over k is $2 - 4/n + O(1/n^2)$. Stirling's approximation can now be applied to $\binom{3n}{n} = (3n)!/(2n)!\,n!$, proving (9.2).

9.26 The minimum occurs at a term $B_{2m}/(2m)(2m-1)n^{2m-1}$ where $2m \approx 2\pi n + \frac{3}{2}$, and this term is approximately equal to $1/(\pi e^{2\pi n}\sqrt{n}\,)$. The absolute error in $\ln n!$ is therefore too large to determine $n!$ exactly by rounding to an integer, when n is greater than about $e^{2\pi+1}$.

9.27 We may assume that $\alpha \neq -1$. Let $f(x) = x^\alpha$; the answer is

$$\sum_{k=1}^{n} k^\alpha \;=\; C_\alpha + \frac{n^{\alpha+1}}{\alpha+1} + \frac{n^\alpha}{2} + \sum_{k=1}^{m} \frac{B_{2k}}{2k}\binom{\alpha}{2k-1} n^{\alpha-2k+1} + O(n^{\alpha-2m-1})\,.$$

In particular, $\zeta(0) = -1/2$, *and* $\zeta(-n) = -B_{n+1}/(n+1)$ *for integer* $n > 0$.

(The constant C_α turns out to be $\zeta(-\alpha)$, which is in fact *defined* by this formula when $\alpha > -1$.)

9.28 In general, suppose $f(x) = x^\alpha \ln x$ in Euler's summation formula, when $\alpha \neq -1$. Proceeding as in the previous exercise, we find

$$\begin{aligned}\sum_{k=1}^{n} k^\alpha \ln k \;=\; & C'_\alpha + \frac{n^{\alpha+1}\ln n}{\alpha+1} - \frac{n^{\alpha+1}}{(\alpha+1)^2} + \frac{n^\alpha \ln n}{2} \\ & + \sum_{k=1}^{m} \frac{B_{2k}}{2k}\binom{\alpha}{2k-1} n^{\alpha-2k+1}(\ln n + H_\alpha - H_{\alpha-2k+1}) \\ & + O(n^{\alpha-2m-1}\log n)\,;\end{aligned}$$

the constant C'_α can be shown [74, §3.7] to be $-\zeta'(-\alpha)$. (The $\log n$ factor in the O term can be removed when α is a positive integer $\leqslant 2m$; in that case we also replace the kth term of the right sum by $B_{2k}\alpha!\,(2k-2-\alpha)! \times (-1)^\alpha n^{\alpha-2k+1}/(2k)!$ when $\alpha < 2k-1$.) To solve the stated problem, we let $\alpha = 1$ and $m = 1$, taking the exponential of both sides to get

$$Q_n \;=\; A \cdot n^{n^2/2+n/2+1/12} e^{-n^2/4}\big(1 + O(n^{-2})\big)\,,$$

where $A = e^{1/12-\zeta'(-1)} \approx 1.2824271291$ is "Glaisher's constant."

9.29 Let $f(x) = x^{-1}\ln x$. A slight modification of the calculation in the previous exercise gives

$$\begin{aligned}\sum_{k=1}^{n} \frac{\ln k}{k} \;=\; & \frac{(\ln n)^2}{2} + \gamma_1 + \frac{\ln n}{2n} \\ & - \sum_{k=1}^{m} \frac{B_{2k}}{2k} n^{-2k}(\ln n - H_{2k-1}) + O(n^{-2m-1}\log n)\,,\end{aligned}$$

where $\gamma_1 \approx -0.07281584548367672486$ is a "Stieltjes constant" (see the answer to 9.57). Taking exponentials gives

$$e^{\gamma_1}\sqrt{n^{\ln n}}\left(1 + \frac{\ln n}{2n} + O\Big(\frac{\log n}{n}\Big)^2\right).$$

9.30 Let $g(x) = x^l e^{-x^2}$ and $f(x) = g(x/\sqrt{n}\,)$. Then $n^{-l/2}\sum_{k\geqslant 0} k^l e^{-k^2/n}$ is

$$\begin{aligned}&\int_0^\infty f(x)\,dx - \sum_{k=1}^{m} \frac{B_k}{k!} f^{(k-1)}(0) - (-1)^m \int_0^\infty \frac{B_m(\{x\})}{m!} f^{(m)}(x)\,dx \\ &\quad = n^{1/2}\int_0^\infty g(x)\,dx - \sum_{k=1}^{m} \frac{B_k}{k!} n^{(k-1)/2} g^{(k-1)}(0) + O(n^{-m/2})\,.\end{aligned}$$

Since $g(x) = x^l - x^{2+l}/1! + x^{4+l}/2! - x^{6+l}/3! + \cdots$, the derivatives $g^{(m)}(x)$ obey a simple pattern, and the answer is

$$\frac{1}{2}n^{(l+1)/2}\Gamma\Bigl(\frac{l+1}{2}\Bigr) - \frac{B_{l+1}}{(l+1)!\,0!} + \frac{B_{l+3}n^{-1}}{(l+3)!\,1!} - \frac{B_{l+5}n^{-2}}{(l+5)!\,2!} + O(n^{-3})\,.$$

9.31 The somewhat surprising identity $1/(c^{m-k}+c^m) + 1/(c^{m+k}+c^m) = 1/c^m$ makes the terms for $0 \leqslant k \leqslant 2m$ sum to $(m+\frac{1}{2})/c^m$. The remaining terms are

$$\begin{aligned}\sum_{k\geqslant 1}\frac{1}{c^{2m+k}+c^m} &= \sum_{k\geqslant 1}\biggl(\frac{1}{c^{2m+k}} - \frac{1}{c^{3m+2k}} + \frac{1}{c^{4m+3k}} - \cdots\biggr)\\ &= \frac{1}{c^{2m+1}-c^{2m}} - \frac{1}{c^{3m+2}-c^{3m}} + \cdots\,,\end{aligned}$$

and this series can be truncated at any desired point, with an error not exceeding the first omitted term.

9.32 $H_n^{(2)} = \pi^2/6 - 1/n + O(n^{-2})$ by Euler's summation formula, since we know the constant; and H_n is given by (9.89). So the answer is

$$ne^{\gamma+\pi^2/6}\bigl(1 - \tfrac{1}{2}n^{-1} + O(n^{-2})\bigr)\,.$$

The world's top three constants, (e, π, γ), all appear in this answer.

9.33 We have $n^{\underline{k}}/n^{\overline{k}} = 1 - k(k-1)n^{-1} + \frac{1}{2}k^2(k-1)^2n^{-2} + O(k^6n^{-3})$; dividing by $k!$ and summing over $k \geqslant 0$ yields $e - en^{-1} + \frac{7}{2}en^{-2} + O(n^{-3})$.

9.34 $A = e^\gamma$; $B = 0$; $C = -\frac{1}{2}e^\gamma$; $D = \frac{1}{2}e^\gamma(1-\gamma)$; $E = \frac{1}{8}e^\gamma$; $F = \frac{1}{12}e^\gamma(3\gamma+1)$.

9.35 Since $1/k\bigl(\ln k + O(1)\bigr) = 1/k\ln k + O\bigl(1/k(\log k)^2\bigr)$, the given sum is $\sum_{k=2}^n 1/k\ln k + O(1)$. The remaining sum is $\ln\ln n + O(1)$ by Euler's summation formula.

9.36 This works out beautifully with Euler's summation formula:

$$\begin{aligned}S_n &= \sum_{0\leqslant k<n}\frac{1}{n^2+k^2} + \frac{1}{n^2+x^2}\biggr|_0^n\\ &= \int_0^n\frac{dx}{n^2+x^2} + \frac{1}{2}\frac{1}{n^2+x^2}\biggr|_0^n + \frac{B_2}{2!}\frac{-2x}{(n^2+x^2)^2}\biggr|_0^n + O(n^{-5})\,.\end{aligned}$$

Hence $S_n = \frac{1}{4}\pi n^{-1} - \frac{1}{4}n^{-2} - \frac{1}{24}n^{-3} + O(n^{-5})$.

9.37 This is

$$\sum_{k,q\geqslant 1}(n-qk)\bigl[n/(q+1) < k \leqslant n/q\bigr]$$

$$= n^2 - \sum_{q\geqslant 1} q\left(\binom{\lfloor n/q\rfloor+1}{2} - \binom{\lfloor n/(q+1)\rfloor+1}{2}\right)$$
$$= n^2 - \sum_{q\geqslant 1} \binom{\lfloor n/q\rfloor+1}{2}.$$

The remaining sum is like (9.55) but without the factor $\mu(q)$. The same method works here as it did there, but we get $\zeta(2)$ in place of $1/\zeta(2)$, so the answer comes to $\left(1-\frac{\pi^2}{12}\right)n^2 + O(n\log n)$.

9.38 Replace k by $n-k$ and let $a_k(n) = (n-k)^{n-k}\binom{n}{k}$. Then $\ln a_k(n) = n\ln n - \ln k! - k + O(kn^{-1})$, and we can use tail-exchange with $b_k(n) = n^n e^{-k}/k!$, $c_k(n) = kb_k(n)/n$, $D_n = \{\, k \mid k \leqslant \ln n \,\}$, to get $\sum_{k=0}^n a_k(n) = n^n e^{1/e}\big(1+O(n^{-1})\big)$.

9.39 Tail-exchange with $b_k(n) = (\ln n - k/n - \frac{1}{2}k^2/n^2)(\ln n)^k/k!$, $c_k(n) = n^{-3}(\ln n)^{k+3}/k!$, $D_n = \{\, k \mid 0 \leqslant k \leqslant 10\ln n \,\}$. When $k \approx 10\ln n$ we have $k! \asymp \sqrt{k}\,(10/e)^k(\ln n)^k$, so the kth term is $O(n^{-10\ln(10/e)}\log n)$. The answer is $n\ln n - \ln n - \frac{1}{2}(\ln n)(1+\ln n)/n + O\big(n^{-2}(\log n)^3\big)$.

9.40 Combining terms two by two, we find that $H_{2k}^m - (H_{2k} - \frac{1}{2k})^m = \frac{m}{2k}H_{2k}^{m-1}$ plus terms whose sum over all $k \geqslant 1$ is $O(1)$. Suppose n is even. Euler's summation formula implies that

$$\sum_{k=1}^{n/2} \frac{H_{2k}^{m-1}}{k} = \sum_{k=1}^{n/2} \frac{(\ln 2e^\gamma k)^{m-1} + O(1/k)}{k} + O(1)$$
$$= \frac{(\ln e^\gamma n)^m}{m} + O(1)\,;$$

hence the sum is $\frac{1}{2}H_n^m + O(1)$. In general the answer is $\frac{1}{2}(-1)^n H_n^m + O(1)$.

9.41 Let $\alpha = \hat{\phi}/\phi = -\phi^{-2}$. We have

$$\sum_{k=1}^{n} \ln F_k = \sum_{k=1}^{n} \big(\ln\phi^k - \ln\sqrt{5} + \ln(1-\alpha^k)\big)$$
$$= \frac{n(n+1)}{2}\ln\phi - \frac{n}{2}\ln 5 + \sum_{k\geqslant 1}\ln(1-\alpha^k) - \sum_{k>n}\ln(1-\alpha^k)\,.$$

The latter sum is $\sum_{k>n} O(\alpha^k) = O(\alpha^n)$. Hence the answer is

$$\phi^{n(n+1)/2}5^{-n/2}C + O(\phi^{n(n-3)/2}5^{-n/2})\,,$$

where $C = (1-\alpha)(1-\alpha^2)(1-\alpha^3)\ldots \approx 1.226742$.

9.42 The hint follows since $\binom{n}{k-1}/\binom{n}{k} = \frac{k}{n-k+1} \leqslant \frac{\alpha n}{n-\alpha n+1} < \frac{\alpha}{1-\alpha}$. Let $m = \lfloor \alpha n \rfloor = \alpha n - \epsilon$. Then

$$\begin{aligned}\binom{n}{m} &< \sum_{k\leqslant m}\binom{n}{k}\\ &< \binom{n}{m}\Bigl(1+\frac{\alpha}{1-\alpha}+\Bigl(\frac{\alpha}{1-\alpha}\Bigr)^2+\cdots\Bigr) = \binom{n}{m}\frac{1-\alpha}{1-2\alpha}.\end{aligned}$$

So $\sum_{k\leqslant\alpha n}\binom{n}{k} = \binom{n}{m}O(1)$, and it remains to estimate $\binom{n}{m}$. By Stirling's approximation we have $\ln\binom{n}{m} = -\frac{1}{2}\ln n - (\alpha n-\epsilon)\ln(\alpha-\epsilon/n) - \bigl((1-\alpha)n+\epsilon\bigr)\times \ln(1-\alpha+\epsilon/n) + O(1) = -\frac{1}{2}\ln n - \alpha n\ln\alpha - (1-\alpha)n\ln(1-\alpha) + O(1)$.

9.43 The denominator has factors of the form $z-\omega$, where ω is a complex root of unity. Only the factor $z-1$ occurs with multiplicity 5. Therefore by (7.31), only one of the roots has a coefficient $\Omega(n^4)$, and the coefficient is $c = 5/(5!\cdot 1\cdot 5\cdot 10\cdot 25\cdot 50) = 1/1500000$.

9.44 Stirling's approximation says that $\ln\bigl(x^{-\alpha}x!/(x-\alpha)!\bigr)$ has an asymptotic series

$$\begin{aligned}-\alpha-(x+\tfrac{1}{2}-\alpha)\ln(1-\alpha/x) &- \frac{B_2}{2\cdot 1}\bigl(x^{-1}-(x-\alpha)^{-1}\bigr)\\ &- \frac{B_4}{4\cdot 3}\bigl(x^{-3}-(x-\alpha)^{-3}\bigr)-\cdots\end{aligned}$$

in which each coefficient of x^{-k} is a polynomial in α. Hence $x^{-\alpha}x!/(x-\alpha)! = c_0(\alpha)+c_1(\alpha)x^{-1}+\cdots+c_n(\alpha)x^{-n}+O(x^{-n-1})$ as $x\to\infty$, where $c_n(\alpha)$ is a polynomial in α. We know that $c_n(\alpha) = \left[{\alpha \atop \alpha-n}\right](-1)^n$ whenever α is an integer, and $\left[{\alpha \atop \alpha-n}\right]$ is a polynomial in α of degree $2n$; hence $c_n(\alpha) = \left[{\alpha \atop \alpha-n}\right](-1)^n$ for all real α. In other words, the asymptotic formulas

(See [220] for further discussion.)

$$\begin{aligned}x^{\underline{\alpha}} &= \sum_{k=0}^{n}\left[{\alpha \atop \alpha-k}\right](-1)^k x^{\alpha-k}+O(x^{\alpha-n-1}),\\ x^{\overline{\alpha}} &= \sum_{k=0}^{n}\left[{\alpha \atop \alpha-k}\right]x^{\alpha-k}+O(x^{\alpha-n-1})\end{aligned}$$

generalize equations (6.13) and (6.11), which hold in the all-integer case.

9.45 Let the partial quotients of α be $\langle a_1, a_2, \ldots\rangle$, and let α_m be the continued fraction $1/(a_m+\alpha_{m+1})$ for $m\geqslant 1$. Then $D(\alpha,n) = D(\alpha_1,n) < D\bigl(\alpha_2,\lfloor\alpha_1 n\rfloor\bigr)+a_1+3 < D\bigl(\alpha_3,\lfloor\alpha_2\lfloor\alpha_1 n\rfloor\rfloor\bigr)+a_1+a_2+6 < \cdots < D\bigl(\alpha_{m+1}, \lfloor\alpha_m\lfloor\ldots\lfloor\alpha_1 n\rfloor\ldots\rfloor\rfloor\bigr)+a_1+\cdots+a_m+3m < \alpha_1\ldots\alpha_m n+a_1+\cdots+a_m+3m$,

for all m. Divide by n and let $n \to \infty$; the limit is less than $\alpha_1 \ldots \alpha_m$ for all m. Finally we have

$$\alpha_1 \ldots \alpha_m = \frac{1}{K(a_1, \ldots, a_{m-1}, a_m + \alpha_m)} < \frac{1}{F_{m+1}}.$$

9.46 For convenience we write just m instead of $m(n)$. By Stirling's approximation, the maximum value of $k^n/k!$ occurs when $k \approx m \approx n/\ln n$, so we replace k by $m+k$ and find that

$$\ln \frac{(m+k)^n}{(m+k)!} = n \ln m - m \ln m + m - \frac{\ln 2\pi m}{2} - \frac{(m+n)k^2}{2m^2} + O(k^3 m^{-2} \log n).$$

Actually we want to replace k by $\lfloor m \rfloor + k$; this adds a further $O(km^{-1} \log n)$. The tail-exchange method with $|k| \leqslant m^{1/2+\epsilon}$ now allows us to sum on k, giving a fairly sharp asymptotic estimate in terms of the quantity Θ in (9.93):

A truly Bell-shaped summand.

$$\begin{aligned} \varpi_n &= \frac{e^{m-1} m^{n-m}}{\sqrt{2\pi m}} \big(\Theta_{2m^2/(m+n)} + O(1)\big) \\ &= e^{m-n-1/2} m^n \sqrt{\frac{m}{m+n}} \left(1 + O\Big(\frac{\log n}{n^{1/2}}\Big)\right). \end{aligned}$$

The requested formula follows, with relative error $O(\log\log n/\log n)$.

9.47 Let $\log_m n = l + \theta$, where $0 \leqslant \theta < 1$. The floor sum is $l(n+1) + 1 - (m^{l+1} - 1)/(m-1)$; the ceiling sum is $(l+1)n - (m^{l+1} - 1)/(m-1)$; the exact sum is $(l+\theta)n - n/\ln m + O(\log n)$. Ignoring terms that are $o(n)$, the difference between ceiling and exact is $\big(1 - f(\theta)\big)n$, and the difference between exact and floor is $f(\theta)n$, where

$$f(\theta) = \frac{m^{1-\theta}}{m-1} + \theta - \frac{1}{\ln m}.$$

This function has maximum value $f(0) = f(1) = m/(m-1) - 1/\ln m$, and its minimum value is $\ln\ln m/\ln m + 1 - \big(\ln(m-1)\big)/\ln m$. The ceiling value is closer when n is nearly a power of m, but the floor value is closer when θ lies somewhere between 0 and 1.

9.48 Let $d_k = a_k + b_k$, where a_k counts digits to the left of the decimal point. Then $a_k = 1 + \lfloor \log H_k \rfloor = \log\log k + O(1)$, where 'log' denotes $\log_{10}$. To estimate b_k, let us look at the number of decimal places necessary to distinguish y from nearby numbers $y - \epsilon$ and $y + \epsilon'$: Let $\delta = 10^{-b}$ be the

length of the interval of numbers that round to $\hat{y}$. We have $|y-\hat{y}| \leqslant \frac{1}{2}\delta$; also $y-\epsilon < \hat{y}-\frac{1}{2}\delta$ and $y+\epsilon' > \hat{y}+\frac{1}{2}\delta$. Therefore $\epsilon+\epsilon' > \delta$. And if $\delta < \min(\epsilon,\epsilon')$, the rounding does distinguish $\hat{y}$ from both $y-\epsilon$ and $y+\epsilon'$. Hence $10^{-b_k} < 1/(k-1)+1/k$ and $10^{1-b_k} \geqslant 1/k$; we have $b_k = \log k + O(1)$. Finally, therefore, $\sum_{k=1}^{n} d_k = \sum_{k=1}^{n}(\log k + \log\log k + O(1))$, which is $n\log n + n\log\log n + O(n)$ by Euler's summation formula.

9.49 We have $H_n > \ln n+\gamma+\frac{1}{2}n^{-1}-\frac{1}{12}n^{-2} = f(n)$, where $f(x)$ is increasing for all $x>0$; hence if $n \geqslant e^{\alpha-\gamma}$ we have $H_n \geqslant f(e^{\alpha-\gamma}) > \alpha$. Also $H_{n-1} < \ln n+\gamma-\frac{1}{2}n^{-1} = g(n)$, where $g(x)$ is increasing for all $x>0$; hence if $n \leqslant e^{\alpha-\gamma}$ we have $H_{n-1} \leqslant g(e^{\alpha-\gamma}) < \alpha$. Therefore $H_{n-1} \leqslant \alpha \leqslant H_n$ implies that $e^{\alpha-\gamma}+1 > n > e^{\alpha+\gamma}-1$. (Sharper results have been obtained by Boas and Wrench [33].)

9.50 (a) The expected return is $\sum_{1\leqslant k\leqslant N} k/(k^2H_N^{(2)}) = H_N/H_N^{(2)}$, and we want the asymptotic value to $O(N^{-1})$:

$$\frac{\ln N+\gamma+O(N^{-1})}{\pi^2/6-N^{-1}+O(N^{-2})} = \frac{6\ln 10}{\pi^2}n+\frac{6\gamma}{\pi^2}+\frac{36\ln 10}{\pi^4}\frac{n}{10^n}+O(10^{-n}).$$

The coefficient $(6\ln 10)/\pi^2 \approx 1.3998$ says that we expect about 40% profit.

(b) The probability of profit is $\sum_{n<k\leqslant N} 1/(k^2H_N^{(2)}) = 1-H_n^{(2)}/H_N^{(2)}$, and since $H_n^{(2)} = \frac{\pi^2}{6}-n^{-1}+\frac{1}{2}n^{-2}+O(n^{-3})$ this is

$$\frac{n^{-1}-\frac{1}{2}n^{-2}+O(n^{-3})}{\pi^2/6+O(N^{-1})} = \frac{6}{\pi^2}n^{-1}-\frac{3}{\pi^2}n^{-2}+O(n^{-3}),$$

actually *decreasing* with n. (The expected value in (a) is high because it includes payoffs so huge that the entire world's economy would be affected if they ever had to be made.)

9.51 Strictly speaking, this is false, since the function represented by $O(x^{-2})$ might not be integrable. (It might be '$[x\in S]/x^2$', where S is not a measurable set.) But if we stipulate that $f(x)$ is an integrable function such that $f(x) = O(x^{-2})$ as $x\to\infty$, then $\left|\int_n^\infty f(x)\,dx\right| \leqslant \int_n^\infty |f(x)|\,dx \leqslant \int_n^\infty Cx^{-2}\,dx = Cn^{-1}$. *(As opposed to an execrable function.)*

9.52 In fact, the stack of n's can be replaced by any function $f(n)$ that approaches infinity, however fast. Define the sequence $\langle m_0, m_1, m_2, \ldots\rangle$ by setting $m_0 = 0$ and letting m_k be the least integer $> m_{k-1}$ such that

$$\left(\frac{k+1}{k}\right)^{m_k} \geqslant f(k+1)^2.$$

Now let $A(z) = \sum_{k\geqslant 1}(z/k)^{m_k}$. This power series converges for all z, because the terms for $k>|z|$ are bounded by a geometric series. Also $A(n+1) \geqslant \big((n+1)/n\big)^{m_n} \geqslant f(n+1)^2$, hence $\lim_{n\to\infty} f(n)/A(n) = 0$.

9.53 By induction, the O term is $(m-1)!^{-1}\int_0^x t^{m-1}f^{(m)}(x-t)\,dt$. Since $f^{(m+1)}$ has the opposite sign to $f^{(m)}$, the absolute value of this integral is bounded by $\bigl|f^{(m)}(0)\bigr|\int_0^x t^{m-1}\,dt$; so the error is bounded by the absolute value of the first discarded term.

Sounds like a nasty theorem.

9.54 Let $g(x)=f(x)/x^\alpha$. Then $g'(x)\sim-\alpha g(x)/x$ as $x\to\infty$. By the mean value theorem, $g(x-\frac12)-g(x+\frac12)=-g'(y)\sim\alpha g(y)/y$ for some y between $x-\frac12$ and $x+\frac12$. Now $g(y)=g(x)\bigl(1+O(1/x)\bigr)$, so $g(x-\frac12)-g(x+\frac12)\sim\alpha g(x)/x=\alpha f(x)/x^{1+\alpha}$. Therefore

$$\sum_{k\geqslant n}\frac{f(k)}{k^{1+\alpha}} \;=\; O\Bigl(\sum_{k\geqslant n}\bigl(g(k-\tfrac12)-g(k+\tfrac12)\bigr)\Bigr) \;=\; O\bigl(g(n-\tfrac12)\bigr)\,.$$

9.55 The estimate of $(n+k+\frac12)\ln(1+k/n)+(n-k+\frac12)\ln(1-k/n)$ is extended to $k^2/n+k^4/6n^3+O(n^{-3/2+5\epsilon})$, so we apparently want to have an extra factor $e^{-k^4/6n^3}$ in $b_k(n)$, and $c_k(n)=2^{2n}n^{-2+5\epsilon}e^{-k^2/n}$. But it turns out to be better to leave $b_k(n)$ untouched and to let

$$c_k(n) \;=\; 2^{2n}n^{-2+5\epsilon}e^{-k^2/n}+2^{2n}n^{-5+5\epsilon}k^4e^{-k^2/n}\,,$$

thereby replacing $e^{-k^4/6n^3}$ by $1+O(k^4/n^3)$. The sum $\sum_k k^4e^{-k^2/n}$ is $O(n^{5/2})$, as shown in exercise 30.

9.56 If $k\leqslant n^{1/2+\epsilon}$ we have $\ln(n^{\underline{k}}/n^k)=-\frac12k^2/n+\frac12k/n-\frac16k^3/n^2+O(n^{-1+4\epsilon})$ by Stirling's approximation, hence

$$n^{\underline{k}}/n^k \;=\; e^{-k^2/2n}\bigl(1+k/2n-\tfrac23k^3/(2n)^2+O(n^{-1+4\epsilon})\bigr)\,.$$

Summing with the identity in exercise 30, and remembering to omit the term for $k=0$, gives $-1+\Theta_{2n}+\Theta_{2n}^{(1)}-\frac23\Theta_{2n}^{(3)}+O(n^{-1/2+4\epsilon})=\sqrt{\pi n/2}-\frac13+O(n^{-1/2+4\epsilon})$.

9.57 Using the hint, the given sum becomes $\int_0^\infty ue^{-u}\zeta(1+u/\ln n)\,du$. The zeta function can be defined by the series

$$\zeta(1+z) \;=\; z^{-1}+\sum_{m\geqslant0}(-1)^m\gamma_mz^m/m!\,,$$

where $\gamma_0=\gamma$ and γ_m is the Stieltjes constant [341, 201]

$$\lim_{n\to\infty}\biggl(\sum_{k=1}^n\frac{(\ln k)^m}{k}\;-\;\frac{(\ln n)^{m+1}}{m+1}\biggr)\,.$$

Hence the given sum is

$$\ln n+\gamma-2\gamma_1(\ln n)^{-1}+3\gamma_2(\ln n)^{-2}-\cdots\,.$$

9.58 Let $0 \leqslant \theta \leqslant 1$ and $f(z) = e^{2\pi i z\theta}/(e^{2\pi i z} - 1)$. We have

$$\begin{aligned} \bigl|f(z)\bigr| &= \frac{e^{-2\pi y\theta}}{1+e^{-2\pi y}} \leqslant 1, && \text{when } x \bmod 1 = \tfrac12; \\ \bigl|f(z)\bigr| &\leqslant \frac{e^{-2\pi y\theta}}{|e^{-2\pi y}-1|} \leqslant \frac{1}{1-e^{-2\pi\epsilon}}, && \text{when } |y| \geqslant \epsilon. \end{aligned}$$

Therefore $\bigl|f(z)\bigr|$ is bounded on the contour, and the integral is $O(M^{1-m})$. The residue of $2\pi i f(z)/z^m$ at $z = k \neq 0$ is $e^{2\pi i k\theta}/k^m$; the residue at $z = 0$ is the coefficient of z^{-1} in

$$\frac{e^{2\pi i z\theta}}{z^{m+1}}\Bigl(B_0 + B_1\frac{2\pi i z}{1!} + \cdots\Bigr) = \frac{1}{z^{m+1}}\Bigl(B_0(\theta) + B_1(\theta)\frac{2\pi i z}{1!} + \cdots\Bigr),$$

namely $(2\pi i)^m B_m(\theta)/m!$. Therefore the sum of residues inside the contour is

$$\frac{(2\pi i)^m}{m!}B_m(\theta) + 2\sum_{k=1}^{M} e^{\pi i m/2}\frac{\cos(2\pi k\theta - \pi m/2)}{k^m}.$$

This equals the contour integral $O(M^{1-m})$, so it approaches zero as $M \to \infty$.

9.59 If $F(x)$ is sufficiently well behaved, we have the general identity

$$\sum_k F(k+t) = \sum_n G(2\pi n)e^{2\pi i n t},$$

where $G(y) = \int_{-\infty}^{+\infty} e^{-iyx}F(x)\,dx$. (This is "Poisson's summation formula," which can be found in standard texts such as Henrici [182, Theorem 10.6e].)

9.60 The stated formula is equivalent to

$$n^{\overline{1/2}} = n^{1/2}\Bigl(1 - \frac{1}{8n} + \frac{1}{128n^2} + \frac{5}{1024n^3} - \frac{21}{32768n^4} + O(n^{-5})\Bigr)$$

by exercise 5.22. Hence the result follows from exercises 6.64 and 9.44.

9.61 The idea is to make α "almost" rational. Let $a_k = 2^{2^{2^k}}$ be the kth partial quotient of α, and let $n = \frac12 a_{m+1}q_m$, where $q_m = K(a_1, \ldots, a_m)$ and m is even. Then $0 < \{q_m\alpha\} < 1/K(a_1, \ldots, a_{m+1}) < 1/(2n)$, and if we take $\nu = a_{m+1}/(4n)$ we get a discrepancy $\geqslant \frac14 a_{m+1}$. If this were less than $n^{1-\epsilon}$ we would have $a_{m+1}^{\epsilon} = O(q_m^{1-\epsilon})$; but in fact $a_{m+1} > q_m^{2^m}$.

9.62 See Canfield [48]; see also David and Barton [71, Chapter 16] for asymptotics of Stirling numbers of both kinds.

9.63 Let $c = \phi^{2-\phi}$. The estimate $cn^{\phi-1}+o(n^{\phi-1})$ was proved by Fine [150]. Ilan Vardi observes that the sharper estimate stated can be deduced from the fact that the error term $e(n) = f(n) - cn^{\phi-1}$ satisfies the approximate recurrence $c^{\phi}n^{2-\phi}e(n) \approx -\sum_k e(k)[1 \leqslant k < cn^{\phi-1}]$. The function

$$\frac{n^{\phi-1}u(\ln\ln n/\ln\phi)}{\ln n}$$

satisfies this recurrence asymptotically, if $u(x+1) = -u(x)$. (Vardi conjectures that

$$f(n) \;=\; n^{\phi-1}\Bigl(c + u\Bigl(\frac{\ln\ln n}{\ln\phi}\Bigr)(\ln n)^{-1} + O\bigl((\log n)^{-2}\bigr)\Bigr)$$

for some such function u.) Calculations for small n show that $f(n)$ equals the nearest integer to $cn^{\phi-1}$ for $1 \leqslant n \leqslant 400$ except in one case: $f(273) = 39 > c \cdot 273^{\phi-1} \approx 38.4997$. But the small errors are eventually magnified, because of results like those in exercise 2.36. For example, $e(201636503) \approx 35.73$; $e(919986484788) \approx -1959.07$.

"The paradox is now fully established that the utmost abstractions are the true weapons with which to control our thought of concrete fact."
—A. N. Whitehead [372]

9.64 (From this identity for $B_2(x)$ we can easily derive the identity of exercise 58 by induction on m.) If $0 < x < 1$, the integral $\int_x^{1/2} \sin N\pi t\,dt/\sin\pi t$ can be expressed as a sum of N integrals that are each $O(N^{-2})$, so it is $O(N^{-1})$; the constant implied by this O may depend on x. Integrating the identity $\sum_{n=1}^{N} \cos 2n\pi t = \Re\bigl(e^{2\pi it}(e^{2N\pi it}-1)/(e^{2\pi it}-1)\bigr) = -\frac{1}{2}+\frac{1}{2}\sin(2N+1)\pi t/\sin\pi t$ and letting $N \to \infty$ now gives $\sum_{n\geqslant 1}(\sin 2n\pi x)/n = \frac{\pi}{2} - \pi x$, a relation that Euler knew ([107] and [110, part 2, §92]). Integrating again yields the desired formula. (This solution was suggested by E. M. E. Wermuth [367]; Euler's original derivation did not meet modern standards of rigor.)

9.65 Since $a_0+a_1n^{-1}+a_2n^{-2}+\cdots = 1+(n-1)^{-1}(a_0+a_1(n-1)^{-1}+a_2(n-1)^{-2}+\cdots)$, we obtain the recurrence $a_{m+1} = \sum_k \binom{m}{k} a_k$, which matches the recurrence for the Bell numbers. Hence $a_m = \varpi_m$.

A slightly longer but more informative proof can be based on the fact that $1/(n-1)\ldots(n-m) = \sum_k \left\{ {k \atop m} \right\}/n^k$, by (7.47).

9.66 The expected number of distinct elements in the sequence 1, $f(1)$, $f(f(1))$, $\ldots$, when f is a random mapping of $\{1, 2, \ldots, n\}$ into itself, is the function $Q(n)$ of exercise 56, whose value is $\frac{1}{2}\sqrt{2\pi n}+O(1)$; this might account somehow for the factor $\sqrt{2\pi n}$.

9.67 It is known that $\ln\chi_n \sim \frac{3}{2}n^2\ln\frac{4}{3}$; the constant $e^{-\pi/6}$ has been verified empirically to eight significant digits.

9.68 This would fail if, for example, $e^{n-\gamma} = m+\frac{1}{2}+\epsilon/m$ for some integer m and some $0 < \epsilon < \frac{1}{8}$; but no counterexamples are known.

B

Bibliography

HERE ARE THE WORKS cited in this book. Numbers in the margin specify the page numbers where citations occur.

"This paper fills a much-needed gap in the literature."
— Math. Reviews

References to published problems are generally made to the places where solutions can be found, instead of to the original problem statements.

Wherever possible, names and titles are spelled here as they appeared in the original publication.

1 N. H. Abel, letter to B. Holmboe (1823), in his *Œuvres Complètes*, first edition, 1839, volume 2, 264–265. Reprinted in the second edition, 1881, volume 2, 254–255. *634.*

2 Milton Abramowitz and Irene A. Stegun, editors, *Handbook of Mathematical Functions*. United States Government Printing Office, 1964. Reprinted by Dover, 1965. *42.*

3 William W. Adams and J. L. Davison, "A remarkable class of continued fractions," *Proceedings of the American Mathematical Society* **65** (1977), 194–198. *635.*

4 A. V. Aho and N. J. A. Sloane, "Some doubly exponential sequences," *Fibonacci Quarterly* **11** (1973), 429–437. *633.*

5 W. Ahrens, *Mathematische Unterhaltungen und Spiele*. Teubner, Leipzig, 1901. Second edition, in two volumes, 1910 and 1918. *8.*

6 Naum Il'ich Akhiezer, *Klassicheskaia Problema Momentov i Nekotorye Voprosy Analiza, Sviazannye s Neiu*. Moscow, 1961. English translation, *The Classical Moment Problem and Some Related Questions in Analysis*, Hafner, 1965. *591.*

7 R. E. Allardice and A. Y. Fraser, "La Tour d'Hanoï," *Proceedings of the Edinburgh Mathematical Society* **2** (1884), 50–53. *2.*

8 Désiré André, "Sur les permutations alternées," *Journal de Mathématiques pures et appliquées*, series 3, **7** (1881), 167–184. *635.*

215, 634. **9** George E. Andrews, "Applications of basic hypergeometric functions," *SIAM Review* **16** (1974), 441–484.

530. **10** George E. Andrews, "On sorting two ordered sets," *Discrete Mathematics* **11** (1975), 97–106.

330. **11** George E. Andrews, *The Theory of Partitions.* Addison-Wesley, 1976.

575. **12** George E. Andrews, "Euler's 'exemplum memorabile inductionis fallacis' and q-trinomial coefficients," *Journal of the American Mathematical Society* **3** (1990), 653–669.

635. **13** George E. Andrews and K. Uchimura, "Identities in combinatorics IV: Differentiation and harmonic numbers," *Utilitas Mathematica* **28** (1985), 265–269.

238, 634. **14** Roger Apéry, "Interpolation de fractions continues et irrationalité de certaines constantes," in *Mathématiques*, Ministère des universités (France), Comité des travaux historiques et scientifiques, Section des sciences, *Bulletin de la Section des Sciences* **3** (1981), 37–53.

635. **15** V. I. Arnold, "Bernoulli-Euler updown numbers associated with function singularities, their combinatorics and arithmetics," *Duke Mathematical Journal* **63** (1991), 537–555.

633. **16** M. D. Atkinson, "The cyclic towers of Hanoi," *Information Processing Letters* **13** (1981), 118–119.

443. **17** Paul Bachmann, *Die analytische Zahlentheorie.* Teubner, Leipzig, 1894.

223, 634. **18** W. N. Bailey, *Generalized Hypergeometric Series.* Cambridge University Press, 1935; second edition, 1964.

548. **19** W. N. Bailey, "The generating function for Jacobi polynomials," *Journal of the London Mathematical Society* **13** (1938), 243–246.

633. **20** W. W. Rouse Ball and H. S. M. Coxeter, *Mathematical Recreations and Essays*, twelfth edition. University of Toronto Press, 1974. (A revision of Ball's *Mathematical Recreations and Problems*, first published by Macmillan, 1892.)

634. **21** P. Barlow, "Demonstration of a curious numerical proposition," *Journal of Natural Philosophy, Chemistry, and the Arts* **27** (1810), 193–205.

633. **22** Samuel Beatty, "Problem 3177," *American Mathematical Monthly* **34** (1927), 159–160.

332. **23** E. T. Bell, "Euler algebra," *Transactions of the American Mathematical Society* **25** (1923), 135–154.

635. **24** E. T. Bell, "Exponential numbers," *American Mathematical Monthly* **41** (1934), 411–419.

25 Edward A. Bender, "Asymptotic methods in enumeration," *SIAM Review* **16** (1974), 485–515. *636.*

26 Jacobi Bernoulli, *Ars Conjectandi*, opus posthumum. Basel, 1713. Reprinted in *Die Werke von Jakob Bernoulli*, volume 3, 107–286. *283.*

27 J. Bertrand, "Mémoire sur le nombre de valeurs que peut prendre une fonction quand on y permute les lettres qu'elle renferme," *Journal de l'École Royale Polytechnique* **18**, cahier 30 (1845), 123–140. *633.*

28 William H. Beyer, editor, *CRC Standard Mathematical Tables and Formulae*, 29th edition. CRC Press, Boca Raton, Florida, 1991. *42.*

29 J. Bienaymé, "Considérations à l'appui de la découverte de Laplace sur la loi de probabilité dans la méthode des moindres carrés," *Comptes Rendus hebdomadaires des séances de l'Académie des Sciences* (Paris) **37** (1853), 309–324. *390.*

30 J. Binet, "Mémoire sur un système de Formules analytiques, et leur application à des considérations géométriques," *Journal de l'École Polytechnique* **9**, cahier 16 (1812), 280–354. *633.*

31 J. Binet, "Mémoire sur l'intégration des équations linéaires aux différences finies, d'un ordre quelconque, à coefficients variables," *Comptes Rendus hebdomadaires des séances de l'Académie des Sciences* (Paris) **17** (1843), 559–567. *299.*

32 Gunnar Blom, "Problem E 3043: Random walk until no shoes," *American Mathematical Monthly* **94** (1987), 78–79. *636.*

33 R. P. Boas, Jr. and J. W. Wrench, Jr., "Partial sums of the harmonic series," *American Mathematical Monthly* **78** (1971), 864–870. *600, 636.*

34 P. Bohl, "Über ein in der Theorie der säkularen Störungen vorkommendes Problem," *Journal für die reine und angewandte Mathematik* **135** (1909), 189–283. *87.*

35 Émile Borel, *Leçons sur les séries à termes positifs*. Gauthier-Villars, 1902. *636.*

36 Jonathan M. Borwein and Peter B. Borwein, *Pi and the AGM*. Wiley, 1987. *635.*

37 Richard P. Brent, "The first occurrence of large gaps between successive primes," *Mathematics of Computation* **27** (1973), 959–963. *525.*

38 Richard P. Brent, "Computation of the regular continued fraction for Euler's constant," *Mathematics of Computation* **31** (1977), 771–777. *306, 564.*

39 John Brillhart, "Some miscellaneous factorizations," *Mathematics of Computation* **17** (1963), 447–450. *633.*

116. **40** Achille Brocot, "Calcul des rouages par approximation, nouvelle méthode," *Revue Chronométrique* **6** (1860), 186–194. (He also published a 97-page monograph with the same title in 1862.)

635. **41** Maxey Brooke and C. R. Wall, "Problem B-14: A little surprise," *Fibonacci Quarterly* **1**, 3 (1963), 80.

633. **42** Brother U. Alfred [Brousseau], "A mathematician's progress," *Mathematics Teacher* **59** (1966), 722–727.

501. **43** Morton Brown, "Problem 6439: A periodic sequence," *American Mathematical Monthly* **92** (1985), 218.

(Such papers aren't cited in this book.) **44** T. Brown, "Infinite multi-variable subpolynormal Woffles which do not satisfy the lower regular Q-property (Piffles)," in *A Collection of 250 Papers on Woffle Theory Dedicated to R. S. Green on His 23rd Birthday*. Cited in A. K. Austin, "Modern research in mathematics," *The Mathematical Gazette* **51** (1967), 149–150.

633. **45** Thomas C. Brown, "Problem E 2619: Squares in a recursive sequence," *American Mathematical Monthly* **85** (1978), 52–53.

358. **46** William G. Brown, "Historical note on a recurrent combinatorial problem," *American Mathematical Monthly* **72** (1965), 973–977.

635. **47** S. A. Burr, "On moduli for which the Fibonacci sequence contains a complete system of residues," *Fibonacci Quarterly* **9** (1971), 497–504.

602, 636. **48** E. Rodney Canfield, "On the location of the maximum Stirling number(s) of the second kind," *Studies in Applied Mathematics* **59** (1978), 83–93.

635. **49** L. Carlitz, "The generating function for $\max(n_1, n_2, \cdots, n_k)$, *Portugaliae Mathematica* **21** (1962), 201–207.

31. **50** Lewis Carroll [pseudonym of C. L. Dodgson], *Through the Looking Glass and What Alice Found There*. Macmillan, 1871.

292. **51** Jean-Dominique Cassini, "Une nouvelle progression de nombres," *Histoire de l'Académie Royale des Sciences*, Paris, volume 1, 201. (Cassini's work is summarized here as one of the mathematical results presented to the academy in 1680. This volume was published in 1733.)

203. **52** E. Catalan, "Note sur une Équation aux différences finies," *Journal de Mathématiques pures et appliquées* **3** (1838), 508–516.

633. **53** Augustin-Louis Cauchy, *Cours d'analyse de l'École Royale Polytechnique*. Imprimerie Royale, Paris, 1821. Reprinted in his *Œuvres Complètes*, series 2, volume 3.

54 Arnold Buffum Chace, *The Rhind Mathematical Papyrus*, volume 1. Mathematical Association of America, 1927. (Includes an excellent bibliography of Egyptian mathematics by R. C. Archibald.) *633.*

55 M. Chaimovich, G. Freiman, and J. Schönheim, "On exceptions to Szegedy's theorem," *Acta Arithmetica* **49** (1987), 107–112. *525.*

56 P. L. Tchebichef [Chebyshev], "Mémoire sur les nombres premiers," *Journal de Mathématiques pures et appliquées* **17** (1852), 366–390. Reprinted in his *Œuvres*, volume 1, 51–70. Russian translation, "O prostykh chislakh," in his *Polnoe Sobranie Sochineniĭ*, volume 1, 191–207. *633.*

57 P. L. Chebyshev, "O srednikh velichinakh," *Matematicheskiĭ Sbornik'* **2** (1867), 1–9. Reprinted in his *Polnoe Sobranie Sochineniĭ*, volume 2, 431–437. French translation, "Des valeurs moyennes," *Journal de Mathématiques pures et appliquées*, series 2, **12** (1867), 177–184; reprinted in his *Œuvres*, volume 1, 685–694. *390.*

58 P. L. Chebyshev, "O priblizhennykh vyrazheniia͡kh odnikh integralov cherez drugie, vzia͡tye v tex zhe predelakh," *Soobshchenīia͡ i protokoly zasie͡danīĭ matematicheskago obshchestva pri Imperatorskum″ Khar'kovskom″ Universitetie͡* **4**,2 (1882), 93–98. Reprinted in his *Polnoe Sobranie Sochineniĭ*, volume 3, 128–131. French translation, "Sur les expressions approximatives des intégrales définies par les autres prises entre les mêmes limites," in his *Œuvres*, volume 2, 716–719. *38.*

59 Fan Chung and Ron Graham, "On digraph polynomials," submitted for publication, 1993. *557, 635.*

60 Th. Clausen, "Ueber die Fälle, wenn die Reihe von der Form *634.*

$$y = 1 + \frac{\alpha}{1} \cdot \frac{\beta}{\gamma} x + \frac{\alpha.\alpha+1}{1.2} \cdot \frac{\beta.\beta+1}{\gamma.\gamma+1} x^2 + \text{etc.}$$

ein Quadrat von der Form

$$z = 1 + \frac{\alpha'}{1} \cdot \frac{\beta'}{\gamma'} \cdot \frac{\delta'}{\epsilon'} x + \frac{\alpha'.\alpha'+1}{1.2} \cdot \frac{\beta'.\beta'+1}{\gamma'.\gamma'+1} \cdot \frac{\delta'.\delta'+1}{\epsilon'.\epsilon'+1} x^2 + \text{etc.} \text{ hat,"}$$

Journal für die reine und angewandte Mathematik **3** (1828), 89–91.

61 Th. Clausen, "Beitrag zur Theorie der Reihen," *Journal für die reine und angewandte Mathematik* **3** (1828), 92–95. *634.*

62 Th. Clausen, "Theorem," *Astronomische Nachrichten* **17** (1840), columns 351–352. *635.*

63 Stuart Dodgson Collingwood, *The Lewis Carroll Picture Book*. T. Fisher Unwin, 1899. Reprinted by Dover, 1961, with the new title *Diversions and Digressions of Lewis Carroll.* *293.*

636. **64** Louis Comtet, *Advanced Combinatorics.* Dordrecht, Reidel, 1974.

501. **65** J. H. Conway and R. L. Graham, "Problem E 2567: A periodic recurrence," *American Mathematical Monthly* **84** (1977), 570–571.

525, 634. **66** Harald Cramér, "On the order of magnitude of the difference between consecutive prime numbers," *Acta Arithmetica* **2** (1937), 23–46.

633. **67** A. L. Crelle, "Démonstration élémentaire du théorème de Wilson généralisé," *Journal für die reine und angewandte Mathematik* **20** (1840), 29–56.

633. **68** D. W. Crowe, "The n-dimensional cube and the Tower of Hanoi," *American Mathematical Monthly* **63** (1956), 29–30.

590. **69** János A. Csirik, "Optimal strategy for the first player in the Penney ante game," *Combinatorics, Probability and Computing* **1** (1992), 311–321.

634. **70** D. R. Curtiss, "On Kellogg's Diophantine problem," *American Mathematical Monthly* **29** (1922), 380–387.

602. **71** F. N. David and D. E. Barton, *Combinatorial Chance.* Hafner, 1962.

210. **72** Philip J. Davis, "Leonhard Euler's integral: A historical profile of the Gamma function," *American Mathematical Monthly* **66** (1959), 849–869.

307, 635. **73** J. L. Davison, "A series and its associated continued fraction," *Proceedings of the American Mathematical Society* **63** (1977), 29–32.

444, 447, 595, 636. **74** N. G. de Bruijn, *Asymptotic Methods in Analysis.* North-Holland, 1958; third edition, 1970. Reprinted by Dover, 1981.

635. **75** N. G. de Bruijn, "Problem 9," *Nieuw Archief voor Wiskunde*, series 3, **12** (1964), 68.

297, 481. **76** Abraham de Moivre, *Miscellanea analytica de seriebus et quadraturis.* London, 1730.

136. **77** R. Dedekind, "Abriß einer Theorie der höheren Congruenzen in Bezug auf einen reellen Primzahl-Modulus," *Journal für die reine und angewandte Mathematik* **54** (1857), 1–26. Reprinted in his *Gesammelte mathematische Werke*, volume 1, 40–67.

510. **78** Leonard Eugene Dickson, *History of the Theory of Numbers.* Carnegie Institution of Washington, volume 1, 1919; volume 2, 1920; volume 3, 1923. Reprinted by Stechert, 1934, and by Chelsea, 1952, 1971.

635. **79** Edsger W. Dijkstra, *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, 1982.

