

Appendix A

Mathematical Notation and Background

The design and analysis of algorithms requires familiarity with certain concepts from mathematics such as functions, sets, various summation formulas, and so forth. In this appendix we review some of these commonly used concepts and establish appropriate notation. Various other mathematical concepts are briefly discussed, including complex numbers, modular arithmetic, random walks, and eigenvectors.

A.1 Basic Mathematical and Built-in Functions

Certain functions occur frequently in the design and analysis of algorithms and are usually implemented as built-in functions in most high-level languages. Two of the most commonly used functions are the ceiling $\lceil x \rceil$ and floor $\lfloor x \rfloor$, defined to be the smallest integer greater than or equal to x and the largest integer smaller than or equal to x , respectively. For example, $\lceil 1.23 \rceil = 2$, and $\lfloor 1.23 \rfloor = 1$. For a positive x , $\lfloor x \rfloor$ is obtained from x by truncating its decimal part. Note that $x - 1 < \lfloor x \rfloor \leq x$, and $x \leq \lceil x \rceil < x + 1$.

When $\lceil x \rceil$ and $\lfloor x \rfloor$ are invoked in our pseudocode as built-in functions, we often retain the mathematical notation for such invocations rather than using names such as **ceiling**(x) and **floor**(x), respectively. On the other hand, for functions such as \sqrt{x} , we usually use the name **sqrt**. The following is a list of names and definitions for some of the more commonly used built-in functions in our pseudocode.

$$\mathbf{sqrt}(x) = \sqrt{x}, x \text{ a nonnegative real number}$$

$$\mathbf{abs}(x) = |x| = \begin{cases} x & x \geq 0 \\ -x & \text{otherwise,} \end{cases} \quad x \text{ a real number.}$$

$$a \bmod b = a - b \lfloor a/b \rfloor, \quad a \text{ and } b \text{ integers, } b \neq 0.$$

$$\mathbf{odd}(n) = \mathbf{true.} \text{ if, and only if, } n \bmod 2 = 1, \quad n \text{ an integer.}$$

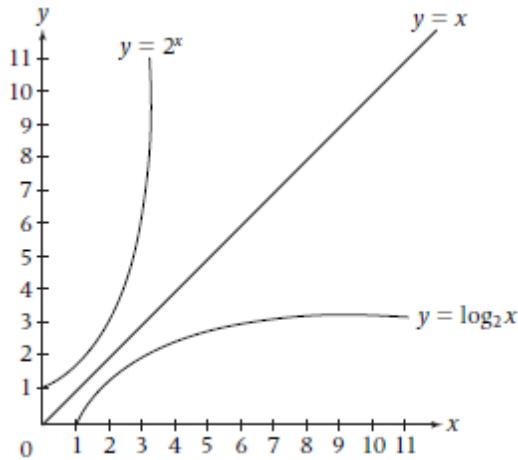
$$\mathbf{even}(n) = \mathbf{true.} \text{ if, and only if, } n \bmod 2 = 0, \quad n \text{ an integer.}$$

Note that for positive integers a and b , $a \bmod b$ is the remainder when a is divided by b .

Another important function in algorithms is the logarithm function. Given a base $b > 1$, the logarithm of x to the base b , denoted by $\log_b x$, is defined to be the functional inverse of the exponential function b^x . In other words, $\log_b x$, $x > 0$, is defined to be the power to which the base b must be raised to equal x , so that

$$b^{\log_b x} = x. \tag{A.1.1}$$

We illustrate the functional inverse relationship of 2^x and $\log_2 x$ in Figure A.1. Note that their graphs are the reflections of one another about the line $y = x$. Since 2^x grows very rapidly, it follows that $\log_2 x$ grows very slowly.



Graphs of $y = \log_2 x$, $y = x$, and $y = 2^x$

Figure A.1

The differences between logarithmic growth ($y = \log_2 x$), linear growth ($y = x$), and exponential growth ($y = 2^x$) are dramatic (see Figure A.2). The first few entries for x in Figure A.2 were obtained by doubling the previous value. Note that doubling the input to the function $\log_2 x$ only increases its output by one. On the other hand, doubling the input to the function 2^x results in squaring its output. This is indeed a dramatic difference!

x	$\log_2 x$	2^x
1	0	2
2	1	4
4	2	16
8	3	256
16	4	65,536
32	5.58496	562,949,953,421,312
64	6.58496	eeeeeeeenormous

Table of values of $y = x$, $y = \log_2 x$, and $y = 2^x$

Figure A.2

The exponential function b^x has the following fundamental properties:

$$b^{x+y} = b^x b^y, \quad (\text{A.1.2})$$

$$(b^x)^y = (b^y)^x = b^{xy}. \quad (\text{A.1.3})$$

The following two properties of $\log_b x$ corresponding to (A.1.2) and (A.1.3) are immediately obtained by raising both sides to the power b and using (A.1.1).

$$\log_b xy = \log_b x + \log_b y, \quad (\text{A.1.4})$$

$$\log_b x^y = y \log_b x. \quad (\text{A.1.5})$$

The following useful formula allows us to change from one base to another:

$$\log_a x = \frac{\log_b x}{\log_b a}. \quad (\text{A.1.6})$$

The most commonly used bases are 2, e , and 10. When the base is e , the logarithm is referred to as the *natural* logarithm and is denoted by $\ln x$.

A.2 Modular Arithmetic

Modular arithmetic is often used in cryptography to compute a cryptographic key. In particular, it is used in Chapter 9 in the design of the RSA public-key cryptosystem. The operations of addition and multiplication for the set $Z_n = \{0, \dots, n - 1\}$ of integers (*residues*) modulo n are defined the same as over the integers, but the result x of each operation is reduced by replacing x with the remainder r when x is divided by n . We denote this remainder by $x \bmod n$. We write

$$x \equiv y \pmod{n}$$

if $x - y$ is divisible by n ; that is, if both x and y have the same remainder upon dividing by n . It is easily verified that Z_n satisfies the following commutative ring properties:

Addition is commutative, associative and every element has an inverse, so that Z_n is a commutative (Abelian) group under addition:

- i) $x + y \equiv y + x \pmod{n}$
- ii) $(x + y) + z \equiv x + (y + z) \pmod{n}$
- iii) $x + (-x) \equiv 0 \pmod{n}$

Multiplication is commutative and associative:

- iv) $x * y \equiv y * x \pmod{n}$
- v) $(x * y) * z \equiv x * (y * z) \pmod{n}$

Multiplication distributes over addition:

- vi) $x * (y + z) \equiv x * y + x * z \pmod{n}$

In the case when n is prime, Z_n is also a commutative group under multiplication, so that it determines a field known as the Galois field of integers modulo n , denoted by $\text{GF}(n)$. It is easily verified that the relation R on the set Z of all integers, given by xRy if and only if $x \equiv y \pmod{n}$, is an equivalence relation, and an element x from $Z_n = \{0, \dots, n - 1\}$ can be identified with the equivalence class $[x] = \{\dots, x - 2n, x - n, x, x + n, x + 2n, \dots\}$ of all integers y such that $x \equiv y \pmod{n}$.

A.3 Some Summation Formulas

Summations involving arithmetic and geometric progressions occur frequently in the analysis of algorithms. Given real numbers a , d , and a positive integer n , the sum of the first n terms of the arithmetic progression with leading term a and difference d is given by

$$a + (a + d) + (a + 2d) + \cdots + (a + (n-1)d) = \frac{n(2a + (n-1)d)}{2}. \quad (\text{A.3.1})$$

Formula (A.3.1) is easily proved by adding the progression to a copy of itself, but where we add the i th term in the original progression to the $(n - i + 1)$ st term in the copy. Then each of the resulting n summands equals $(2a + (n-1)d)$. In algorithm analysis, the formula (A.3.1) occurs most often with $a = d = 1$, in which case it reduces to

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}. \quad (\text{A.3.2})$$

Given a real number x and a positive integer n , the sum of the first n terms of the geometric progression $1, x, x^2, \dots, x^n, \dots$ is given by

$$1 + x + x^2 + \cdots + x^{n-1} = (x^n - 1)/(x - 1). \quad (\text{A.3.3})$$

Formula (A.3.3) follows easily by multiplying both sides by $x - 1$ and simplifying the left-hand side. An important special case in the analysis of algorithms occurs when $x = 2$. Then (A.3.3) reduces to

$$1 + 2 + 2^2 + \cdots + 2^{n-1} = 2^n - 1. \quad (\text{A.3.4})$$

Another way of interpreting formula (A.3.4) is to note that the base-two expansion of the number $2^n - 1$ consists of a sequence of n ones.

A formula that is useful when analyzing the average behavior of algorithms is obtained by replacing n by $n + 1$ in (A.3.3) and then differentiating both sides of the equation, yielding

$$1 + 2x + 3x^2 + \cdots + nx^{n-1} = \frac{nx^{n+1} - (n+1)x^n + 1}{(x-1)^2}. \quad (\text{A.3.5})$$

For x a real number, $-1 < x < 1$, taking the limit of both sides of (A.3.3) and (A.3.5) as n approaches infinity yields the following two useful power series formulas.

$$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x} \quad (\text{A.3.6})$$

$$\sum_{n=0}^{\infty} (n+1)x^n = \frac{1}{(1-x)^2} \quad (\text{A.3.7})$$

Of course, (A.3.7) also can be obtained from (A.3.6) by differentiating both sides of the equation.

A.4 Binomial Coefficients

We now consider two combinatorial quantities that arise often in counting arguments and in probability computations. Suppose we have a finite set S having n elements. For any k between 0 and n , the number of ways to make an *ordered* choice of k elements from S is given by the product

$$n^{(k)} = n(n-1)\dots(n-k+1). \quad (\text{A.4.1})$$

In particular, note that $n! = n^{(n)}$. The following inequality will be useful

$$\frac{n^{(k)}}{m^{(k)}} \leq \left(\frac{n}{m}\right)^k, \quad 0 < k \leq n < m. \quad (\text{A.4.2})$$

Let $C(n, k)$ denote the number of ways to make an *unordered* choice of k elements from S . It then follows that

$$C(n, k) = \frac{n^{(k)}}{k!}, \quad \text{for } k = 0, \dots, n, \quad (\text{A.4.3})$$

since $n^{(k)}$ stood for the number of ordered choices of k elements, and there are $k!$ different orderings of these k elements. Hence, using (A.4.1) and (A.4.3), we have

$$C(n, k) = \frac{n(n-1)\dots(n-k+1)}{k!} = \frac{n!}{(n-k)!k!} \quad \text{for } k = 0, \dots, n. \quad (\text{A.4.4})$$

We refer to $C(n, k)$ as “ n choose k .” We often use the alternate notation $\binom{n}{k}$ for $C(n, k)$. As an illustration, consider the set $S = \{a, b, c, d\}$. There are twelve ways to choose an ordered subset of two elements from S , namely,

$$ab, ba, ac, ca, ad, da, bc, cb, bd, db, cd, dc,$$

but only $12/2!=6$ ways to choose an unordered two-element subset, namely,

$$\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}.$$

Thus, $4^{(2)} = 12$, and $C(4, 2) = \binom{4}{2} = 6$.

The number $C(n, k) = \binom{n}{k}$ is also called a *binomial coefficient* because of its appearance in the binomial expansion:

$$\begin{aligned}
(a+x)^n &= \sum_{i=0}^n \binom{n}{i} a^{n-i} x^i \\
&= \binom{n}{0} a^n x^0 + \binom{n}{1} a^{n-1} x^1 + \binom{n}{2} a^{n-2} x^2 + \dots + \binom{n}{i} a^{n-i} x^i + \dots + \binom{n}{n} a^0 x^n.
\end{aligned} \tag{A.4.5}$$

Taking the partial derivative of both sides of (A.4.5) with respect to x and then multiplying by x yields the following useful identity

$$nx(a+x)^{n-1} = \sum_{i=0}^n \binom{n}{i} a^{n-1} x^i i \tag{A.4.6}$$

The binomial coefficients give us yet another interpretation of the formula $1 + 2 + \dots + n = n(n+1)/2$. Combining (A.3.2) and (A.4.4) yields

$$1 + 2 + \dots + n = \binom{n+1}{2}. \tag{A.4.7}$$

A direct verification of (A.4.7) can be given as follows. Let $\{1, 2, \dots, n+1\}$ be the $(n+1)$ -element set. Then there are n ways to choose a two-element subset containing 1, $n-1$ ways to choose a two-element subset containing 2 but not 1, $n-2$ ways to choose a two-element subset containing 3 but not 1 and 2, and so forth.

The binomial expansion (A.4.5) has an important generalization (due to Newton) to the case where $a = 1$ and $|x| < 1$ and n is an arbitrary real number c .

$$(1+x)^c = \sum_{i=0}^{\infty} \binom{c}{i} x^i, \tag{A.4.8}$$

where the generalized binomial coefficient is defined by

$$\binom{c}{k} = c(c-1)\dots(c-k+1)/k!. \tag{A.4.9}$$

A problem arises when attempting to implement an algorithm for $C(n,k)$ based directly on (A.4.4), namely, the rapid growth of the factorial function $k!$. For example, many compilers store integer variables using four bytes of storage, so that the largest integer that can be stored without overflow is $2^{31} - 1 = 2,147,483,648$. This value is already exceeded by $13!$. One way around this is to calculate $C(n,k)$ as the product of the k

fractions $\frac{n}{1} \frac{(n-1)}{2} \dots \frac{(n-k+1)}{k}$. This avoids integer overflow, but has the disadvantage of using real arithmetic and introducing round-off errors if we simply compute and multiply the above fractions. However, if at the i^{th} stage we multiply the previous result by the numerator $(n-i)$ and then divide by the denominator $i+1$, $i = 1, \dots, k-1$, we avoid round-off errors. In applications where you desire a table of all the binomial

coefficients $C(j,k)$ for $j = 1, \dots, n$, $k = 0, \dots, j$, it is best to use Pascal's recurrence relation (2.2.4):

$$C(n,k) = C(n-1,k-1) + C(n-1,k), \quad \text{init. cond. } C(n,0) = C(n,n) = 1. \quad (\text{A.4.10})$$

To verify (A.23), consider a set of n elements $S = \{s_1, s_2, \dots, s_n\}$. The initial conditions in (A.23) are true since the only subset having zero elements is the empty set and the only subset having n elements is S itself. To prove the recurrence relation, consider the element s_1 . The subsets of S having k elements fall into two disjoint classes: those that contain s_1 and those that don't. If s_1 is in a subset of size k , then the remaining $k-1$ elements of the subset must be chosen from the $n-1$ elements in the subset $S \setminus \{s_1\}$. The number of such choices is $C(n-1, k-1)$. On the other hand, if s_1 is not in a subset of size k , then all k elements of this subset must be chosen from $S \setminus \{s_1\}$. The number of such choices is $C(n-1, k)$. Thus, the total number of ways to choose a subset of k elements is the sum of $C(n-1, k-1)$ and $C(n-1, k)$, which establishes the recurrence relation (A.4.10).

In Figure A.3, we show the famous *Pascal's Triangle*, where each successive row in the triangle is obtained from the row above by using the recurrence (A.4.10). The initial conditions yield the 1s forming two sides of the (infinite) triangle. The famous philosopher and mathematician Blaise Pascal exploited various properties of this triangle in a paper appearing in 1654. However, the triangle itself was already known to the Chinese as early as the eleventh century.

	$k = 0$	1	2	3	4	5	6	7	8	9	.	.
$n = 0$	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		
.
.
.

Pascal's Triangle generated by $C(n,k) = C(n-1,k-1) + C(n-1,k)$, init. cond. $C(n,0) = C(n,n) = 1$

Figure A.3

A.5 Sets

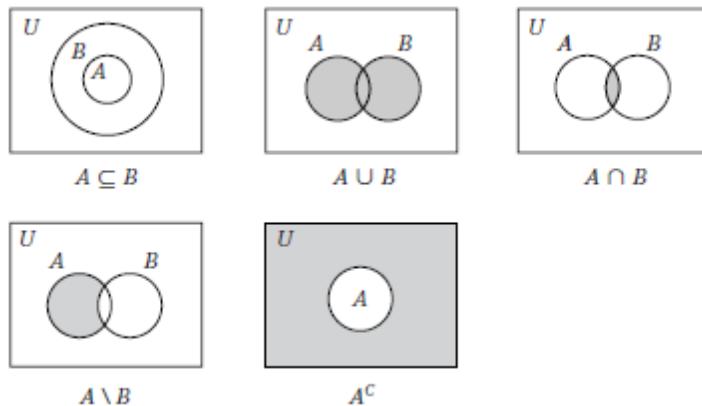
Various definitions and notation from the elementary theory of sets are needed in our discussion of asymptotic behavior given in Chapter 3. These notions from set theory are also required in our discussion of elementary probability theory given in Chapter 6 and

Appendix E. In addition to utilizing sets in the mathematical analysis of algorithms, there are many important algorithms that implement union and find operations for disjoint sets.

Whenever we talk about a set, we assume that the set is contained in a certain *universal set* U . The set U is always clear from context (all edges in a graph, all real-valued functions defined on the nonnegative integers, all outcomes of an experiment, etc.). The set of real numbers and the set of nonnegative integers occur often, and we denote them by \mathbb{R} and \mathbb{N} , respectively.

Given a set A , we indicate that x is a member of (belongs to) A by $x \in A$. We write $x \notin A$ when x is not a member of (does not belong to) A . The set having no members, called the *empty set*, is denoted by \emptyset . Given two sets A and B , we say that A is a *subset* of B , denoted by $A \subseteq B$, if every member of A is also a member of B . We also say that A is *contained in* B . If $A \subseteq B$ and $A \neq B$, then we say that A is a *proper subset* of B , denoted by $A \subset B$. We also say that A is *strictly contained* in B . The *union* of A and B , denoted by $A \cup B$, is the set of all elements that are members of either A or B (or both). The *intersection* of A and B , denoted by $A \cap B$, is the set of all elements that are members of both A and B . The sets A and B are called *disjoint*, if $A \cap B = \emptyset$. The *difference* between A and B , denoted by $A \setminus B$, is the set of all members of A that are not members of B . The *complement* of A , denoted by A^c , is the set of all members of the universal set U that are not members of A , that is $A^c = U \setminus A$. The *Cartesian product* $A \times B$ is the set of all ordered pairs $\{(a,b) : a \in A, b \in B\}$.

Figure A.4 gives *Venn diagrams* illustrating subset, union, intersection, difference, and complement.



Venn diagrams for set operations

Figure A.4

A.6 Complex Numbers

Often when solving problems whose inputs and outputs are real numbers, it is useful in intermediate stages to utilize the complex numbers. For example, in Chapter 5 we develop a fast algorithm for computing the symbolic product of two input polynomials with *real* coefficients (so that the output product polynomial also has *real* coefficients) by using the Fast Fourier Transform (FFT) to evaluate the polynomials at the set of *n*th roots of unity. We now quickly review the necessary theory of complex numbers required by the FFT and similar algorithms.

The use of counting numbers is part of the earliest written historical record. However, the use of negative numbers occurs much later. There are understandable reasons for the relatively late appearance of negative numbers, but their introduction apparently caused little controversy. This was not the case, however, when numbers such as $\sqrt{-1}$ were introduced. Mathematicians wanted to solve simple equations such as $x^2 + 1 = 0$, which had no solutions in the real numbers. Thus, the so-called *complex numbers* of the form $a + ib$ were introduced, where $i = \sqrt{-1}$ stood for a number whose square was -1 . To make the introduction of i more acceptable, it was called an *imaginary number*. More generally, ib is referred to as the *imaginary part* of the complex number $a + ib$ and a is called the *real part*.

The early apologies made for $i = \sqrt{-1}$ are somewhat amusing nowadays, since the complex numbers $a + ib$ can be formally introduced simply as pairs (a, b) of real numbers (so that $a + ib$ corresponds to (a, b)), together with arithmetic operations of addition, subtraction, multiplication (denoted by juxtaposition), and reciprocation defined by

Definitions

- I. $(a, b) \pm (c, d) = (a \pm c, b \pm d)$,
- II. $(a, b)(c, d) = (ac - bd, ad + bc)$,
- III. $(a, b)^{-1} = (a/(a^2 + b^2), -b/(a^2 + b^2))$.

The right-hand sides of I, II, and III use the usual arithmetic operations on real numbers, and the left-hand sides define the corresponding operation (using the same symbols) on the complex numbers. Using III, division is defined as $(a, b)/(c, d) = (a, b)(c, d)^{-1}$. The definitions II and III seem somewhat unmotivated, but they arise naturally as explained shortly. Note that the real number a can be identified with the complex number as $(a, 0)$. The rules I, II and III when restricted to the pairs $(a, 0)$ as a varies over the real numbers coincide with the ordinary arithmetic rules for real numbers. Thus we have naturally extended the real numbers to the larger set of complex numbers. Moreover, $(0, 1)^2 = (-1, 0)$, so that we have indeed found $\sqrt{-1}$, which we denote by i . Of course, $-i = (0, -1)$ also squares to -1 , so that we now have two complex numbers whose square is -1 , whereas the reals have none.

Note that under our identification of b with $(b, 0)$ (and using i to denote $(0, 1)$), we can think of ib as the product of $(0, 1)$ with $(b, 0)$, that is, ib corresponds to $(0, b)$. In the same way, the addition $(a, 0) + (0, b)$ can be written as $a + ib$, and the multiplication $(c, 0)(a, b) = (ca, cb)$ can be viewed as $c(a + ib) = ca + icb$. With similar identifications, we see how definition II arises naturally from the distributive and commutative laws

$$\begin{aligned}
(a,b)(c,d) \text{ corresponds to } & (a+ib)(c+id) \\
&= ac + iad + ibc + i^2 bd \\
&= ac - bd + i(ad + bc) \\
\text{which corresponds to } & (ac - bd, ad + bc).
\end{aligned}$$

Definition III for reciprocals takes its motivation from

$$\frac{1}{a+ib} \frac{a-ib}{a-ib} = \frac{a-ib}{a^2+b^2}.$$

Just as we consider the real numbers as corresponding to points on a straight line, we think of the complex numbers as corresponding to points in a plane. Thus, we identify the complex numbers $z = x + iy$ with points (two-dimensional vectors) (x, y) in the xy plane, with the reals x identified with the points $(x, 0)$ on the x axis. The *modulus* of $z = x + iy$, denoted by $|z|$, is defined to be the distance from (x, y) to the origin $(0, 0)$, that is,

$$|z| = (x^2 + y^2)^{1/2}.$$

Note that the modulus of a point on the x axis corresponds to the absolute value of the point.

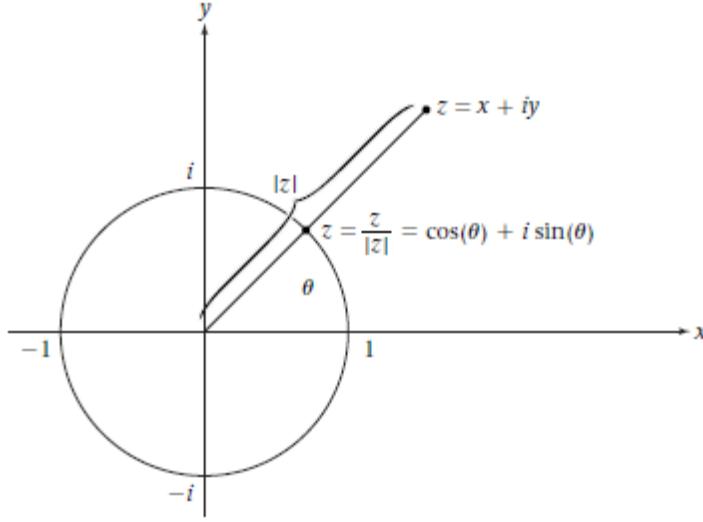
Using the analogy with polar coordinates in the plane, given the complex number $z = x + iy$, it is useful to write z as

$$z = |z| \left(\frac{z}{|z|} \right), \quad (\text{A.6.1})$$

where $\frac{z}{|z|}$ is a point on the unit circle. Hence, $\frac{z}{|z|}$ can be written as $\cos(\theta) + i \sin(\theta)$ for an appropriate angle θ between the vector determined by z and the x axis (see Figure A.5), so that (A.6.1) becomes

$$z = |z| (\cos(\theta) + i \sin(\theta)). \quad (\text{A.6.2})$$

In the representation (A.6.2), the angle θ is referred to as an *argument* of z . If $0 \leq \theta < 2\pi$, then θ is called the *principal argument* of z .



Modulus $|z|$ and principle argument θ of a sample complex number z

Figure A.5

Using the exponential function e^z , we rewrite (A.6.2) as follows. In analogy with the Taylor's expansion for e^x for real x , the Taylor's expansion for e^z for complex z is given by

$$e^z = 1 + z + \frac{z^2}{2!} + \cdots + \frac{z^n}{n!} + \cdots \quad (\text{A.6.3})$$

Substituting $z = i\theta$ into Formula (A.6.3) yields

$$\begin{aligned} e^{i\theta} &= 1 + i\theta - \frac{\theta^2}{2!} - \frac{i\theta^3}{3!} + \frac{\theta^4}{4!} + \frac{i\theta^5}{5!} - \frac{\theta^6}{6!} - \frac{i\theta^7}{7!} + \cdots \\ &= \cos(\theta) + i\sin(\theta). \end{aligned} \quad (\text{A.6.4})$$

Setting $r = |z|$ and substituting (A.6.4) into (A.6.2) yields

$$z = re^{i\theta}, \quad r = |z|, \quad \frac{z}{|z|} = \cos(\theta) + i\sin(\theta). \quad (\text{A.6.5})$$

Remark

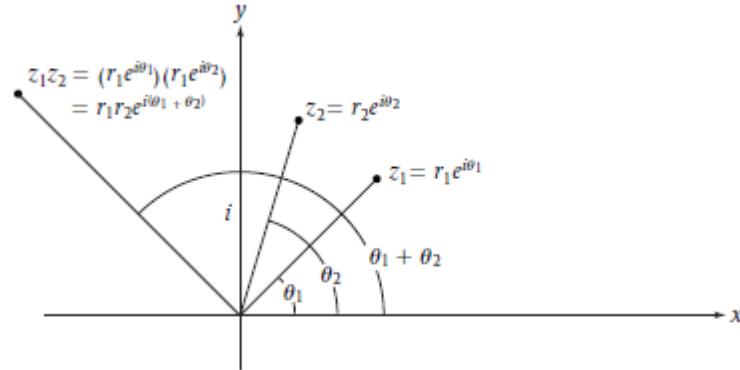
Substituting $\theta = \pi$ into (A.6.5) yields Euler's formula

$$e^{i\pi} = -1,$$

a remarkable formula indeed, since it relates the fundamental mathematical constants e , i , π , and -1 together in such a wonderfully simple way.

The expression (A.6.5) for z yields a nice geometric interpretation (see Figure A.6) for the product of two complex numbers $z_1 = r_1 e^{i\theta_1}$, $z_2 = r_2 e^{i\theta_2}$

$$\begin{aligned} z_1 z_2 &= (r_1 e^{i\theta_1})(r_2 e^{i\theta_2}) \\ &= r_1 r_2 e^{i(\theta_1 + \theta_2)}. \end{aligned} \tag{A.6.6}$$



Geometry of the multiplication of complex numbers

Figure A.6

From the generalization of (A.6.6) to the product of n complex numbers, we have the following multiplication rule for obtaining the modulus and argument of the product of n complex numbers.

Multiplication Rule for n Complex Numbers

Multiply the moduli and add the arguments.

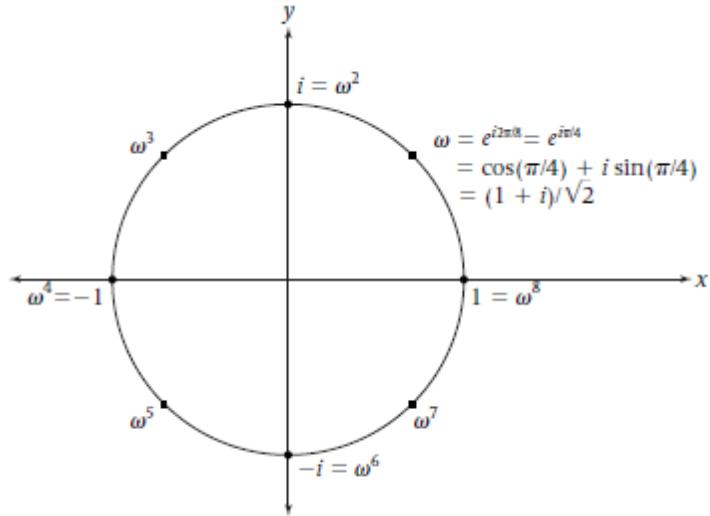
An important special case of the multiplication rule is a formula for the n^{th} power of a complex number $z = r e^{i\theta}$, known as DeMoivre's formula:

$$(r e^{i\theta})^n = r^n e^{in\theta}. \tag{A.6.7}$$

Given the complex number $z \neq 0$, (A.6.7) yields a formula for the n different complex numbers whose n^{th} powers equal z (that is, the n^{th} roots of z):

$$z^{1/n} = \{r^{1/n} (e^{i((\theta+2\pi k)/n)}): z = r e^{i\theta}, \quad 0 \leq \theta < 2\pi, k = 0, 1, \dots, n-1\} \tag{A.6.8}$$

It is easily verified that the n^{th} roots of z defined by (A.6.8) are all distinct. When $z = 1$, (A.6.8) yields the set of n^{th} roots of unity. Note that these points all lie on the unit circle and are equally spaced, starting at 1 (see Figure A.7). Moreover, they are all powers of the single n^{th} root $e^{2\pi i/n}$, which is a primitive n^{th} root of unity. More generally, a primitive n^{th} root of unity ω is a complex number such that $\omega^n = 1$, but $\omega^k \neq 1$ for $k = 1, \dots, n-1$.



Eight roots of unity

Figure A.7

The following useful propositions are easily verified.

Proposition A.6.1

Let ω be a primitive n^{th} root of unity, and let k be an integer such that $0 < k < n$. Then

$$1 + \omega^k + (\omega^k)^2 + \cdots + (\omega^k)^{n-1} = 0.$$

□

Proposition A.6.2

If ω is a primitive n^{th} root of unity, then ω^2 is a primitive $(n/2)^{\text{th}}$ root of unity.

□

Proposition A.6.3

If ω is a primitive n^{th} root of unity and k is a positive integer, then $\omega^k = 1$ if, and only if, n divides k .

□

A.7 Mathematical Induction

Mathematical induction is the most commonly used technique for establishing the correctness of an algorithm. Correctness proofs often proceed by establishing loop invariants with the aid of mathematical induction. Mathematical induction is also useful in both the analysis of algorithms and as an algorithm design tool.

Often we would like to establish the validity of a proposition or formula concerning the set of positive integers such as

$$1^2 + 2^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6} \text{ is true for all positive integers } n.$$

This formula can be easily verified for any particular (small) n by directly computing and comparing both sides of the equation. For example, if $n = 5$, we get $1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 1 + 4 + 9 + 16 + 25 = 55 = 5(6)(11)/6$. The question is: How do we verify that the formula is true for *all* positive integers n ? More generally, suppose we have a sequence of propositions $P(1), P(2), \dots, P(n), \dots$ indexed by the positive integers and we wish to establish the truth of each one of these propositions. For example, the preceding formula is the sequence of propositions

$$P(n) : 1^2 + 2^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

Another example might be

$$P(n) : \text{OddEvenSort1DMesh sorts a list of size } n.$$

Given such a sequence, we clearly can't prove each proposition one at a time; that would take forever. Thus we use the Principle of Mathematical Induction, a very powerful method by which the truth of such a sequence of propositions $P(1), P(2), \dots, P(n), \dots$ can be established.

A.7.1 Principle of Mathematical Induction

We first state the principle of mathematical induction in its usual form. We then state the principle in various alternative (basically equivalent) forms, which are often used in algorithm analysis.

Theorem A.7.1 Principle of Mathematical Induction

Suppose we have a sequence of propositions $P(1), P(2), \dots, P(n), \dots$ for which the following two steps have been established:

Basis step: $P(1)$ is true

Induction (or Implication) step: $\text{if } P(k) \text{ is true for any given } k, \text{ then } P(k+1) \text{ must also be true.}$

Then $P(n)$ is true for all positive integers n .

The validity of the Principle of Mathematical Induction can be seen as follows. Since $P(1)$ is true, the induction step shows that $P(2)$ is true. But the truth of $P(2)$ in turn implies that $P(3)$ is true, and so forth. The induction step allows this process to continue indefinitely. The truth of $P(n)$ for all n rests ultimately on the following property of the positive integers: Every nonempty subset of the positive integers has a smallest element. The assumption that $P(k)$ is false for some k then leads to a contradiction. Indeed, if the set $\{k : P(k) \text{ is false}\}$ is nonempty, then it has a smallest element L . But $l > 1$, so that $P(l-1)$ is true. Letting $n = l-1$, our induction step implies that $P(l) = P(n+1)$ is true, a contradiction.

As our first illustration of the use of mathematical induction, we now establish that the formula $1^2 + 2^2 + \dots + n^2 = n(n+1)(2n+1)/6$ is true for all n . We proceed as follows.

Basis step: $1^2 = 1 = 1(1+1)\frac{1(1+1)(2+1)}{6}$ ($P(1)$ is true).

Induction step: Assume that $P(k)$ is true for a given k , so that $1^2 + 2^2 + \dots + k^2 = k(k+1)(2k+1)/6$. We must show that it would follow that $P(k+1)$ is true, namely, that $1^2 + 2^2 + \dots + (k+1)^2 = (k+1)(k+2)(2k+3)/6$. We have

$$\begin{aligned} 1^2 + 2^2 + \dots + k^2 + (k+1)^2 &= (1^2 + 2^2 + \dots + k^2) + (k+1)^2 \\ &= k(k+1)\frac{(2k+1)}{6} + (k+1)^2 \text{ (since } P(k) \text{ is assumed true)} \\ &= \frac{(k+1)[k(2k+1) + 6(k+1)]}{6} = \frac{(k+1)(2k^2 + 7k + 6)}{6} \\ &= \frac{(k+1)(k+2)(2k+3)}{6}, \end{aligned}$$

and therefore $P(k+1)$ is true.

Thus, we have proved the induction step, and by the Principle of Mathematical Induction, the proposition $1^2 + 2^2 + \dots + n^2 = n(n+1)(2n+1)/6$ is true for all positive integers n .

A.7.2 Variations of the Principle of Mathematical Induction

The following three variations of the principle of mathematical induction are frequently encountered in the analysis of algorithms. In practice, a combination of these variants might be used.

1. Often the sequence of propositions starts with an index different from 1, such as 0. Then the basis step starts with this initial index. The induction step remains the same, and the two steps together establish the truth of the propositions $P(n)$ for all n greater than or equal to this initial index.
2. Sometimes the propositions are only finite in number, $P(1), \dots, P(l)$. Then the induction step is modified to require that $k < l$. Of course, the conclusion then drawn is that $P(1), \dots, P(l)$ are all true if the basis and induction steps are valid.
3. The Principle of Mathematical Induction can also be stated in the following so-called *strong form*, where the induction step is as follows:

Induction step (strong form): For any positive integer k , if $P(j)$ is true for all positive integers $j \leq k$, then $P(k+1)$ must also be true.
is quadratic.

The strong form of induction is very useful in establishing the correctness of recursive algorithms, since for an input of size n the recursive calls often involve smaller input sizes than $n - 1$. For example, to prove that an inorder traversal of a binary search tree on n nodes visits the keys in increasing order, which is the basis of the algorithm *TreeSort* given in Chapter 4, the strong form of induction is needed since the left and right subtrees can have any number of nodes between 0 and $n - 1$.

A.8 Walks in Graphs and Digraphs

A *walk* W of length p in a graph G from u to v or u - v walk is an alternating sequence of vertices and edges $v_0, e_1, v_1, e_2, v_2, \dots, e_p, v_p$, where $v_0 = u$ and $v_p = v$, such that each $e_i = \{v_i, v_{i+1}\}$ is an edge of G , $i = 0, 1, \dots, p$. Vertices v_0 and v_p are the initial and terminal vertices of W , respectively, and vertices v_1, \dots, v_p are the internal vertices of W . A u - v walk in a directed graph D is defined analogously except that $e_i = (v_i, v_{i+1})$ is a directed edge.

The number $w_{ij}(k)$ of walks from v_i to v_j of length k can be computed using the adjacency matrix as follows. We will assume that the node set of the digraph D is $V = \{0, 1, \dots, n - 1\}$.

Theorem A.8.1. The number $w_{ij}(k)$ of walks from i to j of length k in a digraph D equals the ij^{th} entry of the k^{th} power of the adjacency matrix A of D , that is

$$w_{ij}(k) = A^k(i, j), \quad i, j = 0, \dots, n - 1.$$

Proof We prove Theorem A.8.1 using mathematical induction on k . The Theorem is true for $k = 0$ since the only walks of length 0 are the trivial walks with no internal vertices whose initial and terminal vertices are the same, establishing the basis step. Now assume the Theorem is true for walks of length $k - 1$, that is, $w_{ij}(k - 1) = A^{k-1}(i, j)$. It follows from the definition of a walk that

$$w_{ij}(k) = \sum_{il \in E} w_{lj}(k - 1)$$

$$= \sum_{l=0}^{n-1} A(i, l) w_{lj}(k - 1)$$

Thus, by induction hypothesis we have

$$w_{ij}(k) = \sum_{l=0}^{n-1} A(i, l) A^{k-1}(l, j) = A^k(i, j).$$

This completes the induction step and the proof of Theorem A.4. ■

Theorem A.8.1 can be generalized as follows to the case where we associate a real weight p_{ij} with each edge (i, j) .

Theorem A.8.2

Let D be a digraph and let p be a weighting of the edges over the real numbers. Suppose $w_{ij}(k)$ is the sum over all walks W of length k of the product of the p -weights on the edges of W . Then, $w_{ij}(k)$ equals the ij^{th} entry of the k^{th} power of the p -weighted adjacency matrix A_p of D .

An important special case of Theorem A.8.2 is when the vertices of D represent states of a Markov chain and p_{ij} represents the transition probability from state i to state j . The

probability p_{ij} depends only on being at state s_i and moving to state s_j and not on the previous transition history. Further, there is an edge from i to j whenever $p_{ij} > 0$. To represent all possible Markov chains we allow the digraph to have a loop at node i , in which case p_{ii} represents the probability that the state remains the same in the next transition. Letting $p_{ij}(k)$ denote the probability of moving from state i to state j after k transitions, it is easily verified that $p_{ij}(k) = w_{ij}(k)$. Thus, by Theorem A.5, $p_{ij}(k)$ can be computed as the ij^{th} entry of the k^{th} power of the matrix A_p .

A Markov chain can be interpreted as a random walk in the digraph D , or an idle surfer in the case when D is the web digraph W (see Chapter 17), where p_{ij} is the probability that a walker at state i will move to state j in the next step. An important special case is when the transition probabilities for any node i are all same, so that

$$p_{ij} = \begin{cases} 1/d_{\text{out}}(i) & \text{if } (i, j) \in E(D) \\ 0, & \text{otherwise} \end{cases}$$

where $d_{\text{out}}(i)$ denotes the out-degree of vertex i .

We say that the Markov chain is *irreducible* if D is strongly connected and that it is *aperiodic* if for any state i the greatest common divisor over all the numbers k such that $p_{ij}(k) > 0$ is 1; that is, $\gcd \{k \mid p_{ij}(k) > 0\} = 1$. It is easily verified that the latter condition is equivalent to the condition that, for any i , there is a closed walk of length k containing i for all but a finite number of integers k .

A.9 Eigenvalues and Eigenvectors

Let M be an n -by- n matrix over the real numbers. An *eigenvalue* of a M is a real number λ such that

$$MX = \lambda X \tag{A.9.1}$$

for some nonzero column vector X . The vector X is called an *eigenvector* associated with eigenvalue λ . There may be multiple linearly independent eigenvectors associated with the same eigenvalue. For example, if M is the identity matrix then every vector is an eigenvector for eigenvalue 1. Subtracting λX from both sides of Formula (A.9.1), we obtain

$$(M - \lambda I) X = 0.$$

If follows that the matrix $M - \lambda I$ is singular, so that its determinant is zero; that is

$$\det(M - \lambda I) = 0. \tag{A.9.2}$$

All the eigenvalues of M are determined by Formula (A.9.2). Note that the eigenvalues of M are the same as its transpose M^T . If M is the matrix A_p associated with a Markov chain then it can be shown that all the eigenvalues are less than or equal to 1. Further, since $A_p J = J$, where J is the column vector of all 1's, 1 is an eigenvalue of A_p , so that 1 is

the largest eigenvalue of A_p . It follows that 1 is the largest eigenvalue of A_p^T , and there is an eigenvector vector X (written as a column vector) such that

$$A_p^T X = X.$$

If the Markov chain associated with A_p is irreducible and aperiodic, then it can be shown that X is unique. We refer to this unique vector as the *principle eigenvector* or *stationary distribution* of A_p . Further X can be approximated by iterating the formula

$$X_i = A_p X_{i-1}, i = 1, 2, \dots,$$

where the initial column vector X_0 can be taken to be any vector whose entries sum to 1. If the Markov chain associated with the matrix A_p is irreducible and aperiodic then, for any column vector X_0 of positive real numbers whose entries sum to 1, the limit of X_i as i goes to infinity is the principle eigenvector or stationary distribution X of A_p .

PART I

MATHEMATICAL FOUNDATIONS OF ALGORITHMS

1

Introduction and Preliminaries

The use of algorithms did not begin with the introduction of computers. In fact, people have been using algorithms as long as they have been solving problems systematically. While a completely rigorous definition of an algorithm uses the notion of a Turing Machine, the following definition will suffice for our purposes. Informally, an *algorithm* is a complete, step-by-step procedure for solving a specific problem. Each step must be unambiguously expressed in terms of a finite number of rules and guaranteed to terminate in a finite number of applications of these rules. A rule typically calls for the execution of one or more operations. A *sequential* algorithm performs these operations one at a time in sequence, whereas a *parallel* or *distributed* algorithm can perform many operations simultaneously.

In this text, the operations allowed in a sequential algorithm are restricted to instructions found in a typical high-level procedural computer language. These instructions include, for example, arithmetical operations, logical comparisons, and transfer of control. A parallel or distributed algorithm performs the same operations as a sequential algorithm (in addition to communication operations between processors), but a given operation can be performed on multiple data instances simultaneously. For instance, a parallel algorithm might add one to each element in an array of numbers in a single parallel step.

When designing algorithms, it is important to consider the various models and architectures that will implement the algorithms. We discuss these models in a bit more detail later in this chapter. In this chapter we also provide some historical background for the study of sequential, parallel and distributed algorithms, and give a brief trace of how algorithms have developed from ancient times to the present.

1.1 Algorithms from Ancient to Modern Times

The algorithms discussed in this section solve some classical problems in arithmetic. Some algorithms that are commonly executed on today's computers were originally developed more than three thousand years ago. The examples we present illustrate the fact that the most straightforward algorithm for solving a given problem often is not the most efficient.

1.1.1 Evaluating Powers

An ancient problem in arithmetic is the efficient evaluation of integer powers of a number x . The naive approach to evaluating x^n is to repeatedly multiply x by itself $n - 1$ times, yielding the following algorithm.

```
function NaivePowers(x,n)
Input: x (a real number), n (a positive integer)
Output:  $x^n$ 
    Product  $\leftarrow x$ 
    for  $i \leftarrow 1$  to  $n - 1$  do
        Product  $\leftarrow$  Product *  $x$ 
    endfor
    return(Product)
end NaivePowers
```

Clearly, *NaivePowers* performs $n - 1$ multiplications. In particular to compute x^{32} *NaivePowers* uses 31 multiplications. However, after a little thought we could have computed x^{32} using only 5 multiplications, by successive squaring operations, yielding: $x^2, x^4, x^8, x^{16}, x^{32}$. Notice that this simple successive squaring works only for exponents that are a power of 2. In fact we can do much better than the naive algorithm for general n by using an algorithm which has been known for over two millennia, and already was referenced in Pingala's Hindu classic *Chandah-sutra* circa 200 BC. To see how this algorithm might arise, consider the problem of computing x^{108} . By repeatedly applying the simple (recurrence) formula

$$x^n = \begin{cases} (x^{n/2})^2 & n \text{ even,} \\ x * (x^{(n-1)/2})^2 & n \text{ odd,} \end{cases}$$

we obtain the collection of reductions:

$$x^{108} = (x^{54})^2, x^{54} = (x^{27})^2, x^{27} = x(x^{13})^2, x^{13} = x(x^6)^2, x^6 = (x^3)^2, x^3 = x(x^1)^2, x^1 = x(x^0)^2.$$

Now x^{108} can be computed by working our way through these reductions from right to left, involving the sequence of powers (on the left-hand-side of the equalities) 1,3,6,13,27,54,108. Note that when we compute the next power of x , we simply square the previous power of x if the exponent is even, and we multiply the square of the previous power of x by x if the exponent is odd. Determining whether the exponent is even or odd can be conveniently obtained from its binary (base 2) expansion, since it amounts to simply checking whether the least significant binary digit is 0 or 1. Writing the exponents in binary, we obtain the sequence 1, 11, 110, 1101, 11011, 110110, 1101100. Note that taking the least significant digit in each number in the sequence is equivalent to simply scanning the last binary number in the sequence (that is, the original number 108) from left to right, which is why this method is called **left-to-right binary exponentiation**. Note also that, in practice, we can always omit the first step (which always yields x), so that we can start our scan from the second most significant digit. High-level pseudocode for the algorithm follows:

```

function Left-to-Right Binary Method for computing  $x^n$ 
Input:  $x$  (a real number),  $n$  (a positive integer)
Output:  $x^n$ 
    Compute the binary representation of  $n$ 
    Pow  $\leftarrow x$ 
    Scan binary representation from left to right starting with second position
    if 0 is encountered
        Pow  $\leftarrow$  Pow * Pow
    else
        Pow  $\leftarrow x * Pow * Pow$ 
    endif
    return(Product)
end Left-to-Right Binary Method

```

Compute the binary representation of 108 obtaining 110110.

$$\begin{aligned}x \rightarrow x^2 * x &= x^3 \rightarrow (x^3)^2 = x^6 \rightarrow x * (x^6)^2 = x^{13} \rightarrow x * (x^{13})^2 = x^{27} \rightarrow (x^{27})^2 * x \\&= x^{54} \rightarrow (x^{54})^2 = x^{108}\end{aligned}$$

Action of left-to-right binary powers in computing x^{108}

Figure 1.1

The above method of computing x^{108} required only 9 multiplications, as opposed to the 107 multiplications required by the naïve method. For a general positive integer n , the left-to-right binary method for computing requires between $\log_2 n$ and $2\log_2 n$ multiplications (see Exercise 1.1). For large n , the difference between the $n - 1$ multiplications required by *NaivePowers* and the at most $2\log_2 n$ multiplications required by the left-to-right binary method is dramatic indeed. We illustrate the dramatic difference between n and $\log_2 n$ in Figure 1.2. For example, 10^{83} is estimated to be more than the number of atoms in the known universe. However, $\log_2 10^{83}$ is smaller than 276.

$n = 2^m$	$m = \log_2 n$	$n - 1$
1	0	0
2	1	1
16	4	15
128	7	127
1,024	10	1,023
1,048,576	20	1,048,575
562,949,953,421,312	49	562,949,953,421,311

Table of values of $\log_2 n$ versus $n - 1$

Figure 1.2

The left-to-right binary exponentiation method illustrates nicely the fact that the simplest algorithm for solving a problem is often much more inefficient than a more clever but perhaps more complicated algorithm. We state this as an important key fact for algorithm design.

Key Fact

The simplest algorithm for solving a problem is many times not the most efficient. Therefore, when designing an algorithm, don't just settle for any algorithm that works.

Remarks

1. The binary method for exponentiation does not always yield the minimum number of multiplications. For example, computing x^{15} by the binary method requires 6 multiplications. However, it can be done using only 5 multiplications (See exercise 1.3).

2. The binary method allows computers to perform the multiplications required to compute x^n where n is an integer with hundreds of binary digits. However, for such values of n , the successive powers of x being computed are growing exponentially, and will quickly exceed the storage capacity of any conceivable computer that will ever be built. In practice, such as in internet security communication protocols, such as RSA, which involve computing x^n for n having hundreds of digits, the exponential growth in the size of the successive powers is avoided by always reducing the powers modulo some fixed integer p at each stage (called *modular exponentiation*).

There is another method for computing x^n that is called **right-to-left binary exponentiation** since it is based on scanning of the binary expansion of n from right to left. The method was mentioned by al-Kashi in 1427, and is similar to a method used by the Egyptians for multiplication as early as 2000BC. The algorithm is directly based on the law of exponents $x^{y+z} = x^y x^z$. Consider the binary expansion of n ,

$$n = 2^m + b_{m-1}2^{m-1} + b_{m-2}2^{m-2} + \dots + b_0.$$

Letting $z = b_{m-1}2^{m-1} + b_{m-2}2^{m-2} + \dots + b_0$, we have $n = 2^m + z$ and $x^n = x^{2^m+z} = x^{2^m}x^z$.

Note (by the extended law of exponents) that x^z is the product of all terms x^{2^i} as i runs from 0 to $m - 1$ and $b_i \neq 0$. Hence, to compute x^n , we compute x^{2^m} by initializing a variable *Pow* to x , and scanning the binary expansion of n right-to-left from the leftmost digit, squaring *Pow* at each digit position. Clearly, this will result in *Pow* having the value x^{2^m} at the end of the scan. We also initialize a variable *AccumPowers* to one, and during the scan if we encounter a one at digit position i (corresponding to $2^i - 1$), we multiply *AccumPowers* by the current value $x^{2^{i-1}}$ of *Pow* (i.e., the value before *Pow* is squared). Hence, the value of *AccumPowers* is then the product of all terms x^{2^j} as j runs from 0 to $i - 1$ and $b_j \neq 0$. In particular, at the conclusion of the scan (ending at the digit position m corresponding to 2^{m-1}), *AccumPowers* has the value x^z . Finally, the value x^n is computed by taking the product of *Pow* and *AccumPowers*.

```
function Right-to-Left Binary Method for computing  $x^n$ 
Input:  $x$  (a real number),  $n$  (a positive integer)
Output:  $x^n$ 
    Compute the binary representation of  $n$ 
     $Pow \leftarrow x$ 
     $AccumPowers \leftarrow 1$ 
    Scan binary representation from right to left omitting the last (most significant) binary digit
    if 1 is encountered
         $AccumPowers \leftarrow Pow * AccumPowers$ 
    endif
     $Pow \leftarrow Pow * Pow$ 
    return( $Pow * AccumPowers$ )
end Right-to-Left Binary Method
```

1.1.2 The Euclidean Algorithm

One of the oldest problems in number theory is to determine the *greatest common divisor* $\gcd(a,b)$ of two positive integers a and b . The greatest common divisor of a and b is the largest positive integer k that divides both a and b with no remainder. The problem of calculating $\gcd(a,b)$ was already known to the ancient Greek mathematicians. A naive algorithm computes the prime factorization of a and b and collects common prime powers whose product is then equal to $\gcd(a,b)$. However, for large a and b , computing the prime factorizations is very time consuming, even on today's fastest computers. A more efficient algorithm was published in *Euclid's Elements* (c. 300 B.C.). Euclid's algorithm is a refinement of an algorithm that was known 200 years earlier. This earlier algorithm is based on the observation that for $a \geq b$, an integer divides both a and b if, and only if, it divides $a - b$ and b . Thus,

$$\gcd(a,b) = \gcd(a-b, b), \quad a \geq b, \quad \gcd(a,a) = a. \quad (1.1.1)$$

Formula (1.1.1) yields the following algorithm for computing $\gcd(a,b)$.

```

function NaiveGCD( $a,b$ )
Input:  $a,b$  (two positive integers)
Output:  $\gcd(a,b)$  (the greatest common divisor of  $a$  and  $b$ )
    while  $a \neq b$  do
        if  $a > b$  then
             $a \leftarrow a - b$ 
        else
             $b \leftarrow b - a$ 
        endif
    endwhile
    return( $a$ )
end GCDNaive

```

After each iteration of the while loop in *NaiveGCD*, the larger the previous values of a and b is replaced by a strictly smaller positive number. Hence, *NaiveGCD* eventually terminates, having calculated the gcd of the original a and b .

Euclid's gcd algorithm refines the algorithm *NaiveGCD* by utilizing the fact that if $a > b$ and $a - b$ is still greater than b , then $a - b$ in turn is replaced by $a - 2b$, and so forth. Hence if a is not a multiple of b , then a is eventually replaced by $r = a - qb$, where r is the remainder when a is divided by b . Thus, all these successive subtractions can be replaced by the single invocation $a \bmod b$, where **mod** is the built-in function defined by

$$a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor, \quad a \text{ and } b \text{ integers}, \quad b \neq 0,$$

where $\lfloor x \rfloor$ denotes the largest integer less than or equal to x .

For example, when calculating $\gcd(108,8)$, the thirteen subtractions $108 - 8$, $100 - 8$, $92 - 8$, . . . , $12 - 8$ executed by the algorithm *NaiveGCD* can be replaced by the single calculation $108 \bmod 8 = 4$.

The preceding discussion leads to an algorithm based on the following formula:

$$\gcd(a,b) = \gcd(b, a \bmod b). \quad (1.1.2)$$

Note that when a is a multiple of b , $\gcd(a,b) = b$, and $a \bmod b = 0$. So the usual convention $\gcd(b,0) = b$ shows that (1.1.2) remains valid when a is a multiple of b .

Euclid's description of the gcd algorithm based on (1.1.2) was complicated by the fact that the algebraic concept of zero was not yet formalized. The following is a modern version of Euclid's algorithm.

```
function EuclidGCD(a,b)
Input: a,b (two nonnegative integers)
Output: gcd(a,b) (the greatest common divisor of a and b)
    while b ≠ 0 do
        Remainder ← a mod b
        a ← b
        b ← Remainder
    endwhile
    return(a)
end EuclidGCD
```

We illustrate *EuclidGCD* for input $a = 10724$, $b = 864$. We have

$$\gcd(10724, 864) = \gcd(864, 356) = \gcd(356, 152) = \gcd(152, 52) = \gcd(52, 48) = \gcd(48, 4) = \gcd(4, 0) = 4.$$

The problem of computing $\gcd(a,b)$ has very important applications to modern computing, particularly in as it occurs in cryptography and commonly used data security systems (see Chapter 11). It turns out that the **while** loop of *EuclidGCD* never executes more than (roughly) $\log_2(\max(a,b))$ times, so that the algorithm can be executed rapidly even for integers a and b having hundreds of digits.

1.1.3 Babylonian Square Roots

Another mathematical problem gained special significance in the sixth century B.C. when the Pythagorean school of geometers made the startling discovery that the length of the hypotenuse of a right triangle with legs both equal to 1 cannot be expressed as the ratio of two integers. This conclusion is equivalent to saying that $\sqrt{2}$ is not a rational number and therefore its decimal expansion can never be completely calculated. Long before this discovery of irrational numbers people were interested in calculating the square root of a given positive number a to any desired degree of accuracy. A square root algorithm was already known to the Babylonians by 1500 B.C. and is perhaps the first nontrivial mathematical algorithm.

The Babylonian method for calculating \sqrt{a} is based on averaging two points on either side of \sqrt{a} . The Babylonians may have discovered this algorithm by considering the problem of laying out a square plot of a given area. For example, consider an area of 5. As a first approximation, they may have considered a rectangular plot of dimensions 1 by 5. If they replaced one dimension by the average of the previous two dimensions, they would obtain a “more square”

plot of dimension 3 by 5/3. If they next replaced one of the new dimensions by the average of 3 and 5/3, the dimensions of the plot would be 7/3 by 5/(7/3) (roughly 2.33 by 2.14). More repetitions of this technique lead to plots having sides that are better and better approximations to $\sqrt{5}$ (see Figure 1.3).

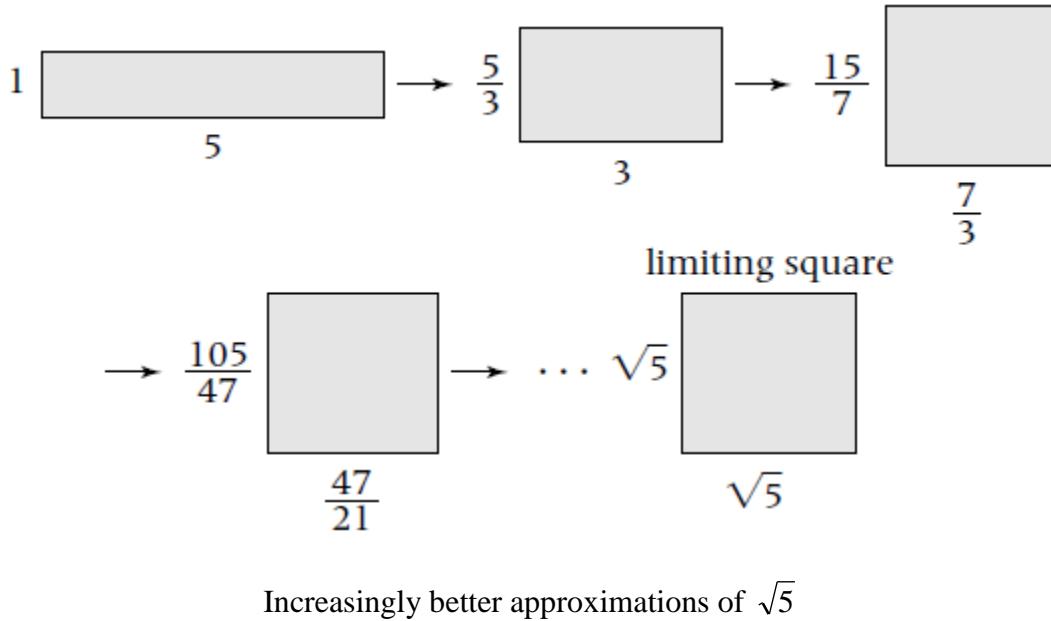


Figure 1.3

We can use the Babylonian square root algorithm to calculate the square root of any positive number a . Start with an initial guess $x = x_1$ for \sqrt{a} ; any guess will do, but a good initial guess leads to more rapid convergence. We calculate successive approximations x_2, x_3, \dots to \sqrt{a} using the formula

$$x_i = \frac{x_{i-1} + (a/x_{i-1})}{2}, \quad i = 2, 3, \dots$$

We write the Babylonian square root algorithm as a function whose input parameters are the number a and a positive real number *error* measuring the desired accuracy for computing \sqrt{a} . For simplicity, we use a as our initial approximation to \sqrt{a} .

```

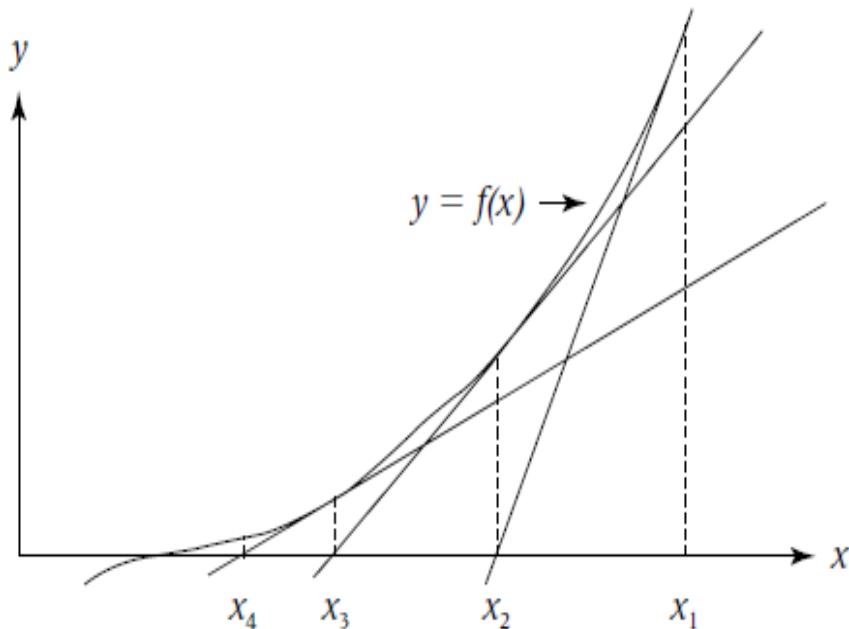
function BabylonianSQRT(a,error)
Input: a (a positive number), error (a positive real number)
Output:  $\sqrt{a}$  accurate to within error
    x  $\leftarrow a$ 
    while  $|x - a/x| > error$  do
        x  $\leftarrow (x + a/x)/2$ 
    endwhile
    return(x)
end BabylonianSQRT

```

Finding square roots is a special instance of the problem of determining the (approximate) roots of a polynomial (note that \sqrt{a} is the positive root of the polynomial $x^2 - a$). More generally, suppose $f(x)$ is a real-valued function, and we wish to find a value of x (called a *zero of f*) such that $f(x) = 0$. If f is a continuous function, then the intermediate value theorem of calculus guarantees that a zero occurs in any closed interval $[a, b]$ where $f(a)$ and $f(b)$ have opposite signs. An algorithm (called the *bisection method*) for determining a zero of f proceeds by bisecting such an interval $[a, b]$ in half, then narrowing the search for a zero to one of the two subintervals where a sign change of f occurs. By repeating this process n times, an approximation to a zero is computed that is no more than $(b - a)2^n$ from an actual zero. We leave the pseudocode for the bisection method to the exercises.

For the case when $f(x)$ is a differentiable function, a more efficient method for finding zeros of f was developed by Sir Isaac Newton in the seventeenth century. Newton's method is based on constructing the tangent line to the graph of f at an initial guess of a zero x_1 . The point x_2 where this tangent line crosses the x axis is taken as the second approximation to a zero of f . This process is then repeated at x_2 , yielding a third approximation x_3 (see Figure 1.4). Successive iterations of yield points x_2, x_3, \dots given by the formula (see Exercise 1.16)

$$x_i = x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})}, \quad i = 2, 3, \dots \quad (1.1.3)$$



Newton's method for finding a zero of a differentiable function f

Figure 1.4

Curiously, when applied to the polynomial $x^2 - a$ Newton's method yields exactly the Babylonian square root algorithm. In general, certain conditions need to be imposed on the

function f and starting point x_1 to guarantee that the points x_i given by (1.1.3) actually converge to a zero.

1.1.4 Evaluating Polynomials

A basic problem in mathematics is the problem of evaluating a polynomial $p(x) = a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$ at a particular value of v of x . The most straightforward solution to the problem would be to compute each term $a_i x^i$ independently, $i = 1, \dots, n$, and sum the individual terms. However, when computing each power v^i , $i = 1, \dots, n$, it is more efficient to obtain v^i by multiplying the already calculated v^{i-1} by v . This simple observation leads to the following algorithm.

```
function PolyEval( $a[0:n], v$ )
Input:  $a[0:n]$  (an array of real numbers),  $v$  (a real number)
Output: the value of the polynomial  $a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$  at  $x = v$ 
    Sum  $\leftarrow a[0]$ 
    Product  $\leftarrow 1$ 
    for  $i \leftarrow 1$  to  $n$  do
        Product  $\leftarrow$  Product *  $v$ 
        Sum  $\leftarrow$  Sum +  $a[i] * Product$ 
    endfor
    return(Sum)
end PolyEval
```

PolyEval clearly does $2n$ multiplications and n additions, and this might seem the best that we can do. However, there is a simple algorithm for polynomial evaluation that cuts the number of multiplications in half. This algorithm goes under the name of Horner's rule, since W. G. Horner popularized the method in 1819, but the algorithm was actually devised by Sir Isaac Newton in 1699. Horner's rule for polynomial evaluation is based on a clever parenthesizing of the polynomial. For example, a fourth-degree polynomial $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ is rewritten as

$$(((a_4 * x + a_3) * x + a_2) * x + a_1) * x + a_0.$$

This rewriting can be done for a polynomial of *any* degree n , yielding the following algorithm.

```
function HornerEval( $a[0:n], v$ )
Input:  $a[0:n]$  (an array of real numbers),  $v$  (a real number)
Output: the value of the polynomial  $a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$  at  $x = v$ 
    Sum  $\leftarrow a[n]$ 
    for  $i \leftarrow n - 1$  downto 0 do
        Sum  $\leftarrow$  Sum *  $v + a[i]$ 
    endfor
    return(Sum)
end HornerEval
```

Clearly, *HornerEval* performs n multiplications and n additions when evaluating a polynomial of degree n . It can be proved that any algorithm for evaluating an n^{th} -degree

polynomial that only uses multiplications or divisions and additions or subtractions must perform at least n multiplications or divisions and at least n additions or subtractions. Hence, *HornerEval* is an optimal algorithm for evaluating polynomials.

Algorithms for various other mathematical problems have been developed over the years. These old algorithms were mostly concerned with algebraic rules for calculating numbers and solving arithmetic equations. Indeed, the word *algorithm* itself is taken from the name of the ninth-century Arabic mathematician and astronomer Al-Khowarizmi, who wrote a famous book (*Al Jabr*, c. 825 A.D.) on the manipulation of numbers and equations. One of Al-Khowarizmi's accomplishments involved converting Jewish-calendar dates to Islamic dates. In fact, many early algorithms involved calendars and fixing the days for feasts.

1.4 Closing Remarks

We have discussed several problems having a long history and presented some algorithms for solving these problems. These examples illustrated the fact that the most straightforward and simple algorithm for solving a given problem is often not the most efficient. Designing more efficient algorithms frequently requires utilizing more sophisticated data structures than were used in this chapter. The next chapter will give a review of some standard data structures, and other data structures will be introduced as needed throughout the text.

Measuring the performance of an algorithm was discussed informally in this chapter. In the next chapter we will formalize performance measures by introducing the notions of best-case, worst-case, and average complexities of an algorithm. In order to describe the asymptotic nature of these complexities, in Chapter 3 we will formalize the notion of asymptotic growth rates of functions.

Exercises

Section 1.1 Algorithms Predating Computers

- 1.1 Trace the action of the left-to-right binary method to compute:
 - a) x^{123}
 - b) x^{64}
 - c) x^{65}
 - d) x^{711}
- 1.2 Repeat exercise 1.1 for the right-to-left binary method.
- 1.3 Show that computing x^{15} by either of the right-to-left or left-to-right binary methods requires 6 multiplications, and demonstrate that it can be done using only 5 multiplications.
- 1.4 For a general positive integer n , show that the left-to-right binary method for computing requires between $\log_2 n$ and $2\log_2 n$ multiplications.
- 1.5 Give pseudocode for implementing the left-to-right binary method when
 - a. the number is input as a binary number
 - b. the number is input as a decimal number.
- 1.6 Show that the right-to-left binary method requires the same number of multiplications as the left-to-right binary method.

- 1.7 Trace the action of the algorithm *GCD* for the following input pairs.
- (24,108)
 - (23,108)
 - (89,144)
 - (1953,1937)
- 1.8 Repeat Exercise 1.7 for the algorithm *EuclidGCD*.
- 1.9 The *least common multiple* $\text{lcm}(a,b)$ of two positive integers a and b is the smallest integer divisible by both a and b . For example, $\text{lcm}(12,20) = 60$. Give a formula for $\text{lcm}(a,b)$ in terms of $\text{gcd}(a,b)$.
- 1.10 Let a and b be positive integers and let $g = \text{gcd}(a,b)$. Then, g can be expressed as an integer linear combination of a and b , that is there exists integers s and t (not necessarily positive) such $sa + tb = g$. (We will see uses for this in Chapter 18 in connection with the RSA public key cryptosystem).
- Design and give pseudocode for an *extended Euclid's algorithm*, which inputs integers a and b and outputs integers g, s, t such that $g = sa + tb$.
 - Prove that g is the smallest integer that can be expressed as an integer linear combination of a and b .
- 1.11 Trace the action of the algorithm *BabylonianSQRT* for the following input values of a and $\text{error} = 0.001$.
- $a = 6$
 - $a = 23$
 - $a = 16$
- 1.12 A continuous function $f(x)$ such that $f(a)$ and $f(b)$ have opposite signs must have a zero (a point x such that $f(x) = 0$) in the interval (a, b) . By checking whether $f(a)$ and $f((a+b)/2)$ have opposite signs, we can determine whether the zero occurs in the subinterval $[a, (a+b)/2]$ or the subinterval $[(a+b)/2, b]$. The *bisection method* for approximating a zero of $f(x)$ is based on repeating this process until a zero is obtained to within a predetermined error. Give pseudocode for the bisection method algorithm *Bisec($f(x), a, b, \text{error}$)* for finding an approximation to a zero of a continuous function $f(x)$ in the interval $[a, b]$ accurate to within error .
- 1.13 a) Show how to use the bisection method of Exercise 1.11 to compute \sqrt{c} .
 b) For $c = 6$ and $\text{error} = 0.00001$, compare the efficiency (number of iterations) of *BabylonianSQRT* for computing \sqrt{c} with the bisection method algorithm on the interval $[1, 6]$. (You might want to write a program for this.)
- 1.14 For each of the following, do three iterations of the bisection method.
- $f(x) = x^3 - 6$, initial interval $[1, 3]$
 - $f(x) = x^3 - 26$, initial interval $[1, 3]$
 - $f(x) = 2^x - 10$, initial interval $[3, 5]$
- 1.15 Show that Newton's method reduces to *BabylonianSQRT* in the special case when $f(x) = x^2 - a$.
- 1.16 Use calculus and the description given in Figure 1.4 to derive Newton's formula (1.1.3).
- 1.17 Trace the action of Horner's rule for the polynomial

$$7x^5 - 3x^3 + 2x^2 + x - 5.$$
- 1.18 In practice, when writing a program requiring interactive input of numeric data, it is useful to check whether the user has entered any “bad characters” (that is, characters not corresponding to a digit between 0 and 9, inclusive). To prevent the program from

crashing, we can input the data as a character string, test for bad characters, and convert it to an integer if none are found. Devise an algorithm based on Horner's Rule for converting a string of alphanumeric digits to its numeric value. Assume a function *ConvertDigit* exists for converting an alphanumeric digit to its integer equivalent (for example, *ConvertDigit* ('5') = 5). Also assume that the alpha-numeric digits are input one at a time in an *online* fashion, so that the total number of digits is not known in advance.

- 1.19 To evaluate a polynomial degree n at v and $-v$, one could simply call *HornerEval* twice, involving $2n$ multiplications and $2n$ additions. Describe an algorithm using *HornerEval* that solves this particular evaluation problem using only $n + 1$ multiplications and $n + 1$ additions. A generalization of this process is the basis of the Fast Fourier Transform (see Chapter 7.)

2

Design and Analysis Fundamentals

In this chapter we introduce some of the fundamental notions involved in the design and analysis of algorithms. Recursion is one of the most fundamental design strategy tools, so we discuss this technique in some detail. In addition to formulating some general guidelines for algorithm design, we also classify some major design strategies that capture the underlying strategy for most of the algorithms discussed in this text. We give a brief discussion of these major design strategies, and will return to them in more detail in Part II.

The basic concepts underlying the analysis of algorithms are also introduced in this chapter. There are two basic considerations in measuring the performance (*complexity*) of an algorithm: its time and space requirements. We are mainly concerned with analyzing the time complexity of an algorithm. Hence, when using the term *complexity*, unless otherwise stated we will mean the time complexity.

We measure the complexity of an algorithm by identifying a basic operation and then counting how many times the algorithm performs that basic operation for an input of size n . However, the number of basic operations performed by an algorithm usually varies for inputs of a given size. Thus, we use the concepts of best-case, worst-case, and average complexity of an algorithm. Occasionally we measure the complexity of an algorithm by considering several different operations performed by the algorithm and amortizing the performance over all these operations.

Determining the complexities of an algorithm exactly is often difficult, and in practice it is usually sufficient to obtain an asymptotic approximation to these complexities. To aid our description of asymptotic behavior, in this chapter we introduce and utilize the standard notation for order and asymptotic growth rate of functions.

2.1 Guidelines for Algorithm Design

Designing an algorithm for solving a complicated task may involve finding algorithms to solve many subtasks. The standard top-down approach to algorithm design involves outlining the major steps to a solution of the given problem and then performing a process of stepwise refinement into simpler and more manageable building blocks. The key to solving the original problem then boils down to finding algorithms for building-block problems that are conceptually self-contained such as evaluating polynomials, computing powers, searching, sorting, and so forth. In this text we are concerned with finding algorithms for solving such building-block problems.

The following may be regarded as a set of general guidelines for designing and analyzing an algorithm for solving a given problem.

Guidelines for Designing an Algorithm for Solving a Given Problem

1. Identify the appropriate data structures to model the problem.
2. Once an algorithm for solving the given problem has been found, verify its correctness and analyze its performance.
3. Decide whether a search for a more efficient algorithm is worthwhile.

In the next few sections we describe some of the fundamental concepts that support the design and analysis of algorithms.

2.2 Recursion

Recursion is one of the most powerful tools in the subject of algorithms and is basis of major design strategies such as divide-and-conquer. Recursion is utilized throughout the text, both for designing algorithms and analyzing their performance. In many situations, the solution to a given problem can be expressed naturally in terms of a solution (or solutions) to a smaller or simpler input (or set of inputs) to the same problem. This expression often takes the form of a *recurrence relation*, which includes an *initial condition* stating the known solution for an initial set of inputs, and which relates the solution to a given input to an input (or set of inputs) that is closer to the known initial inputs. Exploiting such recurrence relations is the essence of *recursive* algorithmic strategy. Not only do recurrence relations yield elegant algorithms for solving many problems, recurrence relations also arise frequently in the analysis of the complexity of algorithms, as we shall see in the next chapter.

Recursive algorithms are implemented in a programming language such as C++, Java, Python, and so forth, as recursive functions or procedures. Simply stated a recursive function or procedure is one that references itself in the code. When any function or procedure is called, there is an implicit stack used to save such things as the current values stored in registers, variables, as well as a return address to be used upon resolution of the call. This overhead can be significant for recursive calls, since there may be many unresolved calls generated by the algorithm. The overhead associated with recursion can be somewhat reduced by simulating the recursion using an explicitly implemented stack in the algorithm (see Appendix B for more details on this simulation).

When there is only one recursive reference in the code, it is called *single-reference* recursion. A recursive function or procedure may contain several recursive reference statements in the code and still be single-reference recursion. For example, the following code for a function f containing the two recursive call statements (and no others)

```
if condition A then
    call  $f$  with given parameters
else
    call  $f$  with other parameters
endif
```

is still considered single-reference recursion, since only one of the two recursive references is executed by any given *current* call of the function f . The recursive function *Powers* given as our next example contains this type of construction. Recursive functions or procedures, which not only contain two or more recursive references in the code, but actually execute at least two of these references in a given *current* call, are referred to as *multiple-reference* recursion. The recursive sorting algorithms *MergeSort* and *QuickSort* described later in this chapter are examples of multiple-reference recursion.

2.2.1 Exponentiation Revisited

As an illustration of the power of recursion in formulating solutions to problems, we reconsider the problem of computing x^n for a positive integer n . In Chapter 1 we presented the algorithm *Left-to-Right Binary Method* for computing x^n . The following algorithm *Powers* is directly based on the recurrence relation

$$x^n = \begin{cases} (x * x)^{n/2} & \text{if } n \text{ is even,} \\ x * (x * x)^{(n-1)/2} & \text{otherwise.} \end{cases} \quad \text{init. cond. } x^1 = x \quad (2.1.1)$$

```
function Powers(x,n) recursive
Input: x (a real number), n (a positive integer)
Output: xn
if n = 1 then
    Pow ← x           //initial condition
else
    if even(n) then      //n even
        Pow ← Powers(x*x,n/2)
    else                  //n odd
        Pow ← x * Powers(x*x(n - 1)/2)
    endif
endif
return(Pow)
end Powers
```

For input (x,n) , similar to the left-to-right binary method, *Powers* performs between $\lfloor \log_2 n \rfloor$ and $2\lfloor \log_2 n \rfloor$ multiplications (see Exercise 2.4). Similar to many recursive algorithms, the correctness of *Powers* seems obvious, since it is based directly on the obviously correct recurrence relation. A formal proof of the correctness of *Powers* is given in the next chapter.

The recursive algorithm *Powers* considered in this section gave an elegant solution to the problem at hand. We will see many more examples of the power and elegance of recursion throughout the text. In fact, recursion is the basis of two important general design strategies, divide-and-conquer and dynamic programming.

2.3 Data Structures and Algorithm Design

The performance of an algorithm often depends on the choice of data organization. An abstract data type (ADT) is concerned with a theoretical description of the organization of the data and the operations to be performed on this data. The implementation of an ADT determines a *data structure*.

We assume that you are familiar with the basics of data structures, but we will review some of the ideas in order to facilitate the discussion of algorithm design and analysis.

Many algorithms discussed in this text use a list as one of the primary abstract data types. Operations associated with a list include, for example, insertion, deletion, and

accessing a particular element in the list. For convenience, when describing algorithms we often implement the list ADT as an array. However, in practice there are a number of situations where a linked list implementation would improve performance. The array implementation of a list has the important advantage of allowing quick and *direct* access to any element of the list. However, there are some disadvantages in using an array. For example, inserting an element at position i of an array $L[0:n - 1]$ with $L[0:m - 1]$ storing the current list requires $m - i$ array assignments. Deleting an element also requires many array assignments. Another disadvantage relates to inefficient use of space. A particular list may occupy only a small portion of an array that has been declared to handle much larger lists. A linked list avoids these disadvantages at the expense of direct access.

We assume that you are familiar with the operations of inserting and deleting elements in a linked list. We also assume that you are familiar with stacks (LIFO list) and queues (FIFO lists), and their linked list and array implementations. A review of linear data structures is given in Appendix B. While you may also be familiar with some aspects of the tree ADT, we will treat this very important ADT in some detail in Chapter 4, including the establishment of some fundamental mathematical properties of trees. Trees are the basis for a number of important data structures, such as balanced search trees for rapid key searching, heaps for maintaining priority queues, forests for maintaining disjoint sets, tries for efficient storing and retrieving of character strings, and so forth. Various other data structures will be introduced throughout the text as appropriate in the design of efficient algorithms.

Modern software design incorporates the notion of object-oriented programming, in which data and associated functions are encapsulated into a single entity called an *object*. Objects themselves are instances of a class, which implements an ADT. Many modern programming languages, such as C++, Java, C#, support classes and objects in an object-oriented programming environment. Since a high-level understanding of algorithms is emphasized in this text, for simplicity the pseudocode utilized will not explicitly mention classes and objects. For example, we represent a list of size n as $L[0:n - 1]$. We typically will implement the list as an array or linked list, without explicitly defining a list class, with associated member functions for insertion, deletion, and so forth. This enables focusing on the algorithmic issues without the encumbrance of object-oriented details.

2.4 Major Design Strategies

The study of algorithms has led to the formulation of a classification scheme, referred to as *major design strategies*, which capture the general design strategy underlying a large percentage of the algorithms in common use today. We now briefly describe each of these strategies. In Part II we will give a fuller treatment of each of the major design strategies. The Greedy method listed first typically leads to very efficient algorithms with a small, but important domain of applicability. Whereas the remaining strategies have a progressively wider applicability, the associated algorithms usually are less efficient.

THE GREEDY METHOD

The Greedy method solves optimization problems by making locally optimal choices and hoping that these choices lead to a globally optimal solution. While the Greedy method leads to a simple and efficient algorithm, care must be taken to verify its correctness. For example, consider the familiar problem of making change (US coins pennies, nickels, dimes, and quarters). We wish to make change using the smallest number of coins. The greedy solution is to first use as many coins of the largest denomination (that is, quarters) as possible. Then use as many coins of the next largest denomination, and so forth. This method always works for the US coins denominations. However obvious this might seem, suppose you did not have any nickels. Then to make 30 cents change, the greedy solution would use 6 coins, 1 quarter and 5 pennies. However, 3 dimes would do the job! The Greedy method is usually very efficient, but, as seen from the above example, its correctness in a given situation must always be proved.

DIVIDE-AND-CONQUER

Divide-and-Conquer is essentially a special case of recursion in which a given problem is divided into (usually) two or more subproblems of the exactly the same type, and the solution to the problem expressed in terms of the solutions to the subproblems. The sorting algorithm merge sort is a good example. Given a list to sort, merge sort divides the list in half, recursively sorts the first half and second half, respectively, and then calls a procedure that we have called *Merge* to merge the two sorted sublists into a sorting of the entire list. Not only are Divide-and-Conquer algorithms perhaps the most important design strategy for sequential algorithms, they are naturally parallelizable, since separate processors can be assigned the tasks of solving the subproblem. Of course, this simple parallelization will usually not give sufficient speedup, and further parallelization is required. For example, in mergesort, good speedup requires finding a parallel merging procedure that has good speedup over the sequential procedure *Merge*.

DYNAMIC PROGRAMMING

Dynamic Programming is a technique in which solutions to a problem are built up from solutions to subproblems (similar to Divide-and-Conquer), but where all the smallest subproblems to a given problem are solved before proceeding on to the next smallest subproblems. For example, a dynamic programming solution to sorting could be based on a "bottom-up" version of merge sort, where the two-element sublists $L[0:1]$, $L[2:3]$, ..., $L[n - 2:n - 1]$ are sorted first, then the four-element sublists $L[0:3]$, $L[4:7]$, ..., $L[n - 4:n - 1]$ are sorted (again, using *Merge*), and so forth. (Note by way of contrast that the recursive version of merge sort will sort the entire first half of the sublist before proceeding to the second half of the sublist.) Dynamic programming is usually applied to situations where the problem is to optimize an objective function, and where the optimal solution to a problem must be built up from optimal solutions to subproblems. Its efficiency results by eliminating suboptimal subproblems when moving up to larger subproblems.

BACKTRACKING AND BRANCH-AND-BOUND

Backtracking and Branch-and-Bound refer to strategies to solve problems that have associated state-space trees. Backtracking searches the state-space tree using depth-first search, whereas Branch-and-Bound uses breadth-first search. These search strategies have wide applicability, and apply to most problems whose solution depends on making a series of decisions. The state-space tree for the problem simply models all possible decisions that can be made at each stage. For example, the children of a node in a state-space tree modeling a board game such as checkers or chess consist of all legal moves available to the player whose move is represented by the node. As another example, consider the famous Traveling Salesman Problem (TSP) in which there are n cities, and a salesman, starting from a given city, wishes to travel to all $n - 1$ remaining cities and return in such a way as to minimize the total distance traveled over all such tours. With the starting city at the root, the children of a given node consist of all cities not yet visited by the partial tour represented by the given node (cities already visited).

The state-space tree is usually quite large, so that examining each node in the tree is not usually feasible. For example, the state-space tree for TSP has $(n - 1)!$ nodes. However, by using appropriate bounding functions, the searches are often efficient due to cutting off large subtrees of the state-space tree when it is determined that no solution can lie in these subtrees. For example, in TSP one could keep track of the best tour so far generated in searching the state-space tree, and cut-off any partial tour whose distance already exceeds the distance of this currently best tour found.

2.5 Analyzing Algorithm Performance

When you can measure what you are speaking about and can express it in numbers, you know something about it. But when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind.

—Lord Kelvin

To analyze the complexity of an algorithm, we usually identify a basic operation and count how many times an algorithm performs this basic operation. Analysis based on a suitably chosen basic operation yields measurements that are proportional to actual run time behavior exhibited when running the algorithm on various computers, so that the analysis is not dependent on a particular computer. Basic operations for some sample algorithms are shown in Figure 2.1.

Problem	Input of size n	Basic operation
Searching a list	lists with n elements	comparison
Sorting a list	lists with n elements	comparison
Multiplying two matrices	two n -by- n matrices	multiplication
Prime factorization	n -digit number	division
Evaluating a polynomial	polynomial of degree n	multiplication
Traversing a tree	tree with n nodes	accessing a node
Towers of Hanoi	n disks	moving a disk

Algorithm complexity as a function of input size n

Figure 2.1

We measure the complexity of an algorithm as a function of the *input size* n to the algorithm. For example, when searching or sorting a list, the input size is the number of elements n in the list; when evaluating a polynomial, the input size is either the degree of the polynomial or the number of nonzero coefficients in the polynomial; when multiplying two square n -by- n matrices, the input size is n ; when testing whether an integer is a prime, the input size is the number of digits of the integer; when traversing a tree, the input size is the number of nodes in the tree; and so forth (see Figure 2.1).

Be careful about what you take for the measure of input size. For example, consider the problem of computing x^p for an arbitrary positive integer p . The naive algorithm takes $p - 1$ multiplications, so that if you take $n = p$ as the input size, you would have an algorithm that performs $n - 1$ multiplications, which is linear in the input size. However, since $n = p$ is a single number (not the size of an aggregate such as a list), this is a deceptive measure of the size of the input. For example in modern security schemes today, such as RSA, the power x^p can involve a power p having hundreds of (binary) digits. Then, the naïve algorithm performing $p - 1$ multiplications would not complete in real time, even for a 64 digit number. From the point of view of storing such a p , it is a relatively small number (for example, a 128-digit binary number requires only 128 bits). However, to perform $p - 1 \geq 2^{127} - 1$ multiplications would take hundreds of years on today's fastest computers.

When analyzing the complexity of an algorithm computing x^p , what is typically taken as the measure of the input size is the number n of digits of p in its base two representation, that is, n is approximately equal to $\log_2 p$. The inefficiency of the naive method then becomes apparent, since the number of multiplications performed is approximately equal to 2^n , which is exponential in the input size.

For some algorithms the number of basic operations performed is the same for any input of size n . However, for many algorithms the number of basic operations performed can differ for two inputs of the same size. For the latter algorithms, we talk about *best-case*, *worst-case* and *average* complexities as functions of the input size n . The formal definitions of these complexities require the specification of the set of inputs of size n to an algorithm. An *input* I to an algorithm is the data, such as numbers, character strings, and records, on which the operations of the algorithm are performed. Let \mathcal{I}_n denote the set of all inputs of size n to an algorithm. The second column of Figure 2.1 shows what is typically considered to be \mathcal{I}_n for various types of problems and their associated algorithms. For $I \in \mathcal{I}_n$ let $\tau(I)$ denote the number of basic operations that are performed when the algorithm is executed with input I .

Definition 2.5.1 Best-Case Complexity

The *best-case complexity* of an algorithm is the function $B(n)$ such that $B(n)$ equals the *minimum* value of $\tau(I)$, where I varies over all inputs of size n . That is,

$$B(n) = \min\{\tau(I) \mid I \in \mathcal{I}_n\} \tag{2.5.1}$$

Far more important than $B(n)$ is the worst-case complexity $W(n)$.

Definition 2.5.2 Worst-Case Complexity

The *worst-case complexity* of an algorithm is the function $W(n)$ such that $W(n)$ equals the *maximum* value of $\tau(I)$, where I varies over all inputs of size n . That is,

$$W(n) = \max \{ \tau(I) \mid I \in \mathcal{I}_n \} \quad (2.5.2)$$

A third important complexity measure of an algorithm is its average complexity $A(n)$, which depends not only on the input size n , but also on the probability distribution on the set of all inputs of size n . More precisely, if $p: \mathcal{I}_n \rightarrow [0,1]$ is a probability function defined on the input space \mathcal{I}_n , then

$$A(n) = \sum_{I \in \mathcal{I}_n} \tau(I) p(I) = E[\tau] \quad (2.5.3)$$

Typically we assume that each input $I \in \mathcal{I}_n$ is equally likely, so that computing $A(n)$ is the usual notion of summing up all the values of $\tau(I)$, $I \in \mathcal{I}_n$ and then dividing this sum by the cardinality of \mathcal{I}_n . The formal discussion of $A(n)$ is postponed until Chapter 3, since this discussion proceeds more naturally when we have established various aspects of the asymptotic behavior of functions. It is in the context of average behavior that the best-case complexity $B(n)$ is useful as a lower bound for $A(n)$. Note also that

$$B(n) \leq A(n) \leq W(n).$$

Remark

In all of the problems mentioned so far, the input size was a function of a single variable n . However, sometimes the input size is most naturally a function of more than one variable. For example, suppose we want to find an efficient sorting algorithm for lists of size n with repeated elements. The input size is then a function of n and the number m of distinct elements in the list, where $1 \leq m \leq n$ (see *BingoSort* in the exercises). In such cases best-case, worst-case and average complexities are functions of the two variables n and m . Other examples arise in the study of graphs (see Chapter 5) where in input size is often taken as a function of the number n of vertices of a graph, and the number m of edges in the graph.

2.6 Design and Analysis of Some Basic Comparison-Based List Algorithms

A *comparison-based* algorithm for searching, finding the maximum and minimum values, or sorting a list is based on making comparison involving list elements and then making decisions based on these comparisons. In particular, comparison-based algorithms make no a priori assumption about the nature of the list elements, other than knowing their relative order. For comparison-based algorithms we can often reduce or transform the sample space to a simpler sample space without affecting the average complexity. For example, the lists $(\sqrt{5}, 1.3, 4, 2)$, $(108, 23, 123, 55)$, $(\text{Mary}, \text{Ann}, \text{Pete}, \text{Joe})$, and $(30.2, \pi, 111.23, 12)$ of size 4 can all be regarded as having the same ordering (third largest, first largest, fourth largest, second largest) as the permutation $(3, 1, 4, 2)$. In

particular, they each require the same number of comparisons when input to a comparison-based sorting algorithm before they are put into increasing order. For this reason, we may reduce our input space for comparison-based sorting algorithms to the set of all permutations on the n integers $1, 2, \dots, n$. This amounts to making two lists in \mathcal{I}_n equivalent if they have the same ordering.

To illustrate the computation of $B(n)$ of $W(n)$, we consider some basic comparison-based algorithms for searching, finding a maximum element and sorting lists of size n . Searching lists is one of the most common tasks performed by computers. If no preconditioning (such as sorting) of the list is assumed, then there is no better algorithm than linear search, which is based on a sequential (linear) scan of the list. On the other hand, for sorted lists there are much more efficient searching algorithms such as binary search.

We begin with the analyses of the two algorithms linear search and binary search. Linear search and binary search are based on making comparisons between a search element and the key field of the list elements. For simplicity, in our pseudocode for linear search and binary search we do not actually identify the key field of a list element, but instead reference the entire list element.

2.6.1 Linear Search

We implement linear search as a function that returns the (first) position in a list (array of size n) $L[0:n - 1]$ where a search element X occurs, or returns -1 if X is not in the list.

```
function LinearSearch ( $L[0:n - 1], X$ )
Input:  $L[0:n - 1]$  (a list of size  $n$ ),  $X$  (a search item)
Output: returns index of first occurrence of  $X$  in the list, or  $-1$  if  $X$  is not in the list
    for  $i \leftarrow 0$  to  $n - 1$  do
        if  $X = L[i]$  then
            return( $i$ )
        endif
    endfor
    return( $-1$ )
end LinearSearch
```

The basic operation of *LinearSearch* is the comparison of the search element to a list element. Clearly, *LinearSearch* performs only one comparison when the input X is the first element in the list, so that the best-case complexity is $B(n) = 1$. The most comparisons are performed when X is not in the list, or when X occurs in the last position only. Thus, the worst-case complexity of *LinearSearch* is $W(n) = n$.

2.6.2 Binary Search

LinearSearch assumes nothing about the order of the elements in the list; in fact, it is an optimal algorithm when no special order is assumed. However, *LinearSearch* is not the algorithm to use when searching ordered lists, at least when direct access to each list element is possible (as with an array implementation of the list). For example, if you are looking up the word *riddle* in a dictionary, and you initially open the dictionary to the page containing the word *middle*, then you know that you only need search for the word

in the pages that follow. Similarly, if you were looking up the word *fiddle* instead of *riddle*, then you would only need to search for the word in the preceding pages. This simple observation is the basis of *BinarySearch*.

The idea behind binary search is to successively cut in half the range of indices in the list where the search element X might be found. We assume that the list is sorted in increasing order. By comparing X with the element $L[mid]$ in the middle of the list, we can determine whether X might be found in the first half of the list or the second half. We have three possibilities:

$$\begin{array}{ll} X = L[mid], & X \text{ is found,} \\ X < L[mid], & \text{search for } X \text{ in } L[0:mid-1], \\ X > L[mid], & \text{search for } X \text{ in } L[mid+1:n-1]. \end{array}$$

This process is then repeated (if necessary) for the relevant “half list.” Thus, the number of elements in a sublist where X might be found is being (roughly) cut in half for each repetition. When cutting a sublist $L[low:high]$ in half, if the size of the sublist is even, then we take the midpoint index to be the smaller of the two middle indices, so that $mid = \lfloor (low + high)/2 \rfloor$. The following pseudocode implements binary search as a function with the same parameters and output as *LinearSearch*, except that the list is assumed to be sorted in increasing order.

```
function BinarySearch ( $L[0:n-1]$ ,  $low$ ,  $high$ ,  $X$ ) recursive
Input:  $L[0:n-1]$  (an array of  $n$  list elements, sorted in increasing order)
         $X$  (a search item)
Output: returns an index of an occurrence of  $X$  in the sublist  $L[low, high]$ , or -1 if  $X$  is
        not in the sublist
if  $low > high$  then return(-1) endif
     $mid \leftarrow \lfloor (low + high)/2 \rfloor$ 
case
    : $X = L[mid]$ : return( $mid$ )
    : $X < L[mid]$ : return(BinarySearch( $L[0:n-1]$ ,  $low$ ,  $mid-1$ ,  $X$ ))
    :otherwise: return(BinarySearch( $L[0:n-1]$ ,  $mid+1$ ,  $high$ ,  $X$ ))
endcase
end BinarySearch
```

Note that when searching for X in the entire list $L[0:n-1]$, we call *BinarySearch* with $low = 0$ and $high = n-1$. It is this situation that we use when measuring the complexity of *BinarySearch*. Just as with *LinearSearch*, we use comparison of X to a list element as our basic operation when analyzing *BinarySearch*. The best-case complexity of *BinarySearch* is 1, which occurs when X is found in the midpoint position $\lfloor (n-1)/2 \rfloor$ of $L[0:n-1]$. The worst-case complexity is equal to twice the longest string of midpoints (values of mid) ever generated by the algorithm for an input X . In particular, if we assume that $n = 2^k - 1$ for some positive integer k , then such a string is generated by searching for $X = L[0]$. We then compare X successively to the midpoints $2^{k-1}-1, 2^{k-2}-1, \dots, 0$, so that this longest string has length k . To express k in terms of n , we note that $n+1 = 2^k$, so that we have $k = \log_2(n+1)$. We leave it as an exercise to verify that for any n , the length of the longest string of midpoints ever generated is $\lceil \log_2(n+1) \rceil$, so that $W(n) = 2 \lceil \log_2(n+1) \rceil$.

2.6.3 Interpolation Search

While *BinarySearch* assumes that the list $L[0:n - 1]$ is in increasing order, it always compares the search element X to the midpoint index entry in the current sublist $L[low:high]$ thereby ignoring the fact that the value of X might suggest a better place to look. Interpolation search computes an index value in the range between low and $high$ that on average is rather more likely to be nearer to where X might occur than the midpoint index. By way of motivation, consider the problem of looking up the word *algorithm* in the dictionary. It certainly would be better to open up the dictionary to a page closer to the beginning of the dictionary than the middle page.

Interpolation search computes the index i in the current range $low:high$ for comparing $L[i]$ to X by making the assumption that the values in the original list $L[0:n - 1]$ are not only increasing, but lie approximately along a straight line joining the points $(0,L[0])$ to $(n - 1,L[n - 1])$ (we are interpreting the list values in L as numbers, which can always be assumed with proper conversions). This assumption of the “linearity” of the data is essential to the efficiency of interpolation search, but not its correctness.

Unfortunately, interpolation search makes n comparisons in the worst case (see Exercise 2.25). However, under suitable assumptions of randomness for the elements of the list $L[0:n - 1]$, it can be shown that the average performance $A(n)$ of interpolation search is approximately $\log_2(\log_2 n)$, a very slowing growing function indeed (see Exercise 2.27). The proof of this average behavior is beyond the scope of this book.

The value for the index i used by interpolation search (instead of mid used by *BinarySearch*) is computed by simply finding the point (i,X) along the line joining $(low,L[low])$ to $(high,L[high])$ corresponding to the search element X . More precisely, i is determined from the equation

$$(X - L[low])/(i - low) = (L(high) - L(low))/(high - low). \quad (2.6.4)$$

Note that even though the entire list $L[0:n - 1]$ may not be approximately linear, a given sublist might be. Thus, recalculating the value of i for each sublist helps make interpolation search very efficient on average. We leave the pseudocode for interpolation search and its worst-case analysis to the exercises.

2.6.4 Finding the Maximum and Minimum Elements in a List

We now consider the problem of finding the maximum (or minimum) value of an element in a list $L[0:n - 1]$ of size n . Finding the maximum can be done with a variation of *LinearSearch* where we keep track and update the maximum (or minimum) value encountered as we scan the list.

```
function Max (L[0:n - 1])
Input: L[0:n - 1] (a list of size n)
Output: returns the maximum value occurring in L[0:n - 1]
    MaxValue ← L[0]
    for i ← 1 to n - 1 do
        if L[i] > MaxValue then
            MaxValue ← L[i]      //update MaxValue
        endif
```

```

endfor
return(MaxValue)
end Max

```

When analyzing the function *Max*, we choose comparison between list elements ($L[i] > \text{MaxValue}$) as our basic operation. The only other operation performed by *Max* is the updating of *MaxValue*. However, for any input list, the number of comparisons between list elements clearly dominates the number of updates of *MaxValue*, which justifies our choice of basic operation.

Note that *Max* performs $n - 1$ comparisons for any input list of size n . Thus, the best-case, worst-case, (and average complexities) of *Max* all equal $n - 1$. It is straightforward to show that any comparison-based algorithm for finding the maximum value of an element in a list of size n must perform at least $n - 1$ comparisons for any input, so that *Max* is an optimal algorithm.

An analogous algorithm *Min* can be given for finding the minimum value in a list. Sometimes it is useful to determine *both* the maximum and the minimum values in a list $L[0:n - 1]$ (thereby determining the *range* of the data in $L[0:n - 1]$). A simple algorithm for solving this problem is to successively invoke *Max* and *Min*, having best-case and worst-case complexities equal to $2n - 2$. It turns out that any algorithm for finding the maximum and minimum in a list of size n must perform at least $\lceil 3n/2 \rceil - 2$ comparisons in the worst case. The following algorithm *MaxMin* achieves this lower bound, and therefore is an *optimal* algorithm for this problem.

To facilitate our discussion of the algorithm *MaxMin*, we introduce the following procedure *M&M*, which solves the max-min problem for a list of size 2.

```

procedure M&M(A,B,MaxValue,MinValue)
Input: A, B
Output:      MaxValue, MinValue (the maximum and minimum of A and B)
if A ≥ B then
    MaxValue  $\leftarrow$  A
    MinValue  $\leftarrow$  B
else
    MaxValue  $\leftarrow$  B
    MinValue  $\leftarrow$  A
endif
end M&M

```

Procedure *MaxMin* works by pairing elements (except for one element when n is odd) in the list $L[0:n - 1]$ and computing the maximum and minimum for each pair, yielding $\lceil n/2 \rceil$ potential maxima and $\lceil n/2 \rceil$ potential minima.

```

procedure MaxMin( $L[0:n - 1]$ , $MaxValue,MinValue$ )
Input:  $L[0:n - 1]$  (array)
Output:  $MaxValue,MinValue$  (the maximum and minimum values in  $L[0:n - 1]$ )
    if even( $n$ ) then { $n$  is even }
         $M\&M(L[0],L[1],MaxValue,MinValue)$ 
        for  $i \leftarrow 2$  to  $n - 2$  by 2 do
             $M\&M(L[i],L[i + 1],b,a)$ 
            if  $a < MinValue$  then  $MinValue \leftarrow a$  endif
            if  $b > MaxValue$  then  $MaxValue \leftarrow b$  endif
        endfor
    else // $n$  is odd
         $MaxValue \leftarrow L[0]; MinValue \leftarrow L[0];$ 
        for  $i \leftarrow 1$  to  $n - 2$  by 2 do
             $M\&M(L[i],L[i + 1],b,a)$ 
            if  $a < MinValue$  then  $MinValue \leftarrow a$  endif
            if  $b > MaxValue$  then  $MaxValue \leftarrow b$  endif
        endfor
    endif
end MaxMin

```

For n even, $L[0]$ and $L[1]$ are compared, and then the (first) for loop of $MaxMin$ performs $3(n - 2)/2$ comparisons. For n odd, there is no initial comparison, and the (second) for loop performs $3(n - 1)/2$ comparisons. Thus, the best-case and worst-case complexities of $MaxMin$ for input size n are both equal to $\lceil 3n/2 \rceil - 2$.

We now design and analyze some basic comparison-based sorting algorithms. We start with the simple sorting algorithm insertion sort. While insertion sort is not efficient (in the worst case) for large lists, we shall see that it does have its uses.

2.6.5 Insertion Sort

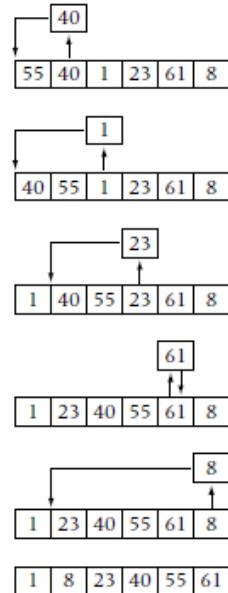
Array Implementation of Insertion Sort

Insertion sort sorts a given list $L[0:n - 1]$ by successively inserting the list element $L[i]$ into its proper place in the sorted list $L[0:i]$, $i = 1, \dots, n - 1$. It works similarly to how a card player might insert a newly dealt card into the previously dealt hand that was already put in order. One starts a scan at one end of the hand and stops at a place where the new card can be inserted and still maintain an ordered hand. This scan can start either at the low end of the hand (forward scan), or at the high end of the hand (backward scan). A card player has no reason (other than a personal preference) to prefer one scan over the other. However, when implementing insertion sort there are several reasons for preferring one scan over the other, depending on the situation.

Given the list $L[0:n - 1]$, clearly the sublist consisting of only the element $L[0]$ is a sorted list. Suppose (after possibly reindexing) we have a list L where the sublist $L[0:i - 1]$ is already sorted. We then can obtain a sorted sublist $L[0:i]$ by inserting the element $L[i]$ in its proper position. In a backward scan, we successively compare $L[i]$ with $L[i - 1]$, $L[i - 2]$, and so forth, until a list element $L[position]$ is found that is not larger than $L[i]$. $L[i]$ can then be inserted at $L[position+1]$. In a forward scan, we

successively compare $L[i]$ with $L[0]$, $L[1]$, and so forth, until a list element $L[position]$ is found that is not smaller than $L[i]$. $L[i]$ can then be inserted at $L[position]$.

Figure 2.2 demonstrates the action of the backward scan version of insertion sort for a list of size 6.



Action of *InsertionSort* (backward scan) for a list of size 6

Figure 2.2

We now give pseudocode for insertionsort using a backward scan.

```

procedure InsertionSort( $L[0:n - 1]$ )
Input:  $L[0:n - 1]$  (a list of size  $n$ )
Output:  $L[0:n - 1]$  (sorted in increasing order)
  for  $i \leftarrow 1$  to  $n - 1$  do //insert  $L[i]$  in its proper position in  $L[0:i - 1]$ 
    Current  $\leftarrow L[i]$ 
    position  $\leftarrow i - 1$ 
    while position  $\geq 1$  .and. Current  $< L[position]$  do
      //Current must precede  $L[position]$ 
       $L[position + 1] \leftarrow L[position]$  //bump up  $L[position]$ 
      position  $\leftarrow position - 1$ 
    endwhile
    //position + 1 is now the proper position for Current =  $L[i]$ 
     $L[position + 1] \leftarrow Current$ 
  endfor
end InsertionSort

```

When analyzing *InsertionSort*, we choose comparison between list elements ($Current < L[position]$) as our basic operation. For any input list of size n , the outer loop of *InsertionSort* is executed $n - 1$ times. If the input list is already sorted in increasing

order, then the inner loop is iterated only once for each iteration of the outer loop. Hence, the best-case complexity of *InsertionSort* is given by

$$B(n) = n - 1 \quad (2.6.5)$$

The worst-case complexity occurs when the inner loop performs the maximum number of comparisons for each value of the outer loop variable i . For a given i , this occurs when $L[i]$ must be compared to each element $L[i-1], L[i-2], \dots, L[0]$, so that $i-1$ comparisons are performed in the inner loop. This, in turn, occurs when the list is in strictly decreasing order. Since i varies from 1 to $n-1$, we have:

$$W(n) = 1 + 2 + \dots + (n-1) = \frac{n(n-1)}{2}. \quad (2.6.6)$$

Thus, $W(n)$ for *InsertionSort* is quadratic in the input size n .

We will show that the average complexity $A(n)$ for *InsertionSort* is about half of $W(n)$, and therefore is also quadratic in n . To see why this is true intuitively, it is reasonable that when inserting the $(i+1)$ st element $L[i]$ into its proper position in the list $L[0], \dots, L[i-1]$, on average $i/2$ comparisons will be made. Hence, it is reasonable that $A(n)$ should be about $1 + (1/2)[2 + 3 + \dots + n-1] = 1 + (1/2)[n(n-1)/2 - 1]$.

Because of its quadratic complexity, *InsertionSort* is impractical to use for sorting general large lists. However, *InsertionSort* does have five advantages:

1. *InsertionSort* works quickly on small lists. Thus, *InsertionSort* is often used as a threshold sorting conjunction with other sorting algorithms like merge sort or quick sort.
2. *InsertionSort* works quickly on large lists that are close to being sorted in the sense that no element has many larger elements occurring before it in the list. This property of *InsertionSort* makes it useful in connection with other sorts such as *ShellSort* (see the discussion in the exercises at the end of this chapter).
3. *InsertionSort* is amenable to implementation as an on-line sorting algorithm. An *on-line* sorting algorithm is one in which the entire list is not input to the algorithm in advance, but instead elements are added to the list over time. On-line sorting algorithms are required to maintain the dynamic list in sorted order.
4. *InsertionSort* is an in-place sorting algorithm. A sorting algorithm with input parameter $L[0:n-1]$ is called *in-place* if only a constant amount of memory is used (for temporary variables, loop control variable, sentinels, and so forth) in addition to that needed for L .
5. *InsertionSort* is a *stable* sorting algorithm in the sense that it maintains the relative order of repeated elements. More precisely, an algorithm that sorts a list $L[0:n-1]$ into increasing order is called stable if, given any two elements $L[i], L[j]$, with $i < j$ and $L[i] = L[j]$, the final positions i', j' of $L[i], L[j]$, respectively, satisfy $i' < j'$.

Stable algorithms are useful when we want to sort the elements in a list of records according to primary and secondary keys in the record, but where the sorts on these two

keys take place independently. By way of illustration, suppose we have already sorted the records in some list of persons alphabetically according to name. For purposes of bulk mailing, we now wish to sort the list according to the zip code key in each record.

However, within a given zip code, we wish to maintain our alphabetical order. Clearly, a stable sorting algorithm is required.

The question arises as to why we use linear scans for finding the correct position to insert the list element $L[i]$ into the already sorted list $L[0:i]$, when a binary search to find this position would drastically reduce the number of comparisons made by the algorithm. For example, the worst-case complexity of *InsertionSort* would be reduced from $n(n - 1)/2$ to approximately $n \log_2 n$. The catch is that this altered version would not reduce the number of array reassessments needed to insert $L[i]$, so that a quadratic number of array reassessments would still be made in the worst case. Thus, even though the altered *InsertionSort* is still comparison-based, the number of comparisons made would no longer be a true measure of the complexity of the algorithm.

Another drawback of the binary search version of *InsertionSort* is that some of the advantages of *InsertionSort* listed earlier would be lost. For example, the binary search version of *InsertionSort* would no longer be stable and would not necessarily work fast on large lists that are close to being sorted.

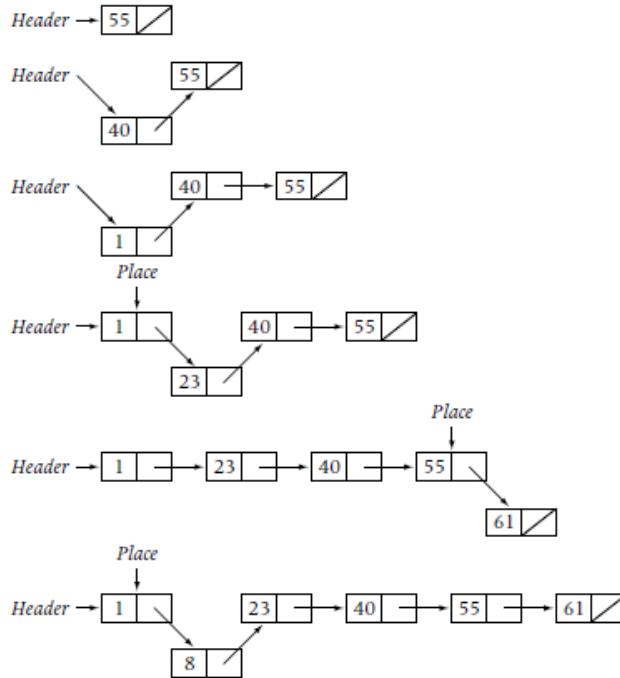
An *adjacent-key* comparison-based sorting algorithm is one in which comparisons between list elements are made only between elements that occupy *adjacent* positions in the list. These elements are interchanged if they are out of order. *InsertionSort* is essentially an adjacent-key comparison sort, since the comparison between $L[position]$ and *current* can be thought of as an assignment of *current* to $L[position]$ and then a comparison between $L[position]$ and $L[position - 1]$. For reasons of efficiency, we chose not to make the actual assignment until the correct position for insertion was determined. Another well-known adjacent-key sorting algorithm is *BubbleSort* (see Exercise 2.34).

In Chapter 3 we show that the worst-case complexity of *any* adjacent-key comparison-based sorting algorithm is at least $n(n - 1)/2$. In view of this result, *InsertionSort* actually has optimal worst-case complexity for an adjacent-key comparison-based sort. If we want to design comparison-based sorting algorithms whose worst-case complexities are smaller than $n(n - 1)/2$, we must look for design strategies that compare nonadjacent (far away) elements in the list. We give examples of algorithms utilizing the latter strategy later in this chapter.

Linked List Insertion Sort

We now implement insertion sort using a linked list rather than an array. To find the correct position to insert an element X , a forward scan of the linked list is performed using the pointer *Place*. This scan first compares X to the first element in the list. If X is not larger than this element, then the new node containing X is inserted at the beginning of the list. Otherwise, the scan continues down the list until the last element in the list smaller than X is pointed to by *Place*. The new node can then be placed immediately after the node containing this latter element.

We illustrate the linked version of insertion sort in Figure 2.3 for the same list of elements used in Figure 2.2. (The pseudocode for this linked version is left as an exercise.) In Figure 2.3, we assume that the elements are placed in nodes and inserted into the sorted linked list immediately after they are input.



Action of the linked version of *InsertionSort* for $L[0:5]$: 55 40 1 23 61 8

Figure 2.3

2.6.6 Merge Sort

Merge sort is a classical example of an algorithm based on the divide-and-conquer design strategy. Merge sort was already in use in the earliest electronic computers. In fact, it was one of the stored programs implemented by von Neumann in 1945. Given a sublist $L[low:high]$ of $L[0:n - 1]$, let x be any index between low and $high$. Let A , B , C denote the sublists $L[low:x]$, $L[x+1:high]$, $L[low:high]$, respectively. The problem of sorting C can be solved by first sorting A (recursive call), then sorting B (another recursive call), and finally calling the procedure *Merge*, which merges the sorted lists A and B to obtain a sorted list C . The following recursive code for the procedure *MergeSort* implements these steps. Note in *MergeSort* that we have chosen x to be the midpoint of low and $high$. The list $L[0:n - 1]$ is sorted by calling *MergeSort* and passing 0 to low and $n - 1$ to $high$.

```

procedure MergeSort( $L[0:n - 1], low, high$ ) recursive
Input:  $L[0:n - 1]$  (an array of  $n$  list elements),  $low, high$  (indices of  $L[0:n - 1]$ )
Output:  $L[low:high]$  (subarray sorted in increasing order)
  if  $low < high$  then
     $mid \leftarrow \lfloor (low + high)/2 \rfloor$ 
    MergeSort( $L[0:n - 1], low, mid$ )
    MergeSort( $L[0:n - 1], mid+1, high$ )
    Merge( $L[0:n - 1], low, mid, high$ )
  endif
end MergeSort

```

We now give the pseudocode for the procedure *Merge* called by *MergeSort*, where an auxiliary array *Temp* is used to aid in the merging process. *Merge* utilizes two variables *CurPos1* and *CurPos2* that refer to the current positions in the already sorted sublists $L[low:x]$ and $L[x+1:high]$, respectively. *Merge* also utilizes a third pointer *Counter*, which refers to the next available position in *Temp*. *CurPos1*, *CurPos2*, and *Counter* are initialized to *low*, $x + 1$, and *low*, respectively.

During each iteration of the while loop in *Merge*, the elements in the positions *CurPos1* and *CurPos2* are compared, and the smaller element is written to the position *Counter* in *Temp*. Then *Counter* and either *CurPos1* or *CurPos2*, depending on which one refers to the element just written to *Temp*, are incremented by one. If *CurPos1* becomes greater than *x*, then the sublist $Temp[low:Counter - 1]$ is a sorted list all of whose elements are not greater than any of the elements in the sublist $L[CurPos2:high]$. Thus, in this case we get a sorting of $L[low:high]$ by copying $Temp[low:Counter - 1]$ back onto $L[low:Counter - 1]$. On the other hand, if *CurPos2* becomes greater than *high*, then every element in the sublist $Temp[low:Counter - 1]$ is a sorted list all of whose elements are not greater than any of the elements in the sublist $L[CurPos1:x]$. Thus, in this case we get a sorting of $L[low:high]$ by first copying $L[CurPos1:x]$ over to $L[Counter:high]$, and then copying $Temp[low:Counter - 1]$ back onto $L[low:Counter - 1]$. The pseudocode for *Merge* follows.

```

procedure Merge( $L[0:n - 1], low, x, high$ )
Input:  $L[0:n - 1]$  (an array of  $n$  list elements),
         $low, x, high$  (indices of array  $L[0:n - 1]$ ; sublists  $L[low:x]$ ,  $L[x+1:high]$  are
        assumed to be sorted in increasing order)
Output:  $L[low:high]$  (sublist sorted in increasing order)

     $CurPos1 \leftarrow low$                                 //initialize pointers
     $CurPos2 \leftarrow x + 1$ 
     $Counter \leftarrow low$ 

    while  $CurPos1 \leq x$  .and.  $CurPos2 \leq high$  do      //while elements remain
        if  $L[CurPos1] \leq L[CurPos2]$  then          //to be written in both
             $Temp[Counter] \leftarrow L[CurPos1]$           //sublists, merge to Temp
             $CurPos1 \leftarrow CurPos1 + 1$ 
        else
             $Temp[Counter] \leftarrow L[CurPos2]$ 
             $CurPos2 \leftarrow CurPos2 + 1$ 
        endif
         $Counter \leftarrow Counter + 1$ 
    endwhile

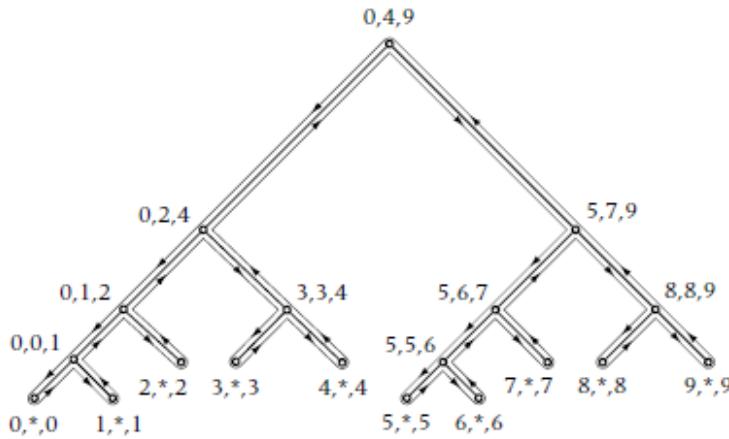
    if  $CurPos1 > x$  then
        for  $k \leftarrow low$  to  $Counter - 1$  do
             $L[k] \leftarrow Temp[k]$ 
        endfor
    else
        for  $k \leftarrow CurPos1$  to  $x$  do
             $L[k + Counter - CurPos1] \leftarrow L[k]$ 
    
```

```

endfor
for  $k \leftarrow low$  to Counter - 1 do
     $L[k] \leftarrow Temp[k]$ 
endfor
endif
end Merge

```

The tree of recursive calls to *MergeSort* is illustrated in Figure 2.4. A node in the tree is labeled by the values $low, mid, high$ involved in the call to *Merge*. (In leaf nodes, mid is not computed, as indicated by the symbol $*$.) Initially, $low = 0$ and $high = 9$. The path around the tree shown in Figure 2.4 indicates how the recursion resolves. Following this path amounts to a postorder traversal of the tree (see Chapter 4), where visiting a node corresponds to a call to *Merge*.



Recursive calls to merge sort for lists of size 10

Figure 2.4

We now analyze the complexity of *MergeSort*. We first discuss the worst-case complexity. Note that each call to *Merge* for merging two sublists of sizes m_1 and m_2 performs at most $m_1 + m_2 - 1$ comparisons. Consider the tree of recursive calls to *MergeSort* for a list of size n (see Figure 2.4). The leaf nodes do not generate calls to *Merge*, whereas each internal node generates a single call to *Merge*. At each level of the tree, the total number of comparisons made by all of the calls to *Merge* is at most n . The depth of the tree of recursive calls is $\lceil \log_2 n \rceil$ (see Exercise 2.36). It follows that *MergeSort* performs at most $n \lceil \log_2 n \rceil$ comparisons for any list of size n , so that $W(n) \leq n \lceil \log_2 n \rceil$.

Now consider $B(n)$. Each call to *Merge* for merging two sublists of sizes m_1 and m_2 performs at least $\min\{m_1, m_2\}$ comparisons. We again consider the tree of recursive calls to *MergeSort* for a list of size n . Except for the last two levels, at each level of the tree of recursive calls, the total number of comparisons made by all of the calls to *Merge* is at least $n/2$. It follows that *MergeSort* performs at least $(n/2)(\lceil \log_2 n \rceil - 1)$ comparisons for any list of size n , so that $B(n) \geq (n/2)(\lceil \log_2 n \rceil - 1)$. Thus, we have

$$(n/2)(\lceil \log_2 n \rceil - 1) \leq B(n) \leq A(n) \leq W(n) \leq n \lceil \log_2 n \rceil.$$

Since $B(n)$, $A(n)$, and $W(n)$ are all squeezed between functions which are very close to $(n/2)\log_2 n$ and $n\log_2 n$, respectively, for asymptotic analysis purposes (where we ignore the effect of positive multiplicative constants) we simply say that these quantities have $n \log n$ complexity (note that we have omitted the base in the log, since change of base formulas show that logarithm functions to two different bases differ by a constant). It turns out that this type of complexity is optimal for $A(n)$ and $W(n)$ for *any* comparison-based sorting algorithm.

By examining Figure 2.4, we can easily come up with a bottom-up version of *MergeSort*. At the bottom level of the tree, we are merging sublists of single adjacent elements in the list. However, as the path around the tree indicates, for a given input list $L[0:n - 1]$ *MergeSort* sorts the list $L[0:mid]$ before going on to any of the sublists of $L[mid+1, n - 1]$. By contrast, a bottom-up version begins by dividing the list into pairs of adjacent elements, $L[0:L[1]], L[2:L[3]], \dots$. Next, these adjacent pairs are merged yielding the sorted lists $L[0:1], L[2:3], \dots$. The process is repeated by merging the adjacent pairs of sorted two-element sublists, $L[0:1]:L[2:3], L[4:5]:L[6:7], \dots$. Continuing this process, we arrive at the root having sorted the entire list $L[0:n - 1]$.

Figure 2.5 shows a tree representing this bottom-up merging of adjacent sublists. Each node represents a call to *Merge* with the indicated values of *low, mid, high*. An asterisk * denotes those nodes where a call to *Merge* is not made. Note that the sublists and resulting tree are quite different from the sublists generated by the tree of recursive calls of *MergeSort* given in Figure 2.4. All the calls to *Merge* for a given level are completed before we go up to the next level. The pseudocode for the nonrecursive version of *MergeSort* based on Figure 2.5 is left to the exercises.

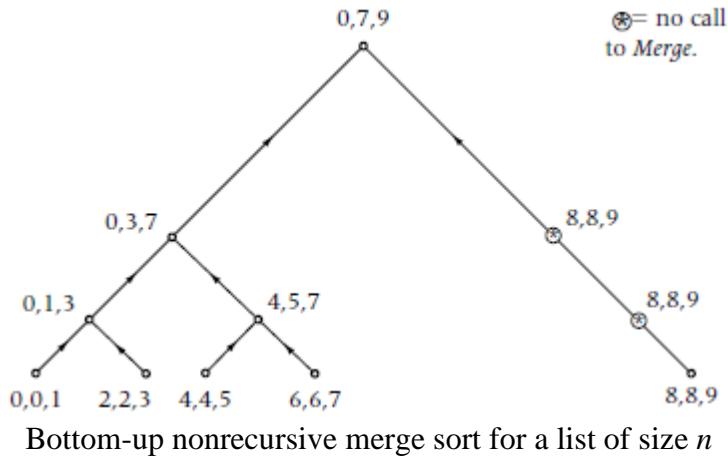


Figure 2.5

2.6.7 Quick Sort

We now discuss the comparison-based sorting algorithm quick sort, discovered by C. A. R. Hoare in 1962. Quick sort is often the sorting algorithm of choice because of its good average behavior. Quick sort, like merge sort, is based on dividing a list into two sublists (these are actually two sublists of a rearrangement of the original list), and then sorting the sublists recursively. The difference between the two sorting strategies is that merge sort does most of the work in the combine (merge) step, whereas quick sort does most of

the work in the divide step. Quick sort is also an “in place” sort, as opposed to merge sort which used an auxiliary array for merging (in-place versions of merge sort can be written, but they are complicated).

In quick sort, the division into sublists is based on rearranging the list $L[low:high]$ with respect to a suitably chosen *pivot element* x . The list $L[low:high]$ is rearranged so that every list element in $L[low:high]$ preceding x (having smaller index than the index of x) is not larger, and every element following x in $L[low:high]$ is not smaller. For example, for the list 23,55,11,17,53,4 and pivot element $x = 23$, such a rearrangement might be 17,11,4,23,55,53. Note that after such a rearrangement, the pivot element x occupies a proper position in a sorting of the list. Thus, if the sublists on either side of x are sorted recursively, then the entire list will be sorted with no need to invoke an algorithm for combining the sorted sublists.

Procedure *QuickSort* sorts $L[low:high]$ into increasing order by first calling an algorithm *Partition* that rearranges the list with respect to pivot element $x = L[low]$ as previously described. *Partition* assumes that the element $L[high+1]$ is defined and is at least as large as $L[low]$. The output parameter *position* of *Partition* returns the index where x is placed. To sort the entire list $L[0:n - 1]$, *QuickSort* would be called initially with $low = 0$ and $high = n - 1$.

```
procedure QuickSort( $L[0:n - 1]$ ,  $low$ ,  $high$ ) recursive
Input:  $L[0:n - 1]$  (an array of  $n$  list elements)
     $low$ ,  $high$  (indices of  $L[0:n - 1]$ )
    //for convenience,  $L[n]$  is assumed to have the sentinel value  $+\infty$ 
Output:  $L[low:high]$  sorted in increasing order
if  $high > low$  then
    Partition( $L[0:n - 1]$ ,  $low$ ,  $high$ , position)
    QuickSort( $L[0:n - 1]$ ,  $low$ , position - 1)
    QuickSort( $L[0:n - 1]$ , position + 1,  $high$ )
endif
end QuickSort
```

We now describe the algorithm *Partition*. *Partition* is based on the following clever interplay between two moving variables *moveright* and *moveleft*, which contain the indices of elements in L and are initialized to $low + 1$ and $high$, respectively.

```
while moveright < moveleft do
    moveright moves to the right (one index at a time) until it assumes the
    index of a list element not smaller than  $x$ , then it stops.
    moveleft moves to the left (one index at a time) until it assumes the index
    of a list element not larger than  $x$ , then it stops.
    if moveright < moveleft then
        interchange  $L[moveright]$  and  $L[moveleft]$ 
    endif
endwhile
```

To guarantee that *moveright* actually finds an element not smaller than x , we assume that $L[high + 1]$ is defined and is not smaller than $L[low]$. As commented in the pseudocode, this is arranged by introducing a sentinel value $L[n] = +\infty$. We leave it as an exercise to check that the condition $L[high + 1] \geq L[low]$ is then automatically guaranteed for all subsequent calls to *Partition* by *QuickSort*. Of course, *Partition* could be written with explicit checking that *moveright* does not run off the list. However, this checking requires additional comparisons, and we prefer to implement the preconditioning. Figure 2.6 illustrates the movement of *moveright* (*mr*) and *moveleft* (*ml*) for a sample list $L[0:6]$. The pseudocode for *Partition* follows.

```

procedure Partition( $L[0:n - 1]$ ,  $low$ ,  $high$ ,  $position$ )
Input:  $L[0:n - 1]$  (an array of  $n$  list elements)
     $low$ ,  $high$  (indices of  $L[0:n - 1]$ )
     $//L[high+1]$  is assumed defined and  $\geq L[low]$ 
Output: a rearranged sublist  $L[low:high]$  such that
     $L[i] \leq L[position]$ ,  $low \leq i \leq position$ ,
     $L[i] \geq L[position]$ ,  $position \leq i \leq high$ 
    where, originally,  $L[low] = L[position]$ 
     $position$  (the position of a proper placement of the original element  $L[low]$ 
        in the list  $L[low:high]$ )
    moveright  $\leftarrow low$ 
    moveleft  $\leftarrow high + 1$ 
     $x \leftarrow L[low]$ 
    while moveright  $< moveleft$  do
        repeat
            moveright  $\leftarrow moveright + 1$ 
        until  $L[moveright] \geq x$ 
        repeat
            moveleft  $\leftarrow moveleft - 1$ 
        until  $L[moveleft] \leq x$ 
        if moveright  $< moveleft$  then
            interchange( $L[moveright], L[moveleft]$ )
        endif
    endwhile
    position  $\leftarrow moveleft$ 
     $L[low] \leftarrow L[position]$ 
     $L[position] \leftarrow x$ 
end Partition

```

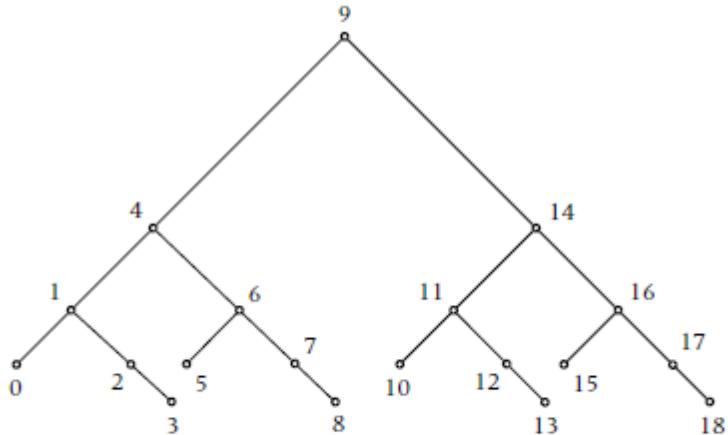
Initially:	index	0	1	2	3	4	5	6	7
	list element	23	9	23	52	15	19	47	$+\infty$
		<i>mr</i>							<i>ml</i>
Rearrange Step									
1st iteration:			19			23			
		23	9	23	52	15	19	47	$+\infty$
			<i>mr</i>				<i>ml</i>		
2nd iteration:				15	52				
		23	9	19	52	15	23	47	$+\infty$
				<i>mr</i>	<i>ml</i>				
3rd iteration:				15	52	23	47		$+\infty$
				<i>ml</i>	<i>mr</i>				
Place Step									
After completion of <i>Partition</i> :		15		23					
		23	9	19	15	52	23	47	$+\infty$
						<i>position</i> = 3			

Action of *Partition* for a sample list $L[0:6]$

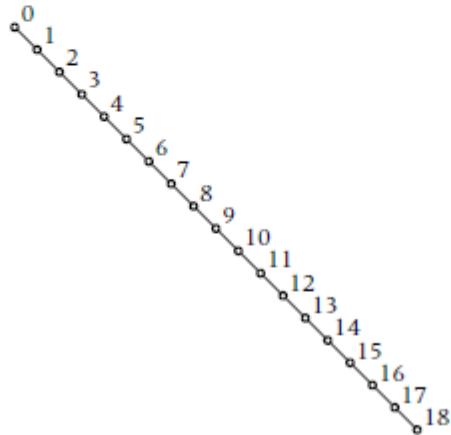
Figure 2.6

To analyze the performance of *QuickSort*, we again use list comparisons as our basic operation for analysis, where *QuickSort* is originally called with a list $L[0:n - 1]$ of distinct elements, and $L[n] = +\infty$. A call to *Partition* with $L[low:high]$ performs exactly $high - low + 2$ comparisons, and this forms the basis of our analysis. We first consider the worst-case complexity.

Unfortunately, the worst-case performance of *QuickSort* occurs for a list that is already in order. In this case the recursive calls to *QuickSort* are always with the empty list $L[low:low-1]$ and the list $L[low+1:high]$ (see Figure 2.7b). Thus, the number of comparisons performed is given by $W(n) = (n + 1) + n + \dots + 3 = (n + 2)(n + 1)/2 - 3$, so that *QuickSort* has quadratic worst-case performance. Note that a decreasing list also generates worst-case performance for *QuickSort*.