80 G. Lejeune Dirichlet, "Verallgemeinerung eines Satzes aus der Lehre von den Kettenbrüchen nebst einigen Anwendungen auf die Theorie der Zahlen," *Bericht über die Verhandlungen der Königlich-Preußischen Akademie der Wissenschaften zu Berlin* (1842), 93–95. Reprinted in his *Werke*, volume 1, 635–638. *633.*

81 A. C. Dixon, "On the sum of the cubes of the coefficients in a certain expansion by the binomial theorem," *Messenger of Mathematics* **20** (1891), 79–80. *634.*

82 John Dougall, "On Vandermonde's theorem, and some more general expansions," *Proceedings of the Edinburgh Mathematical Society* **25** (1907), 114–132. *171.*

83 A. Conan Doyle, "The sign of the four; or, The problem of the Sholtos," *Lippincott's Monthly Magazine* (Philadelphia) **45** (1890), 147–223. *228, 405.*

84 A. Conan Doyle, "The adventure of the final problem," *The Strand Magazine* **6** (1893), 558–570. *162.*

85 P. du Bois-Reymond, "Sur la grandeur relative des infinis des fonctions," *Annali di Matematica pura ed applicata*, series 2, **4** (1871), 338–353. *440.*

86 Harvey Dubner, "Generalized repunit primes," *Mathematics of Computation* **61** (1993), 927–930. *633.*

87 Henry Ernest Dudeney, *The Canterbury Puzzles and Other Curious Problems*. E. P. Dutton, New York, 1908; 4th edition, Dover, 1958. (Dudeney had first considered the generalized Tower of Hanoi in *The Weekly Dispatch*, on 25 May 1902 and 15 March 1903.) *633.*

88 G. Waldo Dunnington, *Carl Friedrich Gauss: Titan of Science*. Exposition Press, New York, 1955. *6.*

89 F. J. Dyson, "Some guesses in the theory of partitions," *Eureka* **8** (1944), 10–15. *172, 239.*

90 A. W. F. Edwards, *Pascal's Arithmetical Triangle*. Oxford University Press, 1987. *155.*

91 G. Eisenstein, "Entwicklung von $\alpha^{\alpha^{\alpha^{\cdot^{\cdot^{\cdot}}}}}$," *Journal für die reine und angewandte Mathematik* **28** (1844), 49–52. Reprinted in his *Mathematische Werke* **1**, 122–125. *202.*

92 Noam D. Elkies, "On $A^4+B^4+C^4=D^4$," *Mathematics of Computation* **51** (1988), 825–835. *131.*

93 Erdős Pál, "Az $\frac{1}{x_1}+\frac{1}{x_2}+\cdots+\frac{1}{x_n}=\frac{a}{b}$ egyenlet egész számú megoldásairól," *Matematikai Lapok* **1** (1950), 192–209. English abstract on page 210. *634.*

418. **94** Paul Erdős, "My Scottish Book 'problems'," in *The Scottish Book: Mathematics from the Scottish Café*, edited by R. Daniel Mauldin, 1981, 35–45.

515, 525, 634, 635, 636. **95** P. Erdős and R. L. Graham, *Old and New Problems and Results in Combinatorial Number Theory*. Université de Genève, L'Enseignement Mathématique, 1980.

525, 548. **96** P. Erdős, R. L. Graham, I. Z. Ruzsa, and E. G. Straus, "On the prime factors of $\binom{2n}{n}$," *Mathematics of Computation* **29** (1975), 83–92.

635. **97** Arulappah Eswarathasan and Eugene Levine, "p-integral harmonic sums," *Discrete Mathematics* **91** (1991), 249–257.

108. **98** Euclid, *ΣΤΟΙΧΕΙΑ*. Ancient manuscript first printed in Basel, 1533. Scholarly edition (Greek and Latin) by J. L. Heiberg in five volumes, Teubner, Leipzig, 1883–1888.

210, 634. **99** Leonhard Euler, letter to Christian Goldbach (13 October 1729), in *Correspondance Mathématique et Physique de Quelques Célèbres Géomètres du XVIIIème Siècle*, edited by P. H. Fuss, St. Petersburg, 1843, volume 1, 3–7.

210. **100** L. Eulero, "De progressionibus transcendentibus seu quarum termini generales algebraice dari nequeunt," *Commentarii academiæ scientiarum imperialis Petropolitanæ* **5** (1730), 36–57. Reprinted in his *Opera Omnia*, series 1, volume 14, 1–24.

469. **101** Leonh. Eulero, "Methodus generalis summandi progressiones," *Commentarii academiæ scientiarum imperialis Petropolitanæ* **6** (1732), 68–97. Reprinted in his *Opera Omnia*, series 1, volume 14, 42–72.

132. **102** Leonh. Eulero, "Observationes de theoremate quodam Fermatiano, aliisque ad numeros primos spectantibus," *Commentarii academiæ scientiarum imperialis Petropolitanæ* **6** (1732), 103–107. Reprinted in his *Opera Omnia*, series 1, volume 2, 1–5. Reprinted in his *Commentationes arithmeticæ collectæ*, volume 1, 1–3.

278. **103** Leonh. Eulero, "De progressionibus harmonicis observationes," *Commentarii academiæ scientiarum imperialis Petropolitanæ* **7** (1734), 150–161. Reprinted in his *Opera Omnia*, series 1, volume 14, 87–100.

267. **104** Leonh. Eulero, "Methodus universalis series summandi ulterius promota," *Commentarii academiæ scientiarum imperialis Petropolitanæ* **8** (1736), 147–158. Reprinted in his *Opera Omnia*, series 1, volume 14, 124–137.

122. **105** Leonh. Euler, "De fractionibus continuis, Dissertatio," *Commentarii academiæ scientiarum imperialis Petropolitanæ* **9** (1737), 98–137. Reprinted in his *Opera Omnia*, series 1, volume 14, 187–215.

106 Leonh. Euler, "Variæ observationes circa series infinitas," *Commentarii academiæ scientiarum imperialis Petropolitanæ* **9** (1737), 160–188. Reprinted in his *Opera Omnia*, series 1, volume 14, 216–244. *633.*

107 Leonhard Euler, letter to Christian Goldbach (4 July 1744), in *Correspondance Mathématique et Physique de Quelques Célèbres Géomètres du XVIIIème Siècle*, edited by P. H. Fuss, St. Petersburg, 1843, volume 1, 278–293. *603.*

108 Leonhardo Eulero, *Introductio in Analysin Infinitorum.* Tomus primus, Lausanne, 1748. Reprinted in his *Opera Omnia*, series 1, volume 8. Translated into French, 1786; German, 1788; Russian, 1936. *635.*

109 L. Eulero, "De partitione numerorum," *Novi commentarii academiæ scientiarum imperialis Petropolitanæ* **3** (1750), 125–169. Reprinted in his *Commentationes arithmeticæ collectæ*, volume 1, 73–101. Reprinted in his *Opera Omnia*, series 1, volume 2, 254–294. *635.*

110 Leonhardo Eulero, *Institutiones Calculi Differentialis cum eius usu in Analysi Finitorum ac Doctrina Serierum.* Berlin, Academiæ Imperialis Scientiarum Petropolitanæ, 1755. Reprinted in his *Opera Omnia*, series 1, volume 10. Translated into German, 1790. *48, 267, 603, 635.*

111 L. Eulero, "Theoremata arithmetica nova methodo demonstrata," *Novi commentarii academiæ scientiarum imperialis Petropolitanæ* **8** (1760), 74–104. (Also presented in 1758 to the Berlin Academy.) Reprinted in his *Commentationes arithmeticæ collectæ*, volume 1, 274–286. Reprinted in his *Opera Omnia*, series 1, volume 2, 531–555. *133, 134.*

112 L. Eulero, "Specimen algorithmi singularis," *Novi commentarii academiæ scientiarum imperialis Petropolitanæ* **9** (1762), 53–69. (Also presented in 1757 to the Berlin Academy.) Reprinted in his *Opera Omnia*, series 1, volume 15, 31–49. *303.*

113 L. Eulero, "Observationes analyticæ," *Novi commentarii academiæ scientiarum imperialis Petropolitanæ* **11** (1765), 124–143. Reprinted in his *Opera Omnia*, series 1, volume 15, 50–69. *299, 575, 636.*

114 Leonhard Euler, *Vollständige Anleitung zur Algebra. Erster Theil. Von den verschiedenen Rechnungs-Arten, Verhältnissen und Proportionen.* St. Petersburg, 1770. Reprinted in his *Opera Omnia*, series 1, volume 1. Translated into Russian, 1768; Dutch, 1773; French, 1774; Latin, 1790; English, 1797. *6, 636.*

115 L. Eulero, "Observationes circa bina biquadrata quorum summam in duo alia biquadrata resolvere liceat," *Novi commentarii academiæ scientiarum imperialis Petropolitanæ* **17** (1772), 64–69. Reprinted in his *131.*

Commentationes arithmeticæ collectæ, volume 1, 473–476. Reprinted in his *Opera Omnia*, series 1, volume 3, 211–217.

513. **116** L. Eulero, "Observationes circa novum et singulare progressionum genus," *Novi commentarii academiæ scientiarum imperialis Petropolitanæ* **20** (1775), 123–139. Reprinted in his *Opera Omnia*, series 1, volume 7, 246–261.

202. **117** L. Eulero, "De serie Lambertina, plurimisque eius insignibus proprietatibus," *Acta academiæ scientiarum imperialis Petropolitanæ* **3**,2 (1779), 29–51. Reprinted in his *Opera Omnia*, series 1, volume 6, 350–369.

207, 634. **118** L. Eulero, "Specimen transformationis singularis serierum," *Nova acta academiæ scientiarum imperialis Petropolitanæ* **12** (1794), 58–70. Submitted for publication in 1778. Reprinted in his *Opera Omnia*, series 1, volume 16(2), 41–55.

288. **119** Johann Faulhabern, *Academia Algebræ*, Darinnen die miraculosische Inventiones zu den höchsten Cossen weiters *continuirt* und *profitiert* werden, ... biß auff die regulierte *Zensicubiccubic* Coß durch offnen Truck *publiciert* worden. Augsburg, 1631.

381, 636. **120** William Feller, *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, 1950; second edition, 1957; third edition, 1968.

131. **121** Pierre de Fermat, letter to Marin Mersenne (25 December 1640), in *Œuvres de Fermat*, volume 2, 212–217.

633, 634. **122** Leonardo Fibonacci [Pisano], *Liber Abaci*. First edition, 1202 (now lost); second edition, 1228. Reprinted in *Scritti di Leonardo Pisano*, edited by Baldassarre Boncompagni, 1857, volume 1.

24. **123** Bruno de Finetti, *Teoria delle Probabilità*. Turin, 1970. English translation, *Theory of Probability*, Wiley, 1974–1975.

636. **124** Michael E. Fisher, "Statistical mechanics of dimers on a plane lattice," *Physical Review* **124** (1961), 1664–1672.

636. **125** R. A. Fisher, "Moments and product moments of sampling distributions," *Proceedings of the London Mathematical Society*, series 2, **30** (1929), 199–238.

634. **126** Pierre Forcadel, *L'arithmeticque*. Paris, 1557.

22. **127** J. Fourier, "Refroidissement séculaire du globe terrestre," *Bulletin des Sciences par la Société philomathique de Paris*, series 3, **7** (1820), 58–70. Reprinted in *Œuvres de Fourier*, volume 2, 271–288.

515, 633. **128** Aviezri S. Fraenkel, "Complementing and exactly covering sequences," *Journal of Combinatorial Theory*, series A, **14** (1973), 8–20.

129 Aviezri S. Fraenkel, "How to beat your Wythoff games' opponent on three fronts," *American Mathematical Monthly* **89** (1982), 353–361. *563.*

130 J. S. Frame, B. M. Stewart, and Otto Dunkel, "Partial solution to problem 3918," *American Mathematical Monthly* **48** (1941), 216–219. *633.*

131 Piero della Francesca, *Libellus de quinque corporibus regularibus.* Vatican Library, manuscript Urbinas 632. Translated into Italian by Luca Pacioli, as part 3 of Pacioli's *Diuine Proportione*, Venice, 1509. *635.*

132 J. Franel, Solutions to questions 42 and 170, in *L'Intermédiare des Mathématiciens* **1** (1894), 45–47; **2** (1895), 33–35. *549.*

133 W. D. Frazer and A. C. McKellar, "Samplesort: A sampling approach to minimal storage tree sorting," *Journal of the ACM* **27** (1970), 496–507. *634.*

134 Michael Lawrence Fredman, *Growth Properties of a Class of Recursively Defined Functions.* Ph.D. thesis, Stanford University, Computer Science Department, 1972. *513.*

135 Nikolao Fuss, "Solutio quæstionis, quot modis polygonum n laterum in polygona m laterum, per diagonales resolvi quæat," *Nova acta academiæ scientiarum imperialis Petropolitanæ* **9** (1791), 243–251. *361.*

136 Martin Gardner, "About phi, an irrational number that has some remarkable geometrical expressions," *Scientific American* **201**, 2 (August 1959), 128–134. Reprinted with additions in his book *The 2nd Scientific American Book of Mathematical Puzzles & Diversions*, 1961, 89–103. *299.*

137 Martin Gardner, "On the paradoxical situations that arise from nontransitive relations," *Scientific American* **231**, 4 (October 1974), 120–124. Reprinted with additions in his book *Time Travel and Other Mathematical Bewilderments*, 1988, 55–69. *410.*

138 Martin Gardner, "From rubber ropes to rolling cubes, a miscellany of refreshing problems," *Scientific American* **232**, 3 (March 1975), 112–114; **232**, 4 (April 1975), 130, 133. Reprinted with additions in his book *Time Travel and Other Mathematical Bewilderments*, 1988, 111–124. *634.*

139 Martin Gardner, "On checker jumping, the amazon game, weird dice, card tricks and other playful pastimes," *Scientific American* **238**, 2 (February 1978), 19, 22, 24, 25, 30, 32. Reprinted with additions in his book *Penrose Tiles to Trapdoor Ciphers*, 1989, 265–280. *636.*

140 J. Garfunkel, "Problem E 1816: An inequality related to Stirling's formula," *American Mathematical Monthly* **74** (1967), 202. *636.*

141 George Gasper and Mizan Rahman, *Basic Hypergeometric Series.* Cambridge University Press, 1990. *223.*

123, 633. **142** C. F. Gauss, *Disquisitiones Arithmeticæ.* Leipzig, 1801. Reprinted in his *Werke*, volume 1.

207, 222, 529, 634. **143** Carolo Friderico Gauss, "Disquisitiones generales circa seriem infinitam

$$1 + \frac{\alpha\beta}{1\,.\,\gamma}x + \frac{\alpha(\alpha+1)\beta(\beta+1)}{1\,.\,2\,.\,\gamma(\gamma+1)}xx + \frac{\alpha(\alpha+1)(\alpha+2)\beta(\beta+1)(\beta+2)}{1\,.\,2\,.\,3\,.\,\gamma(\gamma+1)(\gamma+2)}x^3 + \text{etc.}$$

Pars prior," *Commentationes societatis regiæ scientiarum Gottingensis recentiores* **2** (1813). (Thesis delivered to the Royal Society in Göttingen, 20 January 1812.) Reprinted in his *Werke*, volume 3, 123–163, together with an unpublished sequel on pages 207–229.

633. **144** C. F. Gauss, "Pentagramma mirificum," written prior to 1836. Published posthumously in his *Werke*, volume 3, 480–490.

551. **145** A. Genocchi, "Intorno all' expressioni generali di numeri Bernoulliani," *Annali di Scienze Matematiche e Fisiche* **3** (1852), 395–405.

634. **146** Ira Gessel, "Some congruences for Apéry numbers," *Journal of Number Theory* **14** (1982), 362–368.

270. **147** Ira Gessel and Richard P. Stanley, "Stirling polynomials," *Journal of Combinatorial Theory*, series A, **24** (1978), 24–33.

271. **148** Jekuthiel Ginsburg, "Note on Stirling's numbers," *American Mathematical Monthly* **35** (1928), 77–80.

636. **149** J. W. L. Glaisher, "On the product $1^1.2^2.3^3\ldots n^n$, *Messenger of Mathematics* **7** (1877), 43–47.

603, 633. **150** Solomon W. Golomb, "Problem 5407: A nondecreasing indicator function," *American Mathematical Monthly* **74** (1967), 740–743.

507. **151** Solomon W. Golomb, "The 'Sales Tax' theorem," *Mathematics Magazine* **49** (1976), 187–189.

460. **152** Solomon W. Golomb, "Problem E2529: An application of $\psi(x)$," *American Mathematical Monthly* **83** (1976), 487–488.

634. **153** I. J. Good, "Short proof of a conjecture by Dyson," *Journal of Mathematical Physics* **11** (1970), 1884.

224, 634. **154** R. William Gosper, Jr., "Decision procedure for indefinite hypergeometric summation," *Proceedings of the National Academy of Sciences of the United States of America* **75** (1978), 40–42.

513. **155** R. L. Graham, "On a theorem of Uspensky," *American Mathematical Monthly* **70** (1963), 407–409.

156 R. L. Graham, "A Fibonacci-like sequence of composite numbers," *Mathematics Magazine* **37** (1964), 322–324. *635.*

157 R. L. Graham, "Problem 5749," *American Mathematical Monthly* **77** (1970), 775. *634.*

158 Ronald L. Graham, "Covering the positive integers by disjoint sets of the form $\{\,[n\alpha+\beta] : n = 1, 2, \ldots\,\}$," *Journal of Combinatorial Theory*, series A, **15** (1973), 354–358. *514.*

159 R. L. Graham, "Problem 1242: Bijection between integers and composites," *Mathematics Magazine* **60** (1987), 180. *633.*

160 R. L. Graham and D. E. Knuth, "Problem E 2982: A double infinite sum for $|x|$," *American Mathematical Monthly* **96** (1989), 525–526. *633.*

161 Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley, 1989; second edition, 1994. *102.*

162 R. L. Graham and H. O. Pollak, "Note on a nonlinear recurrence related to $\sqrt{2}$," *Mathematics Magazine* **43** (1970), 143–145. *633.*

163 Guido Grandi, letter to Leibniz (July 1713), in *Leibnizens mathematische Schriften*, volume 4, 215–217. *58.*

164 Daniel H. Greene and Donald E. Knuth, *Mathematics for the Analysis of Algorithms.* Birkhäuser, Boston, 1981; third edition, 1990. *535, 636.*

165 Samuel L. Greitzer, *International Mathematical Olympiads, 1959–1977.* Mathematical Association of America, 1978. *633.*

166 Oliver A. Gross, "Preferential arrangements," *American Mathematical Monthly* **69** (1962), 4–8. *635.*

167 Branko Grünbaum, "Venn diagrams and independent families of sets," *Mathematics Magazine* **48** (1975), 12–23. *498.*

168 L. J. Guibas and A. M. Odlyzko, "String overlaps, pattern matching, and nontransitive games," *Journal of Combinatorial Theory*, series A, **30** (1981), 183–208. *590, 636.*

169 Richard K. Guy, *Unsolved Problems in Number Theory.* Springer-Verlag, 1981. *525.*

170 Inger Håland and Donald E. Knuth, "Polynomials involving the floor function," submitted for publication, 1993. *514, 633.*

171 Marshall Hall, Jr., *The Theory of Groups.* Macmillan, 1959. *553.*

172 P. R. Halmos, "How to write mathematics," *L'Enseignement mathématique* **16** (1970), 123–152. Reprinted in *How to Write Mathematics*, American Mathematical Society, 1973, 19–48. *vi.*

v. **173** Paul R. Halmos, *I Want to Be a Mathematician: An Automathography*. Springer-Verlag, 1985. Reprinted by Mathematical Association of America, 1988.

305. **174** G. H. Halphen, "Sur des suites de fractions analogues à la suite de Farey," *Bulletin de la Société mathématique de France* **5** (1876), 170–175. Reprinted in his *Œuvres*, volume 2, 102–107.

591. **175** Hans Hamburger, "Über eine Erweiterung des Stieltjesschen Momentenproblems," *Mathematische Annalen* **81** (1920), 235–319; **82** (1921), 120–164, 168–187.

v. **176** J. M. Hammersley, "On the enfeeblement of mathematical skills by 'Modern Mathematics' and by similar soft intellectual trash in schools and universities," *Bulletin of the Institute of Mathematics and its Applications* **4**, 4 (October 1968), 66–85.

636. **177** J. M. Hammersley, "An undergraduate exercise in manipulation," *The Mathematical Scientist* **14** (1989), 1–23.

42. **178** Eldon R. Hansen, *A Table of Series and Products*. Prentice-Hall, 1975.

442, 636. **179** G. H. Hardy, *Orders of Infinity: The 'Infinitärcalcül' of Paul du Bois-Reymond*. Cambridge University Press, 1910; second edition, 1924.

636. **180** G. H. Hardy, "A mathematical theorem about golf," *The Mathematical Gazette* **29** (1944), 226–227. Reprinted in his *Collected Papers*, volume 7, 488.

111, 633. **181** G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*. Clarendon Press, Oxford, 1938; fifth edition, 1979.

300, 332, 602, 636. **182** Peter Henrici, *Applied and Computational Complex Analysis*. Wiley, volume 1, 1974; volume 2, 1977; volume 3, 1986.

634. **183** Peter Henrici, "De Branges' proof of the Bieberbach conjecture: A view from computational analysis," *Sitzungsberichte der Berliner Mathematischen Gesellschaft* (1987), 105–121.

555. **184** Charles Hermite, letter to C. W. Borchardt (8 September 1875), in *Journal für die reine und angewandte Mathematik* **81** (1876), 93–95. Reprinted in his *Œuvres*, volume 3, 211–214.

634. **185** Charles Hermite, *Cours de M. Hermite*. Faculté des Sciences de Paris, 1882. Third edition, 1887; fourth edition, 1891.

538, 634. **186** Charles Hermite, letter to S. Pincherle (10 May 1900), in *Annali di Matematica pura ed applicata*, series 3, **5** (1901), 57–60. Reprinted in his *Œuvres*, volume 4, 529–531.

187 I. N. Herstein and I. Kaplansky, *Matters Mathematical.* Harper & Row, 1974. *8.*

188 A. P. Hillman and V. E. Hoggatt, Jr., "A proof of Gould's Pascal hexagon conjecture," *Fibonacci Quarterly* **10** (1972), 565–568, 598. *634.*

189 C. A. R. Hoare, "Quicksort," *The Computer Journal* **5** (1962), 10–15. *28.*

190 L. C. Hsu, "Note on a combinatorial algebraic identity and its application," *Fibonacci Quarterly* **11** (1973), 480–484. *634.*

191 Kenneth E. Iverson, *A Programming Language.* Wiley, 1962. *24, 67, 633.*

192 C. G. J. Jacobi, *Fundamenta nova theoriæ functionum ellipticarum.* Königsberg, Bornträger, 1829. Reprinted in his *Gesammelte Werke*, volume 1, 49–239. *64.*

193 Svante Janson, Donald E. Knuth, Tomasz Łuczak, and Boris Pittel, "The birth of the giant component," *Random Structures and Algorithms* **4** (1993), 233–358. *202.*

194 Dov Jarden and Theodor Motzkin, "The product of sequences with a common linear recursion formula of order 2," *Riveon Lematematika* **3** (1949), 25–27, 38 (Hebrew with English summary). English version reprinted in Dov Jarden, *Recurring Sequences*, Jerusalem, 1958, 42–45; second edition, 1966, 30–33. *556.*

195 Arne Jonassen and Donald E. Knuth, "A trivial algorithm whose analysis isn't," *Journal of Computer and System Sciences* **16** (1978), 301–322. *535.*

196 Bush Jones, "Note on internal merging," *Software — Practice and Experience* **2** (1972), 241–243. *175.*

197 Flavius Josephus, *ΙΣΤΟΡΙΑ ΙΟΥΔΑΪΚΟΥ ΠΟΛΕΜΟΥ ΠΡΟΣ ΡΩΜΑΙΟΥΣ.* English translation, *History of the Jewish War against the Romans*, by H. St. J. Thackeray, in the Loeb Classical Library edition of Josephus's works, volumes 2 and 3, Heinemann, London, 1927–1928. (The "Josephus problem" may be based on an early manuscript now preserved only in the Slavonic version; see volume 2, page xi, and volume 3, page 654.) *8.*

198 R. Jungen, "Sur les séries de Taylor n'ayant que des singularités algébrico-logarithmiques sur leur cercle de convergence," *Commentarii Mathematici Helvetici* **3** (1931), 266–306. *635.*

199 J. Karamata, "Théorèmes sur la sommabilité exponentielle et d'autres sommabilités rattachant," *Mathematica* (Cluj) **9** (1935), 164–178. *257.*

200 I. Kaucký, "Problem E 2257: A harmonic identity," *American Mathematical Monthly* **78** (1971), 908. *635.*

601. **201** J. B. Keiper, "Power series expansions of Riemann's ξ function," *Mathematics of Computation* **58** (1992), 765–773.

633, 635. **202** Murray S. Klamkin, *International Mathematical Olympiads, 1978–1985, and Forty Supplementary Problems.* Mathematical Association of America, 1986.

202. **203** R. Arthur Knoebel, "Exponentials reiterated," *American Mathematical Monthly* **88** (1981), 235–252.

636. **204** Konrad Knopp, *Theorie und Anwendung der unendlichen Reihen.* Julius Springer, Berlin, 1922; second edition, 1924. Reprinted by Dover, 1945. Fourth edition, 1947; fifth edition, 1964. English translation, *Theory and Application of Infinite Series*, 1928; second edition, 1951.

481. **205** Donald E. Knuth, "Euler's constant to 1271 places," *Mathematics of Computation* **16** (1962), 275–281.

553. **206** Donald Knuth, "Transcendental numbers based on the Fibonacci sequence," *Fibonacci Quarterly* **2** (1964), 43–44, 52.

vi, 500, 514, 530, 573, 633, 634, 635, 636. **207** Donald E. Knuth, *The Art of Computer Programming*, volume 1: *Fundamental Algorithms.* Addison-Wesley, 1968; second edition, 1973.

110, 128, 500, 633, 635, 636. **208** Donald E. Knuth, *The Art of Computer Programming*, volume 2: *Seminumerical Algorithms.* Addison-Wesley, 1969; second edition, 1981.

267, 411, 501, 634, 635, 636. **209** Donald E. Knuth, *The Art of Computer Programming*, volume 3: *Sorting and Searching.* Addison-Wesley, 1973; second printing, 1975.

634. **210** Donald E. Knuth, "Problem E 2492: Some sum," *American Mathematical Monthly* **82** (1975), 855.

636. **211** Donald E. Knuth, *Mariages stables et leurs relations avec d'autres problèmes combinatoires.* Les Presses de l'Université de Montréal, 1976. Revised and corrected edition, 1980.

633. **212** Donald E. Knuth, *The TEXbook.* Addison-Wesley, 1984. Reprinted as volume A of *Computers & Typesetting*, 1986.

564. **213** Donald E. Knuth, "An analysis of optimum caching," *Journal of Algorithms* **6** (1985), 181–199.

633. **214** Donald E. Knuth, *Computers & Typesetting*, volume D: *METAFONT: The Program.* Addison-Wesley, 1986.

633. **215** Donald E. Knuth, "Problem 1280: Floor function identity," *Mathematics Magazine* **61** (1988), 319–320.

634. **216** Donald E. Knuth, "Problem E 3106: A new sum for n^2," *American Mathematical Monthly* **94** (1987), 795–797.

217 Donald E. Knuth, "Fibonacci multiplication," *Applied Mathematics Letters* **1** (1988), 57–60. *635.*

218 Donald E. Knuth, "A Fibonacci-like sequence of composite numbers," *Mathematics Magazine* **63** (1990), 21–25. *562.*

219 Donald E. Knuth, "Problem E 3309: A binomial coefficient inequality," *American Mathematical Monthly* **97** (1990), 614. *634.*

220 Donald E. Knuth, "Two notes on notation," *American Mathematical Monthly* **99** (1992), 403–422. *24, 162, 267.*

221 Donald E. Knuth, "Convolution polynomials," *The Mathematica Journal* **2**,4 (Fall 1992), 67–78. *267, 566, 635.*

222 Donald E. Knuth, "Johann Faulhaber and sums of powers," *Mathematics of Computation* **61** (1993), 277–294. *288.*

223 Donald E. Knuth, "Bracket notation for the coefficient-of operator," in *A Classical Mind*, essays in honour of C. A. R. Hoare, edited by A. W. Roscoe, Prentice-Hall, 1994, 247–258. *197.*

224 Donald E. Knuth and Thomas J. Buckholtz, "Computation of Tangent, Euler, and Bernoulli numbers," *Mathematics of Computation* **21** (1967), 663–688. *555.*

225 Donald E. Knuth and Ilan Vardi, "Problem 6581: The asymptotic expansion of the middle binomial coefficient," *American Mathematical Monthly* **97** (1990), 626–630. *636.*

226 Donald E. Knuth and Herbert S. Wilf, "The power of a prime that divides a generalized binomial coefficient," *Journal für die reine und angewandte Mathematik* **396** (1989), 212–219. *530, 635.*

227 Donald E. Knuth and Hermann Zapf, "AMS Euler — A new typeface for mathematics," *Scholarly Publishing* **20** (1989), 131–157. *viii.*

228 C. Kramp, *Élémens d'arithmétique universelle.* Cologne, 1808. *111.*

229 E. E. Kummer, "Ueber die hypergeometrische Reihe *213, 634.*

$$1+\frac{\alpha\beta}{1\,.\,\gamma}x+\frac{\alpha(\alpha+1)\beta(\beta+1)}{1\,.\,2\,.\,\gamma(\gamma+1)}xx$$
$$+\frac{\alpha(\alpha+1)(\alpha+2)\beta(\beta+1)(\beta+2)}{1\,.\,2\,.\,3\,.\,\gamma(\gamma+1)(\gamma+2)}x^3+\ldots,"$$

Journal für die reine und angewandte Mathematik **15** (1836), 39–83, 127–172. Reprinted in his *Collected Papers*, volume 2, 75–166.

634. **230** E. E. Kummer, "Über die Ergänzungssätze zu den allgemeinen Reciprocitätsgesetzen," *Journal für die reine und angewandte Mathematik* **44** (1852), 93–146. Reprinted in his *Collected Papers*, volume 1, 485–538.

501. **231** R. P. Kurshan and B. Gopinath, "Recursively generated periodic sequences," *Canadian Journal of Mathematics* **26** (1974), 1356–1371.

304. **232** Thomas Fantet de Lagny, *Analyse générale ou Méthodes nouvelles pour résoudre les problèmes de tous les genres et de tous les degrés à l'infini.* Published as volume 11 of *Mémoires de l'Académie Royale des Sciences*, Paris, 1733.

635. **233** de la Grange [Lagrange], "Démonstration d'un théorème nouveau concernant les nombres premiers," *Nouveaux Mémoires de l'Académie royale des Sciences et Belles-Lettres*, Berlin (1771), 125–137. Reprinted in his *Œuvres*, volume 3, 425–438.

470. **234** de la Grange [Lagrange], "Sur une nouvelle espèce de calcul rélatif à la différentiation & à l'intégration des quantités variables," *Nouveaux Mémoires de l'Académie royale des Sciences et Belles-Lettres*, Berlin (1772), 185–221. Reprinted in his *Œuvres*, volume 3, 441–476.

634. **235** I. Lah, "Eine neue Art von Zahlen, ihre Eigenschaften und Anwendung in der mathematischen Statistik," *Mitteilungsblatt für Mathematische Statistik* **7** (1955), 203–212.

201. **236** I. H. Lambert, "Observationes variæ in Mathesin puram," *Acta Helvetica* **3** (1758), 128–168. Reprinted in his *Opera Mathematica*, volume 1, 16–51.

201. **237** Lambert, "Observations analytiques," *Nouveaux Mémoires de l'Académie royale des Sciences et Belles-Lettres*, Berlin (1770), 225–244. Reprinted in his *Opera Mathematica*, volume 2, 270–290.

448, 636. **238** Edmund Landau, *Handbuch der Lehre von der Verteilung der Primzahlen*, two volumes. Teubner, Leipzig, 1909.

634. **239** Edmund Landau, *Vorlesungen über Zahlentheorie*, three volumes. Hirzel, Leipzig, 1927.

466. **240** P. S. de la Place [Laplace], "Mémoire sur les approximations des Formules qui sont fonctions de très-grands nombres," *Mémoires de l'Academie royale des Sciences de Paris* (1782), 1–88. Reprinted in his *Œuvres Complètes* **10**, 207–291.

633. **241** Adrien-Marie Legendre, *Essai sur la Théorie des Nombres.* Paris, 1798; second edition, 1808. Third edition (retitled *Théorie des Nombres*, in two volumes), 1830; fourth edition, Blanchard, 1955.

242 D. H. Lehmer, "Tests for primality by the converse of Fermat's theorem," *Bulletin of the American Mathematical Society*, series 2, **33** (1927), 327–340. Reprinted in his *Selected Papers*, volume 1, 69–82. *633.*

243 D. H. Lehmer, "On Stern's diatomic series," *American Mathematical Monthly* **36** (1929), 59–67. *635.*

244 D. H. Lehmer, "On Euler's totient function," *Bulletin of the American Mathematical Society*, series 2, **38** (1932), 745–751. Reprinted in his *Selected Papers*, volume 1, 319–325. *526.*

245 G. W. Leibniz, letter to Johann Bernoulli (May 1695), in *Leibnizens mathematische Schriften*, volume 3, 174–179. *168.*

246 C. G. Lekkerkerker, "Voorstelling van natuurlijke getallen door een som van getallen van Fibonacci," *Simon Stevin* **29** (1952), 190–195. *295.*

247 Tamás Lengyel, "A combinatorial identity and the world series," *SIAM Review* **35** (1993), 294–297. *167.*

248 Tamás Lengyel, "On some properties of the series $\sum_{k=0}^{\infty} k^n x^k$ and the Stirling numbers of the second kind," submitted for publication, 1993. *635.*

249 Li Shan-Lan, *Duò Jī Bǐ Lèi* [Sums of Piles Obtained Inductively]. In his *Zégǔxī Zhaī Suànxué* [Classically inspired meditations on mathematics], Nanjing, 1867. *269.*

250 Elliott H. Lieb, "Residual entropy of square ice," *Physical Review* **162** (1967), 162–172. *636.*

251 J. Liouville, "Sur l'expression $\varphi(n)$, qui marque combien la suite 1, 2, 3, ..., n contient de nombres premiers à n," *Journal de Mathématiques pures et appliquées*, series 2, **2** (1857), 110–112. *136.*

252 B. F. Logan, "The recovery of orthogonal polynomials from a sum of squares," *SIAM Journal on Mathematical Analysis* **21** (1990), 1031–1050. *634.*

253 B. F. Logan, "Polynomials related to the Stirling numbers," AT&T Bell Laboratories internal technical memorandum, August 10, 1987. *635.*

254 Calvin T. Long and Verner E. Hoggatt, Jr., "Sets of binomial coefficients with equal products," *Fibonacci Quarterly* **12** (1974), 71–79. *634.*

255 Shituo Lou and Qi Yao, "A Chebychev's type of prime number theorem in a short interval-II," *Hardy-Ramanujan Journal* **15** (1992), 1–33. *525.*

256 Sam Loyd, *Cyclopedia of Puzzles*. Franklin Bigelow Corporation, Morningside Press, New York, 1914. *560.*

257 E. Lucas, "Sur les rapports qui existent entre la théorie des nombres et le Calcul intégral," *Comptes Rendus hebdomadaires des séances de l'Académie des Sciences* (Paris) **82** (1876), 1303–1305. *633, 634, 635.*

634. **258** Édouard Lucas, "Sur les congruences des nombres eulériens et des coefficients différentiels des fonctions trigonométriques, suivant un module premier," *Bulletin de la Société mathématique de France* **6** (1878), 49–54.

292, 634. **259** Edouard Lucas, *Théorie des Nombres*, volume 1. Gauthier-Villars, Paris, 1891.

1. **260** Édouard Lucas, *Récréations mathématiques*, four volumes. Gauthier-Villars, Paris, 1891–1894. Reprinted by Albert Blanchard, Paris, 1960. (The Tower of Hanoi is discussed in volume 3, pages 55–59.)

501. **261** R. C. Lyness, "Cycles," *The Mathematical Gazette* **29** (1945), 231–233.

501. **262** R. C. Lyness, "Cycles," *The Mathematical Gazette* **45** (1961), 207–209.

469. **263** Colin Maclaurin, *Collected Letters*, edited by Stella Mills. Shiva Publishing, Nantwich, Cheshire, 1982.

140. **264** P. A. MacMahon, "Application of a theory of permutations in circular procession to the theory of numbers," *Proceedings of the London Mathematical Society* **23** (1892), 305–313.

269. **265** J.-C. Martzloff, *Histoire des Mathématiques Chinoises*. Paris, 1988.

294, 635. **266** Ĭu. V. Matiiasevich, "Diofantovost' perechislimykh mnozhestv," *Doklady Akademii Nauk SSSR* **191** (1970), 279–282. English translation, with amendments by the author, "Enumerable sets are diophantine," *Soviet Mathematics* **11** (1970), 354–357.

vi. **267** Z. A. Melzak, *Companion to Concrete Mathematics*. Volume 1, *Mathematical Techniques and Various Applications*, Wiley, 1973; volume 2, *Mathematical Ideas, Modeling & Applications*, Wiley, 1976.

634. **268** N. S. Mendelsohn, "Problem E2227: Divisors of binomial coefficients," *American Mathematical Monthly* **78** (1971), 201.

109. **269** Marini Mersenni, *Cogitata Physico-Mathematica*. Paris, 1644.

139. **270** F. Mertens, "Ueber einige asymptotische Gesetze der Zahlentheorie," *Journal für die reine und angewandte Mathematik* **77** (1874), 289–338.

23. **271** Mertens, "Ein Beitrag zur analytischen Zahlentheorie," *Journal für die reine und angewandte Mathematik* **78** (1874), 46–62.

634. **272** W. H. Mills, "A prime representing function," *Bulletin of the American Mathematical Society*, series 2, **53** (1947), 604.

138. **273** A. F. Möbius, "Über eine besondere Art von Umkehrung der Reihen," *Journal für die reine und angewandte Mathematik* **9** (1832), 105–123. Reprinted in his *Gesammelte Werke*, volume 4, 589–612.

274 A. Moessner, "Eine Bemerkung über die Potenzen der natürlichen Zahlen," *Sitzungsberichte der Mathematisch - Naturwissenschaftliche Klasse der Bayerischen Akademie der Wissenschaften*, 1951, Heft 3, 29. *636.*

275 Hugh L Montgomery, "Fluctuations in the mean of Euler's phi function," *Proceedings of the Indian Academy of Sciences*, Mathematical Sciences, **97** (1987), 239–245. *463.*

276 Peter L. Montgomery, "Problem E 2686: LCM of binomial coefficients," *American Mathematical Monthly* **86** (1979), 131. *634.*

277 Leo Moser, "Problem B-6: Some reflections," *Fibonacci Quarterly* **1**, 4 (1963), 75–76. *291.*

278 T. S. Motzkin and E. G. Straus, "Some combinatorial extremum problems," *Proceedings of the American Mathematical Society* **7** (1956), 1014–1021. *564.*

279 B. R. Myers, "Problem 5795: The spanning trees of an n-wheel," *American Mathematical Monthly* **79** (1972), 914–915. *635.*

280 Isaac Newton, letter to John Collins (18 February 1670), in *The Correspondence of Isaac Newton*, volume 1, 27. Excerpted in *The Mathematical Papers of Isaac Newton*, volume 3, 563. *277.*

281 Ivan Niven, *Diophantine Approximations*. Interscience, 1963. *633.*

282 Ivan Niven, "Formal power series," *American Mathematical Monthly* **76** (1969), 871–889. *332.*

283 Andrew M. Odlyzko and Herbert S. Wilf, "Functional iteration and the Josephus problem," *Glasgow Mathematical Journal* **33** (1991), 235–240. *81.*

284 Blaise Pascal, "De numeris multiplicibus," presented to Académie Parisienne in 1654 and published with his *Traité du triangle arithmétique* [285]. Reprinted in *Œuvres de Blaise Pascal*, volume 3, 314–339. *624, 633.*

285 Blaise Pascal, "Traité du triangle arithmetique," in his *Traité du Triangle Arithmetique, avec quelques autres petits traitez sur la mesme matiere*, Paris, 1665. Reprinted in *Œuvres de Blaise Pascal* (Hachette, 1904–1914), volume 3, 445–503; Latin editions from 1654 in volume 11, 366–390. *155, 156.*

286 G. P. Patil, "On the evaluation of the negative binomial distribution with examples," *Technometrics* **2** (1960), 501–505. *636.*

287 C. S. Peirce, letter to E. S. Holden (January 1901). In *The New Elements of Mathematics*, edited by Carolyn Eisele, Mouton, The Hague, 1976, volume 1, 247–253. (See also page 211.) *634.*

525. **288** C. S. Peirce, letter to Henry B. Fine (17 July 1903). In *The New Elements of Mathematics*, edited by Carolyn Eisele, Mouton, The Hague, 1976, volume 3, 781–784. (See also "Ordinals," an unpublished manuscript from circa 1905, in *Collected Papers of Charles Sanders Peirce*, volume 4, 268–280.)

408. **289** Walter Penney, "Problem 95: Penney-Ante," *Journal of Recreational Mathematics* **7** (1974), 321.

636. **290** J. K. Percus, *Combinatorial Methods*. Springer-Verlag, 1971.

229, 575, 634. **291** Marko Petkovšek, "Hypergeometric solutions of linear recurrences with polynomial coefficients," *Journal of Symbolic Computation* **14** (1992), 243–264.

207, 214, 217, 634. **292** J. F. Pfaff, "Observationes analyticæ ad *L. Euleri* institutiones calculi integralis, Vol. IV, Supplem. II & IV," *Nova acta academiæ scientiarum imperialis Petropolitanæ* **11**, Histoire section, 37–57. (This volume, printed in 1798, contains mostly proceedings from 1793, although Pfaff's memoir was actually received in 1797.)

48. **293** L. Pochhammer, "Ueber hypergeometrische Functionen n^{ter} Ordnung," *Journal für die reine und angewandte Mathematik* **71** (1870), 316–352.

636. **294** H. Poincaré, "Sur les fonctions à espaces lacunaires," *American Journal of Mathematics* **14** (1892), 201–221.

471. **295** S. D. Poisson, "Mémoire sur le calcul numérique des intégrales définies," *Mémoires de l'Académie Royale des Sciences de l'Institut de France*, series 2, **6** (1823), 571–602.

635. **296** G. Pólya, "Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen," *Acta Mathematica* **68** (1937), 145–254. English translation, with commentary by Ronald C. Read, *Combinatorial Enumeration of Groups, Graphs, and Chemical Compounds*, Springer-Verlag, 1987.

vi, 16, 508, 633. **297** George Pólya, *Induction and Analogy in Mathematics*. Princeton University Press, 1954.

327, 635. **298** G. Pólya, "On picture-writing," *American Mathematical Monthly* **63** (1956), 689–697.

636. **299** G. Pólya and G. Szegö, *Aufgaben und Lehrsätze aus der Analysis*, two volumes. Julius Springer, Berlin, 1925; fourth edition, 1970 and 1971. English translation, *Problems and Theorems in Analysis*, 1972 and 1976.

635. **300** R. Rado, "A note on the Bernoullian numbers," *Journal of the London Mathematical Society* **9** (1934), 88–90.

301 Earl D. Rainville, "The contiguous function relations for ${}_pF_q$ with applications to Bateman's $J_n^{u,v}$ and Rice's $H_n(\zeta, p, \nu)$," *Bulletin of the American Mathematical Society*, series 2, **51** (1945), 714–723. *529.*

302 George N. Raney, "Functional composition patterns and power series reversion," *Transactions of the American Mathematical Society* **94** (1960), 441–451. *359, 635.*

303 D. Rameswar Rao, "Problem E 2208: A divisibility problem," *American Mathematical Monthly* **78** (1971), 78–79. *633.*

304 John William Strutt, Third Baron Rayleigh, *The Theory of Sound*. First edition, 1877; second edition, 1894. (The cited material about irrational spectra is from section 92a of the second edition.) *77.*

305 Robert Recorde, *The Whetstone of Witte*. London, 1557. *446.*

306 Simeon Reich, "Problem 6056: Truncated exponential-type series," *American Mathematical Monthly* **84** (1977), 494–495. *636.*

307 Georges de Rham, "Un peu de mathématiques à propos d'une courbe plane," *Elemente der Mathematik* **2** (1947), 73–76, 89–97. Reprinted in his *Œuvres Mathématiques*, 678–689. *635.*

308 Paolo Ribenboim, *13 Lectures on Fermat's Last Theorem*. Springer-Verlag, 1979. *555, 634.*

309 Bernhard Riemann, "Ueber die Darstellbarkeit einer Function durch eine trigonometrische Reihe," Habilitationsschrift, Göttingen, 1854. Published in *Abhandlungen der mathematischen Classe der Königlichen Gesellschaft der Wissenschaften zu Göttingen* **13** (1868), 87–132. Reprinted in his *Gesammelte Mathematische Werke*, 227–264. *633.*

310 Samuel Roberts, "On the figures formed by the intercepts of a system of straight lines in a plane, and on analogous relations in space of three dimensions," *Proceedings of the London Mathematical Society* **19** (1889), 405–422. *633.*

311 Øystein Rødseth, "Problem E 2273: Telescoping Vandermonde convolutions," *American Mathematical Monthly* **79** (1972), 88–89. *634.*

312 J. Barkley Rosser and Lowell Schoenfeld, "Approximate formulas for some functions of prime numbers," *Illinois Journal of Mathematics* **6** (1962), 64–94. *111.*

313 Gian-Carlo Rota, "On the foundations of combinatorial theory. I. Theory of Möbius functions," *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete* **2** (1964), 340–368. *516.*

314 Ranjan Roy, "Binomial identities and hypergeometric series," *American Mathematical Monthly* **94** (1987), 36–46. *634.*

214. **315** Louis Saalschütz, "Eine Summationsformel," *Zeitschrift für Mathematik und Physik* **35** (1890), 186–188.

463. **316** A. I. Saltykov, "O funktsii Éĭlera," *Vestnik Moskovskogo Universiteta*, series 1, Matematika, Mekhanika (1960), number 6, 34–50.

548. **317** A. Sárközy, "On divisors of binomial coefficients, I," *Journal of Number Theory* **20** (1985), 70–80.

207. **318** W. W. Sawyer, *Prelude to Mathematics*. Baltimore, Penguin, 1955.

293. **319** O. Schlömilch, "Ein geometrisches Paradoxon," *Zeitschrift für Mathematik und Physik* **13** (1868), 162.

635. **320** Ernst Schröder, "Vier combinatorische Probleme," *Zeitschrift für Mathematik und Physik* **15** (1870), 361–376.

635. **321** Heinrich Schröter, "Ableitung der Partialbruch- und Produkt-Entwickelungen für die trigonometrischen Funktionen," *Zeitschrift für Mathematik und Physik* **13** (1868), 254–259.

633. **322** R. S. Scorer, P. M. Grundy, and C. A. B. Smith, "Some binary games," *The Mathematical Gazette* **28** (1944), 96–103.

635. **323** J. Sedláček, "On the skeletons of a graph or digraph," in *Combinatorial Structures and their Applications*, Gordon and Breach, 1970, 387–391. (This volume contains proceedings of the Calgary International Conference of Combinatorial Structures and their Applications, 1969.)

635. **324** J. O. Shallit, "Problem 6450: Two series," *American Mathematical Monthly* **92** (1985), 513–514.

273. **325** R. T. Sharp, "Problem 52: Overhanging dominoes," *Pi Mu Epsilon Journal* **1**, 10 (1954), 411–412.

87. **326** W. Sierpiński, "Sur la valeur asymptotique d'une certaine somme," *Bulletin International Académie Polonaise des Sciences et des Lettres* (Cracovie), series A (1910), 9–11.

634. **327** W. Sierpiński, "Sur les nombres dont la somme de diviseurs est une puissance du nombre 2," *Calcutta Mathematical Society Golden Jubilee Commemorative Volume* (1958–1959), part 1, 7–9.

634. **328** Wacław Sierpiński, *A Selection of Problems in the Theory of Numbers*. Macmillan, 1964.

635. **329** David L. Silverman, "Problematical Recreations 447: Numerical links," *Aviation Week & Space Technology* **89**, 10 (1 September 1968), 71. Reprinted as Problem 147 in *Second Book of Mathematical Bafflers*, edited by Angela Fox Dunn, Dover, 1983.

330 N. J. A. Sloane, *A Handbook of Integer Sequences*. Academic Press, 1973. Sequel, *The New Book of Integer Sequences*, Springer, 1994. *42, 341, 464.*

331 A. D. Solov'ev, "Odno kombinatornoe tozhdestvo i ego primenenie k zadache o pervom nastuplenii redkogo sobytiia," *Teoriia veroiatnosteĭ i eë primeneniia* **11** (1966), 313–320. English translation, "A combinatorial identity and its application to the problem concerning the first occurrence of a rare event," *Theory of Probability and its Applications* **11** (1966), 276–282. *408.*

332 William G. Spohn, Jr., "Can mathematics be saved?" *Notices of the American Mathematical Society* **16** (1969), 890–894. *v.*

333 Richard P. Stanley, "Differentiably finite power series," *European Journal of Combinatorics* **1** (1980), 175–188. *636.*

334 Richard P. Stanley, "On dimer coverings of rectangles of fixed width," *Discrete Applied Mathematics* **12** (1985), 81–87. *636.*

335 Richard P. Stanley, *Enumerative Combinatorics*, volume 1. Wadsworth & Brooks/Cole, 1986. *534, 635, 636.*

336 K. G. C. von Staudt, "Beweis eines Lehrsatzes, die Bernoullischen Zahlen betreffend," *Journal für die reine und angewandte Mathematik* **21** (1840), 372–374. *635.*

337 Guy L. Steele Jr., Donald R. Woods, Raphael A. Finkel, Mark R. Crispin, Richard M. Stallman, and Geoffrey S. Goodfellow, *The Hacker's Dictionary: A Guide to the World of Computer Wizards*. Harper & Row, 1983. *124.*

338 J. Steiner, "Einige Gesetze über die Theilung der Ebene und des Raumes," *Journal für die reine und angewandte Mathematik* **1** (1826), 349–364. Reprinted in his *Gesammelte Werke*, volume 1, 77–94. *5, 633.*

339 M. A. Stern, "Ueber eine zahlentheoretische Funktion," *Journal für die reine und angewandte Mathematik* **55** (1858), 193–220. *116.*

340 L. Stickelberger, "Ueber eine Verallgemeinerung der Kreistheilung," *Mathematische Annalen* **37** (1890), 321–367. *633.*

341 T. J. Stieltjes, letters to Hermite (June 1885), in *Correspondance d'Hermite et de Stieltjes*, volume 1, 146–159. *601.*

342 T. J. Stieltjes, "Table des valeurs des sommes $S_k = \sum_1^\infty n^{-k}$," *Acta Mathematica* **10** (1887), 299–302. Reprinted in his *Œuvres Complètes*, volume 2, 100–103. *633.*

343 James Stirling, *Methodus Differentialis*. London, 1730. English translation, *The Differential Method*, 1749. *192, 258, 297.*

549, 634. **344** Volker Strehl, "Binomial identities — combinatorial and algorithmic aspects," *Discrete Mathematics*, to appear in 1994.

481. **345** Dura W. Sweeney, "On the computation of Euler's constant," *Mathematics of Computation* **17** (1963), 170–178.