(a) Tree of recursive calls for best-case behavior of *QuickSort* for a list of size 19. Internal nodes are labeled with pivot elements in a call to *Partition*. Leaf nodes are labeled with single-element sublists that result in immediate returns. For simplicity, we do not show leaf nodes corresponding to calls to *QuickSort* with empty lists.



(b) Tree of recursive calls for worst-case behavior of *QuickSort* for a list of size 19, with nodes labeled using the same convention as in (a).

Tree of recursive calls for *QuickSort*:
(a) best-case behavior; (b) worst-case behavior

Figure 2.7

The quadratic worst-case performance is disappointing, but quick sort is popular because of its (asymptotic) $n \log n$ average behavior. We can expect this behavior since it is reasonable that on average the call to *Partition* results in dividing a given sublist into sublists of roughly equal size. In other words, for an average input, the tree of recursive calls would have a balanced nature similar to that of merge sort, so that we would have roughly $\log_2 n$ levels, with no more than $n + 1$ comparisons made by the calls to *Partition* on each level. We will prove in Chapter 3 that *QuickSort* does indeed have (asymptotic) $n \log n$ average complexity.

Remarks

For simplicity we chose the pivot element to be the first list element. A common alternative is to take the pivot element to be the median of the three elements in positions low , $(low + high)/2$, and $high$. Using this *Median of Three Rule* avoids the bad behavior exhibited by lists that are close to being sorted.

There are many different ways to design an algorithm that accomplishes the same thing as our version of *Partition*. One such alternative version is explored in Exercise 2.62.

Some improvements to *QuickSort* should be made before using it in practice. In particular, *QuickSort* as written generates $n - 1$ unresolved recursive calls (that is, $n - 1$ successive pushes of unresolved activation records) in the worst case. These $n - 1$ successive pushes will no doubt cause stack overflow when run on many computers for even modestly large values of n (that is, even for values of n for which a quadratic number of comparisons will still complete in reasonable time). Thus, it is important to rewrite *QuickSort* in order to reduce the number of unresolved recursive calls.

To reduce the number of successive recursive calls, we first note that *QuickSort* is tail recursive (see Appendix B). However, simply removing the tail recursion will result in a version that still generates $n - 1$ unresolved recursive calls for a decreasing list (however, an increasing list now generates recursive calls only with empty sublists, so that there is only one activation record on the stack generated by recursive calls at any given point in the execution for such lists). The problem is that we are always making a recursive call with the sublist to the left of the partition element, whose size is only reduced by one in the worst case. What we need to do is make the recursive call with the smaller of the two sublists on either side of the partition element, and simply redefine the relevant parameter (low or $high$) for the larger sublist and branch to the beginning of the code. This will reduce the number of unresolved recursive activation records on the stack to at most $\log_2 n$. Moreover, now both the worst cases of increasing or decreasing lists will only make recursive calls with empty sublists. We leave the code for this altered version of *QuickSort* and its analysis to the exercises.

2.7 Radix Sort

The sorting algorithms that we have considered so far are comparison-based. Recall that this means that the only assumption made about the list elements is that they can be compared as to their relative order, and no additional assumptions are made about the specific nature of the elements. In this section we give an example of a sorting algorithm that is not a comparison-based algorithm, but rather depends on the special nature of the list elements. In radix sort, we assume that the list elements are strings of a given fixed length drawn from a given (ordered) alphabet and that the ordering of the strings is the usual lexicographic ordering (that is commonly used in practice). For example, we might be sorting personnel records based on social security numbers. Then the list elements are character strings of length 9 drawn from the ten character digits 0, 1, . . . , 9. Or, we might be sorting the records based on zip codes, so that the list elements are characters strings of length 5 again drawn from 0, 1, . . . , 9.

One sorting method that is sometimes used in practice when the sorting is being done by hand is to make a pass through the list, placing the records in “buckets” based on the leading character in the string. For example, suppose we are sorting zip codes. We would

use ten buckets B_0, B_1, \dots, B_9 . We would then make a linear scan through the list, inserting the record in B_i whenever the zip code has leading digit i , $i = 0, \dots, 9$. The advantage of this method is that to complete the sort, we now just have to sort the records within each bucket. Sorting in each bucket could then proceed analogously, except we would place the records in sub-buckets based on the second leading digit. The disadvantage of this type of sort is the complication arising from the proliferation of buckets. However, in the case of binary strings, a fairly elegant recursive procedure similar to *Quicksort* can be developed (see Exercise 5.11).

The buckets don't proliferate if instead of looking first at the leading digit, we place the records in buckets based on the last (least significant) digit. We then repeat the process by extracting the list from the buckets in the order B_0, B_1, \dots, B_9 and placing the records in these same ten buckets according to the second to the last digit. After five repetitions, the records will be sorted, as shown in Figure 2.8.

Original list:
 45242 45230 45232 97432 74239 12335 43239 40122 98773 41123 61230

Pass 1 Place in buckets based on fifth (least significant) digit	Bucket 0: 45230 61230 Bucket 2: 45242 45232 97432 41022 Bucket 3: 98773 41123 Bucket 5: 12335 Bucket 9: 74239 43239
--	---

Extract list from buckets:
 45230 61230 45242 45232 97432 40122 98773 41123 12335 74239 43239

Pass 2 Place in buckets based on fourth digit	Bucket 2: 40122 41123 Bucket 3: 45230 61230 45232 97432 12335 74239 43239 Bucket 4: 45242 Bucket 7: 98773
---	--

Extract list from buckets:
 40122 41123 45230 61230 45232 97432 12335 74239 43239 45242 98773

Pass 3 Place in buckets based on third digit	Bucket 1: 40122 41123 Bucket 2: 45230 61230 45232 74239 43239 45242 Bucket 3: 12335 Bucket 4: 97432 Bucket 7: 98773
--	---

Extract list from buckets:
 40122 41123 45230 61230 45232 74239 43239 45242 12335 97432 98773

Pass 4 Place in buckets based on second digit	Bucket 0: 40122 Bucket 1: 41123 61230 Bucket 2: 12335 Bucket 3: 43239 Bucket 4: 74239 Bucket 5: 45230 45232 45242 Bucket 7: 97432 Bucket 8: 98773
---	--

Extract list from buckets:
 40122 41123 61230 12335 43239 74239 45230 45232 45242 97432 98773

Pass 5 Place in buckets based on first (most significant) digit	Bucket 1: 12335 Bucket 4: 40122 41123 43239 45230 45232 45242 Bucket 6: 61230 Bucket 7: 74239 Bucket 9: 97432 98773
--	---

Extract (sorted) list from buckets:
 12335 40122 41123 43239 45230 45232 45242 61230 74239 97432 98773

Sorting a sample list of size $n = 11$, where each element is a five-digit numerical character string, so that there are ten buckets and five passes.

Figure 2.8

We now give a high-level description of radixsort based on the procedure illustrated in Figure 2.8. In our high-level description of the procedure *RadixSort*, we assume the built-in function *Substring(string,i)*, which extracts the symbol in position i from *string*. We

also assume high-level procedures $\text{Enqueue}(Q, X)$, which enqueues the element X into the queue Q , and $\text{Dequeue}(Q, Y)$, which dequeues Q and assigns the dequeued element to Y .

```

procedure RadixSort( $L[0:n - 1], k$ )
Input:  $L[0:n - 1]$  (a list of size  $n$ , where  $L[i]$  is a string of length  $k$  over an alphabet  $s_0 < s_1 < \dots < s_m$ )
Output:  $L[0:n - 1]$  (sorted in increasing order)
for  $i \leftarrow k - 1$  downto 0 do
    //place elements in queues  $B_0, B_1, \dots, B_m$  based on position  $i$ 
    for  $j \leftarrow 0$  to  $n - 1$  do
        if  $\text{Substring}(L[j], i) = s_q$  then
             $\text{Enqueue}(B_q, L[i])$ 
        endif
    endfor
    //reassemble list  $L$  by successive dequeuing
     $C_t \leftarrow 1$ 
    for  $j \leftarrow 0$  to  $m$  do
        while (.not.  $\text{Empty}(B_j)$ ) do
             $\text{Dequeue}(B_j, Y)$ 
             $L_{C_t} \leftarrow Y$ 
             $C_t \leftarrow C_t + 1$ 
        endwhile
    endfor
    endfor
end RadixSort
```

We leave the proof of the correctness of RadixSort as an exercise. Clearly, the (best-case, worst-case, and average) complexity of RadixSort is in $\Theta(kn)$. A convenient way to implement the queues used in RadixSort is to use linked lists. The action of the first pass of RadixSort is illustrated in Figure 2.9 with this implementation. We assume that the queues are pointed to by the elements in an array $\text{Bucket}[0:9]$ of pointer variables. We also assume that we have an array $\text{Rear}[0:9]$ of pointers, where $\text{Rear}[i]$ points to the end of the queue whose first element is pointed to by $\text{Bucket}[i]$, $i = 0, \dots, 9$.

Original list given in Figure 5.04:

45242 45230 45232 97432 74239 12335 43239 41022 98773 41123 61230

Bucket[0] → [45230] → [61230] ↗ ← *Rear[0]*

Bucket[1] = null = *Rear[1]*

Bucket[2] → [45242] → [45232] → [97432] → [40122] ↗ ← *Rear[2]*

Bucket[3] → [98773] → [41123] ↗ ← *Rear[3]*

Bucket[4] = null = *Rear[4]*

Bucket[5] → [12335] ↗ ← *Rear[5]*

Bucket[6] = null = *Rear[6]*

Bucket[7] = null = *Rear[7]*

Bucket[8] = null = *Rear[8]*

Bucket[9] → [74239] → [43239] ↗ ← *Rear[9]*

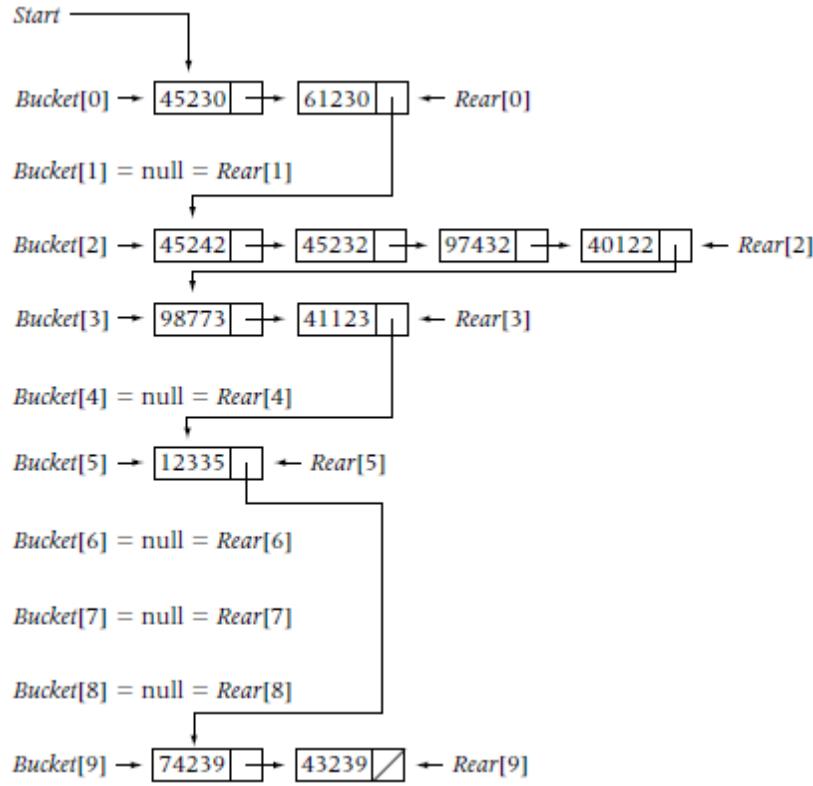
Extract list from buckets:

List → [45230] → [61230] → [45242] → ... [74239] → [43239] ↗

First pass of *RadixSort* for the sample list given in Figure 2.8, where the queues (buckets) are implemented using linked lists.

Figure 2.9

The extracting of the list from the buckets is a simple job of reassigning some pointers, as illustrated in Figure 2.10. Note that these pointer reassessments allow us to completely dequeue a given queue in one fell swoop! The reassembled list is a linked list pointed to by *Start*.



Extract list from buckets in the first pass of *RadixSort* for the sample list given in Figure 2.8, where the queues (buckets) and the reassembled list are implemented using linked lists.

Figure 2.10

In general, *Start* points to the first nonempty queue, that is, $\text{Start} = \text{Bucket}[i]$, where i is the smallest integer such that $\text{Bucket}[i] \neq \text{null}$. The reassembled linked list is then created by calling the following procedure *Reassemble*. For convenience, we write the pseudocode for *Reassemble* under the assumption that our alphabet is the ten decimal digital characters.

```

procedure Reassemble(Bucket[0:9],Rear[0:9],Start)
Input: Bucket[0:9] (an array of pointers to queues)
         Rear[0:9] (an array of pointers to end of queues)
Output: Start (a pointer to the reassembled linked list L)
Ct ← 0
while (Bucket[Ct] = null) do
    Ct ← Ct + 1
endwhile
Start ← Bucket[Ct]
SaveCt ← Ct
while(SaveCt < 9) do
    Ct ← Ct + 1

```

```

while (Bucket[Ct] = null .and. Ct ≤ 9) do
    Ct ← Ct + 1
endwhile
if Ct ≤ 9 then
    Rear[SaveCt] → Next ← Bucket[Ct]
    SaveCt ← Ct
endif
endwhile
end Reassemble

```

2.8 Closing Remarks

For most of the algorithms in this text, our analysis of the complexity consists in identifying a basic operation and then determining the best-case, worst-case, and average complexities with respect to this basic operation. Occasionally, it is important to measure the complexity of an algorithm in terms of more than one operation. This is especially true for algorithms that involve maintaining several data structures, some of which may be altered by insertion or deletion of elements during the performance of the algorithm. *Amortized* analysis of an algorithm involves computing the maximum total number of all operations on the various data structures that are performed during the course of the algorithm for an input of size n . Usually this amortized total is smaller than the number obtained by simply adding together the worst-case complexities of each individual operation for inputs of size n . Indeed, often for an input where one of the operations achieves its worst-case complexity, another of the operations might perform much less than its worst-case complexity, so that a certain trade-off occurs for any input. When designing algorithms where amortized analysis is appropriate, it then becomes a strategy to exploit this trade-off between the various operations.

In this chapter we gave an overview of some of the major issues that arise in the design and analysis of algorithms. This overview sets the stage for the material in the rest of the text. We introduced and analyzed several important classical algorithms. Some of these algorithms were truly old, such as algorithms for exponentiation. Others, such as merge sort and quick sort, date to the period following soon after the advent of the electronic computer. Occurring throughout the rest of the text are algorithms of more recent origin, including algorithms that have become important to the efficient functioning of the Internet. Part III will be devoted to graph and network algorithms having specific application such things as search engines for the Internet.

Exercises

Section 2.2 Recursion

- 2.1 Give pseudocode for a recursive function that outputs the maximum value in a list of size n .
- 2.2 Give pseudocode for a recursive function that tests whether or not an input string of size n is a palindrome (that is, reads the same backwards and forwards).
- 2.3 a) Design a recursive algorithm whose input is a decimal integer and whose output is the binary representation of the input.

- b) Design a recursive algorithm that computes the reverse of the result in part a), that is, converts a binary integer to its decimal equivalent.
- 2.4 Show that for input (x,n) , *Powers* performs between $\lfloor \log_2 n \rfloor$ and $2\lfloor \log_2 n \rfloor$ multiplications.
- 2.5 One of the most famous sequences in computer science (and nature) is the Fibonacci sequence defined by the recurrence

$$fib(n) = fib(n - 1) + fib(n - 2), \text{ init. cond. } fib(0) = 0, fib(1) = 1.$$
- Using the technique of generating functions, it can be shown that:

$$fib(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right].$$

Without using this formula, argue that

$$(1.5)^{n-2} \leq fib(n) \leq 2^{n-1}, n \geq 1.$$

- A formal proof requires induction, which will be discussed in Chapter 3.
- 2.6 Consider the following function *Fib* for computing the n^{th} Fibonacci number based directly on the recursive definition given in the previous question:

```
function Fib(n) recursive
Input: n (a nonnegative integer)
Output: the nth Fibonacci number
  if n ≤ 1 then
    return(n)
  else
    return(Fib(n - 1) + Fib(n - 2))
  endif
end Fib
```

- a) Show that *Fib* is extremely inefficient, since it performs many redundant recalculations. How many times is *fib*(*k*) computed by *Fib* when *Fib* is invoked with input *n*, $k = 0, 1, \dots, n$?
- b) Rewrite *Fib* so that it is still recursive but uses a table to avoid these redundant calculations.
- c) Rewrite *Fib* as a purely iterative function.
- 2.7 Note that for the pair (89,144) given in Exercise 1.1 the algorithms *GCD* and *EuclidGCD* performed identical calculations. This phenomenon holds for infinitely many pairs of integers. One such collection can be obtained as successive pairs in the Fibonacci sequence *fib*(*n*).
- a) Verify that (except for the last step) *GCD* and *EuclidGCD* perform identically on any input pair (*fib*(*n* - 1),*fib*(*n*)).
- b) Obtain a formula for the number of steps required by *EuclidGCD* to compute $\gcd(fib(n - 1), fib(n))$.
- 2.8 Show that the longest number of steps required by *EuclidGCD* for a pair of integers each having *m* digits or less is achieved by a suitable pair (*fib*(*n* - 1),*fib*(*n*)).

Section 2.3 Data Structures and Algorithm Design

- 2.9 Given a linked list, create a linked list with the same elements but in the reverse direction.
- 2.10 a) Give pseudocode for the push and pop operations on a stack implemented using an array.
 b) Repeat (a) for a linked list implementation.
- 2.11 A *circular queue* is implemented using a circular array $queue[0:Max - 1]$ and two pointers (indices) $Front$ and $Rear$. We view the elements $Queue[0]$, $Queue[1]$, $Queue[Max - 1]$ as positioned around a circle in a counterclockwise fashion. Initially, we have $Front = Rear = 0$. For nonempty queues, the variable $Rear$ points to the position of the element at the rear of the queue. However, $Front$ points one position counterclockwise from the element at the front of the queue. An element (in variable) x is enqueued by the two statements

$Rear \leftarrow (Rear + 1) \bmod Max$

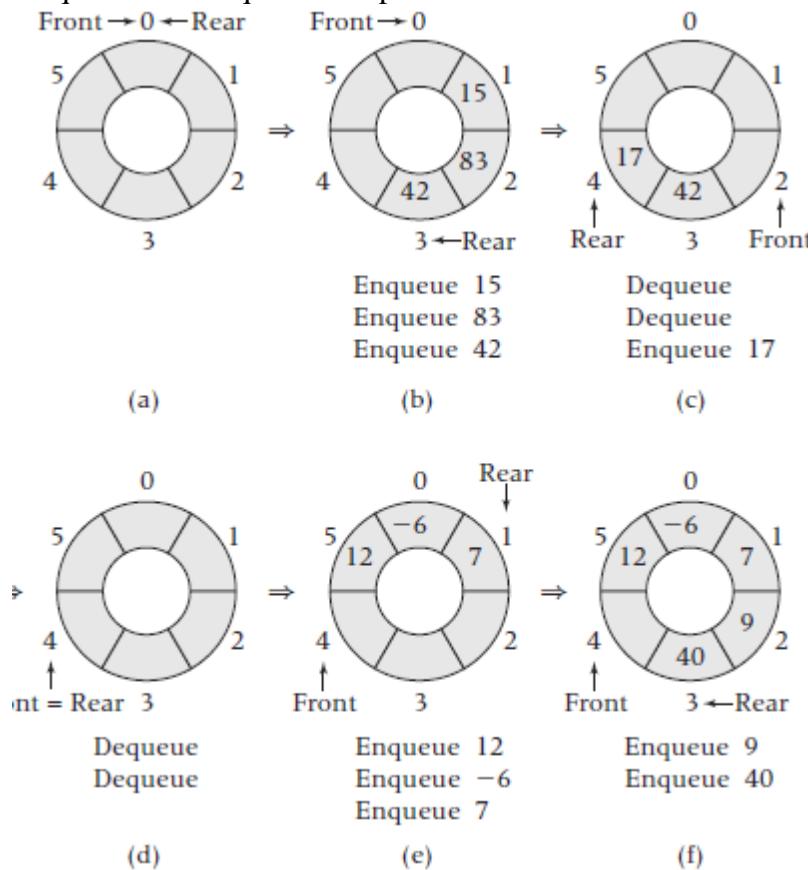
$Queue[Rear] \leftarrow x$

and an element is dequeued and assigned to x by the two statements

$Front \leftarrow (Front + 1) \bmod Max$

$x \leftarrow Queue[Front]$.

Figure 2.11 shows how the circular array $Queue$ with $Max = 6$ is updated as a given sequence of enqueues and dequeues are performed.



Circular array $Queue[0:5]$ after a sequence of enqueues and dequeues

Figure 2.11

If the queue is empty then $Rear = Front$. On the other hand, if the queue is full in the sense that the entire array $Queue$ is filled, then it is also the case that $Rear = Front$. The question then arises as to how to distinguish between these two situations. One way is to introduce a Boolean variable, which tags the queue as being empty or full. However, maintaining such a variable may lead to a noticeable increase in the computing time, since the procedures for implementing a queue are usually called repetitively. Another solution is to consider the queue full when every position of the array $Queue$ but one is filled with elements of the queue. (Of course, this wastes a single memory location in the array $Queue$, but this is hardly significant.)

- a. Give pseudocode for procedures $EnqueueCirc$ and $DequeueCirc$ for enqueueing and dequeuing an element using the latter implementation of a circular queue.
- b. Starting with an empty circular queue $Queue[0:8]$, show the final state of $Queue[0:8]$ and the values of $Rear$ and $Front$ after the following sequence of enqueues and dequeues.
enqueue 23, enqueue 108, enqueue 55, dequeue, enqueue 61,
dequeue, dequeue, enqueue 44, enqueue 21, dequeue

- 2.12 Give pseudocode for inserting and deleting into a linked list implemented using two arrays $L[0:n - 1]$ and $Next[0:n - 1]$.

Section 2.5 Analyzing Algorithm Performance

- 2.13 Design an algorithm that tests whether or not two input lists of size n have at least one element in common. Give formulas for $B(n)$ and $W(n)$ for your algorithm.

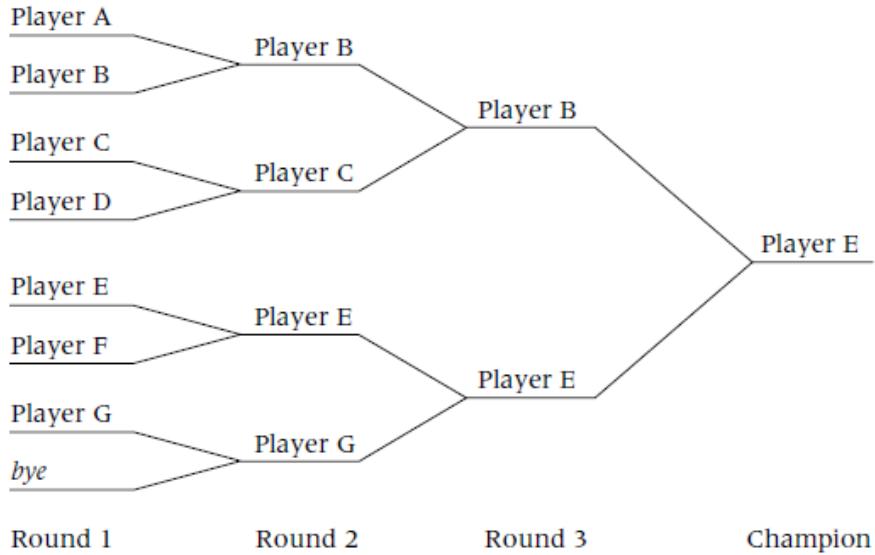
Design and analyze an algorithm that emulates a single elimination tournament (NCAA basketball, Wimbledon tennis, and so forth) in order to find the maximum element in a list of size n . For simplicity, you may assume that n is a power of two (tournaments arrange this by giving byes when n is not a power of two).

- 2.14 Design a (simple) algorithm whose input is a text string T of size n and a pattern string P of size m , and determines whether P occurs as a substring of T . Give formulas for $B(n,m)$ and $W(n,m)$.

Section 2.6 Design and Analysis of Some Basic Comparison-Based List Algorithms

- 2.15 a) Write a procedure that finds both the largest and second-largest elements in a list $L[0:n - 1]$ of size n . You should find a more efficient algorithm than simply performing two scans through the list.
b) Determine $B(n)$ and $W(n)$ for the algorithm in (a).
- 2.16 The idea behind a better (in fact optimal) algorithm for finding the largest and second largest element in a list, is to model the algorithm as a match-play golf or table tennis tournament. We regard the comparison of list elements as a golf or table tennis match between the elements (players). Assume for simplicity that $n = 2^k$. In the first round of the tournament, we pair the n players into $n/2$ matches. In the next round, the $n/2$ winners of the first round are paired into $n/4$ matches, and the $n/4$ winners advance into the third round. After $k = \log_2 n$ rounds, we

obtain the winner of the tournament. We draw such a single-elimination tournament in Figure 2.12. Note that to arrange for n to be a power of two, we gave Player G a *bye* in the first round.



A single-elimination tournament

Figure 2.12

The algorithm based on this tournament method makes $n/2 + n/4 + \dots + n/2^k = n(1/2 + 1/4 + \dots + 1/2^k) = n - 1$ comparisons in determining the largest element. The question remains as to how the algorithm can now proceed to efficiently find the runner-up (second-largest). In actual golf or table tennis tournaments, it is customary to declare the runner-up to be the player who played the winner in the last round (Player B in Figure 2.12). However, sometimes that runner-up is not really the second-best player in the tournament. (Tournament organizers attempt to minimize the latter problem by avoiding early-round matches between highest-ranked, or so-called *top-seeded*, players. Just as in Figure 2.12, they also use the method of seeding to facilitate tournament *byes*, so that the highest-ranked players don't even have to play in the first round or rounds. *Byes* allow them to assume that the first round is between $n = 2^k$ players.)

Certainly for the tournament method of determining the largest element in a list, we cannot assume that the list element that lost the last comparison made by the algorithm is the second-largest element in the list. However, the key observation here is that the second-largest must have lost a comparison to the largest element in *some* round (not necessarily the last round). Thus, for example, in our tournament shown in Figure 2.12, the candidates for second-best are Player F, Player G, and Player B. More generally, if the algorithm maintains a (dynamic) list of losers for each element, then the second-largest element must be found among the largest element's final list of losers (although a linear number of loser list updates are made, no additional comparisons are incurred). This final list contains $\log_2 n$ elements. A linear scan of these $\log_2 n$ elements (using $\log_2 n - 1$

comparisons) then determines the second-largest element in the list. Hence, the algorithm performs a total of $n + \log_2 n - 2$ comparisons (which turns out to be an optimal algorithm for this problem).

- Write pseudocode for the algorithm just described, and prove its correctness.
- 2.17 Trace the action of *BinarySearch*, including listing the value of *Low*, *High*, *Mid* after each iteration, for the list
- $$L: 2 \ 3 \ 5 \ 7 \ 11 \ 13 \ 17 \ 19 \ 23 \ 29 \ 31 \ 37$$
- for each of the following search elements *X*.
- $X = 3$
 - $X = 24$
 - $X = 108$
 - $X = 13$
- 2.18 Show that *BinarySearch* has worst-case complexity $\lceil \log_2(n + 1) \rceil$ for any n .
- 2.19 Describe a modification of *BinarySearch* that returns an index in the list after which the search element can be inserted and still maintain an ordered list (or zero if the search element is smaller than any element in the list).
- 2.20 Design an iterative version of *BinarySearch*.
- 2.21 Design and analyze a variant of *BinarySearch* that performs a single check for equality between the search element and a list element.
- 2.22 Suppose $L[0:n - 1]$ is a sorted list of distinct integers. Design and prove correct an algorithm similar to *BinarySearch* that finds an index *i* such that $L[i] = i$ or returns 0 if no such index exists.
- 2.23 Give pseudocode for interpolation search, and analyze its worst-case complexity.
- 2.24 Give pseudocode for a recursive version of interpolation search.
- 2.25 It has been estimated that there are fewer than 10^{83} atoms in the known universe. Show that $\log_2(\log_2 10^{83}) < 9$.
- 2.26 Design and analyze a forward scan version of *InsertionSort* where the list is implemented using an array. Discuss the drawbacks of the forward scan version as compared with the backward scan version.
- 2.27 Show that *InsertionSort* is a *stable* sorting algorithm.
- 2.28 a) Design a linked list version of *InsertionSort*.
b) Give the stable (but slightly less efficient) version of the algorithm in (a).
- 2.29 a) Design a recursive version of *InsertionSort*.
b) Design a recursive linked list version of *InsertionSort*.
- 2.30 The sorting algorithm *SelectionSort* is based on the simple idea of successively selecting the largest element in a sublist $L[0:n - i]$ and interchanging this element with the element in position $n - i$, $i = 1, \dots, n - 1$.
a) Give pseudocode for *SelectionSort*.
b) Analyze the complexity of *SelectionSort*.
- 2.31 Design a recursive version of *SelectionSort* as described in the previous exercise.
- 2.32 There is another well-known sorting algorithm called *BubbleSort*, which, like *SelectionSort*, results in the i^{th} -largest element being placed in its proper position at the completion of the i^{th} pass. The two algorithms differ in the manner in which this largest element is determined. During the i^{th} pass, *BubbleSort* passes sequentially through the sublist $L[0:n - i]$, comparing adjacent elements in the sublist and interchanging (swapping) them if they are out of order. Since this

sequential pass through the sublist starts at the *beginning* of the sublist, after the i^{th} pass we are guaranteed that the i^{th} -largest element will have “bubbled” to the end of the sublist $L[0:n - i]$, its proper place. In *BubbleSort*, this occurs automatically as a consequence of the interchange process, as opposed to *SelectionSort*’s method of actually determining the position in the sublist where the i^{th} -largest element occurred. Note that if on the i^{th} pass no swapping occurred, the list has been sorted. This suggests including a flag to detect this condition.

- a) Give the pseudocode for *BubbleSort*.
 - b) Determine the best-case and worst-case complexities of *BubbleSort*.
 - c) Design and analyze an improvement to *BubbleSort* based on keeping track of the last positions where swaps occur in each pass and where the passes are made alternately in different directions.
- 2.33 Give the appropriately labeled tree of recursive calls to *MergeSort* for lists of size 18. (See Figure 2.4)
- 2.34 Show that the tree of recursive calls for *MergeSort* has depth $\lceil \log_2 n \rceil$. Conclude that *MergeSort* performs at most $n \lceil \log_2 n \rceil$ comparisons for any list of size n .
- 2.35 Design a nonrecursive version of *MergeSort* based on Figure 2.4.
- 2.36 Given a list $L[0:n - 1]$, one way of maintaining a sorted order of L is to utilize an auxiliary array $Link[0:n - 1]$. the array $Link[0:n - 1]$ serves as a linked list determining the next highest element in L , so that the elements of L can be given in increasing order by
- $$L[Start], L[Link[Start]], L[Link[Link[Start]]], \dots$$
- Then, $Link^{n-1}[Start]$ is the index of the largest element in L , and we set $Link[Link^{n-1}[Start]] = Link^n[Start] = 0$ to signal the end of the linked list. Design a version of *MergeSort* that utilizes the auxiliary array $Link$.
- *2.37 Develop an in-place version of *MergeSort*; that is, a version that does not use an auxiliary array and utilizes only a few additional bookkeeping variables.
- 2.38 a) Give pseudocode for *QuickSort* that incorporates the Median of Three Rule.
b) Show that the worst-case performance of the version of *QuickSort* in part (a) remains quadratic.
- 2.39 It is useful to analyze how much stacking is generated by the recursion in *QuickSort*.
 - a) Show that *QuickSort* may have as many as $n - 1$ unresolved recursive calls active, so that the size of the stack can be as large as $n - 1$.
 - b) Give pseudocode for a modification of *QuickSort* for which the size of the stack is at most $\log_2 n$. Verify your result.
- 2.40 Give a nonrecursive version of *QuickSort* (utilizing explicit stack operations).
- 2.41 Give a stable version of *QuickSort* that avoids making extraneous interchanges of elements having identical values.
- 2.42 There are a number of variants of the algorithm *Partition* used in *QuickSort*. Design and analyze one such variant based on the following strategy for partitioning an array. The elements in the array minus the pivot element p (which we take to be the first element in the array) are dynamically divided into three contiguous blocks, B_1 , B_2 , and B_3 , where each element in B_1 is less than p , each element in B_2 is greater than or equal to p , and the status of the elements in B_3 is

yet to be decided. Initially, B_1 , B_2 are empty (so that B_3 is everything in the subarray except p). The algorithm performs $n - 1$ steps, where each step consists in placing the first element in B_3 into either B_1 or B_2 as appropriate.

Section 2.7 Radix Sort

- 2.43 Demonstrate the action of *RadixSort* for the following sample list, where each element is a 7-digit binary string (see Figure 2.8).
1011101, 0100011, 1010110, 1111101, 0011101, 0000001, 1000000, 1010101
- 2.44 Demonstrate the first two passes of *RadixSort* for the sample list given in Exercise 2.43, where the buckets are implemented as linked queues (see Figures 2.9 and 2.10).
- 2.45 Prove the correctness of *RadixSort*.
- 2.46 Determine whether *RadixSort* is a stable sorting algorithm.
- 2.47 For binary strings stored in an array, design and analyze a recursive radix sort based on partitioning the list into two sublists according to the most significant digit.

Additional Exercises

- 2.48 Although for large lists *InsertionSort* is, on average, much slower than *MergeSort*, it is faster for sufficiently small lists. A useful technique to improve the efficiency of a divide-and-conquer sorting algorithm such as *MergeSort* is to employ a sorting algorithm such as *Insertionsort* when the input size is not larger than some threshold t (a value of t between 8 and 16 usually works well).
- Write a program that computes the number of comparisons performed by *Mergesort* for a given input list and threshold t .
 - Run for thresholds t from 5 to 20 for various randomly generated lists of sizes 100, 1000, 10000. Report on which threshold t you observe to be best, and how it compares to ordinary *MergeSort*.
- 2.49 Repeat Exercise 5.48 for *Quicksort*.
- 2.50 A sloppy chef has delivered a stack of pancakes to a waiter, where the stack is all mixed up with respect to the size of the pancakes. The waiter wishes to rearrange the pancakes in order of their size, with the largest on the bottom. The waiter begins by selecting a pile of pancakes on the top of the stack and simply flips the pile over. He then keeps repeating this pancake-flipping operation until the pancakes are sorted.
- Show that there always exists a set of flips that sorts any given stack of pancakes.
 - Describe a pancake-flipping algorithm that performs at most $2n - 2$ flips in the worst case for n pancakes.
 - Show that any pancake-flipping algorithm must perform at least $n - 1$ flips in the worst case for n pancakes.
 - Suppose now that the pancakes have a burnt side, and the waiter wishes to have the pancakes stacked in order with the unburned side up. Redo parts (a), (b), and (c) for this new problem.

Shell Sort

The Exercises 2.51-2.54 refer to the sorting algorithm Shell sort. Shell sort is often used in practice due to its simplicity of code and good (but not optimal) worst-case performance. When he designed Shell sort in 1959, Donald Shell's idea was to start out by sorting sublists of a list $L[0:n - 1]$ of size n consisting of equally spaced, but far apart, elements. For example, if the successive elements in the sublists occupy positions k distance apart in the original list, then we have k sublists $S_0 = \{L[0], L[k], \dots\}$, $S_1 = \{L[1], L[1 + k], \dots\}$, \dots , $S_{k-1} = \{L[k - 1], L[2k - 1], \dots\}$. The process of sorting these k sublists is called a k -subsort (note that a 1-subsort corresponds to a sorting of the entire list L). A list in which each of the sublists S_0, S_1, \dots, S_{k-1} is sorted is called k -subsorted (also referred to in the literature as k -ordered, or k -sorted). Each of the sublists S_0, S_1, \dots, S_{k-1} contains at most $\lceil n/k \rceil$ list elements, so for a large k , sorting each sublist is fast using a sort like *InsertionSort*. Although the list L itself is not sorted by the k -subsort, in some sense it is closer to being sorted. Shell sort works by performing a succession of k -subsorts with decreasing increments $k_0 > k_1 > \dots > k_{m-1} = 1$.

To accomplish a k -subsort, rather than calling a sorting procedure for each sublist individually, we make a single call to the following simple variant of *InsertionSort*.

```
procedure InsertionSubsort( $L[0:n - 1], k$ )
Input:  $L[0:n - 1]$  (a list of size  $n$ ),  $k$  (a positive integer increment)
Output:  $L[0:n - 1]$  ( $k$ -subsorted in increasing order)
  for  $i \leftarrow k$  to  $n - 1$  do //insert  $L[i]$  in its proper position in the
    //already sorted sublist  $L[j], L[j + k], \dots, L[i - k]$  of  $S_j$ ,  $j = i \bmod k$ 
    Current  $\leftarrow L[i]$ 
    position  $\leftarrow i - k$ 
    while position  $\geq 1$  .and. Current  $< L[position]$  do
      // Current must precede  $L[position]$ 
       $L[position + k] \leftarrow L[position]$  // bump up  $L[position]$ 
      position  $\leftarrow position - k$ 
    endwhile
    // position + k is now the proper position for current =  $L[i]$ 
     $L[position+k] \leftarrow Current$ 
  endfor
end InsertionSubsort
```

The following pseudocode for *ShellSort* merely consists of successively calling *InsertionSubsort* with a given set of diminishing increments.

```

procedure ShellSort(L[0:n − 1],D[0:m − 1])
Input: L[0:n − 1] (a list of size n)
    K[0:m − 1] (an array of diminishing increments
        K[0] > K[1] > . . . > K[m − 1] = 1)
Output: L[0:n − 1] (sorted in increasing order)
for i ← 0 to m − 1 do
    InsertionSubsort(L[0:n − 1],K[i])
endfor
end ShellSort

```

The final stage of *ShellSort* consists of performing *InsertionSort* on the entire list, so that it is unquestionably correct. However, by then the list should be close to being sorted, so *InsertionSort* should work quickly on the list (see Exercise 2.53). Also, successive *k*-subsorting with diminishing *k* tends to reorder a list closer and closer to being sorted (see Exercise 2.54).

Even though Shell sort is a fairly simple algorithm and has been well-studied and used by many researchers, its analysis is not yet complete. The number of comparisons for a given input of size *n* performed by Shell sort depends on what set of increments is used, and the general mathematical analysis is difficult. For a general *n*, it is not known what set of increments exhibits optimal behavior.

A number of results are known for special choices of the increments. An order of complexity improvement over insertion sort is already achieved when only two increments *k* and 1 are used for a suitable *k*. In fact, it has been shown in the two-increment case that the optimal average complexity has order $n^{5/3}$ and is achieved when *k* is approximately $1.72n^{1/3}$. It has also been shown that when the increments are of the form $2^j - 1$, for all such increments less than *n*, then Shell sort exhibits worst-case complexity bounded above by $kn^{3/2}$ for a suitable constant *k* and large *n*. Since $2^{2j} - 1 = (2^j - 1)(2^j + 1)$, a large percentage of these increments divide one another, so we would expect to be able to do even better. Indeed, it has been shown that using increments of the form $2^{i_3}j$ for all such increments less than *n* yields approximately equal to $kn(\log_2 n)^2$ worst-case complexity.

- 2.51 a) Demonstrate the action of *ShellSort* on the list
33,2,56,23,55,78,2,98,61,108,14,60,56,77,5,3,1 with increments 5,3,1.
- b) Repeat (a) with increments 5,2,1.
- 2.52 a) Write a program that compares the number of comparisons made for a given input list *L*[0:*n* − 1] by *InsertionSort* to that for *ShellSort* with a given set of increments *K*[0:*m* − 1].
- b) Run your program with the input list and sets of increments given in Exercise 2.51
- c) Repeat (b) for randomly generated lists of sizes 100, 1000, 10000.
- 2.53 Suppose for each list element *L*[*i*] in a list *L*[0:*n* − 1] of size *n*, there are no more than *k* list elements *L*[*j*] such that *j* < *i* and *L*[*j*] > *L*[*i*]. Then *InsertionSort* makes at most $(k + 1)(n - 1)$ comparisons to sort the list.
- *2.54 For any two integers *k* and *l*, suppose we perform an *l*-subsort on a list *L* that is

already k -subsorted. Prove that then L remains k -subsorted—that is, L becomes both k -subsorted and l -subsorted.

Bingo Sort

In all of the sorting algorithms we have studied so far, the input size was a function of a single variable n . However, there are situations where the input size is most naturally a function of two or more variables. For example, suppose we consider the problem of finding efficient sorting algorithms for lists of size n with repeated elements. The input size is then a function of n and the number m of *distinct* elements in the list, where $1 \leq m \leq n$. The situation of repeated elements arises often in practice. For example, we might wish to sort the people in a large mailing list by zip codes for bulk mailing. Then the number of distinct elements m (zip codes) will be rather less than the number of people n . In this section we discuss bingo sort, which is an example of a sorting algorithm that works well for input lists with many repeated elements.

Bingo sort, which is a variation of selection sort, works as follows. Each distinct value in the list is considered to be a “bingo” value. Each pass of bingo sort corresponds to “calling out” a bingo value. The bingo values are called out in increasing order. During a given pass, all the elements having the current bingo value are placed in their correct positions in the list. The following pseudocode for bingo sort implements this process.

```

procedure BingoSort( $L[0:n - 1]$ )
Input:  $L[0:n - 1]$  (a list of size  $n$ )
Output:  $L[0:n - 1]$  (sorted in increasing order)
// first determine the minimum and maximum value for a list element
 $MaxMin(L[0:n - 1], MaxValue, MinValue)$ 
 $Bingo \leftarrow MinValue$  //initialize
 $NextAvail \leftarrow 0$ 
 $NextBingo \leftarrow MaxValue$ 
while  $Bingo < MaxValue$  do //move up all list elements that equal  $Bingo$ 
     $StartPos \leftarrow NextAvail$ 
    for  $i \leftarrow StartPos$  to  $n - 1$  do
        if  $L[i] = Bingo$  then //BINGO! Interchange  $L[i]$  and  $L[NextAvail]$ 
            interchange( $L[i], L[NextAvail]$ )
             $NextAvail \leftarrow NextAvail + 1$  // update  $NextAvail$ 
        else
            if  $L[i] < NextBingo$  then  $NextBingo \leftarrow L[i]$  endif
        endif
    endfor
     $Bingo \leftarrow NextBingo$  //initialize for next pass
     $NextBingo \leftarrow MaxValue$ 
endwhile
end  $BingoSort$ 
```

The action of $BingoSort$ is illustrated in Figure 2.13 for a sample list of size 10.

Pass 1	23	10	15	10	10	23	15	23	23	10
Start position 0	10	23	15	10	10	23	15	23	23	10
Bingo value 10	10	10	15	23	10	23	15	23	23	10
	10	10	10	23	15	23	15	23	23	10
	10	10	10	10	15	23	15	23	23	23
Pass 2										
Start position 4	10	10	10	10	15	23	15	23	23	23
Bingo value 15	10	10	10	10	15	15	23	23	23	23

Action of *BingoSort* for list 23, 10, 15, 10, 10, 23, 15, 23, 23, 10 after call to *MaxMin*

Figure 2.13

The best-case complexity $B(n,m)$ of *BingoSort* is achieved by a list where $n - m + 1$ elements have the (same) minimum value, and the worst case occurs for a list where $n - m + 1$ elements have the (same) maximum value (see Exercise 2.56).

- 2.55 Show that if a list has m distinct elements, then after exactly $m - 1$ passes of *BingoSort* the list is sorted into increasing order and the algorithm terminates.
- 2.56 a. Analyze the best-case complexity $B(n,m)$ of *BingoSort*.
b. Analyze the worst-case complexity $W(n,m)$ of *BingoSort*.

3

Mathematical Tools for Algorithm Analysis

Determining the complexities of an algorithm exactly is often difficult, and in practice it is usually sufficient to find asymptotic approximations to these complexities. To aid our description of asymptotic behavior, in this chapter we introduce and utilize standard notation for order and growth rate of functions. We also give you tools that can be used to help classify the growth rate of a given function as compared to that of commonly occurring growth rates such as logarithmic, linear, quadratic, and so forth.

In Chapter 2 we discussed the importance of recursion as a design strategy. In this chapter we discuss how recursive algorithms usually immediately yield recurrence relations for their complexities, and we illustrate this by giving an alternate proof of the worst-case complexity of *MergeSort*. We also give a general formula for solving some of the more commonly occurring recurrence relations that arise in algorithm analysis.

Proving correctness of an algorithm is perhaps even more fundamental to its analysis than determining its efficiency. Mathematical induction is the primary tool used to establish the correctness of algorithms. For recursive algorithms, induction on the input size is usually the relevant tool. For algorithms involving loops, establishing loop invariants (again, using induction) is usually the tool most relevant. Both techniques will be discussed in this chapter.

A function $L(n)$ is called a lower bound for, say, the worst case $W(n)$ of a problem Q , if *any* algorithm solving Q must perform at least $L(n)$ basic operations for some problem instance of Q of size n . Establishing lower bounds is an important tool in algorithm analysis, since they can be used to determine how close a given algorithm is to being optimal for the problem. In this chapter we use some simple counting arguments to establish lower bounds for such important problems as finding the maximum in a list, and sorting a list using a comparison-based sorting algorithm.

Unfortunately, there are many important problems for which the gap between a known lower bound for the problem and the performance of the best known algorithm solving the problem is wide indeed. We close this chapter with a discussion of an important class of such problems, known as NP-complete problems. Typically, a known lower bound for a given NP-complete problem is a polynomial of low degree, whereas the best known algorithm solving the problem has exponential (or at least super-polynomial) complexity. But nobody knows whether the NP-complete problems are really hard, or whether, despite over 40 years of searching by the best minds in theoretical computer science, we have simply just not found the key to solving any of them with a polynomial-complexity algorithm.

3.1 Asymptotic Behavior of Functions

Suppose $f(n) = 400n + 23$ and $g(n) = 2n^2 - 1$ are the worst-case complexities for two algorithms A and B , respectively, that solve the same problem. Which exhibits better worst-case performance? If $n \leq 200$, then $f(n)$ is greater than $g(n)$, so that algorithm B outperforms A . On the other hand, if $n > 200$, the reverse is true. When analyzing the complexity of an algorithm, it is

the behavior of the algorithm as the input size n tends to infinity (asymptotic behavior) that is usually considered to be most important. Thus, algorithm A is considered to have better worst-case behavior.

For purposes of algorithm analysis, the asymptotic behavior (*order*) of a function $f(n)$ not only ignores small values of n , but it also doesn't distinguish between $f(n)$ and $cf(n)$, where c is a positive constant. For example, *InsertionSort* performs $n^2/2 - n/2$ comparisons in the worst case for an input of size n . However, $n^2/2 - n/2$ and n^2 are considered to be asymptotically equivalent measures of the complexity of an algorithm.

There is good reason not to be too concerned about constants when we analyze algorithms. After all, the running time of an algorithm when we actually implement the algorithm in code on a computer will vary from one computer to another due to differences in processor speed, memory access, and so forth. However, these differences between the speed of two computers can generally be captured by a constant factor. It is for this reason that we analyze algorithms by identifying a basic operation and counting how many of these basic operations are performed on a given input, as opposed to measuring the running time of the algorithm on this input when implemented on some specific computer. When we talk about asymptotic growth, we will generally drop the phrase "up to a multiplicative constant," it being implicitly understood.

Of course, in practice the constants do matter. For example, if we have an algorithm with worst case $W(n) = 10^{1000}n$, then we do have a *linear* worst-case algorithm. However, in any practical situation, we would prefer to use an algorithm with worst case $W(n) = n^2$ even though the latter algorithm exhibits *quadratic* performance. It is true that eventually (i.e., for large enough n) the linear algorithm will perform fewer basic operations, but this only happens for $n > 10^{1000}$, which is an input size so large that it probably won't ever be encountered in a practical situation. On the other hand, it is theoretically important to know that there is a linear algorithm for this problem, since it gives us the hope that another linear algorithm with a smaller constant might exist for the problem.

3.1.1 The class “big oh” O

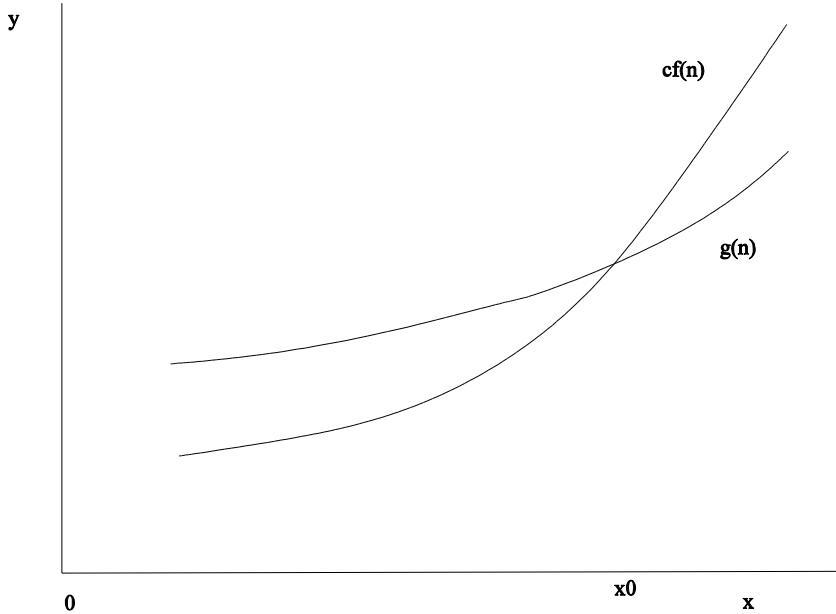
A fundamental notion in algorithm analysis, particularly in regard to the worst-case performance of an algorithm, is the “*big oh*” notation.

Definition 3.1.1

Given a function $f(n) \in \mathcal{F}$ we define $O(f(n))$ to be the set of all functions $g(n) \in \mathcal{F}$ having the property that there exist positive constants c and n_0 such that for all $n \geq n_0$,

$$g(n) \leq c f(n) \tag{3.1.1}$$

If $g(n) \in O(f(n))$, we say that $g(n)$ is *big oh of* $f(n)$, and that (up to a multiplicative constant depending on g and f) $f(n)$ grows at least as fast as $g(n)$ (See Figure 3.7).



$$g(n) \in O(f(n))$$

Figure 3.7

Remarks

By abuse of language, the condition $g(n) \in O(f(n))$ is sometimes written $g(n) = O(f(n))$ (a similar remark holds for the classes $\Omega(f(n))$, and $\Theta(f(n))$ defined later). When discussing algorithm behavior in the literature, you will often see a title for a paper reads something like “An $O(f(n))$ algorithm for ... (a given problem)”. What is meant is that the algorithm described in the paper has worst-case $W(n) \in O(f(n))$. Also, since in asymptotic behavior we generally ignore positive constants, we would not, for example, say that *Insertionsort* is an $O(n^2 / 2 - n/2)$ algorithm, but simply say it is an $O(n^2)$ (or quadratic) algorithm. The reason is (exercise) that $O(n^2 / 2 - n/2) = O(n^2)$.

3.1.2 The Classes “big omega” Ω and “big theta” Θ

Just as big oh is useful in describing the asymptotic behavior of $W(n)$ for an algorithm A , big omega Ω is useful in describing the asymptotic behavior $B(n)$ of A . In other words, big oh is used to establish *upper bounds* on the (asymptotic) behavior of an algorithm, whereas Ω is used to establish *lower bounds* on the (asymptotic) behavior of an algorithm.

Definition 3.1.2

Given a function $f(n) \in \mathcal{F}$, we define $\Omega(f(n))$ to be the set of all functions $g(n) \in \mathcal{F}$ having the property that there exist positive constants c and n_0 (depending on g and f) such that for all $n \geq n_0$,

$$g(n) \geq cf(n). \quad (3.1.2)$$

If $g(n) \in \Omega(f(n))$, we say that $g(n)$ is big omega of $f(n)$, and that (up to a multiplicative constant depending on g and f) $g(n)$ grows at least as fast as $f(n)$.

Note that $g(n) \in \Omega(f(n))$ if, and only if, $f(n) \in O(g(n))$. If $f(n)$ grows at least as fast as $g(n)$ (that is, $g(n) \in O(f(n))$, and $g(n)$ grows at least as fast as $f(n)$ (that is, $g(n) \in \Omega(f(n))$), then it is natural to say that $g(n)$ grows at the same rate as $f(n)$. The following class Θ formalizes this notion.

Definition 3.1.3

Given a function $f(n) \in \mathcal{F}$, we define $\Theta(f(n))$ to be the set of all functions $g(n) \in \mathcal{F}$ having the property that there exist positive constants c_1, c_2 , and n_0 (depending on g and f) such that for all $n \geq n_0$,

$$c_1 f(n) \leq g(n) \leq c_2 f(n). \quad (3.1.3)$$

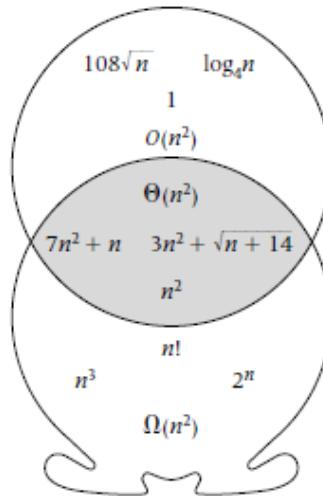
If $g(n) \in \Theta(f(n))$, we say that $g(n)$ is big theta of $f(n)$, and that (up to positive multiplicative constants) $g(n)$ grows at the same rate as $f(n)$.

It is easy to verify that Θ determines an equivalence relation on the set \mathcal{F} (exercise), and that

$$\Theta(f(n)) = \Omega(f(n)) \cap O(f(n)). \quad (3.1.4)$$

Sample members of the sets $O(n^2)$, $\Theta(n^2)$, and $\Omega(n^2)$ are illustrated in Figure 3.2.

Set \mathcal{F} of all (eventually positive)
functions $f: \mathbb{N} \rightarrow \mathbb{R}$



Venn diagram for $O(n^2)$, $\Theta(n^2)$, and $\Omega(n^2)$ and sample set members

Figure 3.2

Remark

As we have said, big oh O is the most important of the three relations in algorithm analysis. Moreover, Ω and Θ can both be defined in terms of O , since (exercise)

$$\begin{aligned} g(n) \in \Omega(f(n)) &\Leftrightarrow f(n) \in O(g(n)), \text{ and} \\ g(n) \in \Theta(f(n)) &\Leftrightarrow f(n) \in O(g(n)) \text{ and } g(n) \in O(f(n)). \end{aligned}$$

3.1.3 Useful Properties of the sets O , Θ , and Ω

Proposition 3.1.1 states some useful properties of the sets O , Θ , and Ω .

Proposition 3.1.1

The following properties hold for $f(n), g(n) \in F$.

1. Given any positive constant c , $\Omega(f(n)) = \Omega(cf(n))$, $\Theta(f(n)) = \Theta(cf(n))$, and $O(f(n)) = O(cf(n))$.
2. $g(n) \in O(f(n)) \Leftrightarrow O(g(n)) \subseteq O(f(n))$.
3. $O(f(n)) = O(g(n)) \Leftrightarrow \Omega(f(n)) = \Omega(g(n)) \Leftrightarrow \Theta(f(n)) = \Theta(g(n))$.
4. $g(n) \in O(f(n)) \Leftrightarrow f(n) \in \Omega(g(n))$. □

The proof of Proposition 3.1.1 is left to the exercises. It is common to use the notation

$$f(n) = g(n) + O(h(n)),$$

when $f(n) - g(n) \in O(h(n))$. Similar notation is used for Θ and Ω . For example, if $f(n) = 3n^3 + 4n^2 + 23n - 108$, then we could write

$$f(n) = 3n^3 + O(n^2).$$

We now give a table (Figure 3.3) that summarizes the order of the best-case, average, and worst-case complexities for most of the algorithms that we have discussed so far. The table also includes the sorting algorithm *HeapSort*, which is discussed in the next chapter.

Algorithm	$B(n)$	$A(n)$	$W(n)$
<i>HornerEval</i>	n	n	n
<i>Towers</i>	2^n	2^n	2^n
<i>LinearSearch</i>	1	n	n
<i>BinarySearch</i>	1	$\log n$	$\log n$
<i>Max, Min, MaxMin</i>	n	n	n
<i>InsertionSort</i>	n	n^2	n^2
<i>MergeSort</i>	$n \log n$	$n \log n$	$n \log n$
<i>HeapSort</i>	$n \log n$	$n \log n$	$n \log n$
<i>QuickSort</i>	$n \log n$	$n \log n$	n^2

Order of best-case, average, and worst-case complexities

Figure 3.3

When comparing the performance of two algorithms we compare their complexities. Say, for example the first algorithm has worst-case complexity $W(n) = f(n)$ and the second worst-case complexity $W(n) = g(n)$. They have the *same* order of complexity (in the worst-case) if $f(n) \in \Theta(g(n))$. On the other hand, the first algorithm has (strictly) *smaller* order of complexity if $f(n)$ belongs to $O(g(n))$, but $f(n)$ does not belong to $\Omega(g(n))$, or equivalently, the set $O(f(n))$ is strictly contained in the set $O(g(n))$, i.e., $O(f(n)) \subset O(g(n))$ (we leave the verification of this equivalency as an exercise). The following proposition provides a useful test for comparing the orders of two functions $f, g \in \mathcal{F}$ when the limit of the ratio $f(n)/g(n)$ exists as n tends to infinity.

Theorem 3.1.2 The Ratio Limit Theorem

Let $f(n), g(n) \in \mathcal{F}$. If the limit of the ratio $f(n)/g(n)$ exists as n tends to infinity, then f and g are comparable. Moreover, assuming $L = \lim_{n \rightarrow \infty} f(n)/g(n)$ exists, then the following results hold.

- | | |
|---|---|
| 1. $0 < L < \infty \Rightarrow f(n) \in \Theta(g(n))$. | <i>f</i> and <i>g</i> have the same order. |
| 2. $L = 0 \Rightarrow O(f(n)) \subset O(g(n))$ | <i>f</i> has a <i>smaller</i> order than <i>g</i> . |
| 3. $L = \infty \Rightarrow O(g(n)) \subset O(f(n))$ | <i>f</i> has a <i>larger</i> order than <i>g</i> . |

Proof The definition of $\lim_{n \rightarrow \infty} f(n)/g(n) = L$ states that for any given real number $\varepsilon > 0$ there exists an integer N_ε (depending on ε) such that for all $n \geq N_\varepsilon$, $|f(n)/g(n) - L| < \varepsilon$ or, equivalently,

$$(L - \varepsilon)g(n) < f(n) < (L + \varepsilon)g(n), \quad \text{for all } n \geq N_\varepsilon. \quad (3.1.5)$$

Case 1: $0 < L < \infty$. We are free to choose ε to be any positive real number. In particular, we may choose $\varepsilon = L/2$. Setting $c_1 = L/2$, $c_2 = 3L/2$, and $n_0 = N_{L/2}$ and substituting these values into (3.1.5) yields:

$$c_1g(n) < f(n) < c_2g(n), \quad \text{for all } n \geq n_0. \quad (3.1.6)$$

Hence, $f(n) \in \Theta(g(n))$.

Case 2: $L = 0$. Substituting $L = 0$ into (3.1.5), for *any* positive real number ε we have

$$f(n) < \varepsilon g(n), \quad \text{for all } n \geq N_\varepsilon. \quad (3.1.7)$$

It follows immediately that $f(n) \in O(g(n))$. To prove that $O(f(n)) \subset O(g(n))$, it is necessary to show that $f(n) \notin \Omega(g(n))$. Assume to the contrary that there exist constants c and n_0 such that

$$cg(n) \leq f(n) \text{ for all } n \geq n_0. \quad (3.1.8)$$

Substituting $\varepsilon = c$ in (3.1.7) yields $f(n) < cg(n)$ for all $n \geq N_c$. Thus, for any integer n larger than the maximum of n_0 and N_c , $f(n) < cg(n)$ and $cg(n) \leq f(n)$, which is a contradiction. Hence, $f(n) \notin \Omega(g(n))$, which implies that $O(f(n)) \subset O(g(n))$.

Case 3: $L = \infty$. For this case, the proof that $O(g(n)) \subset O(f(n))$ is left as an exercise. ■

Remark

A partial converse to case 2 of Theorem 3.1.2 is true: If $O(f(n)) \subset O(g(n))$ and $\lim_{n \rightarrow \infty} f(n)/g(n)$ exists, then this limit must be zero. It is easy to construct examples where $O(f(n)) \subset O(g(n))$ but $\lim_{n \rightarrow \infty} f(n)/g(n)$ does not exist (see Exercise 3.18). Similar remarks hold in the other two cases of Theorem 3.1.2.

The condition $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$ is so useful in mathematics that the following special notation is used to denote the corresponding relation on the set of real-valued functions.

Definition 3.1.6

Given the function $f(n)$, we define $o(f(n))$ to be the set of all the functions $g(n)$ having the property that

$$\lim_{n \rightarrow \infty} g(n)/f(n) = 0.$$

If $f(n) \in o(g(n))$, we say that $f(n)$ is *little oh of* $g(n)$.

Remark

In the mathematical literature, $g(n) \in o(f(n))$ is usually written $g(n) = o(f(n))$.

3.1.4 Strongly Asymptotic Behavior

Definition 3.1.7

Given the function $f(n)$, we define $\sim(f(n))$ to be the set of all functions $g(n)$ having the property that

$$\lim_{n \rightarrow \infty} g(n)/f(n) = 1.$$

If $g(n) \in \sim(f(n))$, then we say that $g(n)$ is *strongly asymptotic* to $f(n)$ and denote this by writing $g(n) \sim f(n)$.

It is easy to see that \sim is an equivalence relation on \mathcal{F} (exercise).

The following proposition follows immediately from the Ratio Limit Theorem and the definitions of o and \sim .

Proposition 3.1.5

Let $f(n), g(n) \in \mathcal{F}$. Then

1. $g(n) \sim f(n) \Rightarrow g(n) \in \Theta(f(n))$
2. $f(n) \in o(g(n)) \Rightarrow O(f(n)) \subset O(g(n))$.

We illustrate Proposition 3.1.5 by proving the following stronger version of Proposition 3.1.3 concerning the asymptotic behavior of polynomials.

Proposition 3.1.6

Let $P(n) = a_k n^k + \dots + a_1 n + a_0$ be any polynomial of degree k , $a_k > 0$. Then $P(n) \sim a_k n^k$.

Proof Consider the ratio $(a_k n^k + \dots + a_0)/a_k n^k$ as n tends to infinity

$$\begin{aligned} & \lim_{n \rightarrow \infty} \frac{(a_k n^k + \dots + a_0)}{a_k n^k} \\ &= \lim_{n \rightarrow \infty} \left(1 + \frac{a_{k-1}}{a_k n} + \dots + \frac{a_1}{a_k n^{k-1}} + \frac{a_0}{a_k n^k} \right) = 1. \end{aligned}$$

Proposition 3.1.6 now follows immediately from property (1) of The Ratio Limit Theorem. ■

As an example, since the binomial coefficient $C(n,k)$ is a polynomial of degree k with leading coefficient $1/k!$, it follows that $C(n,k) \sim n^k/k!$.

Applying Propositions 3.1.4 and 3.1.5 requires calculating $\lim_{n \rightarrow \infty} f(n)/g(n)$, which might be difficult to do directly. When calculating $\lim_{n \rightarrow \infty} f(n)/g(n)$, a powerful tool from calculus, known as L'Hôpital's Rule, is often helpful. Application of L'Hôpital's Rule requires that $f(n)$ and $g(n)$ be extended to functions $f(x)$ and $g(x)$, respectively, both of which are differentiable for sufficiently large real numbers x .

L'Hôpital's Rule

Let $f(x)$ and $g(x)$ be functions that are differentiable for sufficiently large real numbers x . If $\lim_{x \rightarrow \infty} f(x) = \infty$ and $\lim_{x \rightarrow \infty} g(x) = \infty$, then

$$\lim_{n \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{n \rightarrow \infty} \frac{f'(x)}{g'(x)}.$$

We now apply the Ratio Limit Theorem to an example that makes use of L'Hôpital's rule. We show that $2n^{3/2} + 6n \ln n \in \Theta(n^{3/2})$ by considering the ratio $(2n^{3/2} + 6n \ln n)/n^{3/2}$ as n tends to infinity. We have:

$$\begin{aligned} \lim_{x \rightarrow \infty} \frac{(2x^{3/2} + 6x \ln x)}{x^{3/2}} &= 2 + 6 \lim_{x \rightarrow \infty} \frac{\ln x}{x^{1/2}} \\ &= 2 + 6 \lim_{x \rightarrow \infty} \frac{1/x}{(1/2)x^{-1/2}} \\ &= 2 + 12 \lim_{x \rightarrow \infty} \frac{1}{x^{1/2}} = 2 + 0 = 2. \quad (\text{by L'Hôpital's Rule}) \end{aligned}$$

We say that a function $f(n)$ has *exponential order* if there exists positive real numbers a and b , $1 < a < b$, and a positive integer n_0 such that:

$$a^n < f(n) < b^n, \text{ for all } n > n_0.$$

As our second illustration, we show that polynomial order is smaller than any exponential order.