633. **346** J. J. Sylvester, "Problem 6919," *Mathematical Questions with their Solutions from the 'Educational Times'* **37** (1882), 42–43, 80.

133. **347** J. J. Sylvester, "On the number of fractions contained in any 'Farey series' of which the limiting number is given," *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science*, series 5, **15** (1883), 251–257. Reprinted in his *Collected Mathematical Papers*, volume 4, 101–109.

525. **348** M. Szegedy, "The solution of Graham's greatest common divisor problem," *Combinatorica* **6** (1986), 67–71.

635. **349** S. Tanny, "A probabilistic interpretation of Eulerian numbers," *Duke Mathematical Journal* **40** (1973), 717–722.

634. **350** L. Theisinger, "Bemerkung über die harmonische Reihe," *Monatshefte für Mathematik und Physik* **26** (1915), 132–134.

397, 398. **351** T. N. Thiele, *The Theory of Observations.* Charles & Edwin Layton, London, 1903. Reprinted in *The Annals of Mathematical Statistics* **2** (1931), 165–308.

636. **352** E. C. Titchmarsh, *The Theory of the Riemann Zeta-Function.* Clarendon Press, Oxford, 1951; second edition, revised by D. R. Heath-Brown, 1986.

636. **353** F. G. Tricomi and A. Erdélyi, "The asymptotic expansion of a ratio of gamma functions," *Pacific Journal of Mathematics* **1** (1951), 133–142.

280. **354** Peter Ungar, "Problem E 3052: A sum involving Stirling numbers," *American Mathematical Monthly* **94** (1987), 185–186.

633. **355** J. V. Uspensky, "On a problem arising out of the theory of a certain game," *American Mathematical Monthly* **34** (1927), 516–521.

238. **356** Alfred van der Poorten, "A proof that Euler missed ... Apéry's proof of the irrationality of $\zeta(3)$, an informal report," *The Mathematical Intelligencer* **1** (1979), 195–203.

169, 634. **357** A. Vandermonde, "Mémoire sur des irrationnelles de différens ordres avec une application au cercle," *Histoire de l'Académie Royale des Sciences* (1772), part 1, 71–72; *Mémoires de Mathématique et de Physique, Tirés des Registres de l'Académie Royale des Sciences* (1772), 489–498.

633, 636. **358** Ilan Vardi, "The error term in Golomb's sequence," *Journal of Number Theory* **40** (1992), 1–11.

359 J. Venn, "On the diagrammatic and mechanical representation of propositions and reasonings," *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science*, series 5, **9** (1880), 1–18. *498, 633.*

360 John Wallis, *A Treatise of Angular Sections.* Oxford, 1684. *635.*

361 Edward Waring, *Meditationes Algebraïcæ.* Cambridge, 1770; third edition, 1782. *635.*

362 William C. Waterhouse, "Problem E 3117: Even odder than we thought," *American Mathematical Monthly* **94** (1987), 691–692. *635.*

363 Frederick V. Waugh and Margaret W. Maxfield, "Side-and-diagonal numbers," *Mathematics Magazine* **40** (1967), 74–83. *635.*

364 Warren Weaver, "Lewis Carroll and a geometrical paradox," *American Mathematical Monthly* **45** (1938), 234–236. *293.*

365 H. Weber, "Leopold Kronecker," *Jahresbericht der Deutschen Mathematiker-Vereinigung* **2** (1892), 5–31. Reprinted in *Mathematische Annalen* **43** (1893), 1–25. *521.*

366 Louis Weisner, "Abstract theory of inversion of finite series," *Transactions of the American Mathematical Society* **38** (1935), 474–484. *516.*

367 Edgar M. E. Wermuth, "Die erste Fourierreihe," *Mathematische Semesterberichte* **40** (1993), 133–145. *603.*

368 Hermann Weyl, "Über die Gibbs'sche Erscheinung und verwandte Konvergenzphänomene," *Rendiconti del Circolo Matematico di Palermo* **30** (1910), 377–407. *87.*

369 F. J. W. Whipple, "Some transformations of generalized hypergeometric series," *Proceedings of the London Mathematical Society*, series 2, **26** (1927), 257–272. *634.*

370 Alfred North Whitehead, *An Introduction to Mathematics.* London and New York, 1911. *503.*

371 Alfred North Whitehead, "Technical education and its relation to science and literature," chapter 2 in *The Organization of Thought, Educational and Scientific*, London and New York, 1917. Reprinted as chapter 4 of *The Aims of Education and Other Essays*, New York, 1929. *91.*

372 Alfred North Whitehead, *Science and the Modern World.* New York, 1925. Chapter 2 reprinted in *The World of Mathematics*, edited by James R. Newman, 1956, volume 1, 402–416. *603.*

373 Herbert S. Wilf, *generatingfunctionology.* Academic Press, 1990. *575, 634.*

240, 241, 634. **374** Herbert S. Wilf and Doron Zeilberger, "An algorithmic proof theory for hypergeometric (ordinary and 'q') multisum/integral identities," *Inventiones Mathematicae* **108** (1992), 575–633.

633. **375** H. C. Williams and Harvey Dubner, "The primality of R1031," *Mathematics of Computation* **47** (1986), 703–711.

635. **376** J. Wolstenholme, "On certain properties of prime numbers," *Quarterly Journal of Pure and Applied Mathematics* **5** (1862), 35–39.

633. **377** Derick Wood, "The Towers of Brahma and Hanoi revisited," *Journal of Recreational Mathematics* **14** (1981), 17–24.

269. **378** J. Worpitzky, "Studien über die *Bernoulli*schen und *Euler*schen Zahlen," *Journal für die reine und angewandte Mathematik* **94** (1883), 203–232.

633. **379** E. M. Wright, "A prime-representing function," *American Mathematical Monthly* **58** (1951), 616–618; errata in **59** (1952), 99.

635. **380** Derek A. Zave, "A series expansion involving the harmonic numbers," *Information Processing Letters* **5** (1976), 75–77.

295. **381** E. Zeckendorf, "Représentation des nombres naturels par une somme de nombres de Fibonacci ou de nombres de Lucas," *Bulletin de la Société Royale des Sciences de Liège* **41** (1972), 179–182.

564. **382** Doron Zeilberger, "A holonomic systems approach to special functions identities," *Journal of Computational and Applied Mathematics* **32** (1990), 321–368.

229. **383** Doron Zeilberger, "The method of creative telescoping," *Journal of Symbolic Computation* **11** (1991), 195–204.

C

Credits for Exercises

THE EXERCISES in this book have been drawn from many sources. The authors have tried to trace the origins of all the problems that have been published before, except in cases where the exercise is so elementary that its inventor would probably not think anything was being invented.

Many of the exercises come from examinations in Stanford's Concrete Mathematics classes. The teaching assistants and instructors often devised new problems for those exams, so it is appropriate to list their names here:

Year	*Instructor*	*Teaching Assistant(s)*
1970	Don Knuth	Vaughan Pratt
1971	Don Knuth	Leo Guibas
1973	Don Knuth	Henson Graves, Louis Jouaillec
1974	Don Knuth	Scot Drysdale, Tom Porter
1975	Don Knuth	Mark Brown, Luis Trabb Pardo
1976	Andy Yao	Mark Brown, Lyle Ramshaw
1977	Andy Yao	Yossi Shiloach
1978	Frances Yao	Yossi Shiloach
1979	Ron Graham	Frank Liang, Chris Tong, Mark Haiman
1980	Andy Yao	Andrei Broder, Jim McGrath
1981	Ron Graham	Oren Patashnik
1982	Ernst Mayr	Joan Feigenbaum, Dave Helmbold
1983	Ernst Mayr	Anna Karlin
1984	Don Knuth	Oren Patashnik, Alex Schäffer
1985	Andrei Broder	Pang Chen, Stefan Sharkansky
1986	Don Knuth	Arif Merchant, Stefan Sharkansky

The TA sessions were invaluable, I mean really great.

Keep the same instructor and the same TAs next year.

Class notes very good and useful.

I never "got" Stirling numbers.

In addition, David Klarner (1971), Bob Sedgewick (1974), Leo Guibas (1975), and Lyle Ramshaw (1979) each contributed to the class by giving six or more guest lectures. Detailed lecture notes taken each year by the teaching assistants and edited by the instructors have served as the basis of this book.

1.1 Pólya [297, p. 120].
1.2 Scorer, Grundy, and Smith [322].
1.5 Venn [359].
1.6 Steiner [338]; Roberts [310].
1.8 Gauss [144].
1.9 Cauchy [53, note 2, theorem 17].
1.10 Atkinson [16].
1.11 Inspired by Wood [377].
1.14 Steiner [338]; Pólya [297, chapter 3]; Brother Alfred [42].
1.17 Dudeney [87, puzzle 1].
1.21 Ball [20] credits B. A. Swinden.
1.22 Based on an idea of Peter Shor.*
1.23 Bjorn Poonen.*
1.25 Frame, Stewart, and Dunkel [130].
2.2 Iverson [191, p. 11].
2.3 [207, exercise 1.2.3–2].
2.5 [207, exercise 1.2.3–25].
2.22 Binet [30, §4].
2.23 1982 final.
2.26 [207, exercise 1.2.3–26].
2.29 1979 midterm.
2.30 1973 midterm.
2.31 Stieltjes [342].
2.34 Riemann [309, §3].
2.35 Euler [106] gave a fallacious "proof" using divergent series.
2.36 Golomb [150]; Vardi [358].
2.37 Leo Moser.*
3.6 Ernst Mayr, 1982 homework.
3.8 Dirichlet [80].
3.9 Chace [54]; Fibonacci [122, pp. 77–83].
3.12 [207, exercise 1.2.4–48(a)].
3.13 Beatty [22]; Niven [281, theorem 3.7].
3.19 [207, exercise 1.2.4–34].
3.21 1975 midterm.
3.23 [207, exercise 1.2.4–41].
3.28 Brown [45].
3.30 Aho and Sloane [4].
3.31 Greitzer [165, problem 1972/3, solution 2].
3.32 [160].
3.33 1984 midterm.
3.34 1970 midterm.
3.35 1975 midterm.
3.36 1976 midterm.
3.37 1986 midterm; [215].
3.38 1974 midterm.
3.39 1971 midterm.
3.40 1980 midterm.
3.41 Klamkin [202, problem 1978/3].
3.42 Uspensky [355].
3.45 Aho and Sloane [4].
3.46 Graham and Pollak [162].
3.48 Håland and Knuth [170].
3.49 R. L. Graham and D. R. Hofstadter.*
3.52 Fraenkel [128].
3.53 S. K. Stein.*.
4.4 [214, §526].
4.16 Sylvester [346].
4.19 [212, pp. 148–149].
4.20 Bertrand [27, p. 129]; Chebyshev [56]; Wright [379].
4.22 Brillhart [39]; Williams and Dubner [375]; Dubner [86].
4.23 Crowe [68].
4.24 Legendre [241, second edition, introduction].
4.26 [208, exercise 4.5.3–43].
4.31 Pascal [284].
4.36 Hardy and Wright [181, §14.5].
4.37 Aho and Sloane [4].
4.38 Lucas [257].
4.39 [159].
4.40 Stickelberger [340].
4.41 Legendre [241, §135]; Hardy and Wright [181, theorem 82].
4.42 [208, exercise 4.5.1–6].
4.44 [208, exercise 4.5.3–39].
4.45 [208, exercise 4.3.2–13].
4.47 Lehmer [242].
4.48 Gauss [142, §78]; Crelle [67].
4.52 1974 midterm.
4.53 1973 midterm, inspired by Rao [303].
4.54 1974 midterm.

4.56 Logan [252, eq. (6.15)].
4.57 A special case appears in [216].
4.58 Sierpiński [327].
4.59 Curtiss [70]; Erdős [93].
4.60 Mills [272].
4.61 [207, exercise 1.3.2–19].
4.63 Barlow [21]; Abel [1].
4.64 Peirce [287].
4.66 Ribenboim [308]; Sierpiński [328, problem P^2_{10}].
4.67 [157].
4.69 Cramér [66].
4.70 Paul Erdős.*
4.71 [95, p. 96].
4.72 [95, p. 103].
4.73 Landau [239, volume 2, eq. 648].
5.1 Forcadel [126].
5.3 Long and Hoggatt [254].
5.5 1983 in-class final.
5.13 1975 midterm.
5.14 [207, exercise 1.2.6–20].
5.15 Dixon [81].
5.21 Euler [99].
5.25 Gauss [143, §7].
5.28 Euler [118].
5.29 Kummer [229, eq. 26.4].
5.31 Gosper [154].
5.34 Bailey [18, §10.4].
5.36 Kummer [230, p. 116].
5.37 Vandermonde [357].
5.38 [207, exercise 1.2.6–16].
5.40 Rødseth [311].
5.43 Pfaff [292]; [207, exercise 1.2.6–31].
5.48 Ranjan Roy.*
5.49 Roy [314, eq. 3.13].
5.53 Gauss [143]; Richard Askey.*
5.58 Frazer and McKellar [133].
5.59 Stanford Computer Science Comprehensive Exam, Winter 1987.
5.60 [207, exercise 1.2.6–41].
5.61 Lucas [258].
5.62 1971 midterm.
5.63 1974 midterm.
5.64 1980 midterm.
5.65 1983 midterm.
5.66 1984 midterm.
5.67 1976 midterm.
5.68 1985 midterm.
5.69 Lyle Ramshaw, guest lecture in 1986.
5.70 Andrews [9, theorem 5.4].
5.71 Wilf [373, exercise 4.16].
5.72 Hermite [185].
5.74 1979 midterm.
5.75 1971 midterm.
5.76 [207, exercise 1.2.6–59 (corrected)].
5.77 1986 midterm.
5.78 [210].
5.79 Mendelsohn [268]; Montgomery [276].
5.81 1986 final exam; [219].
5.82 Hillman and Hoggatt [188].
5.85 Hsu [190].
5.86 Good [153].
5.88 Hermite [186].
5.91 Whipple [369].
5.92 Clausen [60], [61].
5.93 Gosper [154].
5.95 Petkovšek [291, Corollary 3.1].
5.96 Petkovšek [291, Corollary 5.1].
5.98 Ira Gessel.*
5.102 H. S. Wilf.*
5.104 Volker Strehl.*
5.105 Henrici [183, p. 118].
5.108 Apéry [14].
5.109 Gessel [146].
5.110 R. William Gosper, Jr.*
5.111 [95, p. 71].
5.112 [95, p. 71].
5.113 Wilf and Zeilberger [374].
5.114 Strehl [344] credits A. Schmidt.
6.6 Fibonacci [122, p. 283].
6.15 [209, exercise 5.1.3–2].
6.21 Theisinger [350].
6.25 Gardner [138] credits Denys Wilquin.
6.27 Lucas [257].
6.28 Lucas [259, chapter 18].
6.31 Lah [235]; R. W. Floyd.*

6.35 1977 midterm.
6.37 Shallit [324].
6.39 [207, exercise 1.2.7–15].
6.40 Klamkin [202, problem 1979/1].
6.41 1973 midterm.
6.43 Brooke and Wall [41].
6.44 Matiĭasevich [266].
6.46 Francesca [131]; Wallis [360, chapter 4].
6.47 Lucas [257].
6.48 [208, exercise 4.5.3–9(c)].
6.49 Davison [73].
6.50 1985 midterm; Rham [307]; Dijkstra [79, pp. 230–232].
6.51 Waring [361]; Lagrange [233]; Wolstenholme [376].
6.52 Eswarathasan and Levine [97].
6.53 Kaucký [200] treats a special case.
6.54 Staudt [336]; Clausen [62]; Rado [300].
6.55 Andrews and Uchimura [13].
6.56 1986 midterm.
6.57 1984 midterm, suggested by R. W. Floyd.*
6.58 [207, exercise 1.2.8–30]; 1982 midterm.
6.59 Burr [47].
6.61 1976 final exam.
6.62 Borwein and Borwein [36, §3.7].
6.63 [207, section 1.2.10]; Stanley [335, proposition 1.3.12].
6.65 Tanny [349].
6.66 [209, exercise 5.1.3–3].
6.67 Chung and Graham [59].
6.68 Logan [253].
6.69 [209, exercise 6.1–13].
6.72 Euler [110, part 2, chapter 8].
6.73 Euler [108, chapters 9 and 10]; Schröter [321].
6.75 Arnold [15].
6.76 Lengyel [248].
6.78 Logan [253].
6.79 Comic section, *Boston Herald*, August 21, 1904.
6.80 Silverman and Dunn [329].
6.82 [217].
6.83 [156], modulo a numerical error.
6.85 Burr [47].
6.86 [226.
6.87 [208, exercises 4.5.3–2 and 3].
6.88 Adams and Davison [3].
6.90 Lehmer [243].
6.92 Part (a) is from Eswarathasan and Levine [97].
7.2 [207, exercise 1.2.9–1].
7.8 Zave [380].
7.9 [207, exercise 1.2.7–22].
7.11 1971 final exam.
7.12 [209, pp. 63–64].
7.13 Raney [302].
7.15 Bell [24].
7.16 Pólya [296, p. 149]; [207, exercise 2.3.4.4–1].
7.19 [221].
7.20 Jungen [198, p. 299] credits A. Hurwitz.
7.22 Pólya [298].
7.23 1983 homework.
7.24 Myers [279]; Sedláček [323].
7.25 [208, Carlitz's proof of lemma 3.3.3B].
7.26 [207, exercise 1.2.8–12].
7.32 [95, pp. 25–26] credits L. Mirsky and M. Newman.
7.33 1971 final exam.
7.34 Tomás Feder.*
7.36 1974 final exam.
7.37 Euler [109, §50]; 1971 final exam.
7.38 Carlitz [49].
7.39 [207, exercise 1.2.9–18].
7.41 André [8]; [209, exercise 5.1.4–22].
7.42 1974 final exam.
7.44 Gross [166]; [209, exercise 5.3.1–3].
7.45 de Bruijn [75].
7.47 Waugh and Maxfield [363].
7.48 1984 final exam.
7.49 Waterhouse [362].
7.50 Schröder [320]; [207, exercise 2.3.4.4–31].

7.51 Fisher [124]; Percus [290, pp. 89–123]; Stanley [334].
7.52 Hammersley [177].
7.53 Euler [114, part 2, section 2, chapter 6, §91].
7.54 Moessner [274].
7.55 Stanley [333].
7.56 Euler [113].
7.57 [95, p. 48] credits P. Erdős and P. Turán.
8.13 Thomas M. Cover.*
8.15 [207, exercise 1.2.10–17].
8.17 Patil [286].
8.24 John Knuth (age 4) and DEK; 1975 final.
8.26 [207, exercise 1.3.3–18].
8.27 Fisher [125].
8.29 Guibas and Odlyzko [168].
8.32 1977 final exam.
8.34 Hardy [180] has an incorrect analysis leading to the opposite conclusion.
8.35 1981 final exam.
8.36 Gardner [139] credits George Sicherman.
8.38 [208, exercise 3.3.2–10].
8.39 [211, exercise 4.3(a)].
8.41 Feller [120, exercise IX.33].
8.43 [207, sections 1.2.10 and 1.3.3].
8.44 1984 final exam.
8.46 Feller [120] credits Hugo Steinhaus.
8.47 1974 final, suggested by "fringe analysis" of 2-3 trees.
8.48 1979 final exam.
8.49 Blom [32]; 1984 final exam.
8.50 1986 final exam.
8.51 1986 final exam.
8.53 Feller [120] credits S. N. Bernstein.
8.57 Lyle Ramshaw.*
8.58 Guibas and Odlyzko [168].
9.1 Hardy [179, 1.3(g)].
9.2 Part (c) is from Garfunkel [140].
9.3 [207, exercise 1.2.11.1–6].
9.6 [207, exercise 1.2.11.1–3].
9.8 Hardy [179, 1.2(iv)].
9.9 Landau [238, vol. 1, p. 60].
9.14 [207, exercise 1.2.11.3–6].
9.16 Knopp [204, edition $\geqslant 2$, §64C].
9.18 Bender [25, §3.1].
9.20 1971 final exam.
9.24 [164, §4.1.6].
9.27 Titchmarsh [352].
9.28 Glaisher [149].
9.29 de Bruijn [74, §3.7].
9.32 1976 final exam.
9.34 1973 final exam.
9.35 1975 final exam.
9.36 1980 class notes.
9.37 [208, eq. 4.5.3–21].
9.38 1977 final exam.
9.39 1975 final exam, inspired by Reich [306].
9.40 1977 final exam.
9.41 1980 final exam.
9.42 1979 final exam.
9.44 Tricomi and Erdélyi [353].
9.46 de Bruijn [74, §6.3].
9.47 1980 homework; [209, eq. 5.3.1–34].
9.48 1980 final exam.
9.49 1974 final exam.
9.50 1984 final exam.
9.51 [164, §4.2.1].
9.52 Poincaré [294]; Borel [35, p. 27].
9.53 Pólya and Szegő [299, part 1, problem 140].
9.57 Andrew M. Odlyzko.*
9.58 Henrici [182, exercise 4.9.8].
9.60 [225].
9.62 Canfield [48].
9.63 Vardi [358].
9.65 Comtet [64, chapter 5, exercise 24].
9.66 M. P. Schützenberger.*
9.67 Lieb [250]; Stanley [335, exercise 4.37(c)].
9.68 Boas and Wrench [33].

* Unpublished personal communication.

Index

(Graffiti have been indexed too.)

WHEN AN INDEX ENTRY refers to a page containing a relevant exercise, the answer to that exercise (in Appendix A) might divulge further information; an answer page is not indexed here unless it refers to a topic that isn't included in the statement of the relevant exercise. Some notations not indexed here (like $x^{\underline{n}}$, $\lfloor x \rfloor$, and $\left\langle {n \atop m} \right\rangle$) are listed on pages x and xi, just before the table of contents.

0^0, 162
$\sqrt{2}$ (≈ 1.41421), 100
$\sqrt{3}$ (≈ 1.73205), 378
$\Im$: imaginary part, 64
$\mathfrak{L}$: logarithmico-exponential functions, 442–443
$\Re$: real part, 64, 212, 451
γ (≈ 0.57722), *see* Euler's constant
Γ, *see* Gamma function
δ, 47–56
Δ: difference operator, 47–55, 241, 470–471
$\epsilon_p(n)$: largest power of p dividing n, 112–114, 146
ζ, *see* zeta function
ϑ, 219–221, 310, 347
Θ: Big Theta notation, 448
κ_m, *see* cumulants
μ, *see* Möbius function
ν, *see* nu function
π (≈ 3.14159), 26, 70, 146, 244, 485, 564, 596
$\pi(x)$, *see* pi function
σ: standard deviation, 388; *see also* Stirling's constant
$\sigma_n(x)$, *see* Stirling polynomials
ϕ (≈ 1.61803): golden ratio, 70, 97, 299–301, 310, 553
φ, *see* phi function
Φ: sum of φ, 137–139, 462–463
Ω: Big Omega notation, 448
$\sum$-notation, 22–25, 245
$\prod$-notation, 64, 106
$\bigwedge$-notation, 65
$\Longleftrightarrow$: if and only if, 68
$\Longrightarrow$: implies, 71
$\backslash$: divides, 102
$\backslash\backslash$: exactly divides, 146
$\perp$: is relatively prime to, 115
$\prec$: grows slower than, 440–443
$\succ$: grows faster than, 440–443
$\asymp$: grows as fast as, 442–443
$\sim$: is asymptotic to, 8, 428–429
$\approx$: approximates, 23
$\equiv$: is congruent to, 123–126
#: cardinality, 39
!: factorial, 111–115
¡: subfactorial, 194–200
..: interval notation, 73–74
...: ellipsis, 21, 50, 108, ...

Aaronson, Bette Jane, ix
Abel, Niels Henrik, 604, 634
Abramowitz, Milton, 42, 604
absolute convergence, 60–62, 64
absolute error, 452, 455
absolute value of complex number, 64
absorption identities, 157–158, 261
Acton, John Emerich Edward Dalberg, Baron, 66
Adams, William Wells, 604, 635
Addison-Wesley, ix

addition formula for $\binom{n}{k}$, 158–159
  analog for $\binom{n}{k}$, 268
  analogs for $\left\{{n \atop k}\right\}$ and $\left[{n \atop k}\right]$, 259, 261
  dual, 530
Aho, Alfred Vaino, 604, 633
Ahrens, Wilhelm Ernst Martin Georg, 8, 604
Akhiezer, Naum Il'ich, 604
Alfred [Brousseau], Brother Ulbertus, 607, 633
algebraic integers, 106, 147
algorithms, analysis of, 138, 413–426
  divide and conquer, 79
  Euclid's, 103, 123, 303–304
  Fibonacci's, 95, 101
  Gosper's, 224–227
  Gosper-Zeilberger, 229–241, 254–255
  greedy, 101, 295
  self-certifying, 104
Alice, 31, 408–410, 430
Allardice, Robert Edgar, 2, 604
ambiguous notation, 245
American Mathematical Society, viii
AMS Euler, ix, 657
analysis of algorithms, 138, 413–426
analytic functions, 196
ancestor, 117, 291
André, Antoine Désiré, 604, 635
Andrews, George W. Eyre, 215, 330, 530, 575, 605, 634, 635
answers, notes on, 497, 637, viii
anti-derivative operator, 48, 470–471
anti-difference operator, 48, 54, 470–471
Apéry, Roger, 238, 605, 629, 634
  numbers, 238–239, 255
approximation, *see* asymptotics
  of sums by integrals, 45, 276–277, 469–475
Archibald, Raymond Clare, 608
argument of hypergeometric, 205
arithmetic progression, 30, 376
  floored, 89–94
  sum of, 6, 26, 30–31
Armageddon, 85
Armstrong, Daniel Louis (= Satchmo), 80
Arnol'd, Vladimir Igorevich, 605, 635
art and science, 234
ascents, 267–268, 270
Askey, Richard Allen, 634
associative law, 30, 61, 64
asymptotics, 439–496
  from convergent series, 451
  of Bernoulli numbers, 286, 452
  of binomial coefficients, 248, 251, 495, 598
  of discrepancies, 492, 495
  of factorials, 112, 452, 481–482, 491
  of harmonic numbers, 276–278, 452, 480–481, 491
  of hashing, 426
  of nth prime, 110–111, 456–457, 490
  of Stirling numbers, 495, 602
  of sums, using Euler's summation formula, 469–489
  of sums, using tail-exchange, 466–469, 486–489
  of sums of powers, 491
  of wheel winners, 76, 453–454
  table of expansions, 452
  usefulness of, 76, 439
Atkinson, Michael David, 605, 633
Austin, Alan Keith, 607
automaton, 405
automorphic numbers, 520
average, 384
  of a reciprocal, 432
  variance, 423–425

B_n, *see* Bernoulli numbers
Bachmann, Paul Gustav Heinrich, 443, 462, 605
Bailey, Wilfrid Norman, 223, 548, 605, 634
Ball, Walter William Rouse, 605, 633
Banach, Stefan, 433
Barlow, Peter, 605, 634
Barton, David Elliott, 602, 609
base term, 240
baseball, 73, 148, 195, 519, 648, 653
BASIC, 173, 446
basic fractions, 134, 138
basis of induction, 3, 10–11, 320–321
Bateman, Harry, 626
Baum, Lyman Frank, 581
Beatty, Samuel, 605, 633
bee trees, 291
Beeton, Barbara Ann Neuhaus Friend Smith, viii
Bell, Eric Temple, 332, 605, 635
  numbers, 373, 493, 603
Bender, Edward Anton, 606, 636

Bernoulli, Jakob (= Jacobi = Jacques = James), 283, 470, 606
numbers, *see* Bernoulli numbers
polynomials, 367–368, 470–475
polynomials, graphs of, 473
trials, 402; *see also* coins, flipping
Bernoulli, Johann (= Jean), 622
Bernoulli numbers, 283–290
asymptotics of, 286, 452
calculation of, 288, 620
denominators of, 315
generalized, *see* Stirling polynomials
generating function for, 285, 351, 365
relation to tangent numbers, 287
table of, 284, 620
Bernshteĭn (= Bernstein), Sergeĭ Natanovich, 636
Bertrand, Joseph Louis François, 145, 606, 633
postulate, 145, 500, 550
Bessel, Friedrich Wilhelm, functions, 206, 527
Beyer, William Hyman, 606
biased coin, 401
bicycles, 260, 500
Bieberbach, Ludwig, 617
Bienaymé, Irénée Jules, 606
Big Ell notation, 444
Big Oh notation, 76, 443–449
Big Omega notation, 448
Big Theta notation, 448
bijection, 39
Bill, 408–410, 430
binary logarithm, 70
binary notation (radix 2), 11–13, 15–16, 70, 113–114
binary partitions, 377
binary search, 121, 183
binary trees, 117
Binet, Jacques Philippe Marie, 299, 303, 606, 633
binomial coefficients, 153–242
addition formula, 158–159
asymptotics of, 248, 251, 495, 598
combinatorial interpretation, 153, 158, 160, 169–170
definition, 154, 211
dual, 530
generalized, 211, 318, 530
indices of, 154
middle, 187, 255–256, 495
reciprocal of, 188–189, 246, 254
top ten identities of, 174
wraparound, 250 (exercise 75), 315
binomial convolution, 365, 367
binomial distribution, 401–402, 415, 428, 432
negative, 402–403, 428
binomial number system, 245
binomial series, generalized, 200–204, 243, 252, 363
binomial theorem, 162–163
as hypergeometric series, 206, 221
discovered mechanically, 230–233
for factorial powers, 245
special cases, 163, 199
Blom, Gunnar, 606, 636
bloopergeometric series, 243
Boas, Ralph Philip, Jr., 600, 606, 636, viii
Boggs, Wade Anthony, 195
Bohl, Piers Paul Felix [= Bol', Pirs Georgievich], 87, 606
Bois-Reymond, Paul David Gustav du, 440, 610, 617
Boncompagni, Prince Baldassarre, 613
bootstrapping, 463–466
to estimate nth prime, 456–457
Borchardt, Carl Wilhelm, 617
Borel, Émile Félix Édouard Justin, 606, 636
Borwein, Jonathan Michael, 606, 635
Borwein, Peter Benjamin, 606, 635
bound variables, 22
boundary conditions on sums,
can be difficult, 75, 86
made easier, 24–25, 159
bowling, 6
box principle, 95, 130, 512
bracket notation,
for coefficients, 197, 331
for true/false values, 24–25
Brahma, Tower of, 1, 4, 278
Branges, Louis de, 617
Brent, Richard Peirce, 306, 525, 564, 606
bricks, 313, 374
Brillhart, John David, 606, 633
Brocot, Achille, 116, 607
Broder, Andrei Zary, 632, ix
Brooke, Maxey, 607, 635
Brousseau, Brother Alfred, 607, 633
Brown, Mark Robbin, 632
Brown, Morton, 501, 607

Brown, Roy Howard, ix
Brown, Thomas Craig, 607, 633
Brown, Trivial, 607
Brown, William Gordon, 607
Brown University, ix
Browning, Elizabeth Barrett, 320
Bruijn, Nicolaas Govert de, 444, 447, 500, 609, 635, 636
  cycle, 500
bubblesort, 448
Buckholtz, Thomas Joel, 620
Bulwer-Lytton, Edward George Earle Lytton, Baron, v
Burma-Shave, 541
Burr, Stefan Andrus, 607, 635

calculators, 67, 77, 459
  failure of, 344
calculus, vi, 33
  finite and infinite, 47–56
candy, 36
Canfield, Earl Rodney, 602, 607, 636
cards,
  shuffling, 437
  stacking, 273–274, 280, 309
Carlitz, Leonard, 607, 635
Carroll, Lewis (= Dodgson, Rev. Charles Lutwidge), 31, 293, 607, 608, 630
carries,
  across the decimal point, 70
  in divisibility of $\binom{m+n}{m}$, 245, 536
  in Fibonacci number system, 297, 561
Cassini, Jean Dominique, 292, 607
  identity, 292–293, 300
  identity, converse, 314
  identity, generalized, 303, 310
Catalan, Eugène Charles, 203, 361, 607
Catalan numbers, 203
  combinatorial interpretations, 358–360, 565, 568
  generalized, 361
  in sums, 181, 203, 317
  table of, 203
Cauchy, Augustin Louis, 607, 633
Čech, Eduard, vi
ceiling function, 67–69
  converted to floor, 68, 96
  graph of, 68
center of gravity, 273–274, 309
certificate of correctness, 104
Chace, Arnold Buffum, 608, 633
Chaimovich, Mark, 608
chain rule, 54, 483
change, 327–330, 374
  large amounts of, 344–346, 492
changing the index of summation, 30–31, 39
changing the tails of a sum, 466–469
cheating, viii, 195, 388, 401
  not, 158, 323
Chebyshev, Pafnutiĭ L'vovich, 38, 145, 608, 633
  inequality, 390–391, 428, 430
  monotonic inequalities, 38, 576
cheese slicing, 19
Chen, Pang-Chieh, 632
Chinese Remainder Theorem, 126, 146
Chu Shih-Chieh [= Zhū Shìjié], 169
Chung, Fan-Rong King, ix, 608, 635
Clausen, Thomas, 608, 634, 635
  product identities, 253
clearly, clarified, 417–418, 581
clichés, 166, 324, 357
closed form, 3, 7, 321
  for generating functions, 317
  not, 108, 573
  pretty good, 346
closed interval, 73–74
Cobb, Tyrus Raymond, 195
coefficient extraction, 197, 331
Cohen, Henri José, 238
coins, 327–330
  biased, 401
  fair, 401, 430
  flipping, 401–410, 430–432, 437–438
  spinning, 401
Collingwood, Stuart Dodgson, 608
Collins, John, 624
Colombo, Cristoforo (= Columbus, Christopher), 74
coloring, 496
Columbia University, ix
combinations, 153
common logarithm, 449
commutative law, 30, 61, 64
  failure of, 322, 502, 551
  relaxed, 31
complete graph, 368
complex factorial powers, 211
complex numbers, 64
  roots of unity, 149, 204, 375, 553, 574, 598

composite numbers, 105, 518
composition of generating functions, 428
computer algebra, 42, 268, 501, 539
Comtet, Louis, 609, 636
Concrete Math Club, 74
concrete mathematics, defined, vi
conditional convergence, 59
conditional probability, 416–419, 424–425
confluent hypergeometric series, 206, 245
congruences, 123–126
Connection Machine, 131
contiguous hypergeometrics, 529
continuants, 301–309, 501
  and matrices, 318–319
  Euler's identity for, 303, 312
  zero parameters in, 314
continued fractions, 301, 304–309, 319
  large partial quotients of, 553, 563, 564, 602
convergence,
  absolute, 60–62, 64
  conditional, 59
  of power series, 206, 331, 451, 532
convex regions, 5, 20, 497
convolution, 197, 246, 333, 353–364
  binomial, 365, 367
  identities for, 202, 272, 373
  polynomials, 373
  Stirling, 272, 290
  Vandermonde, *see* Vandermonde convolution
Conway, John Horton, 410, 609
cotangent function, 286, 317
counting,
  combinations, 153
  cycle arrangements, 259–262
  derangements, 193–196, 199–200
  integers in intervals, 73–74
  necklaces, 139–141
  parenthesized formulas, 357–359
  permutations, 111
  permutations by ascents, 267–268
  permutations by cycles, 262
  set partitions, 258–259
  spanning trees, 348–350, 356, 368–369, 374
  with generating functions, 320–330
coupon collecting, 583
Cover, Thomas Merrill, 636
Coxeter, Harold Scott Macdonald, 605
Cramér, Carl Harald, 525, 609, 634
Cray X-MP, 109
Crelle, August Leopold, 609, 633
cribbage, 65
Crispin, Mark Reed, 628
Crowe, Donald Warren, 609, 633
crudification, 447
Csirik, János András, 590, 609
cubes, sum of consecutive, 51, 63, 283, 289, 367
cumulants, 397–401
  infinite, 576
  of binomial distribution, 432
  of discrete distribution, 438
  of Poisson distribution, 428–429
  third and fourth, 429, 579, 589
CUNY (= City University of New York), ix
Curtiss, David Raymond, 609, 634
cycles,
  de Bruijn, 500
  of beads, 139–140
  of permutations, 259–262
cyclic shift, 12
cyclotomic polynomials, 149

D, *see* derivative operator
Dating Game, 506
David, Florence Nightingale, 602, 609
Davis, Philip Jacob, 609
Davison, John Leslie, 307, 604, 609, 635
de Branges, Louis, 617
de Bruijn, Nicolaas Govert, 444, 447, 500, 609, 635, 636
  cycle, 500
de Finetti, Bruno, 24, 613
de Lagny, Thomas Fantet, 304, 621
de Moivre, Abraham, 297, 481, 609
Dedekind, Julius Wilhelm Richard, 136–137, 609
definite sums, analogous to definite integrals, 49–50
deg, 226, 232
degenerate hypergeometric series, 209–210, 216, 222, 247
derangements, 194–196
  generating function, 199–200
derivative operator, 47–49
  converting between D and Δ, 470–471
  converting between D and ϑ, 310
  with generating functions, 33, 333, 364–365
  with hypergeometric series, 219–221
descents, *see* ascents
dgf: Dirichlet generating function, 370

dice, 381–384
  fair, 382, 417, 429
  loaded, 382, 429, 431
  nonstandard, 431
  pgf for, 399–400
  probability of doubles, 427
  supposedly fair, 392
Dickson, Leonard Eugene, 510, 609
Dieudonné, Jean Alexandre, 523
difference operator, 47–55, 241
  converting between D and Δ, 470–471
  nth difference, 187–192, 280–281
  nth difference of product, 571
differentiably finite power series, 374, 380
differential operators, *see* derivative operator, theta operator
difficulty measure for summation, 181
Dijkstra, Edsger Wybe, 173, 609, 635
dimers and dimes, 320, *see* dominoes and change
diphages, 434, 438
Dirichlet, Peter Gustav Lejeune, 370, 610, 633
  box principle, 95, 130, 512
  generating functions, 370–371, 373, 432, 451
  probability generating functions, 432
discrepancy, 88–89, 97
  and continued fractions, 319, 492, 602
  asymptotics of, 492, 495
discrete probability, 381–438
  defined, 381
disease, 333
distribution,
  of fractional parts, 87
  of primes, 111
  of probabilities, *see* probability distributions
  of things into groups, 83–85
distributive law, 30, 35, 60, 64
  for gcd and lcm, 145
  for mod, 83
divergent sums, 57, 60
  considered useful, 346–348, 451
  illegitimate, 504, 532
divide and conquer, 79
divides exactly, 146
  in binomial coefficients, 245
  in factorials, 112–114, 146
divisibility, 102–105
  by 3, 147
  of polynomials, 225
Dixon, Alfred Cardew, 610, 634
  formula, 214
DNA, Martian, 377
Dodgson, Charles Lutwidge, *see* Carroll
domino tilings, 320–327, 371, 379
  ordered pairs of, 375
Dorothy Gale, 581
double generating functions, *see* super generating functions
double sums, 34–41, 246, 249
  considered useful, 46, 183–185
  faulty use of, 63, 65
  infinite, 61
  over divisors, 105
  telescoping, 255
doubloons, 436–437
doubly exponential recurrences, 97, 100, 101, 109
doubly infinite sums, 59, 98, 482–483
Dougall, John, 171, 610
downward generalization, 2, 95, 320–321
Doyle, Sir Arthur Conan, 162, 228–229, 405, 610
drones, 291
Drysdale, Robert Lewis (Scot), III, 632
du Bois-Reymond, Paul David Gustav, 440, 610, 617
duality, 69
  between $\binom{n}{k}$ and $1/n\binom{n-1}{k}$, 530
  between factorial and Gamma functions, 211
  between floors and ceilings, 68–69, 96
  between gcd and lcm, 107
  between rising and falling powers, 63
  between Stirling numbers of different kinds, 267
Dubner, Harvey, 610, 631, 633
Dudeney, Henry Ernest, 610, 633
Dunkel, Otto, 614, 633
Dunn, Angela Fox, 627, 635
Dunnington, Guy Waldo, 610
duplication formulas, 186, 244
Dupré, Lyn Oppenheim, ix
Durst, Lincoln Kearney, viii
Dyson, Freeman John, 172, 239, 610, 615

e (≈ 2.71828),
  as canonical constant, 70, 596
  representations of, 122, 150
e_n, *see* Euclid numbers
E: expected value, 385–386
E: shift operator, 55, 188, 191

E_n, *see* Euler numbers
Edwards, Anthony William Fairbank, 610
eeny-meeny-miny-mo, *see* Josephus problem
efficiency, different notions of, 24, 133
egf: exponential generating function, 364
eggs, 158
Egyptian mathematics, 95, 150
  bibliography of, 608
Einstein, Albert, 72, 307
Eisele, Carolyn, 624–625
Eisenstein, Ferdinand Gotthold Max, 202, 610
Ekhad, Shalosh B, 546
elementary events, 381–382
Elkies, Noam David, 131, 610
ellipsis ($\cdots$), 21
  advantage of, 21, 25, 50
  disadvantage of, 25
  elimination of, 108
empirical estimates, 391–393, 427
empty case,
  for spanning trees, 349, 565
  for Stirling numbers, 258
  for tilings, 320–321
  for Tower of Hanoi, 2
empty product, 48, 106, 111
empty sum, 24, 48
entier function, *see* floor function
equality, one-way, 446–447, 489–490
equivalence relation, 124
Eratosthenes, sieve of, 111
Erdélyi, Arthur, 629, 636
Erdős, Pál (= Paul), 418, 525, 548, 575, 610–611, 634, 636
error function, 166
errors, absolute versus relative, 452, 455
errors, locating our own, 183
Eswarathasan, Arulappah, 611, 635
Euclid (= $E\acute{\upsilon}\kappa\lambda\epsilon\acute{\iota}\delta\eta\varsigma$), 107–108, 147, 611
  algorithm, 103–104, 123, 303–304
  numbers, 108–109, 145, 147, 150, 151
Euler, Leonhard, i, vii, ix, 6, 48, 122, 132–134, 202, 205, 207, 210, 267, 277, 278, 286, 299, 301–303, 469, 471, 513, 529, 575, 603, 605, 609, 611–613, 629, 633–636
  constant (≈ 0.57722), 278, 306–307, 319, 481, 596
  disproved conjecture, 131
  identity for continuants, 303, 312
  identity for hypergeometrics, 244
  numbers, 559, 570, 620; *see also* Eulerian numbers
  polynomials, 574
  pronunciation of name, 147
  summation formula, 469–475
  theorem, 133, 142, 147
  totient function, *see* phi function
  triangle, 268, 316
Eulerian numbers, 267–271, 310, 316, 378, 574
  combinatorial interpretations, 267–268, 557
  generalized, 313
  generating function for, 351
  second-order, 270–271
  table of, 268
event, 382
eventually positive function, 442
exact cover, 376
exactly divides, 146
  in binomial coefficients, 245
  in factorials, 112–114, 146
excedances, 316
exercises, levels of, viii, 72–73, 95, 511
exp: exponential function, 455
expectation, *see* expected value
expected value, 385–387
  using a pgf, 395
exponential function, discrete analog of, 54
exponential generating functions, 364–369, 421–422
exponential series, generalized, 200–202, 242, 364, 369
exponents, laws of, 52, 63

F, *see* hypergeometric functions
F_n, *see* Fibonacci numbers
factorial expansion of binomial coefficients, 156, 211
factorial function, 111–115, 346–348
  approximation to, *see* Stirling's approximation
  duplication formula, 244
  generalized to nonintegers, 192, 210–211, 213–214, 316
factorial powers, *see* falling factorial powers, rising factorial powers
factorization into primes, 106–107, 110
factorization of summation conditions, 36
fair coins, 401, 430
fair dice, 382, 417

falling factorial powers, 47
  binomial theorem for, 245
  complex, 211
  difference of, 48, 53, 188
  negative, 52, 63, 188
  related to ordinary powers, 51, 262–263, 598
  related to rising powers, 63, 312
  summation of, 50–53
fans, ix, 193, 348
Farey, John, series, 118–119, 617
  consecutive elements of, 118–119, 150
  distribution of, 152
  enumeration of, 134, 137–139, 462–463
Faulhaber, Johann, 288, 613, 620
Feder, Tomás, 635
Feigenbaum, Joan, 632
Feller, William, 381, 613, 636
Fermat, Pierre de, 130, 131, 613
  numbers, 131–132, 145, 525
Fermat's Last Theorem, 130–131, 150, 524, 555
Fermat's theorem (= Fermat's Little Theorem), 131–133, 141–143, 149
  converse of, 132, 148
Fibonacci, Leonardo, 95, 292, 549, 613, 633, 634
  addition, 296–297, 317
  algorithm, 95, 101
  factorial, 492
  multiplication, 561
  number system, 296–297, 301, 307, 310, 318
  odd and even, 307–308
Fibonacci numbers, 290–301, 575
  and continuants, 302
  and sunflowers, 291
  closed forms for, 299–300, 331
  combinatorial interpretations of, 291–292, 302, 321, 549
  egf for, 570
  ordinary generating functions for, 297–300, 337–340, 351
  second-order, 375
  table of, 290, 293
Fibonomial coefficients, 318, 556
Fine, Henry Burchard, 625
Fine, Nathan Jacob, 603
Finetti, Bruno de, 24, 613
finite calculus, 47–56
finite state language, 405
Finkel, Raphael Ari, 628
Fisher, Michael Ellis, 613, 636
Fisher, Sir Ronald Aylmer, 613, 636
fixed points, 12, 393–394
  pgf for, 400–401, 428
flipping coins, 401–410, 430–432, 437–438
floor function, 67–69
  converted to ceiling, 68, 96
  graph of, 68
Floyd, Robert W, 634, 635
food, *see* candy, cheese, eggs, pizza, sherry
football, 182
football victory problem, 193–196, 199–200, 428
  generalized, 429
  mean and variance, 393–394, 400–401
Forcadel, Pierre, 613, 634
formal power series, 206, 331, 348, 532
FORTRAN, 446
Fourier, Jean Baptiste Joseph, 22, 613
  series, 495
fractional parts, 70
  in Euler's summation formula, 470
  in polynomials, 100
  related to mod, 83
  uniformly distributed, 87
fractions, 116–123
  basic, 134, 138
  continued, 301, 304–309, 319, 564
  partial, *see* partial fraction expansions
  unit, 95, 150
  unreduced, 134–135, 151
Fraenkel, Aviezri S, 515, 563, 613–614, 633
Frame, James Sutherland, 614, 633
Francesca, Piero della, 614, 635
Franel, Jérome, 614
  number, 549
Fraser, Alexander Yule, 2, 604
Frazer, William Donald, 614, 634
Fredman, Michael Lawrence, 513, 614
free variables, 22
Freĭman, Grigoriĭ Abelevich, 608
friendly monster, 545
frisbees, 434–435, 437
Frye, Roger Edward, 131
Fundamental Theorem of Algebra, 207
Fundamental Theorem of Arithmetic, 106–107
Fundamental Theorem of Calculus, 48
Fuss, Nicolaĭ Ivanovich, 361, 614
  Fuss–Catalan numbers, 361
Fuss, Paul Heinrich von [= Fus, Pavel Nikolaeich], 611–612

Gale, Dorothy, 581
games, *see* bowling, cards, cribbage, dice, Penny ante, sports
Gamma function, 210–214, 609
  duplication formula for, 528
  Stirling's approximation for, 482
gaps between primes, 150–151, 525
Gardner, Martin, 614, 634, 636
Garfunkel, Jack, 614, 636
Gasper, George, Jr., 223, 614
Gauß (= Gauss), Karl (= Carl) Friedrich, vii, 6, 7, 123, 205, 207, 212, 501, 510, 529, 610, 615, 633, 634
  hypergeometric series, 207
  identity for hypergeometrics, 222, 247, 539
  trick, 6, 30, 112, 313
gcd, 103, *see* greatest common divisor
generalization, 11, 13, 16
  downward, 2, 95, 320–321
generalized binomial coefficients, 211, 318, 530
generalized binomial series, 200–204, 243, 252, 363
generalized exponential series, 200–202, 242, 364, 369
generalized factorial function, 192, 210–211, 213–214, 316
generalized harmonic numbers, 277, 283, 286, 370
generalized Stirling numbers, 271–272, 311, 316, 319, 598
generating functions, 196–204, 297–300, 320–380
  composition of, 428
  Dirichlet, 370–371, 373, 432, 451
  exponential, 364–369, 421–422
  for Bernoulli numbers, 285, 351, 365
  for convolutions, 197, 333–334, 353–364, 369, 421
  for Eulerian numbers, 351, 353
  for Fibonacci numbers, 297–300, 337–340, 351, 570
  for harmonic numbers, 351–352
  for minima, 377
  for probabilities, 394–401
  for simple sequences, 335
  for special numbers, 351–353
  for spectra, 307, 319
  for Stirling numbers, 351–352, 559
  Newtonian, 378
  of generating functions, 351, 353, 421
  super, 353, 421
  table of manipulations, 334
Genocchi, Angelo, 615
  numbers, 551, 574
geometric progression, 32
  floored, 114
  generalized, 205–206
  sum of, 32–33, 54
Gessel, Ira Martin, 270, 615, 634
Gibbs, Josiah Willard, 630
Gilbert, William Schwenck, 444
Ginsburg, Jekuthiel, 615
Glaisher, James Whitbread Lee, 615, 636
  constant (≈ 1.28243), 595
God, 1, 307, 521
Goldbach, Christian, 611–612
  theorem, 66
golden ratio, 299, *see* phi
golf, 431
Golomb, Solomon Wolf, 460, 507, 615, 629, 633
  digit-count sum, 460–462, 490 (exercise 22), 494
  self-describing sequence, 66, 495
Good, Irving John, 615, 634
Goodfellow, Geoffrey Scott, 628
Gopinath, Bhaskarpillai, 501, 621
Gordon, Peter Stuart, ix
Gosper, Ralph William, Jr., 224, 564, 615, 634
  algorithm, 224–227
  algorithm, examples, 227–229, 245, 247–248, 253–254, 534
Gosper-Zeilberger algorithm, 229–241, 319
  examples, 254–255, 547
  summary, 233
goto, considered harmful, 173
Gottschalk, Walter Helbig, vii
graffiti, vii, ix, 59, 637
Graham, Cheryl, ix
Graham, Ronald Lewis, iii, iv, vi, ix, 102, 506, 608–609, 611, 615–616, 629, 632, 633, 635
Grandi, Luigi Guido, 58, 616
Granville, Andrew James, 548
graph theory, *see* spanning trees
graphs of functions,
  $1/x$, 262–263
  $e^{-x^2/10}$, 483
  Bernoulli polynomials, 473
  floor and ceiling, 68
  hyperbola, 440
  partial sums of a sequence, 345–346
Graves, William Henson, 632

gravity, center of, 273–274, 309
Gray, Frank, code, 497
greatest common divisor, 92, 103–104, 107, 145
greatest integer function, *see* floor function
greatest lower bound, 65
greed, 74, 387–388; *see also* rewards
greedy algorithm, 101, 295
Green, Research Sink, 607
Greene, Daniel Hill, 616
Greitzer, Samuel Louis, 616, 633
Gross, Oliver Alfred, 616, 635
Grünbaum, Branko, 498, 616
Grundy, Patrick Michael, 627, 633
Guibas, Leonidas Ioannis (= Leo John), 590, 616, 632, 636
Guy, Richard Kenneth, 523, 525, 616