Proposition 3.1.7

For any nonnegative real constants k and a , with $a > 1$,

$$O(n^k) \subset O(a^n).$$

Proof Consider the ratio n^k/a^n . Repeated applications of L'Hôpital's Rule yields

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{n^k}{a^n} &= \lim_{n \rightarrow \infty} \frac{kn^{k-1}}{(\ln a)a^n} \\ &= \lim_{n \rightarrow \infty} \frac{k(k-1)n^{k-2}}{(\ln a)^2 a^n} \\ &\vdots \\ &= \lim_{n \rightarrow \infty} \frac{k!}{(\ln a)^k a^n} = 0. \end{aligned}$$

Proposition 3.1.7 now follows from the Ratio Limit Theorem. ■

Remarks. The complexity of an algorithm is said to be *polynomial* if $W(n) \in O(n^k)$ for some nonnegative integer k . This implies that the commonly occurring logarithmic, linear, quadratic, and $n\log n$ functions are all examples of polynomial complexity. In complexity theory, an algorithm exhibiting polynomial complexity is said to be in P and is usually regarded as “good.” However, there are many important problems for which only super-polynomial algorithms are known. The complexity of an algorithm is said to be *super-polynomial* if $W(n) \in \Omega(n^k)$ for every nonnegative integer k . By Proposition 3.1.7, exponential complexities are super-polynomial.

3.2 Asymptotic Order Formulae for Three Important Series

In this section we determine the asymptotic order for three mathematical series that occur often in algorithm analysis, namely, the sum of powers $S(n,k) = 1^k + 2^k + \dots + n^k$, for k a nonnegative integer, the sum of logarithms $L(b,n) = \log_b 1 + \log_b 2 + \dots + \log_b n = \log_b(n!)$ and the harmonic series $H(n) = 1 + 1/2 + \dots + 1/n$. We will see important applications of all of these formulas throughout the text.

3.2.1 Sums of Powers

$S(n,1) = 1 + 2 + \dots + n$ is certainly one of the most frequently occurring series in algorithm analysis. For example, it will come up later this chapter in the analysis of adjacent-key

comparison sorting. We will now derive the recurrence relation (3.2.1) that does not yield an explicit formula for $S(n,k)$ for a general k , but nevertheless can be used to show that $S(n,k)$ is a polynomial in n of degree $k+1$. It is interesting that recurrence relation (3.2.1) was also known to Pascal, who in fact published it in a slightly more general form (see Exercise 3.33).

Proposition 3.2.1

Given any nonnegative k , $S(n,k) = 1^k + 2^k + \dots + n^k$ satisfies the recurrence relation

$$S(n,k) = \frac{1}{k+1} \left[(n+1)^{k+1} - 1 - \left(\binom{k+1}{0} S(n,0) + \binom{k+1}{1} S(n,1) + \dots + \binom{k+1}{k-1} S(n,k-1) \right) \right] \quad \text{init. cond. } S(n,0) = n. \quad (3.2.1)$$

Proof The following proof amounts to interchanging the order of summation, using the binomial theorem, and noting the resulting telescoping sum.

$$\begin{aligned} \sum_{j=0}^k \binom{k+1}{j} S(n,j) &= \sum_{j=0}^k \binom{k+1}{j} \sum_{i=1}^n i^j \\ &= \sum_{i=1}^n \sum_{j=0}^k \binom{k+1}{j} i^j \quad (\text{by interchanging the order of summation}) \\ &= \sum_{i=1}^n [(1+i)^{k+1} - i^{k+1}] \quad (\text{by the binomial theorem}) \\ &= (1+n)^{k+1} - 1 \quad (\text{by "telescoping sum"}). \end{aligned} \quad (3.2.2)$$

Formula (3.2.1) follows immediately from (3.2.2) by isolating the term $S(n,k)$. ■

To illustrate (3.2.1), let us apply it to obtain an explicit formula for $S_1(n)$ and $S_2(n)$. Substituting in (3.2.1) with $k = 1$, we obtain

$$S(n,1) = \frac{1}{2} \left[(n+1)^2 - 1 - \left(\binom{2}{0} S(n,0) + \binom{2}{1} S(n,1) \right) \right] = \frac{1}{2} (n^2 + 2n - n) = \frac{1}{2} n(n+1).$$

Substituting in (3.2.1) with $k = 2$, we obtain:

$$\begin{aligned} S(n,2) &= \frac{1}{3} \left[(n+1)^3 - 1 - \left(\binom{3}{0} S(n,0) + \binom{3}{1} S(n,1) + \binom{3}{2} S(n,2) \right) \right] \\ &= \frac{1}{3} \left[(n+1)^3 - 1 - (n + 3n(n+1)/2) \right] \\ &= \frac{1}{3} [n^3 + 3n^2/2 + n/2] = \frac{1}{6} n(n+1)(2n+1). \end{aligned}$$

The exact value of $S(n,k)$ is not so important, but rather its asymptotic growth rate as determined from the following proposition.

Proposition 3.2.2

$S(n,k) = 1^k + 2^k + \dots + n^k$ is a polynomial in n of degree $k+1$ with leading (highest degree) coefficient $1/(k+1)$.

Proof The proof proceeds by induction (strong form, see Appendix A) on k .

Basis step: $S(n,0) = 1^0 + 2^0 + \dots + n^0 = n$, a polynomial of degree $0+1$ with leading coefficient $1/(0+1)$.

Induction step: Given a positive integer k , assume $S(n,j)$ is a polynomial in n with leading coefficient $1/(j+1)$ for all positive integers $j < k$. We must show that $S(n,k)$ is a polynomial in n of degree $k+1$ with leading coefficient $1/(k+1)$. From (3.2.1),

$$S(n,k) = \frac{(n+1)^{k+1}}{k+1} + P_k(n), \quad (3.2.3)$$

where

$$P_k(n) = \frac{-1}{k+1} \left(1 + \binom{k+1}{0} S(n,0) + \binom{k+1}{1} S(n,1) + \dots + \binom{k+1}{k+1} S(n,k-1) \right).$$

The strong form of our induction step implies that $P_k(n)$ is a polynomial in n of degree k . By the binomial theorem, $(n+1)^{k+1}$ is a polynomial in n of degree $k+1$ with leading coefficient 1. Proposition 3.2.2 follows from this latter observation and (3.2.3). ■

Various other properties of $S(n,k)$ that follow easily from (3.2.1) and induction are developed in the exercises. For example, based on (3.2.1) it is straightforward to give a recurrence relation for the coefficients of the polynomial representing $S(n,k)$. These coefficients are known as *Bernoulli numbers* $B(j,k)$, where $B(j,k)$ is the coefficient of n^j in the polynomial representing $S(n,k)$. For example, $B(0,k) = 0$ and $B(k,k+1) = 1/(k+1)$, for $k \geq 0$, $B(k,k) = 1/2$, $k \geq 1$, and $B(k-1,k) = k/12$ for $k \geq 2$.

Since $S(n,k) = 1^k + 2^k + \dots + n^k$ is a polynomial of degree $k+1$ with (highest order) leading coefficient $1/(k+1)$, it follows from Proposition 3.1.6 that $S(n,k) \sim n^{k+1}/(k+1)$.

3.2.2 Sums of Logarithms

The sum $L(b,n) = \log_b 1 + \log_b 2 + \dots + \log_b n = \log_b(n!)$ is another frequently occurring series in algorithm analysis. For example, we will see later in this chapter that $L(2,n)$ is a lower bound for the worst-case complexity of any comparison-based sorting algorithm.

Recall that for any real constants a and $b > 1$, $\log_a n \in \Theta(\log_b n)$, so that we may use *any* base when representing this Θ -class. For convenience, we denote this class simply by $\Theta(\log n)$. Similarly, it is clear that $L(a,n) \in \Theta(L(b,n))$, so that again we may use *any* base when representing this latter Θ -class. Thus, for convenience of notation, we drop the base and simply write

$$L(n) = \log(n!) = \log 1 + \log 2 + \cdots + \log n.$$

Proposition 3.2.3 $\log(n!) \in \Theta(n \log n)$.

Proof Clearly, $L(n) \in O(n \log n)$, since

$$\log 1 + \log 2 + \cdots + \log n \leq \log n + \log n + \cdots + \log n = n \log n.$$

It remains to show that $L(n) \in \Omega(n \log n)$. Let $m = \lfloor n/2 \rfloor$. We have

$$\begin{aligned} L(n) &= (\log 1 + \log 2 + \cdots + \log m) + [\log(m+1) + \log(m+2) + \cdots + \log n] \\ &\geq \log(m+1) + \log(m+2) + \cdots + \log n \\ &\geq \log(m+1) + \log(m+1) + \cdots + \log(m+1) \\ &= (n-m) \log(m+1) \\ &\geq (n/2) \log(n/2) = (n/2)(\log n - \log 2). \end{aligned}$$

Clearly, $(n/2)(\log n - \log 2) \geq (n/2)[\log n - (1/2)\log n]$ for all $n \geq n_0$, where n_0 is sufficiently large (the constant n_0 depends on the base chosen). Thus, $L(n) \geq (1/4)(n \log n)$, so that $L(n) \in \Omega(n \log n)$. ■

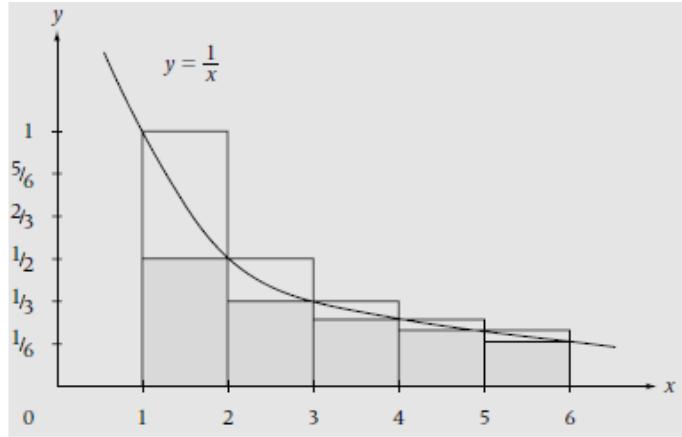
3.2.3 The Harmonic Series

Our third example, the harmonic series $H(n) = 1 + 1/2 + \dots + 1/n$, also occurs frequently in algorithm analysis. For example, $H(n)$ occurs in the average analysis of *QuickSort* given later in this chapter.

Proposition 3.2.4 $H(n) \sim \ln n$. Thus, $H(n) \in \Theta(\log n)$.

Proof By definition, $\int_1^n 1/x dx = \ln n$, so that $\ln n$ is the area bounded above by the graph of $1/x$ and bounded below by the interval $1 \leq x \leq n$ on the x axis (see Figure 3.4). We see from Figure 3.4 that this area is larger than the sum of the areas of the rectangles R_i , $i = 1, \dots, n-1$, where the base of R_i is the interval (of unit length) $i \leq x \leq i+1$, and the height of R_i equals $1/(i+1)$. Clearly, this area is smaller than the sum of the areas of the rectangles R_i^* , $i = 1, \dots, n-1$, where R_i^* has the same base as R , but has height $1/i$. Thus,

$$H(n) - 1 \leq \int_1^n \frac{1}{x} dx \leq H(n) - \frac{1}{n}. \quad (3.2.4)$$



$$H(n)-1 \leq \int_1^n \frac{1}{x} dx \leq H(n) - \frac{1}{n}.$$

Figure 3.4

Proposition 3.2.4 follows after dividing each of the three terms in (3.2.4) by $H(n)$ and taking the limit as n tends to infinity. The two outside limits are 1, so the middle limit must exist and also be 1. Showing that the outside limits are 1 amounts to showing that $\lim_{n \rightarrow \infty} H(n) = \infty$ (exercise).

3.3 Recurrence Relations for Complexity

A recurrence relation typically expresses the value of a function at an input n in terms of the value of the function at a smaller value or values of the input to the function. For example, when analyzing the performance of an algorithm based on a recursive strategy (whether or not the algorithm is expressed explicitly as a recursive algorithm), we often immediately can give a recurrence relation for its complexity for inputs of size n , since it works by repeating the algorithm's operations on an input or inputs of smaller size. Recurrence relations are so natural in determining the complexity of algorithms based on a recursive strategy that we state this as a key fact.

Key Fact

When analyzing the worst-case complexity $W(n)$ of a algorithm based on a recursive strategy, a recurrence relation should immediately emerge, since worst case will usually require that the algorithm performs the repeated operations with smaller input size(s) that exhibit worst-case performance. Similar comments hold for the best case $B(n)$ and the average complexity $A(n)$, although finding (and solving) a recurrence for $A(n)$ is usually more challenging.

As our first illustration of the above key fact, we consider binary search. For an input list generating worst-case behavior, after comparing to the midpoint element, binary search always looks for the search element in the larger sublist, which has size $\lceil n/2 \rceil$. Hence, we obtain the following recurrence for $W(n)$

$$W(n) = W(\lceil n/2 \rceil) + 1, \quad \text{init. cond. } W(1) = 1 \quad (3.3.1)$$

The occurrence of the ceiling function $\lceil \cdot \rceil$ in (3.3.1) makes it cumbersome to solve. However, if we assume that $n = 2^k$, the relation becomes:

$$W(n) = W(n/2) + 1, \quad \text{init. cond. } W(1) = 1. \quad (3.3.2)$$

The recurrence (3.3.2) can be solved by the method of *repeated substitution* (also called *unwinding*). For example, as a first step in solving (3.3.2), note that if we simply substitute $n/2$ for n in (3.3.2), we would get $W(n/2) = W((n/2)/2) + 1$, and hence $W((n/2)/2) + 1$ can be used to replace $W(n/2)$ in (3.3.2). Repeating this substitution process we obtain

$$\begin{aligned} W(n) &= W\left(\frac{n}{2}\right) + 1 \\ &= \left(W\left(\frac{(n/2)}{2}\right) + 1\right) + 1 = W\left(\frac{n}{2^2}\right) + 2 \\ &= \left(W\left(\frac{(n/2^2)}{2}\right) + 1\right) + 2 = W\left(\frac{n}{2^3}\right) + 3 \\ &\vdots \\ &= \left(W\left(\frac{(n/2^{k-1})}{2}\right) + 1\right) + k - 1 = W\left(\frac{n}{2^k}\right) + k = W(1) + k = 1 + k \end{aligned}$$

Since $k = \log_2 n$, we have

$$W(n) = 1 + \log_2 n, \quad n = 2^k. \quad (3.3.3)$$

We now show how formula (3.3.3) can be used to obtain an approximation for $W(n)$, for general n , which is accurate to within a single comparison. Given any positive integer n , there exists a positive integer j such that:

$$2^{j-1} \leq n < 2^j. \quad (3.3.4)$$

Clearly, $W(m) \leq W(n)$ for $m \leq n$. Thus,

$$W(2^{j-1}) \leq W(n) \leq W(2^j). \quad (3.3.5)$$

By (3.3.3), $W(2^{j-1}) = j$ and $W(2^j) = j + 1$. Substituting these results into (3.3.5) yields:

$$j \leq W(n) \leq j + 1. \quad (3.3.6)$$

By taking base-2 logarithms in (3.3.4), we obtain:

$$j-1 \leq \log_2 n < j. \quad (3.3.7)$$

Combining (3.3.6) and (3.3.7) yields:

$$\log_2 n < W(n) \leq 2 + \log_2 n, \quad n \geq 1. \quad (3.3.8)$$

Since $W(n)$ is an integer, (3.3.8) implies:

$$1 + \lfloor \log_2 n \rfloor \leq W(n) \leq 2 + \lfloor \log_2 n \rfloor, \quad n \geq 1. \quad (3.3.9)$$

It follows from (3.3.9) that $W(n) \in \Theta(\log n)$.

For many of the algorithms that we will analyze using recurrence relations, we will make the simplifying assumption that n is a power of some base b . For most functions that we will encounter, the asymptotic behavior of the function is determined by its behavior at the powers of the base b . In other words, we can “interpolate” the asymptotic behavior in between powers of b . We give a general condition for functions for which this interpolation is possible in Appendix D.

As a second example, we now consider the worst-case complexity of *MergeSort*. Again, we assume that $n = 2^k$ for some nonnegative integer k . Since the worst-case complexity of *Merge* is $n - 1$, $W(n)$ satisfies the following recurrence relation.

$$W(n) = 2W\left(\frac{n}{2}\right) + n - 1, \quad W(1) = 0 \quad (3.3.10)$$

Unwinding recurrence relation (3.3.10), we obtain

$$\begin{aligned} W(n) &= 2\left(2W\left(\frac{n}{2^2}\right) + \frac{n}{2} - 1\right) + n - 1 = 2^2W\left(\frac{n}{2^2}\right) + 2n - (1 + 2) \\ &= 2^2\left(2W\left(\frac{n}{2^3}\right) + \frac{n}{2^2} - 1\right) + 2n - (1 + 2) = 2^3W\left(\frac{n}{2^3}\right) + 3n - (1 + 2 + 2^2) \\ &\vdots \\ &= 2^k W(1) + kn - (1 + 2 + 2^2 + \dots + 2^{k-1}) = kn - (2^k - 1). \end{aligned}$$

Since $k = \log_2 n$, we have

$$W(n) = n \log_2 n - n + 1, \text{ for } n = 2^k, k = 0, 1, 2, \dots \quad (3.3.11)$$

Now that we have established a formula for $W(n)$ when n is a power of two, similar to the situation for binary search we can interpolate the asymptotic behavior of $W(n)$ for all n , and conclude that $W(n) \in \Theta(n \log n)$.

Note that the recurrence relations for binary search and *MergeSort* relate $W(n)$ to $W(n/2)$. More generally, whenever the recursive strategy for an algorithm repeats the algorithm’s operations on an input size that is a fixed fraction n/b of the original input size n , such as in Divide-And-Conquer strategies, then the recurrence relations for $W(n)$ will be in terms of $W(n/b)$. In Appendix C we consider the general solution to a recurrence relations that relate the n th term to the (n/b) th term.

Another type of recurrence relation that often arises when analyzing algorithms relates $W(n)$ to $W(n - 1)$. For example, consider the algorithm *InsertionSort* discussed in the previous chapter. Note that for an input list $L[0:n - 1]$, the action of *InsertionSort* can be viewed as first executing *InsertionSort* with input list $L[0:n - 2]$, followed by inserting $L[n - 1]$ into the previously sorted list $L[0:n - 2]$. This observation immediately yields the following recurrence relation for $W(n)$.

$$W(n) = W(n - 1) + n - 1 \quad \text{init. cond. } W(1) = 0. \quad (3.3.18)$$

An explicit formula for $W(n)$ is easily obtained by repeated substitution into (3.3.18). We have

$$\begin{aligned} W(n) &= W(n - 1) + n - 1 \\ &= (W(n - 2) + n - 2) + n - 1 \\ &= (W(n - 3) + n - 3) + n - 2 + n - 1 \\ &\vdots \\ &= W(1) + 1 + 2 + \cdots + n - 1 \\ &= n^2 / 2 - n / 2. \end{aligned}$$

Thus, $W(n) \in \Theta(n^2)$.

As another illustration a recurrence relating the n th term to the $(n - 1)$ st term, consider the number $t(n)$ of moves performed by *Towers* for n disks (see Appendix B). It is easily verified that $t(n)$ satisfies the recurrence relation:

$$t(n) = 2t(n - 1) + 1, \quad \text{init. cond. } t(1) = 1. \quad (3.3.19)$$

By repeated substitution in (3.3.19) we have:

$$\begin{aligned} t(n) &= 2t(n - 1) + 1 \\ &= 2(2t(n - 2) + 1) + 1 = 2^2(t(n - 2)) + 1 + 2 \\ &= 2^2(2t(n - 3) + 1) + 1 + 2 = 2^3(t(n - 3)) + 1 + 2 + 2^2 \\ &\dots \\ &= 2^{n-1}t(1) + 1 + 2 + 2 + \dots + 2^{n-2} \\ &= 2^{n-1} + 2^{n-1} - 1 = 2^n - 1. \end{aligned}$$

Thus, the Tower of Hanoi puzzle requires an exponential number of steps to solve. Using mathematical induction (see Exercise 3.51), it is easily proved that *Towers* solves the Towers of Hanoi puzzle using the fewest possible moves. In Appendix C we consider the general solution to a recurrence relations that relate the n th term to the $(n - 1)$ st term.

3.4 Mathematical Induction and Proving Correctness of Algorithms

In addition to the need to analyze the efficiency of an algorithm, it is even more basic to determine whether the algorithm is correct. Proving algorithms are correct is most often done with the aid of mathematical induction (see Appendix A). The induction usually utilizes one or both of the following two techniques,

1. induction on the input size of the algorithm,
2. induction to establish one or more loop invariants, which typically are stated as assertions concerning the value of a variable after each iteration of a given loop, and whose final value helps establish the correctness of the algorithm.

Induction on the input size is particularly relevant for algorithms based on a recursive strategy, since the recursive execution of the operations usually involves smaller inputs where the induction hypothesis applies. Moreover, it is usually the strong form of induction that is required since the recursion involves inputs whose size can be smaller than one less than the size of the original input. For the purposes of induction, it is important to know what should be considered as the input size. For example, given an algorithm like *MergeSort* or *QuickSort* for sorting a list $L[0:n - 1]$, in order to implement the recursion the additional input parameters *low* and *high* were included. In particular, the correct output for these algorithms is a sorting of $L[\text{low}:\text{high}]$ (so that $L[0:n - 1]$ is sorted when called originally with $\text{low} = 0$ and $\text{high} = n - 1$). Hence, the input size is $k = \text{high} - \text{low} + 1$.

Remark

A common mistake in proving correctness of recursive algorithms involving a list $L[0:n - 1]$ is to attempt induction on n , when, in fact, n remains constant throughout the action of the algorithm (and thus is not the natural integer upon which to do the induction). The recursive algorithm (as in *BinarySearch*, *MergeSort*, *Quicksort*, etc.) usually involves input parameters with names like *low* and *high* determining a sublist $L[\text{low}, \text{high}]$ of size $\text{high} - \text{low} + 1$, whose size is diminished with the recursive calls, and so is amenable to induction.

Proof of Correctness of *BinarySearch*.

We prove correctness using strong induction the size $k = \text{high} - \text{low} + 1$ of the sublist $L[\text{low}, \text{high}]$.

Basis step: $k \leq 1$. If $k = 0$, then $\text{low} > \text{high}$ and the sublist $L[\text{low}, \text{high}]$ is an empty list. *BinarySearch* then correctly returns -1 . If $k = 1$, then $\text{low} = \text{mid} = \text{high}$. If $X = L[\text{mid}]$, then *BinarySearch* correctly returns *mid*. Otherwise, *BinarySearch* is called with $\text{low} > \text{high}$, so that it correctly returns -1 .

Induction step: Assume *BinarySearch* is correct for all lists of size $\text{high} - \text{low} + 1 < k$, and consider the case when $\text{high} - \text{low} + 1 = k$. If $X = L[\text{mid}]$, then *BinarySearch* correctly returns *mid*. If $X < L[\text{mid}]$, then since $L[\text{low}, \text{high}]$ is a nondecreasing list, if X is in the sublist $L[\text{low}, \text{high}]$, it must be in the sublist $L[\text{low}, \text{mid} - 1]$. Since *BinarySearch* makes a recursive call with the sublist $L[\text{low}, \text{mid} - 1]$ of size less than k , it follows by induction that *BinarySearch* correctly returns an index of an element where X occurs in $L[\text{low}, \text{mid} - 1]$ or returns -1 if X is not in this sublist (and hence also not in the original list $L[\text{low}, \text{high}]$). A similar argument shows that *BinarySearch* is correct when $X > L[\text{mid}]$. ■

Proof of Correctness of *MergeSort* if *Merge* is Correct

We prove correctness using strong induction the size $k = \text{high} - \text{low} + 1$ of the sublist $L[\text{low}, \text{high}]$.

Basis step: $k \leq 1$. *MergeSort* is clearly correct in these cases of a one-element sublist or an empty sublist, respectively, since it simply returns without any further action.

Induction step: Given an integer k such that $1 < k \leq n$, assume that *MergeSort* is correct for all sublists with $\text{high} - \text{low} + 1 < k$, and consider a sublist with $\text{high} - \text{low} + 1 = k$. Then mid is assigned the value $\lfloor (\text{low} + \text{high})/2 \rfloor$, and *MergeSort* makes two recursive calls, with sublists $L[\text{low}: \text{mid} - 1]$ and $L[\text{mid} + 1: \text{high}]$. Both of these sublists have smaller size than k , so by our induction hypothesis, *MergeSort* sorts these sublists in increasing order. Hence, *MergeSort* is correct if *Merge* is correct. ■

The proof that *Merge* is correct uses loop invariants, and will be developed in the exercises. We now illustrate loop invariants by proving *HornerEval* is correct.

Proof of Correctness of *HornerEval*.

We establish the correctness of *HornerEval* by using induction to show that the following loop invariant condition holds after the k th pass of the loop in *HornerEval*, $k = 0, 1, \dots, n$:

$$\text{Sum has the value } a[n] * v^k + a[n-1] * v^{k-1} + \dots + a[n-k+1] * v + a[n-k].$$

Basis step: Before the loop is entered, *Sum* is assigned the value $a[n]$, so that the claim is true for $k = 0$.

Induction step: Assuming that the claim is true for $k < n$, we show that it is also true for $k + 1$. By assumption, the value of *Sum* after the k th pass is

$$a[n] * v^k + a[n-1] * v^{k-1} + \dots + a[n-k+1] * v + a[n-k].$$

Now the $(k + 1)$ st pass corresponds to the loop variable i having the value $n - k - 1$. Thus, after this pass *Sum* has the value

$$\begin{aligned} & (a[n] * v^k + a[n-1] * v^{k-1} + \dots + a[n-k+1] * v + a[n-k]) * v + a[n-k-1] \\ &= a[n] * v^{k+1} + a[n-1] * v^k + \dots + a[n-k+1] * v^2 + a[n-k] * v + a[n-k-1]. \end{aligned}$$

Thus, our claim is true for $k + 1$. By induction, our claim is therefore proved for all $k = 0, 1, \dots, n$. The correctness of *HornerEval* now follows from our claim, since the algorithm terminates after the n th pass with the correct evaluation of the polynomial at v .

Our next illustration is proofs of the correctness of the recursive algorithm *Powers*. *Powers* is proven correct by (the strong form of) induction on the exponent n (as we have mentioned, the input size to *Powers* is really $\log_2 n$, but this will not matter in this context).

Proof of Correctness of *Powers*.

Basis step: $n = 1$. When $n = 1$, clearly *Powers* correctly returns x .

Induction step: We assume that *Powers* is correct for all exponents $1 \leq k < n$, and consider an input with $k = n$. Assume first that n is even. Then *Powers* is recursively invoked with input parameters $x*x$ and $n/2$, which, by induction assumption, results in the return of $(x*x)^{n/2} = x^n$. This result is then returned by *Powers*, so that the algorithm is correct when n is even. If n is odd, then *Powers* is recursively invoked with input parameters $x*x$ and $(n-2)/2$, which, by induction assumption, results in the return of $(x*x)^{(n-1)/2} = x^{n-1}$. But then *Powers* returns $x*x^{n-1} = x^n$, so that the algorithm is correct in this case also. ■

Proof of Correctness of *Left-to-Right Binary Method*.

To prove correctness of *Left-to-Right Binary Method*, we use the strong form of induction on n .

Basis step: $n = 1$. When $n = 1$, clearly *Left-to-Right Binary Method* correctly returns x .

Induction step: Assume first that n is even. Then the binary expansion of n is obtained from the binary expansion of $n/2$ by adding a zero in the rightmost (least significant) position. So, by the strong form of induction, the algorithm correctly computes $x^{n/2}$. But the algorithm for input n proceeds exactly the same as with input $n/2$, except at the last stage returns the square of $x^{n/2}$, correctly returning x^n . Now assume that n is odd. Then the binary expansion of n is obtained from the binary expansion of $(n-1)/2$ by adding a one in the rightmost (least significant) position. So, by the strong form of induction, the algorithm correctly computes $x^{(n-1)/2}$. But the algorithm for input n proceeds exactly the same as with input $(n-1)/2$, except at the last stage returns x times the square of $x^{(n-1)/2}$, thereby correctly returning x^n . ■

Remark

Of course, the correctness of *Left-to-Right Binary Method* assumes that the binary expansion of n is correctly computed. We leave computation and correctness proof of an algorithm computing the binary expansion of n to the exercises. This remark also holds for the *Right-to-Left Binary Method*.

Proof of Correctness of *Right-to-Left Binary Method*

To prove the correctness of *Right-to-Left Binary Method*, we use the loop invariants determined by the values of *Pow* and *AccumPowers*. We claim that after processing the i th position in the binary presentation of n (the i th position corresponds to 2^{i-1}), $Pow = n^{2^i}$ and the value of *AccumPowers* is the product of all terms x^{2^j} as j runs from 0 to $i-1$ and $b_j \neq 0$. Our proof uses induction on the index position of the binary representation of n .

Basis step: $i = 1$. The value of *Pow* is x^2 and the value of *AccumPowers* is 1 if $b_0 = 0$, and x if $b_0 = 1$. Thus the basis step is verified.

Induction step: Assume that after we complete the processing at the $(i - 1)$ st position ($i < m$), $Pow = n^{2^{i-1}}$ and the value of $AccumPowers$ is the product of all terms x^{2^j} as j runs from 0 to $i - 2$ and $b_j \neq 0$. At the i th position, if $b_{i-1} = 0$ then $AccumPowers$ is not changed, whereas if $b_i = 1$ then we multiply $AccumPowers$ by $n^{2^{i-1}}$, so that $AccumPowers$ has the appropriate value in either case. After $AccumPowers$ is updated, then Pow is squared, yielding the value n^{2^i} . Thus, Pow and $AccumPowers$ have the correct values at the i th position. Letting $i = m$ completes the verification of correctness of *Right-to-Left Binary Method*.

Remarks

The *Right-to-Left Binary Method* models the way the recursive algorithm *Powers* computes x^n . It is an interesting exercise to verify this by tracing the stack of recursive calls that result in invoking *Powers* on some sample examples. In Exercise 3.x we give an iterative algorithm *Powers2* that is directly based on modeling this stack, and eliminating the stack altogether as can be done in general whenever we are dealing with *tail recursion* (see Appendix C) as is the case with *Powers*.

3.5 Establishing Lower Bounds for Problems

Once we have found one or more algorithms solving a given problem, it is important to know whether these algorithms are the best possible. Lower bound theory aids us in answering this question. The purpose of lower bound theory is to obtain (as sharp as possible) lower bounds for the complexities of *any* algorithm that solves the problem. An algorithm whose complexity equals a lower bound that has been established is an optimal algorithm for the problem and the lower bound is sharp. Such a sharp lower bound is the *complexity of the problem*. For example, we don't need to look for a better algorithm for polynomial evaluation than *HornerEval*, since any correct algorithm for evaluating an n th-degree polynomial must perform at least n multiplications.

In this section we introduce lower bound theory by determining sharp lower bounds for a comparison-based searching algorithm to find the maximum value in a list of size n and for sorting a list of size n using an adjacent-key comparison-based sorting algorithm. A lower bound of $n - 1$ for the best-case, worst-case, and average complexities of any comparison-based algorithm for finding the maximum is established using a simple counting technique. Lower bounds of $n(n - 1)/2$ for the worst-case complexity and $n(n - 1)/4$ for the average complexity of adjacent-key comparison-based sorting are obtained using simple properties of permutations. In the next section, we also establish a $\Omega(n \log n)$ lower bound for the worst-case complexity of *any* comparison-based sorting algorithm.

3.5.1 Lower Bound for Finding the Maximum

Consider the problem of using a comparison-based algorithm to find the maximum value of an element in a list $L[0:n - 1]$ of size n . Clearly, each list element must participate in at least one

comparison, so that $\lceil n/2 \rceil$ is trivially a lower bound for the problem. We now use a simple counting argument to show that any comparison-based algorithm for finding the maximum value of an element in a list of size n must perform at least $n - 1$ comparisons for any input. The algorithm *Max* given in Section 3.3 has best-case, worst-case, and average complexities all equal $n - 1$, so that *Max* is an optimal algorithm.

Suppose for any convenience that we restrict attention to lists that contain *distinct* elements. When a comparison-based algorithm compares two distinct elements X, Y , we say that X *wins* the comparison if $X > Y$, otherwise X *loses*. Thus, each comparison generates exactly one loss, and this is the unit of work that we associate with the comparison. Now we simply count the number of total losses that must occur if the algorithm is to correctly determine the maximum element. We claim that any comparison-based algorithm must generate at least $n - 1$ units of work. More precisely, we claim that each of the $n - 1$ elements that it *not* the maximum must lose at least one comparison. Indeed, suppose there is an input list $L[0:n - 1]$ that causes the algorithm to terminate with two list elements $L[i]$ and $L[j]$ *both* never having lost a comparison. Assume for definiteness that the algorithm declares $L[i]$ to be the maximum. Now consider the input list $L'[0:n - 1]$ that is identical to L except that $L'[j]$ is increased so that it is larger than $L[i]$ and remains different from any other list element. By the definition of comparison-based algorithms, our algorithm performs *identically* on L and L' . To see this, note that the algorithm could only perform differently when making comparisons involving $L'[j]$. But $L[j]$ won all its comparisons, and since $L'[j] \geq L[j]$, so does $L'[j]$. Thus, the algorithm must again declare $L'[i] = L[i]$ to be the maximum element in L' , which is a contradiction.

Thus, we have established that any comparison-based algorithm for finding the maximum element in a list of size n (of distinct elements) must perform at least $n - 1$ comparisons. Since *Max* performs $n - 1$ comparisons for *any* input list of size n , it is an optimal algorithm for the problem, as summarized in the following proposition.

Proposition 3.5.1

The problem of finding the maximum element in a list of size n using a comparison-based algorithm has worst-case, best-case, and average complexities all equal to $n - 1$. Moreover, *Max* is an optimal algorithm for the problem.

3.5.2 Lower Bounds for Adjacent-Key Comparison Sorting

We now consider the problem of sorting a list using an adjacent-key comparison-based algorithm. As discussed earlier, the sorting algorithm *InsertionSort* can be viewed as an adjacent-key comparison sort. Recall that *InsertionSort* has worst-case complexity $W(n) = n(n - 1)/2$. In this section we show that *any* adjacent-key comparison sort performs at least $n(n - 1)/2$ comparisons to sort a list $L[0:n - 1]$ of size n in the worst case. Hence, *InsertionSort* achieves optimal worst-case complexity for an adjacent-key comparison-based sort.

We may assume without loss of generality that the input lists of size n to the algorithm are chosen from the set of all permutations of the set of integers $\{1, 2, \dots, n\}$. We associate with each permutation a combinatorial entity called an *inversion* of the permutation and show that the maximum number of inversions in a permutation on n symbols is given by $n(n - 1)/2$. Since interchanging adjacent list elements (keys) removes at most one inversion, the maximum number

of inversions then serves as a lower bound for the problem of adjacent-key comparison-based sorting.

Proposition 3.5.2

A lower bound for the worst-case complexity $W(n)$ of any adjacent-key comparison-based sorting algorithm is $n(n - 1)/2$.

Proof Given a permutation π of $\{1, 2, \dots, n\}$, an *inversion* of π is a pair $(\pi(a), \pi(b))$, $a, b \in \{1, 2, \dots, n\}$, where $a < b$ but $\pi(a) > \pi(b)$. In Figure 3.5 we illustrate the inversions corresponding to the six permutations of $\{1, 2, 3\}$.

	$\begin{pmatrix} 1 & 2 & 3 \\ \pi(1) & \pi(2) & \pi(3) \end{pmatrix}$	
Representation of a permutation π		
$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$
none	$(3,2)$	$(2,1)$
$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$
		$(2,1), (3,1)$
		$(3,1), (3,2)$
		$(3,2), (3,1), (2,1)$

All permutations of $\{1, 2, 3\}$ and their associated inversions

Figure 3.5

Note that any sorting algorithm must ultimately remove all inversions. Note also that a comparison of list elements utilized by any adjacent-key comparison-based sorting algorithm by definition considers only inversions of the form $(\pi(i), \pi(i + 1))$. Removing any such inversion (by interchanging $\pi(i)$ and $\pi(i + 1)$) results in a decrease of exactly one inversion (see Exercise 3.52). Thus, the worst-case complexity of any adjacent-key comparison-based sorting algorithm is bounded below by the maximum number of inversions in a permutation of $\{1, 2, \dots, n\}$.

Consider the permutation corresponding to a list sorted in decreasing order—that is, the permutation $\pi(i) = n - i + 1$, $i = 1, \dots, n$. Clearly, every pair $(\pi(i), \pi(j))$, $1 \leq i < j \leq n$ is an inversion of π . The number of such pairs is given by the binomial coefficient,

$$\binom{n}{2} = n(n - 1)/2$$

thereby establishing the lower bound given in Proposition 3.5.2. ■

Since *InsertionSort* performs $n(n - 1)/2$ comparisons in the worst case, it follows from Proposition 3.5.2 that *InsertionSort* is an (exactly) optimal worst-case adjacent-key comparison-based sorting algorithm.

We now consider a lower bound for the average behavior of adjacent-key comparison-based sorting algorithms. The following proposition gives us a lower bound that is only half of that found for the worst case.

Proposition 3.5.3

A lower bound for the average complexity $A(n)$ of any adjacent-key comparison-based sorting algorithm is $n(n - 1)/4$.

Proof For any input permutation π , any sorting algorithm must ultimately remove all the inversions in π , so that the average complexity of any adjacent-key comparison sorting algorithm is bounded below by the average number of inversions $\iota(n)$ in a permutation of $\{1, 2, \dots, n\}$. We now give a quick proof of the formula $\iota(n) = n(n - 1)/4$, from which Proposition 3.5.3 follows.

Given any permutation π of $\{1, 2, \dots, n\}$, its *reverse* permutation π_{rev} is defined by

$\pi_{\text{rev}}(i) = \pi(n - i + 1)$, $i \in \{1, 2, \dots, n\}$. Now note that given any $x, y \in \{1, 2, \dots, n\}$, where $x > y$, the pair (x, y) occurs as an inversion in *exactly one* of the permutations π, π_{rev} (see Figure 3.6b).

Thus, the permutations π, π_{rev} have a total of exactly $n(n - 1)/2$ inversions between them. We can sum the number of inversions over all permutations of $\{1, 2, \dots, n\}$ by summing over each (unordered) pair $\{\pi, \pi_{\text{rev}}\}$, thereby obtaining $(n!/2)(n(n - 1)/2)$ inversions altogether. The average is then obtained by dividing this latter number by $n!$, yielding $\iota(n) = n(n - 1)/4$. ■

$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$
none	$(3,2)$	$(2,1)$	$(2,1), (3,1)$	$(3,1), (3,2)$	$(3,2), (3,1), (2,1)$

(a) All permutations of $\{1, 2, 3\}$ and their inversions

$\left\{ \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} \right\}$	$\left\{ \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \right\}$	$\left\{ \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \right\}$
none $(3,2), (3,1), (2,1)$	$(3,2)$ $(2,1), (3,1)$	$(2,1)$ $(3,1), (3,2)$

(b) Partition of the permutations of $\{1, 2, 3\}$ into pairs $\{\pi, \pi_{\text{rev}}\}$, with each pair $\{\pi, \pi_{\text{rev}}\}$ yielding a full set of inversions

Permutations of $\{1, 2, 3\}$

Figure 3.6

In the next section we will show that the average complexity $A(n)$ of *InsertionSort* is given by $A(n) = n^2/4 + 3n/4 - H(n)$, where $H(n)$ is the harmonic series $1 + 2 + \dots + 1/n$. Hence, $A(n)$ is asymptotically very close to $n^2/4$. Thus, by Proposition 3.5.3, *InsertionSort* has basically optimal average complexity.

3.5.3 Lower Bound for Comparison-Based Sorting in General

In the previous section we proved that adjacent key comparison-based sorting algorithms must perform at least $n(n - 1)/2$ comparisons in the worst case for inputs of size n , and $n(n - 1)/4$ comparisons on average. Similar arguments show that for any *fixed* (independent of n) positive integer k , if a comparison-based algorithm always makes comparisons between list elements that occupy positions no more than k indices apart, then the worst case and average performance remains quadratic in the input size n (see Exercise 3.53). Thus, to improve on quadratic performance, we need to compare list elements that are far apart (that is, the difference between index positions of some of the compared elements must *grow* as a function of the input size). Of course, it is easy to design a comparison-based algorithm that does compare far apart elements, but whose performance is still quadratic. What we need is a clever way to use comparison based sorting such as *MergeSort*. But the question arises as to how good we can do in general with comparison-based sorting. The answer for the worst-case complexity is found in the following proposition.

Proposition 3.5.4

A lower bound for the worst-case complexity $W(n)$ of any comparison-based sorting algorithm is $\log_2 n! \in \Omega(n \log n)$.

Proof Given any comparison-based algorithm having worst-case complexity $m = W(n)$, we define a sequence of m partitions C_1, \dots, C_m of the set of all permutations of $\{1, \dots, n\}$ as follows. C_1 is a bipartition of the set of permutations of $\{1, \dots, n\}$ into two sets S_0 and S_1 , where a permutation π is placed in S_0 if, when input to algorithm, the elements that are compared in the first comparison are in order and placed in S_1 if they are out of order(determine an inversion). Inductively, assuming C_1, \dots, C_k have been defined, we define C_{k+1} to be a refinement of C_k as follows. Consider any set S in C_k . We bipartition S into two sets X and Y by placing a permutation $\pi \in S$ in Y if the algorithm performs at least $k + 1$ comparisons with π as input and the elements that are compared in the $(k + 1)^{\text{st}}$ comparison are out of order, and letting X be the set of remaining permutations in S . If any of the resulting sets of C_{k+1} are empty, we simply remove them.

After m comparisons the algorithm has terminated for each input permutation. We now show that each set in C_m contains exactly one permutation. Suppose, to the contrary that some set S of C_m contains two permutations π_1 and π_2 . Then, since the algorithm is comparison-based, it can easily be shown by induction that, for each comparison performed by the algorithm, precisely the same list positions were compared with π_1 as input versus π_2 and the relative order of these elements was the same. It follows that π_1 and π_2 must be the same permutation (a contradiction). Thus, since there are $n!$ permutations, by the pigeon hole principle, $|C_m| \geq n!$. Clearly, $|C_m| \leq 2^m$ so that we have

$$2^m \geq |C_m| \geq n! \Rightarrow m \geq \log_2 n! \in \Omega(n \log n).$$



MergeSort is an example of a sorting algorithm whose worst-case performance is $O(n \log n)$, so that it exhibits order-optimal behavior in the worst case. *MergeSort* is also order-optimal on average, since it turns out that $\Omega(n \log n)$ is also a lower bound for the average behavior of comparison-based sorting. Since *QuickSort* has quadratic worst-case complexity, it is not order-optimal in the worst case. However, it is often used since its average complexity belongs to $O(n \log n)$, so it has order-optimal average complexity, and performs well in practice. We now discuss average behavior of algorithms more formally. We assume that the reader is familiar with the discussion of probability that is contained in Appendix E.

3.6 Probabilistic Analysis of Algorithms

In practice, a given algorithm is usually run repeatedly with varying inputs of size n . Thus, another important measure of the performance of an algorithm is its *average complexity* $A(n)$. Let $\tau: \mathcal{I}_n \rightarrow N$ denote the mapping (random variable) that sends an input I in \mathcal{I}_n to the number of basic operations performed by the algorithm on input I . The average complexity $A(n)$ of the algorithm is defined as the *expected* number $E[\tau]$ of basic operations performed, *and depends on the probability distribution imposed on the sample space I_n* (see Appendix E for definitions of the terms used in this section). For now, we assume that I_n is a finite set and that each input $I \in \mathcal{I}_n$ has probability $p(I)$ of occurring as the input to the algorithm. An important special case is when each input is equally likely (the *uniform distribution*), in which case $p(I) = 1/|\mathcal{I}_n|$, and average complexity is then the familiar

$$A(n) = \left(\sum_{I \in \mathcal{I}_n} \tau(I) \right) / |\mathcal{I}_n| = E[\tau]$$

Since our interest is in algorithm analysis, throughout this chapter we specialize the formulas given in Appendix E for the expectation of a random variable X defined on a sample space S to the case where $X = \tau$ and $S = \mathcal{I}_n$. The general definition of $A(n)$ follows.

Definition 3.6.1 Average Complexity

Given a probability function $p: \mathcal{I}_n \rightarrow [0,1]$ defined on the finite input set \mathcal{I}_n , the *average complexity* of an algorithm is defined as:

$$A(n) = \sum_{I \in \mathcal{I}_n} \tau(I) p(I) = E[\tau] \quad (3.6.1)$$

Formula (3.6.1) for $A(n)$ is rarely used directly, since it is simply too cumbersome to examine each term in \mathcal{I}_n directly. Also, the growth of the summation as a function of the input size n is usually hard to estimate, much less calculate exactly. Thus, when analyzing the average complexity of a given algorithm, we usually seek closed-form expressions or estimates for $A(n)$, or formulas that allow some gathering of terms in (3.6.1). For example, let p_i denote probability that the algorithm performs exactly i basic operations; that is $p_i = p(\tau = i)$. Then:

$$A(n) = E[\tau] = \sum_{i=1}^{W(n)} ip_i, \quad (3.6.2)$$

Formula (3.6.2) is often useful in practice, and follows from (3.6.1) by simply gathering up, for each i between 1 and $W(n)$, all the inputs I such that $p(I) = i$. We illustrate the use of (3.6.2) by computing the average complexity of *LinearSearch*.

3.6.1 Average Complexity of *LinearSearch*

To simplify the discussion of the average behavior of *LinearSearch*, we assume that the search element X is in the list $L[0:n - 1]$ and is equally likely to be found in any of the n positions. Note that i comparisons are performed when X is found at position i in the list. Thus, the probability that *LinearSearch* performs i comparisons is given by $p_i = 1/n$. Substituting these probabilities into (3.6.2) yields:

$$A(n) = \sum_{i=1}^{W(n)} ip_i = \sum_{i=1}^n i \frac{1}{n} = \left(\frac{n(n+1)}{2} \right) \frac{1}{n} = \frac{n+1}{2}. \quad (3.6.3)$$

Formula (3.6.3) is intuitively correct, since under our assumptions X is equally likely to be found in either half of the list. To see that (3.6.3) truly reflects average behavior, suppose that we run *LinearSearch* m times (m large) with a fixed list $L[0:n - 1]$ and with search element X being randomly chosen as one of the list elements. Let m_i denote the number of runs in which X was found at position i . Then the total number of comparisons performed over the m runs is given by $1m_1 + 2m_2 + \dots + nm_n$. Dividing this expression by m gives the average number $A_m(n)$ of comparisons over the entire m runs, as follows:

$$A_m(n) = 1\left(\frac{m_1}{m}\right) + 2\left(\frac{m_2}{m}\right) + \dots + n\left(\frac{m_n}{m}\right). \quad (3.3.6)$$

Note that for large m , the ratios m_i/m occurring in (3.3.6) approach the probability p_i that X occurs in the i th position. Since we have assumed that X is equally likely to be found in any of the n positions, each m_i is approximately equal to m/n . Hence, substituting $m_i = m/n$ into (3.3.6) yields:

$$A_m(n) \approx \frac{1+2+\dots+n}{n} = \frac{n+1}{2} = A(n) \quad (3.6.5)$$

To summarize, *LinearSearch* has best-case, worst-case, and average complexities 1, n , and $(n+1)/2$, respectively. The best-case complexity is a constant independent of the input size n , whereas the worst-case and average complexities are both linear functions of n . For simplicity, we say that *LinearSearch* has *constant* best-case complexity and *linear* worst-case and average complexities.

We calculated $A(n)$ for *LinearSearch* under the assumption that the search element X is in the list. In the exercises we examine a more general situation where we assume that X is in the list with probability p , $0 \leq p \leq 1$.

Often there is a natural way to assign a probability distribution on I_n . For example, when analyzing comparison-based sorting algorithms it is typical to assume that each of the $n!$

permutations (orderings) of a list of size n is equally likely to be input to the algorithm. The average complexity of any comparison-based sorting algorithm is then the sum of the number of comparisons generated by each of the $n!$ permutations divided by $n!$. In practice, it is not feasible to examine each one of these $n!$ permutations individually, as $n!$ simply grows too fast. Fortunately, there are techniques that allow us to calculate this average without resorting to permutation-by-permutation analysis.

3.6.2 Two Major Techniques for Computing Average Behavior

There are two techniques that are particularly useful in computing $A(n)$, namely,

1. partitioning the algorithm into disjoint stages,
or,
2. partitioning the input space into disjoint subsets.

Technique (1) leads to the application of the fundamental additive property of expectation.

Additivity of Expectation. Suppose the random variable $\tau = \tau_1 + \tau_2 + \dots + \tau_m$. Then:

$$A(n) = E[\tau] = E[\tau_1] + E[\tau_2] + \dots + E[\tau_m]. \quad (3.6.6)$$

Technique (2) leads to the application of the additivity of expectation together with the notion of conditional expectation.

Partitioning \mathcal{I}_n and Conditional Expectation. Suppose the \mathcal{I}_n is partitioned into disjoint sets S_1, \dots, S_m . This partition is also equivalently determined (see Appendix E) by defining a second random variable $Y: \mathcal{I}_n \rightarrow [1, m]$ where $Y(i) = i$ if i belongs to S_i , so that $S_i = Y^{-1}(i)$, $i = 1, \dots, m$. Then if $E[\tau|Y=i]$ denotes the conditional expectation of τ given that $Y=i$, we have

$$A(n) = E[\tau] = E[\tau|Y=1]P(Y=1) + \dots + E[\tau|Y=m]P(Y=m). \quad (3.6.7)$$

We illustrate the use of (3.6.6) by computing $A(n)$ for *InsertionSort*, and the use of (3.6.7) by computing $A(n)$ for *QuickSort*. The analysis of both *InsertionSort* and *Quicksort* involve the harmonic series $H(n) = 1 + 1/2 + \dots + 1/n$.

3.6.3 Average Complexity of *InsertionSort*

Since *InsertionSort* is a comparison-based algorithm, we can assume without loss of generality that inputs to *InsertionSort* are permutations of $\{1, 2, \dots, n\}$. We also assume that each permutation is equally likely to be the input to *InsertionSort*. Unlike our analysis of *LinearSearch*, we cannot compute $A(n) = E[\tau]$ directly by applying formula (3.6.2). Instead, we partition the algorithm *InsertionSort* into $n - 1$ stages. The i^{th} stage consists of inserting the $(i + 1)^{\text{st}}$ element $L[i]$ into its proper position in the sublist $L[0:i - 1]$, where the latter sublist has already been sorted by the algorithm. Let τ_i denote the number of comparisons performed in stage i , so that $\tau = \tau_1 + \dots + \tau_{n-1}$ and, by (6.2.5),

$$A(n) = E[\tau] = E[\tau_1] + E[\tau_2] + \dots + E[\tau_{n-1}]. \quad (3.6.8)$$

We now calculate $E[\tau_i]$, $i = 1, \dots, n-1$, using formula (3.6.7). We have

$$E[\tau_i] = \sum_{j=1}^i j P(\tau_i = j). \quad (3.6.9)$$

Our assumption of a uniform distribution on the input space implies that any position in $L[0:i]$ is equally likely to be the correct position for $L[i]$. Thus, the probability that $L[i]$ is the j^{th} largest of the elements in $L[0:i]$ is equal to $1/(i+1)$. If $L[i]$ is the j^{th} largest, where $j \leq i$, then exactly j comparisons are performed by *InsertionSort* when placing $L[i]$ in its correct position. If $L[i]$ is the $(i+1)^{\text{st}}$ largest (that is, the smallest), then exactly i comparisons are performed by *InsertionSort* when placing $L[i]$ in its correct position. It follows that

$$\begin{aligned} P(\tau_i = j) &= \frac{1}{i+1}, \quad j = 1, \dots, i-1, \\ P(\tau_i = i) &= \frac{2}{i+1}, \quad i = 1, \dots, n-1. \end{aligned} \quad (3.6.10)$$

Substituting (3.6.10) into (3.6.9) and simplifying yields

$$E[\tau_i] = \left(\sum_{j=1}^i \frac{j}{i+1} \right) + \frac{i}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}, \quad i = 1, \dots, n-1. \quad (3.6.11)$$

Substituting (3.6.11) into (3.6.8), we have

$$\begin{aligned} A(n) &= \sum_{i=1}^{n-1} \left(\frac{i}{2} + 1 - \frac{1}{i+1} \right) \\ &= (n-1) \frac{n}{4} + (n-1) - \left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \right) \\ &= (n-1) \frac{n}{4} + n - H(n), \end{aligned}$$

where $H(n)$ is the harmonic series $1 + 1/2 + \dots + 1/n \sim \ln n$. In particular, $A(n) \in \Theta(n^2)$. Note that the average complexity of *InsertionSort* is about half that of its worst-case complexity, since the highest-order terms in the expressions for these complexities are $n^2/4$ and $n^2/2$, respectively.

Often it is possible to find a recurrence relation expressing $A(n)$ in terms of one or more of the values $A(m)$ with $m < n$. Of course, if the algorithm itself is written recursively, then the recurrence relation is usually easy to find. In general, finding a recurrence relation for $A(n)$ often involves utilizing the techniques of partitioning the algorithm or the input space. The analysis of the average complexity of *QuickSort* uses both of these techniques.

3.6.4 Average Complexity of *QuickSort*

As with *InsertionSort*, we assume that input lists $L[0:n - 1]$ to *QuickSort* are all permutations of $1, 2, \dots, n$, with each permutation being equally likely. We partition *QuickSort* into two stages, where the first stage is the call to *Partition* and the second stage is the two recursive calls with input lists consisting of the sublists on either side of the proper placement of the pivot element $L[0]$. Thus, $\tau = \tau_1 + \tau_2$, where τ_1 is the (constant) number $n + 1$ of comparisons performed by *Partition* and τ_2 is the number of comparisons performed by the recursive calls. Hence,

$$A(n) = E[\tau] = E[\tau_1] + E[\tau_2] = n + 1 + E[\tau_2]. \quad (3.6.12)$$

We compute $E[\tau_2]$ using (3.6.7) by introducing the random variable Y that maps an input list $L[0:n - 1]$ into the proper place for $L[0]$ as determined by a call to *Partition*. The uniform distribution assumption on the input space implies that:

$$P(Y = i) = \frac{1}{n}, \quad i = 0, \dots, n - 1. \quad (3.6.13)$$

If $Y = i$, then the recursive calls to *QuickSort* are with the two sublists $L[0:i - 1]$ and $L[i+1:n - 1]$. Our assumption of a uniform distribution on the input space implies that the expected number of comparisons performed by *QuickSort* on the sublists $L[0:i - 1]$ and $L[i + 1:n - 1]$ is given by $A(i)$ and $A(n - i - 1)$, respectively. Hence,

$$E[\tau_2 | Y = i] = A(i) + A(n - i - 1), \quad i = 0, \dots, n - 1. \quad (3.6.14)$$

Combining (3.6.7), (3.6.12), (3.6.13), and (3.6.14), we have

$$\begin{aligned} A(n) &= (n + 1) + \sum_{i=0}^{n-1} E[\tau_2 | Y = i] P(Y = i) \\ &= (n + 1) + \sum_{i=0}^{n-1} (A(i) + A(n - i - 1)) \left(\frac{1}{n} \right) \\ &= (n + 1) + \frac{2}{n} (A(0) + A(1) + \dots + A(n - 1)), \\ \text{init. cond. } A(0) &= A(1) = 0. \end{aligned} \quad (3.6.15)$$

Recurrence relation (3.6.15) is an example of what is sometimes referred to as a *full history* recurrence relation, since it relates $A(n)$ to *all* of the previous values $A(i)$, $0 \leq i \leq n - 1$. Fortunately, with some algebraic manipulation, we can transform (3.6.15) into a simpler recurrence relation relating $A(n)$ to just $A(n-1)$. The trick is to first observe that

$$nA(n) = n(n + 1) + 2(A(0) + A(1) + \dots + A(n - 2) + A(n - 1)). \quad (3.6.16)$$

Substituting $n - 1$ for n in (3.6.16), yields

$$(n-1)A(n-1) = n(n-1) + 2(A(0) + A(1) + \dots + A(n-2)). \quad (3.6.17)$$

Hence, subtracting (3.6.17) from (3.6.16) we obtain

$$nA(n) - (n-1)A(n-1) = 2n + 2A(n-1). \quad (3.6.18)$$

Rewriting (3.6.18) by moving the term involving $A(n-1)$ to the right-hand side and dividing both sides by $n(n+1)$ yields

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2}{n+1}. \quad (3.6.19)$$

Letting $t(n) = A(n)/(n+1)$, (3.6.19) becomes

$$t(n) = t(n-1) + \frac{2}{n+1}. \quad (3.6.20)$$

Recurrence relation (3.6.20) directly unwinds to yield

$$\begin{aligned} t(n) &= 2\left(\frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n+1}\right) \\ &= 2H(n+1) - 3, \end{aligned} \quad (3.6.21)$$

where $H(n)$ is the harmonic series. Thus $t(n) \sim 2 \ln n$, so that the average complexity $A(n)$ of *QuickSort* satisfies

$$A(n) \sim 2n \ln n.$$

(3.6.22)

In particular, *QuickSort* exhibits $O(n \log n)$ average behavior, which is order optimal for a comparison-based sorting algorithm.

3.7 Hard Problems

It is a curious fact in the theory of algorithms that most problems of practical importance are either solvable by algorithms having small degree polynomial (worst-case) complexity, or the best known solutions have complexity that grows faster than *any* polynomial (*super-polynomial*). An example of the latter problem is the famous *Traveling Salesman Problem* (TSP), which asks for a minimum length tour of a given set of n cities from a given starting city. Since there are $(n-1)!$ possible tours, even for a relatively small value of n it is computationally infeasible to find an optimal (minimum distance) tour by a brute force examination of each tour. While better algorithms exist for TSP than brute force examination, all known algorithms are super-polynomial. TSP has many important applications, and whole books have been written about the problem. However, finding an algorithm having polynomial complexity solving TSP, or proving that no such algorithm exists, remains a mystery (it is generally believed that there is no polynomial-complexity algorithm for TSP).

Another problem for which no polynomial algorithm has been found is the problem of factoring an integer n . The fact that factoring is generally believed to be a truly hard problem (that is, requiring a super-polynomial algorithm) is the basis for the most widely used Internet security encryption schemes. Yet another problem that appears to be hard is the so-called *discrete logarithm* problem, which for a given positive base b asks for the value of n if the value of b^n is known.

3.6.1 One-way Functions and Sharing Secrets

The fact that the discrete logarithm problem is hard, yet the inverse problem of computing b^n is easy, that is, it can be computed efficiently, has an important application to cryptography. For example, consider the following problem: Alice and Bob wish to send confidential information to one another over some communication medium (e-mail, telephone conversation, fax, and so forth). However, they know that Eve Dropper has the ability to monitor all communication sent over this medium. Moreover, Alice and Bob have not shared in advance any secret information that could be used for encrypting information. We now describe a communication protocol based on exponentiation modulo p that can be used by Alice and Bob to share secrets.

Alice and Bob first send messages, monitored by Eve, in which they agree to use a conventional cryptographic system for communication. The cryptographic system, also assumed to be known by Eve, requires that they exchange a secret key value, which we assume is an arbitrary positive integer. While the key value is arbitrary, it must be known by both Alice and Bob. The following question arises: Can Alice and Bob publicly (that is, known to Eve) exchange information that allows them to efficiently determine a key, but at the same time makes it infeasible for Eve to discern the value of this key?

The answer lies in the notion of *one-way* functions. One-way functions are efficiently computable, but their inverses are infeasible to compute. In 1976, Diffie and Hellman first pointed out the utility of one-way functions in cryptology and identified exponentiation modulo p , where p is an integer with several hundred decimal digits, as an example. We now describe how efficient exponentiation yields an effective method of exchanging a secret key.

Alice and Bob send messages agreeing on the value of p , as well as on an integer b between 2 and $p - 1$. Of course, then Eve also knows the values of p and b . Next, Alice and Bob randomly choose integers m and n , respectively, between 1 and $p - 1$. Alice does not know n , and Bob does not know m (Eve knows neither n nor m). Using a modification *PowersMod* of *Powers* in which all calculations are carried out modulo p (see Exercise 3.69b), Alice then calculates and sends the integer $b^m \text{ mod } p$ to Bob. Similarly, Bob calculates and sends the integer $b^n \text{ mod } p$ to Alice. Here comes the secret: Using *PowersMod* again, Alice calculates $(b^m \text{ mod } p)^n \text{ mod } p$, and Bob calculates $(b^n \text{ mod } p)^m \text{ mod } p$. Lo and behold, they have just calculated the same number $b^{mn} \text{ mod } p$, which is the secret key they now share.

Because Eve has monitored all exchanges between Alice and Bob, she now knows p , b , $b^m \text{ mod } p$, and $b^n \text{ mod } p$. Thus, theoretically Eve can determine the secret key $b^{mn} \text{ mod } p$ by calculating m from her knowledge of $b^m \text{ mod } p$ and then calculating $(b^m \text{ mod } p)^n \text{ mod } p$. The problem of determining m from the known values of b , p , and $b^m \text{ mod } p$ is (a special case of) the *discrete logarithm* problem. The naive algorithm solving the discrete logarithm problem is to compute $b^i \text{ mod } p$ to $b^m \text{ mod } p$ as i takes on successive values 1, 2, 3, . . . , until we reach an i such that $b^i \text{ mod } p = b^m \text{ mod } p$. Since this would take $p/2$ iterations on average, the naive algorithm is obviously infeasible. (Recall that p is an integer having more than a hundred digits!)

While there are better algorithms for the discrete logarithm problem, unfortunately for Eve no efficient algorithm is known.

The algorithm *PowersMod* which Alice and Bob used to calculate powers of b requires special techniques for multiplying large integers. A classical divide-and-conquer method for multiplying large integers is discussed in Chapter 4. Since all calculations are reduced modulo p , *PowersMod* always multiplies or squares numbers less than p , so that each such operation takes time bounded above by a constant $C(p)$ depending only on p . Thus, if Alice and Bob choose p , m , and n having 200 decimal digits each, then they can exchange their secret key after performing 3000 multiplications of 200-digit numbers and 3000 reductions of a 400-digit number modulo p . All these computations can be done within a reasonable time.

There are literally thousands of important problems such as the three we have just mentioned that are generally believed to be hard (since nobody has found polynomial solutions for any of them), but which have not been proven to be hard (that is, super-polynomial lower bounds have not been found for any of these problems). Settling the question of whether these problems are truly hard is the most important open question in theoretical computer science today.

Interestingly, the problem of factoring and finding the discrete logarithm have both been shown to admit polynomial quantum algorithms, but, as we have already mentioned, the open question here is whether or not quantum computers of sufficiently large size will ever be practical.

3.7.2 NP-Complete Problems

A decision problem is a problem having a yes or no answer to its inputs. For example, the prime testing problem is the decision problem that asks whether or not a given integer n is prime. Often optimization problems have associated decision problem versions obtained by adding a (goodness measure) parameter and asking whether there is a solution as good as the parameter. For example, in TSP, one could add the parameter k to the problem, and ask whether or not there exists a tour whose total length is not greater than k . Of course, if the optimization problem is solved, the associated decision version is also solved immediately (just compare the length of the optimal tour to the parameter k). Sometimes, at least in special cases, there is a polynomial procedure to solve the optimization problem given a solution to the general decision problem. For example, such a polynomial procedure exists for TSP if all the distances between cities are integers of bounded size.

There is a class of decision problems called NP (for Non-deterministic Polynomial) that can be thought of as problems where candidate solutions to “yes” instances can be constructed (“guessed”) in polynomial time, and a given candidate can be checked to see if it is a solution in polynomial time. For example, given TSP with the parameter k , a candidate solution is simply a permutation of the $n - 1$ cities to be visited, and such a permutation can be constructed (guessed) in linear time. Moreover, given a candidate solution, checking whether or not the length of the tour corresponding to this permutation is not larger than k can clearly be done in linear time, since it amounts to adding n numbers.

The class P is the class of decision problems having a polynomial deterministic solution to all instances. Note that it is clear that $P \subseteq NP$, since the candidate solution to “yes” instances can be taken to be the solution constructed by the algorithm in polynomial time. Since allowing nondeterminism (perfect guessing) should help, one would expect that $P \neq NP$, but this

important question remains unsolved despite the efforts of the best theoretical computer scientists over more than the last 40 years.