H_n, *see* harmonic numbers
Haar, Alfréd, vii
Hacker's Dictionary, 124, 628
Haiman, Mark, 632
Håland, Inger Johanne, 616, 633
half-open interval, 73–74
Hall, Marshall, Jr., 616
Halmos, Paul Richard, v, vi, 616–617
Halphen, Georges Henri, 305, 617
halving, 79, 186–187
Hamburger, Hans Ludwig, 591, 617
Hammersley, John Michael, v, 617, 636
Hanoi, Tower of, 1–4, 26–27, 109, 146
  variations on, 17–20
Hansen, Eldon Robert, 42, 617
Hardy, Godfrey Harold, 111, 442–443, 617, 633, 636
harmonic numbers, 29, 272–282
  analogous to logarithms, 53
  asymptotics of, 276–278, 452, 480–481, 491
  complex, 311, 316
  divisibility of, 311, 314, 319
  generalized, 277, 283, 286, 370
  generating function for, 351–352
  second-order, 277, 280, 311, 550–552
  sums of, 41, 313, 316, 354–355
  sums using summation by parts, 56, 279–282, 312
  table of, 273
harmonic series, divergence of, 62, 275–276
Harry, Matthew Arnold, double sum, 249
hashing, 411–426, 430
hats, *see* football victory problem
hcf, 103, *see* greatest common divisor
Heath-Brown, David Rodney, 629
Heiberg, Johan Ludvig, 611
Heisenberg, Werner Karl, 481
Helmbold, David Paul, 632
Henrici, Peter Karl Eugen, 332, 545, 602, 617, 634, 636
Hermite, Charles, 538, 555, 617, 628, 634
herring, red, 497
Herstein, Israel Nathan, 8, 618
hexagon property, 155–156, 242, 251
highest common factor, *see* greatest common divisor
Hillman, Abraham P, 618, 634
Hoare, Charles Antony Richard, 28, 73, 618, 620
Hofstadter, Douglas Richard, 633
Hoggatt, Verner Emil, Jr., 618, 622, 634
Holden, Edward Singleton, 624
Holmboe, Berndt Michael, 604
Holmes, Thomas Sherlock Scott, 162, 228–229
holomorphic functions, 196
homogeneous linear equations, 239, 543
horses, 17, 18, 468, 503
Hsu, Lee-Tsch (= Lietz = Leetch) Ching-Siur, 618, 634
Hurwitz, Adolf, 635
hyperbola, 440
hyperbolic functions, 285–286
hyperfactorial, 243, 491
hypergeometric series, 204–223
  confluent, 206, 245
  contiguous, 529
  degenerate, 209–210, 216, 222, 247
  differential equation for, 219–221
  Gaussian, 207
  partial sums of, 165–166, 223–230, 224, 245
  transformations of, 216–223, 247, 253
hypergeometric terms, 224, 243, 245, 527, 575
  similar, 541

i, 22
implicit recurrences, 136–139, 193–195, 284
indefinite summation, 48–49
  by parts, 54–56
  of binomial coefficients, 161, 223–224, 246, 248, 313
  of hypergeometric terms, 224–229

independent random variables, 384, 427
  pairwise, 437
  products of, 386
  sums of, 386, 396–398
index set, 22, 30, 61
index variable, 22, 34, 60
induction, 3, 7, 10–11, 43
  backwards, 18
  basis of, 3, 320–321
  failure of, 17, 575
  important lesson about, 508, 549
inductive leap, 4, 43
infinite sums, 56–62, 64
  doubly, 59, 98, 482–483
information retrieval, 411–413
INT function, 67
insurance agents, 391
integer part, 70
integration, 45–46, 48
  by parts, 54, 472
  of generating functions, 333, 365
interchanging the order of summation, 34–41, 105, 136, 183, 185, 546
interpolation, 191–192
intervals, 73–74
invariant relation, 117
inverse modulo m, 125, 132, 147
inversion formulas, 193
  for binomial coefficients, 192–196
  for Stirling numbers, 264, 310
  for sums over divisors, 136–139
irrational numbers, 238
  continued fraction representations, 306
  rational approximations to, 122–123
  spectra of, 77, 96, 514
  Stern–Brocot representations, 122–123
Iverson, Kenneth Eugene, 24, 67, 618, 633
  convention, 24–25, 31, 34, 68, 75

Jacobi, Carl Gustav Jacob, 64, 618
  polynomials, 543, 605
Janson, Carl Svante, 618
Jarden, Dov, 556, 618
Jeopardy, 361
joint distribution, 384
Jonassen, Arne Tormod, 618
Jones, Bush, 618
Josephus, Flavius, 8, 12, 19–20, 618
  numbers, 81, 97, 100
  problem, 8–17, 79–81, 95, 100, 144
  recurrence, generalized, 13–16, 79–81, 498
  subset, 20
Jouaillec, Louis Maurice, 632
Jungen, Reinwald, 618, 635

K, *see* continuants
Kafkaesque scenario, 274
Kaplansky, Irving, 8, 618
Karamata, Jovan, 257, 618
Karlin, Anna Rochelle, 632
Kaucký, Josef, 618, 635
Keiper, Jerry Bruce, 619
Kellogg, Oliver Dimon, 609
Kent, Clark (= Kal-El), 372
kernel functions, 370
Ketcham, Henry King, 148
kilometers, 301, 310, 550
Kilroy, James Joseph, vii
Kipling, Joseph Rudyard, 260
Kissinger, Henry Alfred, 379
Klamkin, Murray Seymour, 619, 633, 635
Klarner, David Anthony, 632
knockout tournament, 432–433
Knoebel, Robert Arthur, 619
Knopp, Konrad, 619, 636
Knuth, Donald Ervin, iii–vi, viii, ix, 102, 267, 411, 506, 553, 616, 618–620, 632, 633, 636, 657
  numbers, 78, 97, 100
Knuth, John Martin, 636
Knuth, Nancy Jill Carter, ix
Kramp, Christian, 111, 620
Kronecker, Leopold, 521
  delta notation, 24
Kruk, John Martin, 519
Kummer, Ernst Eduard, 206, 529, 620–621, 634
  formula for hypergeometrics, 213, 217, 535
Kurshan, Robert Paul, 501, 621

L_n, *see* Lucas numbers
Lagny, Thomas Fantet de, 304, 621
Lagrange (= de la Grange), Joseph Louis, comte, 470, 621, 635
  identity, 64
Lah, Ivo, 621, 634
Lambert, Johann Heinrich, 201, 363, 613, 621
Landau, Edmund Georg Hermann, 443, 448, 621, 634, 636
Laplace, Pierre Simon, marquis de, 466, 606, 621
last but not least, 132, 469

Law of Large Numbers, 391
lcm, 103, *see* least common multiple
leading coefficient, 235
least common multiple, 103, 107, 145
  of $\{1, \ldots, n\}$, 251, 319, 500
least integer function, *see* ceiling function
least upper bound, 57, 61
LeChiffre, Mark Well, 148
left-to-right maxima, 316
Legendre, Adrien Marie, 621, 633
  polynomials, 543, 573, 575
Lehmer, Derrick Henry, 526, 622, 633, 635
Leibniz, Gottfried Wilhelm, Freiherr von, vii, 168, 616, 622
Lekkerkerker, Cornelius Gerrit, 622
Lengyel, Tamás Lóránt, 622, 635
levels of problems, viii, 72–73, 95, 511
Levine, Eugene, 611, 635
lexicographic order, 441
lg: binary logarithm, 70
L'Hospital, Guillaume François Antoine de, marquis de Sainte Mesme, rule, 340, 396, 542
Lǐ Shànlán Rénshū [= Qiūrèn], 269, 622
Liang, Franklin Mark, 632
Lieb, Elliott Hershel, 622, 636
lies, and statistics, 195
Lincoln, Abraham, 401
linear difference operators, 240
lines in the plane, 4–8, 17, 19
Liouville, Joseph, 136–137, 622
little oh notation, 448
  considered harmful, 448–449
Littlewood, John Edensor, 239
ln: natural logarithm, 276
  discrete analog of, 53–54
  sum of, 481–482
log: common logarithm, 449
Logan, Benjamin Franklin (= Tex), Jr., 287, 622, 634–635
logarithmico-exponential functions, 442–443
logarithms, 449
  binary, 70
  discrete analog of, 53–54
  in O-notation, 449
  natural, 276
Long, Calvin Thomas, 622, 634
lottery, 387–388, 436–437
Loú Shìtūo, 622
lower index of binomial coefficient, 154
  complex valued, 211
lower parameters of hypergeometric series, 205
Loyd, Samuel, 560, 622
Lucas, François Édouard Anatole, 1, 292, 622–623, 633–635
  numbers, 312, 316, 556
Łuczak, Tomasz Jan, 618
Lyness, Robert Cranston, 501, 623

Maclaurin, Colin, 469, 623
MacMahon, Maj. Percy Alexander, 140, 623
magic tricks, 293
Mallows, Colin Lingwood, 506
Markov, Andreĭ Andreevich (the elder), processes, 405
Martian DNA, 377
Martzloff, Jean-Claude, 623
mathematical induction, 3, 7, 10–11, 43
  backwards, 18
  basis of, 3, 320–321
  failure of, 17, 575
  important lesson about, 508, 549
Mathews, Edwin Lee (= 41), 8, 21, 94, 105, 106, 343
Matiĭasevich (= Matijasevich), Ĭuriĭ (= Yuri) Vladimirovich, 294, 623, 635
Mauldin, Richard Daniel, 611
Maxfield, Margaret Waugh, 630, 635
Mayr, Ernst, ix, 632, 633
McEliece, Robert James, 71
McGrath, James Patrick, 632
McKellar, Archie Charles, 614, 634
mean (average) of a probability distribution, 384–399
median, 384, 385, 437
mediant, 116
Melzak, Zdzislaw Alexander, vi, 623
Mendelsohn, Nathan Saul, 623, 634
Merchant, Arif Abdulhussein, 632
merging, 79, 175
Mersenne, Marin, 109–110, 131, 613, 623
  numbers, 109–110, 151, 292
  primes, 109–110, 127, 522–523
Mertens, Franz Carl Joseph, 139, 623
  constant, 23
miles, 301, 310, 550
Mills, Stella, 623
Mills, William Harold, 623, 634
minimum, 65, 249, 377

Mirsky, Leon, 635
mixture of probability distributions, 428
mnemonics, 74, 164
Möbius, August Ferdinand, 136, 138, 623
  function, 136–139, 145, 149, 370–371, 462–463
mod: binary operation, 81–85
mod: congruence relation, 123–126
mod 0, 82–83, 515
mode, 384, 385, 437
modular arithmetic, 123–129
modulus, 82
Moessner, Alfred, 624, 636
Moivre, Abraham de, 297, 481, 609
moments, 398–399
Montgomery, Hugh Lowell, 463, 624
Montgomery, Peter Lawrence, 624, 634
Moriarty, James, 162
Morse, Samuel Finley Breese, code, 302–303, 324, 551
Moser, Leo, 624, 633
Motzkin, Theodor Samuel, 556, 564, 618, 624
mountain ranges, 359, 565
mu function, *see* Möbius function
multinomial coefficients, 168, 171–172, 569
  recurrence for, 252
multinomial theorem, 149, 168
multiple of a number, 102
multiple sums, 34–41, 61; *see also* double sums
multiple-precision numbers, 127
multiplicative functions, 134–136, 144, 371
multisets, 77, 270
mumble function, 83, 84, 88, 507, 513
Murdock, Phoebe James, viii
Murphy's Law, 74
Myers, Basil Roland, 624, 635

name and conquer, 2, 32, 88, 139
National Science Foundation, ix
natural logarithm, 53–54, 276, 481–482
Naval Research, ix
Navel research, 299
nearest integer, 95
  rounding to, 195, 300, 344, 491
  unbiased, 507
necessary and sufficient conditions, 72
necklaces, 139–141, 259
negating the upper index, 164–165
negative binomial distribution, 402–403, 428
negative factorial powers, 52, 63, 188
Newman, James Roy, 630
Newman, Morris, 635
Newton, Sir Isaac, 189, 277, 624
  series, 189–192
Newtonian generating function, 378
Niven, Ivan Morton, 332, 624, 633
nonprime numbers, 105, 518
nontransitive paradox, 410
normal distribution, 438
notation, x–xi, 2, 637
  extension of, 49, 52, 154, 210–211, 266, 271, 311, 319
  ghastly, 67, 175
  need for new, 83, 115, 267
nu function: sum of digits,
  binary (radix 2), 12, 114, 250, 525, 557
  other radices, 146, 525, 552
null case,
  for spanning trees, 349, 565
  for Stirling numbers, 258
  for tilings, 320–321
  for Tower of Hanoi, 2
number system, 107, 119
  binomial, 245
  Fibonacci, 296–297, 301, 307, 310, 318
  prime-exponent, 107, 116
  radix, *see* radix notation
  residue, 126–129, 144
  Stern–Brocot, *see* Stern–Brocot number system
number theory, 102–152

o, considered harmful, 448–449
O-notation, 76, 443–449
  abuse of, 447–448, 489
  one-way equalities with, 446–447, 489–490
obvious, clarified, 417, 526
odds, 410
Odlyzko, Andrew Michael, 81, 564, 590, 616, 624, 636
Office of Naval Research, ix
one-way equalities, 446–447, 489–490
open interval, 73–74, 96
operators, 47
  anti-derivative ($\int$), 48
  anti-difference ($\sum$), 48
  derivative (D), 47, 310
  difference (Δ), 47
  equations of, 188, 191, 241, 310, 471
  shift (E, K, N), 55, 240
  theta (ϑ), 219, 310

optical illusions, 292, 293, 560
organ-pipe order, 524
Oz, Wizard of, 581

Pacioli, Luca, 614
Palais, Richard Sheldon, viii
paradoxes,
  chessboard, 293, 317
  coin flipping, 408–410
  pair of boxes, 531, 535, 539
paradoxical sums, 57
parallel summation, 159, 174, 208–210
parentheses, 357–359
parenthesis conventions, xi
partial fraction expansions, 298–299, 338–341
  for easy summation and differentiation, 64, 376, 476, 504, 586
  not always easiest, 374
  of $1/x^{\underline{x+n}}$, 189
  of $1/(z^n - 1)$, 558
  powers of, 246, 376
partial quotients, 306
  and discrepancies, 319, 598–599, 602
  large, 553, 563, 564, 602
partial sums, *see* indefinite summation
  required to be positive, 359–362
partition into nearly equal parts, 83–85
partitions, of the integers, 77–78, 96, 99, 101
  of a number, 330, 377
  of a set, 258–259, 373
Pascal, Blaise, 155, 156, 624, 633
Pascal's triangle, 155
  extended upward, 164
  hexagon property, 155–156, 242, 251
  row lcms, 251
  row products, 243
  row sums, 163, 165–166
  variant of, 250
Patashnik, Amy Markowitz, ix
Patashnik, Oren, iii, iv, vi, ix, 102, 506, 616, 632
Patil, Ganapati Parashuram, 624, 636
Paule, Peter, 537, 546
Peirce, Charles Santiago Sanders, 525, 624–625, 634
  sequence, 151
Penney, Walter Francis, 408, 625
Penney ante, 408–410, 430, 437, 438
pentagon, 314 (exercise 46), 430, 434
pentagonal numbers, 380
Percus, Jerome Kenneth, 625, 636
perfect powers, 66
periodic recurrences, 20, 179, 498
permutations, 111–112
  ascents in, 267–268, 270
  cycles in, 259–262
  excedances in, 314
  fixed points in, 193–196, 393–394, 400–401, 418
  left-to-right maxima in, 314
  random, 393–394, 400–401, 428
  up-down, 377
  without fixed points, *see* derangements
personal computer, 109
perspiration, 234–235
perturbation method, 32–33, 43–44, 64, 179, 284–285
Petkovšek, Marko, 229, 575, 625, 634
Pfaff, Johann Friedrich, 207, 214, 217, 529, 625, 634
  reflection law, 217, 247, 539
pgf: probability generating function, 394
phages, 434, 438
phi (≈ 1.61803), 299–301
  as canonical constant, 70
  continued fraction for, 310
  in fifth roots of unity, 553
  in solutions to recurrences, 97, 99, 285–286
  Stern–Brocot representation of, 550
phi function, 133–135
  dgf for, 371
  divisibility by, 151
Phi function: sum of ϕ, 137–139, 462–463
Phidias, 299
philosophy, vii, 11, 16, 46, 71, 72, 75, 91, 170, 181, 194, 331, 467, 503, 508, 603
phyllotaxis, 291
pi (≈ 3.14159, 26, 286
  as canonical constant, 70, 416, 423
  large partial quotients of, 564
  Stern–Brocot representation of, 146
pi function, 110–111, 452, 593
  preposterous expressions for, 516
Pig, Porky, 496
pigeonhole principle, 130
Pincherle, Salvatore, 617
Pisano, Leonardo, 613, *see* Fibonacci
Pittel, Boris Gershon, 576, 618
pizza, 4, 423
planes, cutting, 19

pneumathics, 164
Pochhammer, Leo, 48, 625
  symbol, 48
pocket calculators, 67, 77, 459
  failure of, 344
Poincaré, Jules Henri, 625, 636
Poisson, Siméon Denis, 471, 625
  distribution, 428–429, 579
  summation formula, 602
Pollak, Henry Otto, 616, 633
Pólya, George (= György), vi, 16, 327, 508, 625, 633, 635, 636
polygons,
  dissection of, 379
  triangulation of, 374
  Venn diagrams with, 20
polynomial argument, 158, 163
  for rational functions, 527
  opposite of, 210
polynomially recursive sequence, 374
polynomials, 189
  Bernoulli, 367–368, 470–475
  continuant, 301–309
  convolution, 373
  cyclotomic, 149
  degree of, 158, 226
  divisibility of, 225
  Euler, 574
  Jacobi, 543, 605
  Legendre, 543, 573, 575
  Newton series for, 189–191
  reflected, 339
  Stirling, 271–272, 290, 311, 317, 352
Poonen, Bjorn, 501, 633
Porter, Thomas K, 632
Portland cement, *see* concrete (in another book)
power series, 196, *see* generating functions
  formal, 206, 331, 348, 532
Pr, 381–382
Pratt, Vaughan Ronald, 632
preferential arrangements, 378 (exercise 44)
primality testing, 110, 148
  impractical method, 133
prime algebraic integers, 106, 147
prime numbers, 105–111
  gaps between, 150–151, 525
  largest known, 109–110
  Mersenne, 109–110, 127, 522
  size of nth, 110–111, 456–457
  sum of reciprocals, 22–25
prime to, 115
prime-exponent representation, 107, 116
Princeton University, ix, 427
probabilistic analysis of an algorithm, 413–426
probability, 195, 381–438
  conditional, 416–419, 424–425
  discrete, 381–438
  generating functions, 394–401
  spaces, 381
probability distributions, 367
  binomial, 401–402, 415, 428, 432
  composition or mixture of, 428
  joint, 384
  negative binomial, 402–403, 428
  normal, 438
  Poisson, 428–429, 579
  uniform, 395–396, 420–421
problems, levels of, viii, 72–73, 95, 511
product notation, 64, 106
product of consecutive odd numbers, 186, 270
progression, *see* arithmetic progression, geometric progression
proof, 4, 7
proper terms, 239–241, 255–256
properties, 23, 34, 72–73
prove or disprove, 71–72
psi function, 551
pulling out the large part, 453, 458
puns, ix, 220
Pythagoras of Samos, theorem, 510

quadratic domain, 147
quicksort, 28–29, 54
quotation marks, xi
quotient, 81

rabbits, 310
radix notation, 11–13, 15–16, 109, 195, 526
  length of, 70, 460
  related to prime factors, 113–114, 146–148, 245
Rado, Richard, 625, 635
Rahman, Mizanur, 223, 614
Rainville, Earl David, 529, 626
Rāmānujan Aiyaṅgār, Srīnivāsa, 330
Ramaré, Olivier, 548
Ramshaw, Lyle Harold, 73, 632, 634, 636
random constant, 399
random variables, 383–386; *see also* independent random variables

Raney, George Neal, 359, 362, 626, 635
lemma, 359–360
lemma, generalized, 362, 372
sequences, 360–361
Rao, Dekkata Rameswar, 626, 633
rational functions, 207–208, 224–226, 338, 527
rational generating functions, 338–346
expansion theorems for, 340–341
Rayleigh, John William Strutt, 3rd Baron, 77, 626
Read, Ronald Cedric, 625
real part, 64, 212, 451
reciprocity law, 94
Recorde, Robert, 446, 626
recurrences, 1–20
and sums, 25–29
doubly exponential, 97, 100, 101, 109
floor/ceiling, 78–81
implicit, 136–138, 193–194, 284
periodic, 20, 179, 498
solving, 337–350
unfolding, 6, 100, 159–160, 312
unfolding asymptotically, 456
referee, 175
reference books, 42, 223, 616, 619
reflected light rays, 291–292
reflected polynomials, 339
reflection law for hypergeometrics, 217, 247, 539
regions, 4–8, 17, 19
Reich, Simeon, 626, 636
Reingold, Edward Martin, 70
relative error, 452, 455
relatively prime integers, 108, 115–123
remainder after division, 81–82
remainder in Euler's summation formula, 471, 474–475, 479–480
Renz, Peter Lewis, viii
repertoire method, 14–15, 19, 250
for Fibonacci-like recurrences, 312, 314, 372
for sums, 26, 44–45, 63
replicative function, 100
repunit primes, 516
residue calculus, 495
residue number system, 126–129, 144
retrieving information, 411–413
rewards, monetary, ix, 256, 497, 525, 575
Rham, Georges de, 626, 635
Ribenboim, Paolo, 555, 626, 634
Rice, Stephan Oswald, 626
Rice University, ix
Riemann, Georg Friedrich Bernhard, 205, 626, 633
hypothesis, 526
Riemann's zeta function, 65, 595
as generalized harmonic number, 277–278, 286
as infinite product, 371
as power series, 601
dgf's involving, 370–371, 373, 463, 566, 569
evaluated at integers, 238, 286, 571, 595, 597
rising factorial powers, 48
binomial theorem for, 245
complex, 211
negative, 63
related to falling powers, 63, 312
related to ordinary powers, 263, 598
Roberts, Samuel, 626, 633
rocky road, 36, 37
Rødseth, Øystein Johan, 626, 634
Rolletschek, Heinrich Franz, 514
roots of unity, 149, 204, 375, 574, 598
fifth, 553
modulo m, 128–129
Roscoe, Andrew William, 620
Rosser, John Barkley, 111, 626
Rota, Gian-Carlo, 516, 626
roulette wheel, 74–76, 453
rounding to nearest integer, 95, 195, 300, 344, 491
unbiased, 507
Roy, Ranjan, 626, 634
rubber band, 274–275, 278, 312, 493
ruler function, 113, 146, 148
running time, 413, 425–426
O-notation for, abused, 447–448
Ruzsa, Imre Zoltán, 611

Saalschütz, Louis, 214, 627, 634
identity, 214–215, 234–235, 529, 531
Saltykov, Al'bert Ivanovich, 463, 627
sample mean and variance, 391–393, 427
sample third cumulant, 429
samplesort, 354
sandwiching, 157, 165
Sárközy, András, 548, 627
Sawyer, Walter Warwick, 207, 627
Schäffer, Alejandro Alberto, 632
Schinzel, Andrzej, 525
Schlömilch, Oscar Xaver, 627
Schmidt, Asmus Lorenzen, 634

Schoenfeld, Lowell, 111, 626
Schönheim, Johanen, 608
Schröder, Ernst, 627, 635
Schrödinger, Erwin, 430
Schröter, Heinrich Eduard, 627, 635
Schützenberger, Marcel Paul, 636
science and art, 234
Scorer, Richard Segar, 627, 633
searching a table, 411–413
Seaver, George Thomas (= 41), 8, 21, 94, 105, 106, 343
secant numbers, 317, 559, 570, 620
second-order Eulerian numbers, 270–271
second-order Fibonacci numbers, 375
second-order harmonic numbers, 277, 280, 311, 550–552
Sedgewick, Robert, 632
Sedláček, Jiří, 627, 635
self-certifying algorithms, 104
self-describing sequence, 66, 495
self reference, 59, 95, 531–540, 616, 653
set inclusion in O-notation, 446–447, 490
Shallit, Jeffrey Outlaw, 627, 635
Sharkansky, Stefan Michael, 632
Sharp, Robert Thomas, 273, 627
sherry, 433
shift operator, 55, 240
  binomial theorems for, 188, 191
Shiloach, Joseph (= Yossi), 632
Shor, Peter Williston, 633
Sicherman, George Leprechaun, 636
sideways addition, 12, 114, 146, 250, 552
Sierpiński, Wacław, 87, 627, 634
sieve of Eratosthenes, 111
Sigma-notation, 22–25
  ambiguity of, 245
signum function, 502
Silverman, David L, 627, 635
similar hypergeometric terms, 541
skepticism, 71
Skiena, Steven Sol, 548
Sloane, Neil James Alexander, 42, 341, 464, 604, 628, 633
Slowinski, David Allen, 109
small cases, 2, 5, 9, 155, 320–321; *see also* empty case
Smith, Cedric Austen Bardell, 627, 633
Snowwalker, Luke, 435
Solov'ev, Aleksandr Danilovitch, 408, 628
solution, 3, 337
sorting,
  asymptotic efficiency of, 447–449
  bubblesort, 448
  merge sort, 79, 175
  possible outcomes, 378
  quicksort, 28–29, 54
  samplesort, 354
spanning trees,
  of complete graphs, 368–369
  of fans, 348–350, 356
  of wheels, 374
Spec, *see* spectra
special numbers, 257–319
spectra, 77–78, 96, 97, 99, 101
  generating functions for, 307, 319
spinning coins, 401
spiral function, 99
Spohn, William Gideon, Jr., 628
Sports, *see* baseball, football, frisbees, golf, tennis
Sprugnoli, Renzo, 564
square pyramidal numbers, 42
square root,
  of 1 (mod m), 128–129
  of 2, 100
  of 3, 378
  of -1, 22
squarefree, 145, 151, 373, 525, 548
squares, sum of consecutive, 41–46, 51, 180, 245, 269, 284, 288, 367, 444, 470
stack size, 360–361
stacking bricks, 313, 374
stacking cards, 273–274, 278, 309
Stallman, Richard Matthew, 628
standard deviation, 388, 390–394
Stanford University, v, vii, ix, 427, 458, 632, 634, 657
Stanley, Richard Peter, 270, 534, 615, 628, 635, 636
Staudt, Karl Georg Christian von, 628, 635
Steele, Guy Lewis, Jr., 628
Stegun, Irene Anne, 42, 604
Stein, Sherman Kopald, 633
Steiner, Jacob, 5, 628, 633
Steinhaus, Hugo Dyonizy, 636
Stengel, Charles Dillon (= Casey), 42
step functions, 87
Stern, Moriz Abraham, 116, 628

Stern–Brocot number system, 119–123
  related to continued fractions, 306
  representation of $\sqrt{3}$, 572
  representation of γ, 306
  representation of π, 146
  representation of ϕ, 550
  representation of e, 122, 150
  simplest rational approximations from, 122–123, 146, 519
Stern–Brocot tree, 116–123, 148, 525
  largest denominators in, 319
  related to continued fractions, 305–306
Stern–Brocot wreath, 515
Stewart, Bonnie Madison, 614, 633
Stickelberger, Ludwig, 628, 633
Stieltjes, Thomas Jan, 617, 628, 633
  constants, 595, 601
Stirling, James, 192, 195, 210, 257, 258, 297, 481, 628
  approximation, 112, 452, 481–482, 491, 496
  approximation, perturbed, 454–455
  constant, 481, 485–489
  polynomials, 271–272, 290, 311, 317, 352
  triangles, 258, 259, 267
Stirling numbers, 257–267
  as sums of products, 570
  asymptotics of, 495, 602
  combinatorial interpretations, 258–262
  convolution formulas, 272, 290
  duality of, 267
  generalized, 271–272, 311, 316, 319, 598
  generating functions for, 351–352, 559
  identities for, 264–265, 269, 272, 290, 311, 317, 378
  inversion formulas for, 310
  of the first kind, 259
  of the second kind, 258
  related to Bernoulli numbers, 289–290, 317 (exercise 76)
  table of, 258, 259, 267
Stone, Marshall Harvey, vi
Straus, Ernst Gabor, 564, 611, 624
Strehl, Karl Ernst Volker, 549, 629, 634
subfactorial, 194–196, 250
summand, 22
summation, 21–66
  asymptotic, 87–89, 466–496
  by parts, 54–56, 63, 279
  changing the index of, 30–31, 39
  definite, 49–50, 229–241
  difficulty measure for, 181
  factors, 27–29, 64, 236, 248, 275, 543
  in hypergeometric terms, 224–229
  indefinite, *see* indefinite summation
  infinite, 56–62, 64
  interchanging the order of, 34–41, 105, 136, 183, 185, 546
  mechanical, 229–241
  on the upper index, 160–161, 175–176
  over divisors, 104–105, 135–137, 141, 370
  over triangular arrays, 36–41
  parallel, 159, 174, 208–210
sums, 21–66; *see also* summation
  absolutely convergent, 60–62, 64
  and recurrences, 25–29
  approximation of, by integrals, 45, 276–277, 469–475
  divergent, *see* divergent sums
  double, *see* double sums
  doubly infinite, 59, 98, 482–483
  empty, 24, 48
  floor/ceiling, 86–94
  formal, 321; *see also* formal power series
  hypergeometric, *see* hypergeometric series
  infinite, 56–62, 64
  multiple, 34–41, 61; *see also* double sums
  notations for, 21–25
  of consecutive cubes, 51, 63, 283, 289, 367
  of consecutive integers, 6, 44, 65
  of consecutive mth powers, 42, 283–285, 288–290, 366–368
  of consecutive squares, 41–46, 51, 180, 245, 269, 284, 288, 367, 444, 470
  of harmonic numbers, 41, 56, 279–282, 312–313, 316, 354–355
  paradoxical, 57
  tails of, 466–469, 488–489, 492
Sun Tsŭ [= Sūnzĭ, Master Sun], 126
sunflower, 291
super generating functions, 353, 421
superfactorials, 149, 243
Swanson, Ellen Esther, viii
Sweeney, Dura Warren, 629
Swinden, Benjamin Alfred, 633
Sylvester, James Joseph, 133, 629, 633
symmetry identities,
  for binomial coefficients, 156–157, 183
  for continuants, 303
  for Eulerian numbers, 268

Szegedy, Márió, 525, 608, 629
Szegő, Gábor, 625, 636

T_n, *see* tangent numbers
tail exchange, 466–469, 486–489
tail inequalities, 428, 430
tail of a sum, 452–455
tale of a sum, *see* squares
tangent function, 287, 317
tangent numbers, 287, 312, 317, 620
Tanny, Stephen Michael, 629, 635
Tartaglia, Nicolò, triangle, 155
Taylor, Brook, series, 163, 191, 287, 396, 470–471
telescoping, 50, 232, 236, 255
tennis, 432–433
term, 21
  hypergeometric, 224, 243, 245, 527, 575
term ratio, 207–209, 211–212, 224–225
TEX, 219, 432, 657
Thackeray, Henry St. John, 618
Theisinger, Ludwig, 629, 634
theory of numbers, 102–152
theory of probability, 381–438
theta functions, 483, 524
theta operator, 219–221, 347
  converting between D and ϑ, 310
Thiele, Thorvald Nicolai, 397, 398, 629
thinking, 503
  big, 2, 441, 458, 483, 486
  not at all, 56, 230, 503
  small, *see* downward generalization, small cases
three-dots ($\cdots$) notation, 21
  advantage of, 21, 25, 50
  disadvantage of, 25
  elimination of, 108
tilings, *see* domino tilings
Titchmarsh, Edward Charles, 629, 636
Todd, Horace, 501
Toledo, Ohio, 73
Tong, Christopher Hing, 632
totient function, 133–135
  dgf for, 371
  divisibility by, 151
  summation of, 137–144, 150, 462–463
Toto, 581
tournament, 432–433
Tower of Brahma, 1, 4, 278
Tower of Hanoi, 1–4, 26–27, 109, 146
  variations on, 17–20
Trabb Pardo, Luis Isidoro, 632
transitive law, 124
  failure of, 410
traps, 154, 157, 183, 222, 542
trees,
  2-3 trees, 636
  binary, 117
  of bees, 291
  spanning, 348–350, 356, 368–369, 374
  Stern–Brocot, *see* Stern–Brocot tree
triangular array, summation over, 36–41
triangular numbers, 6, 155, 195–196, 260, 380
triangulation, 374
Tricomi, Francesco Giacomo Filippo, 629, 636
tridiagonal matrix, 319
trigonometric functions,
  related to Bernoulli numbers, 286–287, 317
  related to probabilities, 435, 437
  related to tilings, 379
trinomial coefficients, 168, 171, 255, 571
  middle, 490
trinomial theorem, 168
triphages, 434
trivial, clarified, 129, 417–418, 618
Turán, Paul, 636
typefaces, viii–ix, 657

Uchimura, Keisuke, 605, 635
unbiased estimate, 392, 429
unbiased rounding, 507
uncertainty principle, 481
undetermined coefficients, 529
unexpected sum, 167, 215–216, 236, 247
unfolding a recurrence, 6, 100, 159–160, 312
  asymptotically, 456
Ungar, Peter, 629
uniform distribution, 395–396, 418–419
uniformity, deviation from, 152; *see also* discrepancy
unique factorization, 106–107, 147
unit, 147
unit fractions, 95, 101, 150
unwinding a recurrence, *see* unfolding a recurrence
up-down permutations, 377
upper index of binomial coefficient, 154
upper negation, 164–165
upper parameters of hypergeometric series, 205

upper summation, 160–161, 176
useless identity, 223, 254
Uspensky, James Victor, 615, 629, 633

V: variance, 387–398, 419–425
van der Poorten, Alfred Jacobus, 629
Vandermonde, Alexandre Théophile, 169, 629, 634
Vandermonde's convolution, 169–170
  as a hypergeometric series, 211–213
  combinatorial interpretation, 169–170
  derived mechanically, 234
  derived from generating functions, 198
  generalized, 201–202, 218–219, 248
  with half-integers, 187
vanilla, 36
Vardi, Ilan, 525, 548, 603, 620, 629, 633, 636
variance of a probability distribution, 387–398, 419–425
  infinite, 428, 587
Veech, William Austin, 514
Venn, John, 498, 630, 633
  diagram, 17, 20
venture capitalists, 493–494
violin string, 29
vocabulary, 75
Voltaire, de (= Arouet, François Marie), 450
von Staudt, Karl Georg Christian, 628, 635
Vyssotsky, Victor Alexander, 548

Wall, Charles Robert, 607, 635
Wallis, John, 630, 635
Wapner, Joseph Albert, 43
war, 8, 16, 85, 434
Waring, Edward, 630, 635
Waterhouse, William Charles, 630, 635
Watson, John Hamish, 229, 405
Waugh, Frederick Vail, 630, 635
Weaver, Warren, 630
Weber, Heinrich, 630
Weisner, Louis, 516, 630
Wermuth, Edgar Martin Emil, 603, 630
Weyl, Claus Hugo Hermann, 87, 630
Wham-O, 435, 443
wheel, 74, 374
  big, 75
  of Fortune, 453
Whidden, Samuel Blackwell, viii
Whipple, Francis John Welsh, 630, 634
  identity, 253
Whitehead, Alfred North, 91, 503, 603, 630
Wiles, Andrew John, 131
Wilf, Herbert Saul, 81, 240, 241, 514, 549, 575, 620, 624, 630–631, 634
Williams, Hugh Cowie, 631, 633
Wilquin, Denys, 634
Wilson, Sir John, theorem, 132–133, 148, 516, 609
Wilson, Martha, 148
wine, 433
Witty, Carl Roger, 509
Wolstenholme, Joseph, 631, 635
  theorem, 554
Wood, Derick, 631, 633
Woods, Donald Roy, 628
Woolf, William Blauvelt, viii
worm,
  and apple, 430
  on rubber band, 274–275, 278, 312, 493
Worpitzky, Julius Daniel Theodor, 631
  identity, 269
wreath, 515
Wrench, John William, Jr., 600, 606, 636
Wright, Sir Edward Maitland, 111, 617, 631, 633
Wythoff (= Wijthoff), Willem Abraham, 614

Yao, Andrew Chi-Chih, ix, 632
Yao, Foong Frances, ix, 632
Yaó, Qí, 622
Youngman, Henry (= Henny), 175

zag, *see* zig
Zagier, Don Bernard, 238
Zapf, Hermann, viii, 620, 657
Zave, Derek Alan, 631, 635
Zeckendorf, Edouard, 631
  theorem, 295–296, 563
Zeilberger, Doron, ix, 229–231, 238, 240, 241, 631, 634
zero, not considered harmful, 24–25, 159
  strongly, 24–25
zeta function, 65, 595
  and the Riemann hypothesis, 526
  as generalized harmonic number, 277–278, 286
  as infinite product, 371
  as power series, 601
  dgf's involving, 370–371, 373, 463, 566, 569
  evaluated at integers, 238, 286, 571, 595, 597
Zhu Shijie, *see* Chu Shih-Chieh
zig, 7–8, 19
zig-zag, 19
Zipf, George Kingsley, law, 419

List of Tables

Sums and differences 55
Pascal's triangle 155
Pascal's triangle extended upward 164
Sums of products of binomial coefficients 169
The top ten binomial coefficient identities 174
General convolution identities 202
Stirling's triangle for subsets 258
Stirling's triangle for cycles 259
Basic Stirling number identities 264
Additional Stirling number identities 265
Stirling's triangles in tandem 267
Euler's triangle 268
Second-order Eulerian triangle 270
Stirling convolution formulas 272
Generating function manipulations 334
Simple sequences and their generating functions 335
Generating functions for special numbers 351
Asymptotic approximations 452

THIS BOOK was composed at Stanford University using the TEX system for technical text developed by D. E. Knuth. The mathematics is set in a new typeface called AMS Euler (Version 2.1), designed by Hermann Zapf for the American Mathematical Society. The text is set in a new typeface called Concrete Roman and Italic, a special version of Knuth's Computer Modern family with weights designed to blend with AMS Euler. The paper is 50-lb.-basis Clearfield Opaque, which has a neutral pH and a life expectancy of several hundred years. The offset printing and notch binding were done by Halliday Lithograph Corporation in Hanover, Massachusetts.

Errata, 1994–1997

"And if you happen in reading to finde any more faultes not here mentioned, as peraduenture you may, ... I trust you will therefore impute no blame either vnto me or to the Printer, but gently amend and correct them, accepting our good minde, which was to haue had the booke passed to your handes vtterly without fault, as touching the Printing."
—*from the first English edition of Euclid [98]*

THIS IS A LIST of all corrections made to the second edition of *Concrete Mathematics* by Graham, Knuth, and Patashnik, after the first appearance of that book in February 1994 until the final updates to the hardcover version made at the beginning of 1998. All subsequent updates are posted on Internet page `http://www-cs-faculty.stanford.edu/~knuth/gkp.html`, where you can also find additional material such as exam questions and answers that were prepared after the book was published.

Minor changes in typographic layout are not listed here; neither are changes to the index that were precipitated by changes to the text. With these modifications, the book should be perfect.

But you may disagree. To claim a reward for any anomalies you spot that aren't shown here, please send your comments to D. E. Knuth, Computer Science Department, Stanford University, Stanford CA 94305-9045, or by email to `knuth-bug@cs.stanford.edu`, as soon as possible. Please use email only to report errors, not to ask questions.

All of the items dated before February 1995, and a few items dated after that, were corrected in the second printing.

Page 20, line 16 (10 Jan 95)

---

whose solution is periodic regardless of the initial values $X_0, \ldots, X_{k-1}$.

Page 76, line 8 from the bottom (18 Dec 96)

1,000,000,000	1500000.0	1502497	0.166

Page 87, lines 11 to 14 from the bottom (13 May 95)

for all irrational α and all bounded functions f that are continuous almost everywhere. For example, the *average* value of $\{n\alpha\}$ can be found by setting $f(x) = x$; we get $\frac{1}{2}$. (That's exactly what we might expect; but it's nice to know that it is really, provably true, no matter how irrational α is.)

Page 88, last two lines and top line of page 89 (31 Jan 95)

The quantity $j\alpha'$ will be an integer only when $j = 0$, since α (hence α') is irrational; and $j\alpha' - \nu'$ will be an integer for at most one value of j. So we can change the ceiling terms to floors:

Page 89, line 14 (31 Jan 95)

Here ϵ is a positive error less than $\nu\alpha^{-1}$. Exercise 18 proves that S is, simi-

Page 89, line 18 [the first printing was correct] (26 Sep 95)

$$D(\alpha, n) \leqslant D(\alpha', \lfloor \alpha n \rfloor) + \alpha^{-1} + 2\,. \tag{3.31}$$

Page 89, lines 11 and 12 from the bottom (26 Sep 95)

sufficiently large. Hence theorem (3.28) is true; however, convergence to the limit is not always very fast. (See exercises 9.45 and 9.61.)

Page 97, line 7 from the bottom [first printing was correct] (26 Sep 95)

$$D(\alpha, n) \geqslant D\big(\alpha', \lfloor \alpha n \rfloor\big) - \alpha^{-1} - 2\,.$$

Page 109, line 23 (13 Jan 95)

out, its 65,050 decimal digits require 78 cents U.S. postage to mail first class.

Page 109, line 7 from the bottom (31 Dec 97)

seventeenth century [269]. The Mersenne primes known prior to 1998 occur

Page 109, line 4 from the bottom (31 Dec 97)

110503, 132049, 216091, 756839, 859433, 1257787, 1398269, and 2976221.

Page 110, lines 8 and following (4 Jan 94)

been a Mersenne prime, although only a few dozen Mersenne primes are known. Many people are trying ...

Page 116, line 19 (31 Dec 95)

independently by Moritz Stern [339], a German mathematician, and Achille

Page 122, new graffito for middle of page (15 Apr 96)

Hermann Minkowski illustrated this remarkable binary representation at the International Congress of Mathematicians in Heidelberg, 1904.

Page 123, lines 2–4 (26 Sep 95)

example, $\frac{878}{323} \approx 2.718266 \approx .999994e$; we obtained this fraction from the first 16 letters of e's Stern–Brocot representation, and the accuracy is about what we would get with 16 bits of e's binary representation.

Page 126, line 21 (28 Mar 97)

discovered by Sun Tzŭ in China, about A.D. 350.

Page 130, line 4 from the bottom in second printing (1 Mar 95)

ception of one that became the most famous of all, because it baffled the

Page 131, lines 1–4 (1 Mar 95)

for all positive integers a, b, c, and n, when $n > 2$. (Of course there are lots of solutions to the equations $a + b = c$ and $a^2 + b^2 = c^2$.) Andrew Wiles has apparently settled the question at long last; his intricate, epoch-making proof of (4.46) appears in *Annals of Mathematics* **142** (1995), 443–551.

Page 146, lines 2–4 from the bottom (24 May 96)

> $m_1, \ldots, m_r$ be positive integers with $m_j \perp m_k$ for $1 \leqslant j < k \leqslant r$; let $m = m_1 \ldots m_r$; and let $a_1, \ldots, a_r, A$ be integers. Then there is exactly one integer a such that

Page 149, lines 1 and 2 (25 Apr 95)

49 Let $R(N)$ be the number of pairs of integers (m, n) such that $1 \leqslant m \leqslant N$, $1 \leqslant n \leqslant N$, and $m \perp n$.

Page 149, line 7 from the bottom (24 May 96)

53 Find all positive integers n such that $n \setminus \lceil (n-1)!/(n+1) \rceil$.

Page 171, line 7 from the bottom (9 April 96)

This follows from a five-parameter identity discovered by John Dougall [82] early in the twentieth century.

Page 172, line 9 (13 May 95)

several people shortly thereafter. Exercise 86 gives a "simple" proof of (5.31).

Page 177, line 6 from the bottom (24 May 96)

seen so far. And we're faced with a sum of $2^{1000000} + 1$ terms, so we can't just

Page 200, line 4 from the bottom (1 Dec 94)

We will prove in Section 7.5 that these functions satisfy the identities

Page 201, addendum to the graffito (12 May 95)

Exercise 84 explains how to derive (5.61) from (5.60).

Page 202, line 8 from the bottom (20 Jun 96)

$$\mathcal{E}(z) \;=\; \sum_{k\geqslant 0}(k+1)^{k-1}\frac{z^k}{k!} \;=\; 1+z+\frac{3}{2}z^2+\frac{8}{3}z^3+\frac{125}{24}z^4+\cdots, \qquad (5.66)$$

Page 206, new graffito about 10 lines from the bottom (6 Jun 96)

We didn't discuss convergence of (5.56), (5.57), (5.58), ... either.

Page 211, addendum to the graffito in middle of page (7 Nov 94)

The value is infinite when z is a negative integer and w is not an integer.

Page 215, line 4 (13 May 95)

$$F\left(\begin{matrix} n+1,\ m-n,\ 1,\ \frac{1}{2} \\ \frac{1}{2}m+1,\ \frac{1}{2}m+\frac{1}{2},\ 2 \end{matrix}\middle|\, 1\right) \;=\; \frac{m}{n}, \qquad \text{integer } n \geqslant m > 0,$$

Page 215, line 6 (13 May 95)

new. But it really isn't; the left-hand side can be replaced by a multiple of

Page 215, line 11 from the bottom (13 May 95)

$$F\left(\begin{matrix} \frac{1}{3}r,\ \frac{1}{3}r+\frac{1}{3},\ \frac{1}{3}r+\frac{2}{3},\ -n,\ -n-\frac{1}{2}s,\ -n-\frac{1}{2}s+\frac{1}{2} \\ \frac{1}{2}r+\frac{1}{2},\ \frac{1}{2}r+1,\ -n-\frac{1}{3}s,\ -n-\frac{1}{3}s+\frac{1}{3},\ -n-\frac{1}{3}s+\frac{2}{3} \end{matrix}\middle|\, 1\right)$$

Page 215, line 8 from the bottom (13 May 95)

quantities (r, s, n) are replaced respectively by $(1, m-2n-1, n-m)$.

Page 222, line 10 (12 Aug 96)

whenever both infinite sums converge. And in fact both sums always do converge, except in the degenerate case when $a+b+\frac{1}{2}$ is a nonpositive integer.

Page 232, line 2 in first printing (11 Apr 95)

$\deg(R) = 1$ (assuming that $z \neq \pm 1$), we have $d = \deg(\hat{p}) - \deg(Q) = 0$ and

Page 233, line 11 (1 Jun 94)

algorithm can be formulated as follows, when $t(n, k)$ is given:

Page 237, line 2 from the bottom (26 Sep 95)

And $\binom{1}{n+1}z^{n+1} = \binom{0}{n}z^{n+1} = T(n, n+1)$ for all $n \geqslant 0$, so we obtain

Page 241, line 4 from the bottom (13 Jan 95)

Exercises 98–108 provide additional examples of the Gosper-Zeilberger

Page 244, lines 2 and 3 (24 May 96)

definition, by showing that the limit in (5.83) is $1/m!$ when $z = m$ is a positive integer.

Page 245, line 14 (24 May 96)

when c is a nonnegative integer. (See (5.115).) Use this idea to evaluate

Page 253, line 3 from the bottom (26 Sep 95)

94 Find $\sum \binom{a}{k}\binom{-a}{n-k}\,\delta k$ when n is a positive integer.

Page 254, line 2 of exercise 99 (26 Sep 95)

when $t(n,k) = (n+a+b+c+k)!/(n+k)!\,(c+k)!\,(b-k)!\,(a-k)!\,k!$,

Page 272, equations (6.50), (6.51), (6.53) (4 Jun 95)

change $\displaystyle\sum_{n\geqslant 0}$ to $\displaystyle\sum_{n}$

Page 289, lines 6 and 7 (4 Jun 95)

$$S_6(n) \;=\; n(n-\frac{1}{2})(n-1)(n-\frac{1}{2}+\alpha)(n-\frac{1}{2}-\alpha)(n-\frac{1}{2}+\overline{\alpha})(n-\frac{1}{2}-\overline{\alpha})/7\,,$$
$$\text{where } \alpha = 2^{-3/2}3^{-1/4}\bigl(\sqrt{\sqrt{31}+\sqrt{27}}+i\sqrt{\sqrt{31}-\sqrt{27}}\bigr).$$

Page 289, line 10 from the bottom (4 Jun 95)

$$\sum_{j\geqslant 0}\left\{{m \atop j}\right\}\left[{j+1 \atop k}\right]\frac{(-1)^{j+1-k}}{j+1} \;=\; \frac{1}{m+1}\binom{m+1}{k}B_{m+1-k}\,,\quad k>0. \qquad (6.99)$$

Page 290, line 8 (4 Jun 95)

$$\sum_{k}\left\{{n \atop k}\right\}\left[{k \atop m}\right]\frac{(-1)^{k-m}}{k} \;=\; \frac{1}{n}\binom{n}{m}B_{n-m} + [m=n-1]\,,\quad m>0. \qquad (6.100)$$

Page 292, line 13 from the bottom (27 Oct 97)

the Italian astronomer Gian Domenico Cassini in 1680 [51], is the identity

Page 292, lines 10 and 11 from the bottom (27 May 94)

When $n=6$, for example, Cassini's identity correctly claims that $13\cdot 5-8^2$ equals 1. (Johannes Kepler knew this law already in 1608 [202].)

Page 301, lines 19–24 (9 Jul 95)

of miles per n kilometers for all $n \leqslant 100$, except in the cases $n = 4$, 12, 54, 62, 75, 83, 91, 96, and 99, when it is off by less than 2/3 mile. And the shift-up rule gives either the correctly rounded number of kilometers for n miles, or 1 km too many, for all $n \leqslant 113$. (The only really embarrassing case is $n = 4$, where the individual rounding errors for $n = 3+1$ both go the same direction instead of cancelling each other out.)

Page 316, line 4 from the bottom (9 Jul 95)

$$\prod_{k\geqslant 1}\Big(1+\frac{z}{k}\Big)e^{-z/k} \;=\; \frac{e^{-\gamma z}}{z!}\,,$$

Page 317, line 4 (4 Jun 95)

$$\frac{z}{2^n}\cot\frac{z}{2^n} - \frac{z}{2^n}\tan\frac{z}{2^n} + \sum_{k=1}^{2^{n-1}-1}\frac{z}{2^n}\Big(\cot\frac{z+k\pi}{2^n} + \cot\frac{z-k\pi}{2^n}\Big)\,,$$

Page 317, lines 6–9 from the bottom (21 Jun 94)

77 When m and n are integers, $n \geqslant 0$, the value of $\sigma_n(m)$ is given by (6.48) if $m < 0$, by (6.49) if $m > n$, and by (6.101) if $m = 0$. Show that in the remaining cases we have

$$\sigma_n(m) \;=\; \frac{(-1)^{m+n-1}}{m!\,(n-m)!}\sum_{k=0}^{m-1}\begin{bmatrix} m \\ m-k \end{bmatrix}\frac{B_{n-k}}{n-k}\,, \quad \text{integer } n \geqslant m > 0.$$

Page 319, line 14 from the bottom (1 Nov 94)

a Are there infinitely many n with $p \backslash a_n$, for some fixed prime p?

Page 353, line 6 (4 Jun 95)

$$\frac{e^w - e^z}{we^z - ze^w} \;=\; \sum_{m,n\geqslant 0}\Big\langle {m+n+1 \atop m} \Big\rangle \frac{w^m z^n}{(m+n+1)!}\,. \tag{7.60}$$

Page 361, line 20 and following (13 May 95)

sequences $\langle a_0, \ldots, a_{mn} \rangle$ arise uniquely in this way, if $n > 0$: The last term a_{mn} must be $(1 - m)$. The partial sums ...

Page 379, line 2 from the bottom (17 May 94)

have the form $p_n(y) = \sum_{m=0}^{n} \left|{n \atop m}\right| y^m$, where $\left|{n \atop m}\right|$ is a positive integer for

Page 391, line 5 (12 May 95)

between 6.975 million and 7.025 million.

Page 393, line 1 (12 May 95)

We estimate the average spot sum of these dice to be $7.4 \pm 2.1/\sqrt{10} \approx 7.4 \pm 0.7$,

Page 419, line 4 (4 Jun 95)

Once again we have gained the desired speedup factor of $1/m$. If $m \approx n/\ln n$

Page 419, lines 8 and 9 (4 Jun 95)

this distribution is called "Zipf's law." Then $\mathrm{Mean}(S) = n/H_n - 1$, $\mathrm{Var}(S) = \frac{1}{2}n(n+1)/H_n - n^2/H_n^2$. The average number of probes for $m \approx n/\ln n$ as

Page 425, line 18 (24 May 96)

must set a `NEXT` entry to $n + 1$. These two cases may take different amounts

Page 426, bottom three lines (4 Jun 95)

$m = n/\ln n + O(1)$ and $n \to \infty$ the corresponding results are

$$\begin{aligned} \mathrm{Mean}(G_T) &= \beta \ln n + \alpha + O\big((\log n)^2/n\big)\,; \\ \mathrm{Var}(G_T) &= \beta^2 \ln n + O\big((\log n)^2/n\big)\,. \end{aligned}$$

Page 433, second graffito should be in French (8 May 96)

"Une rapide opération arithmétique montre que, grâce à cette ingénieuse cascade, les xérès ont toujours au moins trois ans. Pousser plus loin le calcul de leur âge donne le vertige."
—Revue du vin de France (Nov 1984)

Page 457, line 4 from the bottom (9 Jul 95)

For example, when $n = 10^6$ this estimate comes to $15631363.6 + O(n/\log n)$;

Page 466, line 5 (26 Sep 95)

$$O\Bigl(\sum_{0<k<n} \frac{(\log n)^2}{k(n-k)}\Bigr) \;=\; O\Bigl(\frac{(\log n)^3}{n}\Bigr),$$

Page 474, line 3 (9 Jul 95)

negative multiple of $\cos(2\pi x - \frac{1}{2}\pi m)$, with error $O\bigl(2^{-m} \max_x B_m(\{x\})\bigr)$.

Page 475, line 11 (9 Jul 95)

$$R_{2m} \;=\; \theta_m \frac{B_{2m+2}}{(2m+2)!} f^{(2m+1)}(x)\Big|_a^b, \qquad \text{for some } 0 \leqslant \theta_m \leqslant 1; \tag{9.80}$$

Page 477, line 14 from the bottom (9 Jul 95)

So $R_4(n)$ is actually equal to $\frac{89}{128} - \ln 2 + O(n^{-5})$, but Euler's summation

Page 480, line 4 from the bottom (9 Jul 95)

$$H_n \;=\; \ln n + \gamma + \frac{1}{2n} - \sum_{k=1}^{m} \frac{B_{2k}}{2kn^{2k}} - \theta_{m,n} \frac{B_{2m+2}}{(2m+2)n^{2m+2}}, \tag{9.88}$$

Page 481, line 5 (27 May 94)

the value of γ correct to 1271 decimal places, beginning thus:

Page 500, lines 15–18 (31 Jan 95)

1.19 Not when $n > 5$. A bent line whose half-lines run at angles θ and $\theta + 30^\circ$ from its apex can intersect four times with another whose half-lines run at angles ϕ and $\phi + 30^\circ$ only if $30^\circ < |\theta - \phi| < 150^\circ$. We can't choose more than 5 angles this far apart from each other. (It is possible to choose 5.)

Page 501, lines 5 and following (20 Feb 95)

1.24 The only known examples are: $X_n = 2i \sin \pi r + 1/X_{n-1}$, where r is rational and $0 \leqslant r < \frac{1}{2}$ (all period lengths $\geqslant 2$ occur as r varies); Gauss's ...

Page 509, lines 5 and 6 (26 Sep 95)

3.29 $D\big(\alpha', \lfloor \alpha n \rfloor\big)$ is at most the maximum of the absolute value of

$$s\big(\alpha', \lfloor n\alpha \rfloor, \nu'\big) \;=\; -s(\alpha, n, \nu) - S + \epsilon + \{0 \text{ or } 1\} + \nu' - \{0 \text{ or } 1\}\,.$$

Page 512, line 5 from the bottom (8 May 96)

$(-1)^{[x(n)<y(n)]}$.

Page 520, lines 4–6 from the bottom (25 Apr 95)

4.49 (a) Either $m < n$ ($\Phi(N) - 1$ cases) or $m = n$ (one case) or $m > n$ ($\Phi(N) - 1$ again). Hence $R(N) = 2\Phi(N) - 1$. (b) From (4.62) we get

$$2\Phi(N) - 1 \;=\; -1 + \sum_{d \geqslant 1} \mu(d) \lfloor N/d \rfloor \lfloor 1 + N/d \rfloor\,;$$

Page 521, bottom line (24 May 96)

$$\big\lceil (n-1)!/(n+1) \big\rceil \;=\; \big((n-1)! + n\big)/(n+1)\,;$$

Page 525, line 4 of answer 4.69 (20 Jun 96)

in exercise 60 are the best that were known in 1994 [255]. Exercise 68

Page 530, line 15 (6 Jun 96)

for all $m > 0$ and $n > 0$, by induction on $m + n$.

Page 531, line 5 from the bottom (4 Jun 95)

$$\binom{n+a-1}{n} F\left(\begin{matrix} a, b, -n \\ c, a \end{matrix} \middle| 1\right) = \frac{a^{\overline{n}}}{n!} \frac{(c-b)^{\overline{n}}}{c^{\overline{n}}}$$

Page 532, new graffito for top of page (13 Jan 95)

Term limits?

Page 540, line 7 from the bottom (26 Sep 95)

quently, when $m \geqslant 0$ is an integer less than n, we have

Page 547, correction to first line of graffito (21 Jan 95)

Notice that $1/nk$

Page 551, lines 5–8 from the bottom (9 Jul 95)

valid for $k = n - 1$ if we write it in the form

$$\begin{aligned} &K_{m+n}(x_1, \ldots, x_{m+n})\, K_{n-1}(x_{m+n-1}, \ldots, x_{m+1}) \\ &\quad = K_{m+n-1}(x_1, \ldots, x_{m+n-1})\, K_n(x_{m+n}, \ldots, x_{m+1}) \\ &\qquad - (-1)^n K_{m-1}(x_1, \ldots, x_{m-1})\,. \end{aligned}$$

[Also delete '$k = 2$,' where it appears on the bottom line.]

Page 555, line 2 (1 Nov 94)

16735, and 102728. See the answer to exercise 92.)

Page 564, lines 13 and 14 (14 Jun 94)

$\left\{ {n \atop m} \right\} = \sum_k \binom{m}{k} k^n (-1)^{m-k}/m!$ for integer $m \geqslant 0$ and arbitrary $n \geqslant 0$; then (6.3) holds for all $n \geqslant 1$.

Page 564, lines 15 and 16 (1 Nov 94)

6.92 (a) David Boyd has shown that there are only finitely many solutions for all $p < 500$, except perhaps $p = 83, 127, 397$. (b) The behavior of b_n is quite ...

Page 568, new graffito for the answer to 7.23 (15 Aug 94)

"Curiously, a_{2n} is equal to U_{2n}^2, the square of the number of ways to tile a $3 \times 2n$ rectangle with dominoes; and $a_{2n+1} = 2V_{2n+1}^2$."
—I. Kaplansky

Page 578, lines 3 and 4 of answer 22 (13 May 96)

$\big(E\big(E(X|Y)\big)\big)^2$. But $E\big(E(X|Y)\big) = \sum_y \Pr(Y=y)E(X|y) = \sum_{x,y} \Pr(Y=y) \times \Pr\big((X|y)=x\big)x = EX$ and $E\big(E(X^2|Y)\big) = E(X^2)$, so the result is just VX.

Page 581, correction to the graffito (18 Aug 96)

"Toto, I've a feeling we're not in Kansas anymore."
—Dorothy

Page 594, line 2 (9 Jul 95)

$-5.5/(\ln n)^2 + O(\log\log n/\log n)^3$; then we estimate $P_{1000000} \approx 15480992.8$.)

Page 597, line 4 of answer 9.40 (20 Jun 96)

$$\sum_{k=1}^{n/2} \frac{H_{2k}^{m-1}}{k} = \sum_{k=1}^{n/2} \frac{(\ln 2e^{\gamma}k)^{m-1} + O(k^{-1}(\log k)^{m-2})}{k}$$

Page 599, lines 1 and 2 (26 Sep 95)

for all m. Divide by n and let $n \to \infty$; the limit points are bounded above by $\alpha_1 \ldots \alpha_m$ for all m. Finally we have

Page 603, new graffito for answer 9.63 (31 Dec 97)

Additional progress on this problem has been made by Jean-Luc Rémy, Journal of Number Theory, vol. 66 (1997), 1–28.

Page 605, lines 16–20 (29 Mar 97)

15 M. D. Atkinson, "The cyclic towers of Hanoi," *Information Processing Letters* **13** (1981), 118–119. *633.*

16 M. D. Atkinson, "How to compute the series expansions of $\sec x$ and $\tan x$," *American Mathematical Monthly* **93** (1986), 387–389. [See also A. J. Kempner, "On the shape of polynomial curves," *Tôhoku Math. J.* **37** (1933), 347–362; R. C. Entringer, "A combinatorial interpretation of the Euler and Bernoulli numbers," *Nieuw Archief voor Wiskunde* **14** (1966), 241–246.] *635.*

Page 608, lines 22 and 23 (1 Feb 96)

59 F. R. K. Chung and R. L. Graham, "On the cover polynomial of a digraph," *Journal of Combinatorial Theory*, series B, **65** (1995), 273–290.

Page 612, line 10 (31 Dec 95)

lated into French, 1786; German, 1788; Russian, 1936; English, 1988.

Page 612, line 16 and following (31 Dec 95)

Analysi Finitorum ac Doctrina Serierum. St. Petersburg, Academiæ Imperialis ...

Page 614, line 8 (1 Nov 96)

132 J. Franel, Solutions to questions 42 and 170, in L'*Intermédiaire des*

Page 614, line 11 (31 Dec 95)

minimal storage tree sorting," *Journal of the Association for Computing Machinery* **27** (1970), 496–507.

Page 615, line 12 (24 May 96)

145 Angelo Genocchi, "Intorno all'espressione generale de'numeri Bernulliani,"

Page 616, lines 5 and 6 from the bottom (12 Feb 96)

170 Inger Johanne Håland and Donald E. Knuth, "Polynomials involving the floor function," *Mathematica Scandinavica* **76** (1995), 194–200.

Page 619, after line 2 (27 May 94)

202 Johannes Kepler, letter to Joachim Tancke (12 May 1608), in his *Gesammelte Werke*, volume 16, 154–165.

[Items previously numbered 202, 203, and 204 are now numbered 203, 204, 205; the item previously numbered 205 has been dropped.]

Page 620, line 5 from the bottom (31 Dec 95)

229 E. E. Kummer, "Über die hypergeometrische Reihe

Page 622, lines 16 and 17 (11 Apr 96)

Stirling numbers of the second kind," *Discrete Mathematics* **150** (1996), 281–292.

Page 622, lines 18 and 19 (28 Mar 97)

249 Li Shan-Lan, *Duò Jī Bǐ Lèi* [Sums of Piles Obtained Inductively]. In his *Zégǔxī Zhaī Suànxué* [Classically Inspired Meditations on Mathematics],

Page 623, line 6 (10 Oct 97)

premier," *Bulletin de la Société mathématique de France* **6** (1877), 49–54.

Page 624, line 5 (31 Dec 95)

Zahlen," *Sitzungsberichte der Mathematisch-Naturwissenschaftlichen*

Page 626, line 18 (3 Aug 95)

308 Paulo Ribenboim, *13 Lectures on Fermat's Last Theorem*. Springer-

Page 626, line 5 from the bottom (6 Mar 96)

313 Gian-Carlo Rota, "On the foundations of combinatorial theory. I. The-

Page 627, line 20 (31 Dec 95)

ence on Combinatorial Structures and their Applications, 1969.)

Page 627, lines 10 and 11 from the bottom (1 Nov 96)

letin International de l'Académie Polonaise des Sciences et des Lettres (Cracovie), series A (1910), 9–11.

Page 627, line 6 from the bottom (31 Dec 95)

328 Wacław Sierpiński, *A Selection of Problems in the Theory of Numbers*.

Page 628, lines 1 and 2 (1 Nov 94)

330 N. J. A. Sloane, *A Handbook of Integer Sequences*. Academic Press, 1973. Sequel, with Simon Plouffe, *The Encyclopedia of Integer Sequences*, Academic Press, 1995.

Page 628, lines 4 and 5 (4 Mar 96)

331 A. D. Solov'ev, "Odno kombinatornoe tozhdestvo i ego primenenie k zadache o pervom nastuplenii redkogo sobytiia͡," *Teoriia͡ veroi͡atnosteĭ*

Page 629, lines 1 and 2 (21 Jan 95)

344 Volker Strehl, "Binomial identities — combinatorial and algorithmic aspects," *Discrete Mathematics* **136** (1994), 309–346.

Page 630, bottom line (21 Jan 95)

373 Herbert S. Wilf, *generatingfunctionology*. Academic Press, 1990; second edition, 1994.

Page 633, left column (26 Sep 96)

1.10 Atkinson [15].

Page 635, left column (26 Sep 96)

6.75 Atkinson [16].

Page 635, left column (2 Oct 97)

6.76 [209, answer 5.1.3–3]; Lengyel [248].

Page 635, right column (13 Jan 95)

6.86 [226].

Page 636, missing entry (8 May 96)

8.45 1985 final exam.

Page 638, left column (26 Sep 96)

[remove the reference to V. I. Arnol'd]

Page 638, right column (26 Sep 96)

Atkinson, Michael David, 605, 633, 635.

Page 639, right column (7 Jul 97)

Blom, Carl Gunnar, 606, 636

Page 639, right column (23 Jan 95)

Boyd, David William, 564

Page 640, left column (27 Oct 97)

Cassini, Gian (= Giovanni = Jean) Domenico (= Dominique), 292, 607

Page 641, left column (18 Dec 96)

Concrete Math Club, 74, 453

Page 642, left column (1 Oct 94)

Dirichlet, Johann Peter Gustav Lejeune, 370, 610, 633

Page 643, left column (28 Dec 96)

Entringer, Roger Charles, 605

Page 645, left column (1 Oct 94)

Gauß (= Gauss), Johann Friedrich Carl (= Carl Friedrich), vii, 6, 7, ...

Page 647, right column (29 Mar 97)

Kempner, Aubrey John, 605

Page 648, left column (28 Mar 97)

Lǐ Shànlán Rénshū (= Qiūrèn), 269, 622

Page 648, right column (15 Apr 96)

Minkowski, Hermann, 122

Page 650, right column (29 Aug 96)

Phi function: sum of φ, 137–139, 462–463

Page 651, right column (1 Sep 94)

Ramanujan Iyengar, Srinivasa, 330

Page 652, left column (31 Dec 97)

Rémy, Jean-Luc, 603

Page 652, left column (3 Aug 95)

Ribenboim, Paulo, 555, 626, 634

Page 653, right column (31 Dec 95)

Stern, Moritz Abraham, 116, 628

Page 654, right column (28 Mar 97)

Sun Tzu (= Sūnzǐ, Master Sun), 126

Page 655, right column (15 Dec 96)

umop-apısdn function, 193.

Page 656, right column in second printing (1 Mar 95)

Wiles, Andrew John, 131

Finally, on the back cover, line 7 (5 Feb 96)

change ‘indispensible’ to ‘indispensable’.

Michael Spivak

CALCULUS ON MANIFOLDS

Michael Spivak
Brandeis University

Calculus on Manifolds

A MODERN APPROACH TO CLASSICAL THEOREMS OF ADVANCED CALCULUS

ADDISON-WESLEY PUBLISHING COMPANY
The Advanced Book Program
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn
Sydney • Singapore • Tokyo • Madrid • San Juan • Paris
Seoul • Milan • Mexico City • Taipei

Calculus on Manifolds

A Modern Approach to Classical Theorems of Advanced Calculus

Copyright © 1965 by Addison-Wesley Publishing Company
All rights reserved

Library of Congress Card Catalog Number 66-10910
Manufactured in the United States of America
The manuscript was put into production on April 21, 1965;
this volume was published on October 26, 1965

ISBN 0-8053-9021-9

24 25 26 27 28-CRW-9998979695
Twenty-fourth printing, January 1995

Editors' Foreword

Mathematics has been expanding in all directions at a fabulous rate during the past half century. New fields have emerged, the diffusion into other disciplines has proceeded apace, and our knowledge of the classical areas has grown ever more profound. At the same time, one of the most striking trends in modern mathematics is the constantly increasing interrelationship between its various branches. Thus the present-day students of mathematics are faced with an immense mountain of material. In addition to the traditional areas of mathematics as presented in the traditional manner—and these presentations do abound—there are the new and often enlightening ways of looking at these traditional areas, and also the vast new areas teeming with potentialities. Much of this new material is scattered indigestibly throughout the research journals, and frequently coherently organized only in the minds or unpublished notes of the working mathematicians. And students desperately need to learn more and more of this material.

This series of brief topical booklets has been conceived as a possible means to tackle and hopefully to alleviate some of

these pedagogical problems. They are being written by active research mathematicians, who can look at the latest developments, who can use these developments to clarify and condense the required material, who know what ideas to underscore and what techniques to stress. We hope that they will also serve to present to the able undergraduate an introduction to contemporary research and problems in mathematics, and that they will be sufficiently informal that the personal tastes and attitudes of the leaders in modern mathematics will shine through clearly to the readers.

The area of differential geometry is one in which recent developments have effected great changes. That part of differential geometry centered about Stokes' Theorem, sometimes called the fundamental theorem of multivariate calculus, is traditionally taught in advanced calculus courses (second or third year) and is essential in engineering and physics as well as in several current and important branches of mathematics. However, the teaching of this material has been relatively little affected by these modern developments; so the mathematicians must relearn the material in graduate school, and other scientists are frequently altogether deprived of it. Dr. Spivak's book should be a help to those who wish to see Stoke's Theorem as the modern working mathematician sees it. A student with a good course in calculus and linear algebra behind him should find this book quite accessible.

Robert Gunning
Hugo Rossi

Princeton, New Jersey
Waltham, Massachusetts
August 1965

Preface

This little book is especially concerned with those portions of "advanced calculus" in which the subtlety of the concepts and methods makes rigor difficult to attain at an elementary level. The approach taken here uses elementary versions of modern methods found in sophisticated mathematics. The formal prerequisites include only a term of linear algebra, a nodding acquaintance with the notation of set theory, and a respectable first-year calculus course (one which at least mentions the least upper bound (sup) and greatest lower bound (inf) of a set of real numbers). Beyond this a certain (perhaps latent) rapport with abstract mathematics will be found almost essential.

The first half of the book covers that simple part of advanced calculus which generalizes elementary calculus to higher dimensions. Chapter 1 contains preliminaries, and Chapters 2 and 3 treat differentiation and integration.

The remainder of the book is devoted to the study of curves, surfaces, and higher-dimensional analogues. Here the modern and classical treatments pursue quite different routes; there are, of course, many points of contact, and a significant encounter

occurs in the last section. The very classical equation reproduced on the cover appears also as the last theorem of the book. This theorem (Stokes' Theorem) has had a curious history and has undergone a striking metamorphosis.

The first statement of the Theorem appears as a postscript to a letter, dated July 2, 1850, from Sir William Thomson (Lord Kelvin) to Stokes. It appeared publicly as question 8 on the Smith's Prize Examination for 1854. This competitive examination, which was taken annually by the best mathematics students at Cambridge University, was set from 1849 to 1882 by Professor Stokes; by the time of his death the result was known universally as Stokes' Theorem. At least three proofs were given by his contemporaries: Thomson published one, another appeared in Thomson and Tait's *Treatise on Natural Philosophy*, and Maxwell provided another in *Electricity and Magnetism* [13]. Since this time the name of Stokes has been applied to much more general results, which have figured so prominently in the development of certain parts of mathematics that Stokes' Theorem may be considered a case study in the value of generalization.

In this book there are three forms of Stokes' Theorem. The version known to Stokes appears in the last section, along with its inseparable companions, Green's Theorem and the Divergence Theorem. These three theorems, the classical theorems of the subtitle, are derived quite easily from a modern Stokes' Theorem which appears earlier in Chapter 5. What the classical theorems state for curves and surfaces, this theorem states for the higher-dimensional analogues (manifolds) which are studied thoroughly in the first part of Chapter 5. This study of manifolds, which could be justified solely on the basis of their importance in modern mathematics, actually involves no more effort than a careful study of curves and surfaces alone would require.

The reader probably suspects that the modern Stokes' Theorem is at least as difficult as the classical theorems derived from it. On the contrary, it is a very simple consequence of yet another version of Stokes' Theorem; this very abstract version is the final and main result of Chapter 4.

It is entirely reasonable to suppose that the difficulties so far avoided must be hidden here. Yet the proof of this theorem is, in the mathematician's sense, an utter triviality—a straightforward computation. On the other hand, even the statement of this triviality cannot be understood without a horde of difficult definitions from Chapter 4. There are good reasons why the theorems should all be easy and the definitions hard. As the evolution of Stokes' Theorem revealed, a single simple principle can masquerade as several difficult results; the proofs of many theorems involve merely stripping away the disguise. The definitions, on the other hand, serve a twofold purpose: they are rigorous replacements for vague notions, and machinery for elegant proofs. The first two sections of Chapter 4 define precisely, and prove the rules for manipulating, what are classically described as "expressions of the form" $P\,dx + Q\,dy + R\,dz$, or $P\,dx\,dy + Q\,dy\,dz + R\,dz\,dx$. Chains, defined in the third section, and partitions of unity (already introduced in Chapter 3) free our proofs from the necessity of chopping manifolds up into small pieces; they reduce questions about manifolds, where everything seems hard, to questions about Euclidean space, where everything is easy.

Concentrating the depth of a subject in the definitions is undeniably economical, but it is bound to produce some difficulties for the student. I hope the reader will be encouraged to learn Chapter 4 thoroughly by the assurance that the results will justify the effort: the classical theorems of the last section represent only a few, and by no means the most important, applications of Chapter 4; many others appear as problems, and further developments will be found by exploring the bibliography.

The problems and the bibliography both deserve a few words. Problems appear after every section and are numbered (like the theorems) within chapters. I have starred those problems whose results are used in the text, but this precaution should be unnecessary—the problems are the most important part of the book, and the reader should at least attempt them all. It was necessary to make the bibliography either very incomplete or unwieldy, since half the major

branches of mathematics could legitimately be recommended as reasonable continuations of the material in the book. I have tried to make it incomplete but tempting.

Many criticisms and suggestions were offered during the writing of this book. I am particularly grateful to Richard Palais, Hugo Rossi, Robert Seeley, and Charles Stenard for their many helpful comments.

I have used this printing as an opportunity to correct many misprints and minor errors pointed out to me by indulgent readers. In addition, the material following Theorem 3-11 has been completely revised and corrected. Other important changes, which could not be incorporated in the text without excessive alteration, are listed in the Addenda at the end of the book.

Michael Spivak

Waltham, Massachusetts
March 1968

Contents

Editors' Foreword, v
Preface, vii

1. Functions on Euclidean Space ***1***

NORM AND INNER PRODUCT, 1
SUBSETS OF EUCLIDEAN SPACE, 5
FUNCTIONS AND CONTINUITY, 11

2. Differentiation ***15***

BASIC DEFINITIONS, 15
BASIC THEOREMS, 19
PARTIAL DERIVATIVES, 25
DERIVATIVES, 30
INVERSE FUNCTIONS, 34
IMPLICIT FUNCTIONS, 40
NOTATION, 44

3. *Integration* **46**

BASIC DEFINITIONS, 46

MEASURE ZERO AND CONTENT ZERO, 50

INTEGRABLE FUNCTIONS, 52

FUBINI'S THEOREM, 56

PARTITIONS OF UNITY, 63

CHANGE OF VARIABLE, 67

4. *Integration on Chains* **75**

ALGEBRAIC PRELIMINARIES, 75

FIELDS AND FORMS, 86

GEOMETRIC PRELIMINARIES, 97

THE FUNDAMENTAL THEOREM OF CALCULUS, 100

5. *Integration on Manifolds* **109**

MANIFOLDS, 109

FIELDS AND FORMS ON MANIFOLDS, 115

STOKES' THEOREM ON MANIFOLDS, 122

THE VOLUME ELEMENT, 126

THE CLASSICAL THEOREMS, 134

Bibliography, 139

Index, 141

Calculus on Manifolds

1

Functions on Euclidean Space

NORM AND INNER PRODUCT

Euclidean n-space $\mathbf{R}^n$ is defined as the set of all n-tuples $(x^1, \ldots, x^n)$ of real numbers x^i (a "1-tuple of numbers" is just a number and $\mathbf{R}^1 = \mathbf{R}$, the set of all real numbers). An element of $\mathbf{R}^n$ is often called a point in $\mathbf{R}^n$, and $\mathbf{R}^1$, $\mathbf{R}^2$, $\mathbf{R}^3$ are often called the line, the plane, and space, respectively. If x denotes an element of $\mathbf{R}^n$, then x is an n-tuple of numbers, the ith one of which is denoted x^i; thus we can write

$$x = (x^1, \ldots, x^n).$$

A point in $\mathbf{R}^n$ is frequently also called a vector in $\mathbf{R}^n$, because $\mathbf{R}^n$, with $x + y = (x^1 + y^1, \ldots, x^n + y^n)$ and $ax = (ax^1, \ldots, ax^n)$, as operations, *is* a vector space (over the real numbers, of dimension n). In this vector space there is the notion of the length of a vector x, usually called the **norm** $|x|$ of x and defined by $|x| = \sqrt{(x^1)^2 + \cdots + (x^n)^2}$. If $n = 1$, then $|x|$ is the usual absolute value of x. The relation between the norm and the vector space structure of $\mathbf{R}^n$ is very important.

1-1 Theorem. *If $x,y \in \mathbf{R}^n$ and $a \in \mathbf{R}$, then*

(1) $|x| \geq 0$, *and* $|x| = 0$ *if and only if* $x = 0$.

(2) $|\Sigma_{i=1}^n x^i y^i| \leq |x| \cdot |y|$; *equality holds if and only if x and y are linearly dependent.*

(3) $|x + y| \leq |x| + |y|$.

(4) $|ax| = |a| \cdot |x|$.

Proof

(1) is left to the reader.

(2) If x and y are linearly dependent, equality clearly holds. If not, then $\lambda y - x \neq 0$ for all $\lambda \in \mathbf{R}$, so

$$0 < |\lambda y - x|^2 = \sum_{i=1}^n (\lambda y^i - x^i)^2$$

$$= \lambda^2 \sum_{i=1}^n (y^i)^2 - 2\lambda \sum_{i=1}^n x^i y^i + \sum_{i=1}^n (x^i)^2.$$

Therefore the right side is a quadratic equation in λ with no real solution, and its discriminant must be negative. Thus

$$4\left(\sum_{i=1}^n x^i y^i\right)^2 - 4\sum_{i=1}^n (x^i)^2 \cdot \sum_{i=1}^n (y^i)^2 < 0.$$

(3)
$$\begin{aligned} |x + y|^2 &= \Sigma_{i=1}^n (x^i + y^i)^2 \\ &= \Sigma_{i=1}^n (x^i)^2 + \Sigma_{i=1}^n (y^i)^2 + 2\Sigma_{i=1}^n x^i y^i \\ &\leq |x|^2 + |y|^2 + 2|x| \cdot |y| \qquad \text{by (2)} \\ &= (|x| + |y|)^2. \end{aligned}$$

(4) $|ax| = \sqrt{\Sigma_{i=1}^n (ax^i)^2} = \sqrt{a^2 \Sigma_{i=1}^n (x^i)^2} = |a| \cdot |x|$. ∎

The quantity $\Sigma_{i=1}^n x^i y^i$ which appears in (2) is called the **inner product** of x and y and denoted $\langle x,y \rangle$. The most important properties of the inner product are the following.

1-2 Theorem. *If x, x_1, x_2 and y, y_1, y_2 are vectors in $\mathbf{R}^n$ and $a \in \mathbf{R}$, then*

(1) $\langle x,y \rangle = \langle y,x \rangle$ *(symmetry).*

(2) $\langle ax,y\rangle = \langle x,ay\rangle = a\langle x,y\rangle$ (*bilinearity*).
$\langle x_1 + x_2, y\rangle = \langle x_1,y\rangle + \langle x_2,y\rangle$
$\langle x, y_1 + y_2\rangle = \langle x,y_1\rangle + \langle x,y_2\rangle$

(3) $\langle x,x\rangle \geq 0$, *and* $\langle x,x\rangle = 0$ *if and only if* $x = 0$ (*positive definiteness*).

(4) $|x| = \sqrt{\langle x,x\rangle}$.

(5) $\langle x,y\rangle = \dfrac{|x+y|^2 - |x-y|^2}{4}$ (*polarization identity*).

Proof

(1) $\langle x,y\rangle = \Sigma_{i=1}^n x^i y^i = \Sigma_{i=1}^n y^i x^i = \langle y,x\rangle$.

(2) By (1) it suffices to prove

$$\langle ax,y\rangle = a\langle x,y\rangle,$$
$$\langle x_1 + x_2, y\rangle = \langle x_1,y\rangle + \langle x_2,y\rangle.$$

These follow from the equations

$$\langle ax,y\rangle = \sum_{i=1}^n (ax^i)y^i = a\sum_{i=1}^n x^i y^i = a\langle x,y\rangle,$$
$$\langle x_1 + x_2, y\rangle = \sum_{i=1}^n (x_1{}^i + x_2{}^i)y^i = \sum_{i=1}^n x_1{}^i y^i + \sum_{i=1}^n x_2{}^i y^i$$
$$= \langle x_1,y\rangle + \langle x_2,y\rangle.$$

(3) and (4) are left to the reader.

(5) $\dfrac{|x+y|^2 - |x-y|^2}{4}$

$= \frac{1}{4}[\langle x+y, x+y\rangle - \langle x-y, x-y\rangle]$ by (4)

$= \frac{1}{4}[\langle x,x\rangle + 2\langle x,y\rangle + \langle y,y\rangle - (\langle x,x\rangle - 2\langle x,y\rangle + \langle y,y\rangle)]$

$= \langle x,y\rangle$. ∎

We conclude this section with some important remarks about notation. The vector $(0, \ldots ,0)$ will usually be denoted simply 0. The **usual basis** of $\mathbf{R}^n$ is $e_1, \ldots ,e_n$, where $e_i = (0, \ldots ,1, \ldots ,0)$, with the 1 in the ith place. If $T: \mathbf{R}^n \to \mathbf{R}^m$ is a linear transformation, the matrix of T with respect to the usual bases of $\mathbf{R}^n$ and $\mathbf{R}^m$ is the $m \times n$ matrix $A = (a_{ij})$, where $T(e_i) = \Sigma_{j=1}^m a_{ji}e_j$ —the coefficients of $T(e_i)$

appear in the ith *column* of the matrix. If $S\colon \mathbf{R}^m \to \mathbf{R}^p$ has the $p \times m$ matrix B, then $S \circ T$ has the $p \times n$ matrix BA [here $S \circ T(x) = S(T(x))$; most books on linear algebra denote $S \circ T$ simply ST]. To find $T(x)$ one computes the $m \times 1$ matrix

$$\begin{pmatrix} y^1 \\ \cdot \\ \cdot \\ \cdot \\ y^m \end{pmatrix} = \begin{pmatrix} a_{11}, & \cdots & ,a_{1n} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{m1}, & \cdots & ,a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x^1 \\ \cdot \\ \cdot \\ \cdot \\ x^n \end{pmatrix};$$

then $T(x) = (y^1, \ldots ,y^m)$. One notational convention greatly simplifies many formulas: if $x \in \mathbf{R}^n$ and $y \in \mathbf{R}^m$, then (x,y) denotes

$$(x^1, \ldots ,x^n,y^1, \ldots ,y^m) \in \mathbf{R}^{n+m}.$$

Problems. **1-1.*** Prove that $|x| \leq \sum_{i=1}^n |x^i|$.

1-2. When does equality hold in Theorem 1-1(3)? *Hint:* Re-examine the proof; the answer is not "when x and y are linearly dependent."

1-3. Prove that $|x - y| \leq |x| + |y|$. When does equality hold?

1-4. Prove that $\big|\, |x| - |y| \,\big| \leq |x - y|$.

1-5. The quantity $|y - x|$ is called the **distance** between x and y. Prove and interpret geometrically the "triangle inequality": $|z - x| \leq |z - y| + |y - x|$.

1-6. Let f and g be integrable on $[a,b]$.

(a) Prove that $\left|\int_a^b f \cdot g\right| \leq \left(\int_a^b f^2\right)^{\frac{1}{2}} \cdot \left(\int_a^b g^2\right)^{\frac{1}{2}}$. *Hint:* Consider separately the cases $0 = \int_a^b (f - \lambda g)^2$ for some $\lambda \in \mathbf{R}$ and $0 < \int_a^b (f - \lambda g)^2$ for all $\lambda \in \mathbf{R}$.

(b) If equality holds, must $f = \lambda g$ for some $\lambda \in \mathbf{R}$? What if f and g are continuous?

(c) Show that Theorem 1-1(2) is a special case of (a).

1-7. A linear transformation $T\colon \mathbf{R}^n \to \mathbf{R}^n$ is **norm preserving** if $|T(x)| = |x|$, and **inner product preserving** if $\langle Tx,Ty\rangle = \langle x,y\rangle$.

(a) Prove that T is norm preserving if and only if T is inner-product preserving.

(b) Prove that such a linear transformation T is 1-1 and T^{-1} is of the same sort.

1-8. If $x,y \in \mathbf{R}^n$ are non-zero, the **angle** between x and y, denoted $\angle(x,y)$, is defined as $\arccos(\langle x,y\rangle/|x| \cdot |y|)$, which makes sense by Theorem 1-1(2). The linear transformation T is **angle preserving** if T is 1-1, and for $x,y \neq 0$ we have $\angle(Tx,Ty) = \angle(x,y)$.

(a) Prove that if T is norm preserving, then T is angle preserving.

(b) If there is a basis $x_1, \ldots, x_n$ of $\mathbf{R}^n$ and numbers $\lambda_1, \ldots, \lambda_n$ such that $Tx_i = \lambda_i x_i$, prove that T is angle preserving if and only if all $|\lambda_i|$ are equal.

(c) What are all angle preserving $T: \mathbf{R}^n \to \mathbf{R}^n$?

1-9. If $0 \leq \theta < \pi$, let $T: \mathbf{R}^2 \to \mathbf{R}^2$ have the matrix $\begin{pmatrix} \cos\theta, & \sin\theta \\ -\sin\theta, & \cos\theta \end{pmatrix}$.

Show that T is angle preserving and if $x \neq 0$, then $\angle(x,Tx) = \theta$.

1-10.* If $T: \mathbf{R}^m \to \mathbf{R}^n$ is a linear transformation, show that there is a number M such that $|T(h)| \leq M|h|$ for $h \in \mathbf{R}^m$. *Hint:* Estimate $|T(h)|$ in terms of $|h|$ and the entries in the matrix of T.

1-11. If $x,y \in \mathbf{R}^n$ and $z,w \in \mathbf{R}^m$, show that $\langle (x,z),(y,w)\rangle = \langle x,y\rangle + \langle z,w\rangle$ and $|(x,z)| = \sqrt{|x|^2 + |z|^2}$. Note that (x,z) and (y,w) denote points in $\mathbf{R}^{n+m}$.

1-12.* Let $(\mathbf{R}^n)^*$ denote the dual space of the vector space $\mathbf{R}^n$. If $x \in \mathbf{R}^n$, define $\varphi_x \in (\mathbf{R}^n)^*$ by $\varphi_x(y) = \langle x,y\rangle$. Define $T: \mathbf{R}^n \to (\mathbf{R}^n)^*$ by $T(x) = \varphi_x$. Show that T is a 1-1 linear transformation and conclude that every $\varphi \in (\mathbf{R}^n)^*$ is φ_x for a unique $x \in \mathbf{R}^n$.

1-13.* If $x,y \in \mathbf{R}^n$, then x and y are called **perpendicular** (or **orthogonal**) if $\langle x,y\rangle = 0$. If x and y are perpendicular, prove that $|x + y|^2 = |x|^2 + |y|^2$.

SUBSETS OF EUCLIDEAN SPACE

The closed interval $[a,b]$ has a natural analogue in $\mathbf{R}^2$. This is the **closed rectangle** $[a,b] \times [c,d]$, defined as the collection of all pairs (x,y) with $x \in [a,b]$ and $y \in [c,d]$. More generally, if $A \subset \mathbf{R}^m$ and $B \subset \mathbf{R}^n$, then $A \times B \subset \mathbf{R}^{m+n}$ is defined as the set of all $(x,y) \in \mathbf{R}^{m+n}$ with $x \in A$ and $y \in B$. In particular, $\mathbf{R}^{m+n} = \mathbf{R}^m \times \mathbf{R}^n$. If $A \subset \mathbf{R}^m$, $B \subset \mathbf{R}^n$, and $C \subset \mathbf{R}^p$, then $(A \times B) \times C = A \times (B \times C)$, and both of these are denoted simply $A \times B \times C$; this convention is extended to the product of any number of sets. The set $[a_1,b_1] \times \cdots \times [a_n,b_n] \subset \mathbf{R}^n$ is called a **closed rectangle** in $\mathbf{R}^n$, while the set $(a_1,b_1) \times \cdots \times (a_n,b_n) \subset \mathbf{R}^n$ is called an **open rectangle.** More generally a set $U \subset \mathbf{R}^n$ is called **open** (Figure 1-1) if for each $x \in U$ there is an open rectangle A such that $x \in A \subset U$.

A subset C of $\mathbf{R}^n$ is **closed** if $\mathbf{R}^n - C$ is open. For example, if C contains only finitely many points, then C is closed.

FIGURE 1-1

The reader should supply the proof that a closed rectangle in $\mathbf{R}^n$ is indeed a closed set.

If $A \subset \mathbf{R}^n$ and $x \in \mathbf{R}^n$, then one of three possibilities must hold (Figure 1-2):

1. There is an open rectangle B such that $x \in B \subset A$.
2. There is an open rectangle B such that $x \in B \subset \mathbf{R}^n - A$.
3. If B is any open rectangle with $x \in B$, then B contains points of both A and $\mathbf{R}^n - A$.

FIGURE 1-2

Those points satisfying (1) constitute the **interior** of A, those satisfying (2) the **exterior** of A, and those satisfying (3) the **boundary** of A. Problems 1-16 to 1-18 show that these terms may sometimes have unexpected meanings.

It is not hard to see that the interior of any set A is open, and the same is true for the exterior of A, which is, in fact, the interior of $\mathbf{R}^n - A$. Thus (Problem 1-14) their union is open, and what remains, the boundary, must be closed.

A collection $\mathcal{O}$ of open sets is an **open cover** of A (or, briefly, **covers** A) if every point $x \in A$ is in some open set in the collection $\mathcal{O}$. For example, if $\mathcal{O}$ is the collection of all open intervals $(a, a + 1)$ for $a \in \mathbf{R}$, then $\mathcal{O}$ is a cover of $\mathbf{R}$. Clearly no finite number of the open sets in $\mathcal{O}$ will cover $\mathbf{R}$ or, for that matter, any unbounded subset of $\mathbf{R}$. A similar situation can also occur for bounded sets. If $\mathcal{O}$ is the collection of all open intervals $(1/n, 1 - 1/n)$ for all integers $n > 1$, then $\mathcal{O}$ is an open cover of $(0,1)$, but again no finite collection of sets in $\mathcal{O}$ will cover $(0,1)$. Although this phenomenon may not appear particularly scandalous, sets for which this state of affairs cannot occur are of such importance that they have received a special designation: a set A is called **compact** if every open cover $\mathcal{O}$ contains a finite subcollection of open sets which also covers A.

A set with only finitely many points is obviously compact and so is the infinite set A which contains 0 and the numbers $1/n$ for all integers n (reason: if $\mathcal{O}$ is a cover, then $0 \in U$ for some open set U in $\mathcal{O}$; there are only finitely many other points of A not in U, each requiring at most one more open set).

Recognizing compact sets is greatly simplified by the following results, of which only the first has any depth (i.e., uses any facts about the real numbers).

1-3 Theorem (Heine-Borel). *The closed interval* $[a,b]$ *is compact.*

Proof. If $\mathcal{O}$ is an open cover of $[a,b]$, let

$$A = \{x : a \le x \le b \text{ and } [a,x] \text{ is covered by some finite number of open sets in } \mathcal{O}\}.$$

FIGURE 1-3

Note that $a \in A$ and that A is clearly bounded above (by b). We would like to show that $b \in A$. This is done by proving two things about α = least upper bound of A; namely, (1) $\alpha \in A$ and (2) $b = \alpha$.

Since $\mathcal{O}$ is a cover, $\alpha \in U$ for some U in $\mathcal{O}$. Then all points in some interval to the left of α are also in U (see Figure 1-3). Since α is the least upper bound of A, there is an x in this interval such that $x \in A$. Thus $[a,x]$ is covered by some finite number of open sets of $\mathcal{O}$, while $[x,\alpha]$ is covered by the single set U. Hence $[a,\alpha]$ is covered by a finite number of open sets of $\mathcal{O}$, and $\alpha \in A$. This proves (1).

To prove that (2) is true, suppose instead that $\alpha < b$. Then there is a point x' between α and b such that $[\alpha,x'] \subset U$. Since $\alpha \in A$, the interval $[a,\alpha]$ is covered by finitely many open sets of $\mathcal{O}$, while $[\alpha,x']$ is covered by U. Hence $x' \in A$, contradicting the fact that α is an upper bound of A. ∎

If $B \subset \mathbf{R}^m$ is compact and $x \in \mathbf{R}^n$, it is easy to see that $\{x\} \times B \subset \mathbf{R}^{n+m}$ is compact. However, a much stronger assertion can be made.

1-4 Theorem. *If B is compact and $\mathcal{O}$ is an open cover of $\{x\} \times B$, then there is an open set $U \subset \mathbf{R}^n$ containing x such that $U \times B$ is covered by a finite number of sets in $\mathcal{O}$.*

Proof. Since $\{x\} \times B$ is compact, we can assume at the outset that $\mathcal{O}$ is finite, and we need only find the open set U such that $U \times B$ is covered by $\mathcal{O}$.

For each $y \in B$ the point (x,y) is in some open set W in $\mathcal{O}$. Since W is open, we have $(x,y) \in U_y \times V_y \subset W$ for some open rectangle $U_y \times V_y$. The sets V_y cover the compact set B, so a finite number $V_{y_1}, \ldots, V_{y_k}$ also cover B. Let $U = U_{y_1} \cap \cdots \cap U_{y_k}$. Then if $(x',y') \in U \times B$, we have

FIGURE 1-4

$y' \in V_{y_i}$ for some i (Figure 1-4), and certainly $x' \in U_{y_i}$. Hence $(x',y') \in U_{y_i} \times V_{y_i}$, which is contained in some W in $\mathcal{O}$. ∎

1-5 Corollary. *If $A \subset \mathbf{R}^n$ and $B \subset \mathbf{R}^m$ are compact, then $A \times B \subset \mathbf{R}^{n+m}$ is compact.*

Proof. If $\mathcal{O}$ is an open cover of $A \times B$, then $\mathcal{O}$ covers $\{x\} \times B$ for each $x \in A$. By Theorem 1-4 there is an open set U_x containing x such that $U_x \times B$ is covered by finitely many sets in $\mathcal{O}$. Since A is compact, a finite number $U_{x_1}, \ldots, U_{x_n}$ of the U_x cover A. Since finitely many sets in $\mathcal{O}$ cover each $U_{x_i} \times B$, finitely many cover all of $A \times B$. ∎

1-6 Corollary. *$A_1 \times \cdots \times A_k$ is compact if each A_i is. In particular, a closed rectangle in $\mathbf{R}^k$ is compact.*

1-7 Corollary. *A closed bounded subset of* $\mathbf{R}^n$ *is compact.* (The converse is also true (Problem 1-20).)

Proof. If $A \subset \mathbf{R}^n$ is closed and bounded, then $A \subset B$ for some closed rectangle B. If $\mathcal{O}$ is an open cover of A, then $\mathcal{O}$ together with $\mathbf{R}^n - A$ is an open cover of B. Hence a finite number $U_1, \ldots, U_n$ of sets in $\mathcal{O}$, together with $\mathbf{R}^n - A$ perhaps, cover B. Then $U_1, \ldots, U_n$ cover A. ∎

Problems. **1-14.*** Prove that the union of any (even infinite) number of open sets is open. Prove that the intersection of two (and hence of finitely many) open sets is open. Give a counterexample for infinitely many open sets.

1-15. Prove that $\{x \in \mathbf{R}^n: |x - a| < r\}$ is open (see also Problem 1-27).

1-16. Find the interior, exterior, and boundary of the sets

$$\{x \in \mathbf{R}^n: |x| \leq 1\}$$
$$\{x \in \mathbf{R}^n: |x| = 1\}$$
$$\{x \in \mathbf{R}^n: \text{each } x^i \text{ is rational}\}.$$

1-17. Construct a set $A \subset [0,1] \times [0,1]$ such that A contains at most one point on each horizontal and each vertical line but boundary $A = [0,1] \times [0,1]$. *Hint:* It suffices to ensure that A contains points in each quarter of the square $[0,1] \times [0,1]$ and also in each sixteenth, etc.

1-18. If $A \subset [0,1]$ is the union of open intervals (a_i,b_i) such that each rational number in $(0,1)$ is contained in some (a_i,b_i), show that boundary $A = [0,1] - A$.

1-19.* If A is a closed set that contains every rational number $r \in [0,1]$, show that $[0,1] \subset A$.

1-20. Prove the converse of Corollary 1-7: A compact subset of $\mathbf{R}^n$ is closed and bounded (see also Problem 1-28).

1-21.* (a) If A is closed and $x \notin A$, prove that there is a number $d > 0$ such that $|y - x| \geq d$ for all $y \in A$.

(b) If A is closed, B is compact, and $A \cap B = \varnothing$, prove that there is $d > 0$ such that $|y - x| \geq d$ for all $y \in A$ and $x \in B$. *Hint:* For each $b \in B$ find an open set U containing b such that this relation holds for $x \in U \cap B$.

(c) Give a counterexample in $\mathbf{R}^2$ if A and B are closed but neither is compact.

1-22.* If U is open and $C \subset U$ is compact, show that there is a compact set D such that $C \subset \text{interior } D$ and $D \subset U$.