The class of NP-complete problems are the problems in NP that, roughly speaking, are as hard to solve (up to polynomial factors) as any other problem in NP. Somewhat more precisely, problem A is as hard to solve as problem B if there is a mapping t from the inputs of size n to problem B to the inputs of size at most $p(n)$ to problem A for a suitable polynomial p , such that t is constructible in polynomial time (and space), and such that an input I to problem B is a “yes” instance if, and only if, $t(I)$ is a “yes” instance to problem A . We say that problem B is (polynomially) *reducible* to problem A . A problem A in NP is *NP-complete* if every problem in NP is reducible to A . It is certainly not obvious that NP-complete problems exist. But, if NP-complete problems exist, and if any one of them belongs to P, then every NP problem belongs to P (which would imply that $P = NP$, an equality generally believed to be false).

The first NP-complete example was found by Cook, who showed in 1970 that the problem of determining whether or not the variables in a given Boolean expression can be assigned values (true or false) in such a way as to make the Boolean expression true (called a satisfying assignment). Levin found an example about the same time. Since then thousands of problems have been shown to be NP-complete. Many NP-complete problems have important applications, so it is frustrating not to know whether they are all super-polynomial in complexity. However, it is true that sometimes important special cases of a given NP-complete problem can be solved, or in some useful sense approximately solved, in polynomial time. We will discuss NP-complete problems and approximation algorithms in more detail in Chapter 10.

Interestingly, prime testing was not known to be in P until 2002, when it was shown to be solvable by a 6th degree polynomial in the size of the input. Good probabilistic algorithms for prime testing have been known for some time, and these algorithms still are more efficient in practice than the recently found deterministic polynomial-time algorithm. However, showing that prime testing is in P was nevertheless a major achievement.

3.8 Closing Remarks

An algorithm’s complexity is often described in the literature in terms of belonging to an O -class, since an O -class bound for $W(n)$ also gives a bound on the computing time for any input of size n . Also, the O -class value for complexity of an algorithm automatically applies to the problem itself. Another reason for using the O -class notation is that it might yield a much simpler measure of the complexity of an algorithm than the more exact order formula using Θ . For example, consider the naïve algorithm for testing whether an integer m is prime by simply successively testing the potential divisors $2, 3, \dots, m^{1/2}$. This algorithm performs only one division when m is even, but will perform $m^{1/2}$ divisions when m is a prime. Thus, the algorithm performs $O(m^{1/2})$ divisions, that is, has $O((\sqrt{2})^n)$ complexity where the input size is the number of binary digits n of m . In this case, the Θ -class for the exact worst-case complexity contains no simply expressible function.

The asymptotic classes O , Θ , Ω do not take into account the size of the constants involved. In practice, as we mentioned before, the constants sometimes do matter. For example, there are algorithms known for solving certain problems that have linear order, but for which the explicit constants involved are so large that the algorithms are totally impractical to implement. Of course, for sufficiently large input size n , a linear algorithm will outperform, say, a quadratic algorithm, but the latter algorithm might have sufficiently small associated explicit and implicit

constants that it outperforms the linear algorithm for any input size n small enough to be of practical interest. The linear algorithm in such cases is then mostly of theoretical interest but nevertheless important: It gives hope that a linear algorithm with reasonably small constants can be found.

Exercises

Section 3.1 Asymptotic Behavior of Functions

3.1 Prove each of the following directly from the definition of Θ .

- a) $100,000,000 \in \Theta(1)$
- b) $n^2/2 + 2n - 5 \in \Theta(n^2)$
- c) $\log_{10}(n^2) \in \Theta(\log_2 n)$

3.2 Prove each of the following directly from the definitions of O and Ω .

- a) $17n^{1/6} \in O(n^{1/5})$
- b) $1000n^2 \in O(n^2)$
- c) $n/1000 - 500 \in \Omega(n)$
- d) $30n\log_2 n - 23 \in \Omega(n)$
- e) $O(n!) \subset O((n+1)!)$

3.3 Repeat Exercise 3.1 using the Ratio Limit Theorem.

3.4 Repeat Exercise 3.2 using the Ratio Limit Theorem.

3.5 Using the Ratio Limit Theorem and (possibly) L'Hôpital's rule, prove each of the following.

- a) $(n^3 + n)/(2n + \ln n) \in \Theta(n^2)$
- b) $O(n^4) \subset O(3^n)$
- c) $n2^n + n^9 \in O(e^n)$
- d) $n^{1/2} \in \Omega((\log n)^4)$
- e) $n^{\log_2 n} \in O(2^n)$

3.6 Using the Ratio Limit Theorem, prove the following.

$$O(108) \subset O(\ln n) \subset O(n) \subset O(n \ln n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(3^n)$$

3.7 For any constants k and $b > 1$, show that

$$O(n^k) \subset O(n^{\ln n}) \subset O(b^n).$$

3.8 Prove property (1) of Proposition 3.1.2: For any positive constant c ,

$$\Omega(f(n)) = \Omega(cf(n)), \Theta(f(n)) = \Theta(cf(n)), \text{ and } O(f(n)) = O(cf(n)).$$

3.9 Prove property (3.1.4):

$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in \Omega(g(n)) \cap O(g(n)).$$

3.10 Show that $f(n) \in \Theta(g(n))$ if, and only if, $\Theta(f(n)) = \Theta(g(n))$.

3.11 Prove properties (3) and (5) of Proposition 3.1.2:

- a) if $f(n) \in O(g(n))$, then $O(f(n)) \subseteq O(g(n))$.
- b) $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$.

3.12 Prove property (4) of Proposition 3.1.2:

$$O(f(n)) = O(g(n)) \Leftrightarrow \Omega(f(n)) = \Omega(g(n)) \Leftrightarrow \Theta(f(n)) = \Theta(g(n)).$$

- 3.13 Prove each of the following.
- $4n^3 + \sqrt{n} \sim 4n^3 - n^2 + 500$
 - $2^n + 50n^5 + 13n^2 + 42 \sim 2^n$
 - $(6n^2 + 50\sqrt{n})/(3\sqrt{n} + 5 \ln n) \sim 2n^{3/2}$
 - $30n - 50 = O(n^2/23)$
 - $1000 \log_2 n = O(\sqrt{n})$
- 3.14 Suppose a and c are positive constants.
- If f is a polynomial, show that $f(n+c) \sim f(n)$ and $f(cn) \in \Theta(f(n))$.
 - If $f(n) = a^n$, show that $f(n+c) \in \Theta(f(n))$ and $O(f(n)) \subset O(f(cn))$, $c > 1$.
 - If $f(n) = n!$, show that $O(f(n)) \subset O(f(n+c))$, (c a positive integer).
- 3.15 Let $P(n)$ be any polynomial of degree k whose leading coefficient is positive, and let a be any real number $a > 1$. Show that $P(n) = O(a^n)$.
- 3.16 Show that if $1 < a < b$, then $a^n = O(b^n)$.
- 3.17 Complete the proof of the Ratio Limit Theorem by proving case 3.
- 3.18 The Ratio Limit Theorem states that $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$ implies that $O(f(n)) \subset O(g(n))$.
- Show that a partial converse is true: If $O(f(n)) \subset O(g(n))$ and $\lim_{n \rightarrow \infty} f(n)/g(n)$ exists, then this limit must be zero.
 - Construct examples where $O(f(n)) \subset O(g(n))$ but $\lim_{n \rightarrow \infty} f(n)/g(n)$ does not exist.
 - Give examples similar to (b) for the other two cases of the Ratio Limit Theorem.
- 3.19 Prove the following result from the definition of Θ and \sim . If $f(n)$ and $g(n)$ are functions such that $g(n) \in O(f(n))$, then
- $$f(n) \pm g(n) \sim f(n).$$
- 3.20 The notions of Ω , Θ , O can be viewed as binary relations on the set F of (eventually positive) functions $f: \mathbb{N} \rightarrow \mathbb{R}$. A *(binary) relation* R on a set S is any subset of the Cartesian product $S \times S$. For $x, y \in S$, we say that x is *related* to y , written xRy , if the ordered pair $(x, y) \in R$. Note that Θ determines a relation on the set F by defining $f\Theta g$ to mean $g(n) \in \Theta(f(n))$. In a similar way, Ω and O determine relations on F .
- A very important class of relations on a set S are the so-called equivalence relations. Equivalence relations on S correspond precisely to partitions of S into pair-wise disjoint subsets (see Exercise 3.22).

Definition

A relation R on S is said to be an *equivalence* relation if the following three properties are satisfied.

1. *Reflexive Property*: xRx , $\forall x \in S$.
2. *Symmetric Property*: $xRy \Rightarrow yRx$, $\forall x, y \in S$.
3. *Transitive Property*: xRy and $yRz \Rightarrow xRz$, $\forall x, y, z \in S$.

- a) Show that the relation Θ is an equivalence relation on \mathcal{F} (that is, verify properties (1), (2), and (3) of Proposition 3.1.1).
- b) Show that relations O and Ω are reflexive and transitive but are not symmetric.
- 3.21 Given an equivalence relation R on a set S , each element $x \in S$ determines an

equivalence class, denoted by $[x]$, consisting of all elements y such that xRy . It is easily proved that $[x] = [y]$ iff xRy . A finite or infinite collection of subsets of a set S is said to be a *partition* of S if the sets are pairwise disjoint and their union is S .

- a) Given an equivalence relation R on a set S , show that the set of equivalence classes is a partition of the set S .
 - b) Conversely, given any partition of the set S , show there is a unique equivalence relation on S whose equivalence classes are precisely the subsets of the given partition (define xRy iff x and y lie in the same subset).
- 3.22 a) Show that the relation \sim is actually an equivalence relation on \mathcal{F} .
- b) Show that \sim is a stronger relation than Θ in the sense that \sim refines Θ . Given two relations R_1, R_2 on a set S , we say that R_2 *refines* R_1 if $xR_2y \Rightarrow xR_1y$. If both R_1 and R_2 are equivalence relations, and R_2 refines R_1 , then each R_2 -equivalence class is contained in an R_1 -equivalence class. We say that the R_2 -equivalence classes form a *refinement* of the R_1 -equivalence classes.
- 3.23 Consider the relation K on \mathcal{F} defined as follows:
- $$f Kg \text{ iff } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L, \text{ where } 0 < L < \infty.$$
- The value of L varies with f and g .
- a) Show that K is an equivalence relation on \mathcal{F} .
 - b) Show that K refines Θ .
 - c) Show that K is refined by \sim .
- 3.24 Show that the functions $f(n) = n^3$ and $g(n) = n^4(n \bmod 2) + n^2$ are not comparable.
- 3.25 Find two strictly increasing functions $f, g \in \mathcal{F}$ whose orders are not comparable.

Section 3.2 Asymptotic Order Formulae for Three Important Series

- 3.26 Obtain a formula for the order of $S(n) = \sum_{i=1}^n (\log i)^2$.
- 3.27 Obtain an approximation for $\log n! = \sum_{i=1}^n (\log i)$ by using a technique similar to that used in Section 3.7 for approximating the harmonic series. (*Hint:* Use $\int_1^n \log x dx$.)
- 3.28 Show that $\lim_{n \rightarrow \infty} H(n) = \infty$.
- 3.29 Show directly from the definition of $S(n,k)$ that $S(n,k) \in \Theta(n^{k+1})$ (that is, without using recurrence relation (2.2.5)).
- 3.30 Show that $S(n, -k) = 1 + \left(\frac{1}{2}\right)^k + \left(\frac{1}{3}\right)^k + \dots + \left(\frac{1}{n}\right)^k \in \Theta(1)$, for all integers $k \geq 2$.
- 3.31 Proposition 7.7.2 states that $S(n,k) = 1^k + 2^k + \dots + n^k$ is a polynomial in n of degree $k+1$ whose leading coefficient is $1/(k+1)$. Using induction and Proposition 3.2.1, show that the coefficients of n^k and n^{k-1} in $S(n,k)$ are $1/2$ and $k/12$, respectively, for $k > 1$.
- 3.32 a) $S(n,k) = 1^k + 2^k + \dots + n^k$ is a polynomial in n of degree $k+1$ whose constant term is zero. Let $B(j,k)$ denote the coefficient of n^j in polynomial $S(n,k)$, $j = 1, \dots, k+1$. Using the recurrence relation (3.2.1) for $S(n,k)$, obtain a recurrence relation for $B(j,k)$.
- b) Using the recurrence relation for $B(j,k)$, obtained in (a), give pseudocode for the

- procedure *SumOfPowers*, which outputs the (coefficients of) the polynomial $S(n,k)$.
- c) Modify your algorithm to avoid truncation errors when dividing by large integers by storing the coefficients $B(j,k)$ as fractions in lowest form. These fractions can be stored as pairs of integers (*Numer*,*Denom*), where *Numer* is the numerator and *Denom* is the denominator. Obtain the lowest form of the fraction represented by (*Numer*,*Denom*) by employing Euclid's gcd algorithm.
- 3.33 Generalize the recurrence relation (3.2.1) for $S(n,k)$ to a recurrence relation for the sum of the k th powers of the first n terms in the arbitrary arithmetic progression

$$S(a,d,n,k) = a^k + (a+d)^k + \cdots + (a+(n-1)d)^k.$$

Section 3.3 Recurrence Relations for Complexities

- 3.34 Derive and solve a recurrence relation for the best-case complexity $B(n)$ of *MergeSort*.
- 3.35 Solve the following recurrence relations:
- $t(n) = 3t(n-1) + n$, $n \geq 1$, **init. cond.** $t(0) = 0$.
 - $t(n) = 4t(n-1) + 5$, $n \geq 1$, **init. cond.** $t(0) = 2$.
 - $t(n) = 2t(n/3) + n$, $n \geq 1$, **init. cond.** $t(0) = 1$. You may assume n is a power of 3.
- 3.36 Determine the Θ -class of $t(n)$ where

$$t(n) = 2t(n-1) + n^4 + 1, \quad \text{init. cond. } t(0) = 0.$$
- 3.37 A very natural question to ask related to *MergeSort* is whether we can do better by dividing the list into more than two parts. It turns out we can't do any better, at least in the worst case. In this exercise you will verify this fact mathematically. *TriMergeSort*, given below, is a variant of *MergeSort*, where the list is split into three equal parts instead of two. For convenience, assume that $n = 3^k$, for some nonnegative integer k .

```

procedure TriMergeSort(Low,High) recursive
Input: Low,High (indices of a global array L[0:n-1], initially Low = 0 and
           High = n - 1 =  $3^k - 1$ )
Output: sublist L[Low:High] is sorted in nondecreasing order
if High ≤ Low then return endif
    Third ← Low + ⌊(High - Low + 1)/3⌋
    TwoThirds ← Low + 2*⌊(High - Low + 1)/3⌋
    TriMergeSort(Low,Third)
    TriMergeSort(Third+1,TwoThirds)
    TriMergeSort(TwoThirds+1,High)
    //merge sublists L[Low:Third] and L[Third:TwoThirds] of sizes  $n/3$  and  $n/3$ 
    Merge(Low,Third,TwoThirds)
    //merge sublists L[Low:TwoThirds] and L[TwoThirds:High] of sizes  $2n/3$  and  $n/3$ 
    Merge(Low,TwoThirds,High)
end TriMergeSort

```

- Give a recurrence relation for the worst-case complexity $W(n)$ of *TriMergeSort* for an input list of size n .
- Solve the recurrence formula you have given in (a) to obtain an explicit formula for the worst-case complexity $W(n)$ of *TriMergeSort*.
- Which is more efficient in the worst case, *MergeSort* or *TriMergeSort*? Discuss.
- Repeat (a), (b), and (c) for the best-case complexity $B(n)$.

- 3.38 Design and analyze a variant of *TriMergeSort* which uses a single call to a merge procedure for merging three sorted sublists, where three pointers are used, one for each sublist.
- 3.39 In this section we established a formula for the worst-case complexity $W(n)$ of *MergeSort* when n is a power of two. Find an approximation to $W(n)$ for a general n , showing that $W(n) \in \Theta(n \log n)$.

Section 3.4 Mathematical Induction and Proving Correctness of Algorithms

- 3.40 Prove by induction that

$$C(n,0) + C(n,1) + C(n,2) + \dots + C(n,n) = 2^n.$$

- 3.41 Prove by induction that

$$C(n,0) - C(n,1) + \dots + (-1)^i C(n,i) + \dots + (-1)^n C(n,n) = 0.$$

- 3.42 Prove by induction that

$$\text{fib}(1) + \text{fib}(2) + \dots + \text{fib}(n) = \text{fib}(n+2) - 1$$

where $\text{fib}(n)$ denotes the n^{th} Fibonacci number.

- 3.43 a) Prove by induction that

$$\begin{pmatrix} \text{fib}(n) \\ \text{fib}(n+1) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

- b) Briefly describe how the preceding formula can be employed to design an algorithm for computing $\text{fib}(n)$ using only at most $8\log_2 n$ multiplications

- 3.44 Consider the famous Towers of Hanoi puzzle (see Appendix B). Prove by induction that the minimum number of moves needed to solve the Towers of Hanoi puzzle is $2^n - 1$ (so that the solution given in Appendix B is optimal).

- 3.45 Suppose you have a collection of n lines in the plane such that no two are parallel and no three meet in a point. Find a formula for the number of regions in the plane determined by these lines, and prove your formula using induction.

- 3.43 Prove *Merge* is correct.

- 3.44 Prove *QuickSort* is correct.

- 3.45 Prove *BubbleSort* (see Exercise 2.34) is correct

- 3.46 Prove *InsertionSort* is correct

- 3.47 Prove *BinarySearch* is correct

- 3.48 Write an iterative version of binary search, and prove the correctness of your algorithm.

- 3.49 Prove *TriMergeSort* (see Exercise 3.37) is correct.

- 3.50 We remarked that *Right-to-Left Binary Powers* is directly related to how the recursive algorithm *Powers* computes x^n , except that *Powers* does not require the explicit binary representation of n . The following is a direct translation of *Powers* into an iterative version that is based on a canonical way of removing tail recursion. Of course, *Powers2* is very similar to *Right-to-Left Binary Powers*, but (as with *Powers*) without requiring the explicit binary representation of n .

```
function Powers2(x,n)
Input: x (a real number), n (a nonnegative number)
Output: xn
```

```

if  $n = 0$  then return(1) endif
AccumPowers  $\leftarrow 1$ 
Pow  $\leftarrow x$ 
while  $n > 1$  do
  if even( $n$ ) then
     $n \leftarrow n/2$ 
  else
    AccumPowers  $\leftarrow$  AccumPowers * Pow
     $n \leftarrow (n - 1)/2$ 
  endif
  Pow  $\leftarrow$  Pow * Pow
endwhile
return(Pow * AccumPowers)
end Powers 2

```

Prove the correctness of *Powers2* using (strong) induction on n (and without explicit use of the binary representation of n).

- 3.51 a) Using mathematical induction prove that the Fibonacci numbers satisfy:

$$(1.5)^{n-2} \leq fib(n) \leq 2^{n-1}, n \geq 1.$$

- b) Applying the upper bound for $fib(n)$ in part (a) and Exercise 2.7 from Chapter 2, show that the worst-case complexity of *EuclidGCD* belongs to $O(n)$.

Section 3.5 Establishing Lower Bounds for Problems

- 3.52 a) Show that removing an inversion $(\pi(i), \pi(i+1))$ by interchanging $\pi(i)$ and $\pi(i+1)$ results in a decrease of (exactly) one in the total number of inversions in the permutation π .
 b) Generalize the result in part a. by showing that interchanging $\pi(i)$ and $\pi(i+k)$ results in a decrease of at most $2k-1$ in the total number of inversions in the permutation π .
- 3.53 Consider any comparison-based sorting algorithm, where each comparison involves elements that are at most k positions apart (that is, elements indexed in positions i and $i+j$ for some $j \leq k$). Use the result from Exercise 3.52 to show that the worst-case complexity $W(n)$ of such an algorithm is at least $(n^2/2 - n/2)/(2k-1)$. Note that this generalizes Proposition 3.5.2.
- 3.53 Show that any comparison-based algorithm for merging two sorted lists of size $n/2$ to form a sorted list of size n cannot have complexity belonging to $O(n^{1-\varepsilon})$ for any $\varepsilon > 0$. Hint: we have already seen that $\Omega(n \log n)$ is a lower bound for comparison-based sorting.
- 3.55 a) For the algorithm *InsertionSort*, exhibit the permutations in each set of the partition C_p , $p = 1, 2, 3$ as described in the proof of Proposition 3.5.4.
 b) Repeat part a) for *MergeSort*.

Section 3.6 Probabilistic Analysis of Algorithms

For Exercises 3.56-3.69, we often refer to Appendix E for propositions and formulas.

- 3.56 Consider the sample space S corresponding to rolling two dice; that is, $S = \{(r_1, r_2) \mid r_1, r_2 \in \{1, \dots, 6\}\}$. Assume that the first die is fair but the second die is loaded, with probabilities $1/10, 1/10, 1/10, 1/10, 1/10, 1/2$ of rolling a 1,2,3,4,5,6, respectively.
- Give a table showing the probability distribution for rolling these dice (the sample space is shown in Figure E.1).
 - Compute the probability that at least one of the dice comes up 6.
 - Compute the conditional probability that the sum of the dice is 10 given that the loaded die does not come up 4.
- 3.57 Consider the random variable $X = r_1 + r_2$ defined on the sample space S given in Exercise 3.56.
- Compute the density function $f(x) = P(X = x)$ and verify that it is a probability distribution on the sample $S_X = \{2, \dots, 12\}$.
 - Calculate the expectation $E[X]$.
- 3.58 Repeat Exercise 3.56 for the random variable $X = \max \{r_1, r_2\}$.
- 3.59 Let S be the sample space consisting of the positive integers. For a fixed p , $0 < p < 1$, show that the function $P(i) = (1 - p)^{i-1}p$ is a probability distribution on S .
- 3.60 Consider the sample space S corresponding to rolling three fair dice; that is, $S = \{(r_1, r_2, r_3) \mid r_1, r_2, r_3 \in \{1, \dots, 6\}\}$. Calculate the expectation $E[X]$ for each of the following random variables X .
- $X = r_1 + r_2 + r_3$
 - $X = r_1 + r_2$
 - $X = \max \{r_1, r_2, r_3\}$
- 3.61 Give an alternate derivation of the expectation of a binomial distribution using Proposition E.1.4. (*Hint:* Let X_i be the random variable that has the value 1 if there was a success in the i th trial, and 0 otherwise.)
- 3.62 Verify that the expectation of the geometric distribution (see Appendix E) with probability p of success is $1/p$.
- 3.63 Prove Proposition E.1.1.
- 3.64 Prove Propositions E.1.3 and E.1.4.
- 3.65 Verify that the probabilities given by (E.1.12) satisfy the three axioms for a probability distribution on $\{0, 1, \dots, n\}$.
- 3.66 Verify that P_F defined by (E.1.24) satisfies the three axioms for a probability distribution on F .
- 3.67 Given a (discrete) random variable X , show that the probability density function $f(x) = P(X = x)$ determines a probability distribution on $S_X = \{x : f(x) \neq 0\}$.
- 3.68 Show that formula (E.1.30) of Proposition E.1.7 reduces to formula (E.1.15) when $X = Y$.
- 3.69 Prove the following generalization of (E.1.5) to n events E_1, E_2, \dots, E_n :
- $$P(E_1 \cap E_2 \cap \dots \cap E_n) = P(E_1)P(E_2 | E_1)P(E_3 | E_1 \cap E_2) \dots P(E_n | E_1 \cap E_2 \cap \dots \cap E_{n-1}).$$
- 3.70 In Chapter 2 we described a (worst-case optimal) algorithm *MaxMin* for determining the maximum and minimum elements in a list $L[0:n - 1]$. The following simple algorithm *MaxMin2* for solving this problem has better best-case complexity, but it is not as efficient in the worst-case. It becomes an interesting question as to what is its average behavior.

$$\text{MaxMin2}(L[0:n - 1], \text{MaxValue}, \text{MinValue})$$

Input: $L[0:n - 1]$ (a list of size n)

Output: $MaxValue, MinValue$ (maximum and minimum values occurring in $L[0:n - 1]$)

```
    MaxValue  $\leftarrow L[0]$ 
    MinValue  $\leftarrow L[0]$ 
    for  $i \leftarrow 1$  to  $n - 1$  do
        if  $L[i] > MaxValue$  then
            MaxValue  $\leftarrow L[i]$ 
        else
            if  $L[i] < MinValue$  then
                MinValue  $\leftarrow L[i]$ 
            endif
        endif
    endfor
end MaxMin2
```

a) Show that $B(n) = n - 1$ and $W(n) = 2n - 2$ for *MaxMin2*

b) Show that $A(n) \sim 2n - \ln n$, so that $A(n)$ is strongly asymptotic to $W(n)$.

Hint: The stated result for $A(n)$ follows from the curious fact that the average number of times $\lambda(n)$ that the maximum is updated by *MaxMin* (and hence also by *MaxMin2*) over all permutations of size n is strongly asymptotic to $\ln n$. To prove this, let $Y:I_n \rightarrow [1,n]$ be defined by $Y(\pi)$ = the position where the maximum occurs for input permutation π .

Using (3.6.6) and the fact that $P(Y=i) = 1/n$, $i = 1, \dots, n$, show that $\lambda(n) = \lambda(n-1) + 1/n$. The result follows by unwinding this recurrence relation for $\lambda(n)$.

Section 3.7 Hard Problems

3.71 a) Show that

$$(x + y) \bmod p = ((x \bmod p) + (y \bmod p)) \bmod p$$
$$(xy) \bmod p = ((x \bmod p)(y \bmod p)) \bmod p.$$

b) Give pseudocode for a modification *PowersMod* of *Powers* in which all calculations are carried out modulo p . The correctness of *PowersMod* follows from (a).

4

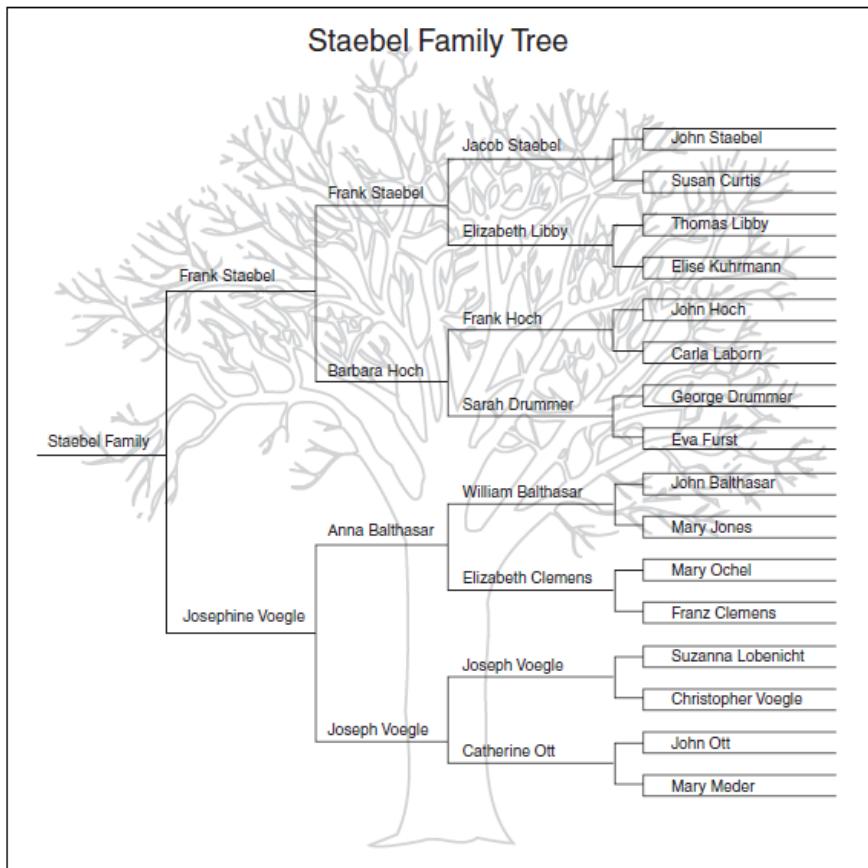
Trees

I think that I shall never see, a poem as lovely as a tree.

—Joyce Kilmer

Computer scientists, too, are enamored with trees. Trees, as defined in computer science and mathematics, sprout up everywhere in the subject of algorithms. Trees are often explicitly implemented in code for an algorithm or are implicitly present in the hierarchical logical organization of an algorithm.

You are no doubt familiar with tree-based hierarchical organizations such as family trees or organizational charts (see Figure 4.1). A table of contents in a book can be thought of as having a tree structure. The directory structure used by the operating system (UNIX, windows, Mac OS, or otherwise) on your computer for organizing files is a tree structure.



A family tree

Figure 4.1

Trees play multiple roles in the design and analysis of algorithms and in computing in general. Often it is very useful to store data in a tree-based hierarchical structure, since efficient access to the data is thereby facilitated. Imposing a tree structure on stored data will often lead to logarithmic complexity of operations versus the linear behavior that would result in viewing the data as a linear structure. The tree structure can either be explicitly or implicitly implemented. For example, we will see in section 4.6 how maintaining a priority queue using a tree-based structure known as a heap will allow addition and deletion of elements to be done in logarithmic time as opposed to the linear time that would result if the priority queue were maintained as a linear data structure. In this case the tree structure is actually overlaid implicitly on the data that is stored contiguously in a (linear) array. As another example, the binary search algorithm for searching an ordered list stored contiguously in an array is based on the implicit binary search tree associated with the data (midpoint element being the root, etc.) In other cases, the data is stored in an explicitly created tree (typically using dynamically allocated storage). For example, balanced tree structures such as red-black trees and B-trees (see Chapter 20) are often explicitly built and maintained to facilitate rapid key location in a database. Internet searches and other textual pattern matching algorithms are usually based on storing pattern strings in explicitly constructed trees, such as suffix trees or tries.

We saw in Chapter 2 that trees are also always implicitly present when modeling the resolution of recursive algorithms. Examining the tree of recursive calls implicitly generated by the complete resolution of a recursive algorithm is often the technique used to measure the complexity of the algorithm. Comparison and decision trees modeling the action of algorithms are other ways in which trees are implicitly present in algorithm analysis. For example, a comparison tree modeling the action of a comparison-based algorithm for sorting a list of size n can be used to establish lower bounds for the complexity of the algorithm. These lower bounds follow from various mathematical properties of trees. Elementary properties of trees, such as depth, will be discussed in Section 4.6, whereas some more sophisticated properties, such as leaf path length, are the subject of Section 4.8.

In addition to all of the above ways that trees arise in the design and analysis of algorithms, their topological structure is frequently utilized as the basis for solving problems related to networks, such as the Internet, wired and wireless communication networks, computer networks, and so forth. For example, in Chapter 7 the fundamental problem of finding a minimal collection of links connecting the nodes in a network is solved by constructing a spanning tree of minimal cost. Another fundamental problem, discussed in Chapters 6 and 8, that of finding paths of shortest length from a given node in a connected network to all other nodes, is solved by constructing a shortest-path spanning tree.

4.1 Definitions

There are a thousand hacking at the leaves of evil to one who is striking at the root. —
Henry Thoreau

In this section we give the formal mathematical definition of trees, and establish some of their elementary properties. Our trees are actually *rooted* trees, but we will usually simply

refer to them as trees. We begin our discussion by defining a special class of trees known as binary trees. These are simply trees where every node has at most two children.

4.1.1 Binary Trees

There are many equivalent definitions of a (rooted) binary tree, but the following recursive definition is certainly one of the most elegant.

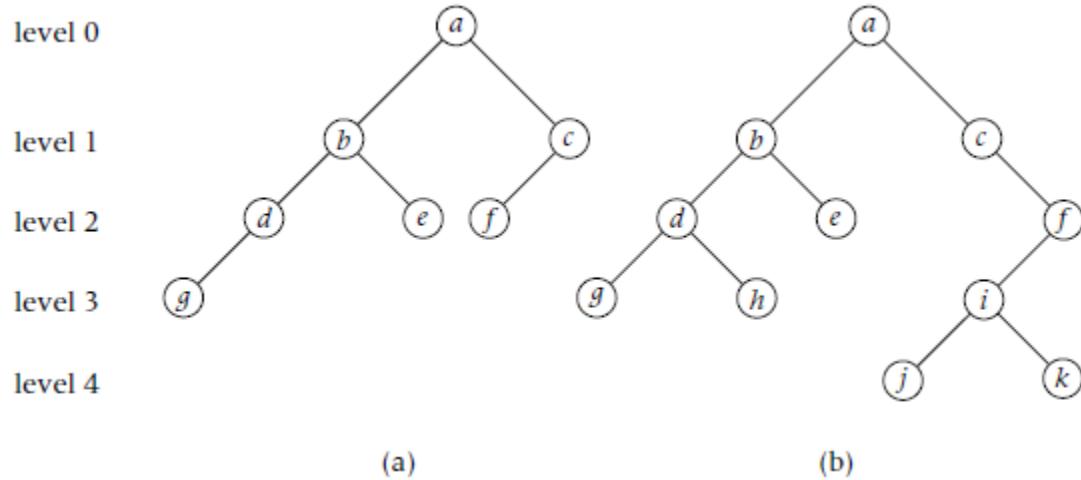
Definition 4.1.1

A binary tree T consists of a set of nodes, which is either empty or has the following properties:

1. One of the nodes, say R , is designated the root node.
2. The remaining nodes (if any) are partitioned into two disjoint subsets, called the left subtree and right subtree, respectively, each of which is a binary tree.

The roots of the left and right subtrees described in property (2) of the definition are called the left child and right child of R , respectively. Some examples of binary trees are given in Figure 4.2. A node is called a *leaf* if it has no children. Just as in family trees, we utilize genealogical notation such as *parent*, *grandchildren*, *great-grandchildren*, *sibling*, *descendant*, *ancestor*, and so forth.

The nodes of T can be partitioned into disjoint sets (levels) depending on their (genealogical) distance from the root R . In particular, the root R is at level 0. The *depth* (also called *height*) of T is the maximum distance from R to a leaf. The trees shown in Figure 4.2a and 4.2b have depths 3 and 4, respectively.



Sample binary trees of depths 3 and 4

Figure 4.2

4.1.1 General Trees

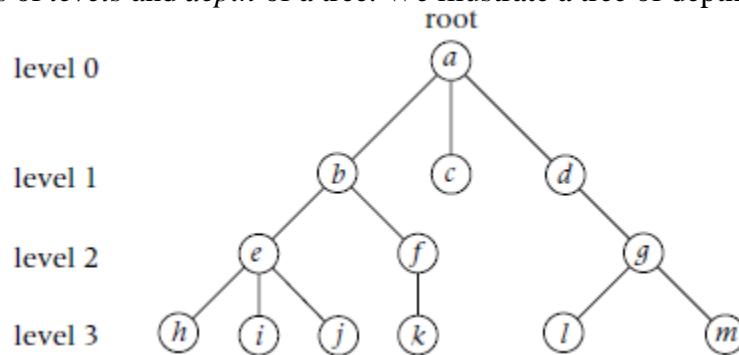
The design of many algorithms requires more general trees than binary trees. General trees are often implicitly present in the logical organization of an algorithm. The following definition extends the concept of a (rooted) binary tree given by Definition 4.1.1 to a general (rooted) tree.

Definition 4.1.2

A tree T consists of a set of nodes (also called vertices), which is either empty or has the following properties:

1. One of the nodes, say R , is designated the *root* node.
2. The remaining nodes (if any) are partitioned into j disjoint subsets T_1, T_2, \dots, T_j , each of which is a tree (called subtrees).

Given the root R , the roots of the subtrees described in (2) are called *children* of R and R is called the *parent*. A node is called a *leaf* if it has no children. It should be noted that when applying property 2 to a subtree, the value of j will, in general, vary with the subtree. Just as for binary trees, we utilize genealogical terminology such as *parent*, *grandchildren*, *great-grandchildren*, *sibling*, *descendant*, *ancestor*, and so forth. We also have the notions of *levels* and *depth* of a tree. We illustrate a tree of depth 3 in Figure 4.3.



A sample tree of depth 3

Figure 4.3

The unordered pair consisting of a node and its parent is referred to as an edge. We leave the following proposition as an exercise.

Proposition 4.1.1 The number of edges of a tree is one less than the number of nodes.

4.2 Mathematical Properties of Binary Trees

The analysis of the complexity of many problems and algorithms discussed in this text depends on various mathematical properties of binary trees. In this section we establish a number of these properties relating to depth, internal path length, and leaf path length.

Given a binary tree T , for ease of discussion, we denote the number of nodes, the number of leaf nodes, the number of internal nodes, and the depth by $n = n(T)$, $L = L(T)$, $I = I(T)$, and $d = d(T)$, respectively. Let n_i denote the number of nodes of T at level i , $i = 0, \dots, d$.

\dots, d . Since T is a binary tree, each node of T has at most two children, which yields the following recurrence relation:

$$n_i \leq 2n_{i-1}, \quad i = 1, \dots, d, \quad \text{init cond. } n_0 = 1. \quad (4.2.1)$$

Thus, a simple induction yields:

$$n_i \leq 2^i, \quad i = 0, \dots, d. \quad (4.2.2)$$

For a given level $i \in \{1, \dots, d\}$, we say that T is full at level i , if $n_i = 2^i$.

4.2.1 Complete and Full Binary Trees

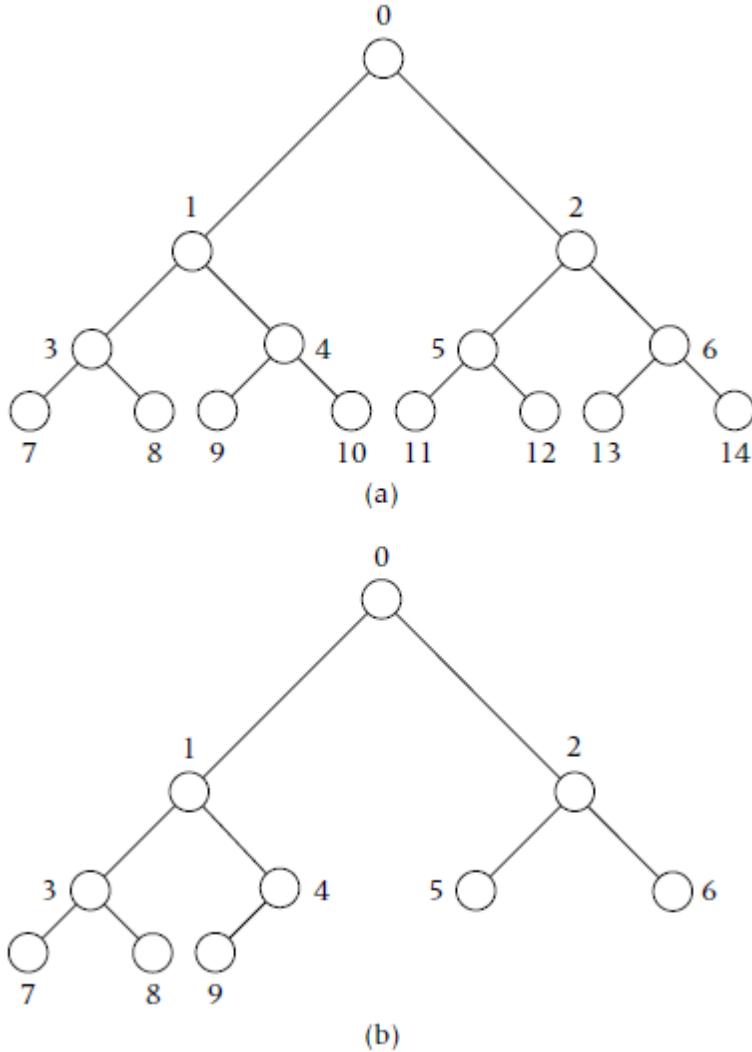
It can easily be verified that if a binary tree T is full at level i , then it is full at every level j smaller than i . A binary tree T is *full* if it is full at every level. Equivalently, a binary tree is full if it is full at the last level. Note that a full binary tree exists only for those n satisfying:

$$n = 1 + 2 + 2^2 + \dots + 2^d = 2^{d+1} - 1. \quad (4.2.1)$$

We denote the full binary tree on n nodes by T_n , $n = 1, 3, 7, 15, \dots$. The full binary tree T_{15} is shown in Figure 4.4a. Solving for d in (4.2.1) yields the following formula for the depth of T_n

$$d = \log_2(n+1) - 1 = \lfloor \log_2 n \rfloor. \quad (4.2.2)$$

Full binary trees are special cases of complete binary trees, where n is one less than a power of 2. To define the *complete* binary tree T_n for a general n , we let k be the (unique) positive integer such that $2^k - 1 < n \leq 2^{k+1} - 1$. Then, for n different from $2^{k+1} - 1$, T_n is defined as a canonical subtree of the full binary tree T_n , $n = 2^{k+1} - 1$ having the same depth and differing from T_n only at the last level. The $q = n - (2^k - 1)$ nodes of T_n on the last level occupy the left-most q positions of T_n . More precisely, consider the following labeling of the nodes of T_n : label the root node 0. Inductively, label the left and right children of vertex i as $2i + 1$ and $2i + 2$, respectively. Then the complete binary tree T_n on n nodes is the subtree of T_n consisting of those nodes labeled $0, 1, \dots, n - 1$ (see Figure 4.4b illustrating T_{10}).



a) Full binary tree T_{15} ; b) complete binary tree T_{10}

Figure 4.4

Since the depth of T_n , $2^k - 1 < n \leq 2^{k+1} - 1$, is the same as the depth k of the full tree on $2^{k+1} - 1$ nodes obtained by filling out the last level, and since $k = \lfloor \log_2 n \rfloor$, we have the following proposition for the depth of an arbitrary complete tree on n nodes.

Proposition 4.2.1 The depth of the complete binary tree T_n for a general n is given by

$$d(T_n) = \lfloor \log_2 n \rfloor. \quad (4.2.3)$$

Since T_n is as full as possible for a binary tree on n nodes, it is not surprising that T_n has minimum depth for all binary trees on n nodes. The latter fact and other mathematical properties of binary trees are discussed in the next section.

Remark

The terminology for complete and full binary trees varies in the literature. For example, sometimes the name *complete* is used for what we have called *full* and the name *left complete* is used for what we have called *complete*.

4.2.2 Logarithmic Lower Bound for the Depth of Binary Trees

The following proposition establishes the useful and intuitively clear result that the depth of a binary tree having n nodes is at least the depth of the complete binary tree T_n on n nodes.

Proposition 4.2.2

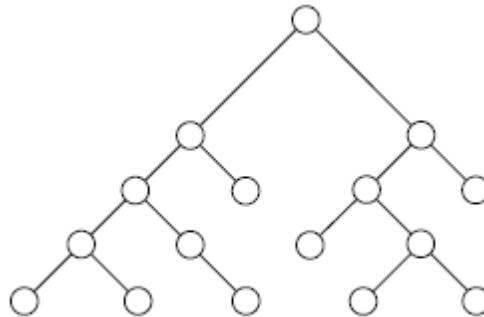
Suppose T is any binary tree having n nodes. Then T has depth at least $\lfloor \log_2 n \rfloor$, that is,

$$D(T) \geq \lfloor \log_2 n \rfloor. \quad (4.2.4)$$

The lower bound $\lfloor \log_2 n \rfloor$ is achieved for the complete binary tree T_n .

Proof The argument follows a similar pattern to the argument that led to (4.2.3), except inequalities are used in place of equalities. We leave the complete proof as an exercise.

For $n = 2^k - 1$, then T_n is the only binary tree whose depth achieves the lower bound of $\lfloor \log_2 n \rfloor$. For a general n (different from $n = 2^k - 1$), there are many binary trees on n nodes that achieve the lower bound depth of $\lfloor \log_2 n \rfloor$. In Figure 4.5, a tree having 16 nodes and depth 4 ($= \lfloor \log_2 16 \rfloor$) is given, which is quite different from the complete tree T_{16} .



A noncomplete tree T with $n = 16$ having depth $= \log_2 16 = 4$

Figure 4.5

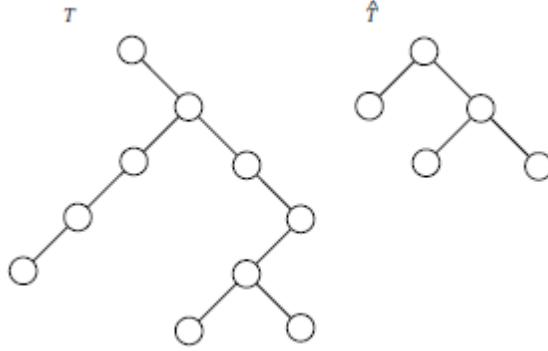
To establish various other mathematical properties of binary trees, we need to introduce the notion of a 2-tree.

4.2.3 2-Trees

A *2-tree* is a binary tree such that every node that is not a leaf has exactly two children. Any binary tree T can be canonically mapped to an associated 2-tree \hat{T} having the same

number of leaf nodes and no larger depth. The 2-tree \hat{T} is obtained from T as follows: If a node of T has one child, then we contract the edge joining this node with its child (so that the node is identified with its child). This process is repeated until we end up at the 2-tree \hat{T} (see Figure 4.6). Clearly, $L(T) = L(\hat{T})$, and $d(T) \geq d(\hat{T})$.

The following proposition establishes a simple relationship between the number of internal nodes and the number of leaf nodes of a 2-tree.



A binary tree T and its associated 2-tree \hat{T}

Figure 4.6

Proposition 4.2.3 Suppose T is any 2-tree. Then the number of leaf nodes is one greater than the number of internal nodes of T ; that is,

$$I(T) = L(T) - 1. \quad (4.2.5)$$

Equivalently, we have

$$n(T) = 2L(T) - 1. \quad (4.2.6) \square$$

Proposition 4.2.3 is easily proved by induction. Proposition 4.2.3 fails dramatically for binary trees in general, since an entirely skewed binary tree has only one leaf node.

Proposition 4.2.4

Suppose T is any binary tree. Then the depth of T satisfies

$$d(T) \geq \lceil \log_2 L(T) \rceil. \quad (4.2.7)$$

Proof Consider the 2-tree \hat{T} associated with T . By Proposition 4.2.3, $n(\hat{T}) = 2L(\hat{T}) - 1$. Using Proposition 4.2.2 and the fact that $\lfloor \log_2(2n - 1) \rfloor = \lceil \log_2 n \rceil$ for any integer $n > 1$, we have

$$d(\hat{T}) \geq \lfloor \log_2(2L(\hat{T}) - 1) \rfloor = \lceil \log_2 L(\hat{T}) \rceil.$$

Since $L(T) = L(\hat{T})$ and $d(T) \geq d(\hat{T})$, it follows that

$$d(T) \geq d(\hat{T}) \geq \lceil \log_2 L(\hat{T}) \rceil = \lceil \log_2 L(T) \rceil. \quad \blacksquare$$

Recall that a binary tree T is called full at level i if T contains the maximum number 2^i nodes at that level. A full binary tree T is full at every level i , $i = 0, \dots, d(T)$. Proofs of the following two propositions are left to the exercises.

Proposition 4.2.5

Suppose T is a 2-tree. Then T is full at the second-deepest level if, and only if, all the leaf nodes are contained in two levels ($d - 1$ and d). \square

Proposition 4.2.6

If a 2-tree T is full at the second-deepest level, then the depth $d(T)$ and the number of leaf nodes $L(T)$ are related by

$$d(T) = \lceil \log_2 L(T) \rceil. \quad (4.2.8) \square$$

4.2.4 Internal and Leaf Path Lengths of Binary Trees

Inputs to search algorithms that are based implicitly (such as *BinarySearch*) or explicitly on binary trees usually follow paths from the root to an internal or leaf node. Moreover, the number of basic operations performed by the algorithm is usually proportional to the length of such a path. In particular, determining the average complexity of such algorithms requires computing the sum of the lengths of all paths from the root to nodes in the tree, which motivates the following definitions. The *internal path length* $IPL(T)$ of a binary tree T is defined as the sum of the lengths of the paths from the root to the internal nodes as the internal nodes vary over the entire tree. The *leaf path length* $LPL(T)$ is defined similarly. Note that the length of a path from the root to a node at level i is i .

When the binary tree T is a 2-tree, then $LPL(T)$ is precisely $2I$ more than $IPL(T)$. This simple relationship between $IPL(T)$ and $LPL(T)$ for 2-trees is stated as Proposition 4.2.7; its proof is left as an exercise.

Proposition 4.2.7

Given any 2-tree T having I internal nodes,

$$IPL(T) = LPL(T) - 2I. \quad (4.2.9) \square$$

The following theorem establishes a useful lower bound for the leaf path length of a binary tree in terms of the number of leaf nodes.

Theorem 4.2.8

Given any binary tree T with L leaf nodes

$$LPL(T) \geq L \lfloor \log_2 L \rfloor + 2(L - 2^{\lfloor \log_2 L \rfloor}). \quad (4.2.10)$$

Moreover, inequality (4.2.10) is an equality if, and only if, T is a 2-tree that is full at the second-deepest level.

Proof We first verify that inequality (4.2.10) is an equality if T is a 2-tree that is full at the second-deepest level. Suppose that $L = 2^k$, so that $\lfloor \log_2 L \rfloor = \log_2 L = k$. Then T is a full binary tree, and (4.2.10) correctly yields the equality $LPL(T) = L \log_2 L$. Now suppose $2^{k-1} < L < 2^k$ for a suitable $k > 1$. By Proposition 4.2.4, T has depth $d = \lceil \log_2 L \rceil$. Since T is full at level $d-1 = \lfloor \log_2 L \rfloor$, each of the L paths in T from the root to a leaf reach level $d-1$. Hence, the total contribution to $LPL(T)$ from reaching level $d-1$ is $L \lfloor \log_2 L \rfloor$, so that

$$LPL(T) = L \lfloor \log_2 L \rfloor + \text{the number of nodes at level } d.$$

If x denotes the number of nodes at level $d-1$ having two children (non-leaf nodes), then $2x$ equals the number of nodes at level d . The remaining $2^{d-1} - x$ nodes at level $d-1$ are leaf nodes, so that $L = (2^{d-1} - x) + 2x$. Thus, we have $x = L - 2^{d-1}$, and $LPL(T) = L \lfloor \log_2 L \rfloor + 2(L - 2^{\lfloor \log_2 L \rfloor})$.

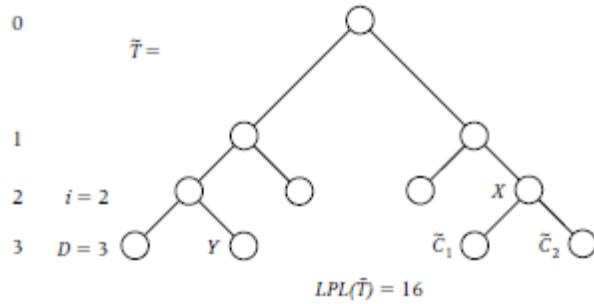
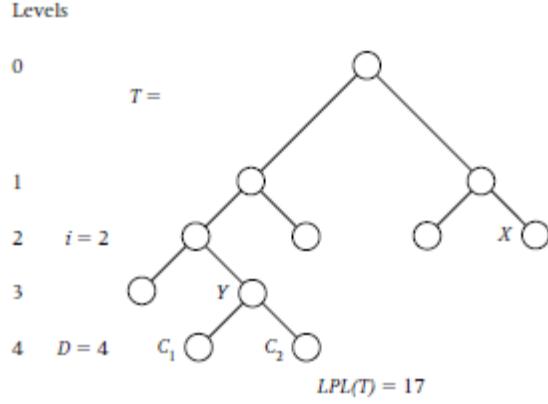
We now verify that inequality (4.2.10) holds for any binary tree. Clearly, if T is any binary tree with L leaf nodes, then its associated 2-tree \tilde{T} has L leaf nodes and $LPL(\tilde{T}) \leq LPL(T)$. Thus, to complete the proof of Theorem 4.2.8, it is sufficient to show that $LPL(T)$ is not minimized for 2-trees T having L leaf nodes that are not full at the second-deepest level.

Suppose T is a 2-tree of depth d having L leaf nodes that is not full at level $d-1$. We show that there then exists a 2-tree T having L leaf nodes such that $LPL(\tilde{T}) < LPL(T)$. Since level $d-1$ is not full, there exists a leaf node, say X , at some level $i \leq d-2$. Let Y be any node at level $d-1$ that is not a leaf node, and let C_1 and C_2 denote the children of Y . We construct \tilde{T} from T by removing the leaf nodes C_1 and C_2 and adding two new leaf nodes \tilde{C}_1 and \tilde{C}_2 as the children of X (see Figure 4.7). Let $L(T)$ and $L(\tilde{T})$ denote the set of all leaf nodes of T and \tilde{T} , respectively. Then,

$$L(\tilde{T}) = (L(T) - \{C_1, C_2\}) \cup \{\tilde{C}_1, \tilde{C}_2, Y\}.$$

Since C_1 , C_2 , and X are at levels d , d , and i in T , respectively, and \tilde{C}_1 , \tilde{C}_2 , and Y are at levels $i+1$, $i+1$, and $d-1$ in \tilde{T} , respectively, it follows that

$$LPL(\tilde{T}) = LPL(T) - (2d + i) + (2(i+1) + d - 1) = LPL(T) - d + i + 1 < LPL(T). \blacksquare$$



Construction of \tilde{T} from T used in the proof of Theorem 4.2.8

Figure 4.7

Corollary 4.2.9

If T is any binary tree having L leaf nodes, then

$$LPL(T) \geq \lceil L \log_2 L \rceil. \quad (4.2.11)$$

Further, if T is a full binary tree, then inequality (4.2.11) is an equality.

Proof Inequality (4.2.11) follows immediately from inequality (4.2.10) of Theorem 4.2.8 by using the inequality

$$x \lfloor \log_2 x \rfloor + 2(x - 2^{\lfloor \log_2 x \rfloor}) \geq \lceil x \log_2 x \rceil, \text{ for any integer } x \geq 1.$$

Moreover, we have already observed in the proof of Theorem 4.2.8 that if T is a full binary tree, then $LPL(T) = L \log_2 L = \lceil L \log_2 L \rceil$. ■

4.2.5 Number of Binary Trees

For any nonnegative integer n , let b_n denote the number of different binary trees on n nodes. The following theorem, which is proved in Appendix D, gives a simple formula b_n .

Theorem 4.2.10

Let b_n denote the number of different binary trees on n nodes. Then,

$$b_n = \frac{1}{n+1} \binom{2n}{n}, \quad n \geq 1 \quad (4.2.12)$$

The value $b_n = \frac{1}{n+1} \binom{2n}{n}$, $n \geq 1$, occurs frequently in enumeration and is also known as the n th Catalan number. Formula (4.2.12), together with the lower bound estimate $\binom{2n}{n} \geq \frac{4^n}{(2n+1)}$, $n \geq 1$ (see Exercise 4.14), so that the number of binary trees on n nodes grows exponentially with n .

4.3 Implementation of Trees and Forests

4.3.1 Array Implementation of Complete Trees

The labeling of the nodes of the complete binary tree T_n illustrated in Figure 4.4 is particularly useful when the tree is implemented using an array $T[0:n - 1]$, where $T[i]$ corresponds to the i th node in the labeling, $i = 0, \dots, n - 1$. For example, in Section 4.6 we will see how the heap ADT is implemented very effectively using the array implementation of T_n . Writing code for algorithms implementing the creation of a heap, and deletion and insertion operations, is greatly facilitated using the fact that the children of the i^{th} node $T[i]$ are $T[2i + 1]$ and $T[2i + 2]$, and the parent of the i th node is $T[\lfloor (i - 1)/2 \rfloor]$.

4.3.2 Implementing General Binary Trees Using Dynamic Variables

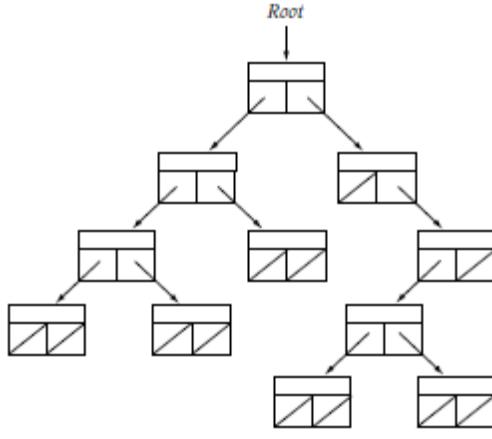
Binary trees can be implemented using pointer and dynamic variables as follows. The nodes of the tree are represented by a dynamic variable having the following structure.

```
BinaryTreeNode = record
  Info: InfoType
  LeftChild: → BinaryTreeNode
  RightChild: → BinaryTreeNode
end BinaryTreeNode
```

In addition to the dynamic variables representing the nodes of the binary tree, we have a pointer variable `Root` pointing to the root node. The implementation of the binary tree from Figure 4.2b using the pointer and dynamic variables is given in Figure 4.8.

Info	
LeftChild	RightChild

(a)



(b)

- (a) Representation of a node of a binary tree; (b) implementation of a binary tree using pointers and dynamic variables

Figure 4.8

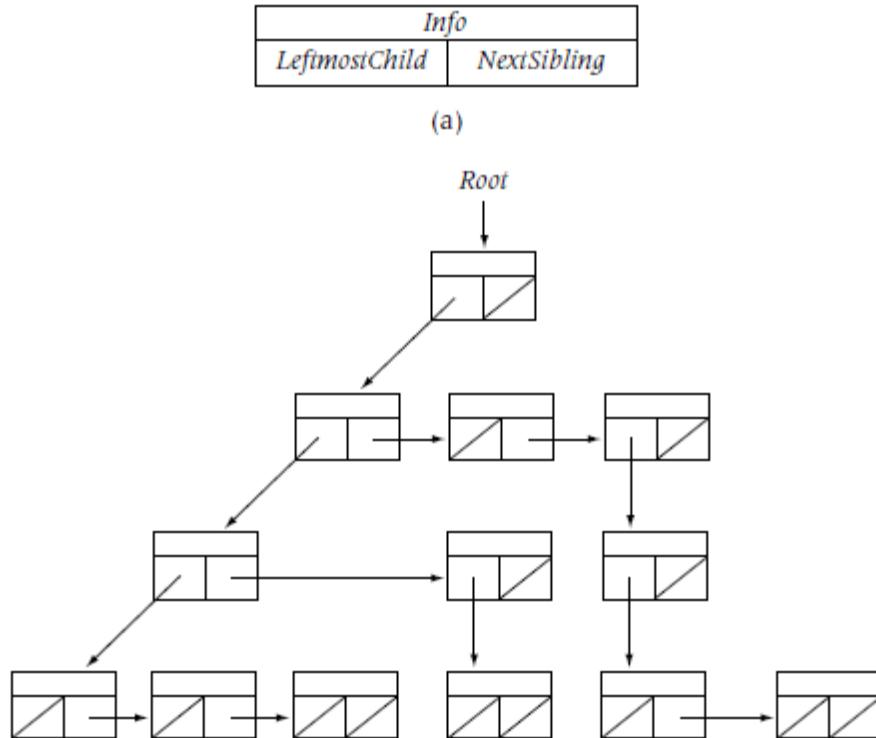
4.3.3 Child-Sibling Representation of a General Tree

A child-sibling representation of a tree T can be implemented using pointer and dynamic variables as follows. The nodes of the tree are represented by the following structure.

```

TreeNode = record
  Info: InfoType
  LeftmostChild: → TreeNode
  NextSibling: → TreeNode
end TreeNode
  
```

As with binary trees, we keep a pointer variable `Root` that points to the root node of the tree. For example, the tree from Figure 4.3 is redrawn in Figure 4.9 using pointers and dynamic variables.



(a) Representation of a node of a general tree; (b) child-sibling implementation of a tree using pointers and dynamic variables

Figure 4.9

Remark

This implementation of a general tree leads naturally to a transformation (known as the Knuth transformation) from a general tree to a binary tree. We simply think of the *LeftmostChild* as the left child of the node and the *NextSibling* as the right child. Indeed, Figure 4.9, showing the general tree implementation of the tree from Figure 4.3, is actually a binary tree.

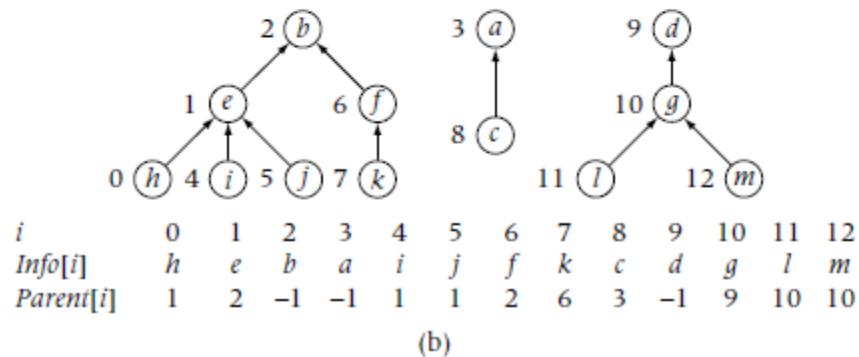
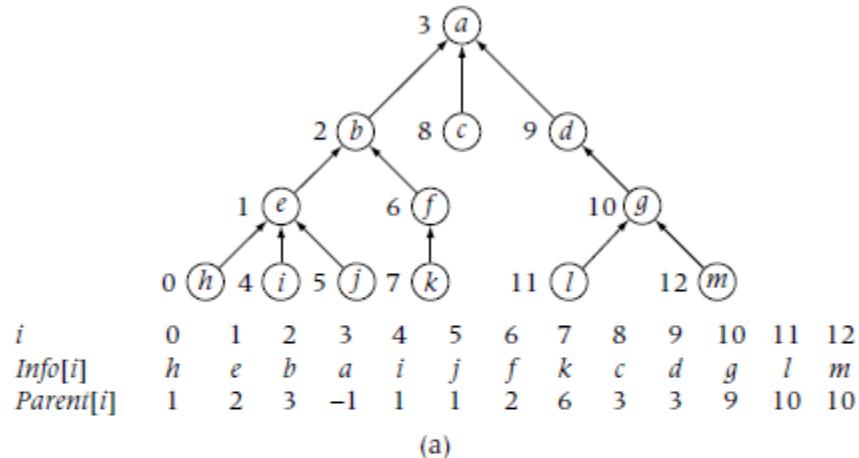
4.3.4 Parent Representation of Trees and Forests

Another representation of a general rooted tree, called the *parent representation*, keeps track of the parent of a node instead of its children. The parent representation has the advantage of efficiently generating the path in the tree from any given node of the tree to the root. However, the parent representation does not facilitate traversals of the tree.

The parent representation can actually be used to implement a slightly more general ADT known as a rooted forest. A *rooted forest* is a union F of (pairwise disjoint) rooted trees. A natural way to implement the parent representation of a (rooted) forest F uses an array $\text{Parent}[0:n - 1]$. For convenience, we assume that F has n nodes labeled $0, 1, \dots, n - 1$. When referring to these nodes, we do not distinguish between a node and its label. For each $i \in \{0, 1, \dots, n - 1\}$, $\text{Parent}[i]$ is -1 , if i is a root of one of the trees in F , and $\text{Parent}[i]$ is the parent of node i , otherwise. Information associated with the node i , $i = 0, \dots, n - 1$, is stored in $\text{Info}[i]$.

$\dots, n - 1$, can be stored in a second array $Info[0:n - 1]$. In Figure 4.10a the values of the two arrays $Parent$ and $Info$ are given for the tree from Figure 4.3. The number beside each node of the tree in the diagram is the label of that node, and the character inside each node is the information stored in that node. In Figure 4.10b the parent array representation of a sample forest is given.

$$7, Parent[7] = 6, Parent[6] = 2, Parent[2] = 3$$



(a) Parent array implementation of a tree; (b) parent array implementation of a forest

Figure 4.10

The parent representation of a forest can also be implemented using pointers and dynamic variables. The nodes of the forest in this representation are given by the following structure.

```

ForestNode = record
  Info: Infotype
  Parent: → ForestNode
end ForestNode
  
```

The parent field of a node contains a pointer to the parent of that node if it has a parent. If the node corresponds to a root of one of the trees in the forest, then it contains the value **null**. The parent implementation of trees and forests has many important applications. Many of these applications exploit the fact that the parent implementation of a tree T allows efficient generation of the *path* P in T from any given node u of T to the root r . (The path in T from u to r is the sequence of nodes u_0, u_1, \dots, u_p , where $u_0 = u$ and $u_p = r$, such that u_i is the parent of u_{i-1} , $i = 1, 2, \dots, p$.) For example, in Figure 4.10a we generate the path in the indicated tree T from vertex 8 to the root 4 as follows:

$$8, \text{Parent}[8] = 7, \text{Parent}[7] = 3, \text{Parent}[3] = 4$$

4.4 Tree Traversal

When a tree is implemented as a data structure in an algorithm, the nodes of the tree contain information (data) germane to the application in question. In many applications it is necessary to access the fields in every node of the tree. During the execution of an algorithm, the current value of a variable (pointer variable, index variable, and so forth) that refers to a node in the tree thereby makes the fields in a particular node accessible. A node is visited when the node is accessed and some operation is performed on the information field(s) contained in the node. For example, visiting a node might consist in simply printing an information field. A tree is traversed if all its nodes are visited. The number of node accesses performed by a tree-traversal algorithm is used to measure the efficiency of the algorithm. (Normally, a node is visited only once in a traversal, but may be accessed several times.)

4.4.1 Traversing Binary Trees

We discuss three standard traversals of a binary tree T , namely, the preorder, inorder, and postorder traversals of T . Each traversal can be defined recursively, as illustrated by the following definition of preorder traversal.

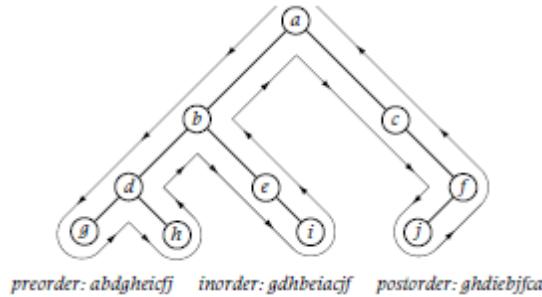
Definition 4.4.1

The preorder traversal of T is defined recursively as follows:

If T is not empty, then

1. visit the root node R ,
2. perform a preorder traversal of the left subtree LT of the root,
3. perform a preorder traversal of the right subtree RT of the root.

Note that no action is performed in the preorder traversal of an empty tree T . The inorder and postorder traversals of T differ from the preorder traversal only in the order in which the root is visited. In the inorder traversal the root is visited after the left subtree tree has been traversed. In the postorder traversal the root is visited last (see Figure 4.11).



Here visiting a node consists of printing the single character information field. All three traversals have the same *path around the tree* as indicated.

Figure 4.11

In Figure 4.11, the information in each node of the binary tree is a single alphabetical character. Assuming that visiting a node consists of simply printing the information stored in the node, then the resulting character strings printed out when the tree is traversed in preorder, inorder, and postorder are *abdgheicjf*, *gdhbeiacjf*, and *ghdiebjfca*, respectively.

All three traversals have the same “path around the tree,” as shown in Figure 4.11. The difference between them is the order in which visits occur. Figure 4.12 illustrates the sequence of accesses and visits for the tree given in Figure 4.12 for each of the three traversals.

preorder traversal	inorder traversal	postorder traversal
access	access	access
<i>a</i>	visit	<i>a</i>
<i>b</i>	visit	<i>b</i>
<i>d</i>	visit	<i>d</i>
<i>g</i>	visit	<i>g</i>
<i>d</i>		<i>d</i>
<i>h</i>	visit	<i>h</i>
<i>d</i>		<i>d</i>
<i>b</i>	visit	<i>b</i>
<i>e</i>	visit	<i>e</i>
<i>i</i>	visit	<i>i</i>
<i>e</i>		<i>e</i>
<i>b</i>		<i>b</i>
<i>a</i>		<i>a</i>
<i>c</i>	visit	<i>c</i>
<i>f</i>	visit	<i>f</i>
<i>j</i>	visit	<i>j</i>
<i>f</i>		<i>f</i>
<i>c</i>		<i>c</i>
<i>a</i>		<i>a</i>

Accessing versus visiting for the binary tree given in Figure 4.11 for preorder, inorder and postorder

Figure 4.12

The traversals preorder, inorder, and postorder have many applications. For example, preorder and postorder traversals of algebraic expression trees generate *prefix* and *postfix* expressions, respectively (see Exercise 4.19). A postorder traversal can be used to free all the nodes in a dynamically allocated tree in order to delete the tree, whereas a preorder traversal in such a tree can be used to copy the tree. A preorder traversal of an appropriate binary tree can be used to generate a binary coding for a given alphabet for text compression purposes (see Chapter 7). Inorder traversals are particularly useful in connection with binary search trees. In particular, an inorder traversal of a binary search tree visits the nodes in sorted order, which is the basis for the sorting algorithm *Treesort* given in Section 4.5.6.

We now give a recursive procedure for a preorder traversal of the binary tree T implemented using pointers and dynamic variables, where the pointer variable Root points to the root of T . The procedure Visit performs some action with the data in Info , such as printing it out.

```
procedure Preorder(Root) recursive
Input: Root (pointer to the root node of a binary tree  $T$ )
Output: the preorder traversal of  $T$ 
  if Root  $\neq \text{null}$  then
    Visit(Root→Info)
    Preorder(Root→LeftChild)
    Preorder(Root→RightChild)
  endif
end Preorder
```

The procedures *Inorder* and *Postorder* have pseudocode identical to *Preorder* except for the order of the call to *Visit*.

We now analyze the performance of *Inorder* (a similar analysis holds for *Preorder* and *Postorder*). The two main operations performed by *Inorder* are visiting a node (calls to *Visit*) and accessing a node. Since *Inorder* visits each node of T exactly once, the total number of node visits is n . During the performance of *Inorder*, each edge (line joining two nodes) in the tree T is traversed exactly twice, in opposite directions. For each edge traversal, the endpoint node (in the direction of the traversal) of the edge is accessed. Since the number of edges of a tree is one less than the number of nodes, T has $n - 1$ edges. It follows that *Inorder* performs $2(n - 1)$ node accesses, plus one additional initial access of the root. Therefore, the total number of node accesses performed by *Inorder* (or by *Preorder* or *Postorder*) is

$$1 + 2(n - 1) = 2n - 1.$$

4.4.4 Traversing General Trees

The three standard traversals of a binary tree can be generalized to any tree as follows. In fact, if the tree is implemented using the leftmostchild-nextsibling representation, then these traversals simply reduce to the corresponding traversals of a binary tree. We can also define the preorder, inorder, and postorder traversals of a general T directly in terms of its recursive Definition 4.1.2.

Definition 4.4.2

The preorder traversal of T is defined recursively as follows:

If T is not empty, then

1. visit the root node R ,
2. perform successively a preorder traversal of T_1, T_2, \dots, T_j .

Again, the postorder traversal is identical to the preorder traversal except that steps 1 and 2 are interchanged. The definition of the inorder traversal of a general T is not as natural as it was for binary trees, since there are many intermediate orders in which to perform a visit. In Figure 4.13 we formulate an inorder traversal that visits a node immediately after traversing the leftmost subtree of the node.

<i>preorder:</i>	visit R	traverse T_1	traverse $T_2 \dots$	traverse T_j
<i>inorder:</i>	traverse T_1	visit R	traverse $T_2 \dots$	traverse T_j
<i>postorder:</i>	traverse T_1	traverse $T_2 \dots$	traverse T_j	visit R

Preorder, inorder and postorder traversals of general tree T

Figure 4.13

Pseudocode for implementing preorder, inorder, and postorder traversals of general trees, as well as their generalizations to traversals of rooted forests, is developed in the exercises. As with binary trees, each of the standard traversals of general trees has its particular uses. For example, what amounts to a postorder traversal of a forest arises in an algorithm for determining the strongly connected components of the digraph (see Chapter 13). It turns out that a multivisit version of an inorder traversal is very useful in multiway search trees.

Traversals with Multivisits

For general trees, visiting a node might occur more than once in the course of a traversal. For example, an inorder traversal might actually take the following form:

inorder: traverse T_1 visit₁ R traverse T_2 visit₂ R ... visit _{$j-1$} R traverse T_j

where each visit of the node R , visit₁ R , visit₂ R , ..., visit _{$j-1$} R results in a (possibly) different operation performed on the information fields in R . We call this type of traversal a multivisit inorder traversal. Multivisit inorder traversals can be used to visit the keys in a multiway search trees as defined in Section 4.4.4.

4.5 Binary Search Trees

Given a totally ordered set S (elements of S will be called keys or identifiers), the *dictionary problem* is the well-known problem of designing an ADT to maintain a collection of items drawn from S . The classical dictionary problem restricts attention to the dictionary operations of inserting a key, deleting a key, and searching for a key, but we also might consider performing additional operations, such as finding the maximum or minimum elements or accessing the keys in sorted order.

In this section we introduce the binary search tree ADT and show how it supports the dictionary operations. In what follows we assume that the entire binary search tree is kept in internal memory, which is a reasonable assumption if the number of keys to be maintained is not too large. For larger sets of keys (such as typically encountered in databases stored in external memory) other ADTs such as B-trees are more appropriate (see Chapter 20). However, internal memory is rapidly becoming larger and cheaper, so that binary trees held in internal memory and containing keys from database records held in external memory are becoming more realistic.

Binary search trees are explicit generalizations of the implicit tree underlying the algorithm *BinarySearch*. The algorithm for inserting an element in a binary search tree is a variation of the searching strategy. Deletion is slightly more complicated, requiring in some cases a replacement of the deleted node by its inorder successor. We could also use the inorder predecessor for deletion, but for definiteness we always use the inorder successor. A useful property of a binary search tree is that an inorder traversal visits the nodes in sorted order.

4.5.1 Definition and Examples of Binary Search Trees

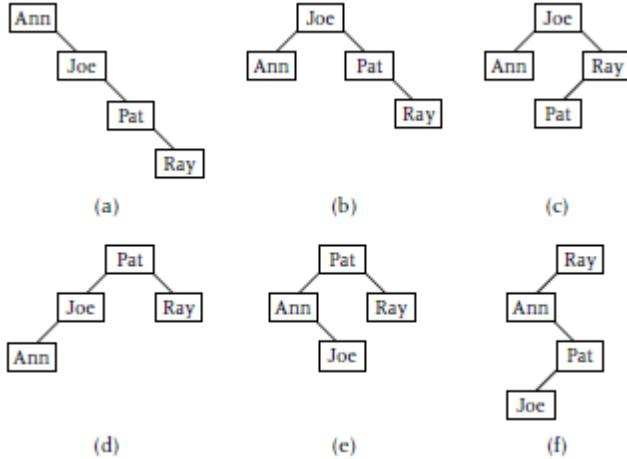
A binary search tree T with respect to keys in the nodes of T can be defined recursively as follows.

Definition 4.5.1

A binary tree T is called a binary search tree (with respect to a key field in each node) if T is empty, or if T has root R , and its right and left subtrees RT , LT satisfy the following.

1. Each key in LT is not larger than the key in R .
2. Each key in RT is not smaller than the key in R .
3. Both LT and RT are binary search trees.

Figure 4.14 gives 6 of the 24 different binary search trees for the set of four keys: ‘Ann’, ‘Joe’, ‘Pat’, ‘Ray’.



Sample binary search trees for the keys Ann, Joe, Pat and Ray

Figure 4.14

Given any set of n distinct keys (identifiers) and an arbitrary binary tree T on n nodes, there is a unique assignment of the keys to the nodes in T such that T becomes a binary search tree for the keys (see Exercise 4.29). Hence, the number of binary search trees

containing a given set of n distinct keys equals the number $b_n = \frac{1}{n+1} \binom{2n}{n}$ of binary trees on n nodes.

4.5.2 Searching a Binary Search Tree

A simple and efficient algorithm (*BinSrchTreeSearch*) exists for locating information stored in binary search trees. For example, if we wish to determine which node in the tree (if any) has its key equal to a search element X , then we first compare X to the key contained in the root. If X equals this key, we are done. Otherwise, if X is less than the key in the root, then we search the left subtree, else we search the right subtree. The following algorithm *BinSrchTreeSearch* based on this simple process returns a pointer *Location* to a node where a search element X is found. If X is not found in the tree, then *BinSrchTreeSearch* sets *Location* to a node in the tree where X can be inserted as a child and still maintain a search tree. In the pseudocode for *BinSrchTreeSearch*, we assume that the nodes in the tree have the structure as described in Section 4.3.2.

```

procedure BinSrchTreeSearch(Root,S,Location,Found)
Input: Root ( $\rightarrow$ BinaryTreeNode)//points at root of a binary search tree
        X (KeyType)
Output: Location ( $\rightarrow$ BinaryTreeNode) //points at occurrence of X, if any
          Found(Boolean) //.false. if X not in tree, .true. otherwise
Found  $\leftarrow$  .false.
Location  $\leftarrow$  null
Current  $\leftarrow$  Root

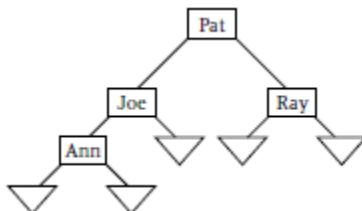
```

```

while Current ≠ null .and. .not. Found do
    Location ← Current
    if X = Current→Key then
        Found ← .true.
    else
        if X < Current→Key then
            Current ← Current→LeftChild
        else
            Current ← Current→RightChild
        endif
    endif
endwhile
end BinSrchTreeSearch

```

In Figure 4.15 we have not drawn the (implicit) leaf nodes corresponding to unsuccessful searches. The tree in Figure 4.14d is redrawn in Figure 4.15 with leaf nodes corresponding to unsuccessful searches added. For example, a search for ‘Mary’ would end up at the right child of ‘Joe’. The latter node is where the key ‘Mary’ can be inserted and still maintain the binary search tree property.



Search tree with leaf nodes drawn representing unsuccessful searches

Figure 4.15

Inserting a node in a binary tree is similar to searching. In the exercises we develop algorithms for inserting and deleting from a binary search tree.

4.5.5 Multiway Search Trees

Multiway search trees generalize binary search trees by allowing more than one key to be stored in a given node. Thus, multiway search trees allow multiway branching to occur at a given node, instead of the two-way branching allowed by binary trees. The keys in a given node of a multiway search tree are maintained as an ordered list. During a search of a multiway search tree, when a search element reaches a given node, a search of the keys in the node is performed. (The latter search is usually a linear search, but if the keys are maintained in an array, then binary search might be used if the node contains a sufficiently large number of keys.) Either the search element equals one of the keys, or a branch is made to an appropriate child's subtree (which contains the key if it is in the tree at all), where the search continues. The following recursive definition places the fairly obvious restriction on the keys in a multiway search tree that must hold to support the

natural generalization of the search strategy for binary search trees. In the definition, the value of j varies with the particular node in T .

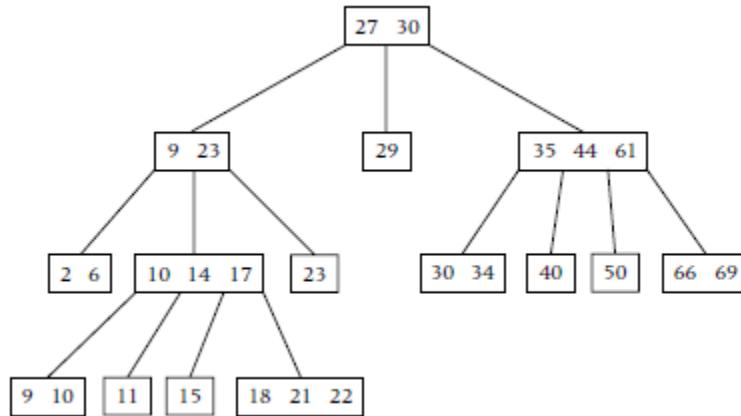
Definition 4.5.2

A multiway search tree T is either empty or consists of a root R containing j keys, $key_1 \leq key_2 \leq \dots \leq key_j$, $j \geq 1$, and a collection of subtrees T_1, T_2, \dots, T_{j+1} having the following two properties:

1. for any $k_i \in T_i$, $1 \leq i \leq j + 1$, we have

$$k_1 \leq key_1 \leq k_2 \leq key_2 \leq \dots \leq k_j \leq key_j \leq k_{j+1},$$
2. every T_i is a multiway search tree.

For a given fixed constant m , multiway search trees for which the maximum number of keys in a given node is bounded above by $m - 1$ are called m -way search trees. For example, a four-way search tree on 15 nodes is shown in Figure 4.16. Note that a binary search tree is a two-way search tree.



A four-way search tree on 15 nodes

Figure 4.16

Remarks

A commonly used multiway search tree is the B-tree. B-trees are subject to several additional restrictions, including bounds on the minimum and maximum number of keys allowed in each node. B-trees also satisfy a more restrictive condition than (2), namely:

- 2.' Either *every* T_i is empty, or *every* T_i is a nonempty multiway search tree.

We adopt the more general condition (2) in our definition of multiway search trees so that multiway search trees generalize the notion of binary search trees.

Figure 4.17 contains a high-level procedure determining a location in the multiway search tree T where the key k is found. In our high-level description, we do not specify how the tree T is implemented. The location of k consists of a pair (v, i) , where v is a node of T in which k occurs, and i is a position of the occurrence of k in the ordered list of keys

of v . If k does not occur in the tree, we set $i = 0$ and regard v as undefined. We initially perform step 1 in Figure 4.17 with a variable Root pointing at the root of T . Before repeating step 1, Root is redefined to point to the appropriate subtree being searched.

1. If k occurs in position i of the node v pointed to by Root , then return (v, i) .
2. If we are at a leaf, then return $i = 0$.
3. Redefine Root to point at the root of subtree T_i and repeat step 1, where T_i is determined by one of the following three conditions:
 - a. $i = 1$ and $k < \text{key}_1$,
 - b. $1 < i < j$ and $\text{key}_{i-1} < k < \text{key}_1$,
 - c. $i = j$ and $k > \text{key}_j$.

High-level description of procedure for search a multiway search tree

Figure 4.17

4.5.6 Treesort

The algorithm tree sort is based on the following interesting key fact.

Key Fact An inorder traversal of a binary search tree visits the nodes in nondecreasing order of the keys.

You should convince yourself of the validity of this key fact (a formal proof uses mathematical induction, see Exercise 4.35). Thus, given a list $L[0:n - 1]$, we can sort the list by creating a binary search tree T whose keys are the elements of L , and then performing an inorder traversal of T . The algorithm *CreateBinSrchTree* for creating a binary tree simply performs n successive insertions of $L[0], L[1], \dots, L[n - 1]$, respectively. In the case where the list is already sorted, *CreateBinSrchTree* creates a completely skewed tree and has worst-case complexity given by:

$$W(n) = 1 + 2 + \dots + n - 1 = \frac{n(n-1)}{2} \in \Theta(n^2).$$

However, in spite of the quadratic worst-case complexity of *CreateBinSrchTree*, it turns out that its average complexity belongs to $\Theta(n \log n)$. Since an inorder traversal of a tree is linear, tree sort also has quadratic worst-case complexity, and $\Theta(n \log n)$ average complexity.

The following proposition extends to multiway search trees the useful property that an inorder traversal of a binary search tree visits the nodes in sorted order. In our multivisit inorder traversal, if node v contains q keys, then $\text{Visit}(v, i)$ processes the i th key in the sequential ordering of the keys in the node, $i = 1, \dots, q$.

Proposition 4.4.1

A multivisit inorder traversal of a multiway search tree visits the nodes in sorted order. \square

A formal proof of Proposition 4.4.1 is left to the exercises.

4.6 Priority Queues and Heaps

A *priority queue* is a collection of elements each having a priority value, and such that when an element is deleted from the collection, an element of highest priority is always chosen for deletion. (Note that a priority queue is *not* a queue; that is, is not a FIFO list, but this ambiguity has become standard terminology.) One way to maintain a priority queue would be as a list ordered by the priority values, so that the highest priority element would be at the end of the list. Then deletion of the element would consist of a single list operation, and have $\Theta(1)$ complexity. However, insertion of an element would have $\Theta(n)$ complexity in the worst case. Similarly, a priority queue could simply be maintained as an unordered list, in which case deletion would require a linear scan to find an element of highest priority, and deletion would now have $\Theta(n)$ complexity in the worst case. Insertion, however, would now have $\Theta(1)$ complexity. It would be nice to find a data structure that would have, say, logarithmic complexity for both operations of deletion and insertion. The data structure known as a heap does the job.

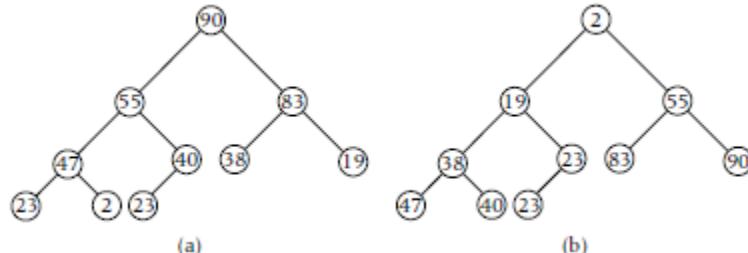
4.6.1 Heap Implementation of a Priority Queue

The underlying structure of a heap is a complete binary tree whose root contains the highest priority element.

Definition 4.6.1

A *max-heap (min-heap)* H is a complete binary tree whose information fields contain values (keys) having the following *max-heap (min-heap) property*: Given any node v in H , the value of v is not smaller (not larger) than the value of any node in the subtree having v as a root.

For convenience of notation, throughout the remainder of this section we discuss *max-heaps* and sometimes simply use the word *heap* for *max-heap*. A completely analogous discussion can be given for *min-heaps*. (See Figure 4.18.)



a) Max-heap; b) min-heap

Figure 4.18

Key Fact

Clearly, the max-heap property can be equivalently expressed by requiring that the value of each node v is not smaller than the value of its left child or right child (if they exist). This allows us to localize the heap property to each node. Thus, we can talk about the heap property holding or not holding at a given node.

A very convenient implementation of a heap results from storing the n key values in an array $A[0:m - 1]$, where $m \geq n$. The convenience of the array implementation results from the ease in which we can reference the children of a node, as well as its parent. Indeed, given a node of index i , its left child has index $2i + 1$ and its right child has index $2i + 2$ (assuming that these children exist). Also, if $i > 0$, the parent of a node of index i is given by $\lfloor (i - 1) / 2 \rfloor$.

Figure 4.19 shows the max-heap given in Figure 4.18 stored in an array $A[0:m - 1]$, $m \geq 10$.

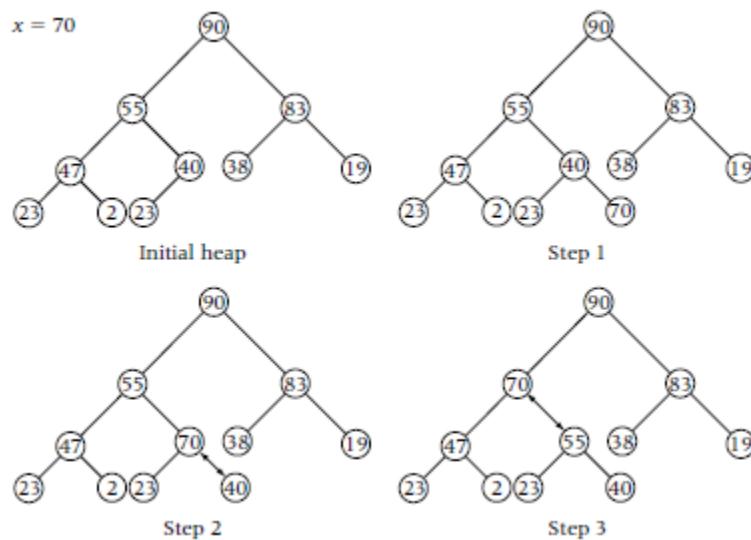
A	Index	0	1	2	3	4	5	6	7	8	9
	Value	90	55	83	47	40	38	19	23	2	23

Max-heap stored in array $A[0:m - 1]$, $m \geq 10$.

Figure 4.19

4.6.2 Inserting into a Heap

We now describe an algorithm for inserting a new element into an existing heap. We suppose that the existing heap occupies positions $A[0], \dots, A[\text{HeapSize} - 1]$ within the array $A[0:m - 1]$. The idea is to initially place the new element in position $A[\text{HeapSize}]$ and let it “rise” along the unique path from $A[\text{HeapSize}]$ to the root until it finds a proper position. Letting $i = \text{Heapsize}$, we begin by comparing x to the value of the parent node $A[\lfloor (i - 1)/2 \rfloor]$. If x has a larger value than its parent, we move the parent down to position i and move x up to the (old) position of its parent. The argument is then repeated until x has been moved up the path to a proper position. We illustrate this movement in Figure 4.20, where we insert the value $x = 70$ into the max-heap given in Figure 4.18.



Inserting the value $x = 70$ into the existing heap $A[0:9]$

Figure 4.20

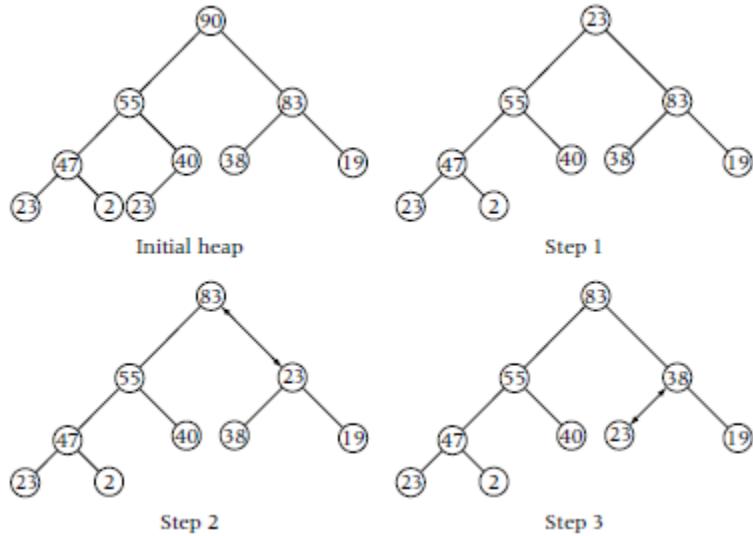
The pseudocode for the insertion operation follows. Note that we do not need to make an assignment of x until a proper place for its insertion into the heap is found.

```
procedure InsertMaxHeap( $A[0:m - 1], HeapSize, x$ )
Input:  $A[0:m - 1]$  (an array containing a heap in positions  $0, \dots, HeapSize - 1$ )
     $HeapSize$  (the number of elements in the heap)
     $x$  (a value to be inserted into heap)
Output:  $A[0:m - 1]$  (array altered by addition of  $x$  and heap maintained)
     $HeapSize$  (input  $HeapSize + 1$ )
 $i \leftarrow HeapSize$ 
 $j \leftarrow \lfloor (i - 1)/2 \rfloor$ 
//traverse path to root until proper position for inserting  $x$  is found
while  $j \geq 0$  .and.  $A[j] < x$  do
     $A[i] \leftarrow A[j]$  //move parent down one position in path
     $i \leftarrow j$  //move  $x$  up one position in path
     $j \leftarrow \lfloor (j - 1)/2 \rfloor$ 
endwhile
// $i$  is now a proper position for  $x$ 
 $A[i] \leftarrow x$ 
 $HeapSize \leftarrow HeapSize + 1$ 
end InsertMaxHeap
```

Using comparisons to an element in the heap as our basic operation, the worst-case complexity for *InsertMaxHeap* occurs when the inserted element must rise all the way to the root. Thus, the worst-case complexity of *InsertMaxHeap* belongs to $\Theta(\log n)$.

4.6.3 Deleting from a Heap

The other basic operation for a priority queue that we need to implement is to remove the highest priority element. For our heap implementation, this element is at the root. The removal of the root is accomplished by elevating the bottom heap element $x = A[HeapSize - 1]$ to the root and then letting x “fall” down an appropriate path in the complete binary tree $A[0:HeapSize - 2]$ until it finds a proper position. Initially we compare x to the larger of the two children $A[1]$ and $A[2]$. If x is smaller, the larger child is elevated to the root (made “king of the heap”) and x moves down to the position formerly occupied by the elevated child. The argument is then repeated until a proper position for x is found. We illustrate this in Figure 4.21 by removing the root from the max-heap given in Figure 4.20.



Removing the root from a heap

Figure 4.21

The adjustment operation illustrated in steps 1 through 3 in Figure 4.21 can be applied to the subtree rooted at any node having index i in a complete binary tree $A[0:n - 1]$, where the subtrees of $A[0:n - 1]$ with roots at the two children $2i + 1$ and $2i + 2$ are both heaps. The algorithm *AdjustMaxHeap* shows how to let $A[i]$ follow an appropriate path in the subtree with i as a root so as to make a heap out of this subtree.

```

procedure AdjustMaxHeap( $A[0:m - 1], n, i$ )
Input:  $A[0:m - 1]$  (an array)
     $n$  (consider  $A[0:n - 1]$  as complete binary tree,  $n \leq m$ )
     $i$  (index where subtrees of  $A[0:n - 1]$  rooted at  $2i + 1$  and  $2i + 2$  are heaps)
Output:  $A[0:m - 1]$  (subtree of  $A[0:n - 1]$  rooted at  $A[i]$ , adjusted so that it becomes a
            heap)

 $Temp \leftarrow A[i]$ 
//traverse down a path until a proper position for  $Temp = A[i]$  is found
 $Found \leftarrow .false.$  // $Found$  signals when a proper position is found
 $j \leftarrow 2*i + 1$  // $j$  is the path finder. At completion of loop,
// $\lfloor(j - 1)/2\rfloor$  is a proper position for  $Temp = A[i]$ 
while  $j \leq n$  .and. .not.  $Found$  do
    if  $j < n - 1$ .and.  $A[j] < A[j + 1]$  then //then move to right child  $j + 1$ 
         $j \leftarrow j + 1$  //path finder updated to right child
    endif
    if  $Temp \geq A[j]$  then
         $Found \leftarrow .true.$ 
    else
         $A[\lfloor(j - 1)/2\rfloor] \leftarrow A[j]$  //move larger child up one position in path
         $j \leftarrow 2*j + 1$  //move path finder to next possible position for  $Temp$ 
    
```

```

endif
endwhile
 $A[\lfloor(j - 1)/2\rfloor] \leftarrow Temp$ 
end AdjustMaxHeap

```

Clearly, the worst-case complexity of *AdjustMaxHeap* occurs when the element $A[i]$ must fall all the way down to the bottom of the heap. Thus, when calling *AdjustMaxHeap* with the parameter i , in the worst case we make $\log_2 n - \log_2 i$ comparisons.

The operation of removing the root from a heap is accomplished by the following pseudocode.

```

procedure RemoveMaxHeap( $A[0:m - 1], HeapSize, x$ )
Input:  $A[0:m - 1]$  (an array containing a heap in positions  $0, \dots, HeapSize - 1$ )
     $HeapSize$  (the number of elements in the heap)
Output:  $A[0:m - 1]$  (array altered by removal of the root  $A[0]$  and heap maintained)
     $x$  (assigned the value of the (original) root  $A[0]$ )
     $HeapSize$  (input  $HeapSize - 1$ )
 $x \leftarrow A[0]$ 
 $HeapSize \leftarrow HeapSize - 1$ 
 $A[0] \leftarrow A[HeapSize]$ 
 $AdjustMaxHeap(A[0:m - 1], HeapSize, 0)$ 
end RemoveMaxHeap

```

The worst-case complexity of *RemoveMaxHeap* belongs to $\Theta(\log n)$, so that we have implemented both the insertion of an element and the removal of the highest priority element with worst-case complexity $\Theta(\log n)$.

4.6.4 Creating a Heap

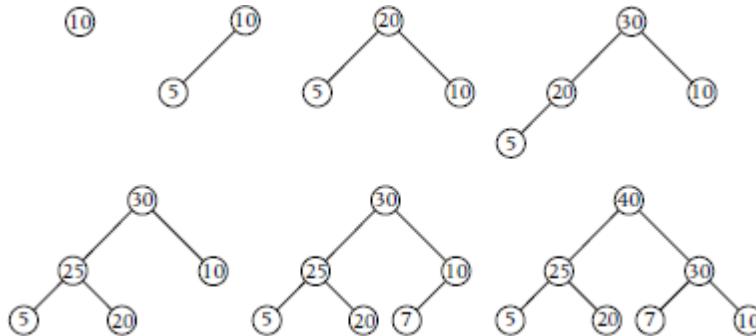
Given n values in an array $A[0:m - 1]$, the question arises as to how to make $A[0:n - 1]$ into a heap. The most obvious way is to simply sequentially invoke *InsertMaxHeap*($A[0:m - 1], HeapSize, x$) for $x = A[0], \dots, A[n - 1]$. In fact, this method is well suited to applications where the heap is maintained dynamically, with new elements being added to an already existing heap.

```

procedure MakeMaxHeap1( $A[0:m - 1], n$ )
Input:  $A[0:m - 1]$  (an array)
     $n$  (values in  $A[0:n - 1]$  considered as complete binary tree,  $n \leq m$ )
Output:  $A[0:m - 1]$  ( $A[0:n - 1]$  made into a heap)
 $HeapSize \leftarrow 0$ 
for  $j \leftarrow 0$  to  $n - 1$ 
     $InsertMaxHeap(A[0:m - 1], HeapSize, A[j])$ 
endfor
end MakeMaxHeap1

```

Figure 4.22 shows the action of *MakeMaxHeap1* on the set of elements (10, 5, 20, 30, 25, 7, 40). We show the result after each iteration of the **for** loop.



Action of *MakeMaxHeap1* on the set of elements (10, 5, 20, 30, 25, 7, 40)

Figure 4.22

Since the worst-case complexity of *InsertMaxHeap* belongs to $O(\log n)$, the worst-case complexity $W(n)$ of *MakeMaxHeap1* belongs to $O(n \log n)$. The worst case occurs when the elements are in increasing order, so that each element must rise to the root when it is inserted. We now show that $W(n)$ belongs to $\Omega(n \log n)$. Let $k = \lfloor \log_2 n \rfloor$ and recall that a complete binary tree on n elements has 2^i nodes at level i for $i = 0, \dots, k - 1$. Since we make i comparisons for each node at level i , and applying identity (1.2.11), we have

$$W(n) > \sum_{i=0}^{k-1} i 2^i = 2^k (k - 2) + 2 \in \Omega(n \log n).$$

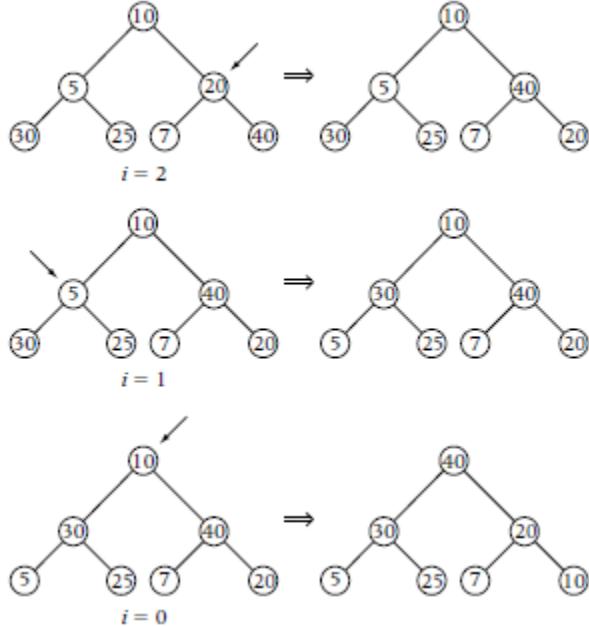
Hence, $W(n)$ belongs to $\Omega(n \log n) \cap O(n \log n) = \Theta(n \log n)$.

An alternative algorithm *MakeMaxHeap2* for creating a heap has worst-case complexity in $O(n)$. *MakeMaxHeap2* works by repeated calls to *AdjustMaxHeap*($A[0:m-1], n, i$). Recall that *AdjustMaxHeap*($A[0:m-1], n, i$) requires that the subtrees in $A[0:n-1]$ with roots $2i+1$ and $2i+2$ are already heaps. This will certainly be true if these latter two roots are leaf nodes of $A[0:n-1]$. Thus, we can make a heap by initially calling *AdjustMaxHeap*($A[0:m-1], n, i$) with $i = \lfloor (n-1)/2 \rfloor$ and then sequentially calling *AdjustMaxHeap*($A[0:m-1], n, i$) as i is decremented by one each time, until we finally reach the root.

```

procedure MakeMaxHeap2( $A[0:m-1], n$ )
Input:  $A[0:m-1]$  (an array)
         $n$  (values in  $A[0:n-1]$  considered as complete binary tree,  $n \leq m$ )
Output:  $A[0:m-1]$  ( $A[0:n-1]$  made into a heap)
for  $i \leftarrow \lfloor (n-1)/2 \rfloor$  down to 0 do
    AdjustMaxHeap( $A[0:m-1], n, i$ )
endfor
end MakeMaxHeap2
  
```

The action of *MakeMaxHeap2* is illustrated in Figure 4.23 on the set $(10, 5, 20, 30, 25, 7, 40)$, which was the same set illustrated in Figure 4.22 for *MakeMaxHeap1*. Note that the heaps created are rather different, but both satisfy the heap property. In Figure 4.29 we show the array before and after each call to *AdjustMaxHeap*. We indicate in each “before” picture the particular subtree that is being made into a heap.



Making a max-heap using *MakeMaxHeap2*

Figure 4.23

During the execution of the call to $\text{AdjustMaxHeap}(A[0:m - 1], n, i)$, an element at level j will fall at most $k - j$ levels, where $k = \lfloor \log_2 n \rfloor$. Again, for $j < k$, there are exactly 2^j nodes at level j . We begin calling $\text{AdjustMaxHeap}(A[0:m - 1], n, i)$ with $i = \lfloor (n - 1)/2 \rfloor$, which is (the index of) a node at level $k - 1$. Hence, the worst-case complexity of *MakeMaxHeap2* satisfies

$$W(n) \leq \sum_{i=0}^{k-1} 2^i (k-i) = \sum_{i=1}^k i 2^{k-i} = 2^k \sum_{i=1}^k \frac{i}{2^i}.$$

Applying identity (1.2.11) with $x = 1/2$, we obtain

$$W(n) \leq 2^k \left\lfloor 2 - \frac{k+2}{2^k} \right\rfloor = 2^{k+1} - (k+2) \leq 2n \in O(n).$$

Since *MakeMaxHeap2* does $\lfloor n/2 \rfloor$ comparisons in the best case (when we have a heap at the outset), we have shown that the best-case, worst-case, and average complexities of *MakeMaxHeap2* all belong to $\Theta(n)$.

4.6.5 Sorting by Selection: Selection Sort versus Heap Sort

Certainly one of the simplest strategies for sorting a list $L[0:n - 1]$ of size n in nondecreasing order is as follows. Select the largest element in L and interchange it with $L[n - 1]$. Then select the largest element in the sublist $L[0:n - 2]$ and interchange it with $L[n - 2]$. In general, in the i^{th} step, select the largest element in the sublist $L[0:n - i]$ and interchange it with $L[n - i]$, $i = 1, \dots, n - 1$. Clearly, this strategy yields a sorted list. The most straightforward strategy for selecting the largest elements is to make a linear scan as in the procedure *Max*. The resulting algorithm is known as *selectionsort*. Clearly, *selectionsort* performs $(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2$ comparisons for any input list of size n , so that its worst-case, best-case, and average complexities are all in $\Theta(n^2)$.

The inefficiency of *selectionsort* results from the procedure used to select the largest elements. Our selection procedure amounted to regarding the list as a priority queue, where the priority is the value (key) of the list element. The algorithm *heapsort* emulates *selectionsort* except that the priority queue of list elements is maintained as a heap, thereby reducing the complexity from $\Theta(n^2)$ to $\Theta(n \log n)$.

Procedure *HeapSort* utilizes the algorithm *AdjustMaxHeap* as follows. We first invoke *MakeMaxHeap2* to create the heap $A[0:n - 1]$. Then we interchange the root with the element at the bottom of the heap (position n) and invoke *AdjustMaxHeap*($A[0:m - 1], n - 1, 1$). The new root is now interchanged with the element at position $n - 1$, and the process is continued. The code for *HeapSort* follows.

```
procedure HeapSort( $A[0:n - 1]$ )
Input:  $A[0:n - 1]$  (a list of size  $n$ )
Output:  $A[0:n - 1]$  sorted in nondecreasing order
    MakeMaxHeap2( $A[0:n - 1], n$ )
    for  $i \leftarrow 1$  to  $n - 1$  do
        //interchange  $A[0]$  with  $A[n - i]$ 
        interchange ( $A[0], A[n - i]$ )
        AdjustMaxHeap( $A[0:n - 1], n - i + 1, 0$ )
    endfor
end HeapSort
```

The call to *MakeMaxHeap2* has best-case and worst-case complexities $\Theta(n)$, whereas the $n - 1$ calls to *AdjustMaxHeap* each have $\Theta(1)$ best-case and $\Theta(\log n)$ worst-case complexities. Thus, *HeapSort* has $\Theta(n)$ best-case complexity and $\Theta(n \log n)$ worst-case complexity. *HeapSort* is a comparison-based sorting algorithm, so by lower bound theory (see Chapter 25), it has $\Omega(n \log n)$ average complexity, which implies that *HeapSort* has $\Theta(n \log n)$ average complexity.

4.7 Implementing Disjoint Sets

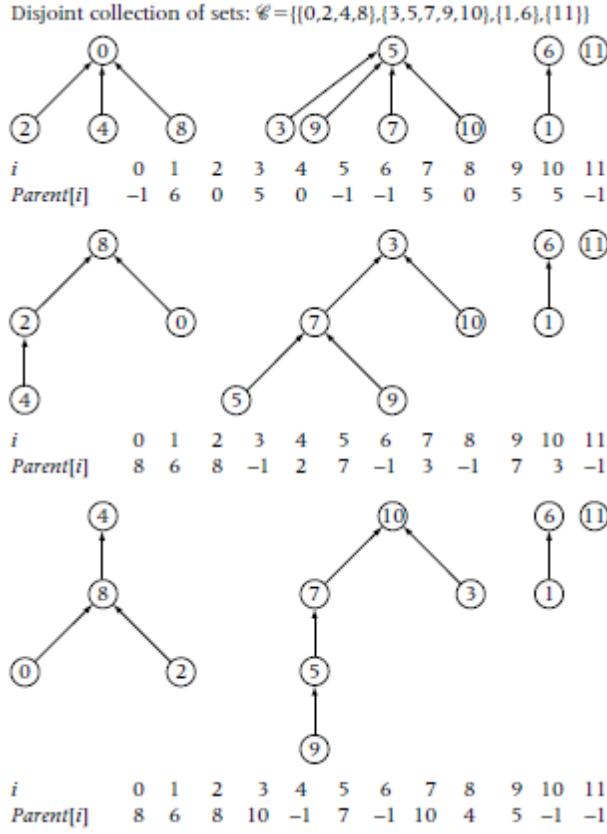
Many algorithms, particularly graph algorithms, involve maintaining a (dynamically changing) equivalence relation on some universal (or base) set $S = \{s_1, s_2, \dots, s_n\}$. For example, an efficient implementation of Kruskal's algorithm for finding a minimum cost spanning tree in a weighted graph involves successively taking unions of vertex sets in different equivalence classes determined by trees in a forest. Since an equivalence relation on S determines a partition of S into disjoint subsets, the problem of maintaining

equivalence relations reduces to the problem of maintaining disjoint sets. The two main operations associated with the disjoint set ADT are efficiently finding the set containing a given element and forming the union of two sets.

One method of maintaining disjoint sets would be to represent each subset $A \subseteq S$ by its characteristic vector (v_1, v_2, \dots, v_n) , where $v_i = 0$ if $v_i \notin A$, and $v_i = 1$ if $v_i \in A$, $i = 1, 2, \dots, n$. Given any collection C of k disjoint subsets A_1, A_2, \dots, A_k of the base set S , finding the set containing a given element s_i would amount to checking which of the k characteristic vectors contains a 1 in position i . For example, if we maintain a linked list of pointers to the k characteristic vectors, then this check has $\Omega(k)$ worst-case complexity, and requires $\Omega(kn)$ storage locations. Taking the union of two sets would amount to making a linear scan of the two characteristic vectors representing the two sets, resulting in linear complexity (again, assuming the above implementation of maintaining the characteristic vectors). Using trees, we now show how to achieve logarithmic behavior for both union and find operations, and also require only a one-dimensional integer array of length n to represent the disjoint sets.

4.7.1 Union and Find Algorithms

For disjoint sets, an efficient implementation of the union and find operations is based on the parent array representation of a forest. Consider any collection C of k disjoint subsets A_1, A_2, \dots, A_k of the base set S , where for convenience we assume that $S = \{0, 1, \dots, n - 1\}$. A rooted forest F represents the collection C of disjoint sets if it consists of k rooted trees T_1, T_2, \dots, T_k such that the vertex set of T_i is A_i , $i = 1, 2, \dots, k$. Clearly, the k sets A_1, A_2, \dots, A_k are (uniquely) determined by the single array $Parent[0:n - 1]$ of the parent-array implementation of F . Further, once the array $Parent[0:n - 1]$ is given, the set A_i can be identified by the single element r_i , where r_i is the root of T_i , $i = 1, 2, \dots, k$ (see Figure 4.24).



Three sample forest representation of the collection C

Figure 4.24

Given an element $x \in A_1 \cup A_2 \cup \dots \cup A_k$, the following procedure *Find1* returns the root r of the tree in forest F containing x .

```

procedure Find1( $\text{Parent}[0:n - 1], x, r$ )
Input:  $\text{Parent}[0:n - 1]$  (array representing disjoint subsets of  $S$ )
       $x$  (an element of  $S$ )
Output:  $r$  (the root of the tree corresponding to the subset containing  $x$ )
     $r \leftarrow x$ 
    while  $\text{Parent}[r] \geq 0$  do
       $r \leftarrow \text{Parent}[r]$ 
    endwhile
end Find1

```

The complexity of *Find1* is measured by the number of iterations of the **while** loop. Clearly, the complexity of *Find1* depends on the depths of the trees T_i in F . Thus, it is desirable to keep the depths of the trees in the forest relatively small.

Given $C = \{A_1, A_2, \dots, A_k\}$, let C_{ij} denote the collection of $k - 1$ sets obtained from C by removing sets A_i and A_j and adding the set $A_i \cup A_j$. Consider the two forests F_{ij} and F_{ji} ,

where F_{ij} is obtained from F by setting $\text{Parent}(r_j) = r_i$, and F_{ji} is obtained from F by setting $\text{Parent}(r_i) = r_j$. Both forests represent the collection of sets C_{ij} . To help minimize the depth of the tree representing the union of A_i and A_j , we choose the forest F_{ij} if T_i has more vertices than T_j , otherwise we choose the forest F_{ji} . This choice can be made instantly (in constant time) by dynamically keeping track of the number n_i of vertices in each tree T_i . An efficient way of keeping track of n_i without allocating additional memory locations is to (dynamically) store the negative of n_i in $\text{Parent}[r_i]$. For example the revised parent array for the first forest in Figure 4.24 becomes.

The root vertices are still distinguishable from the other vertices, since they index those locations of the array $\text{Parent}[0:n - 1]$ having negative values.

The following pseudocode for the algorithm *Union* computes the parent array implementation of the union of two sets based on the preceding discussion.

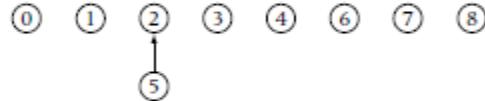
```
procedure Union( $\text{Parent}[0:n - 1], r, s$ )
Input:  $\text{Parent}[0:n - 1]$  (an array representing disjoint sets)
         $r, s$  (roots of the trees representing two disjoint sets  $A, B$ )
Output:  $\text{Parent}[0:n - 1]$  (an array representing disjoint sets after forming
         $A \cup B$ )
    sum  $\leftarrow \text{Parent}[r] + \text{Parent}[s]$ 
    if      $\text{Parent}[r] > \text{Parent}[s]$  then   //tree rooted at  $s$  has more vertices
         $\text{Parent}[r] \leftarrow s$                                   //than tree rooted at  $r$ 
         $\text{Parent}[s] \leftarrow \text{sum}$ 
    else
         $\text{Parent}[s] \leftarrow r$ 
         $\text{Parent}[r] \leftarrow \text{sum}$ 
    endif
end Union
```

When implementing disjoint sets in a particular algorithm, the initial collection C of disjoint sets is usually the collection of all singleton subsets of the base set S . Figure 4.25 illustrates a sample sequence of calls to the procedure *Union*, starting with the collection of singleton subsets of the base set $S = \{0, 1, \dots, 8\}$.

Initial collection: $\{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}\}$

i	0	1	2	3	4	5	6	7	8
$Parent[i]$	-1	-1	-1	-1	-1	-1	-1	-1	-1

Call $Union[Parent[0:8], 2, 5] \Rightarrow \{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{6\}, \{7\}, \{8\}\}$



i	0	1	2	3	4	5	6	7	8
$Parent[i]$	-1	-1	-2	-1	-1	2	-1	-1	-1

Call $Union[Parent[0:8], 2, 7] \Rightarrow \{\{0\}, \{1\}, \{2, 5, 7\}, \{3\}, \{4\}, \{6\}, \{8\}\}$



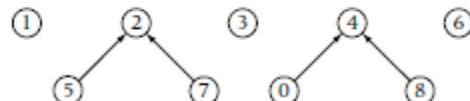
i	0	1	2	3	4	5	6	7	8
$Parent[i]$	-1	-1	-3	-1	-1	2	-1	2	-1

Call $Union[Parent[0:8], 4, 8] \Rightarrow \{\{0\}, \{1\}, \{2, 5, 7\}, \{3\}, \{4, 8\}, \{6\}\}$



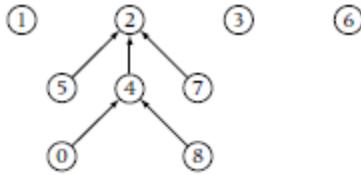
i	0	1	2	3	4	5	6	7	8
$Parent[i]$	-1	-1	-3	-1	-2	2	-1	2	4

Call $Union[Parent[0:8], 0, 4] \Rightarrow \{\{0, 4, 8\}, \{1\}, \{2, 5, 7\}, \{3\}, \{6\}\}$



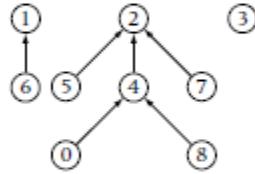
i	0	1	2	3	4	5	6	7	8
$Parent[i]$	4	-1	-3	-1	-3	2	-1	2	4

Call `Union[Parent[0:8],2,4]` $\Rightarrow \{\{0,2,4,5,7,8\},\{1\},\{3\},\{6\}\}$



<i>i</i>	0	1	2	3	4	5	6	7	8
<i>Parent[i]</i>	4	-1	-6	-1	2	2	-1	2	4

Call `Union[Parent[0:8],1,6]` $\Rightarrow \{\{0,2,4,5,7,8\},\{1,6\},\{3\}\}$



<i>i</i>	0	1	2	3	4	5	6	7	8
<i>Parent[i]</i>	4	-2	-6	-1	2	2	1	2	4

Call `Union[Parent[0:8],1,2]` $\Rightarrow \{\{0,1,2,4,5,6,7,8\},\{3\}\}$



<i>i</i>	0	1	2	3	4	5	6	7	8
<i>Parent[i]</i>	4	2	-8	-1	2	2	1	2	4

...

Representative forests and parent arrays for a sample sequence of calls to procedure *Union*

Figure 4.25

Proposition 4.7.1

Let F be any forest resulting from some sequence of calls to *Union*, where the initial input forest consists of isolated vertices. Then the depth of any tree in F having j vertices is at most $\lfloor \log_2 j \rfloor$.

We leave the proof of Proposition 4.7.1 as an exercise. It follows from Proposition 4.7.1 that if we start with a collection C consisting of singleton sets, and perform a sequence of calls to *Union*, then the computing time of *Find1* for a given input x is $O(\log j)$, where j is the cardinality of the set containing x . Thus, the worst-case complexity in making an intermixed sequence of calls to *Union* and *Find1*, n calls to *Union*, and $m \geq n$ calls to *Find1*, is $O(m \log n)$.

4.7.2 Improved *Union* and *Find* Using the Collapsing Rule

Even though the procedures *Union* and *Find1* are very efficient and might seem the best possible, there is still room for improvement. The improvement comes by modifying

Find1(*Parent*[0:*n* – 1],*x,r*) as follows. After *r* is obtained, we can traverse the path from *x* to *r* again and set *Parent*[*v*] = *r* for each vertex *v* encountered. This “collapsing” of the path basically doubles the computing time of *Find1*. However, it tends to reduce the depth of the trees in the forest so that an overall improvement is achieved in the efficiency of making an intermixed sequence of union and find operations. The following procedure *Find2* implements the collapsing rule.

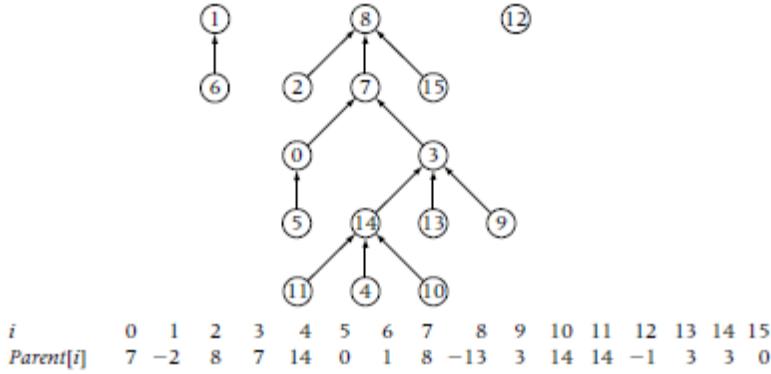
```

procedure Find2(Parent[0:n – 1],x,r)
Input: Parent[0:n – 1] (an array representing disjoint subsets of S)
          x (an element of S)
Output: r (the root of the tree corresponding to subset containing x)
r  $\leftarrow$  x
while Parent[r] > 0 do
    r  $\leftarrow$  Parent[r]
endwhile
y  $\leftarrow$  x
while y  $\neq$  r do
    Temp  $\leftarrow$  Parent[y]
    Parent[y]  $\leftarrow$  r
    y  $\leftarrow$  Temp
endwhile
end Find2
```

Figure 4.26 illustrates the collapsing rule for a call to *Find2* with *x* = 14, for a sample input forest representing the sets

$$\{ \{0,2,3,4,5,7,8,9,10,11,13,14,15\}, \{1,6\}, \{12\} \}.$$

Forest representing collection $\{\{0,2,3,4,5,7,8,9,10,11,13,14,15\}, \{1,6\}, \{12\}\}$
before call to *Find2*:



Forest representing collection $\{\{0,2,3,4,5,7,8,9,10,11,13,14,15\}, \{1,6\}, \{12\}\}$
after call to *Find2* with $x = 14$:

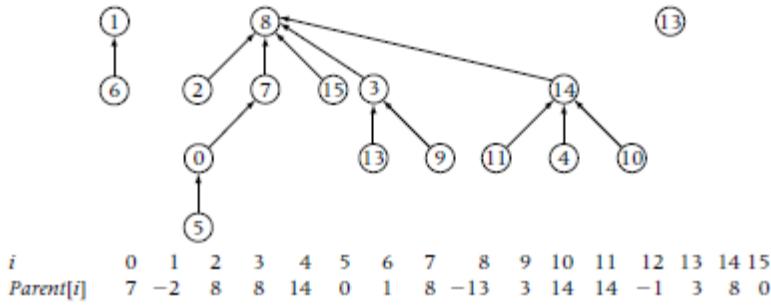


Illustration of collapsing rule for sample call to *Find2*

Figure 4.26

Tarjan has shown that the worst-case complexity in making an intermixed sequence of calls to *Union* and *Find2*, $n - 1$ calls to *Union* and $m \geq n$ calls to *Find2*, is $O(m \alpha(m,n))$, where $\alpha(m,n)$ is an extremely slow-growing function. In fact, it grows so slowly that, for all practical purposes, we can regard $\alpha(m,n)$ as a constant function of m and n . The function $\alpha(m,n)$ is related to a functional inverse of the extremely fast-growing Ackermann's function $A(m,n)$ given by the recurrence relation:

$$A(m, n) = A(m - 1, A(m, n - 1)), m \geq 1 \text{ and } n \geq 2,$$

$$A(0, n) = 2n, A(m, 0) = 0, m \geq 1, A(m, 1) = 2, m \geq 1.$$

4.8 Closing Remarks

Further applications of trees will be found throughout the text. For example, in Chapter 5 we discuss the depth-first and breadth-first search trees in graphs and digraphs, and in Chapter 6 we describe various algorithms for minimum spanning trees and shortest-path trees in weighted graphs and digraphs. In Chapter 6 we describe the tree associated with Huffman coding for data compression. In Chapter 8 we use dynamic programming to construct optimal binary search trees. In Chapter 9 we discuss the state space tree

associated with backtracking and branch-and-bound algorithms. In Chapter 12 we discuss the application of trees to routing tables in a network.

Exercises

Section 4.2 Mathematical Properties of Binary Trees

- 4.1 Show that if a binary tree T is full at level i , then it is full at every level j smaller than i .
- 4.2 Show that the depth of the complete binary tree T_n for a general n is given by

$$D(T_n) = \lfloor \log_2 n \rfloor$$
- 4.3 Using induction, prove Proposition 4.2.1.
- 4.4 Using induction, prove Proposition 4.2.2.
- 4.5 Using induction, give a direct proof of Proposition 4.2.3 without using the transformation to 2-trees.
- 4.6 Prove Proposition 4.2.4.
- 4.7 Prove Proposition 4.2.5.
- 4.8 Prove Proposition 4.2.6.
- 4.9 Prove Proposition 4.2.7.
- 4.10 Complete the verification of Theorem 4.2.8 by establishing the inequality

$$x\lfloor \log_2 x \rfloor + 2(x - 2^{\lfloor \log_2 x \rfloor}) \geq \lceil x \log_2 x \rceil \quad \text{for any integer } x \geq 1.$$
- 4.11 Given a 2-tree T having L leaf nodes, let T_{left} and T_{right} denote the left and right subtrees (of the root) of T , respectively.
 - a) Give a recurrence relation for $LPL(T)$ in terms of $LPL(T_{\text{left}})$, $LPL(T_{\text{right}})$, and L .
 - b) Solve the recurrence relation in (a) to give an alternate proof of Corollary 4.2.9.
- 4.12 Show that the implicit search tree for *BinarySearch* is a 2-tree that is full at the second-deepest level.
- 4.13 A k -ary tree T is a rooted tree where every node has at most k children. A k -tree T is a k -ary tree where every nonleaf node has exactly k children. In each of the following, state and prove a generalization of the result for binary trees or 2-trees to k -ary trees or k -trees.
 - a) Proposition 4.2.1
 - b) Proposition 4.2.2
 - c) Proposition 4.2.3
 - d) Proposition 4.2.4
 - e) Proposition 4.2.5
 - f) Proposition 4.2.6
 - g) Proposition 4.2.7
 - h) Theorem 4.2.8
- 4.14 Prove by induction that

$$\binom{2n}{n} \geq \frac{4^n}{2n+1}, \quad n \geq 1,$$
and conclude that the n th Catalan number $b_n = \frac{1}{n+1} \binom{2n}{n} \in \Omega(4^n / n^2)$.
- 4.15 Verify that the number b_n of different binary trees on n nodes satisfies the following recurrence relation:

$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1} = b_0 b_{n-1} + b_1 b_{n-2} + \cdots + b_{n-1} b_0, \text{init. cond. } b_0 = 1.$$

Section 4.3 Implementation of Trees and Forests

- 4.16 Write a program that inputs a tree using its parent array representation and converts the tree to its child-sibling representation.

Sections 4.4 Tree Traversals

- 4.17 Consider the child-sibling implementation of a general tree using pointer and dynamic variables.
- a) Give pseudocode for a recursive procedure *TreePreorder* that performs a preorder traversal of the tree whose root node is pointed to by the pointer variable *Root*.
 - b) In the case where T is a nonempty binary tree, show that the traversal of T generated by *TreePreorder* coincides with the traversal generated by the algorithm *Preorder* for binary trees.
 - c) Further, show that if T is any tree and T' is the binary tree obtained from T via the Knuth transformation, then the traversal of T generated by *TreePreorder* coincides with the traversal of T' generated by *Preorder*.
- 4.18 a) Show that a binary tree T can be reconstructed from its inorder and preorder traversal sequences.
- b) Show that part (a) is not true in general for its postorder and preorder traversal sequences.
- 4.19 Give a recursive procedure for swapping the left and right children of every node in a binary tree.
- 4.20 Associated with each arithmetic expression E involving binary or unary operations is a binary expression tree T defined recursively as follows. The leaf nodes of T correspond to the operands in E and the internal nodes correspond to the operators in E and are labeled accordingly. We may assume without loss of generality that E is fully bracketed. If $E = (E_1 \odot E_2)$ then the root node is labeled \odot , and the left and right subtrees of the root are expression trees for E_1 and E_2 , respectively. The reverse Polish (the postfix) form of expression E is defined recursively as follows. If F_1 and F_2 are reverse Polish expressions for E_1 and E_2 and \odot is binary operator, then the reverse Polish expression of $E_1 \odot E_2$ is $F_1 F_2 \odot$. If \odot is a unary operator, then the reverse Polish expression of $\odot E_1$ is $E_1 \odot$. For example, the reverse Polish expression for $(A + B)^*(-C)$ is $AB + C - ^*$.
 - a) Given a string representing a fully bracketed arithmetic expression E involving the binary operators $+$, $-$, $*$, $/$ and the unary operator of minus, write a program that creates an expression tree for E .
 - b) Show that a postorder traversal of the expression tree for E yields the reverse Polish expression for E .
- 4.21 Give pseudocode for a version of the inorder-successor function *InorderSucc* when there is a *Parent* field in each node.
- 4.22 Give pseudocode for a simple modification of the algorithm *Inorder3* that creates an inorder threading.
- 4.23 Give pseudocode for an inorder traversal of an inorder-threaded binary tree.

- 4.24 Give pseudocode for an inorder traversal of a binary tree that does not use an additional field for the parent pointer (or inorder threading), but instead simulates a parent pointer by dynamically reversing and resetting child pointers during the course of the traversal.
- 4.25 Show that a binary code allows for unambiguous (unique) decoding if, and only if, it is a binary prefix code.
- 4.26 Show that if T is any tree and T' is the binary tree obtained from T via the Knuth transformation, then the traversal of T generated by *TreeInorder* coincides with the traversal of T' generated by *Inorder*.
- 4.27 Give pseudocode for nonrecursive versions of
 - preorder traversal.
 - postorder traversal.

Section 4.5 Binary Search Trees

- 4.28 Give pseudocode for a recursive version of *BinSrchTreeInsert*.
- 4.29
 - Give pseudocode for a procedure *CreateBinSrchTree*, which creates a binary search tree by repeated calls to *BinSrchTreeInsert*.
 - Show the binary search tree created by *CreateBinSrchTree* for the keys 22,11,0,72,27,55,23,108,1, inserted in that order.
 - Give three orderings for insertion of the keys in part (b) for which the binary search tree created by *CreateBinSrchTree* is a path; that is, each node has at most one child.
 - Give two orderings for insertion of the keys in part (b) for which the binary search tree created by *CreateBinSrchTree* is complete.
- 4.30 Given any set of n distinct keys (identifiers) and an *arbitrary* binary tree T on n nodes, show that there is a unique assignment of the keys to the nodes in T such that T becomes a binary search tree for the keys.
- 4.31 Show that a binary search tree without the (implicit) external leaf nodes added is full at the second-deepest level if, and only if, the associated binary tree with the external leaf nodes added is full at the second-deepest level.
- 4.32 Give pseudocode for an altered *BinSrchTreeSearch* that determines the locations of both the search element and its parent.
- 4.33
 - Our pseudocode for inserting a key into a binary search tree assumed that the tree did not contain duplicate keys. Give pseudocode for an alternate version *BinSrchTreeSearch2* that allows duplicate keys to be inserted. When inserting a duplicate key X , assume that you always move to the right child of a node already containing X (the *move-to-the-right rule*).
 - Show that when creating a binary search tree by successively inserting a sequence of keys using the move-to-the-right rule, an inorder traversal of the resulting search tree will visit duplicate keys in the order in which they were inserted. Moreover, when using *BinSrchTreeSearch2* for a tree built with the move-to-the-right rule, show that the position in the tree of first key inserted amongst a given set of duplicate keys is found. Thus, all keys having a given value can be found by executing *BinSrchTreeSearch* to find the position of the first one that was inserted and then executing an inorder traversal of the right subtree of this position until a key having a different value is encountered.

- 4.34 Discuss an implementation of an m -way search tree using pointers and dynamic variables.
- 4.35 For the implementation of the m -way search tree given in the previous exercise, give pseudocode for the high-level searching procedure described in Figure 4.22.
- 4.36 Show by induction that an inorder traversal of a binary search tree visits the nodes in nondecreasing order of the keys.
- 4.37 Prove that *TreeSort* is a stable sorting algorithm.
- 4.38 Given a permutation π of a set of keys, the algorithm *BinSearchTreeCreate* creates a binary search tree T_π . Given T_π , show that π can be uniquely determined if, and only if, T_π is a path.
- 4.39 Show that the best-case and worst-case complexities of *BinSrchTreeCreate* for a list of size n belong to $\Theta(n \log n)$, $\Theta(n^2)$, respectively.
- 4.40 a) Show that an inorder traversal of a multiway search tree visits the keys in increasing order.
 b) Design a sorting algorithm based on a multiway search tree that generalizes *TreeSort*.
 c) Discuss how to guarantee that your sorting algorithm in (a) is stable.