FUNCTIONS AND CONTINUITY

A **function** from $\mathbf{R}^n$ to $\mathbf{R}^m$ (sometimes called a (vector-valued) function of n variables) is a rule which associates to each point in $\mathbf{R}^n$ some point in $\mathbf{R}^m$; the point a function f associates to x is denoted $f(x)$. We write $f\colon \mathbf{R}^n \to \mathbf{R}^m$ (read "f takes $\mathbf{R}^n$ into $\mathbf{R}^m$" or "f, taking $\mathbf{R}^n$ into $\mathbf{R}^m$," depending on context) to indicate that $f(x) \in \mathbf{R}^m$ is defined for $x \in \mathbf{R}^n$. The notation $f\colon A \to \mathbf{R}^m$ indicates that $f(x)$ is defined only for x in the set A, which is called the **domain** of f. If $B \subset A$, we define $f(B)$ as the set of all $f(x)$ for $x \in B$, and if $C \subset \mathbf{R}^m$ we define $f^{-1}(C) = \{x \in A : f(x) \in C\}$. The notation $f\colon A \to B$ indicates that $f(A) \subset B$.

A convenient representation of a function $f\colon \mathbf{R}^2 \to \mathbf{R}$ may be obtained by drawing a picture of its graph, the set of all 3-tuples of the form $(x,y,f(x,y))$, which is actually a figure in 3-space (see, e.g., Figures 2-1 and 2-2 of Chapter 2).

If $f,g\colon \mathbf{R}^n \to \mathbf{R}$, the functions $f + g$, $f - g$, $f \cdot g$, and f/g are defined precisely as in the one-variable case. If $f\colon A \to \mathbf{R}^m$ and $g\colon B \to \mathbf{R}^p$, where $B \subset \mathbf{R}^m$, then the **composition** $g \circ f$ is defined by $g \circ f(x) = g(f(x))$; the domain of $g \circ f$ is $A \cap f^{-1}(B)$. If $f\colon A \to \mathbf{R}^m$ is 1-1, that is, if $f(x) \neq f(y)$ when $x \neq y$, we define $f^{-1}\colon f(A) \to \mathbf{R}^n$ by the requirement that $f^{-1}(z)$ is the unique $x \in A$ with $f(x) = z$.

A function $f\colon A \to \mathbf{R}^m$ determines m **component functions** $f^1, \ldots, f^m\colon A \to \mathbf{R}$ by $f(x) = (f^1(x), \ldots, f^m(x))$. If conversely, m functions $g_1, \ldots, g_m\colon A \to \mathbf{R}$ are given, there is a unique function $f\colon A \to \mathbf{R}^m$ such that $f^i = g_i$, namely $f(x) = (g_1(x), \ldots, g_m(x))$. This function f will be denoted $(g_1, \ldots, g_m)$, so that we always have $f = (f^1, \ldots, f^m)$. If $\pi\colon \mathbf{R}^n \to \mathbf{R}^n$ is the identity function, $\pi(x) = x$, then $\pi^i(x) = x^i$; the function π^i is called the ith **projection function.**

The notation $\lim_{x \to a} f(x) = b$ means, as in the one-variable case, that we can get $f(x)$ as close to b as desired, by choosing x sufficiently close to, but not equal to, a. In mathematical terms this means that for every number $\varepsilon > 0$ there is a number $\delta > 0$ such that $|f(x) - b| < \varepsilon$ for all x in the domain of f which

satisfy $0 < |x - a| < \delta$. A function $f: A \to \mathbf{R}^m$ is called **continuous** at $a \in A$ if $\lim_{x \to a} f(x) = f(a)$, and f is simply called continuous if it is continuous at each $a \in A$. One of the pleasant surprises about the concept of continuity is that it can be defined without using limits. It follows from the next theorem that $f: \mathbf{R}^n \to \mathbf{R}^m$ is continuous if and only if $f^{-1}(U)$ is open whenever $U \subset \mathbf{R}^m$ is open; if the domain of f is not all of $\mathbf{R}^n$, a slightly more complicated condition is needed.

1-8 Theorem. *If $A \subset \mathbf{R}^n$, a function $f: A \to \mathbf{R}^m$ is continuous if and only if for every open set $U \subset \mathbf{R}^m$ there is some open set $V \subset \mathbf{R}^n$ such that $f^{-1}(U) = V \cap A$.*

Proof. Suppose f is continuous. If $a \in f^{-1}(U)$, then $f(a) \in U$. Since U is open, there is an open rectangle B with $f(a) \in B \subset U$. Since f is continuous at a, we can ensure that $f(x) \in B$, provided we choose x in some sufficiently small rectangle C containing a. Do this for each $a \in f^{-1}(U)$ and let V be the union of all such C. Clearly $f^{-1}(U) = V \cap A$. The converse is similar and is left to the reader. ∎

The following consequence of Theorem 1-8 is of great importance.

1-9 Theorem. *If $f: A \to \mathbf{R}^m$ is continuous, where $A \subset \mathbf{R}^n$, and A is compact, then $f(A) \subset \mathbf{R}^m$ is compact.*

Proof. Let $\mathcal{O}$ be an open cover of $f(A)$. For each open set U in $\mathcal{O}$ there is an open set V_U such that $f^{-1}(U) = V_U \cap A$. The collection of all V_U is an open cover of A. Since A is compact, a finite number $V_{U_1}, \ldots, V_{U_n}$ cover A. Then $U_1, \ldots, U_n$ cover $f(A)$. ∎

If $f: A \to \mathbf{R}$ is bounded, the extent to which f fails to be continuous at $a \in A$ can be measured in a precise way. For $\delta > 0$ let

$$M(a,f,\delta) = \sup\{f(x): x \in A \text{ and } |x - a| < \delta\},$$
$$m(a,f,\delta) = \inf\{f(x): x \in A \text{ and } |x - a| < \delta\}.$$

The **oscillation** $o(f,a)$ of f at a is defined by $o(f,a) = \lim_{\delta \to 0}[M(a,f,\delta) - m(a,f,\delta)]$. This limit always exists, since $M(a,f,\delta) - m(a,f,\delta)$ decreases as δ decreases. There are two important facts about $o(f,a)$.

1-10 Theorem. *The bounded function f is continuous at a if and only if $o(f,a) = 0$.*

Proof. Let f be continuous at a. For every number $\varepsilon > 0$ we can choose a number $\delta > 0$ so that $|f(x) - f(a)| < \varepsilon$ for all $x \in A$ with $|x - a| < \delta$; thus $M(a,f,\delta) - m(a,f,\delta) \leq 2\varepsilon$. Since this is true for every ε, we have $o(f,a) = 0$. The converse is similar and is left to the reader. ∎

1-11 Theorem. *Let $A \subset \mathbf{R}^n$ be closed. If $f\colon A \to \mathbf{R}$ is any bounded function, and $\varepsilon > 0$, then $\{x \in A\colon o(f,x) \geq \varepsilon\}$ is closed.*

Proof. Let $B = \{x \in A\colon o(f,x) \geq \varepsilon\}$. We wish to show that $\mathbf{R}^n - B$ is open. If $x \in \mathbf{R}^n - B$, then either $x \notin A$ or else $x \in A$ and $o(f,x) < \varepsilon$. In the first case, since A is closed, there is an open rectangle C containing x such that $C \subset \mathbf{R}^n - A \subset \mathbf{R}^n - B$. In the second case there is a $\delta > 0$ such that $M(x,f,\delta) - m(x,f,\delta) < \varepsilon$. Let C be an open rectangle containing x such that $|x - y| < \delta$ for all $y \in C$. Then if $y \in C$ there is a δ_1 such that $|x - z| < \delta$ for all z satisfying $|z - y| < \delta_1$. Thus $M(y,f,\delta_1) - m(y,f,\delta_1) < \varepsilon$, and consequently $o(y,f) < \varepsilon$. Therefore $C \subset \mathbf{R}^n - B$. ∎

Problems. **1-23.** If $f\colon A \to \mathbf{R}^m$ and $a \in A$, show that $\lim_{x \to a} f(x) = b$ if and only if $\lim_{x \to a} f^i(x) = b^i$ for $i = 1, \ldots, m$.

1-24. Prove that $f\colon A \to \mathbf{R}^m$ is continuous at a if and only if each f^i is.

1-25. Prove that a linear transformation $T\colon \mathbf{R}^n \to \mathbf{R}^m$ is continuous. *Hint:* Use Problem 1-10.

1-26. Let $A = \{(x,y) \in \mathbf{R}^2\colon x > 0 \text{ and } 0 < y < x^2\}$.

(a) Show that every straight line through (0,0) contains an interval around (0,0) which is in $\mathbf{R}^2 - A$.

(b) Define $f\colon \mathbf{R}^2 \to \mathbf{R}$ by $f(x) = 0$ if $x \notin A$ and $f(x) = 1$ if $x \in A$. For $h \in \mathbf{R}^2$ define $g_h\colon \mathbf{R} \to \mathbf{R}$ by $g_h(t) = f(th)$. Show that each g_h is continuous at 0, but f is not continuous at (0,0).

1-27. Prove that $\{x \in \mathbf{R}^n: |x - a| < r\}$ is open by considering the function $f: \mathbf{R}^n \to \mathbf{R}$ with $f(x) = |x - a|$.

1-28. If $A \subset \mathbf{R}^n$ is not closed, show that there is a continuous function $f: A \to \mathbf{R}$ which is unbounded. *Hint:* If $x \in \mathbf{R}^n - A$ but $x \notin$ interior $(\mathbf{R}^n - A)$, let $f(y) = 1/|y - x|$.

1-29. If A is compact, prove that every continuous function $f: A \to \mathbf{R}$ takes on a maximum and a minimum value.

1-30. Let $f: [a,b] \to \mathbf{R}$ be an increasing function. If $x_1, \ldots, x_n \in [a,b]$ are distinct, show that $\sum_{i=1}^{n} o(f,x_i) < f(b) - f(a)$.

2

Differentiation

BASIC DEFINITIONS

Recall that a function $f\colon \mathbf{R} \to \mathbf{R}$ is differentiable at $a \in \mathbf{R}$ if there is a number $f'(a)$ such that

$$(1)\quad \lim_{h\to 0} \frac{f(a+h) - f(a)}{h} = f'(a).$$

This equation certainly makes no sense in the general case of a function $f\colon \mathbf{R}^n \to \mathbf{R}^m$, but can be reformulated in a way that does. If $\lambda\colon \mathbf{R} \to \mathbf{R}$ is the linear transformation defined by $\lambda(h) = f'(a) \cdot h$, then equation (1) is equivalent to

$$(2)\quad \lim_{h\to 0} \frac{f(a+h) - f(a) - \lambda(h)}{h} = 0.$$

Equation (2) is often interpreted as saying that $\lambda + f(a)$ is a good approximation to f at a (see Problem 2-9). Henceforth we focus our attention on the linear transformation λ and reformulate the definition of differentiability as follows.

A function $f\colon \mathbf{R} \to \mathbf{R}$ is differentiable at $a \in \mathbf{R}$ if there is a linear transformation $\lambda\colon \mathbf{R} \to \mathbf{R}$ such that

$$\lim_{h\to 0} \frac{f(a+h) - f(a) - \lambda(h)}{h} = 0.$$

In this form the definition has a simple generalization to higher dimensions:

A function $f\colon \mathbf{R}^n \to \mathbf{R}^m$ is **differentiable** at $a \in \mathbf{R}^n$ if there is a linear transformation $\lambda\colon \mathbf{R}^n \to \mathbf{R}^m$ such that

$$\lim_{h\to 0} \frac{|f(a+h) - f(a) - \lambda(h)|}{|h|} = 0.$$

Note that h is a point of $\mathbf{R}^n$ and $f(a+h) - f(a) - \lambda(h)$ a point of $\mathbf{R}^m$, so the norm signs are essential. The linear transformation λ is denoted $Df(a)$ and called the **derivative** of f at a. The justification for the phrase "*the* linear transformation λ" is

2-1 Theorem. *If $f\colon \mathbf{R}^n \to \mathbf{R}^m$ is differentiable at $a \in \mathbf{R}^n$ there is a* unique *linear transformation $\lambda\colon \mathbf{R}^n \to \mathbf{R}^m$ such that*

$$\lim_{h\to 0} \frac{|f(a+h) - f(a) - \lambda(h)|}{|h|} = 0.$$

Proof. Suppose $\mu\colon \mathbf{R}^n \to \mathbf{R}^m$ satisfies

$$\lim_{h\to 0} \frac{|f(a+h) - f(a) - \mu(h)|}{|h|} = 0.$$

If $d(h) = f(a+h) - f(a)$, then

$$\begin{aligned}\lim_{h\to 0} \frac{|\lambda(h) - \mu(h)|}{|h|} &= \lim_{h\to 0} \frac{|\lambda(h) - d(h) + d(h) - \mu(h)|}{|h|} \\ &\le \lim_{h\to 0} \frac{|\lambda(h) - d(h)|}{|h|} + \lim_{h\to 0} \frac{|d(h) - \mu(h)|}{|h|} \\ &= 0.\end{aligned}$$

If $x \in \mathbf{R}^n$, then $tx \to 0$ as $t \to 0$. Hence for $x \neq 0$ we have

$$0 = \lim_{t\to 0} \frac{|\lambda(tx) - \mu(tx)|}{|tx|} = \frac{|\lambda(x) - \mu(x)|}{|x|}.$$

Therefore $\lambda(x) = \mu(x)$. ∎

We shall later discover a simple way of finding $Df(a)$. For the moment let us consider the function $f\colon \mathbf{R}^2 \to \mathbf{R}$ defined by $f(x,y) = \sin x$. Then $Df(a,b) = \lambda$ satisfies $\lambda(x,y) = (\cos a) \cdot x$. To prove this, note that

$$\lim_{(h,k)\to 0} \frac{|f(a+h,\, b+k) - f(a,b) - \lambda(h,k)|}{|(h,k)|}$$
$$= \lim_{(h,k)\to 0} \frac{|\sin(a+h) - \sin a - (\cos a)\cdot h|}{|(h,k)|}.$$

Since $\sin'(a) = \cos a$, we have

$$\lim_{h\to 0} \frac{|\sin(a+h) - \sin a - (\cos a)\cdot h|}{|h|} = 0.$$

Since $|(h,k)| \geq |h|$, it is also true that

$$\lim_{h\to 0} \frac{|\sin(a+h) - \sin a - (\cos a)\cdot h|}{|(h,k)|} = 0.$$

It is often convenient to consider the matrix of $Df(a)\colon \mathbf{R}^n \to \mathbf{R}^m$ with respect to the usual bases of $\mathbf{R}^n$ and $\mathbf{R}^m$. This $m \times n$ matrix is called the **Jacobian matrix** of f at a, and denoted $f'(a)$. If $f(x,y) = \sin x$, then $f'(a,b) = (\cos a, 0)$. If $f\colon \mathbf{R} \to \mathbf{R}$, then $f'(a)$ is a 1×1 matrix whose single entry is the number which is denoted $f'(a)$ in elementary calculus.

The definition of $Df(a)$ could be made if f were defined only in some open set containing a. Considering only functions defined on $\mathbf{R}^n$ streamlines the statement of theorems and produces no real loss of generality. It is convenient to define a function $f\colon \mathbf{R}^n \to \mathbf{R}^m$ to be differentiable on A if f is differentiable at a for each $a \in A$. If $f\colon A \to \mathbf{R}^m$, then f is called differentiable if f can be extended to a differentiable function on some open set containing A.

Problems. **2-1.*** Prove that if $f\colon \mathbf{R}^n \to \mathbf{R}^m$ is differentiable at $a \in \mathbf{R}^n$, then it is continuous at a. *Hint:* Use Problem 1-10.

2-2. A function $f\colon \mathbf{R}^2 \to \mathbf{R}$ is **independent of the second variable** if for each $x \in \mathbf{R}$ we have $f(x,y_1) = f(x,y_2)$ for all $y_1,y_2 \in \mathbf{R}$. Show that f is independent of the second variable if and only if there is a function $g\colon \mathbf{R} \to \mathbf{R}$ such that $f(x,y) = g(x)$. What is $f'(a,b)$ in terms of g'?

2-3. Define when a function $f\colon \mathbf{R}^2 \to \mathbf{R}$ is independent of the first variable and find $f'(a,b)$ for such f. Which functions are independent of the first variable and also of the second variable?

2-4. Let g be a continuous real-valued function on the unit circle $\{x \in \mathbf{R}^2\colon |x| = 1\}$ such that $g(0,1) = g(1,0) = 0$ and $g(-x) = -g(x)$. Define $f\colon \mathbf{R}^2 \to \mathbf{R}$ by

$$f(x) = \begin{cases} |x| \cdot g\left(\dfrac{x}{|x|}\right) & x \neq 0, \\ 0 & x = 0. \end{cases}$$

(a) If $x \in \mathbf{R}^2$ and $h\colon \mathbf{R} \to \mathbf{R}$ is defined by $h(t) = f(tx)$, show that h is differentiable.

(b) Show that f is not differentiable at $(0,0)$ unless $g = 0$. *Hint:* First show that $Df(0,0)$ would have to be 0 by considering (h,k) with $k = 0$ and then with $h = 0$.

2-5. Let $f\colon \mathbf{R}^2 \to \mathbf{R}$ be defined by

$$f(x,y) = \begin{cases} \dfrac{x|y|}{\sqrt{x^2 + y^2}} & (x,y) \neq 0, \\ 0 & (x,y) = 0. \end{cases}$$

Show that f is a function of the kind considered in Problem 2-4, so that f is not differentiable at $(0,0)$.

2-6. Let $f\colon \mathbf{R}^2 \to \mathbf{R}$ be defined by $f(x,y) = \sqrt{|xy|}$. Show that f is not differentiable at $(0,0)$.

2-7. Let $f\colon \mathbf{R}^n \to \mathbf{R}$ be a function such that $|f(x)| \leq |x|^2$. Show that f is differentiable at 0.

2-8. Let $f\colon \mathbf{R} \to \mathbf{R}^2$. Prove that f is differentiable at $a \in \mathbf{R}$ if and only if f^1 and f^2 are, and that in this case

$$f'(a) = \begin{pmatrix} (f^1)'(a) \\ (f^2)'(a) \end{pmatrix}.$$

2-9. Two functions $f,g\colon \mathbf{R} \to \mathbf{R}$ are **equal up to nth order** at a if

$$\lim_{h \to 0} \frac{f(a+h) - g(a+h)}{h^n} = 0.$$

(a) Show that f is differentiable at a if and only if there is a function g of the form $g(x) = a_0 + a_1(x - a)$ such that f and g are equal up to first order at a.

(b) If $f'(a), \ldots, f^{(n)}(a)$ exist, show that f and the function g defined by

$$g(x) = \sum_{i=0}^{n} \frac{f^{(i)}(a)}{i!}(x - a)^i$$

are equal up to nth order at a. *Hint:* The limit

$$\lim_{x \to a} \frac{f(x) - \sum_{i=0}^{n-1} \frac{f^{(i)}(a)}{i!}(x-a)^i}{(x-a)^n}$$

may be evaluated by L'Hospital's rule.

BASIC THEOREMS

2-2 Theorem (Chain Rule). *If f: $\mathbf{R}^n \to \mathbf{R}^m$ is differentiable at a, and g: $\mathbf{R}^m \to \mathbf{R}^p$ is differentiable at $f(a)$, then the composition $g \circ f$: $\mathbf{R}^n \to \mathbf{R}^p$ is differentiable at a, and*

$$D(g \circ f)(a) = Dg(f(a)) \circ Df(a).$$

Remark. This equation can be written

$$(g \circ f)'(a) = g'(f(a)) \cdot f'(a).$$

If $m = n = p = 1$, we obtain the old chain rule.

Proof. Let $b = f(a)$, let $\lambda = Df(a)$, and let $\mu = Dg(f(a))$. If we define

$$\begin{aligned}
&(1)\ \varphi(x) = f(x) - f(a) - \lambda(x-a),\\
&(2)\ \psi(y) = g(y) - g(b) - \mu(y-b),\\
&(3)\ \rho(x) = g \circ f(x) - g \circ f(a) - \mu \circ \lambda(x-a),
\end{aligned}$$

then

$$(4)\ \lim_{x \to a} \frac{|\varphi(x)|}{|x-a|} = 0,$$

$$(5)\ \lim_{y \to b} \frac{|\psi(y)|}{|y-b|} = 0,$$

and we must show that

$$\lim_{x \to a} \frac{|\rho(x)|}{|x-a|} = 0.$$

Now

$$\begin{aligned}
\rho(x) &= g(f(x)) - g(b) - \mu(\lambda(x-a))\\
&= g(f(x)) - g(b) - \mu(f(x) - f(a) - \varphi(x)) \qquad \text{by (1)}\\
&= [g(f(x)) - g(b) - \mu(f(x) - f(a))] + \mu(\varphi(x))\\
&= \psi(f(x)) + \mu(\varphi(x)) \qquad \text{by (2).}
\end{aligned}$$

Thus we must prove

$$(6)\quad \lim_{x\to a} \frac{|\psi(f(x))|}{|x-a|} = 0,$$

$$(7)\quad \lim_{x\to a} \frac{|\mu(\varphi(x))|}{|x-a|} = 0.$$

Equation (7) follows easily from (4) and Problem 1-10. If $\varepsilon > 0$ it follows from (5) that for some $\delta > 0$ we have

$$|\psi(f(x))| < \varepsilon|f(x) - b| \qquad \text{if } |f(x) - b| < \delta,$$

which is true if $|x - a| < \delta_1$, for a suitable δ_1. Then

$$\begin{aligned} |\psi(f(x))| &< \varepsilon|f(x) - b| \\ &= \varepsilon|\varphi(x) + \lambda(x-a)| \\ &\le \varepsilon|\varphi(x)| + \varepsilon M|x-a| \end{aligned}$$

for some M, by Problem 1-10. Equation (6) now follows easily. ▮

2-3 Theorem

(1) *If f: $\mathbf{R}^n \to \mathbf{R}^m$ is a constant function (that is, if for some $y \in \mathbf{R}^m$ we have $f(x) = y$ for all $x \in \mathbf{R}^n$), then*

$$Df(a) = 0.$$

(2) *If f: $\mathbf{R}^n \to \mathbf{R}^m$ is a linear transformation, then*

$$Df(a) = f.$$

(3) *If f: $\mathbf{R}^n \to \mathbf{R}^m$, then f is differentiable at $a \in \mathbf{R}^n$ if and only if each f^i is, and*

$$Df(a) = (Df^1(a), \ldots, Df^m(a)).$$

Thus $f'(a)$ is the $m \times n$ matrix whose ith row is $(f^i)'(a)$.

(4) *If s: $\mathbf{R}^2 \to \mathbf{R}$ is defined by $s(x,y) = x + y$, then*

$$Ds(a,b) = s.$$

(5) *If p: $\mathbf{R}^2 \to \mathbf{R}$ is defined by $p(x,y) = x \cdot y$, then*

$$Dp(a,b)(x,y) = bx + ay.$$

Thus $p'(a,b) = (b,a)$.

Proof

(1) $$\lim_{h\to 0} \frac{|f(a+h) - f(a) - 0|}{|h|} = \lim_{h\to 0} \frac{|y - y - 0|}{|h|} = 0.$$

(2) $$\lim_{h\to 0} \frac{|f(a+h) - f(a) - f(h)|}{|h|}$$
$$= \lim_{h\to 0} \frac{|f(a) + f(h) - f(a) - f(h)|}{|h|} = 0.$$

(3) If each f^i is differentiable at a and

$$\lambda = (Df^1(a), \ldots, Df^m(a)),$$

then

$$f(a+h) - f(a) - \lambda(h)$$
$$= (f^1(a+h) - f^1(a) - Df^1(a)(h), \ldots,$$
$$f^m(a+h) + f^m(a) - Df^m(a)(h)).$$

Therefore

$$\lim_{h\to 0} \frac{|f(a+h) - f(a) - \lambda(h)|}{|h|}$$
$$\leq \lim_{h\to 0} \sum_{i=1}^{m} \frac{|f^i(a+h) - f^i(a) - Df^i(a)(h)|}{|h|} = 0.$$

If, on the other hand, f is differentiable at a, then $f^i = \pi^i \circ f$ is differentiable at a by (2) and Theorem 2-2.

(4) follows from (2).

(5) Let $\lambda(x,y) = bx + ay$. Then

$$\lim_{(h,k)\to 0} \frac{|p(a+h, b+k) - p(a,b) - \lambda(h,k)|}{|(h,k)|}$$
$$= \lim_{(h,k)\to 0} \frac{|hk|}{|(h,k)|}.$$

Now

$$|hk| \leq \begin{cases} |h|^2 & \text{if } |k| \leq |h|, \\ |k|^2 & \text{if } |h| \leq |k|. \end{cases}$$

Hence $|hk| \leq |h|^2 + |k|^2$. Therefore

$$\frac{|hk|}{|(h,k)|} \leq \frac{h^2 + k^2}{\sqrt{h^2 + k^2}} = \sqrt{h^2 + k^2},$$

so

$$\lim_{(h,k)\to 0} \frac{|hk|}{|(h,k)|} = 0. \quad \blacksquare$$

2-4 Corollary. *If f,g: $\mathbf{R}^n \to \mathbf{R}$ are differentiable at a, then*

$$D(f+g)(a) = Df(a) + Dg(a),$$
$$D(f \cdot g)(a) = g(a)Df(a) + f(a)Dg(a).$$

If, moreover, $g(a) \neq 0$, then

$$D(f/g)(a) = \frac{g(a)Df(a) - f(a)Dg(a)}{[g(a)]^2}.$$

Proof. We will prove the first equation and leave the others to the reader. Since $f + g = s \circ (f,g)$, we have

$$\begin{aligned} D(f+g)(a) &= Ds(f(a),g(a)) \circ D(f,g)(a) \\ &= s \circ (Df(a),Dg(a)) \\ &= Df(a) + Dg(a). \quad \blacksquare \end{aligned}$$

We are now assured of the differentiability of those functions f: $\mathbf{R}^n \to \mathbf{R}^m$, whose component functions are obtained by addition, multiplication, division, and composition, from the functions π^i (which are linear transformations) and the functions which we can already differentiate by elementary calculus. Finding $Df(x)$ or $f'(x)$, however, may be a fairly formidable task. For example, let f: $\mathbf{R}^2 \to \mathbf{R}$ be defined by $f(x,y) = \sin(xy^2)$. Since $f = \sin \circ (\pi^1 \cdot [\pi^2]^2)$, we have

$$\begin{aligned} f'(a,b) &= \sin'(ab^2) \cdot [b^2(\pi^1)'(a,b) + a([\pi^2]^2)'(a,b)] \\ &= \sin'(ab^2) \cdot [b^2(\pi^1)'(a,b) + 2ab(\pi^2)'(a,b)] \\ &= (\cos(ab^2)) \cdot [b^2(1,0) + 2ab(0,1)] \\ &= (b^2 \cos(ab^2),\ 2ab \cos(ab^2)). \end{aligned}$$

Fortunately, we will soon discover a much simpler method of computing f'.

Problems. **2-10.** Use the theorems of this section to find f' for the following:

(a) $f(x,y,z) = x^y$.
(b) $f(x,y,z) = (x^y,z)$.
(c) $f(x,y) = \sin(x \sin y)$.
(d) $f(x,y,z) = \sin(x \sin(y \sin z))$.
(e) $f(x,y,z) = x^{y^z}$.
(f) $f(x,y,z) = x^{y+z}$.
(g) $f(x,y,z) = (x + y)^z$.
(h) $f(x,y) = \sin(xy)$.
(i) $f(x,y) = [\sin(xy)]^{\cos 3}$.
(j) $f(x,y) = (\sin(xy), \sin(x \sin y), x^y)$.

2-11. Find f' for the following (where g: $\mathbf{R} \to \mathbf{R}$ is continuous):

(a) $f(x,y) = \int_a^{x+y} g$.
(b) $f(x,y) = \int_a^{x \cdot y} g$.
(c) $f(x,y,z) = \int_{xy}^{\sin(x \sin(y \sin z))} g$.

2-12. A function f: $\mathbf{R}^n \times \mathbf{R}^m \to \mathbf{R}^p$ is **bilinear** if for $x,x_1,x_2 \in \mathbf{R}^n$, $y,y_1,y_2 \in \mathbf{R}^m$, and $a \in \mathbf{R}$ we have

$$\begin{aligned} f(ax,y) &= af(x,y) = f(x,ay), \\ f(x_1 + x_2,y) &= f(x_1,y) + f(x_2,y), \\ f(x,y_1 + y_2) &= f(x,y_1) + f(x,y_2). \end{aligned}$$

(a) Prove that if f is bilinear, then

$$\lim_{(h,k)\to 0} \frac{|f(h,k)|}{|(h,k)|} = 0.$$

(b) Prove that $Df(a,b)(x,y) = f(a,y) + f(x,b)$.

(c) Show that the formula for $Dp(a,b)$ in Theorem 2-3 is a special case of (b).

2-13. Define IP: $\mathbf{R}^n \times \mathbf{R}^n \to \mathbf{R}$ by $IP(x,y) = \langle x,y \rangle$.

(a) Find $D(IP)(a,b)$ and $(IP)'(a,b)$.

(b) If f,g: $\mathbf{R} \to \mathbf{R}^n$ are differentiable and h: $\mathbf{R} \to \mathbf{R}$ is defined by $h(t) = \langle f(t),g(t) \rangle$, show that

$$h'(a) = \langle f'(a)^{\mathrm{T}},g(a) \rangle + \langle f(a),g'(a)^{\mathrm{T}} \rangle.$$

(Note that $f'(a)$ is an $n \times 1$ matrix; its transpose $f'(a)^{\mathrm{T}}$ is a $1 \times n$ matrix, which we consider as a member of $\mathbf{R}^n$.)

(c) If f: $\mathbf{R} \to \mathbf{R}^n$ is differentiable and $|f(t)| = 1$ for all t, show that $\langle f'(t)^{\mathrm{T}},f(t) \rangle = 0$.

(d) Exhibit a differentiable function f: $\mathbf{R} \to \mathbf{R}$ such that the function $|f|$ defined by $|f|(t) = |f(t)|$ is not differentiable.

2-14. Let E_i, $i = 1, \ldots ,k$ be Euclidean spaces of various dimensions. A function f: $E_1 \times \cdots \times E_k \to \mathbf{R}^p$ is called **multilinear** if for each choice of $x_j \in E_j$, $j \neq i$ the function g: $E_i \to \mathbf{R}^p$ defined by $g(x) = f(x_1, \ldots ,x_{i-1},x,x_{i+1}, \ldots ,x_k)$ is a linear transformation.

(a) If f is multilinear and $i \neq j$, show that for $h = (h_1, \ldots, h_k)$, with $h_l \in E_l$, we have

$$\lim_{h \to 0} \frac{|f(a_1, \ldots, h_i, \ldots, h_j, \ldots, a_k)|}{|h|} = 0.$$

Hint: If $g(x,y) = f(a_1, \ldots, x, \ldots, y, \ldots, a_k)$, then g is bilinear.

(b) Prove that

$$Df(a_1, \ldots, a_k)(x_1, \ldots, x_k) = \sum_{i=1}^{k} f(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_k).$$

2-15. Regard an $n \times n$ matrix as a point in the n-fold product $\mathbf{R}^n \times \cdots \times \mathbf{R}^n$ by considering each row as a member of $\mathbf{R}^n$.

(a) Prove that det: $\mathbf{R}^n \times \cdots \times \mathbf{R}^n \to \mathbf{R}$ is differentiable and

$$D(\det)(a_1, \ldots, a_n)(x_1, \ldots, x_n) = \sum_{i=1}^{n} \det \begin{pmatrix} a_1 \\ \cdot \\ \cdot \\ \cdot \\ x_i \\ \cdot \\ \cdot \\ \cdot \\ a_n \end{pmatrix}.$$

(b) If a_{ij}: $\mathbf{R} \to \mathbf{R}$ are differentiable and $f(t) = \det(a_{ij}(t))$, show that

$$f'(t) = \sum_{j=1}^{n} \det \begin{pmatrix} a_{11}(t), & \ldots & ,a_{1n}(t) \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{j1}'(t), & \ldots & ,a_{jn}'(t) \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{n1}(t), & \ldots & ,a_{nn}(t) \end{pmatrix}.$$

(c) If $\det(a_{ij}(t)) \neq 0$ for all t and $b_1, \ldots, b_n$: $\mathbf{R} \to \mathbf{R}$ are differentiable, let $s_1, \ldots, s_n$: $\mathbf{R} \to \mathbf{R}$ be the functions such that $s_1(t), \ldots, s_n(t)$ are the solutions of the equations

$$\sum_{j=1}^{n} a_{ji}(t) s_j(t) = b_i(t) \qquad i = 1, \ldots, n.$$

Show that s_i is differentiable and find $s_i'(t)$.

2-16. Suppose f: $\mathbf{R}^n \to \mathbf{R}^n$ is differentiable and has a differentiable inverse f^{-1}: $\mathbf{R}^n \to \mathbf{R}^n$. Show that $(f^{-1})'(a) = [f'(f^{-1}(a))]^{-1}$. *Hint:* $f \circ f^{-1}(x) = x$.

PARTIAL DERIVATIVES

We begin the attack on the problem of finding derivatives "one variable at a time." If f: $\mathbf{R}^n \to \mathbf{R}$ and $a \in \mathbf{R}^n$, the limit

$$\lim_{h \to 0} \frac{f(a^1, \ldots, a^i + h, \ldots, a^n) - f(a^1, \ldots, a^n)}{h},$$

if it exists, is denoted $D_i f(a)$, and called the ith **partial derivative** of f at a. It is important to note that $D_i f(a)$ is the ordinary derivative of a certain function; in fact, if $g(x) = f(a^1, \ldots, x, \ldots, a^n)$, then $D_i f(a) = g'(a^i)$. This means that $D_i f(a)$ is the slope of the tangent line at $(a, f(a))$ to the curve obtained by intersecting the graph of f with the plane $x^j = a^j$, $j \neq i$ (Figure 2-1). It also means that computation of $D_i f(a)$ is a problem we can already solve. If $f(x^1, \ldots, x^n)$ is

FIGURE 2-1

given by some formula involving $x^1, \ldots, x^n$, then we find $D_i f(x^1, \ldots, x^n)$ by differentiating the function whose value at x^i is given by the formula when all x^j, for $j \neq i$, are thought of as constants. For example, if $f(x,y) = \sin(xy^2)$, then $D_1 f(x,y) = y^2 \cos(xy^2)$ and $D_2 f(x,y) = 2xy \cos(xy^2)$. If, instead, $f(x,y) = x^y$, then $D_1 f(x,y) = yx^{y-1}$ and $D_2 f(x,y) = x^y \log x$.

With a little practice (e.g., the problems at the end of this section) you should acquire as great a facility for computing $D_i f$ as you already have for computing ordinary derivatives.

If $D_i f(x)$ exists for all $x \in \mathbf{R}^n$, we obtain a function $D_i f$: $\mathbf{R}^n \to \mathbf{R}$. The jth partial derivative of this function at x, that is, $D_j(D_i f)(x)$, is often denoted $D_{i,j} f(x)$. Note that this notation reverses the order of i and j. As a matter of fact, the order is usually irrelevant, since most functions (an exception is given in the problems) satisfy $D_{i,j} f = D_{j,i} f$. There are various delicate theorems ensuring this equality; the following theorem is quite adequate. We state it here but postpone the proof until later (Problem 3-28).

2-5 Theorem. *If $D_{i,j} f$ and $D_{j,i} f$ are continuous in an open set containing a, then*

$$D_{i,j} f(a) = D_{j,i} f(a).$$

The function $D_{i,j} f$ is called a **second-order (mixed) partial derivative** of f. Higher-order (mixed) partial derivatives are defined in the obvious way. Clearly Theorem 2-5 can be used to prove the equality of higher-order mixed partial derivatives under appropriate conditions. The order of $i_1, \ldots, i_k$ is completely immaterial in $D_{i_1, \ldots, i_k} f$ if f has continuous partial derivatives of all orders. A function with this property is called a C^∞ function. In later chapters it will frequently be convenient to restrict our attention to C^∞ functions.

Partial derivatives will be used in the next section to find derivatives. They also have another important use—finding maxima and minima of functions.

2-6 Theorem. *Let $A \subset \mathbf{R}^n$. If the maximum (or minimum) of f: $A \to \mathbf{R}$ occurs at a point a in the interior of A and $D_i f(a)$ exists, then $D_i f(a) = 0$.*

Proof. Let $g_i(x) = f(a^1, \ldots, x, \ldots, a^n)$. Clearly g_i has a maximum (or minimum) at a^i, and g_i is defined in an open interval containing a^i. Hence $0 = g_i'(a^i) = D_i f(a)$. ∎

The reader is reminded that the converse of Theorem 2-6 is false even if $n = 1$ (if f: $\mathbf{R} \to \mathbf{R}$ is defined by $f(x) = x^3$, then $f'(0) = 0$, but 0 is not even a local maximum or minimum). If $n > 1$, the converse of Theorem 2-6 may fail to be true in a rather spectacular way. Suppose, for example, that f: $\mathbf{R}^2 \to \mathbf{R}$ is defined by $f(x,y) = x^2 - y^2$ (Figure 2-2). Then $D_1 f(0,0) = 0$ because g_1 has a minimum at 0, while $D_2 f(0,0) = 0$ because g_2 has a maximum at 0. Clearly (0,0) is neither a relative maximum nor a relative minimum.

FIGURE 2-2

If Theorem 2-6 is used to find the maximum or minimum of f on A, the values of f at boundary points must be examined separately—a formidable task, since the boundary of A may be all of A! Problem 2-27 indicates one way of doing this, and Problem 5-16 states a superior method which can often be used.

Problems. **2-17.** Find the partial derivatives of the following functions:

(a) $f(x,y,z) = x^y$.
(b) $f(x,y,z) = z$.
(c) $f(x,y) = \sin(x \sin y)$.
(d) $f(x,y,z) = \sin(x \sin(y \sin z))$.
(e) $f(x,y,z) = x^{y^z}$.
(f) $f(x,y,z) = x^{y+z}$.
(g) $f(x,y,z) = (x + y)^z$.
(h) $f(x,y) = \sin(xy)$.
(i) $f(x,y) = \{\sin(xy)\}^{\cos x}$.

2-18. Find the partial derivatives of the following functions (where g: $\mathbf{R} \to \mathbf{R}$ is continuous):

(a) $f(x,y) = \int_a^{x+y} g$.
(b) $f(x,y) = \int_y^x g$.
(c) $f(x,y) = \int_a^{xy} g$.
(d) $f(x,y) = \int_a^{\left(\int_b^y g\right)} g$.

2-19. If $f(x,y) = x^{xxxy} + (\log x)(\arctan(\arctan(\arctan(\sin(\cos xy) - \log(x + y)))))$ find $D_2 f(1,y)$. *Hint:* There is an easy way to do this.

2-20. Find the partial derivatives of f in terms of the derivatives of g and h if

(a) $f(x,y) = g(x)h(y)$.
(b) $f(x,y) = g(x)^{h(y)}$.
(c) $f(x,y) = g(x)$.
(d) $f(x,y) = g(y)$.
(e) $f(x,y) = g(x + y)$.

2-21.* Let g_1, g_2: $\mathbf{R}^2 \to \mathbf{R}$ be continuous. Define f: $\mathbf{R}^2 \to \mathbf{R}$ by

$$f(x,y) = \int_0^x g_1(t,0)\,dt + \int_0^y g_2(x,t)\,dt.$$

(a) Show that $D_2 f(x,y) = g_2(x,y)$.
(b) How should f be defined so that $D_1 f(x,y) = g_1(x,y)$?
(c) Find a function f: $\mathbf{R}^2 \to \mathbf{R}$ such that $D_1 f(x,y) = x$ and $D_2 f(x,y) = y$. Find one such that $D_1 f(x,y) = y$ and $D_2 f(x,y) = x$.

2-22.* If f: $\mathbf{R}^2 \to \mathbf{R}$ and $D_2f = 0$, show that f is independent of the second variable. If $D_1f = D_2f = 0$, show that f is constant.

2-23.* Let $A = \{(x,y) \in \mathbf{R}^2: x < 0, \text{ or } x \geq 0 \text{ and } y \neq 0\}$.

(a) If f: $A \to \mathbf{R}$ and $D_1f = D_2f = 0$, show that f is constant. *Hint:* Note that any two points in A can be connected by a sequence of lines each parallel to one of the axes.

(b) Find a function f: $A \to \mathbf{R}$ such that $D_2f = 0$ but f is not independent of the second variable.

2-24. Define f: $\mathbf{R}^2 \to \mathbf{R}$ by

$$f(x,y) = \begin{cases} xy\dfrac{x^2 - y^2}{x^2 + y^2} & (x,y) \neq 0, \\ 0 & (x,y) = 0. \end{cases}$$

(a) Show that $D_2f(x,0) = x$ for all x and $D_1f(0,y) = -y$ for all y.

(b) Show that $D_{1,2}f(0,0) \neq D_{2,1}f(0,0)$.

2-25.* Define f: $\mathbf{R} \to \mathbf{R}$ by

$$f(x) = \begin{cases} e^{-x^{-2}} & x \neq 0, \\ 0 & x = 0. \end{cases}$$

Show that f is a C^∞ function, and $f^{(i)}(0) = 0$ for all i. *Hint:* The limit $f'(0) = \lim\limits_{h\to 0} \dfrac{e^{-h^{-2}}}{h} = \lim\limits_{h\to 0} \dfrac{1/h}{e^{h^{-2}}}$ can be evaluated by L'Hospital's rule. It is easy enough to find $f'(x)$ for $x \neq 0$, and $f''(0) = \lim\limits_{h\to 0} f'(h)/h$ can then be found by L'Hospital's rule.

2-26.* Let $$f(x) = \begin{cases} e^{-(x-1)^{-2}} \cdot e^{-(x+1)^{-2}} & x \in (-1,1), \\ 0 & x \notin (-1,1). \end{cases}$$

(a) Show that f: $\mathbf{R} \to \mathbf{R}$ is a C^∞ function which is positive on $(-1,1)$ and 0 elsewhere.

(b) Show that there is a C^∞ function g: $\mathbf{R} \to [0,1]$ such that $g(x) = 0$ for $x \leq 0$ and $g(x) = 1$ for $x \geq \varepsilon$. *Hint:* If f is a C^∞ function which is positive on $(0,\varepsilon)$ and 0 elsewhere, let $g(x) = \int_0^x f / \int_0^\varepsilon f$.

(c) If $a \in \mathbf{R}^n$, define g: $\mathbf{R}^n \to \mathbf{R}$ by

$$g(x) = f([x^1 - a^1]/\varepsilon) \cdot \ldots \cdot f([x^n - a^n]/\varepsilon).$$

Show that g is a C^∞ function which is positive on

$$(a^1 - \varepsilon, a^1 + \varepsilon) \times \cdots \times (a^n - \varepsilon, a^n + \varepsilon)$$

and zero elsewhere.

(d) If $A \subset \mathbf{R}^n$ is open and $C \subset A$ is compact, show that there is a non-negative C^∞ function f: $A \to \mathbf{R}$ such that $f(x) > 0$ for $x \in C$ and $f = 0$ outside of some closed set contained in A.

(e) Show that we can choose such an f so that f: $A \to [0,1]$ and $f(x) = 1$ for $x \in C$. *Hint:* If the function f of (d) satisfies $f(x) \geq \varepsilon$ for $x \in C$, consider $g \circ f$, where g is the function of (b).

2-27. Define g, h: $\{x \in \mathbf{R}^2: |x| \leq 1\} \to \mathbf{R}^3$ by

$$g(x,y) = (x,y, \sqrt{1 - x^2 - y^2}),$$
$$h(x,y) = (x,y, -\sqrt{1 - x^2 - y^2}).$$

Show that the maximum of f on $\{x \in \mathbf{R}^3: |x| = 1\}$ is either the maximum of $f \circ g$ or the maximum of $f \circ h$ on $\{x \in \mathbf{R}^2: |x| \leq 1\}$.

DERIVATIVES

The reader who has compared Problems 2-10 and 2-17 has probably already guessed the following.

2-7 Theorem. *If f: $\mathbf{R}^n \to \mathbf{R}^m$ is differentiable at a, then $D_j f^i(a)$ exists for $1 \leq i \leq m$, $1 \leq j \leq n$ and $f'(a)$ is the $m \times n$ matrix $(D_j f^i(a))$.*

Proof. Suppose first that $m = 1$, so that f: $\mathbf{R}^n \to \mathbf{R}$. Define h: $\mathbf{R} \to \mathbf{R}^n$ by $h(x) = (a^1, \ldots, x, \ldots, a^n)$, with x in the jth place. Then $D_j f(a) = (f \circ h)'(a^j)$. Hence, by Theorem 2-2,

$$(f \circ h)'(a^j) = f'(a) \cdot h'(a^j)$$
$$= f'(a) \cdot \begin{pmatrix} 0 \\ \cdot \\ \cdot \\ \cdot \\ 1 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{pmatrix} \leftarrow j\text{th place.}$$

Since $(f \circ h)'(a^j)$ has the single entry $D_j f(a)$, this shows that $D_j f(a)$ exists and is the jth entry of the $1 \times n$ matrix $f'(a)$.

The theorem now follows for arbitrary m since, by Theorem 2-3, each f^i is differentiable and the ith row of $f'(a)$ is $(f^i)'(a)$. ∎

There are several examples in the problems to show that the converse of Theorem 2-7 is false. It is true, however, if one hypothesis is added.

2-8 Theorem. *If f: $\mathbf{R}^n \to \mathbf{R}^m$, then $Df(a)$ exists if all $D_j f^i(x)$ exist in an open set containing a and if each function $D_j f^i$ is continuous at a.*
(Such a function f is called **continuously differentiable** at a.)

Proof. As in the proof of Theorem 2-7, it suffices to consider the case $m = 1$, so that f: $\mathbf{R}^n \to \mathbf{R}$. Then

$$\begin{aligned} f(a+h) - f(a) = {} & f(a^1 + h^1, a^2, \ldots, a^n) - f(a^1, \ldots, a^n) \\ & + f(a^1 + h^1, a^2 + h^2, a^3, \ldots, a^n) \\ & \qquad - f(a^1 + h^1, a^2, \ldots, a^n) \\ & + \cdots \\ & + f(a^1 + h^1, \ldots, a^n + h^n) \\ & \qquad - f(a^1 + h^1, \ldots, a^{n-1} + h^{n-1}, a^n). \end{aligned}$$

Recall that $D_1 f$ is the derivative of the function g defined by $g(x) = f(x, a^2, \ldots, a^n)$. Applying the mean-value theorem to g we obtain

$$\begin{aligned} f(a^1 + h^1, a^2, \ldots, a^n) - f(a^1, \ldots, a^n) \\ = h^1 \cdot D_1 f(b_1, a^2, \ldots, a^n) \end{aligned}$$

for some b_1 between a^1 and $a^1 + h^1$. Similarly the ith term in the sum equals

$$h^i \cdot D_i f(a^1 + h^1, \ldots, a^{i-1} + h^{i-1}, b_i, \ldots, a^n) = h^i D_i f(c_i),$$

for some c_i. Then

$$\begin{aligned} \lim_{h \to 0} \frac{\left| f(a+h) - f(a) - \sum_{i=1}^{n} D_i f(a) \cdot h^i \right|}{|h|} \\ = \lim_{h \to 0} \frac{\left| \sum_{i=1}^{n} [D_i f(c_i) - D_i f(a)] \cdot h^i \right|}{|h|} \\ \le \lim_{h \to 0} \sum_{i=1}^{n} |D_i f(c_i) - D_i f(a)| \cdot \frac{|h^i|}{|h|} \\ \le \lim_{h \to 0} \sum_{i=1}^{n} |D_i f(c_i) - D_i f(a)| \\ = 0, \end{aligned}$$

since $D_i f$ is continuous at a. ∎

Although the chain rule was used in the proof of Theorem 2-7, it could easily have been eliminated. With Theorem 2-8 to provide differentiable functions, and Theorem 2-7 to provide their derivatives, the chain rule may therefore seem almost superfluous. However, it has an extremely important corollary concerning partial derivatives.

2-9 Theorem. *Let $g_1, \ldots, g_m$: $\mathbf{R}^n \to \mathbf{R}$ be continuously differentiable at a, and let f: $\mathbf{R}^m \to \mathbf{R}$ be differentiable at $(g_1(a), \ldots, g_m(a))$. Define the function F: $\mathbf{R}^n \to \mathbf{R}$ by $F(x) = f(g_1(x), \ldots, g_m(x))$. Then*

$$D_iF(a) = \sum_{j=1}^{m} D_jf(g_1(a), \ldots, g_m(a)) \cdot D_ig_j(a).$$

Proof. The function F is just the composition $f \circ g$, where $g = (g_1, \ldots, g_m)$. Since g_i is continuously differentiable at a, it follows from Theorem 2-8 that g is differentiable at a. Hence by Theorem 2-2,

$$F'(a) = f'(g(a)) \cdot g'(a) =$$

$$(D_1f(g(a)), \ldots, D_mf(g(a))) \cdot \begin{pmatrix} D_1g_1(a), & \cdots & ,D_ng_1(a) \\ \vdots & & \vdots \\ D_1g_m(a), & \cdots & ,D_ng_m(a) \end{pmatrix}$$

But $D_iF(a)$ is the ith entry of the left side of this equation, while $\sum_{j=1}^{m} D_jf(g_1(a), \ldots, g_m(a)) \cdot D_ig_j(a)$ is the ith entry of the right side. ∎

Theorem 2-9 is often called the *chain rule*, but is weaker than Theorem 2-2 since g could be differentiable without g_i being continuously differentiable (see Problem 2-32). Most computations requiring Theorem 2-9 are fairly straightforward. A slight subtlety is required for the function F: $\mathbf{R}^2 \to \mathbf{R}$ defined by

$$F(x,y) = f(g(x,y),h(x),k(y))$$

where h,k: $\mathbf{R} \to \mathbf{R}$. In order to apply Theorem 2-9 define $\bar{h},\bar{k}$: $\mathbf{R}^2 \to \mathbf{R}$ by

$$\bar{h}(x,y) = h(x) \qquad \bar{k}(x,y) = k(y).$$

Then

$$\begin{aligned} D_1\bar{h}(x,y) &= h'(x) & D_2\bar{h}(x,y) &= 0, \\ D_1\bar{k}(x,y) &= 0 & D_2\bar{k}(x,y) &= k'(y), \end{aligned}$$

and we can write

$$F(x,y) = f(g(x,y),\bar{h}(x,y),\bar{k}(x,y)).$$

Letting $a = (g(x,y),h(x),k(y))$, we obtain

$$\begin{aligned} D_1F(x,y) &= D_1f(a) \cdot D_1g(x,y) + D_2f(a) \cdot h'(x), \\ D_2F(x,y) &= D_1f(a) \cdot D_2g(x,y) + D_3f(a) \cdot k'(y). \end{aligned}$$

It should, of course, be unnecessary for you to actually write down the functions $\bar{h}$ and $\bar{k}$.