Section 4.6 Priority Queues and Heaps

- 4.41 Demonstrate the action of *MakeMaxHeap1* on the set of elements
 55, 23, 65, 108, 2, 73, 41, 52, 34.
 (See Figure 4.22.)
- 4.42 Demonstrate the action of *MakeMaxHeap2* (see Figure 4.23) on the same set of elements used in Exercise 4.41.
- 4.43 Another priority queue operation that is sometimes useful is *ChangePriority*(Q, x, v), which changes the priority value of an element x in the queue Q to v (and maintains a priority queue). Give pseudocode for *ChangePriority* when the priority queue is implemented as a min-heap.
ChangePriority should have worst-case complexity $O(\log n)$, where n is the number of elements in the min-heap.
- 4.44 The concept of a complete binary tree extends to the concept of a complete k -ary tree in the obvious way, $k \geq 2$.
 a) Show how a complete k -ary tree can be implemented efficiently using an array.
 b) Design and analyze a procedure for inserting an element into a k -ary heap implemented using an array.
 c) Design and analyze a procedure for deleting an element from a k -ary heap implemented using an array.
 d) Design and analyze a k -ary heapsort.
 e) Compare the result in (d) to that for an ordinary heap sort ($k = 2$).

Section 4.7 Implementing Disjoint Sets

- 4.45 Prove Proposition 4.7.1.
- 4.46 Suppose we have the following parent implementation of a forest representing a partition of a set of 17 elements:

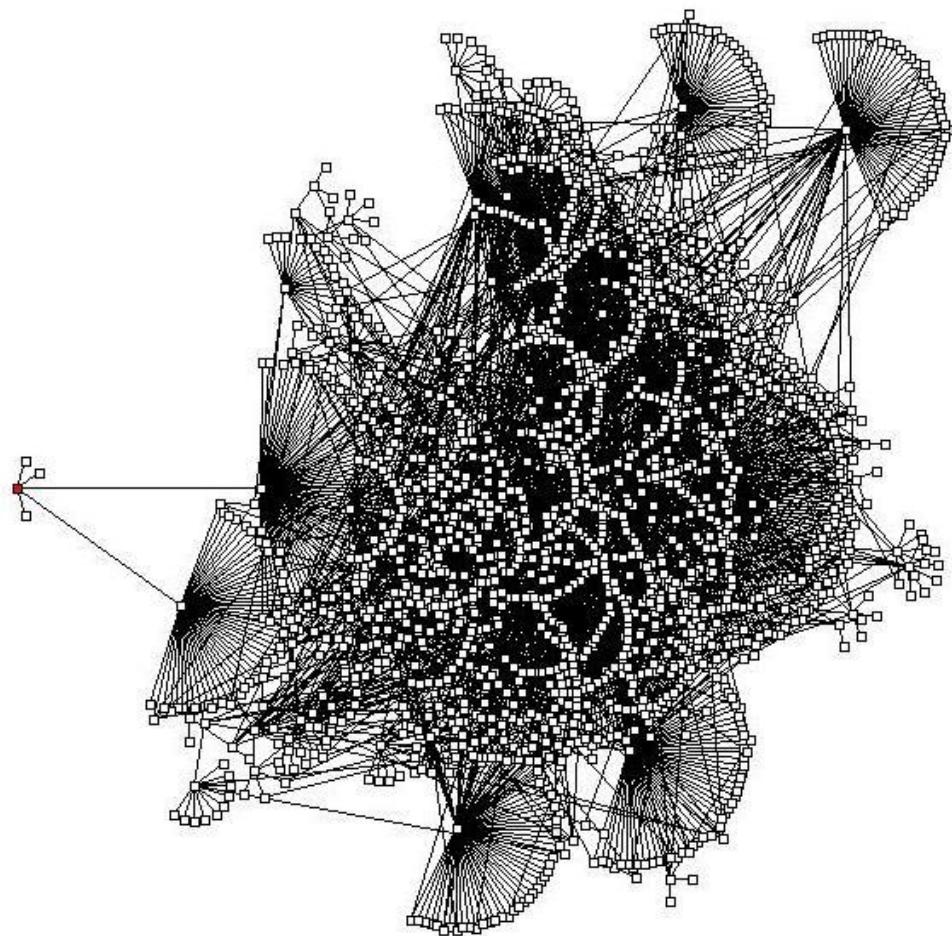
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$Parent[i]$	7	-3	8	7	14	0	1	8	-13	3	14	4	-1	3	3	0	1

- Sketch the trees in F .
- Show the state of $Parent[0:16]$ after a call to $Union(Parent[0:16], 1, 8)$ and sketch the trees in F .
- Given the state of $Parent[0:16]$ in part (b), show the state of $Parent[0:16]$ after an invocation of $Find2(Parent[0:16], 4)$ and sketch the trees in F .

5

Graphs and Digraphs

Many problems are naturally modeled using graphs and digraphs, and data structures implementing the graph or the digraph ADT are commonly used throughout computer science. The subject of graph algorithms is a very active area of research today. Graphs and digraphs also play an important role in the Internet and other networks, such as communication, transportation, commodity flow, and so forth. The underlying structure of these networks is naturally modeled using a graph or a digraph. For example, in interstate highway system the nodes of a graph may represent cities (or junctions) and the edges may represent highways linking these cities. As a second example, the so-called *web digraph* has nodes corresponding to webpages, and a directed arc exists from web page *A* to webpage *B* if there is a hyperlink reference (*href*) to web page *B* in webpage *A*. As a third example, in parallel computing graphs serve as models for interconnection networks. As a fourth example, we may represent the underlying topology of networks, such as the peer-to-peer network Gnutella (see Figure 5.1). Besides network applications, graphs and digraphs serve as natural models for a host of other applications. To cite an example from computer science, the logical flow of a computer program written in a high-level language is naturally modeled by a program digraph (flow chart). Optimizing compilers then utilize various properties of this digraph, such as strongly connected components and vertex coloring, to help achieve the goal of translating the high-level code into machine code that exhibits optimal performance.



A snapshot of a portion of the Gnutella peer-to-peer network

Figure 5.1

In this chapter we provide an introduction to some basic graph theory concepts. We also discuss the two basic search strategies known as depth-first search and breadth-first search, and apply these strategies to such problems as topological sorting, and finding shortest paths in graphs.

5.1 Graphs and Digraphs

In this section, we give a brief introduction to the theory of graphs and digraphs. We introduce some basic terminology, prove several elementary results, and present two standard representations of graphs and digraphs.

5.1.1 Graphs

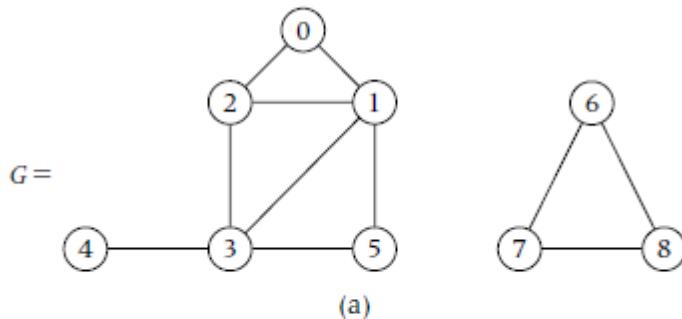
Formally, a *graph* $G = (V, E)$ is a set $V = V(G)$ called *vertices* (or *nodes*), together with a set $E = E(G)$ called *edges*, such that an edge is an *unordered* pair of vertices. An edge $\{u, u\}$ is called a

loop. Unless otherwise stated, we restrict our attention to graphs without loops. For simplicity, we sometimes denote the edge $\{u,v\}$ by uv . Given an edge $e = uv$ in a graph $G = (V,E)$, we refer to vertices u and v as the *end* vertices of e , and we say that e joins u and v . Two vertices u and v are *adjacent* if they are the two end vertices of an edge in the graph (that is, $uv \in E$). The set of all vertices adjacent to u is the *neighborhood* of u . A node is *isolated* if its neighborhood is empty. A vertex v and an edge e are *incident* if e contains vertex v , that is, if $e = vw$ for some vertex w . Two edges e and f are *adjacent* if they have an end vertex in common.

A graph can be represented pictorially by a drawing in the plane, where the vertices are represented by points in the plane and an edge $\{u,v\} \in E$ is represented by a continuous curve joining u and v (see Figure 5.2). Note from Figure 5.2b that such drawings allow the continuous curves to intersect at points (called *crossings*) other than vertices. It turns out that the graph in Figure 5.2b cannot be drawn in the plane without at least one crossing.

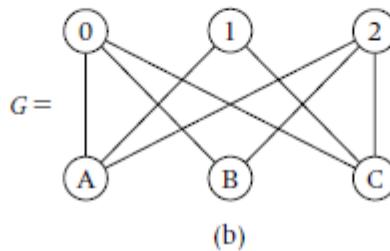
$$V = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$$

$$E = \{\{0,1\}, \{0,2\}, \{1,2\}, \{1,3\}, \{1,5\}, \{2,3\}, \{3,4\}, \{3,5\}, \{6,7\}, \{6,8\}, \{7,8\}\}$$



$$V = \{0, 1, 2, A, B, C\}$$

$$E = \{\{0,A\}, \{0,B\}, \{0,C\}, \{1,A\}, \{1,B\}, \{1,C\}, \{2,A\}, \{2,B\}, \{2,C\}\}$$



(a) A planar graph, and (b) a nonplanar

Figure 5.2

Graphs such as the one in Figure 5.2a, which can be drawn in the plane without crossings, are called *planar graphs*. Such a drawing is called an *embedding* of G in the plane. Planar graphs are particularly important in computer science. For example, they are useful in VLSI design and flow diagrams. If a graph is not planar, then we say that it is *nonplanar*.

Two graphs $G = (V,E)$ and $G' = (V',E')$ are *isomorphic* if there exists a bijective mapping $\beta: V \rightarrow V'$ from the vertex set V of G onto the vertex set V' of G' such that adjacency relationships are preserved; that is, $\{u,w\} \in E$ if, and only if, $\{\beta(u),\beta(w)\} \in E'$. The mapping β is called an *isomorphism*. Deciding whether two graphs are isomorphic is, in general, a difficult problem. A graph is *complete* if every pair of distinct vertices is joined by an edge. Clearly, any

two complete graphs having the same number of vertices are isomorphic. A complete graph on n vertices is denoted by K_n .

We denote the number of vertices and edges by $n = n(G)$ and $m = m(G)$, respectively, so that $n = |V|$ and $m = |E|$. The *degree* of a vertex $v \in V$, denoted by $d(v)$, is the number of edges incident with v . Let $\delta = \delta(G)$ and $\Delta = \Delta(G)$ denote the minimum and maximum degrees, respectively, over all the vertices in G , so that $\delta \leq d(v) \leq \Delta$ for all $v \in V$.

The following useful formula due to Euler relates the number m of edges to the sum of the degrees of the vertices.

Proposition 5.1.1

The sum of the degrees over all the vertices of a graph G equals twice the number of edges; that is,

$$\sum_{v \in V} d(v) = 2m. \quad (5.1.1)$$

Proof

Every edge is incident with exactly two vertices. Therefore, when summing the degrees over all the vertices we count each edge exactly twice, once for each of its end vertices. ■

A graph is *r-regular* if every vertex has degree r . A useful corollary of Proposition 5.1.1 is the following result relating the number of vertices and edges of an *r-regular* graph.

Corollary 5.1.2

If G is an *r-regular* graph with n vertices and m edges, then

$$m = \frac{rn}{2}. \quad (5.1.2) \quad \square$$

Another corollary of Proposition 5.1.1 is obtained by taking both sides of (5.1.1) **mod 2**.

Corollary 5.1.3

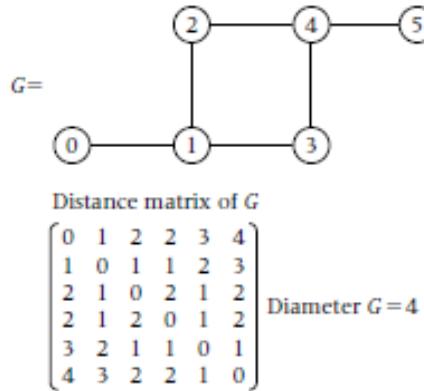
For any graph G , the number of vertices of odd degree is even. □

A *path* P of length p ($p \geq 0$) joining vertices u and v is an alternating sequence of $p + 1$ vertices and p edges $u_0e_1u_1e_2 \dots e_pu_p$ such that $u = u_0$, $v = u_p$, where e_i joins u_{i-1} and u_i , $i = 1, 2, \dots, p$. We call u_0 and u_p the *initial* and *terminal* vertices, respectively, and the remaining vertices in the path the *interior* vertices. Vertices in a path can be repeated, but edges must be distinct. Since the path P is completely defined by the sequence of vertices $u_0u_1 \dots u_p$, we often use this shorter sequence to denote P . If $u = v$, then the path is called a *closed path* or *circuit*. We call a path *simple* if the interior vertices in the path are all distinct and are different from the initial and terminal vertices. A *simple circuit* or *cycle* is a simple closed path. A path that contains every edge in the graph exactly once is called an *Eulerian path*. A circuit that contains every edge exactly once is called an *Eulerian circuit* or *Eulerian tour*. A simple path that contains every vertex in the graph is called a *Hamiltonian path*. A cycle that contains every vertex in the graph is called a *Hamiltonian cycle*.

Two vertices $u, v \in V$ are *connected* if there exists a path (of length 0 when $u = v$) that joins them. The relation u is *connected to* v is an equivalence relation on V . Each equivalence

class C of vertices, together with all incident edges, is called a (*connected*) *component* of G . When G has only one component, then G is *connected*; otherwise, G is *disconnected*.

The *distance* between u and v , denoted by $d(u,v)$, is the length of a path from u to v that has the shortest (minimum) length among all such paths. By convention, if u and v are not connected, then $d(u,v) = \infty$. The *diameter* of G is the maximum distance between any two vertices. In Figure 5.3 the distance between every pair of vertices and the diameter is given for a sample graph G . We show the distances using a 6-by-6 *distance matrix*, whose ij th entry is given by $d(i,j)$.



Distances between vertices for a sample graph G on six vertices

Figure 5.3

A *tree* is a connected graph without cycles. A rooted tree is simply a tree with one vertex designated as the root. Every tree is planar, so that it can be drawn in the plane without crossings. The following propositions are not difficult, and their proofs are left as exercises.

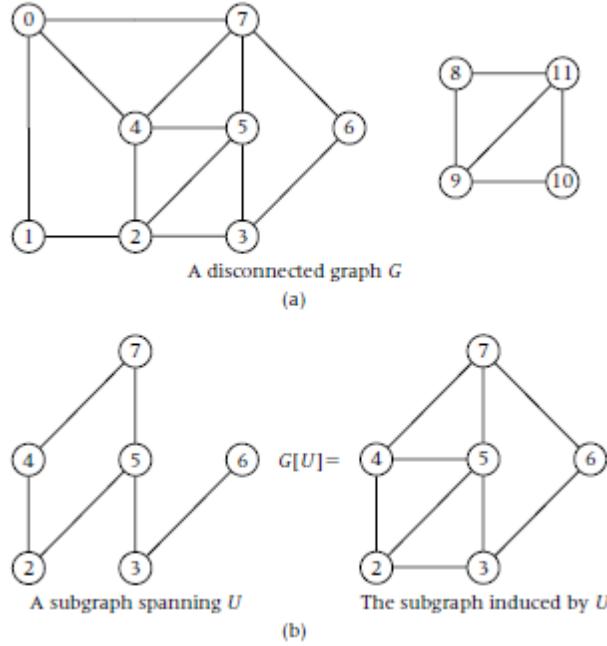
Proposition 5.1.4

A graph T is a tree if, and only if, there exists a unique path joining every pair of distinct vertices of T . □

Proposition 5.1.5

A connected graph T is a tree if, and only if, the number of edges of T is one less than the number of vertices. □

A *subgraph* H of G is a graph such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. A subgraph H that is a tree is a *subtree* of G , or simply a *tree* of G . A subgraph H of G is called a *spanning* subgraph if H contains all the vertices of G . When a subgraph H of G is a tree, we use the term *spanning tree* of G . Given a subset U of vertices of G , the subgraph $G[U]$ *induced* by U is the subgraph with vertex set U and edge set consisting of all edges in G having both end vertices in U . In Figure 5.4b we show a subgraph and an induced subgraph on the same set of vertices U . Note that a component of a graph G is a connected induced subgraph of G that is not contained in a strictly larger connected subgraph (see Figure 5.4a).



(a) The components of G are the subgraphs, $G[A]$ and $G[B]$, induced by the vertex sets $A = \{0,1,2,3,4,5,6,7\}$ and $B = \{8,9,10,11\}$, respectively; (b) a subgraph spanning $U = \{2,3,4,5,6,7\}$ and the induced subgraph $G[U]$

Figure 5.4

A graph is *bipartite* if there exists a bipartition of the vertex set V into two sets X and Y such that every edge has one end in X and the other in Y . The *complete bipartite* graph $K_{i,j}$ is the bipartite graph with $n = i + j$ vertices, i vertices belonging to X and j vertices belonging to Y , such that there is an edge joining every vertex $x \in X$ to every vertex $y \in Y$. The complete bipartite graph $K_{3,3}$ is shown in Figure 5.2b. Two other examples of complete bipartite graphs are given in Figure 5.5.

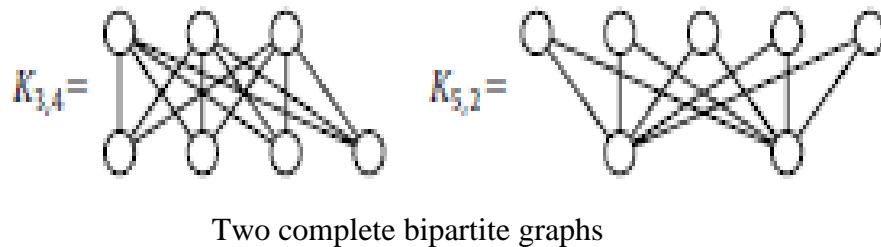
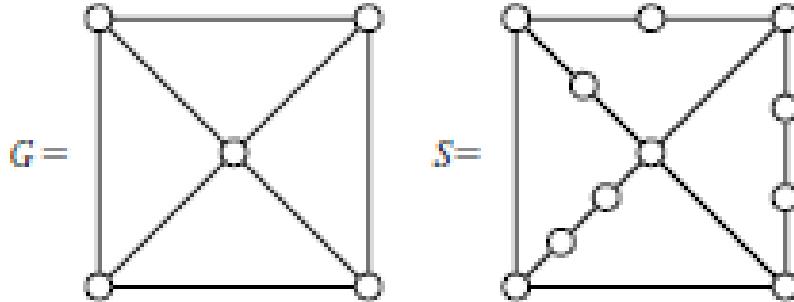


Figure 5.5

The graphs K_5 and $K_{3,3}$ are both nonplanar graphs. If you try drawing them in the plane without edge crossings, you will have trouble. In fact, K_5 and $K_{3,3}$ are in some sense the quintessential nonplanar graphs (see Theorem 5.1.6). A *subdivision* S of G is a graph obtained from G by replacing each edge e with a path joining the same two vertices as e (subdividing the edge e , as in Figure 5.6). Observe that G is a subdivision of itself (replace each edge with a path

of length 1). Clearly, if G is nonplanar, then every subdivision of G is nonplanar. It is also clear that if G contains a nonplanar graph, then G is nonplanar. It follows that a graph is nonplanar if it contains a subgraph that is a subdivision of K_5 or $K_{3,3}$. The fact that the converse is also true is a surprising and deep result discovered by Kuratowski. We state Kuratowski's theorem without proof.



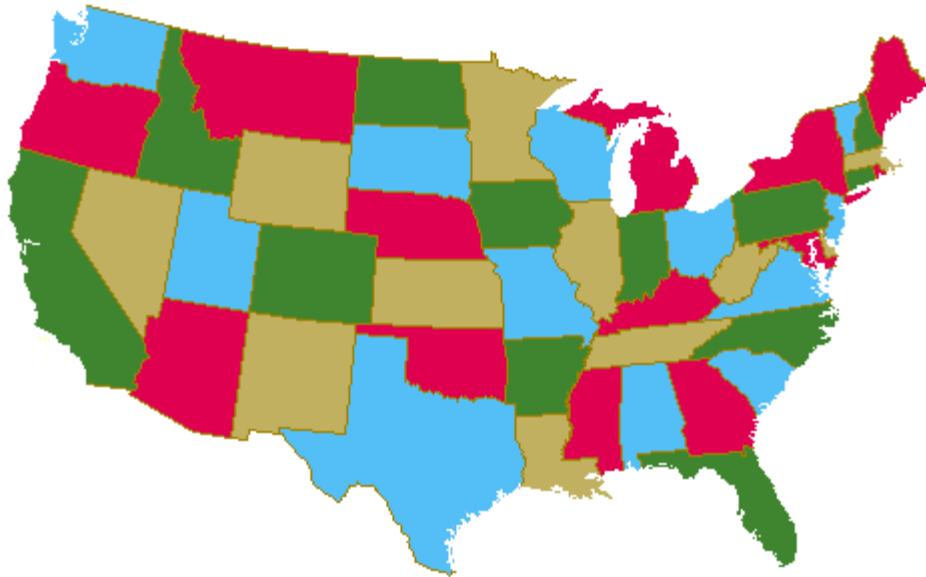
Subdividing a graph

Figure 5.6

Theorem 5.1.6 (Kuratowski's Theorem)

A graph G is nonplanar if, and only if, it contains a subgraph that is (isomorphic to) a subdivision of K_5 or $K_{3,3}$. \square

The condition of being planar occurs in many computer applications, such as the design of VLSI circuits. Moreover, planar graphs became of interest long before the advent of computers. For example, in 1750 Euler gave his famous polyhedron formula relating the number of vertices, edges, and faces of a connected planar graph. As another example, a very famous mathematical question known as the four color conjecture can be modeled using planar graphs. For centuries map makers had known implicitly that they only needed four colors to color the countries in any map drawn on the globe so that no two neighboring countries (that is, countries sharing a common boundary consisting of more than an isolated point) get the same color (called a *proper* coloring). See Figure 5.7 for a four-coloring of the map of the 48 contiguous states of the U.S.A. That four colors suffice for any map on the globe was stated by Guthrie in 1856 as a formal conjecture, which resisted proof for over 100 years.



A proper 4-coloring of the map of the USA (color Alaska and Hawaii as you like)

Figure 5.7

Given any map on the globe, there is a naturally associated planar graph whose vertices correspond to the countries, and where two vertices are adjacent if, and only if, the countries they represent are neighbors. It is easy to see that the graph associated with a map on the globe is planar. The map coloring problem then transforms to the equivalent problem of coloring the vertices of a planar graph using no more than four colors so that no two adjacent vertices get the same color. Certainly the transformed problem has a rather elegant mathematical formulation unencumbered by geometrically complex boundary curves associated with maps. Moreover, graphs can be input to a computer using simple data structures such as adjacency matrices, so proofs involving exhaustive case checking are sometimes possible. In fact, it was just that type of proof (together with deep mathematical insight) that was used by Appel and Haken in 1970 to settle the four color conjecture in the affirmative. The proof consisted in showing that two contradictory conditions hold for planar graphs that are not four-colorable: there is a certain finite set of planar graphs S (at least) one of which would have to occur as a subgraph of any planar graph G that is not four-colorable (S is an *unavoidable* set), and if a graph G contains one of these subgraphs, then there would be another graph G' on fewer vertices that is also not four-colorable (each graph in S is *reducible*).

Clearly, these two conditions rule out the existence of a planar graph that is not four-colorable, since if such things exist there would have to be one having a minimal number of vertices amongst all such graphs. Appel and Haken proved condition (1) mathematically, but used the computer to check that each graph in S had the reducibility property given in condition (2). Since then other proofs of the four color conjecture have been given along the same lines but using a smaller set of unavoidable reducible graphs. However, exhaustive computer checking of cases remains a component of all known proofs of the four color conjecture.

For an arbitrary graph G and an integer c , it is a difficult problem (in fact, it is NP-complete) to determine whether or not the vertices of G can be colored using no more than c colors such that no two adjacent vertices get the same color (called a proper coloring of G). The minimum number of colors needed to properly color the vertices of G is called the *chromatic number* of G . This invariant is discussed further in the exercises at the end of this chapter.

5.1.3 Digraphs

Graphs with directed edges are natural models for many physical phenomena. For example, the pairings of the players in a (match-play golf, table tennis, etc.) tournament can be represented by a graph, where the vertices correspond to the players and where two vertices A and B are joined with an edge if players A and B are to play each other. The winner of the match between each pair of players A and B can be recorded by “orienting” the edge joining A and B , so that it is directed from A to B if A defeats B . The resulting oriented graph is called a *directed graph* or *digraph*.

Formally, a *digraph* D is a set $V = V(D)$ of vertices together with a set $E = E(D)$ called *directed edges* (sometimes called *arcs*) such that a directed edge is an *ordered* pair of vertices. A digraph can be represented by a drawing in the plane in the same way that a graph can, except that we add an arrow to the curve representing each edge to indicate the orientation of that edge (see Figure 5.8).

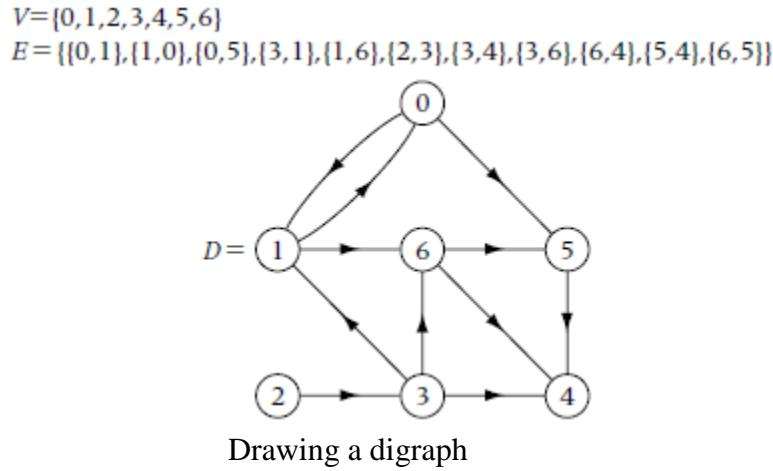
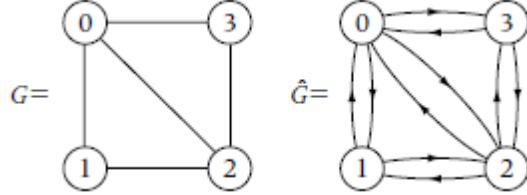


Figure 5.8

Since an unordered pair $\{a,b\}$ is combinatorially equivalent to the two ordered pairs (a,b) , (b,a) , a digraph is actually a generalization of a graph. Associated with any given undirected graph $G = (V,E)$ is the combinatorial equivalent digraph $\hat{G} = (\hat{V}, \hat{E})$, where $V = \hat{V}$, and where the ordered pairs (u,v) , (v,u) are in \hat{E} if, and only if, the unordered pair $\{u,v\}$ is in E (see Figure 5.9).



Graph G and its combinatorial equivalent digraph \hat{G}

Figure 5.9

For simplicity, we sometimes denote the directed edge (u,v) by uv . We refer to u as the *tail* of e and v as the *head* of e . The *out-degree* of a vertex $v \in V$, denoted by $d_{\text{out}}(v)$ is the number of edges having tail v . Similarly, the *in-degree* of a vertex $v \in V$, denoted by $d_{\text{in}}(v)$, is the number of edges having head v . It is easily verified that

$$\sum_{v \in V} d_{\text{out}}(v) = \sum_{v \in V} d_{\text{in}}(v) = m,$$

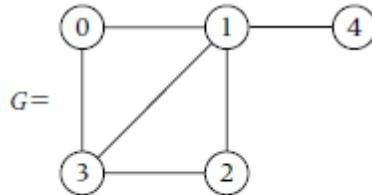
where m denotes the number of directed edges of D . The *out-neighborhood (in-neighborhood) of vertex u* is the set of all vertices v such that uv (vu) is a directed edge of D .

5.2 Implementing Graphs and Digraphs

Two standard implementations of a graph G are the adjacency matrix implementation and the adjacency lists implementation. For convenience, assume that the vertices of G are labeled $0, 1, \dots, n-1$ (when referring to a vertex we do not distinguish between the label of the vertex and the vertex itself). The *adjacency matrix* of a graph G is the $n \times n$ symmetric matrix $A = (a_{ij})$ given by

$$a_{ij} = \begin{cases} 1 & \text{vertices } i \text{ and } j \text{ are adjacent in } G. \\ 0 & \text{otherwise,} \end{cases} \quad i, j \in \{0, \dots, n-1\}.$$

A sample graph G and its adjacency matrix are given in Figure 5.5.



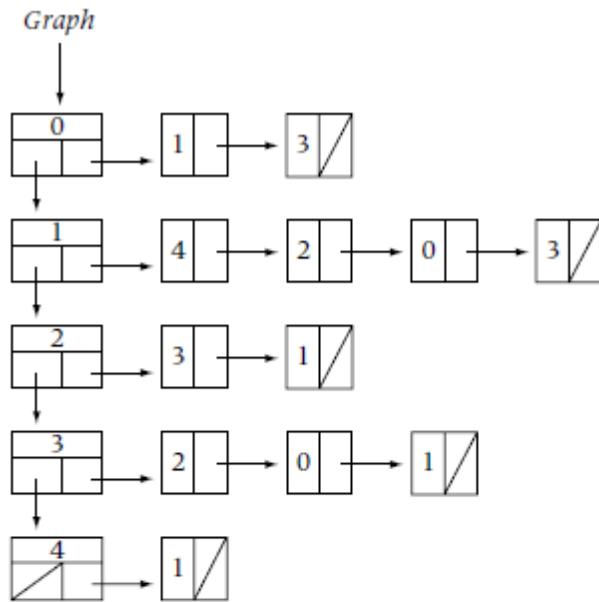
$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Adjacency matrix of a graph G

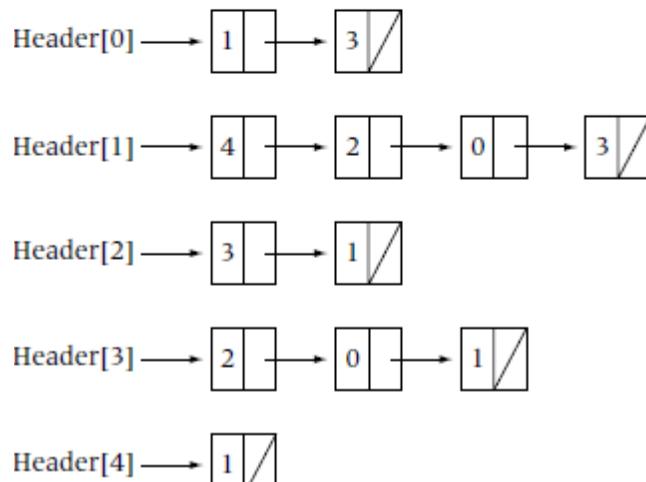
Figure 5.10

Implementing G using its adjacency matrix makes it easy to perform many standard operations on G , such as adding a new edge or deleting an existing edge. Note that the adjacency matrix of G allocates n^2 memory locations no matter how many edges are in the graph. Thus, implementing G using its adjacency matrix is inefficient if the number of edges of G is small relative to the number of vertices. For example, if G is a tree with n vertices, then G has only $n - 1$ edges. On the other hand, if $m \in \Theta(n^2)$, then the adjacency matrix representation is probably an efficient way to implement G .

Given a graph $G = (V, E)$, for $v \in V$, the *adjacency list of v* is the list consisting of all the vertices adjacent to v . The order in which the vertices are listed usually does not matter. The adjacency lists of G are generally implemented as linked lists, where pointers to the beginning of the linked lists are stored in *header* nodes. The header nodes can themselves be implemented using a linked list, or they can be stored in an array $\text{Header}[0:n - 1]$. Figures 5.11a and 5.11b illustrate the adjacency lists implementation using linked lists for the graph given in Figure 5.10, where the header nodes are implemented using a linked list and an array, respectively. We do not assume that the nodes in each adjacency list are maintained in increasing order of node labels.



(a) Linked list of header nodes



(b) Array of header nodes

Adjacency list implementations for the sample graph G given in Figure 5.10

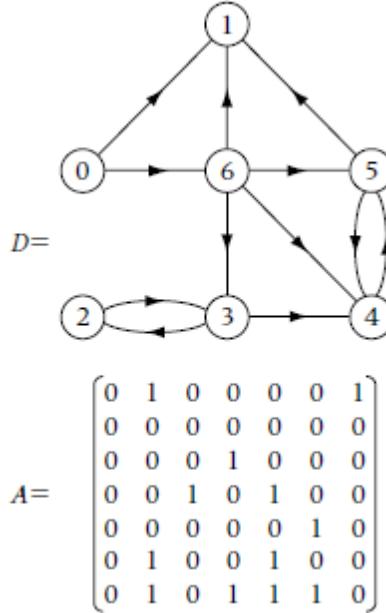
Figure 5.11

Each node of the adjacency list of vertex i corresponds to an edge $\{i,j\}$ incident with i . In the adjacency lists implementation of a graph, where the header nodes belong to a linked list (see Figure 5.11a), it is convenient in practice to maintain a second pointer p in each list node defined as follows: If the list node belongs to the adjacency list of vertex i and corresponds to edge $\{i,j\}$ then p points to the header node of adjacency list j .

Both the adjacency matrix and adjacency lists implementations of graphs generalize naturally to digraphs. The *adjacency matrix* of a digraph D , whose vertices are labeled $0, 1, \dots, n - 1$, is the $n \times n$ matrix $A = (a_{ij})$ defined by

$$a_{ij} = \begin{cases} 1 & \text{if there is an arc from } i \text{ and } j, \\ 0 & \text{otherwise,} \end{cases} \quad i, j \in \{0, \dots, n-1\}.$$

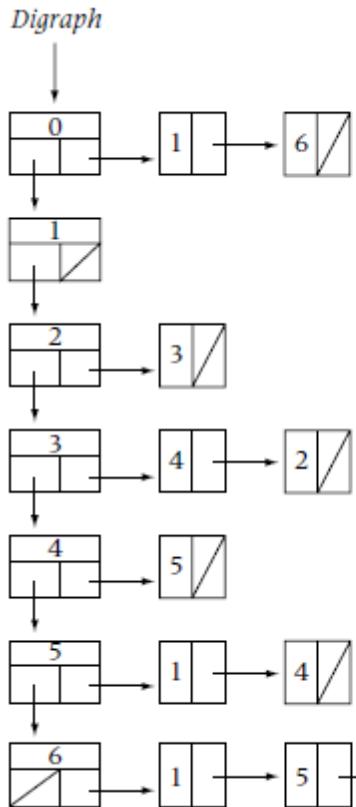
The adjacency matrix of a sample digraph D is given in Figure 5.12.



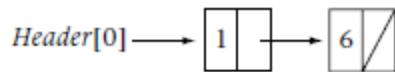
Adjacency matrix of a digraph D

Figure 5.12

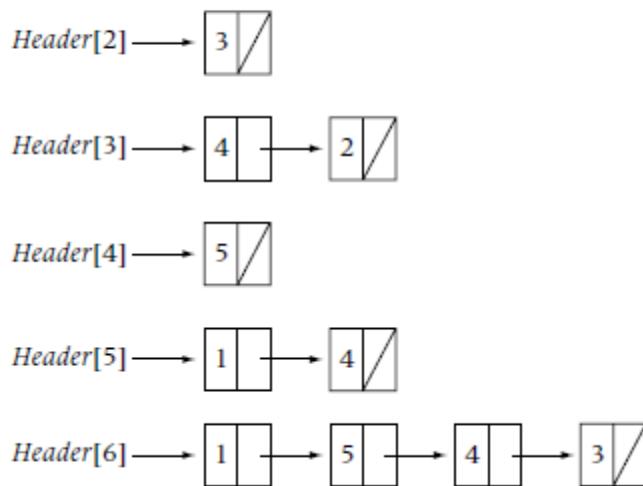
In Figure 5.13a and 5.13b, the adjacency lists implementations using a linked list of header nodes and an array of header nodes, respectively, are illustrated for the sample digraph given in Figure 5.13.



(a) Linked list of header nodes



Header[1]=null



(b) Array of header nodes

Adjacency list implementations of the sample digraph D given in Figure 5.12

Figure 5.13

The adjacency matrix and adjacency lists representations of digraphs are generalizations of the adjacency matrix and adjacency lists representations of (undirected) graphs. To see this, consider any undirected graph G and its combinatorially equivalent digraph \hat{G} (see Figure 5.9). Clearly, the adjacency matrix and adjacency lists of G and \hat{G} are identical.

5.3 Graphs as Interconnection Networks

The underlying structure for an interconnection network model for parallel communication is a graph. The nodes of the graph correspond to the processors. Two nodes are joined with an edge (adjacent) whenever the corresponding two processors communicate directly with one another. In the interconnection network model, information is communicated between nonadjacent processors P and Q by relaying the information along a path in the network joining P and Q . The diameter is an upper bound on the number of communication steps needed to relay information between any two processors. Having small diameter is clearly a desirable property of an interconnection network. Another desirable property is having small maximum degree, since such interconnection network models are easier to build than those having large maximum degree.

The complete graph interconnection network K_n having n processors has diameter 1 (since the distance between every pair of vertices is 1), but is not practical to build since each vertex has very high degree ($n - 1$). At the other extreme is the one-dimensional mesh M_p , $p = n$, which has maximum degree 2 and diameter $n - 1$ (the distance between processor P_0 and processor P_{p-1} is $n - 1$). The two-dimensional $M_{q,q}$ ($n = q^2$) has maximum degree 4 but has diameter $2(\sqrt{n} - 1)$, which is still quite large (the maximum distance $2(q - 1)$ occurs between processors $P_{0,0}$ and $P_{q-1,q-1}$).

There are some other interconnection networks based on graphs that achieve nice compromises between extreme values of degree and diameter. One such example is the hypercube interconnection network model. The k -dimensional hypercube H_k has 2^k nodes consisting of the set of all 0/1 k -tuples; that is, $V(H_k) = \{(x_1, \dots, x_k) \mid x_i \in \{0,1\}, i = 1, \dots, k\}$. Two nodes in $V(H_k)$ are joined with an edge of H_k if, and only if, they differed in exactly one component. We illustrate the hypercubes up to dimension 4 in Figure 11.8. Note that the hypercube of dimension k can be obtained by joining corresponding vertices in two copies of the hypercube of dimension $k - 1$.

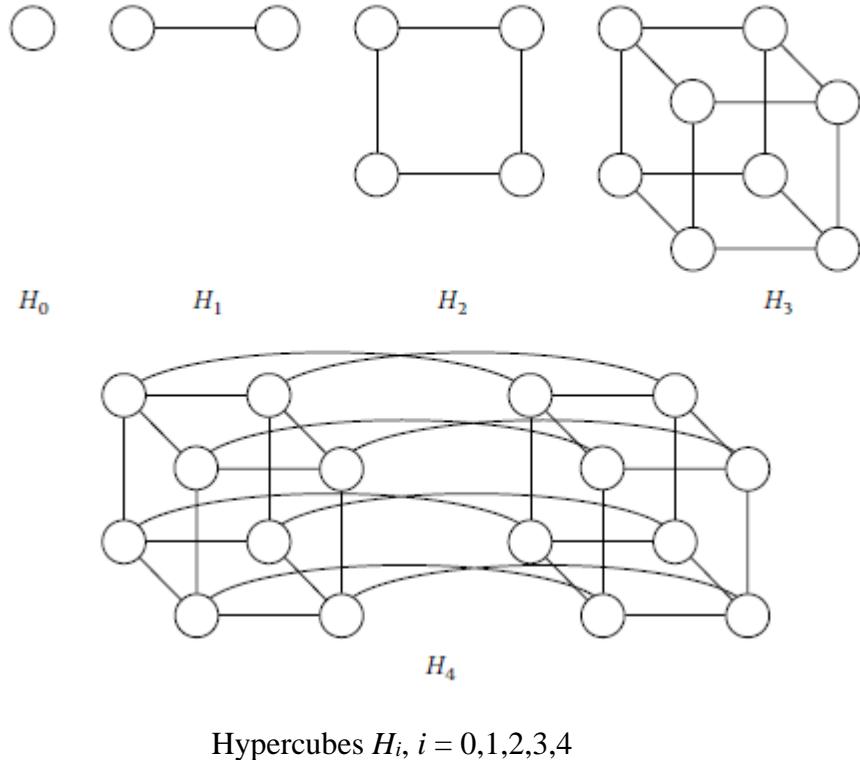


Figure 5.14

The hypercube H_k of dimension k is k -regular and has diameter $k = \log_2 n$ (see Exercise 5.x). Thus, the maximum degree and diameter of H_k are both logarithmic in the number of vertices n .

5.4 Search and Traversal of Graphs

The solutions to many important problems require an examination (visit) of the nodes of a graph. Two standard search techniques are *depth-first search* and *breadth-first search*.

Depth-first search and breadth-first search differ in their exploring philosophies: Depth-first search always longs to see what's over the next hill (where pastures might be greener), whereas breadth-first search visits the immediate neighborhood thoroughly before moving on. After visiting a node, breadth-first search explores this node (visits all neighbors of the node that have not already been visited) before moving on. On the other hand, depth-first search immediately moves on to an unvisited neighbor, if one exists, after visiting a node. Whenever depth-first search is at an explored node, it “backtracks” until an unexplored node is encountered and then continues. This backtracking often returns to the same node many times before it is explored.

5.4.1 Depth-First Search

We first give the pseudocode *DFS* for depth-first search. For simplicity, we assume that G has n vertices labeled $0, 1, \dots, n-1$; we use the symbol v to simultaneously denote a vertex and its label. We maintain an auxiliary array $Mark[0:n-1]$ to keep track of the nodes that have been

visited.

procedure *DFS*(*G,v*) **recursive**

Input: *G* (a graph with *n* vertices and *m* edges)

v (a vertex) //The array *Mark*[0:*n*–1] is global and initialized to 0s

Output: the depth-first search of *G* with starting vertex *v*

Mark[*v*] \leftarrow 1

call *Visit*(*v*)

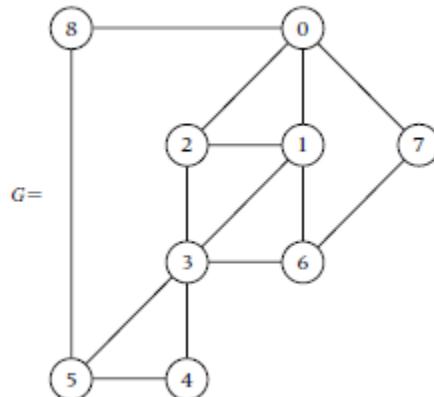
for each vertex *u* adjacent to *v* **do**

if *Mark*[*u*] = 0 **then call** *DFS*(*G,u*) **endif**

endfor

end DFS

The **for** loop in the *DFS* pseudocode did not explicitly describe the order in which the vertices are considered. This order is incidental to the nature of the search, and would, in general, depend on the particular implementation of the graph *G* (for example, adjacency matrix, adjacency lists, and so forth). Since we have labeled the vertices 0,1, …, *n*–1, we assume that the vertices are accessed by the **for** loop in increasing order of their labels (thereby making the order of visiting the nodes independent of the implementation). The graph in Figure 5.15 illustrates this convention.



DFS with *v* = 6 visits vertices in the order 6,1,0,2,3,4,5,8,7

Figure 5.15

For the graph *G* in Figure 5.15, *DFS* visits all the vertices of *G*. For a general graph *G*, *DFS* with starting vertex *v* visits all the vertices in the (connected) component containing *v*.

To further illustrate the **for** loop in *DFS*, we show how this loop can be written in the two standard ways to implement a graph. Suppose first that *G* is implemented using its adjacency matrix *A*[0:*n*–1,0:*n*–1]. We assume that the vertex *v* is labeled *i*. In this case the **for** loop becomes

```

for j  $\leftarrow$  0 to n–1 do
    if (A[i,j] = 1) .and. (Mark[j] = 0) then
        call DFS(G,j)

```

```

endif
endfor

```

On the other hand, suppose that G is implemented using adjacency lists. Recall that a typical version of this implementation has an array $\text{Header}[0:n-1]$ of header nodes, where $\text{Header}[v]$ is a pointer to the adjacency list of v . A node in the adjacency list for v corresponding to vertex u contains a field Vertex containing the index (label) of u . It also contains a pointer NextVertex to the next vertex in the adjacency list. Under these assumptions, the **for** loop in DFS becomes

```

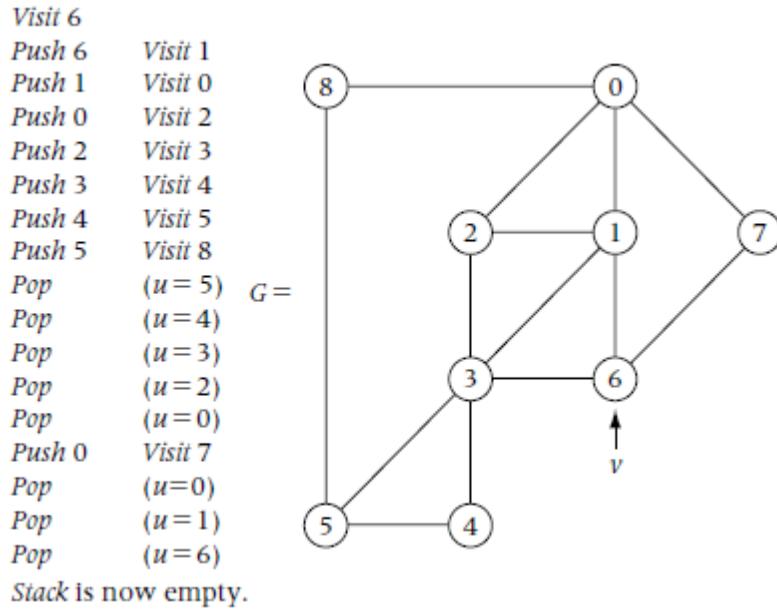
 $p \leftarrow \text{Header}[v]$ 
while ( $p \neq \text{null}$ ) do
    if  $\text{Mark}[p \rightarrow \text{Vertex}] = 0$  then
        call  $\text{DFS}(G, p \rightarrow \text{Vertex})$ 
    endif
     $p \leftarrow p \rightarrow \text{NextVertex}$ 
endwhile

```

It is useful to write DFS as a nonrecursive procedure. For convenience, the nonrecursive version calls a procedure Next that determines the next unvisited node w adjacent to the node u just visited. If no such node w exists, then a Boolean parameter found is set to be **.false.**.

procedure $\text{DFS}(G, v)$ Input: G (a graph with n vertices and m edges) v (a vertex) Output: the depth-first search of G starting from vertex v S a stack initialized as empty $\text{Mark}[0:n-1]$ a 0/1 array initialized to 0s $\text{Mark}[v] \leftarrow 1$ call $\text{Visit}(v)$ $u \leftarrow v$ $\text{Next}(u, w, \text{found})$ while found or. (.not. $\text{Empty}(S)$) if found then //go deeper $\quad \text{Push}(S, u)$ $\quad \text{Mark}[w] \leftarrow 1$ $\quad \text{Visit}(w)$ $\quad u \leftarrow w$ else $\quad \text{Pop}(S, u)$ //backtrack endif $\text{Next}(u, w, \text{found})$ endwhile end DFS

Figure 5.16 illustrates the sequence of pushes, visits, and pops performed by *DFS* for the graph of Figure 5.15.



DFS with $v = 6$ with stack operations illustrated

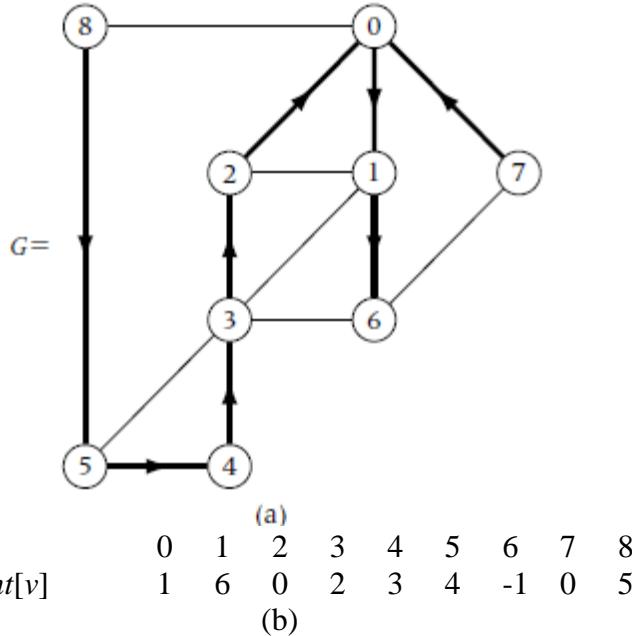
Figure 5.16

We analyze the complexity of *DFS* from the point of view of two basic operations: visiting a node and examining a node (by calls to *Next*) to see if it has been marked as visited. The worst-case complexity for both operations occurs when the graph is connected. If G is connected, then it is easily verified that every vertex is visited exactly once, so that we have a total of n node visits. Each edge uw in the graph gives rise to exactly two vertex examinations by *Next*, one with u as input parameter and one with w as input parameter. This shows that the worst-case complexity of *DFS* in terms of the number of vertices examined by *Next* is $2m$. Therefore, the total number of basic operations of the two types performed by *DFS* in the worst case is $n + 2m \in O(n + m)$.

5.4.2 Depth-First Search Tree

Depth-first search with starting vertex v determines a tree, called the *depth-first search tree* (*DFS-tree*) *rooted at* v . During a depth-first search, whenever we move from a vertex u to an adjacent unvisited vertex w , we add the edge uw to the tree. This tree is naturally implemented using the parent array representation $\text{Parent}[0:n-1]$ where u is the parent of w . For example, in the nonrecursive procedure *DFS*, we merely need to add the statement $\text{Parent}[w] \leftarrow u$ before pushing u on the stack, and add the array $\text{Parent}[0:n-1]$ as a third parameter.

The depth-first search tree rooted at vertex 6 of the graph of Figure 5.16 is illustrated in Figure 5.17a, and the associated array *Parent* is given in Figure 5.17b.



(a) The *DFS* tree rooted at vertex 6 for the graph given in Figure 5.16; (b) the associated array $Parent[0:8]$

Figure 5.17

The array $Parent[0:n-1]$ gives us the unique path from any given vertex w back to v in the depth-first search tree rooted at v . For example, suppose we wish to find the path in the depth-first search tree of Figure 5.17 from vertex 8 to vertex 6. By simply starting at vertex 7 and following the pointers in the array $Parent[0:n-1]$, we obtain the path

8, $Parent[8]=5$, $Parent[5]=4$, $Parent[4]=3$, $Parent[3]=2$, $Parent[2]=0$,
 $Parent[0]=1$, $Parent[1]=6$.

5.4.3 Depth-First Traversal

If the graph G is connected, then *DFS* visits *all* the vertices of G so that *DFS* performs a *traversal* of G . For a general graph G , the following simple algorithm based on repeated calls to *DFS* with different starting vertices performs a traversal of G .

In the pseudocode that follow we utilize the version *DFS* that keeps track of the *DFS* tree by maintain its parent array. In particular, we call $DFS(G, v, Parent[0:n-1])$ where $Parent[0:n-1]$ is an input /output parameter . This results in a forest of *DFS*-trees being generated (a *DFS*-tree rooted at i is generated for each i such that i is not visited when $v = i$). We refer to this forest as the *depth-first traversal forest* or *DFT-forest* of G .

```

procedure DFT( $G, Parent[0:n - 1]$ )
Input:  $G$  (a graph with  $n$  vertices and  $m$  edges)
Output: Depth-first traversal
     $Parent[0:n - 1]$  (an array implementing the DFT forest of  $G$ )
     $Mark[0:n - 1]$  a 0/1 array initialized to 0s
    for  $v \leftarrow 0$  to  $n - 1$  do
         $Parent[v] = -1$ 
    endfor
    for  $v \leftarrow 0$  to  $n - 1$  do
        if  $Mark[v] = 0$  then
             $DFS(G, v, Parent[0:n - 1])$ 
        endif
    endfor
end DFT

```

5.4.4 Breadth-First Search

Recall that the strategy underlying breadth-first search is to visit all unvisited vertices adjacent to a given vertex before moving on. The breadth-first strategy is easily implemented by visiting these adjacent vertices and placing them on a queue. Then a vertex is removed from the queue, and the process repeated. Breadth-first search starts by inserting a vertex v onto an initially empty queue, and continues until the queue is empty.

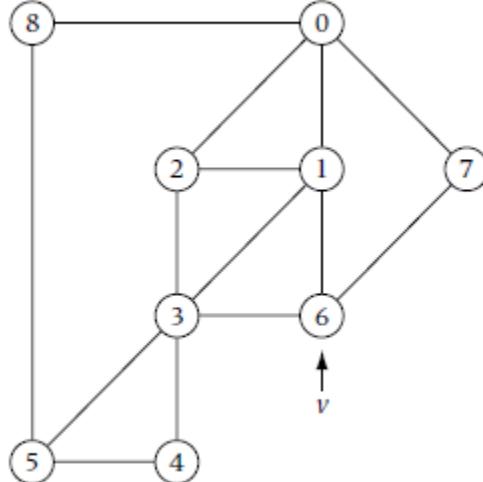
```

procedure BFS( $G, v$ )
Input:  $G$  (a graph with  $n$  vertices and  $m$  edges)
     $v$  (vertex) //the array  $Mark[0:n - 1]$  is global and initialized to 0s
Output: the breadth-first search of  $G$  starting from vertex  $v$ 
     $Q$  a queue initialized as empty
    call Enqueue( $Q, v$ )
     $Mark[v] \leftarrow 1$ 
    call Visit( $v$ )
    while .not.  $Empty(Q)$  do
        Dequeue( $Queue, u$ )
        for each vertex  $w$  adjacent to  $u$  do
            if  $Mark[w] = 0$  then
                Enqueue( $Q, w$ )
                 $Mark[w] \leftarrow 1$ 
                Visit( $w$ )
            endif
        endfor
    endwhile
end BFS

```

Clearly, *BFS* has the same $O(n + m)$ complexity as *DFS*. Figure 5.18 illustrates *BFS* starting at vertex 6 for the same graph as given in Figure 5.16.

<i>Enqueue</i> 6	<i>Visit</i> 6
<i>Dequeue</i>	($u = 6$)
<i>Enqueue</i> 1	<i>Visit</i> 1
<i>Enqueue</i> 3	<i>Visit</i> 3
<i>Enqueue</i> 7	<i>Visit</i> 7
<i>Dequeue</i>	($u = 1$)
<i>Enqueue</i> 0	<i>Visit</i> 0
<i>Enqueue</i> 2	<i>Visit</i> 2
<i>Dequeue</i>	($u = 3$)
<i>Enqueue</i> 4	<i>Visit</i> 4
<i>Enqueue</i> 5	<i>Visit</i> 5
<i>Dequeue</i>	($u = 7$)
<i>Dequeue</i>	($u = 0$)
<i>Enqueue</i> 8	<i>Visit</i> 8
<i>Dequeue</i>	($u = 2$)
<i>Dequeue</i>	($u = 4$)
<i>Dequeue</i>	($u = 5$)
<i>Dequeue</i>	($u = 8$)
<i>Queue is now empty.</i>	

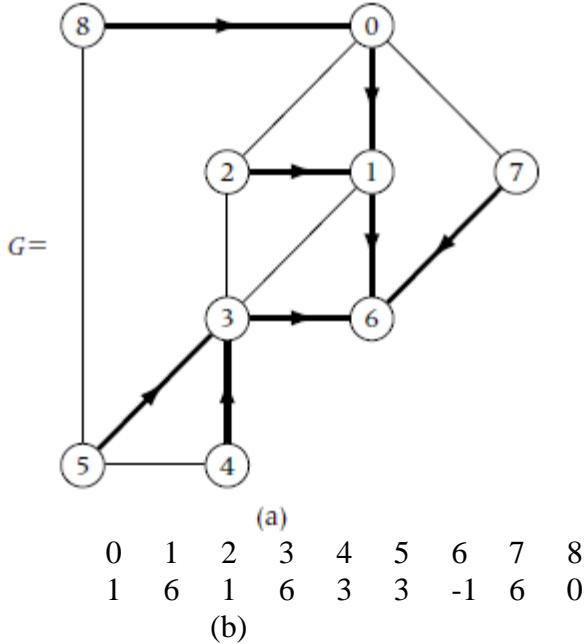


BFS with $v = 6$ visits vertices in the order 6,1,3,7,0,2,4,5,8

Figure 5.18

Breadth-first search with starting vertex v determines a tree spanning the component of the graph G containing v . As with *DFS*, a minor modification of the pseudocode for *BFS* generates this *breadth-first search tree (BFS-tree)* rooted at v . We maintain a parent array $\text{Parent}[0:n - 1]$ and when enqueueing a non-visited vertex w adjacent to u , we define the parent of w to be u .

The breadth-first search tree rooted at vertex 6 and the associated array Parent for the graph in Figure 5.18 is illustrated in Figure 5.19.



BFS tree rooted at vertex 6 and its associated array $Parent[0:8]$

Figure 5.19

Just as was the case for depth-first search, breadth-first search can be used to determine whether or not the graph G is connected. Also, a breadth-first *traversal* of an arbitrary graph (connected or not) is obtained from the following algorithm.

procedure $BFT(G)$

Input: G (a graph with n vertices and m edges)

Output: Breadth-first traversal

$Parent[0:n-1]$ (an array implementing the BFT forest of G)

$Mark[0:n-1]$ a 0/1 array initialized to -1's

for $v \leftarrow 0$ **to** $n-1$ **do**

$Parent[v] \leftarrow -1$

endfor

for $v \leftarrow 0$ **to** $n-1$ **do**

if $Mark[v] = -1$ **then**

$BFS(G, v, Parent[0:n-1])$

endif

endfor

end BFT

Similar to depth-first traversal, breadth-first traversal generates a breadth-first search forest as implemented by the parent array $BFSForestParent[0:n-1]$.

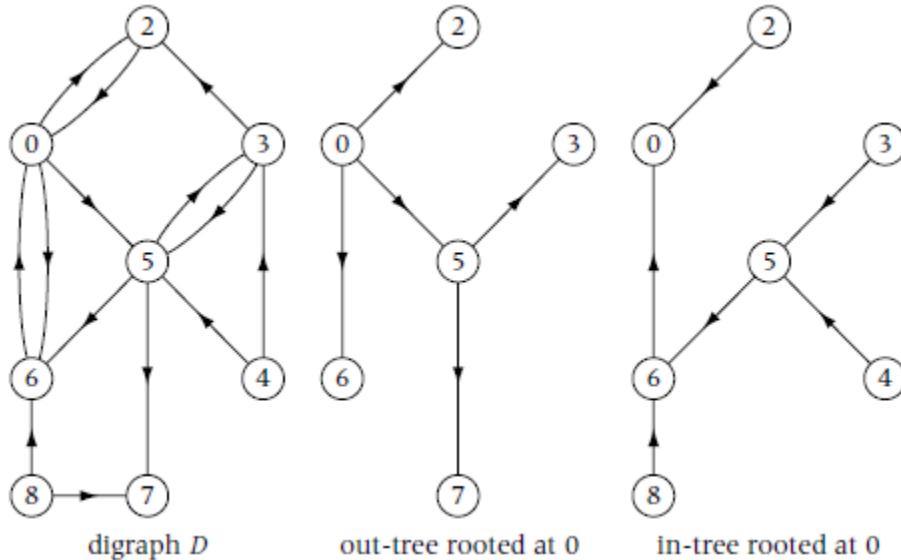
5.4.5 Breadth-First Search Tree and Shortest Paths

The breadth-first search tree starting at vertex v is actually a *shortest path tree* in the graph rooted at v ; that is, it contains a shortest path from v to every vertex in the same component as v . We leave the proof of this shortest path property as an exercise. The shortest path property is in sharp contrast to the paths generated by depth-first search. Indeed, we have seen that paths generated by depth-first search with starting vertex v are often rather longer than shortest paths to v , a fact well illustrated by the complete graph K_n on n vertices. Given any $v \in V(K_n)$ breadth-first search generates (shortest) paths of length 1 from v to all other vertices. On the other hand, depth-first search applied to K_n generates a depth-first search tree that is a path of length $n - 1$.

Clearly, any algorithm that finds a shortest path tree must visit each vertex and examine each edge. Thus, a lower bound on the complexity of the shortest path problem is $\Omega(n + m)$. Since the algorithm *BFSTree* has worst-case complexity $O(n + m)$, it is an (order) optimal algorithm.

5.4.6 Searching and Traversing Digraphs

Each search and traversal technique for a graph G generalizes naturally to a digraph D . It is convenient to define both an in-version and an out-version of these searches and traversals (the out-version is equivalent to the in-version in the digraph with the edge orientations reversed). A *directed path from u to v* is a sequence of vertices $u = w_0w_1 \dots w_p = v$, such that w_iw_{i+1} is a directed edge of D , $i = 0, \dots, p - 1$. For convenience, we sometimes refer to a directed path simply as a path. Given a vertex r in a digraph D , an *out-tree T rooted at r* is a minimal subdigraph that contains a directed path from r to any other vertex v in T . An *in-tree T rooted at r* is defined analogously. See the example shown in Figure 5.20.



An in-tree and an out-tree of a digraph D

Figure 5.20

Corresponding to the algorithm DFS for graphs we have the two algorithms indirected depth-first search $DFSIn$ and out-directed depth-first search $DFSOut$. Analogously, we have algorithms $BFSIn$, $BFSOut$ for breadth-first search, and similar algorithms for breadth-first traversals of digraphs.

Figure 5.21a illustrates the sequence of pushes, visits, and pops performed by $DFSOut$ and $DFSIn$ for a sample digraph D starting at vertex 1, and 5.21b gives the DFS out-tree and DFS in-tree rooted at 1. Figure 5.22a illustrates the sequence of enqueues, visits, and dequeues performed by $BFSOut$ and $BFSIn$ for the digraph D of Figure 5.21. Figure 5.22b gives the BFS out-tree and BFS in-tree rooted at 1.

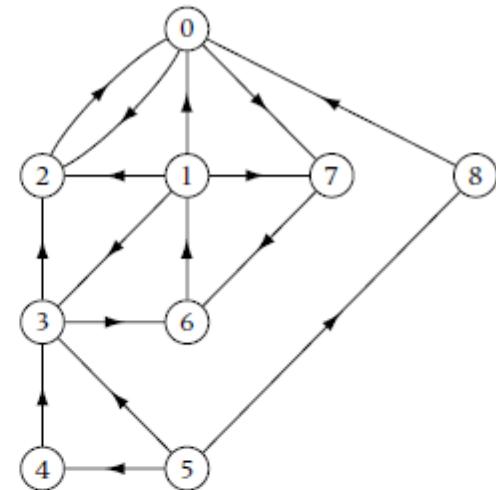
Visit 1
 Push 1 Visit 0
 Push 0 Visit 2
 Pop ($u=0$)
 Push 0 Visit 7
 Push 7 Visit 6
 Pop ($u=7$)
 Pop ($u=0$)
 Pop ($u=1$)
 Push 1 Visit 3
 Pop ($u=3$)
 Pop ($u=1$)
 Stack is now empty.

Visit 1
 Push 1 Visit 6
 Push 6 Visit 3
 Push 3 Visit 4
 Push 4 Visit 5
 Pop ($u=4$)
 Pop ($u=3$)
 Pop ($u=6$)
 Push 6 Visit 7
 Push 7 Visit 0
 Push 0 Visit 2
 Pop ($u=0$)
 Push 0 Visit 8
 Pop ($u=0$)
 Pop ($u=7$)
 Pop ($u=6$)
 Pop ($u=1$)
 Stack is now empty.

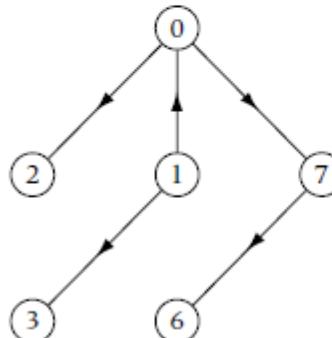
DFSOut starting at 1

DFSIn starting at 1

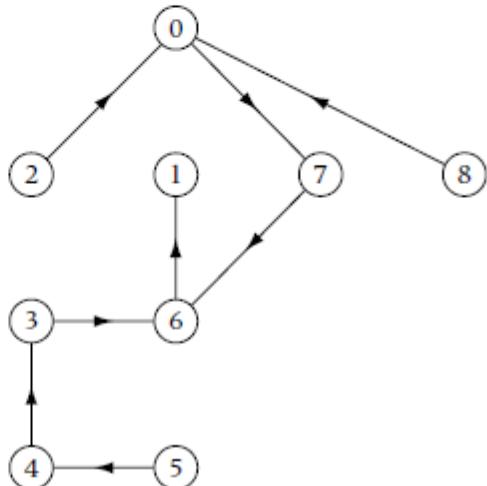
digraph D



(a)



DFS out-tree rooted at 1



DFS in-tree rooted at 1

v	0	1	2	3	4	5	6	7	8
<i>ParentOutTree</i> [v]	1	-1	0	1	-1	-1	7	0	-1
<i>ParentInTree</i> [v]	7	-1	0	6	3	4	1	6	0

(b)

(a) $DFSOut$ and $DFSIn$ with input D and starting vertex $v = 1$ visits vertices in the order 1,0,2,7,6,3 and 1,6,3,4,5,7,0,2,8, respectively; (b) DFS out-tree and in-tree rooted at 1 and their associated parent arrays $ParentOutTree[0:8]$ and $ParentInTree[0:8]$.