Problems. **2-28.** Find expressions for the partial derivatives of the following functions:

(a) $F(x,y) = f(g(x)k(y),\ g(x) + h(y))$.
(b) $F(x,y,z) = f(g(x + y),\ h(y + z))$.
(c) $F(x,y,z) = f(x^y,y^z,z^x)$.
(d) $F(x,y) = f(x,g(x),h(x,y))$.

2-29. Let f: $\mathbf{R}^n \to \mathbf{R}$. For $x \in \mathbf{R}^n$, the limit

$$\lim_{t\to 0} \frac{f(a + tx) - f(a)}{t},$$

if it exists, is denoted $D_xf(a)$, and called the **directional derivative** of f at a, in the direction x.

(a) Show that $D_{e_i}f(a) = D_if(a)$.
(b) Show that $D_{tx}f(a) = tD_xf(a)$.
(c) If f is differentiable at a, show that $D_xf(a) = Df(a)(x)$ and therefore $D_{x+y}f(a) = D_xf(a) + D_yf(a)$.

2-30. Let f be defined as in Problem 2-4. Show that $D_xf(0,0)$ exists for all x, but if $g \neq 0$, then $D_{x+y}f(0,0) = D_xf(0,0) + D_yf(0,0)$ is not true for all x and y.

2-31. Let f: $\mathbf{R}^2 \to \mathbf{R}$ be defined as in Problem 1-26. Show that $D_xf(0,0)$ exists for all x, although f is not even continuous at $(0,0)$.

2-32. (a) Let f: $\mathbf{R} \to \mathbf{R}$ be defined by

$$f(x) = \begin{cases} x^2 \sin \dfrac{1}{x} & x \neq 0, \\ 0 & x = 0. \end{cases}$$

Show that f is differentiable at 0 but f' is not continuous at 0.

(b) Let $f\colon \mathbf{R}^2 \to \mathbf{R}$ be defined by

$$f(x,y) = \begin{cases} (x^2 + y^2)\sin \dfrac{1}{\sqrt{x^2 + y^2}} & (x,y) \neq 0, \\ 0 & (x,y) = 0. \end{cases}$$

Show that f is differentiable at (0,0) but $D_i f$ is not continuous at (0,0).

2-33. Show that the continuity of $D_1 f^j$ at a may be eliminated from the hypothesis of Theorem 2-8.

2-34. A function $f\colon \mathbf{R}^n \to \mathbf{R}$ is **homogeneous** of degree m if $f(tx) = t^m f(x)$ for all x. If f is also differentiable, show that

$$\sum_{i=1}^{n} x^i D_i f(x) = mf(x).$$

Hint: If $g(t) = f(tx)$, find $g'(1)$.

2-35. If $f\colon \mathbf{R}^n \to \mathbf{R}$ is differentiable and $f(0) = 0$, prove that there exist $g_i\colon \mathbf{R}^n \to \mathbf{R}$ such that

$$f(x) = \sum_{i=1}^{n} x^i g_i(x).$$

Hint: If $h_x(t) = f(tx)$, then $f(x) = \int_0^1 h_x'(t)dt$.

INVERSE FUNCTIONS

Suppose that $f\colon \mathbf{R} \to \mathbf{R}$ is continuously differentiable in an open set containing a and $f'(a) \neq 0$. If $f'(a) > 0$, there is an open interval V containing a such that $f'(x) > 0$ for $x \in V$, and a similar statement holds if $f'(a) < 0$. Thus f is increasing (or decreasing) on V, and is therefore 1-1 with an inverse function f^{-1} defined on some open interval W containing $f(a)$. Moreover it is not hard to show that f^{-1} is differentiable, and for $y \in W$ that

$$(f^{-1})'(y) = \frac{1}{f'(f^{-1}(y))}.$$

An analogous discussion in higher dimensions is much more involved, but the result (Theorem 2-11) is very important. We begin with a simple lemma.

2-10 Lemma. *Let $A \subset \mathbf{R}^n$ be a rectangle and let $f\colon A \to \mathbf{R}^n$ be continuously differentiable. If there is a number M such that $|D_j f^i(x)| \leq M$ for all x in the interior of A, then*

$$|f(x) - f(y)| \leq n^2 M |x - y|$$

for all $x,y \in A$.

Proof. We have

$$f^i(y) - f^i(x) = \sum_{j=1}^{n} [f^i(y^1, \ldots, y^j, x^{j+1}, \ldots, x^n) - f^i(y^1, \ldots, y^{j-1}, x^j, \ldots, x^n)].$$

Applying the mean-value theorem we obtain

$$f^i(y^1, \ldots, y^j, x^{j+1}, \ldots, x^n) - f^i(y^1, \ldots, y^{j-1}, x^j, \ldots, x^n) = (y^j - x^j) \cdot D_j f^i(z_{ij})$$

for some z_{ij}. The expression on the right has absolute value less than or equal to $M \cdot |y^j - x^j|$. Thus

$$|f^i(y) - f^i(x)| \leq \sum_{j=1}^{n} |y^j - x^j| \cdot M \leq nM|y - x|$$

since each $|y^j - x^j| \leq |y - x|$. Finally

$$|f(y) - f(x)| \leq \sum_{i=1}^{n} |f^i(y) - f^i(x)| \leq n^2 M \cdot |y - x|. \quad \blacksquare$$

2-11 Theorem (Inverse Function Theorem). *Suppose that $f\colon \mathbf{R}^n \to \mathbf{R}^n$ is continuously differentiable in an open set containing a, and $\det f'(a) \neq 0$. Then there is an open set V containing a and an open set W containing $f(a)$ such that $f\colon V \to W$ has a continuous inverse $f^{-1}\colon W \to V$ which is differentiable and for all $y \in W$ satisfies*

$$(f^{-1})'(y) = [f'(f^{-1}(y))]^{-1}.$$

Proof. Let λ be the linear transformation $Df(a)$. Then λ is non-singular, since $\det f'(a) \neq 0$. Now $D(\lambda^{-1} \circ f)(a) = D(\lambda^{-1})(f(a)) \circ Df(a) = \lambda^{-1} \circ Df(a)$ is the identity linear

transformation. If the theorem is true for $\lambda^{-1} \circ f$, it is clearly true for f. Therefore we may assume at the outset that λ is the identity. Thus whenever $f(a + h) = f(a)$, we have

$$\frac{|f(a + h) - f(a) - \lambda(h)|}{|h|} = \frac{|h|}{|h|} = 1.$$

But

$$\lim_{h \to 0} \frac{|f(a + h) - f(a) - \lambda(h)|}{|h|} = 0.$$

This means that we cannot have $f(x) = f(a)$ for x arbitrarily close to, but unequal to, a. Therefore there is a closed rectangle U containing a in its interior such that

1. $f(x) \neq f(a)$ if $x \in U$ and $x \neq a$.

Since f is continuously differentiable in an open set containing a, we can also assume that

2. $\det f'(x) \neq 0$ for $x \in U$.
3. $|D_j f^i(x) - D_j f^i(a)| < 1/2n^2$ for all i, j, and $x \in U$.

Note that (3) and Lemma 2-10 applied to $g(x) = f(x) - x$ imply for $x_1, x_2 \in U$ that

$$|f(x_1) - x_1 - (f(x_2) - x_2)| \leq \tfrac{1}{2}|x_1 - x_2|.$$

Since

$$\begin{aligned} |x_1 - x_2| - |f(x_1) - f(x_2)| &\leq |f(x_1) - x_1 - (f(x_2) - x_2)| \\ &\leq \tfrac{1}{2}|x_1 - x_2|, \end{aligned}$$

we obtain

4. $|x_1 - x_2| \leq 2|f(x_1) - f(x_2)|$ for $x_1, x_2 \in U$.

Now f(boundary U) is a compact set which, by (1), does not contain $f(a)$ (Figure 2-3). Therefore there is a number $d > 0$ such that $|f(a) - f(x)| \geq d$ for $x \in$ boundary U. Let $W = \{y : |y - f(a)| < d/2\}$. If $y \in W$ and $x \in$ boundary U, then

5. $|y - f(a)| < |y - f(x)|$.

We will show that for any $y \in W$ there is a unique x in interior U such that $f(x) = y$. To prove this consider the

FIGURE 2-3

function $g\colon U \to \mathbf{R}$ defined by

$$g(x) = |y - f(x)|^2 = \sum_{i=1}^{n} (y^i - f^i(x))^2.$$

This function is continuous and therefore has a minimum on U. If $x \in$ boundary U, then, by (5), we have $g(a) < g(x)$. Therefore the minimum of g does *not* occur on the boundary of U. By Theorem 2-6 there is a point $x \in$ interior U such that $D_j g(x) = 0$ for all j, that is

$$\sum_{i=1}^{n} 2(y^i - f^i(x)) \cdot D_j f^i(x) = 0 \qquad \text{for all } j.$$

By (2) the matrix $(D_j f^i(x))$ has non-zero determinant. Therefore we must have $y^i - f^i(x) = 0$ for all i, that is $y = f(x)$. This proves the existence of x. Uniqueness follows immediately from (4).

If $V = (\text{interior } U) \cap f^{-1}(W)$, we have shown that the function $f\colon V \to W$ has an inverse $f^{-1}\colon W \to V$. We can rewrite (4) as

6. $|f^{-1}(y_1) - f^{-1}(y_2)| \leq 2|y_1 - y_2| \qquad \text{for } y_1, y_2 \in W.$

This shows that f^{-1} is continuous.

Only the proof that f^{-1} is differentiable remains. Let $\mu = Df(x)$. We will show that f^{-1} is differentiable at $y = f(x)$ with derivative μ^{-1}. As in the proof of Theorem 2-2, for $x_1 \in V$, we have

$$f(x_1) = f(x) + \mu(x_1 - x) + \varphi(x_1 - x),$$

where

$$\lim_{x_1 \to x} \frac{|\varphi(x_1 - x)|}{|x_1 - x|} = 0.$$

Therefore

$$\mu^{-1}(f(x_1) - f(x)) = x_1 - x + \mu^{-1}(\varphi(x_1 - x)).$$

Since every $y_1 \in W$ is of the form $f(x_1)$ for some $x_1 \in V$, this can be written

$$f^{-1}(y_1) = f^{-1}(y) + \mu^{-1}(y_1 - y) - \mu^{-1}(\varphi(f^{-1}(y_1) - f^{-1}(y))),$$

and it therefore suffices to show that

$$\lim_{y_1 \to y} \frac{|\mu^{-1}(\varphi(f^{-1}(y_1) - f^{-1}(y)))|}{|y_1 - y|} = 0.$$

Therefore (Problem 1-10) it suffices to show that

$$\lim_{y_1 \to y} \frac{|\varphi(f^{-1}(y_1) - f^{-1}(y))|}{|y_1 - y|} = 0.$$

Now

$$\begin{aligned}\frac{|\varphi(f^{-1}(y_1) - f^{-1}(y))|}{|y_1 - y|} \\ = \frac{|\varphi(f^{-1}(y_1) - f^{-1}(y))|}{|f^{-1}(y_1) - f^{-1}(y)|} \cdot \frac{|f^{-1}(y_1) - f^{-1}(y)|}{|y_1 - y|}.\end{aligned}$$

Since f^{-1} is continuous, $f^{-1}(y_1) \to f^{-1}(y)$ as $y_1 \to y$. Therefore the first factor approaches 0. Since, by (6), the second factor is less than 2, the product also approaches 0. ∎

It should be noted that an inverse function f^{-1} may exist even if $\det f'(a) = 0$. For example, if $f\colon \mathbf{R} \to \mathbf{R}$ is defined by $f(x) = x^3$, then $f'(0) = 0$ but f has the inverse function $f^{-1}(x) = \sqrt[3]{x}$. One thing is certain however: if $\det f'(a) = 0$, then f^{-1} cannot be differentiable at $f(a)$. To prove this note that $f \circ f^{-1}(x) = x$. If f^{-1} were differentiable at $f(a)$, the chain rule would give $f'(a) \cdot (f^{-1})'(f(a)) = I$, and consequently $\det f'(a) \cdot \det(f^{-1})'(f(a)) = 1$, contradicting $\det f'(a) = 0$.

Problems. **2-36.*** Let $A \subset \mathbf{R}^n$ be an open set and $f\colon A \to \mathbf{R}^n$ a continuously differentiable 1-1 function such that $\det f'(x) \neq 0$ for all x. Show that $f(A)$ is an open set and $f^{-1}\colon f(A) \to A$ is differentiable. Show also that $f(B)$ is open for any open set $B \subset A$.

2-37. (a) Let $f\colon \mathbf{R}^2 \to \mathbf{R}$ be a continuously differentiable function. Show that f is *not* 1-1. *Hint:* If, for example, $D_1 f(x,y) \neq 0$ for all (x,y) in some open set A, consider $g\colon A \to \mathbf{R}^2$ defined by $g(x,y) = (f(x,y),y)$.

(b) Generalize this result to the case of a continuously differentiable function $f\colon \mathbf{R}^n \to \mathbf{R}^m$ with $m < n$.

2-38. (a) If $f\colon \mathbf{R} \to \mathbf{R}$ satisfies $f'(a) \neq 0$ for all $a \in \mathbf{R}$, show that f is 1-1 (on all of $\mathbf{R}$).

(b) Define f: $\mathbf{R}^2 \to \mathbf{R}^2$ by $f(x,y) = (e^x \cos y,\ e^x \sin y)$. Show that $\det f'(x,y) \neq 0$ for all (x,y) but f is not 1-1.

2-39. Use the function f: $\mathbf{R} \to \mathbf{R}$ defined by

$$f(x) = \begin{cases} \dfrac{x}{2} + x^2 \sin \dfrac{1}{x} & x \neq 0, \\ 0 & x = 0, \end{cases}$$

to show that continuity of the derivative cannot be eliminated from the hypothesis of Theorem 2-11.

IMPLICIT FUNCTIONS

Consider the function f: $\mathbf{R}^2 \to \mathbf{R}$ defined by $f(x,y) = x^2 + y^2 - 1$. If we choose (a,b) with $f(a,b) = 0$ and $a \neq 1, -1$, there are (Figure 2-4) open intervals A containing a and B containing b with the following property: if $x \in A$, there is a unique $y \in B$ with $f(x,y) = 0$. We can therefore define

FIGURE 2-4

a function $g: A \to \mathbf{R}$ by the condition $g(x) \in B$ and $f(x,g(x)) = 0$ (if $b > 0$, as indicated in Figure 2-4, then $g(x) = \sqrt{1 - x^2}$). For the function f we are considering there is another number b_1 such that $f(a,b_1) = 0$. There will also be an interval B_1 containing b_1 such that, when $x \in A$, we have $f(x,g_1(x)) = 0$ for a unique $g_1(x) \in B_1$ (here $g_1(x) = -\sqrt{1 - x^2}$). Both g and g_1 are differentiable. These functions are said to be defined **implicitly** by the equation $f(x,y) = 0$.

If we choose $a = 1$ or -1 it is impossible to find any such function g defined in an open interval containing a. We would like a simple criterion for deciding when, in general, such a function can be found. More generally we may ask the following: If $f: \mathbf{R}^n \times \mathbf{R} \to \mathbf{R}$ and $f(a^1, \ldots, a^n, b) = 0$, when can we find, for each $(x^1, \ldots, x^n)$ near $(a^1, \ldots, a^n)$, a unique y near b such that $f(x^1, \ldots, x^n, y) = 0$? Even more generally, we can ask about the possibility of solving m equations, depending upon parameters $x^1, \ldots, x^n$, in m unknowns: If

$$f_i: \mathbf{R}^n \times \mathbf{R}^m \to \mathbf{R} \qquad i = 1, \ldots, m$$

and

$$f_i(a^1, \ldots, a^n, b^1, \ldots, b^m) = 0 \qquad i = 1, \ldots, m,$$

when can we find, for each $(x^1, \ldots, x^n)$ near $(a^1, \ldots, a^n)$ a unique $(y^1, \ldots, y^m)$ near $(b^1, \ldots, b^m)$ which satisfies $f_i(x^1, \ldots, x^n, y^1, \ldots, y^m) = 0$? The answer is provided by

2-12 Theorem (Implicit Function Theorem). *Suppose $f: \mathbf{R}^n \times \mathbf{R}^m \to \mathbf{R}^m$ is continuously differentiable in an open set containing (a,b) and $f(a,b) = 0$. Let M be the $m \times m$ matrix*

$$(D_{n+j} f^i(a,b)) \qquad 1 \le i, j \le m.$$

If $\det M \neq 0$, there is an open set $A \subset \mathbf{R}^n$ containing a and an open set $B \subset \mathbf{R}^m$ containing b, with the following property: for each $x \in A$ there is a unique $g(x) \in B$ such that $f(x,g(x)) = 0$. The function g is differentiable.

Proof. Define $F\colon \mathbf{R}^n \times \mathbf{R}^m \to \mathbf{R}^n \times \mathbf{R}^m$ by $F(x,y) = (x,f(x,y))$. Then $\det F'(a,b) = \det M \neq 0$. By Theorem 2-11 there is an open set $W \subset \mathbf{R}^n \times \mathbf{R}^m$ containing $F(a,b) = (a,0)$ and an open set in $\mathbf{R}^n \times \mathbf{R}^m$ containing (a,b), which we may take to be of the form $A \times B$, such that $F\colon A \times B \to W$ has a differentiable inverse $h\colon W \to A \times B$. Clearly h is of the form $h(x,y) = (x,k(x,y))$ for some differentiable function k (since F is of this form). Let $\pi\colon \mathbf{R}^n \times \mathbf{R}^m \to \mathbf{R}^m$ be defined by $\pi(x,y) = y$; then $\pi \circ F = f$. Therefore

$$\begin{aligned} f(x,k(x,y)) &= f \circ h(x,y) = (\pi \circ F) \circ h(x,y) \\ &= \pi \circ (F \circ h)(x,y) = \pi(x,y) = y. \end{aligned}$$

Thus $f(x,k(x,0)) = 0$; in other words we can define $g(x) = k(x,0)$. ∎

Since the function g is known to be differentiable, it is easy to find its derivative. In fact, since $f^i(x,g(x)) = 0$, taking D_j of both sides gives

$$0 = D_j f^i(x,g(x)) + \sum_{\alpha=1}^{m} D_{n+\alpha} f^i(x,g(x)) \cdot D_j g^\alpha(x) \qquad i,j = 1, \ldots ,m.$$

Since $\det M \neq 0$, these equations can be solved for $D_j g^\alpha(x)$. The answer will depend on the various $D_j f^i(x,g(x))$, and therefore on $g(x)$. This is unavoidable, since the function g is not unique. Reconsidering the function $f\colon \mathbf{R}^2 \to \mathbf{R}$ defined by $f(x,y) = x^2 + y^2 - 1$, we note that two possible functions satisfying $f(x,g(x)) = 0$ are $g(x) = \sqrt{1 - x^2}$ and $g(x) = -\sqrt{1 - x^2}$. Differentiating $f(x,g(x)) = 0$ gives

$$D_1 f(x,g(x)) + D_2 f(x,g(x)) \cdot g'(x) = 0,$$

or

$$\begin{aligned} 2x + 2g(x) \cdot g'(x) &= 0, \\ g'(x) &= -x/g(x), \end{aligned}$$

which is indeed the case for either $g(x) = \sqrt{1 - x^2}$ or $g(x) = -\sqrt{1 - x^2}$.

A generalization of the argument for Theorem 2-12 can be given, which will be important in Chapter 5.

2-13 Theorem. *Let f: $\mathbf{R}^n \to \mathbf{R}^p$ be continuously differentiable in an open set containing a, where $p \leq n$. If $f(a) = 0$ and the $p \times n$ matrix $(D_j f^i(a))$ has rank p, then there is an open set $A \subset \mathbf{R}^n$ containing a and a differentiable function h: $A \to \mathbf{R}^n$ with differentiable inverse such that*

$$f \circ h(x^1, \ldots, x^n) = (x^{n-p+1}, \ldots, x^n).$$

Proof. We can consider f as a function f: $\mathbf{R}^{n-p} \times \mathbf{R}^p \to \mathbf{R}^p$. If det $M \neq 0$, then M is the $p \times p$ matrix $(D_{n-p+j} f^i(a))$, $1 \leq i, j \leq p$, then we are precisely in the situation considered in the proof of Theorem 2-12, and as we showed in that proof, there is h such that $f \circ h(x^1, \ldots, x^n) = (x^{n-p+1}, \ldots, x^n)$.

In general, since $(D_j f^i(a))$ has rank p, there will be $j_1 < \cdots < j_p$ such that the matrix $(D_j f^i(a))$ $1 \leq i \leq p$, $j = j_1, \ldots, j_p$ has non-zero determinant. If g: $\mathbf{R}^n \to \mathbf{R}^n$ permutes the x^j so that $g(x^1, \ldots, x^n) = (\ldots, x^{j_1}, \ldots, x^{j_p})$, then $f \circ g$ is a function of the type already considered, so $((f \circ g) \circ k)(x^1, \ldots, x^n) = (x^{n-p+1}, \ldots, x^n)$ for some k. Let $h = g \circ k$. ∎

Problems. **2-40.** Use the implicit function theorem to re-do Problem 2-15(c).

2-41. Let f: $\mathbf{R} \times \mathbf{R} \to \mathbf{R}$ be differentiable. For each $x \in \mathbf{R}$ define g_x: $\mathbf{R} \to \mathbf{R}$ by $g_x(y) = f(x,y)$. Suppose that for each x there is a unique y with $g_x'(y) = 0$; let $c(x)$ be this y.

(a) If $D_{2,2}f(x,y) \neq 0$ for all (x,y), show that c is differentiable and

$$c'(x) = -\frac{D_{2,1}f(x,c(x))}{D_{2,2}f(x,c(x))}.$$

Hint: $g_x'(y) = 0$ can be written $D_2 f(x,y) = 0$.

(b) Show that if $c'(x) = 0$, then for some y we have

$$D_{2,1}f(x,y) = 0,$$
$$D_2 f(x,y) = 0.$$

(c) Let $f(x,y) = x(y \log y - y) - y \log x$. Find

$$\max_{\frac{1}{2} \leq x \leq 2} \left(\min_{\frac{1}{2} \leq y \leq 1} f(x,y) \right).$$

NOTATION

This section is a brief and not entirely unprejudiced discussion of classical notation connected with partial derivatives.

The partial derivative $D_1 f(x,y,z)$ is denoted, among devotees of classical notation, by

$$\frac{\partial f(x,y,z)}{\partial x} \quad \text{or} \quad \frac{\partial f}{\partial x} \quad \text{or} \quad \frac{\partial f}{\partial x}(x,y,z) \quad \text{or} \quad \frac{\partial}{\partial x} f(x,y,z)$$

or any other convenient similar symbol. This notation forces one to write

$$\frac{\partial f}{\partial u}(u,v,w)$$

for $D_1 f(u,v,w)$, although the symbol

$$\left.\frac{\partial f(x,y,z)}{\partial x}\right|_{(x,y,z)=(u,v,w)} \quad \text{or} \quad \frac{\partial f(x,y,z)}{\partial x}(u,v,w)$$

or something similar may be used (and must be used for an expression like $D_1 f(7,3,2)$). Similar notation is used for $D_2 f$ and $D_3 f$. Higher-order derivatives are denoted by symbols like

$$D_2 D_1 f(x,y,z) = \frac{\partial^2 f(x,y,z)}{\partial y\, \partial x}.$$

When $f\colon \mathbf{R} \to \mathbf{R}$, the symbol ∂ automatically reverts to d; thus

$$\frac{d \sin x}{dx}, \quad \text{not} \quad \frac{\partial \sin x}{\partial x}.$$

The mere statement of Theorem 2-2 in classical notation requires the introduction of irrelevant letters. The usual evaluation for $D_1(f \circ (g,h))$ runs as follows:

If $f(u,v)$ is a function and $u = g(x,y)$ and $v = h(x,y)$, then

$$\frac{\partial f(g(x,y),\, h(x,y))}{\partial x} = \frac{\partial f(u,v)}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial f(u,v)}{\partial v}\frac{\partial v}{\partial x}.$$

[The symbol $\partial u/\partial x$ means $\partial/\partial x\, g(x,y)$ and $\partial/\partial u\, f(u,v)$ means

$D_1 f(u,v) = D_1 f(g(x,y), h(x,y))$.] This equation is often written simply

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial x}.$$

Note that f means something different on the two sides of the equation!

The notation df/dx, always a little too tempting, has inspired many (usually meaningless) definitions of dx and df separately, the sole purpose of which is to make the equation

$$df = \frac{df}{dx} \cdot dx$$

work out. If $f\colon \mathbf{R}^2 \to \mathbf{R}$ then df is *defined*, classically, as

$$df = \frac{\partial f}{\partial x}\,dx + \frac{\partial f}{\partial y}\,dy$$

(whatever dx and dy mean).

Chapter 4 contains rigorous definitions which enable us to prove the above equations as theorems. It is a touchy question whether or not these modern definitions represent a real improvement over classical formalism; this the reader must decide for himself.

3

Integration

BASIC DEFINITIONS

The definition of the integral of a function $f\colon A \to \mathbf{R}$, where $A \subset \mathbf{R}^n$ is a closed rectangle, is so similar to that of the ordinary integral that a rapid treatment will be given.

Recall that a partition P of a closed interval $[a,b]$ is a sequence $t_0, \ldots, t_k$, where $a = t_0 \leq t_1 \leq \cdots \leq t_k = b$. The partition P divides the interval $[a,b]$ into k subintervals $[t_{i-1},t_i]$. A **partition** of a rectangle $[a_1,b_1] \times \cdots \times [a_n,b_n]$ is a collection $P = (P_1, \ldots, P_n)$, where each P_i is a partition of the interval $[a_i,b_i]$. Suppose, for example, that $P_1 = t_0, \ldots, t_k$ is a partition of $[a_1,b_1]$ and $P_2 = s_0, \ldots, s_l$ is a partition of $[a_2,b_2]$. Then the partition $P = (P_1,P_2)$ of $[a_1,b_1] \times [a_2,b_2]$ divides the closed rectangle $[a_1,b_1] \times [a_2,b_2]$ into $k \cdot l$ subrectangles, a typical one being $[t_{i-1},t_i] \times [s_{j-1},s_j]$. In general, if P_i divides $[a_i,b_i]$ into N_i subintervals, then $P = (P_1, \ldots, P_n)$ divides $[a_1,b_1] \times \cdots \times [a_n,b_n]$ into $N = N_1 \cdot \ldots \cdot N_n$ subrectangles. These subrectangles will be called **subrectangles of the partition** P.

Suppose now that A is a rectangle, $f\colon A \to \mathbf{R}$ is a bounded

function, and P is a partition of A. For each subrectangle S of the partition let

$$m_S(f) = \inf\{f(x) : x \in S\},$$
$$M_S(f) = \sup\{f(x) : x \in S\},$$

and let $v(S)$ be the volume of S [the **volume** of a rectangle $[a_1,b_1] \times \cdots \times [a_n,b_n]$, and also of $(a_1,b_1) \times \cdots \times (a_n,b_n)$, is defined as $(b_1 - a_1) \cdot \ldots \cdot (b_n - a_n)$]. The **lower** and **upper sums** of f for P are defined by

$$L(f,P) = \sum_S m_S(f) \cdot v(S) \quad \text{and} \quad U(f,P) = \sum_S M_S(f) \cdot v(S).$$

Clearly $L(f,P) \leq U(f,P)$, and an even stronger assertion (3-2) is true.

3-1 Lemma. *Suppose the partition P' refines P (that is, each subrectangle of P' is contained in a subrectangle of P). Then*

$$L(f,P) \leq L(f,P') \qquad \textit{and} \qquad U(f,P') \leq U(f,P).$$

Proof. Each subrectangle S of P is divided into several subrectangles $S_1, \ldots, S_\alpha$ of P', so $v(S) = v(S_1) + \cdots + v(S_\alpha)$. Now $m_S(f) \leq m_{S_i}(f)$, since the values $f(x)$ for $x \in S$ include all values $f(x)$ for $x \in S_i$ (and possibly smaller ones). Thus

$$\begin{aligned} m_S(f) \cdot v(S) &= m_S(f) \cdot v(S_1) + \cdots + m_S(f) \cdot v(S_\alpha) \\ &\leq m_{S_1}(f) \cdot v(S_1) + \cdots + m_{S_\alpha}(f) \cdot v(S_\alpha). \end{aligned}$$

The sum, for all S, of the terms on the left side is $L(f,P)$, while the sum of all the terms on the right side is $L(f,P')$. Hence $L(f,P) \leq L(f,P')$. The proof for upper sums is similar. ∎

3-2 Corollary. *If P and P' are any two partitions, then $L(f,P') \leq U(f,P)$.*

Proof. Let P'' be a partition which refines both P and P'. (For example, let $P'' = (P_1'', \ldots, P_n'')$, where P_i'' is a par-

tition of $[a_i,b_i]$ which refines both P_i and P'_i.) Then

$$L(f,P') \leq L(f,P'') \leq U(f,P'') \leq U(f,P). \quad \blacksquare$$

It follows from Corollary 3-2 that the least upper bound of all lower sums for f is less than or equal to the greatest lower bound of all upper sums for f. A function $f\colon A \to \mathbf{R}$ is called **integrable** on the rectangle A if f is bounded and $\sup\{L(f,P)\} = \inf\{U(f,P)\}$. This common number is then denoted $\int_A f$, and called the **integral** of f over A. Often, the notation $\int_A f(x^1, \ldots, x^n)dx^1 \cdots dx^n$ is used. If $f\colon [a,b] \to \mathbf{R}$, where $a \leq b$, then $\int_a^b f = \int_{[a,b]} f$. A simple but useful criterion for integrability is provided by

3-3 Theorem. *A bounded function $f\colon A \to \mathbf{R}$ is integrable if and only if for every $\varepsilon > 0$ there is a partition P of A such that $U(f,P) - L(f,P) < \varepsilon$.*

Proof. If this condition holds, it is clear that $\sup\{L(f,P)\} = \inf\{U(f,P)\}$ and f is integrable. On the other hand, if f is integrable, so that $\sup\{L(f,P)\} = \inf\{U(f,P)\}$, then for any $\varepsilon > 0$ there are partitions P and P' with $U(f,P) - L(f,P') < \varepsilon$. If P'' refines both P and P', it follows from Lemma 3-1 that $U(f,P'') - L(f,P'') \leq U(f,P) - L(f,P') < \varepsilon$. $\blacksquare$

In the following sections we will characterize the integrable functions and discover a method of computing integrals. For the present we consider two functions, one integrable and one not.

1. Let $f\colon A \to \mathbf{R}$ be a constant function, $f(x) = c$. Then for any partition P and subrectangle S we have $m_S(f) = M_S(f) = c$, so that $L(f,P) = U(f,P) = \Sigma_S c \cdot v(S) = c \cdot v(A)$. Hence $\int_A f = c \cdot v(A)$.

2. Let $f\colon [0,1] \times [0,1] \to \mathbf{R}$ be defined by

$$f(x,y) = \begin{cases} 0 & \text{if } x \text{ is rational,} \\ 1 & \text{if } x \text{ is irrational.} \end{cases}$$

If P is a partition, then every subrectangle S will contain points (x,y) with x rational, and also points (x,y) with x

irrational. Hence $m_S(f) = 0$ and $M_S(f) = 1$, so

$$L(f,P) = \sum_S 0 \cdot v(S) = 0$$

and

$$U(f,P) = \sum_S 1 \cdot v(S) = v([0,1] \times [0,1]) = 1.$$

Therefore f is not integrable.

Problems. **3-1.** Let f: $[0,1] \times [0,1] \to \mathbf{R}$ be defined by

$$f(x,y) = \begin{cases} 0 & \text{if } 0 \leq x < \frac{1}{2}, \\ 1 & \text{if } \frac{1}{2} \leq x \leq 1. \end{cases}$$

Show that f is integrable and $\int_{[0,1]\times[0,1]} f = \frac{1}{2}$.

3-2. Let f: $A \to \mathbf{R}$ be integrable and let $g = f$ except at finitely many points. Show that g is integrable and $\int_A f = \int_A g$.

3-3. Let f,g: $A \to \mathbf{R}$ be integrable.

(a) For any partition P of A and subrectangle S, show that

$$m_S(f) + m_S(g) \leq m_S(f+g) \quad \text{and} \quad M_S(f+g) \leq M_S(f) + M_S(g)$$

and therefore

$$L(f,P) + L(g,P) \leq L(f+g, P) \quad \text{and} \quad U(f+g, P) \leq U(f,P) + U(g,P).$$

(b) Show that $f + g$ is integrable and $\int_A f + g = \int_A f + \int_A g$.

(c) For any constant c, show that $\int_A cf = c\int_A f$.

3-4. Let f: $A \to \mathbf{R}$ and let P be a partition of A. Show that f is integrable if and only if for each subrectangle S the function $f|S$, which consists of f restricted to S, is integrable, and that in this case $\int_A f = \Sigma_S \int_S f|S$.

3-5. Let f,g: $A \to \mathbf{R}$ be integrable and suppose $f \leq g$. Show that $\int_A f \leq \int_A g$.

3-6. If f: $A \to \mathbf{R}$ is integrable, show that $|f|$ is integrable and $|\int_A f| \leq \int_A |f|$.

3-7. Let f: $[0,1] \times [0,1] \to \mathbf{R}$ be defined by

$$f(x,y) = \begin{cases} 0 & x \text{ irrational}, \\ 0 & x \text{ rational}, y \text{ irrational}, \\ 1/q & x \text{ rational}, y = p/q \text{ in lowest terms}. \end{cases}$$

Show that f is integrable and $\int_{[0,1]\times[0,1]} f = 0$.

MEASURE ZERO AND CONTENT ZERO

A subset A of $\mathbf{R}^n$ has (n-dimensional) **measure** 0 if for every $\varepsilon > 0$ there is a cover $\{U_1, U_2, U_3, \ldots\}$ of A by closed rectangles such that $\sum_{i=1}^{\infty} v(U_i) < \varepsilon$. It is obvious (but nevertheless useful to remember) that if A has measure 0 and $B \subset A$, then B has measure 0. The reader may verify that open rectangles may be used instead of closed rectangles in the definition of measure 0.

A set with only finitely many points clearly has measure 0. If A has infinitely many points which can be arranged in a sequence $a_1, a_2, a_3, \ldots$, then A also has measure 0, for if $\varepsilon > 0$, we can choose U_i to be a closed rectangle containing a_i with $v(U_i) < \varepsilon/2^i$. Then $\sum_{i=1}^{\infty} v(U_i) < \sum_{i=1}^{\infty} \varepsilon/2^i = \varepsilon$.

The set of all rational numbers between 0 and 1 is an important and rather surprising example of an infinite set whose members can be arranged in such a sequence. To see that this is so, list the fractions in the following array in the order indicated by the arrows (deleting repetitions and numbers greater than 1):

$$\begin{array}{ccccc}
0/1 & 1/1 & 2/1 & 3/1 & 4/1 \\
0/2 & 1/2 & 2/2 & 3/2 & 4/2 \\
0/3 & 1/3 & 2/3 & 3/3 & 4/3 \\
0/4 & & & &
\end{array}$$

An important generalization of this idea can be given.

3-4 Theorem. *If* $A = A_1 \cup A_2 \cup A_3 \cup \cdots$ *and each* A_i *has measure* 0, *then* A *has measure* 0.

Proof. Let $\varepsilon > 0$. Since A_i has measure 0, there is a cover $\{U_{i,1}, U_{i,2}, U_{i,3}, \ldots\}$ of A_i by closed rectangles such that $\sum_{j=1}^{\infty} v(U_{i,j}) < \varepsilon/2^i$. Then the collection of all $U_{i,j}$ is a cover

of A. By considering the array

$$\begin{array}{cccc} U_{1,1} & U_{1,2} & U_{1,3} & \cdots \\ U_{2,1} & U_{2,2} & U_{2,3} & \cdots \\ U_{3,1} & U_{3,2} & U_{3,3} & \cdots \end{array}$$

we see that this collection can be arranged in a sequence $V_1, V_2, V_3, \ldots$. Clearly $\Sigma_{i=1}^{\infty} v(V_i) < \Sigma_{i=1}^{\infty} \varepsilon/2^i = \varepsilon$. ∎

A subset A of $\mathbf{R}^n$ has (n-dimensional) **content** 0 if for every $\varepsilon > 0$ there is a *finite* cover $\{U_1, \ldots, U_n\}$ of A by closed rectangles such that $\Sigma_{i=1}^{n} v(U_i) < \varepsilon$. If A has content 0, then A clearly has measure 0. Again, open rectangles could be used instead of closed rectangles in the definition.

3-5 Theorem. *If $a < b$, then $[a,b] \subset \mathbf{R}$ does* not *have content* 0. *In fact, if $\{U_1, \ldots, U_n\}$ is a finite cover of $[a,b]$ by closed intervals, then $\Sigma_{i=1}^{n} v(U_i) \geq b - a$.*

Proof. Clearly we can assume that each $U_i \subset [a,b]$. Let $a = t_0 < t_1 < \ldots < t_k = b$ be all endpoints of all U_i. Then each $v(U_i)$ is the sum of certain $t_j - t_{j-1}$. Moreover, each $[t_{j-1},t_j]$ lies in at least one U_i (namely, any one which contains an interior point of $[t_{j-1},t_j]$), so $\Sigma_{i=1}^{n} v(U_i) \geq \Sigma_{j=1}^{k}(t_j - t_{j-1}) = b - a$. ∎

If $a < b$, it is also true that $[a,b]$ does not have measure 0. This follows from

3-6 Theorem. *If A is compact and has measure* 0, *then A has content* 0.

Proof. Let $\varepsilon > 0$. Since A has measure 0, there is a cover $\{U_1, U_2, \ldots\}$ of A by open rectangles such that $\Sigma_{i=1}^{\infty} v(U_i)$

$< \varepsilon$. Since A is compact, a finite number $U_1, \ldots, U_n$ of the U_i also cover A and surely $\sum_{i=1}^{n} v(U_i) < \varepsilon$. ∎

The conclusion of Theorem 3-6 is false if A is not compact. For example, let A be the set of rational numbers between 0 and 1; then A has measure 0. Suppose, however, that $\{[a_1,b_1], \ldots, [a_n,b_n]\}$ covers A. Then A is contained in the closed set $[a_1,b_1] \cup \cdots \cup [a_n,b_n]$, and therefore $[0,1] \subset [a_1,b_1] \cup \cdots \cup [a_n,b_n]$. It follows from Theorem 3-5 that $\sum_{i=1}^{n}(b_i - a_i) \geq 1$ for any such cover, and consequently A does not have content 0.

Problems. **3-8.** Prove that $[a_1,b_1] \times \cdots \times [a_n,b_n]$ does not have content 0 if $a_i < b_i$ for each i.

3-9. (a) Show that an unbounded set cannot have content 0.

(b) Give an example of a closed set of measure 0 which does not have content 0.

3-10. (a) If C is a set of content 0, show that the boundary of C has content 0.

(b) Give an example of a bounded set C of measure 0 such that the boundary of C does not have measure 0.

3-11. Let A be the set of Problem 1-18. If $\sum_{i=1}^{\infty}(b_i - a_i) < 1$, show that the boundary of A does not have measure 0.

3-12. Let $f\colon [a,b] \to \mathbf{R}$ be an increasing function. Show that $\{x\colon f$ is discontinuous at $x\}$ has measure 0. *Hint:* Use Problem 1-30 to show that $\{x\colon o(f,x) > 1/n\}$ is finite, for each integer n.

3-13.* (a) Show that the collection of all rectangles $[a_1,b_1] \times \cdots \times [a_n,b_n]$ with all a_i and b_i rational can be arranged in a sequence.

(b) If $A \subset \mathbf{R}^n$ is any set and $\mathcal{O}$ is an open cover of A, show that there is a sequence $U_1, U_2, U_3, \ldots$ of members of $\mathcal{O}$ which also cover A. *Hint:* For each $x \in A$ there is a rectangle $B = [a_1,b_1] \times \cdots \times [a_n,b_n]$ with all a_i and b_i rational such that $x \in B \subset U$ for some $U \in \mathcal{O}$.

INTEGRABLE FUNCTIONS

Recall that $o(f,x)$ denotes the oscillation of f at x.

3-7 Lemma. *Let A be a closed rectangle and let $f\colon A \to \mathbf{R}$ be a bounded function such that $o(f,x) < \varepsilon$ for all $x \in A$. Then there is a partition P of A with $U(f,P) - L(f,P) < \varepsilon \cdot v(A)$.*

Proof. For each $x \in A$ there is a closed rectangle U_x, containing x in its interior, such that $M_{U_x}(f) - m_{U_x}(f) < \varepsilon$. Since A is compact, a finite number $U_{x_1}, \ldots, U_{x_n}$ of the sets U_x cover A. Let P be a partition for A such that each subrectangle S of P is contained in some U_{x_i}. Then $M_S(f) - m_S(f) < \varepsilon$ for each subrectangle S of P, so that $U(f,P) - L(f,P) = \Sigma_S[M_S(f) - m_S(f)] \cdot v(S) < \varepsilon \cdot v(A)$. ∎

3-8 Theorem. *Let A be a closed rectangle and $f\colon A \to \mathbf{R}$ a bounded function. Let $B = \{x\colon f$ is not continuous at $x\}$. Then f is integrable if and only if B is a set of measure* 0.

Proof. Suppose first that B has measure 0. Let $\varepsilon > 0$ and let $B_\varepsilon = \{x\colon o(f,x) \geq \varepsilon\}$. Then $B_\varepsilon \subset B$, so that B_ε has measure 0. Since (Theorem 1-11) B_ε is compact, B_ε has content 0. Thus there is a finite collection $U_1, \ldots, U_n$ of closed rectangles, whose interiors cover B_ε, such that $\Sigma_{i=1}^n v(U_i) < \varepsilon$. Let P be a partition of A such that every subrectangle S of P is in one of two groups (see Figure 3-1):

FIGURE 3-1. *The shaded rectangles are in* $\mathcal{S}_1$.

(1) $\mathcal{S}_1$, which consists of subrectangles S, such that $S \subset U_i$ for some i.

(2) $\mathcal{S}_2$, which consists of subrectangles S with $S \cap B_\varepsilon = \emptyset$.

Let $|f(x)| < M$ for $x \in A$. Then $M_S(f) - m_S(f) < 2M$ for every S. Therefore

$$\sum_{S \in \mathcal{S}_1} [M_S(f) - m_S(f)] \cdot v(S) < 2M \sum_{i=1}^{n} v(U_i) < 2M\varepsilon.$$

Now, if $S \in \mathcal{S}_2$, then $o(f,x) < \varepsilon$ for $x \in S$. Lemma 3-7 implies that there is a refinement P' of P such that

$$\sum_{S' \subset S} [M_{S'}(f) - m_{S'}(f)] \cdot v(S') < \varepsilon \cdot v(S)$$

for $S \in \mathcal{S}_2$. Then

$$\begin{aligned} U(f,P') - L(f,P') &= \sum_{S' \subset S \in \mathcal{S}_1} [M_{S'}(f) - m_{S'}(f)] \cdot v(S') \\ &\quad + \sum_{S' \subset S \in \mathcal{S}_2} [M_{S'}(f) - m_{S'}(f)] \cdot v(S') \\ &< 2M\varepsilon + \sum_{S \in \mathcal{S}_2} \varepsilon \cdot v(S) \\ &\leq 2M\varepsilon + \varepsilon \cdot v(A). \end{aligned}$$

Since M and $v(A)$ are fixed, this shows that we can find a partition P' with $U(f,P') - L(f,P')$ as small as desired. Thus f is integrable.

Suppose, conversely, that f is integrable. Since $B = B_1 \cup B_{\frac{1}{2}} \cup B_{\frac{1}{3}} \cup \cdots$, it suffices (Theorem 3-4) to prove that each $B_{1/n}$ has measure 0. In fact we will show that each $B_{1/n}$ has content 0 (since $B_{1/n}$ is compact, this is actually equivalent).

If $\varepsilon > 0$, let P be a partition of A such that $U(f,P) - L(f,P) < \varepsilon/n$. Let $\mathcal{S}$ be the collection of subrectangles S of P which intersect $B_{1/n}$. Then $\mathcal{S}$ is a cover of $B_{1/n}$. Now if

$S \in \mathcal{S}$, then $M_S(f) - m_S(f) \geq 1/n$. Thus

$$\begin{aligned}\frac{1}{n} \cdot \sum_{S \in \mathcal{S}} v(S) &\leq \sum_{S \in \mathcal{S}} [M_S(f) - m_S(f)] \cdot v(S) \\ &\leq \sum_{S} [M_S(f) - m_S(f)] \cdot v(S) \\ &< \frac{\varepsilon}{n},\end{aligned}$$

and consequently $\Sigma_{S \in \mathcal{S}} v(S) < \varepsilon$. ∎

We have thus far dealt only with the integrals of functions over rectangles. Integrals over other sets are easily reduced to this type. If $C \subset \mathbf{R}^n$, the **characteristic function** χ_C of C is defined by

$$\chi_C(x) = \begin{cases} 0 & x \notin C, \\ 1 & x \in C. \end{cases}$$

If $C \subset A$ for some closed rectangle A and $f\colon A \to \mathbf{R}$ is bounded, then $\int_C f$ is defined as $\int_A f \cdot \chi_C$, provided $f \cdot \chi_C$ is integrable. This certainly occurs (Problem 3-14) if f and χ_C are integrable.

3-9 Theorem. *The function $\chi_C\colon A \to \mathbf{R}$ is integrable if and only if the boundary of C has measure* 0 (*and hence content* 0).

Proof. If x is in the interior of C, then there is an open rectangle U with $x \in U \subset C$. Thus $\chi_C = 1$ on U and χ_C is clearly continuous at x. Similarly, if x is in the exterior of C, there is an open rectangle U with $x \in U \subset \mathbf{R}^n - C$. Hence $\chi_C = 0$ on U and χ_C is continuous at x. Finally, if x is in the boundary of C, then for every open rectangle U containing x, there is $y_1 \in U \cap C$, so that $\chi_C(y_1) = 1$ and there is $y_2 \in U \cap (\mathbf{R}^n - C)$, so that $\chi_C(y_2) = 0$. Hence χ_C is not continuous at x. Thus $\{x\colon \chi_C$ is not continuous at $x\} =$ boundary C, and the result follows from Theorem 3-8. ∎

A bounded set C whose boundary has measure 0 is called **Jordan-measurable.** The integral $\int_C 1$ is called the (n-dimensional) **content** of C, or the (n-dimensional) **volume** of C. Naturally one-dimensional volume is often called **length,** and two-dimensional volume, **area.**

Problem 3-11 shows that even an open set C may not be Jordan-measurable, so that $\int_C f$ is not necessarily defined even if C is open and f is continuous. This unhappy state of affairs will be rectified soon.

Problems. **3-14.** Show that if $f,g\colon A \to \mathbf{R}$ are integrable, so is $f \cdot g$.

3-15. Show that if C has content 0, then $C \subset A$ for some closed rectangle A and C is Jordan-measurable and $\int_A \chi_C = 0$.

3-16. Give an example of a bounded set C of measure 0 such that $\int_A \chi_C$ does not exist.

3-17. If C is a bounded set of measure 0 and $\int_A \chi_C$ exists, show that $\int_A \chi_C = 0$. *Hint:* Show that $L(f,P) = 0$ for all partitions P. Use Problem 3-8.

3-18. If $f\colon A \to \mathbf{R}$ is non-negative and $\int_A f = 0$, show that $\{x\colon f(x) \neq 0\}$ has measure 0. *Hint:* Prove that $\{x\colon f(x) > 1/n\}$ has content 0.

3-19. Let U be the open set of Problem 3-11. Show that if $f = \chi_U$ except on a set of measure 0, then f is not integrable on $[0,1]$.

3-20. Show that an increasing function $f\colon [a,b] \to \mathbf{R}$ is integrable on $[a,b]$.

3-21. If A is a closed rectangle, show that $C \subset A$ is Jordan-measurable if and only if for every $\varepsilon > 0$ there is a partition P of A such that $\sum_{S\in\mathcal{S}_1} v(S) - \sum_{S\in\mathcal{S}_2} v(S) < \varepsilon$, where $\mathcal{S}_1$ consists of all subrectangles intersecting C and $\mathcal{S}_2$ all subrectangles contained in C.

3-22.* If A is a Jordan-measurable set and $\varepsilon > 0$, show that there is a compact Jordan-measurable set $C \subset A$ such that $\int_{A-C} 1 < \varepsilon$.

FUBINI'S THEOREM

The problem of calculating integrals is solved, in some sense, by Theorem 3-10, which reduces the computation of integrals over a closed rectangle in $\mathbf{R}^n$, $n > 1$, to the computation of integrals over closed intervals in $\mathbf{R}$. Of sufficient importance to deserve a special designation, this theorem is usually referred to as Fubini's theorem, although it is more or less a

special case of a theorem proved by Fubini long after Theorem 3-10 was known.

The idea behind the theorem is best illustrated (Figure 3-2) for a positive continuous function f: $[a,b] \times [c,d] \to \mathbf{R}$. Let $t_0, \ldots, t_n$ be a partition of $[a,b]$ and divide $[a,b] \times [c,d]$ into n strips by means of the line segments $\{t_i\} \times [c,d]$. If g_x is defined by $g_x(y) = f(x,y)$, then the area of the region under the graph of f and above $\{x\} \times [c,d]$ is

$$\int_c^d g_x = \int_c^d f(x,y)dy.$$

The volume of the region under the graph of f and above $[t_{i-1},t_i] \times [c,d]$ is therefore approximately equal to $(t_i - t_{i-1}) \cdot \int_c^d f(x,y)dy$, for any $x \in [t_{i-1},t_i]$. Thus

$$\int_{[a,b]\times[c,d]} f = \sum_{i=1}^{n} \int_{[t_{i-1},t_i]\times[c,d]} f$$

is approximately $\Sigma_{i=1}^{n}(t_i - t_{i-1}) \cdot \int_c^d f(x_i,y)dy$, with x_i in

FIGURE 3-2

$[t_{i-1},t_i]$. On the other hand, sums similar to these appear in the definition of $\int_a^b(\int_c^d f(x,y)dy)dx$. Thus, if h is defined by $h(x) = \int_c^d g_x = \int_c^d f(x,y)dy$, it is reasonable to hope that h is integrable on $[a,b]$ and that

$$\int_{[a,b]\times[c,d]} f = \int_a^b h = \int_a^b \left(\int_c^d f(x,y)dy\right) dx.$$

This will indeed turn out to be true when f is continuous, but in the general case difficulties arise. Suppose, for example, that the set of discontinuities of f is $\{x_0\} \times [c,d]$ for some $x_0 \in [a,b]$. Then f is integrable on $[a,b] \times [c,d]$ but $h(x_0) = \int_c^d f(x_0,y)dy$ may not even be defined. The statement of Fubini's theorem therefore looks a little strange, and will be followed by remarks about various special cases where simpler statements are possible.

We will need one bit of terminology. If $f\colon A \to \mathbf{R}$ is a bounded function on a closed rectangle, then, whether or not f is integrable, the least upper bound of all lower sums, and the greatest lower bound of all upper sums, both exist. They are called the **lower** and **upper integrals** of f on A, and denoted

$$\mathbf{L}\int_A f \quad \text{and} \quad \mathbf{U}\int_A f,$$

respectively.

3-10 Theorem (Fubini's Theorem). *Let $A \subset \mathbf{R}^n$ and $B \subset \mathbf{R}^m$ be closed rectangles, and let $f\colon A \times B \to \mathbf{R}$ be integrable. For $x \in A$ let $g_x\colon B \to \mathbf{R}$ be defined by $g_x(y) = f(x,y)$ and let*

$$\mathfrak{L}(x) = \mathbf{L}\int_B g_x = \mathbf{L}\int_B f(x,y)dy,$$
$$\mathfrak{U}(x) = \mathbf{U}\int_B g_x = \mathbf{U}\int_B f(x,y)dy.$$

Then $\mathfrak{L}$ and $\mathfrak{U}$ are integrable on A and

$$\int_{A\times B} f = \int_A \mathfrak{L} = \int_A \left(\mathbf{L}\int_B f(x,y)dy\right) dx,$$
$$\int_{A\times B} f = \int_A \mathfrak{U} = \int_A \left(\mathbf{U}\int_B f(x,y)dy\right) dx.$$

(The integrals on the right side are called **iterated integrals** for f.)

Proof. Let P_A be a partition of A and P_B a partition of B. Together they give a partition P of $A \times B$ for which any subrectangle S is of the form $S_A \times S_B$, where S_A is a subrectangle of the partition P_A, and S_B is a subrectangle of the partition P_B. Thus

$$L(f,P) = \sum_S m_S(f) \cdot v(S) = \sum_{S_A,S_B} m_{S_A \times S_B}(f) \cdot v(S_A \times S_B)$$
$$= \sum_{S_A} \Big(\sum_{S_B} m_{S_A \times S_B}(f) \cdot v(S_B) \Big) \cdot v(S_A).$$

Now, if $x \in S_A$, then clearly $m_{S_A \times S_B}(f) \leq m_{S_B}(g_x)$. Consequently, for $x \in S_A$ we have

$$\sum_{S_B} m_{S_A \times S_B}(f) \cdot v(S_B) \leq \sum_{S_B} m_{S_B}(g_x) \cdot v(S_B) \leq \mathbf{L} \int_B g_x = \mathfrak{L}(x).$$

Therefore

$$\sum_{S_A} \Big(\sum_{S_B} m_{S_A \times S_B}(f) \cdot v(S_B) \Big) \cdot v(S_A) \leq L(\mathfrak{L},P_A).$$

We thus obtain

$$L(f,P) \leq L(\mathfrak{L},P_A) \leq U(\mathfrak{L},P_A) \leq U(\mathfrak{U},P_A) \leq U(f,P),$$

where the proof of the last inequality is entirely analogous to the proof of the first. Since f is integrable, $\sup\{L(f,P)\} = \inf\{U(f,P)\} = \int_{A \times B} f$. Hence

$$\sup\{L(\mathfrak{L},P_A)\} = \inf\{U(\mathfrak{L},P_A)\} = \int_{A \times B} f.$$

In other words, $\mathfrak{L}$ is integrable on A and $\int_{A \times B} f = \int_A \mathfrak{L}$. The assertion for $\mathfrak{U}$ follows similarly from the inequalities

$$L(f,P) \leq L(\mathfrak{L},P_A) \leq L(\mathfrak{U},P_A) \leq U(\mathfrak{U},P_A) \leq U(f,P). \quad \blacksquare$$

Remarks. 1. A similar proof shows that

$$\int_{A \times B} f = \int_B \Big(\mathbf{L} \int_A f(x,y)dx \Big) dy = \int_B \Big(\mathbf{U} \int_A f(x,y)dx \Big) dy.$$

These integrals are called *iterated integrals* for f in the reverse order from those of the theorem. As several problems show, the possibility of interchanging the orders of iterated integrals has many consequences.

2. In practice it is often the case that each g_x is integrable, so that $\int_{A\times B} f = \int_A (\int_B f(x,y)dy)dx$. This certainly occurs if f is continuous.

3. The worst irregularity commonly encountered is that g_x is not integrable for a finite number of $x \in A$. In this case $\mathfrak{L}(x) = \int_B f(x,y)dy$ for all but these finitely many x. Since $\int_A \mathfrak{L}$ remains unchanged if $\mathfrak{L}$ is redefined at a finite number of points, we can still write $\int_{A\times B} f = \int_A (\int_B f(x,y)dy)dx$, provided that $\int_B f(x,y)dy$ is defined arbitrarily, say as 0, when it does not exist.

4. There are cases when this will not work and Theorem 3-10 must be used as stated. Let $f\colon [0,1] \times [0,1] \to \mathbf{R}$ be defined by

$$f(x,y) = \begin{cases} 1 & \text{if } x \text{ is irrational,} \\ 1 & \text{if } x \text{ is rational and } y \text{ is irrational,} \\ 1 - 1/q & \text{if } x = p/q \text{ in lowest terms and } y \text{ is rational.} \end{cases}$$

Then f is integrable and $\int_{[0,1]\times[0,1]} f = 1$. Now $\int_0^1 f(x,y)dy = 1$ if x is irrational, and does not exist if x is rational. Therefore h is not integrable if $h(x) = \int_0^1 f(x,y)dy$ is set equal to 0 when the integral does not exist.

5. If $A = [a_1,b_1] \times \cdots \times [a_n,b_n]$ and $f\colon A \to \mathbf{R}$ is sufficiently nice, we can apply Fubini's theorem repeatedly to obtain

$$\int_A f = \int_{a_n}^{b_n} \left(\cdots \left(\int_{a_1}^{b_1} f(x^1, \ldots, x^n)dx^1 \right) \cdots \right) dx^n.$$

6. If $C \subset A \times B$, Fubini's theorem can be used to evaluate $\int_C f$, since this is by definition $\int_{A\times B} \chi_C f$. Suppose, for example, that

$$C = [-1,1] \times [-1,1] - \{(x,y)\colon |(x,y)| < 1\}.$$

Then

$$\int_C f = \int_{-1}^{1} \left(\int_{-1}^{1} f(x,y) \cdot \chi_C(x,y)dy \right) dx.$$

Now

$$\chi_C(x,y) = \begin{cases} 1 & \text{if } y > \sqrt{1-x^2} \text{ or } y < -\sqrt{1-x^2}, \\ 0 & \text{otherwise.} \end{cases}$$

Therefore

$$\int_{-1}^{1} f(x,y) \cdot \chi_C(x,y)dy = \int_{-1}^{-\sqrt{1-x^2}} f(x,y)dy + \int_{\sqrt{1-x^2}}^{1} f(x,y)dy.$$

In general, if $C \subset A \times B$, the main difficulty in deriving expressions for $\int_C f$ will be determining $C \cap (\{x\} \times B)$ for $x \in A$. If $C \cap (A \times \{y\})$ for $y \in B$ is easier to determine, one should use the iterated integral

$$\int_C f = \int_B \left(\int_A f(x,y) \cdot \chi_C(x,y)dx \right) dy.$$

Problems. **3-23.** Let $C \subset A \times B$ be a set of content 0. Let $A' \subset A$ be the set of all $x \in A$ such that $\{y \in B\colon (x,y) \in C\}$ is *not* of content 0. Show that A' is a set of measure 0. *Hint:* χ_C is integrable and $\int_{A\times B} \chi_C = \int_A \mathfrak{U} = \int_A \mathfrak{L}$, so $\int_A \mathfrak{U} - \mathfrak{L} = 0$.

3-24. Let $C \subset [0,1] \times [0,1]$ be the union of all $\{p/q\} \times [0, 1/q]$, where p/q is a rational number in $[0,1]$ written in lowest terms. Use C to show that the word "measure" in Problem 3-23 cannot be replaced by "content."

3-25. Use induction on n to show that $[a_1,b_1] \times \cdots \times [a_n,b_n]$ is not a set of measure 0 (or content 0) if $a_i < b_i$ for each i.

3-26. Let $f\colon [a,b] \to \mathbf{R}$ be integrable and non-negative and let $A_f = \{(x,y)\colon a \le x \le b \text{ and } 0 \le y \le f(x)\}$. Show that A_f is Jordan-measurable and has area $\int_a^b f$.

3-27. If $f\colon [a,b] \times [a,b] \to \mathbf{R}$ is continuous, show that

$$\int_a^b \int_a^y f(x,y)dx\, dy = \int_a \int_x f(x,y)dy\, dx.$$

Hint: Compute $\int_C f$ in two different ways for a suitable set $C \subset [a,b] \times [a,b]$.

3-28.* Use Fubini's theorem to give an easy proof that $D_{1,2}f = D_{2,1}f$ if these are continuous. *Hint:* If $D_{1,2}f(a) - D_{2,1}f(a) > 0$, there is a rectangle A containing a such that $D_{1,2}f - D_{2,1}f > 0$ on A.

3-29. Use Fubini's theorem to derive an expression for the volume of a set of $\mathbf{R}^3$ obtained by revolving a Jordan-measurable set in the yz-plane about the z-axis.

3-30. Let C be the set in Problem 1-17. Show that

$$\int_{[0,1]} \left(\int_{[0,1]} \chi_C(x,y)dx \right) dy = \int_{[0,1]} \left(\int_{[0,1]} \chi_C(y,x)dy \right) dx = 0$$

but that $\int_{[0,1]\times[0,1]} \chi_C$ does not exist.

3-31. If $A = [a_1,b_1] \times \cdots \times [a_n,b_n]$ and $f\colon A \to \mathbf{R}$ is continuous, define $F\colon A \to \mathbf{R}$ by

$$F(x) = \int_{[a_1,x^1]\times \cdots \times [a_n,x^n]} f.$$

What is $D_iF(x)$, for x in the interior of A?

3-32.* Let $f\colon [a,b] \times [c,d] \to \mathbf{R}$ be continuous and suppose D_2f is continuous. Define $F(y) = \int_a^b f(x,y)dx$. Prove *Leibnitz's rule:* $F'(y) = \int_a^b D_2f(x,y)dx$. *Hint:* $F(y) = \int_a^b f(x,y)dx = \int_a^b(\int_c^y D_2f(x,y)dy + f(x,c))dx$. (The proof will show that continuity of D_2f may be replaced by considerably weaker hypotheses.)

3-33. If $f\colon [a,b] \times [c,d] \to \mathbf{R}$ is continuous and D_2f is continuous, define $F(x,y) = \int_a^x f(t,y)dt$.

(a) Find D_1F and D_2F.

(b) If $G(x) = \int_a^{g(x)} f(t,x)dt$, find $G'(x)$.

3-34.* Let $g_1,g_2\colon \mathbf{R}^2 \to \mathbf{R}$ be continuously differentiable and suppose $D_1g_2 = D_2g_1$. As in Problem 2-21, let

$$f(x,y) = \int_0^x g_1(t,0)dt + \int_0^y g_2(x,t)dt.$$

Show that $D_1f(x,y) = g_1(x,y)$.

3-35.* (*a*) Let $g\colon \mathbf{R}^n \to \mathbf{R}^n$ be a linear transformation of one of the following types:

$$\begin{cases} g(e_i) = e_i & i \neq j \\ g(e_j) = ae_j \end{cases}$$

$$\begin{cases} g(e_i) = e_i & i \neq j \\ g(e_j) = e_j + e_k \end{cases}$$

$$\begin{cases} g(e_k) = e_k & k \neq i, j \\ g(e_i) = e_j \\ g(e_j) = e_i. \end{cases}$$

If U is a rectangle, show that the volume of $g(U)$ is $|\det g| \cdot v(U)$.

(b) Prove that $|\det g| \cdot v(U)$ is the volume of $g(U)$ for any linear transformation $g\colon \mathbf{R}^n \to \mathbf{R}^n$. *Hint:* If $\det g \neq 0$, then g is the composition of linear transformations of the type considered in (a).

3-36. (Cavalieri's principle). Let A and B be Jordan-measurable subsets of $\mathbf{R}^3$. Let $A_c = \{(x,y)\colon (x,y,c) \in A\}$ and define B_c similarly. Suppose each A_c and B_c are Jordan-measurable and have the same area. Show that A and B have the same volume.

PARTITIONS OF UNITY

In this section we introduce a tool of extreme importance in the theory of integration.

3-11 Theorem. *Let $A \subset \mathbf{R}^n$ and let $\mathcal{O}$ be an open cover of A. Then there is a collection Φ of C^∞ functions φ defined in an open set containing A, with the following properties:*

(1) *For each $x \in A$ we have $0 \leq \varphi(x) \leq 1$.*
(2) *For each $x \in A$ there is an open set V containing x such that all but finitely many $\varphi \in \Phi$ are 0 on V.*
(3) *For each $x \in A$ we have $\Sigma_{\varphi \in \Phi} \varphi(x) = 1$* (by (2) for each x this sum is finite in some open set containing x).
(4) *For each $\varphi \in \Phi$ there is an open set U in $\mathcal{O}$ such that $\varphi = 0$ outside of some closed set contained in U.*

(A collection Φ satisfying (1) to (3) is called a C^∞ **partition of unity** for A. If Φ also satisfies (4), it is said to be **subordinate** to the cover $\mathcal{O}$. In this chapter we will only use continuity of the functions φ.)

Proof. *Case 1. A is compact.*

Then a finite number $U_1, \ldots, U_n$ of open sets in $\mathcal{O}$ cover A. It clearly suffices to construct a partition of unity subordinate to the cover $\{U_1, \ldots, U_n\}$. We will first find compact sets $D_i \subset U_i$ whose interiors cover A. The sets D_i are constructed inductively as follows. Suppose that $D_1, \ldots, D_k$ have been chosen so that $\{\text{interior } D_1, \ldots, \text{interior } D_k, U_{k+1}, \ldots, U_n\}$ covers A. Let

$$C_{k+1} = A - (\text{int } D_1 \cup \cdots \cup \text{int } D_k \cup U_{k+2} \cup \cdots \cup U_n).$$

Then $C_{k+1} \subset U_{k+1}$ is compact. Hence (Problem 1-22) we can find a compact set D_{k+1} such that

$$C_{k+1} \subset \text{interior } D_{k+1} \quad \text{and} \quad D_{k+1} \subset U_{k+1}.$$

Having constructed the sets $D_1, \ldots, D_n$, let ψ_i be a nonnegative C^∞ function which is positive on D_i and 0 outside of some closed set contained in U_i (Problem 2-26). Since

$\{D_1, \ldots, D_n\}$ covers A, we have $\psi_1(x) + \cdots + \psi_n(x) > 0$ for all x in some open set U containing A. On U we can define

$$\varphi_i(x) = \frac{\psi_i(x)}{\psi_1(x) + \cdots + \psi_n(x)}.$$

If $f\colon U \to [0,1]$ is a C^∞ function which is 1 on A and 0 outside of some closed set in U, then $\Phi = \{f \cdot \varphi_1, \ldots, f \cdot \varphi_n\}$ is the desired partition of unity.

Case 2. $A = A_1 \cup A_2 \cup A_3 \cup \cdots$, *where each* A_i *is compact and* $A_i \subset$ *interior* A_{i+1}.

For each i let $\mathcal{O}_i$ consist of all $U \cap$ (interior $A_{i+1} - A_{i-2}$) for U in $\mathcal{O}$. Then $\mathcal{O}_i$ is an open cover of the compact set $B_i = A_i -$ interior A_{i-1}. By case 1 there is a partition of unity Φ_i for B_i, subordinate to $\mathcal{O}_i$. For each $x \in A$ the sum

$$\sigma(x) = \sum_{\varphi \in \Phi_i,\ \text{all } i} \varphi(x)$$

is a finite sum in some open set containing x, since if $x \in A_i$ we have $\varphi(x) = 0$ for $\varphi \in \Phi_j$ with $j \geq i + 2$. For each φ in each Φ_i, define $\varphi'(x) = \varphi(x)/\sigma(x)$. The collection of all φ' is the desired partition of unity.

Case 3. *A is open.*

Let $A_i =$

$$\{x \in A\colon |x| \leq i \text{ and distance from } x \text{ to boundary } A \geq 1/i\},$$

and apply case 2.

Case 4. *A is arbitrary.*

Let B be the union of all U in $\mathcal{O}$. By case 3 there is a partition of unity for B; this is also a partition of unity for A. ∎

An important consequence of condition (2) of the theorem should be noted. Let $C \subset A$ be compact. For each $x \in C$ there is an open set V_x containing x such that only finitely many $\varphi \in \Phi$ are not 0 on V_x. Since C is compact, finitely many such V_x cover C. Thus only finitely many $\varphi \in \Phi$ are not 0 on C.

One important application of partitions of unity will illustrate their main role—piecing together results obtained locally.

An open cover $\mathcal{O}$ of an open set $A \subset \mathbf{R}^n$ is **admissible** if each $U \in \mathcal{O}$ is contained in A. If Φ is subordinate to $\mathcal{O}$, $f: A \to \mathbf{R}$ is bounded in some open set around each point of A, and $\{x: f \text{ is discontinuous at } x\}$ has measure 0, then each $\int_A \varphi \cdot |f|$ exists. We define f to be **integrable** (in the extended sense) if $\Sigma_{\varphi \in \Phi} \int_A \varphi \cdot |f|$ converges (the proof of Theorem 3-11 shows that the φ's may be arranged in a sequence). This implies convergence of $\Sigma_{\varphi \in \Phi} \left| \int_A \varphi \cdot f \right|$, and hence absolute convergence of $\Sigma_{\varphi \in \Phi} \int_A \varphi \cdot f$, which we define to be $\int_A f$. These definitions do not depend on $\mathcal{O}$ or Φ (but see Problem 3-38).

3-12 Theorem.

(1) *If Ψ is another partition of unity, subordinate to an admissible cover $\mathcal{O}'$ of A, then $\Sigma_{\psi \in \Psi} \int_A \psi \cdot |f|$ also converges, and*

$$\sum_{\varphi \in \Phi} \int_A \varphi \cdot f = \sum_{\psi \in \Psi} \int_A \psi \cdot f.$$

(2) *If A and f are bounded, then f is integrable in the extended sense.*

(3) *If A is Jordan-measurable and f is bounded, then this definition of $\int_A f$ agrees with the old one.*

Proof

(1) Since $\varphi \cdot f = 0$ except on some compact set C, and there are only finitely many ψ which are non-zero on C, we can write

$$\sum_{\varphi \in \Phi} \int_A \varphi \cdot f = \sum_{\varphi \in \Phi} \int_A \sum_{\psi \in \Psi} \psi \cdot \varphi \cdot f = \sum_{\varphi \in \Phi} \sum_{\psi \in \Psi} \int_A \psi \cdot \varphi \cdot f.$$

This result, applied to $|f|$, shows the convergence of $\Sigma_{\varphi \in \Phi} \Sigma_{\psi \in \Psi} \int_A \psi \cdot \varphi \cdot |f|$, and hence of $\Sigma_{\varphi \in \Phi} \Sigma_{\psi \in \Psi} \left| \int_A \psi \cdot \varphi \cdot f \right|$. This absolute convergence justifies interchanging the order of summation in the above equation; the resulting double sum clearly equals $\Sigma_{\psi \in \Psi} \int_A \psi \cdot f$. Finally, this result applied to $|f|$ proves convergence of $\Sigma_{\psi \in \Psi} \int_A \psi \cdot |f|$.

(2) If A is contained in the closed rectangle B and $|f(x)| \leq M$ for $x \in A$, and $F \subset \Phi$ is finite, then

$$\sum_{\varphi \in F} \int_A \varphi \cdot |f| \leq \sum_{\varphi \in F} M \int_A \varphi = M \int_A \sum_{\varphi \in F} \varphi \leq Mv(B),$$

since $\Sigma_{\varphi \in F}\, \varphi \leq 1$ on A.

(3) If $\varepsilon > 0$ there is (Problem 3-22) a compact Jordan-measurable $C \subset A$ such that $\int_{A-C} 1 < \varepsilon$. There are only finitely many $\varphi \in \Phi$ which are non-zero on C. If $F \subset \Phi$ is any finite collection which includes these, and $\int_A f$ has its old meaning, then

$$\begin{aligned}\left|\int_A f - \sum_{\varphi \in F} \int_A \varphi \cdot f\right| &\leq \int_A \left|f - \sum_{\varphi \in F} \varphi \cdot f\right| \\ &\leq M \int_A \left(1 - \sum_{\varphi \in F} \varphi\right) \\ &= M \int_A \sum_{\varphi \in \Phi - F} \varphi \leq M \int_{A-C} 1 \leq M\varepsilon. \quad \blacksquare\end{aligned}$$

Problems. **3-37.** (a) Suppose that $f\colon (0,1) \to \mathbf{R}$ is a non-negative continuous function. Show that $\int_{(0,1)} f$ exists if and only if $\lim_{\varepsilon \to 0} \int_\varepsilon^{1-\varepsilon} f$ exists.

(b) Let $A_n = [1 - 1/2^n, 1 - 1/2^{n+1}]$. Suppose that $f\colon (0,1) \to \mathbf{R}$ satisfies $\int_{A_n} f = (-1)^n/n$ and $f(x) = 0$ for $x \notin$ any A_n. Show that $\int_{(0,1)} f$ does not exist, but $\lim_{\varepsilon \to 0} \int_{(\varepsilon, 1-\varepsilon)} f = \log 2$.

3-38. Let A_n be a closed set contained in $(n, n+1)$. Suppose that $f\colon \mathbf{R} \to \mathbf{R}$ satisfies $\int_{A_n} f = (-1)^n/n$ and $f = 0$ for $x \notin$ any A_n. Find two partitions of unity Φ and Ψ such that $\Sigma_{\varphi \in \Phi} \int_{\mathbf{R}} \varphi \cdot f$ and $\Sigma_{\psi \in \Psi} \int_{\mathbf{R}} \psi \cdot f$ converge absolutely to different values.

CHANGE OF VARIABLE

If $g\colon [a,b] \to \mathbf{R}$ is continuously differentiable and $f\colon \mathbf{R} \to \mathbf{R}$ is continuous, then, as is well known,

$$\int_{g(a)}^{g(b)} f = \int_a^b (f \circ g) \cdot g'.$$

The proof is very simple: if $F' = f$, then $(F \circ g)' = (f \circ g) \cdot g'$; thus the left side is $F(g(b)) - F(g(a))$, while the right side is $F \circ g(b) - F \circ g(a) = F(g(b)) - F(g(a))$.

We leave it to the reader to show that if g is 1-1, then the above formula can be written

$$\int_{g((a,b))} f = \int_{(a,b)} f \circ g \cdot |g'|.$$

(Consider separately the cases where g is increasing and where g is decreasing.) The generalization of this formula to higher dimensions is by no means so trivial.

3-13 Theorem. *Let $A \subset \mathbf{R}^n$ be an open set and $g\colon A \to \mathbf{R}^n$ a 1-1, continuously differentiable function such that $\det g'(x) \neq 0$ for all $x \in A$. If $f\colon g(A) \to \mathbf{R}$ is integrable, then*

$$\int_{g(A)} f = \int_A (f \circ g)|\det g'|.$$

Proof. We begin with some important reductions.

1. Suppose there is an admissible cover $\mathcal{O}$ for A such that for each $U \in \mathcal{O}$ and any integrable f we have

$$\int_{g(U)} f = \int_U (f \circ g)|\det g'|.$$

Then the theorem is true for all of A. (Since g is automatically 1-1 in an open set around each point, it is not surprising that this is the only part of the proof using the fact that g is 1-1 on all of A.)

Proof of (1). The collection of all $g(U)$ is an open cover of $g(A)$. Let Φ be a partition of unity subordinate to this cover. If $\varphi = 0$ outside of $g(U)$, then, since g is 1-1, we have $(\varphi \cdot f) \circ g$

$= 0$ outside of U. Therefore the equation

$$\int_{g(U)} \varphi \cdot f = \int_U [(\varphi \cdot f) \circ g]|\det g'|.$$

can be written

$$\int_{g(A)} \varphi \cdot f = \int_A [(\varphi \cdot f) \circ g]|\det g'|.$$

Hence

$$\begin{aligned}\int_{g(A)} f = \sum_{\varphi \in \Phi} \int_{g(A)} \varphi \cdot f &= \sum_{\varphi \in \Phi} \int_A [(\varphi \cdot f) \circ g]|\det g'| \\ &= \sum_{\varphi \in \Phi} \int_A (\varphi \circ g)(f \circ g)|\det g'| \\ &= \int_A (f \circ g)|\det g'|.\end{aligned}$$

Remark. The theorem also follows from the assumption that

$$\int_V f = \int_{g^{-1}(V)} (f \circ g)|\det g'|$$

for V in some admissible cover of $g(A)$. This follows from (1) applied to g^{-1}.

2. It suffices to prove the theorem for the function $f = 1$.

Proof of (*2*). If the theorem holds for $f = 1$, it holds for constant functions. Let V be a rectangle in $g(A)$ and P a partition of V. For each subrectangle S of P let f_S be the constant function $m_S(f)$. Then

$$\begin{aligned}L(f,P) &= \sum_S m_S(f) \cdot v(S) = \sum_S \int_{\text{int } S} f_S \\ &= \sum_S \int_{g^{-1}(\text{int } S)} (f_S \circ g)|\det g' \leq \sum_S \int_{g^{-1}(\text{int } S)} (f \circ g)|\det g'| \\ &\leq \int_{g^{-1}(V)} (f \circ g)|\det g'|.\end{aligned}$$

Since $\int_V f$ is the least upper bound of all $L(f,P)$, this proves that $\int_V f \leq \int_{g^{-1}(V)} (f \circ g)|\det g'|$. A similar argument, letting $f_S = M_S(f)$, shows that $\int_V f \geq \int_{g^{-1}(V)} (f \circ g)|\det g'|$. The result now follows from the above Remark.

3. If the theorem is true for $g\colon A \to \mathbf{R}^n$ and for $h\colon B \to \mathbf{R}^n$, where $g(A) \subset B$, then it is true for $h \circ g\colon A \to \mathbf{R}^n$.

Proof of (3).

$$\begin{aligned}\int_{h \circ g(A)} f = \int_{h(g(A))} f &= \int_{g(A)} (f \circ h)|\det h'| \\ &= \int_A [(f \circ h) \circ g] \cdot [|\det h'| \circ g] \cdot |\det g'| \\ &= \int_A f \circ (h \circ g)|\det (h \circ g)'|.\end{aligned}$$

4. The theorem is true if g is a linear transformation.

Proof of (4). By (1) and (2) it suffices to show for any open rectangle U that

$$\int_{g(U)} 1 = \int_U |\det g'|.$$

This is Problem 3-35.

Observations (3) and (4) together show that we may assume for any particular $a \in A$ that $g'(a)$ is the identity matrix: in fact, if T is the linear transformation $Dg(a)$, then $(T^{-1} \circ g)'(a) = I$; since the theorem is true for T, if it is true for $T^{-1} \circ g$ it will be true for g.

We are now prepared to give the proof, which preceeds by induction on n. The remarks before the statement of the theorem, together with (1) and (2), prove the case $n = 1$. Assuming the theorem in dimension $n - 1$, we prove it in dimension n. For each $a \in A$ we need only find an open set U with $a \in U \subset A$ for which the theorem is true. Moreover we may assume that $g'(a) = I$.

Define $h\colon A \to \mathbf{R}^n$ by $h(x) = (g^1(x), \ldots, g^{n-1}(x), x^n)$. Then $h'(a) = I$. Hence in some open U' with $a \in U' \subset A$, the function h is 1-1 and $\det h'(x) \neq 0$. We can thus define $k\colon h(U') \to \mathbf{R}^n$ by $k(x) = (x^1, \ldots, x^{n-1}, g^n(h^{-1}(x)))$ and $g = k \circ h$. We have thus expressed g as the composition

FIGURE 3-3

of two maps, each of which changes fewer than n coordinates (Figure 3-3).

We must attend to a few details to ensure that k is a function of the proper sort. Since

$$(g^n \circ h^{-1})'(h(a)) = (g^n)'(a) \cdot [h'(a)]^{-1} = (g^n)'(a),$$

we have $D_n(g^n \circ h^{-1})(h(a)) = D_n g^n(a) = 1$, so that $k'(h(a)) = I$. Thus in some open set V with $h(a) \in V \subset h(U')$, the function k is 1-1 and $\det k'(x) \neq 0$. Letting $U = k^{-1}(V)$ we now have $g = k \circ h$, where $h\colon U \to \mathbf{R}^n$ and $k\colon V \to \mathbf{R}^n$ and $h(U) \subset V$. By (3) it suffices to prove the theorem for h and k. We give the proof for h; the proof for k is similar and easier.

Let $W \subset U$ be a rectangle of the form $D \times [a_n, b_n]$, where D is a rectangle in $\mathbf{R}^{n-1}$. By Fubini's theorem

$$\int_{h(W)} 1 = \int_{[a_n,b_n]} \left(\int_{h(D \times \{x^n\})} 1\, dx^1 \cdots dx^{n-1} \right) dx^n.$$

Let $h_{x^n}\colon D \to \mathbf{R}^{n-1}$ be defined by $h_{x^n}(x^1, \ldots, x^{n-1}) = (g^1(x^1, \ldots, x^n), \ldots, g^{n-1}(x^1, \ldots, x^n))$. Then each h_{x^n} is clearly 1-1 and

$$\det (h_{x^n})'(x^1, \ldots, x^{n-1}) = \det h'(x^1, \ldots, x^n) \neq 0.$$

Moreover

$$\int_{h(D \times \{x^n\})} 1\, dx^1 \cdots dx^{n-1} = \int_{h_{x^n}(D)} 1\, dx^1 \cdots dx^{n-1}.$$

Applying the theorem in the case $n - 1$ therefore gives

$$\begin{aligned}
\int_{h(W)} 1 &= \int_{[a_n,b_n]} \left(\int_{h_{x^n}(D)} 1\, dx^1 \cdots dx^{n-1} \right) dx^n \\
&= \int_{[a_n,b_n]} \left(\int_D |\det (h_{x^n})'(x^1, \ldots, x^{n-1})| dx^1 \cdots dx^{n-1} \right) dx^n \\
&= \int_{[a_n,b_n]} \left(\int_D |\det h'(x^1, \ldots, x^n)| dx^1 \cdots dx^{n-1} \right) dx^n \\
&= \int_W |\det h'|. \quad \blacksquare
\end{aligned}$$

The condition $\det g'(x) \neq 0$ may be eliminated from the

hypotheses of Theorem 3-13 by using the following theorem, which often plays an unexpected role.

3-14. Theorem (Sard's Theorem). *Let $g: A \to \mathbf{R}^n$ be continuously differentiable, where $A \subset \mathbf{R}^n$ is open, and let $B = \{x \in A: \det g'(x) = 0\}$. Then $g(B)$ has measure 0.*

Proof. Let $U \subset A$ be a closed rectangle such that all sides of U have length l, say. Let $\varepsilon > 0$. If N is sufficiently large and U is divided into N^n rectangles, with sides of length l/N, then for each of these rectangles S, if $x \in S$ we have

$$|Dg(x)(y - x) - g(y) - g(x)| < \varepsilon|x - y| \leq \varepsilon\sqrt{n}\,(l/N)$$

for all $y \in S$. If S intersects B we can choose $x \in S \cap B$; since $\det g'(x) = 0$, the set $\{Dg(x)(y - x): y \in S\}$ lies in an $(n - 1)$-dimensional subspace V of $\mathbf{R}^n$. Therefore the set $\{g(y) - g(x): y \in S\}$ lies within $\varepsilon\sqrt{n}\,(l/N)$ of V, so that $\{g(y): y \in S\}$ lies within $\varepsilon\sqrt{n}\,(l/N)$ of the $(n - 1)$-plane $V + g(x)$. On the other hand, by Lemma 2-10 there is a number M such that

$$|g(x) - g(y)| < M|x - y| \leq M\sqrt{n}\,(l/N).$$

Thus, if S intersects B, the set $\{g(y): y \in S\}$ is contained in a cylinder whose height is $< 2\varepsilon\sqrt{n}\,(l/N)$ and whose base is an $(n - 1)$-dimensional sphere of radius $< M\sqrt{n}\,(l/N)$. This cylinder has volume $< C(l/N)^n\varepsilon$ for some constant C. There are at most N^n such rectangles S, so $g(U \cap B)$ lies in a set of volume $< C(l/N)^n \cdot \varepsilon \cdot N^n = Cl^n \cdot \varepsilon$. Since this is true for all $\varepsilon > 0$, the set $g(U \cap B)$ has measure 0. Since (Problem 3-13) we can cover all of A with a sequence of such rectangles U, the desired result follows from Theorem 3-4. ∎

Theorem 3-14 is actually only the easy part of Sard's Theorem. The statement and proof of the deeper result will be found in [17], page 47.

Problems. **3-39.** Use Theorem 3-14 to prove Theorem 3-13 without the assumption $\det g'(x) \neq 0$.

3-40. If g: $\mathbf{R}^n \to \mathbf{R}^n$ and $\det g'(x) \neq 0$, prove that in some open set containing x we can write $g = T \circ g_n \circ \cdots \circ g_1$, where g_i is of the form $g_i(x) = (x^1, \ldots, f_i(x), \ldots, x^n)$, and T is a linear transformation. Show that we can write $g = g_n \circ \cdots \circ g_1$ if and only if $g'(x)$ is a diagonal matrix.

3-41. Define f: $\{r: r > 0\} \times (0,2\pi) \to \mathbf{R}^2$ by $f(r,\theta) = (r \cos \theta, r \sin \theta)$.

(a) Show that f is 1-1, compute $f'(r,\theta)$, and show that $\det f'(r,\theta) \neq 0$ for all (r,θ). Show that $f(\{r: r > 0\} \times (0,2\pi))$ is the set A of Problem 2-23.

(b) If $P = f^{-1}$, show that $P(x,y) = (r(x,y),\theta(x,y))$, where

$$r(x,y) = \sqrt{x^2 + y^2},$$

$$\theta(x,y) = \begin{cases} \arctan y/x & x > 0,\ y > 0, \\ \pi + \arctan y/x & x < 0, \\ 2\pi + \arctan y/x & x > 0,\ y < 0, \\ \pi/2 & x = 0,\ y > 0, \\ 3\pi/2 & x = 0,\ y < 0. \end{cases}$$

(Here arctan denotes the inverse of the function tan: $(-\pi/2,\pi/2) \to \mathbf{R}$.) Find $P'(x,y)$. The function P is called the **polar coordinate system** on A.

(c) Let $C \subset A$ be the region between the circles of radii r_1 and r_2 and the half-lines through 0 which make angles of θ_1 and θ_2 with the x-axis. If h: $C \to \mathbf{R}$ is integrable and $h(x,y) = g(r(x,y),\theta(x,y))$, show that

$$\int_C h = \int_{r_1}^{r_2} \int_{\theta_1}^{\theta_2} rg(r,\theta)\,d\theta\,dr.$$

If $B_r = \{(x,y): x^2 + y^2 \leq r^2\}$, show that

$$\int_{B_r} h = \int_0^r \int_0^{2\pi} rg(r,\theta)\,d\theta\,dr.$$

(d) If $C_r = [-r,r] \times [-r,r]$, show that

$$\int_{B_r} e^{-(x^2+y^2)}\,dx\,dy = \pi(1 - e^{-r^2})$$

and

$$\int_{C_r} e^{-(x^2+y^2)}\,dx\,dy = \left(\int_{-r}^{r} e^{-x^2}\,dx\right)^2.$$

(e) Prove that

$$\lim_{r\to\infty} \int_{B_r} e^{-(x^2+y^2)}\,dx\,dy = \lim_{r\to\infty} \int_{C_r} e^{-(x^2+y^2)}\,dx\,dy$$

and conclude that

$$\int_{-\infty}^{\infty} e^{-x^2}\,dx = \sqrt{\pi}.$$

"A mathematician is one to whom *that* is as obvious as that twice two makes four is to you. Liouville was a mathematician."

—Lord Kelvin

4

Integration on Chains

ALGEBRAIC PRELIMINARIES

If V is a vector space (over $\mathbf{R}$), we will denote the k-fold product $V \times \cdots \times V$ by V^k. A function $T\colon V^k \to \mathbf{R}$ is called **multilinear** if for each i with $1 \leq i \leq k$ we have

$$\begin{aligned} T(v_1, \ldots, v_i + v_i', \ldots, v_k) &= T(v_1, \ldots, v_i, \ldots, v_k) \\ &\quad + T(v_1, \ldots, v_i', \ldots, v_k), \\ T(v_1, \ldots, av_i, \ldots, v_k) &= aT(v_1, \ldots, v_i, \ldots, v_k). \end{aligned}$$

A multilinear function $T\colon V^k \to \mathbf{R}$ is called a ***k*-tensor** on V and the set of all k-tensors, denoted $\mathfrak{J}^k(V)$, becomes a vector space (over $\mathbf{R}$) if for $S, T \in \mathfrak{J}^k(V)$ and $a \in \mathbf{R}$ we define

$$\begin{aligned} (S + T)(v_1, \ldots, v_k) &= S(v_1, \ldots, v_k) + T(v_1, \ldots, v_k), \\ (aS)(v_1, \ldots, v_k) &= a \cdot S(v_1, \ldots, v_k). \end{aligned}$$

There is also an operation connecting the various spaces $\mathfrak{J}^k(V)$. If $S \in \mathfrak{J}^k(V)$ and $T \in \mathfrak{J}^l(V)$, we define the **tensor product** $S \otimes T \in \mathfrak{J}^{k+l}(V)$ by

$$\begin{aligned} S \otimes T(v_1, \ldots, v_k, v_{k+1}, \ldots, v_{k+l}) \\ = S(v_1, \ldots, v_k) \cdot T(v_{k+1}, \ldots, v_{k+l}). \end{aligned}$$

Note that the order of the factors S and T is crucial here since $S \otimes T$ and $T \otimes S$ are far from equal. The following properties of $\otimes$ are left as easy exercises for the reader.

$$\begin{aligned}(S_1 + S_2) \otimes T &= S_1 \otimes T + S_2 \otimes T,\\ S \otimes (T_1 + T_2) &= S \otimes T_1 + S \otimes T_2,\\ (aS) \otimes T &= S \otimes (aT) = a(S \otimes T),\\ (S \otimes T) \otimes U &= S \otimes (T \otimes U).\end{aligned}$$

Both $(S \otimes T) \otimes U$ and $S \otimes (T \otimes U)$ are usually denoted simply $S \otimes T \otimes U$; higher-order products $T_1 \otimes \cdots \otimes T_r$ are defined similarly.

The reader has probably already noticed that $\mathfrak{J}^1(V)$ is just the dual space V^*. The operation $\otimes$ allows us to express the other vector spaces $\mathfrak{J}^k(V)$ in terms of $\mathfrak{J}^1(V)$.

4-1 Theorem. *Let $v_1, \ldots, v_n$ be a basis for V, and let $\varphi_1, \ldots, \varphi_n$ be the dual basis, $\varphi_i(v_j) = \delta_{ij}$. Then the set of all k-fold tensor products*

$$\varphi_{i_1} \otimes \cdots \otimes \varphi_{i_k} \qquad 1 \leq i_1, \ldots, i_k \leq n$$

is a basis for $\mathfrak{J}^k(V)$, which therefore has dimension n^k.

Proof. Note that

$$\begin{aligned}\varphi_{i_1} \otimes \cdots \otimes \varphi_{i_k}(v_{j_1}, \ldots, v_{j_k}) &\\ &= \delta_{i_1,j_1} \cdot \ldots \cdot \delta_{i_k,j_k}\\ &= \begin{cases} 1 & \text{if } j_1 = i_1, \ldots, j_k = i_k,\\ 0 & \text{otherwise.}\end{cases}\end{aligned}$$

If $w_1, \ldots, w_k$ are k vectors with $w_i = \Sigma_{j=1}^n a_{ij}v_j$ and T is in $\mathfrak{J}^k(V)$, then

$$T(w_1, \ldots, w_k) = \sum_{j_1, \ldots, j_k = 1}^{n} a_{1,j_1} \cdot \ldots \cdot a_{k,j_k} T(v_{j_1}, \ldots, v_{j_k})$$

$$= \sum_{i_1, \ldots, i_k = 1}^{n} T(v_{i_1}, \ldots, v_{i_k}) \cdot \varphi_{i_1} \otimes \cdots \otimes \varphi_{i_k}(w_1, \ldots, w_k).$$

Thus $T = \Sigma_{i_1, \ldots, i_k = 1}^n T(v_{i_1}, \ldots, v_{i_k}) \cdot \varphi_{i_1} \otimes \cdots \otimes \varphi_{i_k}$.

Consequently the $\varphi_{i_1} \otimes \cdots \otimes \varphi_{i_k}$ span $\mathfrak{J}^k(V)$.

Suppose now that there are numbers $a_{i_1,\ldots,i_k}$ such that

$$\sum_{i_1,\ldots,i_k=1}^{n} a_{i_1,\ldots,i_k} \cdot \varphi_{i_1} \otimes \cdots \otimes \varphi_{i_k} = 0.$$

Applying both sides of this equation to $(v_{j_1}, \ldots, v_{j_k})$ yields $a_{j_1,\ldots,j_k} = 0$. Thus the $\varphi_{i_1} \otimes \cdots \otimes \varphi_{i_k}$ are linearly independent. ∎

One important construction, familiar for the case of dual spaces, can also be made for tensors. If $f\colon V \to W$ is a linear transformation, a linear transformation $f^*\colon \mathfrak{J}^k(W) \to \mathfrak{J}^k(V)$ is defined by

$$f^*T(v_1, \ldots, v_k) = T(f(v_1), \ldots, f(v_k))$$

for $T \in \mathfrak{J}^k(W)$ and $v_1, \ldots, v_k \in V$. It is easy to verify that $f^*(S \otimes T) = f^*S \otimes f^*T$.

The reader is already familiar with certain tensors, aside from members of V^*. The first example is the inner product $\langle , \rangle \in \mathfrak{J}^2(\mathbf{R}^n)$. On the grounds that any good mathematical commodity is worth generalizing, we define an **inner product** on V to be a 2-tensor T such that T is **symmetric,** that is $T(v,w) = T(w,v)$ for $v,w \in V$ and such that T is **positive-definite,** that is, $T(v,v) > 0$ if $v \neq 0$. We distinguish $\langle , \rangle$ as the **usual inner product** on $\mathbf{R}^n$. The following theorem shows that our generalization is not too general.

4-2 Theorem. *If T is an inner product on V, there is a basis $v_1, \ldots, v_n$ for V such that $T(v_i,v_j) = \delta_{ij}$. (Such a basis is called* **orthonormal** *with respect to T.) Consequently there is an isomorphism $f\colon \mathbf{R}^n \to V$ such that $T(f(x),f(y)) = \langle x,y \rangle$ for $x,y \in \mathbf{R}^n$. In other words $f^*T = \langle , \rangle$.*

Proof. Let $w_1, \ldots, w_n$ be any basis for V. Define

$$w_1' = w_1,$$

$$w_2' = w_2 - \frac{T(w_1',w_2)}{T(w_1',w_1')} \cdot w_1',$$

$$w_3' = w_3 - \frac{T(w_1',w_3)}{T(w_1',w_1')} \cdot w_1' - \frac{T(w_2',w_3)}{T(w_2',w_2')} \cdot w_2',$$

etc.

It is easy to check that $T(w_i',w_j') = 0$ if $i \neq j$ and $w_i' \neq 0$ so that $T(w_i',w_i') > 0$. Now define $v_i = w_i'/\sqrt{T(w_i',w_i')}$. The isomorphism f may be defined by $f(e_i) = v_i$. ∎

Despite its importance, the inner product plays a far lesser role than another familiar, seemingly ubiquitous function, the tensor det $\in \mathfrak{J}^n(\mathbf{R}^n)$. In attempting to generalize this function, we recall that interchanging two rows of a matrix changes the sign of its determinant. This suggests the following definition. A k-tensor $\omega \in \mathfrak{J}^k(V)$ is called **alternating** if

$$\begin{aligned}\omega(v_1, \ldots, v_i, \ldots, v_j, \ldots, v_k) \\ = -\omega(v_1, \ldots, v_j, \ldots, v_i, \ldots, v_k) \\ \text{for all } v_1, \ldots, v_k \in V.\end{aligned}$$

(In this equation v_i and v_j are interchanged and all other v's are left fixed.) The set of all alternating k-tensors is clearly a subspace $\Lambda^k(V)$ of $\mathfrak{J}^k(V)$. Since it requires considerable work to produce the determinant, it is not surprising that alternating k-tensors are difficult to write down. There is, however, a uniform way of expressing all of them. Recall that the sign of a permutation σ, denoted sgn σ, is $+1$ if σ is even and -1 if σ is odd. If $T \in \mathfrak{J}^k(V)$, we define Alt(T) by

$$\mathrm{Alt}(T)(v_1, \ldots, v_k) = \frac{1}{k!} \sum_{\sigma \in S_k} \mathrm{sgn}\, \sigma \cdot T(v_{\sigma(1)}, \ldots, v_{\sigma(k)}),$$

where S_k is the set of all permutations of the numbers 1 to k.

4-3 Theorem

(1) *If* $T \in \mathfrak{J}^k(V)$, *then* $Alt(T) \in \Lambda^k(V)$.
(2) *If* $\omega \in \Lambda^k(V)$, *then* $Alt(\omega) = \omega$.
(3) *If* $T \in \mathfrak{J}^k(V)$, *then* $Alt(Alt(T)) = Alt(T)$.

Proof

(1) Let (i,j) be the permutation that interchanges i and j and leaves all other numbers fixed. If $\sigma \in S_k$, let $\sigma' = \sigma \cdot (i,j)$. Then

$$\begin{aligned}
&\mathrm{Alt}(T)(v_1, \ldots, v_j, \ldots, v_i, \ldots, v_k) \\
&= \frac{1}{k!} \sum_{\sigma \in S_k} \operatorname{sgn} \sigma \cdot T(v_{\sigma(1)}, \ldots, v_{\sigma(j)}, \ldots, v_{\sigma(i)}, \ldots, v_{\sigma(k)}) \\
&= \frac{1}{k!} \sum_{\sigma \in S_k} \operatorname{sgn} \sigma \cdot T(v_{\sigma'(1)}, \ldots, v_{\sigma'(i)}, \ldots, v_{\sigma'(j)}, \ldots, v_{\sigma'(k)}) \\
&= \frac{1}{k!} \sum_{\sigma' \in S_k} -\operatorname{sgn} \sigma' \cdot T(v_{\sigma'(1)}, \ldots, v_{\sigma'(k)}) \\
&= -\mathrm{Alt}(T)(v_1, \ldots, v_k).
\end{aligned}$$

(2) If $\omega \in \Lambda^k(V)$, and $\sigma = (i,j)$, then $\omega(v_{\sigma(1)}, \ldots, v_{\sigma(k)}) = \operatorname{sgn} \sigma \cdot \omega(v_1, \ldots, v_k)$. Since every σ is a product of permutations of the form (i,j), this equation holds of all σ. Therefore

$$\begin{aligned}
\mathrm{Alt}(\omega)(v_1, \ldots, v_k) &= \frac{1}{k!} \sum_{\sigma \in S_k} \operatorname{sgn} \sigma \cdot \omega(v_{\sigma(1)}, \ldots, v_{\sigma(k)}) \\
&= \frac{1}{k!} \sum_{\sigma \in S_k} \operatorname{sgn} \sigma \cdot \operatorname{sgn} \sigma \cdot \omega(v_1, \ldots, v_k) \\
&= \omega(v_1, \ldots, v_k).
\end{aligned}$$

(3) follows immediately from (1) and (2). ∎

To determine the dimensions of $\Lambda^k(V)$, we would like a theorem analogous to Theorem 4-1. Of course, if $\omega \in \Lambda^k(V)$ and $\eta \in \Lambda^l(V)$, then $\omega \otimes \eta$ is usually not in $\Lambda^{k+l}(V)$. We will therefore define a new product, the **wedge** product $\omega \wedge \eta \in \Lambda^{k+l}(V)$ by

$$\omega \wedge \eta = \frac{(k+l)!}{k!\, l!} \mathrm{Alt}(\omega \otimes \eta).$$

(The reason for the strange coefficient will appear later.) The following properties of $\wedge$ are left as an exercise for the reader:

$$\begin{aligned}
(\omega_1 + \omega_2) \wedge \eta &= \omega_1 \wedge \eta + \omega_2 \wedge \eta, \\
\omega \wedge (\eta_1 + \eta_2) &= \omega \wedge \eta_1 + \omega \wedge \eta_2, \\
a\omega \wedge \eta &= \omega \wedge a\eta = a(\omega \wedge \eta), \\
\omega \wedge \eta &= (-1)^{kl} \eta \wedge \omega, \\
f^*(\omega \wedge \eta) &= f^*(\omega) \wedge f^*(\eta).
\end{aligned}$$