Figure 5.21

Enqueue 1 Visit 1
 Dequeue ($u=1$)
 Enqueue 0 Visit 0
 Enqueue 2 Visit 2
 Enqueue 3 Visit 3
 Enqueue 7 Visit 7
 Dequeue ($u=0$)
 Dequeue ($u=2$)
 Dequeue ($u=3$)
 Enqueue 6 Visit 6
 Dequeue ($u=7$)
 Dequeue ($u=6$)
 Queue is now empty.

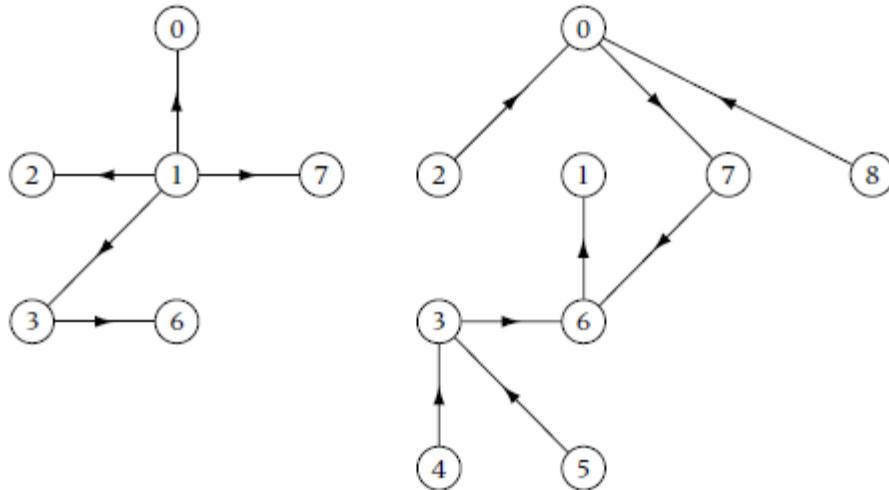
Enqueue 1 Visit 1
 Dequeue ($u=1$)
 Enqueue 6 Visit 6
 Dequeue ($u=6$)
 Enqueue 3 Visit 3
 Enqueue 7 Visit 7
 Dequeue ($u=3$)
 Enqueue 4 Visit 4
 Enqueue 5 Visit 5
 Enqueue 0 Visit 0
 Dequeue ($u=4$)
 Dequeue ($u=5$)
 Dequeue ($u=0$)
 Enqueue 2 Visit 2
 Enqueue 8 Visit 8
 Dequeue ($u=2$)
 Dequeue ($u=8$)
 Queue is now empty.

BFSOut starting at 1

BFSIn starting at 1

digraph D

(a)



BFS out-tree rooted at 1

BFS in-tree rooted at 1

v	0	1	2	3	4	5	6	7	8
$ParentOutTree[v]$	1	-1	1	1	-1	-1	3	1	-1
$ParentInTree[v]$	7	-1	0	6	3	3	1	6	0

(b)

(a) $BFSOut$ and $BFSIn$ with input D and starting vertex $v = 1$ visits vertices in the order 1,0,2,3,7,6 and 1,6,3,7,4,5,0,2,8 respectively; (b) BFS out-tree and in-tree rooted at 1 and their associated parent arrays $ParentOutTree[0:8]$ and $ParentInTree[0:8]$.

Figure 5.22

5.4 Topological Sorting

Suppose that there are n tasks to be performed and that certain tasks must be performed before others. For example, if we are building a house, the task of pouring the foundation must precede the task of laying down the first floor. However, another pair of tasks might not need to be done in a particular order, such as painting the kitchen and painting the bathroom. The problem is to obtain a linear ordering of the tasks in such a way that if task u must be done before task v , then u occurs before v in the linear ordering.

The dependencies of the tasks can be naturally modeled using a directed acyclic graph (*dag*) D —that is, a directed graph without any directed cycles. The vertices of D correspond to the tasks, and a directed edge from u to v is in D if, and only if, task u must precede task v . A *topological sorting* of D is a listing of the vertices such that if uv is an edge of D , then u precedes v in the list. A *topological-sort labeling* of D is a labeling of the vertices in D with the labels $0, \dots, n - 1$ such that for any edge uv in D , the label of u is smaller than the label of v .

A topological-sort labeling is obtained by performing an out-directed depth-first traversal, where we keep track of the order in which vertices become explored. A vertex becomes *explored* when the traversal accesses u having visited all vertices in the out-neighborhood of u (we assume here that a vertex is visited when it is first accessed by the traversal). The following proposition tells us immediately that a topological-sort labeling can be obtained by the reverse order in which the vertices become explored. That is, a vertex u is given topological-sort label $n - i$ if u is the i^{th} vertex to become explored, $i = 1, \dots, n$.

Proposition 5.3.1. Suppose uv is an arc in the dag D . Then v is explored before u in any depth-first out traversal of D .

Proof

We break the proof down into two cases.

Case 1. v is visited before u in the traversal.

In this case, note that there can not be a (directed) path from v to u in D , otherwise D would contain a cycle. Hence, after visiting v , a depth-first search is done and u is not visited in this part of the traversal. In other words, v will become explored before u is visited in the traversal.

Case 2. u is visited before v in the traversal.

In this case, when u is visited, there are unvisited vertices adjacent to u (namely, v), so that u will be pushed on the stack (either the explicit stack in a nonrecursive encoding, or the implicit activation record stack in a recursive encoding), and a depth-first search from u will commence as part of the traversal. Eventually, v will be visited in the depth-first search proceeding from u . Now when any vertex w is visited, if there are no unvisited vertices adjacent to w , then w becomes immediately explored. On the other hand, if there are unvisited vertices adjacent to w , then w is pushed on the stack. Note that when a vertex w is popped from the stack, either w is immediately pushed again if there is an unvisited vertex adjacent to w , or w never is pushed again, that is, w becomes explored. This implies that anything on the stack when a vertex w is visited remains on the stack until w is explored, i.e., all vertices on the stack when w is visited are explored after w . Since u is on the stack when v is visited, it follows that v is again explored before u . Note that in case (2) we did not use the fact that D contains no cycles.



The following Key Fact follows immediately from Proposition 5.3.1, and is the basis of the procedure *Topological* that computes the topological-sort ordering.

Key Fact

Given a dag D , a topological-sort ordering results from the reverse order in which the vertices are explored.

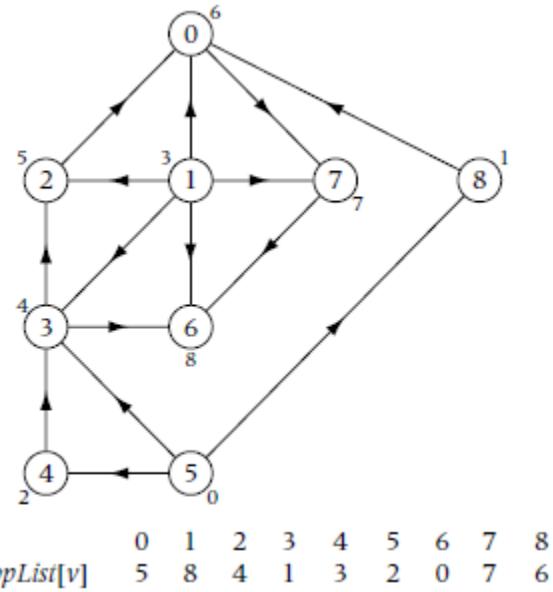
The following procedure *TopologicalSort* computes the topological-sort ordering resulting from the reverse order in which the vertices are explored.

```

procedure: TopologicalSort( $D$ , $TopList[0:n-1]$ )
Input:  $D$  (dag with vertex set  $V = \{0, \dots, n-1\}$  and edge set  $E$ )
Output:  $TopList[0:n-1]$  (array containing a topologically sorted list of the vertices in  $D$ )
     $Mark[0:n-1]$  a 0/1 array initialized to 0s
     $Counter \leftarrow n-1$ 
    for  $v \leftarrow 0$  to do
        if  $Mark[v] \leftarrow 0$  then
             $DFSOutTopLabel(D,v)$ 
        endif
    endfor
    procedure DFSOutTopLabel( $D,v$ ) recursive
         $Mark[v] \leftarrow 1$ 
        for each  $w \in V$  such that  $vw \in E$  do
            if  $Mark[w] = 0$  then
                 $DFSOutTopLabel(D,w)$ 
            endif
        endfor
         $TopList[Counter] \leftarrow v$ 
         $Counter \leftarrow Counter - 1$ 
    end DFSOutTopLabel
end TopologicalSort

```

The array $TopList[0:n-1]$ output by the procedure *TopologicalSort* is illustrated in Figure 5.23 for a sample dag D . In Figure 5.23 we show the position in the list $TopList$ outside each vertex.



A dag D and the array $\text{TopList}[0:8]$ output by TopologicalSort

Figure 5.23

5.6 Closing Remarks

In this chapter we saw how breadth-first search yields shortest paths in (unweighted) graphs and digraphs. The study of shortest path algorithms has a long history, and there are many such algorithms to be found in the literature. In Chapter 6 we will present algorithms for finding shortest paths in weighted graphs and digraphs based on the greedy method and in Chapter 8 we will present algorithms for all-pairs shortest paths based on dynamic programming.

Depth-first search has a host of applications. We saw how a labeling generated by a depth-first traversal yielded a topological sort of a dag. Other vertex labelings generated by depth-first search are the basis of many classical graph algorithms. Depth-first search has a number of useful properties, such as a parenthesis structure associated with the first and last access of nodes (see Exercise 5.31).

In Chapters 11 and 12 we discuss applications of graphs to the Internet and routing and connectivity of networks

Exercises

Section 5.1 Graphs and Digraphs

- 5.1 a) Show that the average degree α of a vertex in a graph G is given by $\alpha = 2m / n$.
b) Show that $\delta \leq 2m/n \leq \Delta$ (where δ and Δ denote the minimum and maximum degrees, respectively, of a vertex in G).
- 5.2 Give an example of each of the following types of graph.
 - a) a 2-regular graph with diameter 8
 - b) a 3-regular graph with diameter 5

- 5.3 Suppose a 6-regular graph G has 20 more edges than vertices. How many vertices and edges does G have?

5.4 Does there exist a 5-regular graph with 44 edges? Explain.

5.5 a) Using Euler's degree formula and Proposition 5.1.5 prove Proposition 4.2.3, which states that the number of interior vertices in a 2-tree is one less than the number of leaf nodes.
 b) Repeat part (a) for k -ary trees.

5.6 Prove that a graph is bipartite if it contains no odd cycle (i.e., cycle of odd length).

5.7 Prove Proposition 5.1.4.

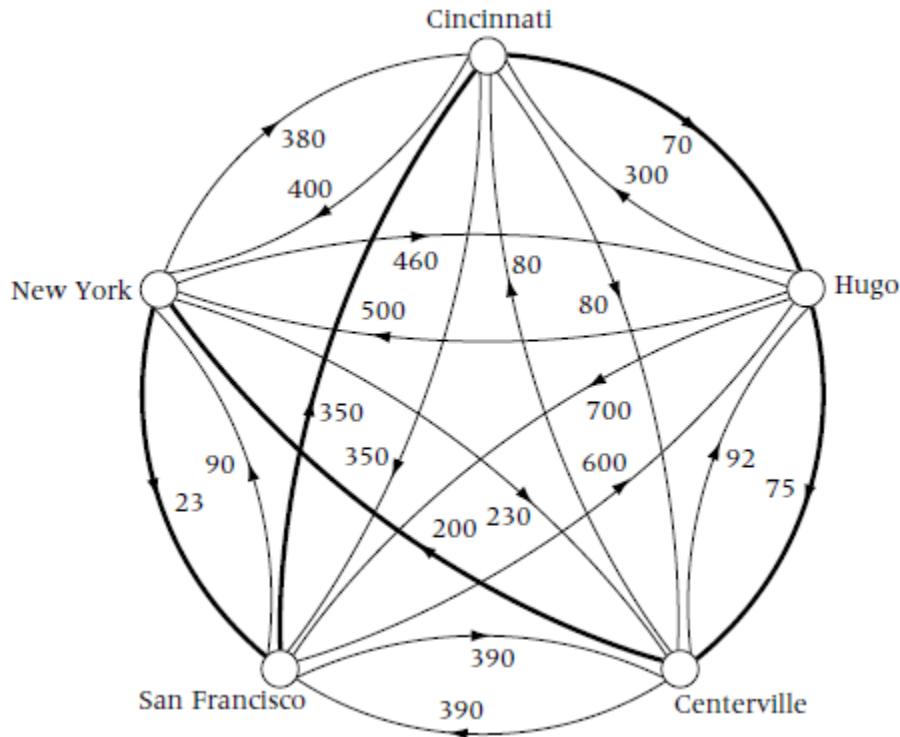
5.8 Prove Proposition 5.1.5.

5.9 Show that for a digraph D having m directed edges, we have

$$\sum_{v \in V} d_{\text{out}}(v) = \sum_{v \in V} d_{\text{in}}(v) = m.$$

5.10 Design an algorithm for finding a proper coloring of a graph G using $\Delta + 1$ colors (where Δ denotes the maximum degree of a vertex).

5.11 Suppose that a salesperson must visit n cities and that there is a cost associated with traveling from one city to another. Starting from a home city, the salesperson wishes to visit each of the other $n - 1$ cities once and return home in such an order that the total cost incurred is a minimum among all such tours (see Figure 5.24). The Traveling Salesman problem (TSP) is to determine such a minimum cost tour.



A minimum length tour of five cities amongst the 24 possible tours

Figure 5.24

The most straightforward algorithm is an *exhaustive search* that enumerates all $(n - 1)!$ tours and keeps track of the shortest tour generated. In this exercise you will design a dynamic programming algorithm for TSP having worst-case complexity in $\Theta(n2^n)$. While this is certainly an improvement over $(n - 1)!$, this exponential complexity is disappointing. However, TSP belongs to a class of problems (*NP-hard*) for which it is believed that no polynomial-time algorithm exists for any problem in the class.

TSP is equivalent to finding a minimum cost Hamiltonian cycle in a weighted digraph D . We may assume without loss of generality that D is the complete digraph $\hat{K}_n = (V, \hat{E})$, where $|V| = n$ and \hat{E} consists of all pairs of distinct vertices (u, v) , $u, v \in V$. Given a weighted digraph $D = (V, E)$, we merely extend the cost weighting c to \hat{K}_n by setting $c(uv) = \infty$ for every (u, v) not in E . TSP for the (undirected) complete graph K_n with cost weighting c is a special case of the Traveling Salesman problem for the digraph \hat{K}_n : For each edge e of K_n assign both of the directed edges e^- and e^+ of \hat{K}_n corresponding to e the weight $c(e)$.

Consider a Hamiltonian cycle H of \hat{K}_n , where without loss of generality we assume that vertex 0 is the initial and terminal vertex of H . Let i denote the first vertex visited by H after leaving vertex 0, and let H_i denote the subpath of H from vertex i to vertex 0, which passes through each vertex of \hat{K}_n once (such a path is called a *Hamiltonian path*). If H is a minimum cost Hamiltonian cycle, then H_i must be a shortest path from i to 0 whose interior vertices consist of the set $V - \{0, i\}$. (A shortest path P joining two vertices i and j is a path that minimizes $Cost(P)$, where $Cost(P)$ is the sum of the costs over all the edges of P .) Now consider any subset U of V . For i a vertex not in U , let $MinCost(i, U)$ denote the cost of a shortest (minimum cost) path P from i to 0 whose interior vertices consist of the set U . Note that since P is a shortest path, P must pass through each vertex of U exactly once. Note also that $MinCost(0, V - \{0\})$ is the minimum cost of a Hamiltonian cycle, that is, the cost of an optimal salesman tour.

- a) Derive the following recurrence relation for $MinCost(i, U)$

$$MinCost(i, U) = \min_{j \in U} \{c(i, j) + MinCost(j, U - \{j\})\},$$

$$\textbf{init.cond. } MinCost(i, \emptyset) = c(i, 0), 1 \leq i \leq n - 1$$

- b) Give pseudocode for an algorithm *TravellingSalesman* based on the recurrence relation in part (a).
- c) Verify that the worst-case complexity of your algorithm *TravellingSalesman* in part (b) is given by $W(n) = (n - 1)2^{n-2} \in \Theta(n2^n)$.
- 5.12 a) Prove that a graph G has an Eulerian path from a to b if, and only if, G is connected, every vertex different from a and b has even degree, and a and b have odd degree.
- b) Prove that a graph G has an Eulerian cycle if, and only if, every vertex has even degree.
- c) State and prove a necessary and sufficient condition for the existence of Eulerian paths and cycles in digraphs.

5.2 Implementing Graphs and Digraphs

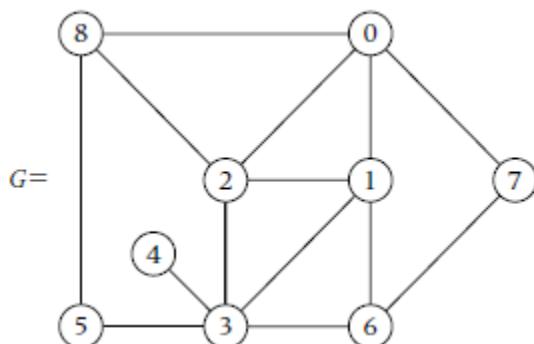
- 5.13 Give pseudocode for the standard ADT operations of adding and deleting an edge for a graph G for each of the following implementations.
- adjacency matrix
 - adjacency lists with array of header nodes
 - adjacency lists with a linked list of header nodes
- 5.14 In the adjacency lists implementation of a graph, where the header nodes belong to a linked list (see Figure 5.13a), it is convenient in practice to maintain a second pointer p in each list node defined as follows: If the list node belongs to the adjacency list of vertex i and corresponds to edge (i,j) , then p points to the header node of adjacency list j . Redo Exercise 5.24c with this enhanced implementation.

5.3 Graphs as Interconnection Networks

- 5.15 Prove by induction that the hypercube H_k of dimension k has
- 2^k vertices
 - $k2^{k-1}$ edges
- 5.16 Consider the hypercube H_k of dimension k on $n = 2^k$ vertices.
- Show that H_k is k -regular.
 - Show that H_k has diameter $k = \log_2 n$.
- 5.17 Give an alternate proof of the result in part (b) of Exercise 5.14 using part (a) of that exercise and Corollary 5.1.2.
- 5.18 Prove by induction that the k -dimensional hypercube H_k has exactly $C(k,j)2^{k-j}$ different sub-hypercubes of dimension j , $j = 0, \dots, k$.
- 5.19 Design an $O(\log n)$ algorithm for summing n numbers on the k -dimensional hypercube H_k , where $n = 2^k$.

Section 5.4 Search and Traversal of Graphs and Digraphs

- 5.20 Consider the following graph.

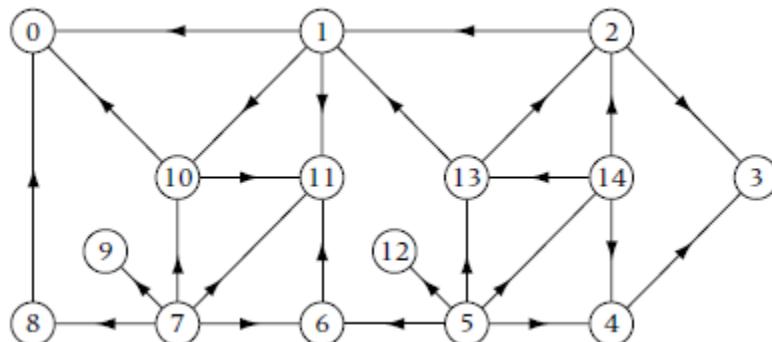


- a) Starting at vertex 3, list the vertices in the order in which they are visited by breadth-first search, and give the array $TreeBFS[0:8]$ that gives a parent implementation of the breadth-first search spanning tree.
- b) Repeat part (a) for depth-first search, giving the $DFSTreeParent[0:8]$.
- 5.21 a) Prove that a depth-first search with starting vertex v of a graph G visits every vertex of G if, and only if, G is connected.

- b) More generally, prove that a depth-first search with starting vertex v of a graph G visits every vertex of the component of G containing v .
- c) Repeat (a) and (b) for breadth-first search.
- 5.22 Give complete pseudocode for *DFS* for the following implementations of a graph G , where the array $\text{Current}[0:n-1]$ is used by the procedure *Next*:
- adjacency matrix
 - adjacency lists with an array of header nodes
 - adjacency lists with a linked list of header nodes
- 5.23 Repeat Exercise 5.28 for *BFS*.
- 5.24 Prove that the breadth-first search out-tree rooted at a vertex v is a *shortest path* out-tree rooted at v in the digraph; that is, it contains a shortest path from v to every vertex accessible from v .
- 5.25 In a depth-first search, show that if we output a left parenthesis when a node is accessed for the first time and a right parenthesis when a node is accessed for the last time (that is, when backtracking from the node), the resulting parenthesization is proper; this is each left parenthesis is properly matched with a corresponding right parenthesis.
- 5.26 Consider an (out-directed) depth-first traversal of a directed graph D . A *back edge* is an edge vu not belonging to the depth-first search forest F , such that u and v belong to the same tree in F and u is an ancestor of v . Show that a directed graph D is a dag if, and only if, there are no back edges.
- 5.27 a) Design an algorithm for determining whether a digraph D is a dag.
 b) Modify your algorithm in (a) to output a directed cycle if D is not a dag.
- 5.28 Design and analyze an algorithm that determines whether a graph is bipartite and identifies the bipartition of the vertices if it is bipartite for the following implementations of the graph.
- adjacency matrix
 - adjacency lists
- 5.29 Design and analyze an algorithm that computes the diameter of a connected graph G .
- 5.30 The *girth* of a graph G is the length of the smallest cycle in G . Design and analyze an algorithm for determining the girth of G .

Section 5.5 Topological Sorting

- 5.31 Show the topological-sort labeling generated by *TopologicalSort* for the following dag.



PART II

MAJOR DESIGN STRATEGIES

6

The Greedy Method

The *greedy method* for solving optimization problems follows the philosophy of greedily maximizing (minimizing) short-term gain and hoping for the best without regard to long-term consequences. Algorithms based on the greedy method are usually very simple, easy to code, and efficient. Unfortunately, just as in real life, in the theory of algorithms the greedy method applied to a problem often leads to less than optimal results. However, there are important problems where the greedy method does yield optimal results, such as computing optimal data compression codes, finding a shortest path between two vertices in a weighted digraph or determining a minimum spanning tree in a weighted graph (the latter two problems will be discussed in Chapter 12). Moreover, even when the greedy method does not yield optimal results, there are important problems in which it yields solutions that in some sense are good approximations to optimal.

6.1 General Description

Because the subject is greed, it seems appropriate to illustrate the greedy method with a problem about money: making change. Suppose that we have just purchased a sleeve of golf balls, and the salesperson wishes to give back (exact) change using the *fewest* number of coins. We assume that there is a sufficient amount of pennies (1¢), nickels (5¢), dimes (10¢), quarters (25¢), and half-dollars (50¢) to make change in any manner whatsoever.

A greedy algorithm for making change uses as many coins of the largest denomination as possible, then uses as many coins of the next largest denomination, and so forth. For example, if the change was 74 cents, the greedy solution uses one half-dollar, two dimes, and four pennies, for a total of seven coins. It is easy to see that seven coins is the fewest number of coins that makes 74 cents change. For any amount of change, this greedy method uses the fewest coins for coins of the above denominations. However, when other denominations of coins are used, this greedy method of making change does not always yield the fewest coins. For example, suppose we use the same denomination coins as before, except that we do not have any nickels. For 30 cents change, the greedy method yields one quarter and five pennies, whereas three dimes does the job.

The coin-changing example serves as a nice illustration of the general type of problem that might be amenable to the greedy method. These problems generally have the goal of maximizing or minimizing an associated *objective function* over all feasible solutions. In most cases this objective function is a real-valued function defined on the subsets of a given base set S . For example, coin changing is a minimization problem, where the base set S is the set of denominations, a feasible solution is a set of coins making correct change, and the objective function is the number of coins used.

The greedy method might be applied to any problem whose solution can be viewed as the result of building up solutions to the problem via a sequence of partial solutions in the following manner. Partial solutions are built up by choosing elements x_1, x_2, \dots from a set R , where R is initially equal to some base set S . Whenever the greedy

method makes a choice x_i , then x_i is either added to the current partial solution or its addition is found to be infeasible and is rejected forever. In either case, x_i is removed from the set R .

In the following general paradigm for the greedy method, choices are made by invoking a suitably defined function $\text{GreedySelect}(R)$. For example, $\text{GreedySelect}(R)$ might be an element that optimizes (minimizes or maximizes) the change in the value of the objective function or some closely related function. When there are several choices that are equivalent, we assume GreedySelect chooses arbitrarily from the equivalent choices.

```

procedure Greedy( $S, Solution$ )
Input:  $S$  (base set) //it is assumed that there is an associated objective
           //function  $f$  defined on (possibly ordered) subsets of  $S$ 
Output:  $Solution$  (an ordered subset of  $S$  that potentially optimizes the
           objective function  $f$ , or a message that  $\text{Greedy}$  doesn't even
           produce a solution (optimal or not))

 $PartialSolution \leftarrow \emptyset$  //initialize the partial solution to be empty
 $R \leftarrow S$ 
while  $PartialSolution$  is not a solution .and.  $R \neq \emptyset$  do
     $x \leftarrow \text{GreedySelect}(R)$ 
     $R \leftarrow R \setminus \{x\}$ 
    if  $PartialSolution \cup \{x\}$  is feasible then
         $PartialSolution \leftarrow PartialSolution \cup \{x\}$ 
    endif
endwhile
if  $PartialSolution$  is a solution then
     $Solution \leftarrow PartialSolution$ 
else
    write("Greedy fails to produce a solution")
endif
end Greedy

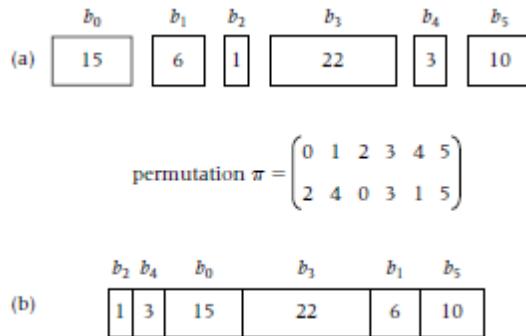
```

In most applications of the greedy method, an associated weighting w of the elements of the base set S is given, and $\text{GreedySelect}(R)$ merely chooses the smallest weight or largest weight element in R . For convenience of discussion, assume that GreedySelect chooses the smallest weight element. The choice of a smallest element is facilitated by preconditioning the base set by sorting with respect to w or by maintaining the elements in the base set in a priority queue where w determines the priority value of the elements.

Even though the greedy method paradigm can be applied in a given situation, and even though it might output a feasible solution it still might not output an *optimal* solution (recall the problem of making 30 cents change without nickels). Thus, proofs of optimality should always accompany solutions generated by greedy algorithms. In spite of its somewhat limited applicability, the greedy method solves a number of classic optimization problems. Moreover, when the greedy method does the job optimally, it generally leads to elegant and efficient code for solving the problem.

6.2 Optimal Sequential Storage Order

Suppose we are given a set of objects b_0, b_1, \dots, b_{n-1} arranged sequentially in some order $b_{\pi(0)}, b_{\pi(1)}, \dots, b_{\pi(n-1)}$, where π is some permutation of $\{0, 1, \dots, n - 1\}$, such as files stored on a sequential access tape. We assume that there is a weight c_i associated with object b_i (for example, the length of a file) that represents the individual cost of processing b_i . We also assume that to process object $b_{\pi(i)}$, we must first process the objects $b_{\pi(0)}, b_{\pi(1)}, \dots, b_{\pi(i-1)}$ that precede $b_{\pi(i)}$ in the sequential arrangement. For example, to read the i^{th} file on a sequential tape (assuming that the tape is rewound to the beginning), we must first read the preceding $i - 1$ files on the tape. Thus the total cost of processing $b_{\pi(i)}$ is $c_{\pi(0)} + c_{\pi(1)} + \dots + c_{\pi(i)}$. In Figure 6.1, we illustrate this example with six files b_0, b_1, \dots, b_5 of lengths 15, 6, 1, 22, 3, 10, respectively. A sample storage order of these files on a tape is given in Figure 6.1b. If we wish to read the file of length 22, we first have to read the files of length 1, 3, 15 that precede this file on the tape. Thus, reading the file of length 22 results in reading files whose total length is $1 + 3 + 15 + 22 = 41$.



(a) Files of lengths 15, 6, 1, 22, 3, 10; (b) a sample storage order of files on a sequential access tape corresponding to the permutation π

Figure 6.1

The *optimal sequential storage* problem for processing objects b_0, b_1, \dots, b_{n-1} is to find an arrangement (permutation) of the objects $b_{\pi(0)}, b_{\pi(1)}, \dots, b_{\pi(n-1)}$ that minimizes the average time it takes to process an object. Let p_i denote the probability that the object b_i is to be processed, $i = 0, \dots, n - 1$. Given these probabilities and a permutation π , clearly the average total cost of processing an object is given by

$$\text{AveCost}(\pi) = p_{\pi(0)}(c_{\pi(0)}) + p_{\pi(1)}(c_{\pi(0)} + c_{\pi(1)}) + \dots + p_{\pi(n-1)}(c_{\pi(0)} + \dots + c_{\pi(n-1)}). \quad (6.2.1)$$

We first consider the special case where each object is equally likely to be processed, so that $p_i = 1/n$, $i = 1, \dots, n$. Then (6.2.1) becomes

$$\begin{aligned} \text{AveCost}(\pi) &= \frac{1}{n}[(c_{\pi(0)}) + (c_{\pi(0)} + c_{\pi(1)}) + \dots + (c_{\pi(0)} + c_{\pi(1)} + \dots + c_{\pi(n-1)})] \\ &= \frac{1}{n} \sum_{i=0}^{n-1} (n-i)c_{\pi(i)}. \end{aligned} \quad (6.2.2)$$

Looking at (6.2.2), we see that the coefficients of $c_{\pi(i)}$, $i = 0, \dots, n - 1$, decrease as i increases. Thus, it is intuitively obvious that $\text{AveCost}(\pi)$ is minimized when $c_{\pi(0)} \leq c_{\pi(1)} \leq \dots \leq c_{\pi(n-1)}$, so that the greedy strategy we use is to select the objects in increasing order of their costs.

Now consider the problem of minimizing $\text{AveCost}(\pi)$ as given by (6.2.1) for a general set of probabilities p_i , $i = 1, \dots, n$. Various greedy strategies come to mind. For example, we might use the same greedy strategy given previously, which give the optimal solution when all the probabilities are equal. However, it is not hard to find examples where this strategy yields suboptimal results. We might also arrange the objects in decreasing order of probabilities (placing the objects most likely to be processed near the beginning), but, again, it is not hard to find examples where this strategy fails. Alas, is there a greedy strategy that works? Yes, we leave it as an exercise to show that arranging the objects in increasing order of the ratios c_i/p_i yields an optimal solution.

6.3 The Knapsack Problem

The *Knapsack problem* involves n objects b_0, b_1, \dots, b_{n-1} and a knapsack of capacity C . We assume that each object b_i has a given positive weight (or volume) w_i , and a given positive value v_i (or profit, cost, and so forth) $i = 0, \dots, n - 1$. We place the objects or fractions of objects in the knapsack, taking care that the capacity of the knapsack is not exceeded. If a fraction f_i of object b_i is placed in the knapsack ($0 \leq f_i \leq 1$), then it contributes $f_i v_i$ to the total value in the knapsack. The Knapsack problem is to place objects or fractions of objects in the knapsack, without exceeding the capacity, so that total value of the objects in the knapsack is maximized. A more formal statement of the Knapsack problem is

$$\begin{aligned} & \text{maximize} \quad \sum_{i=0}^{n-1} f_i v_i, \\ & \text{subject to the constraints : } \sum_{i=0}^{n-1} f_i w_i \leq C, \\ & \quad 0 \leq f_i \leq 1, \quad i = 0, \dots, n-1. \end{aligned} \tag{6.3.1}$$

As with the sequential storage order problem, several possible greedy methods for solving the Knapsack problem come to mind. For example, we might put as much as possible of the object with highest value into the knapsack, then as much as possible of the object of second-highest value into the knapsack, and so forth, until the knapsack is filled to capacity. Another greedy strategy would be to follow this same scheme, but where the objects are placed in the knapsack according to decreasing weights. But we can easily find examples where neither of these strategies yields optimal solutions. We will show that the strategy of putting the objects into the knapsack according to decreasing *ratios (densities)* v_i/w_i until the knapsack is full yields an optimal solution.

The following interpretation of the Knapsack problem makes it clear that the ratio v_i/w_i is the critical issue. Suppose the (greedy) owner of a gourmet coffee shop wishes to mix a 10-pound bag of coffee using various types of coffee beans in such a way as to produce the coffee blend of maximum cost. The weights of the objects in the Knapsack problem correspond to the quantity in pounds that are available of each type of coffee

bean. The value of each quantity of coffee bean is the total cost of that quantity in dollars. In Figure 6.2 we list the types and amounts of coffee beans that are available.

Type	Quantity Available In Pounds	Total Cost In Dollars for That Quantity
Colombian	8	32
Jamaican	2	32
Java	4	25
Bicentennial	1	10
Mountain	2	9
Roast	6	18
Dark	10	55
Special	50	100

Coffee shop inventory

Figure 6.2

Intuitively, we obtain the most expensive 10-pound bag of coffee if we first use as much as possible of the bean whose cost/pound ratio is the largest, then as much as possible of the bean whose cost/pound ratio is the second-largest, and so on until we reach 10 pounds. Figure 6.3 lists the coffee beans in decreasing order of their cost/pound ratios and the quantity of each type of bean used as determined by this greedy method.

Type	Cost/Pound	Fraction of Available Quantity Chosen	Quantity Chosen
Jamaican	16	1	2
Bicentennial	10	1	1
Java	6.25	1	4
Dark	5.5	0.3	3
Mountain	4.5	0	0
Colombian	4	0	0
Roast	3	0	0
Special	2	0	0

Greedy solution for 10-pound bag of coffee

Figure 6.3

Generalizing the greedy strategy for making expensive coffee leads to a greedy algorithm for the Knapsack problem. The algorithm *Knapsack* assumes as a precondition that the objects have been sorted in decreasing order of the ratios v_i/w_i , $i = 0, \dots, n - 1$.

Procedure *Knapsack*($V[0:n - 1]$, $W[0:n - 1]$, C , $F[0:n - 1]$)

Input: $V[0:n - 1]$ (an array of positive values)

$W[0:n - 1]$ (an array of positive weights)

C (a positive capacity)

Output: $F[0:n - 1]$ (array of nonnegative fractions)

Sort the arrays $V[0:n - 1]$ and $W[0:n - 1]$ in decreasing order of densities, so that

$$V[0]/W[0] \geq V[1]/W[1] \geq \dots \geq V[n - 1]/W[n - 1]$$

for $i \leftarrow 0$ **to** $n - 1$ **do**

$$F[i] \leftarrow 0$$

```

endfor
RemainCap =  $C$ 
i  $\leftarrow$  0
if  $W[0] \leq C$  then
    Fits  $\leftarrow$  .true.
else
    Fits  $\leftarrow$  .false.
endif
while Fits .and.  $i \leq n - 1$  do
     $F[i] \leftarrow 1$ 
    RemainCap  $\leftarrow$  RemainCap -  $W[i]$ 
    i  $\leftarrow$  i + 1
    if  $W[i] \leq RemainCap$  then
        Fits  $\leftarrow$  .true.
    else
        Fits  $\leftarrow$  .false.
    endif
endwhile
if  $i \leq n - 1$  then  $F[i] \leftarrow RemainCap/W[i]$  endif
end Knapsack

```

Clearly, procedure *Knapsack* generates an optimal solution when $\sum_{i=0}^{n-1} w_i \leq C$, since all the objects are then placed in the knapsack. Using a proof by contradiction, we now show that *Knapsack* generates an optimal solution when $\sum_{i=0}^{n-1} w_i > C$. Suppose to the contrary that the solution $F = (f_0, f_1, \dots, f_{n-1})$ generated by *Knapsack* is suboptimal. Now consider an optimal solution $Y = (y_0, y_1, \dots, y_{n-1})$, and let j denote the first index i such that $f_i \neq y_i$ so that $1 = f_j = y_j$, $0 \leq i \leq j - 1$. We choose Y such that j is maximized over all optimal solutions. Note that $\sum_{i=0}^{n-1} y_i w_i = C$ (otherwise we could increase one of the y_i 's and trivially increase the total value of the solution). Because of the greedy strategy used by *Knapsack*, we have $f_j > y_j$.

By altering Y , we now construct an optimal solution $Z = (z_1, z_2, \dots, z_n)$ that agrees with F in one more initial position. We set $z_i = f_i$, $1 \leq i \leq j$, and for $i > j$ the value of z_i is obtained by suitably decreasing y_i so that $\sum_{i=0}^{n-1} z_i w_i = C$. It follows that:

$$(z_j - y_j)w_j = \sum_{i=j+1}^n (y_i - z_i)w_i. \quad (6.3.2)$$

Comparing the total values of the solutions Z and Y , we obtain

$$\begin{aligned}
\sum_{i=0}^{n-1} z_i v_i - \sum_{i=0}^{n-1} y_i v_i &= (z_j - y_j)v_j - \sum_{i=j+1}^{n-1} (y_i - z_i)v_i \\
&= (z_j - y_j)w_j v_j / w_j - \sum_{i=j+1}^{n-1} (y_i - z_i)w_i v_i / w_i \\
&\geq \left[(z_j - y_j)w_j - \sum_{i=j+1}^{n-1} (y_i - z_i)w_i \right] v_j / w_j \quad (\text{since } y_i \geq z_i \\
&\quad \text{and } v_0 / w_0 \geq v_1 / w_1 \geq \dots \geq v_{n-1} / w_{n-1}) \\
&= 0 \quad (\text{since the number inside the square brackets is 0 by (7.3.2)}).
\end{aligned}$$

Thus, the value of Z is not less than the value of Y , so that Z is also an optimal solution. But Z agrees with F in the first j initial positions, which contradicts the assumption that Y is an optimal solution that agrees with F in the most initial positions.

Except for the $\Theta(n \log n)$ sorting step, the remainder of *Knapsack* has linear complexity in the worst case.

A natural variation of the Knapsack problem, called the *0/1 Knapsack problem*, is the same as the Knapsack problem except that fractional objects are not allowed. More precisely, the 0/1 Knapsack problem has the following formulation:

$$\begin{aligned}
&\text{maximize} \sum_{i=0}^{n-1} f_i v_i \\
&\text{subject to the constraints : } \sum_{i=0}^{n-1} f_i w_i \leq C \\
&\quad f_i \in \{0,1\}, \quad i = 0, \dots, n-1.
\end{aligned} \tag{6.3.3}$$

The greedy strategy of placing the objects in the knapsack in decreasing order of the ratios v_i/w_i , $i = 0, \dots, n-1$, rejecting objects when their addition would overflow the knapsack, may yield a suboptimal solution to the 0/1 Knapsack problem. While the Knapsack problem is easily solved, the 0/1 Knapsack problem is NP-hard. Note that we took the input size for the Knapsack problems to be the number of objects n . This measure does not take into account the sizes of the weights and values of the objects. Assuming the capacity, weights and values are positive integers, a more precise measure of the input size (in binary) would be the sum of the number of digits of each of these integers; that is

$$b = \lceil \log_2 C \rceil + \sum_{i=0}^{n-1} \lceil \log_2 (w_i + 1) \rceil + \sum_{i=0}^{n-1} \lceil \log_2 (v_i + 1) \rceil.$$

We chose n as the input size because it is simpler to analyze the algorithm in terms of n . Also, since $b \geq n$, any upper bound formula for the complexity of the algorithm in terms of n applies when n is replaced with b . Sometimes it is useful to measure the input size in unary (base 1); for example, assuming the capacity, weights and values are positive integers, the input size to the Knapsack problems is measured in terms of $C + \sum_{i=0}^{n-1} w_i + \sum_{i=0}^{n-1} v_i$. For many NP-complete problems, such as the coin changing and 0/1 Knapsack problem, the problem as measured by unary inputs has polynomial

complexity. In exercises 8.x and 8.y of Chapter 8 polynomial-time dynamic programming solutions to coin changing and 0/1 Knapsack problem are developed when the input numbers are measured as unary integers.

6.4 Huffman Codes

With the emergence of the Internet, the need for data compression has become increasingly important. Transporting extremely large files across the internet is commonplace today, and these files must be compressed in order to save time and storage resources. For example, there are standard data compression algorithms such as jpeg or gif that are used to compress image files (such as photographs). There are also standard data compression algorithms for text and program files, such as those used to zip and unzip files. Even before the advent of the Internet, the need for compressing large text files was necessary in order to minimize storage requirements. A classical data compression algorithm, due to Huffman in a 1952 paper, is based on a greedy strategy for constructing a code for a given alphabet of symbols. Not only are Huffman codes useful for compressing text files, they also are used as part of image or audio file compression techniques.

For a given *alphabet* of symbols $A = \{a_0, a_1, \dots, a_{n-1}\}$, one approach to the problem of data compression involves encoding each symbol in the alphabet with a binary string. The alphabet A might be letters and punctuation symbols from the English language, the standard symbol set used by personal computers, Discrete Cosine Transform (*DCT*) coefficients in a jpeg compression of an image file, and so forth. We refer to the set of binary strings encoding the symbols in A as a *binary code* for A . Consider, for example, the *ASCII* encoding of the alphabet A of standard symbols used by computers consisting of alphabetical characters, numeric characters, punctuation characters, control characters, and so forth. Each symbol of this alphabet is represented by an 8-bit (1-byte) binary string. In this example, since each symbol is encoded with a fixed length binary string, there is basically no compression being used. Using a fixed length binary string to store symbols is very convenient for computer implementation, but has the disadvantage that it does not optimize storage usage. By allowing variable length binary strings for binary codes, we can save space when storing files (text, jpeg, mpeg, etc.) made up of symbols from our given alphabet.

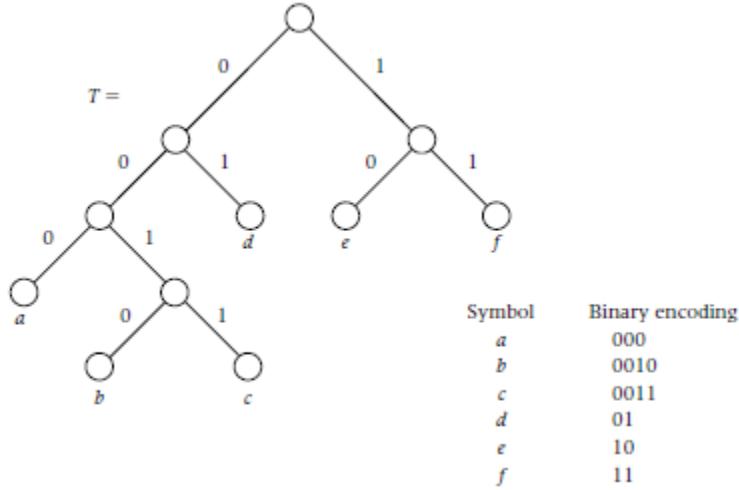
Given a binary code for A , we obtain a binary encoding of any string consisting of symbols from A . We simply replace each symbol in the string with its binary encoding. For example, suppose $A = \{a, b, c\}$ and symbols a, b, c are encoded as 1, 00, 01, respectively. Then the binary encoding for the string *cabbc* is 01100001.

To illustrate ambiguous codes, suppose a, b, c are encoded as 0, 1, 01, respectively. Then the binary string 011 can be decoded as either *abb* or *cb*. Note that the binary string 0 is a prefix for the binary string 01. It is easily verified that a binary code allows for unambiguous (unique) decoding if, and only if, no binary string in the code is a prefix of any other binary string in the code.

6.4.1 Binary Prefix Codes

Binary codes having the property that no binary string in the code is a prefix of any other binary string in the code are called *prefix codes*. We can readily create a prefix

code by utilizing a 2-tree T , which is a binary tree where every non-leaf node has exactly two children. The leaf nodes of T correspond to the n symbols in A . A high-level description of the process is as follows. Label each edge corresponding to a left child with a zero and each edge corresponding to a right child with a one. Assign to each symbol a_i the binary string obtained by reading the labels on the edges when following a path from the root to the leaf node corresponding to a_i . The binary strings generated by a sample tree for the alphabet $A = \{a,b,c,d,e,f\}$ are shown in Figure 6.4.



Encoding of alphabet $A = \{a,b,c,d,e,f\}$ generated by a sample 2-tree T

Figure 6.4

For each node N of the 2-tree T , let $B(N)$ denote the binary string corresponding to following a path from the root to N in the manner described earlier. Let L and R denote the left and right children of N , respectively. Note that $B(L)$ and $B(R)$ can be computed from $B(N)$ by concatenating ‘0’ and ‘1’, respectively, to the end of $B(N)$; that is, $B(L) = B(N) + ‘0’$ and $B(R) = B(N) + ‘1’$ (where $+$ denotes the operation of concatenation). We now give pseudocode for the algorithm *GenerateCode*, which utilizes this formula to compute the array *Code*, where *Code*[i] is the binary code for symbol a_i , $i = 1, \dots, n$.

GenerateCode computes $B(N)$ for each node by performing a preorder traversal of the 2-tree, where the generalized visit operation at node N involves computing $B(L)$ and $B(R)$ from $B(N)$ using the formula described above. We assume that the 2-tree T is implemented using pointers and dynamic variables, where each node contains two pointer fields corresponding to the left and right children and two information fields *BinaryCode* and *SymbolIndex*. Before calling *GenerateCode*, we assume that the value of *SymbolIndex* for each leaf node has been initialized to contain the index i of the alphabet symbol a_i corresponding to the leaf node (*SymbolIndex* is left undefined for the internal nodes of T). The computed value $B(N)$ for each node N is stored in the variable *BinaryCode*. The root node’s value of *BinaryCode* is initialized to the null string before calling *GenerateCode*.

```

procedure GenerateCode(Root,Code[0:n – 1]) recursive
Input: Root (a pointer to root of 2-tree T) //Root→BinaryCode is //initialized to the
           null string. The leaf node corresponding to  $a_i$  //has its SymbolIndex field
           initialized to i,  $i = 0, \dots, n - 1$ 
Output: Code[0:n – 1] (array of binary strings, where Code[i] is the code for symbol
            $a_i$ )
if Root→LeftChild = null then //a leaf is reached
    Code[Root→SymbolIndex] ← Root→BinaryCode
else
    (Root→LeftChild)→BinaryCode ← Root→BinaryCode + ‘0’
    (Root→RightChild)→BinaryCode ← Root→BinaryCode + ‘1’
    GenerateCode(Root→LeftChild,Code)
    GenerateCode(Root→RightChild,Code)
endif
end GenerateCode

```

The table *Code*[0:*n* – 1] of binary codes can be used to encode text consisting of strings of symbols from the alphabet *A*. Going the other direction, encoded text can be decoded using the 2-tree *T* in the obvious way.

6.4.2 The Huffman Algorithm

Now suppose we are encoding text composed of symbols from *A*, where the frequency of the occurrence of symbol a_i in such text is given by f_i , $i = 0, \dots, n - 1$. An important problem in data compression is to find an *optimal* binary prefix code, that is, a binary prefix code that minimizes the expected length of a *coded symbol*. Clearly, an optimal binary prefix code also minimizes the expected length of text generated using the symbols from *A*. The problem of finding optimal binary prefix codes is equivalent to the following problem for 2-trees. Each unambiguous binary code can be generated by traversing a suitable 2-tree *T*. We let λ_i denote the length of the path in *T* from the root to (the leaf node corresponding to) a_i , $i = 0, \dots, n - 1$. The expected length of the coded symbols determined by *T* is closely related to a generalization of the leaf path length called the *weighted leaf path length* *WLPL*(*T*) of *T*, defined by

$$WLPL(T) = \sum_{i=0}^{n-1} \lambda_i f_i. \quad (6.4.1)$$

Note that *WLPL*(*T*) becomes *LPL*(*T*) when all the f_i 's equal 1. For example, given frequencies 9,8,5,3,15,2 for the symbols *a,b,c,d,e,f*, respectively, the weighted leaf path length for the binary tree *T* in Figure 6.4 is given by

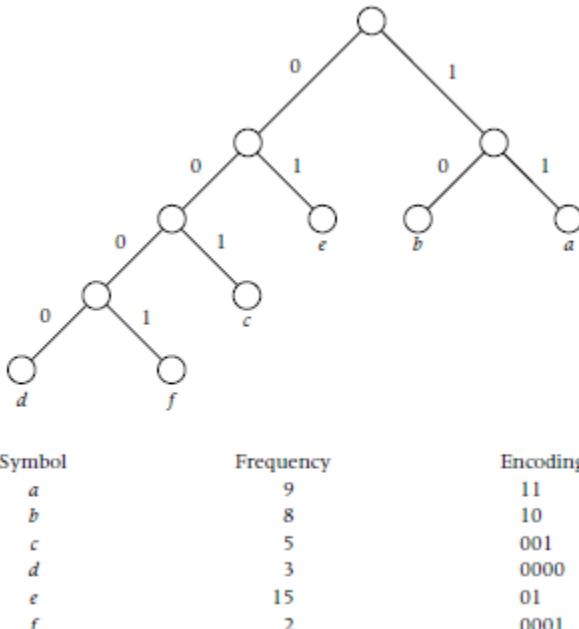
$$WLPL(T) = (3)(9) + (4)(8) + (4)(5) + (2)(3) + (2)(15) + (2)(2) = 119. \quad (6.4.2)$$

Since symbol a_i occurs with frequency f_i the probability p_i , $i = 0, \dots, n - 1$ that symbol a_i occurs in a given text position is

$$p_i = \frac{f_i}{\sum_{j=0}^{n-1} f_j}, \quad i = 0, \dots, n - 1.$$

Each binary digit in the binary string encoding a_i corresponds to an edge of the path in T from the root node to the leaf node containing a_i . Hence, the length of the binary string encoding a_i (in the binary code determined by T) equals λ_i , $i = 0, \dots, n - 1$, so that the expected length of a coded symbol is given by $WLPL(T) / (\sum_{i=0}^{n-1} f_i)$. Thus, the problem of finding an optimal binary prefix code is equivalent to finding a 2-tree having minimum $WLPL(T)$.

Given the set of frequencies 9, 8, 5, 3, 15, 2 for the symbols a, b, c, d, e, f , respectively, the 2-tree in Figure 6.4, which has weighted leaf path length equal to 119, is not optimal. The binary tree given in Figure 6.5, which has weighted leaf path length equal to 99, turns out to be an optimal 2-tree with respect to these frequencies.



A tree having minimum weighted leaf path length for alphabet $A = \{a, b, c, d, e, f\}$ with respect to given frequencies

Figure 6.5

We now describe Huffman's greedy algorithm for computing an optimal binary prefix code with respect to a given set of frequencies f_i , $i = 0, \dots, n - 1$. The procedure *HuffmanCode* computes a 2-tree T minimizing $WLPL(T)$ by constructing a sequence of forests as follows. At each stage in the algorithm, the root of each tree in the forest is assigned a frequency. The initial forest F consists of the n single node trees corresponding to the n elements of A . At each stage *HuffmanCode* finds two trees T_1 and T_2 whose roots R_1 and R_2 have the smallest and second-smallest frequencies over all the trees in the current forest. A new (internal) node R is then added to the forest, together with two edges joining R to R_1 and R to R_2 , so that a new tree is created with R as the root and T_1 and T_2 as the left and right subtrees of R . The frequency of the new root vertex R

is taken to be the sum of the frequencies of the old root vertices R_1 and R_2 . The Huffman tree is constructed after $n - 1$ stages, involving the addition of $n - 1$ internal nodes.

The procedure *HuffmanCode* calls the procedure *GenerateCode* described earlier. We utilize high-level parameter descriptions in our calls to procedures implementing the various priority queue operations. For example, a call to *CreatePriorityQueue* with parameters $Leaf[0:n - 1]$ and Q creates a priority queue Q of pointers to the leaf nodes. A call to *RemovePriorityQueue* with parameters Q and L removes the element at the front of the queue Q and assigns it to L .

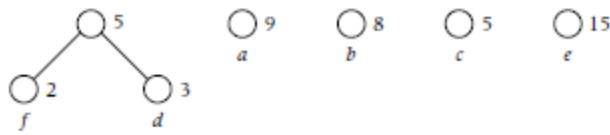
```
Procedure HuffmanCode( $A[0:n - 1]$ , $Freq[0:n - 1]$ , $Code[0:n - 1]$ )
Input:  $Freq[0:n - 1]$  (an array of nonnegative frequencies:  $Freq[i] = f_i$ )
Output:  $Code[0:n - 1]$  (an array of binary strings for Huffman code:  $Code[i]$  is the
           binary string encoding symbol  $a_i$ ,  $i = 0, \dots, n - 1$ )
for  $i \leftarrow 0$  to  $n - 1$  do //initialize leaf nodes
    AllocateHuffmanNode( $P$ )
     $P \rightarrow SymbolIndex \leftarrow i$ 
     $P \rightarrow Frequency \leftarrow Freq[i]$ 
     $P \rightarrow LeftChild \leftarrow null$ 
     $P \rightarrow RightChild \leftarrow null$ 
     $Leaf[i] \leftarrow P$ 
endfor
CreatePriorityQueue( $Leaf[0:n - 1]$ , $Q$ ) //create priority queue of
           //pointers to leaf nodes with Frequency as the key
for  $i \leftarrow 1$  to  $n - 1$  do
    RemovePriorityQueue( $Q,L$ ) // $L,R$  point to root nodes of //smallest and
    RemovePriorityQueue( $Q,R$ ) //second-smallest frequency, //respectively
    AllocateHuffmanNode( $Root$ )
     $Root \rightarrow LeftChild \leftarrow L$ 
     $Root \rightarrow RightChild \leftarrow R$ 
     $Root \rightarrow Frequency \leftarrow (L \rightarrow Frequency) + (R \rightarrow Frequency)$ 
    InsertPriorityQueue( $Q,Root$ )
endfor
 $Root \rightarrow BinaryString \leftarrow ''$  //BinaryString of root initialized to null //string
GenerateCode( $Root,Code[0:n - 1]$ )
end HuffmanCode
```

The action of *HuffmanCode* is illustrated in Figure 6.6 for the same sample alphabet and set of frequencies as in Figure 6.5.

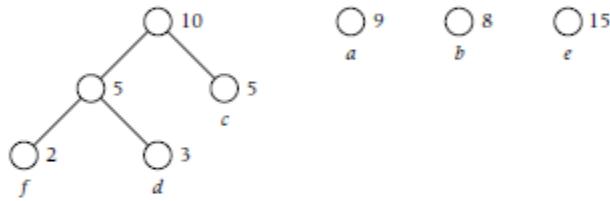
Initial forest



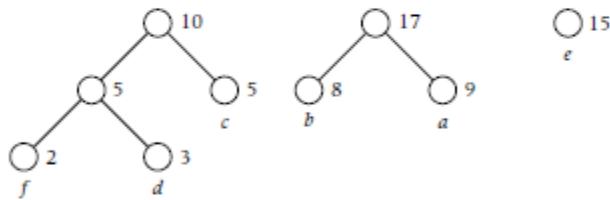
Stage 1



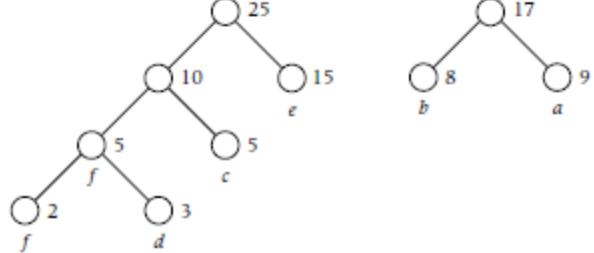
Stage 2



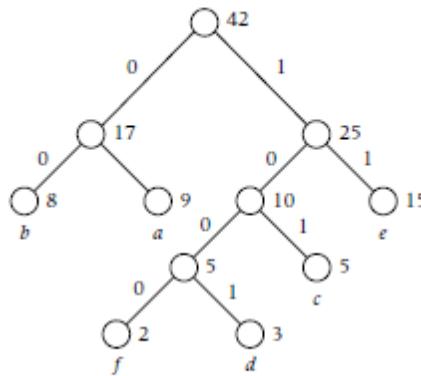
Stage 3



Stage 4



Stage 5: Huffman tree



Symbol	Frequency	Encoding
a	9	01
b	8	00
c	5	101
d	3	1001
e	15	11
f	2	1000

$$WLPL(T) = (2)(9) + (2)(8) + (3)(5) + (4)(3) + (2)(15) + (4)(2) = 99$$

Action of *HuffmanCode* for the alphabet $A = \{a, b, c, d, e, f\}$ with the same set of frequencies as in Figure 6.5

Figure 6.6

An efficient way to implement the priority queue Q is to use a min-heap (see Chapter 4). Then *CreatePriorityQueue* (*CreateMinHeap*), *RemovePriorityQueue* (*RemoveMinHeap*), and *InsertPriorityQueue* (*InsertMinHeap*) have worst-case complexities belonging to $O(n)$, $O(\log n)$, and $O(\log n)$, respectively. Hence, the worst-case complexity of *HuffmanCode* belongs to $O(n \log n)$.

The correctness of the algorithm *HuffmanCode* depends on the following proposition, whose proof is left to the exercises.

Proposition 6.4.1

Given a set of frequencies f_i , $i = 0, \dots, n - 1$, the Huffman tree T constructed by *HuffmanCode* has minimal weighted $WLPL(T)$ over all 2-trees whose edges are weighted by these frequencies. \square

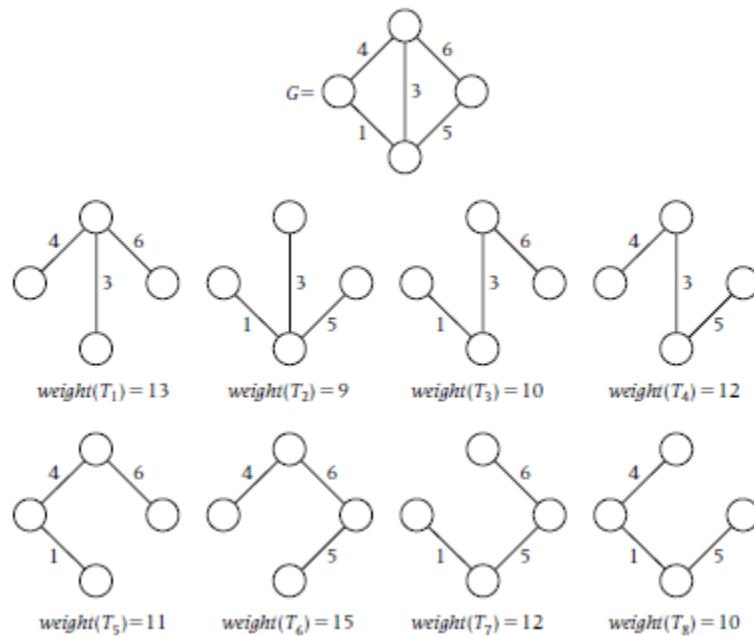
Finding a binary tree of minimum weighted leaf path length has other important applications besides Huffman codes. For example, the problem of finding optimal merge patterns is solved by constructing a binary tree having minimum weighted leaf path length, where the leaf nodes of the tree correspond to files or sublists and the weight of a leaf is the length of its associated file or sublist.

6.5 Minimum Spanning Tree

The problem of finding a minimum spanning tree in a weighted graph is particularly important in network applications. For example, the problem arises when designing any physical network connecting n nodes, where the connections between nodes are subject to feasibility and weight constraints. Examples of such physical networks include communication networks, transportation networks, energy pipelines, VLSI chips, and so forth. In all these examples, the weight of a minimum spanning tree provides a lower bound on the cost of building the network.

Given a spanning tree T in a graph G with a weighting w of the edges E of G , the *weight* of T , denoted $\text{weight}(T)$, is the sum of the w -weights of its edges. If T has minimum weight over all spanning trees of G , then we call T a *minimum spanning tree*.

In Figure 6.7, all the spanning trees of a weighted graph on four vertices are enumerated. The minimum spanning tree is then obtained by inspection.



Enumeration of the spanning trees of G . Minimum spanning tree is T_2 .

Figure 6.7

The number of spanning trees, even for relatively small graphs, is usually enormous, making an enumerative brute-force search infeasible. Indeed, Cayley proved that the complete graph K_n on n vertices contains n^{n-2} spanning trees!

Fortunately, efficient algorithms based on the greedy method do exist for finding minimum spanning trees. We discuss two of the more famous ones, namely, Kruskal's and Prim's algorithms. These algorithms are similar in the sense that their selection functions always choose an edge of minimum weight among the remaining edges. However, Prim's algorithm selection function only considers edges incident to vertices already included in the tree. In particular, the partial solutions built by Prim's algorithm are trees, whereas the partial solutions built by Kruskal's algorithm are forests.

6.5.1 Kruskal's Algorithm

Given a connected graph $G = (V, E)$, with $|V| = n$ and $|E| = m$, and a weight function w defined on the edge set E , Kruskal's algorithm finds a minimum spanning tree T of G by constructing a sequence of n forests

F_0, F_1, \dots, F_{n-1} , where F_0 is the empty forest and F_i is obtained from F_{i-1} by adding a single edge e . The edge e is chosen so that it has minimum weight among all the edges not belonging to F_{i-1} and doesn't form a cycle when added to F_{i-1} . Kruskal's algorithm is illustrated for a sample graph in Figure 6.8.

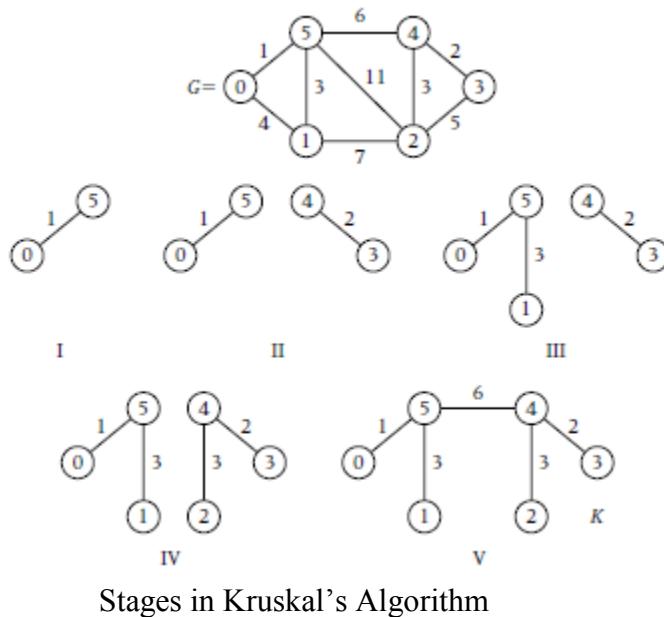


Figure 6.8

Remark

For convenience, we assume that the input graph G to Kruskal's algorithm is connected. However, Kruskal's algorithm is easily modified to accept *any* graph G (connected or disconnected) and to output a minimum (weight) forest, each of whose trees spans a (connected) component of G .

Using a proof by contradiction, we now show that the spanning tree K generated by Kruskal's algorithm is a minimum spanning tree. For convenience, we will assume that the edge weights are distinct (the proof can be slightly modified to hold for nondistinct edge weights, see Exercise 12.4). Let the edges of K be denoted by x_1, x_2, \dots, x_{n-1} , listed in increasing order of their weights. Assume that K is not a minimum spanning tree, and let T be a minimum spanning tree with edges y_1, y_2, \dots, y_{n-1} , listed in increasing order of their weights. Let j denote the first index i such that $x_i \neq y_i$, so that $x_i = y_i$, $1 \leq i \leq j-1$. Since x_j was chosen by the greedy strategy, we have $w(x_j) < w(y_j)$.

We now construct a spanning tree T' whose weight is smaller than T . Consider the subgraph H obtained by adding the edge x_j to T . It is easily verified that H contains a unique cycle C and that this cycle contains x_j . Since $y_i = x_i$, $i = 1, 2, \dots, j-1$, and the tree K does not contain C , it follows that C must contain the edge y_k for some index $k \geq j$. Let T'

be the spanning tree obtained from H by deleting the edge y_k . Since $w(y_k) \geq w(y_j) > w(x_j)$, we have

$$weight(T') = weight(T) + w(x_j) - w(y_k) < weight(T),$$

contradicting our assumption that T is a minimum spanning tree.

The key implementation detail in Kruskal's algorithm is the detection of whether the addition of a given edge e to the existing forest F creates a cycle. Observe that a cycle is formed by the edge $e = uv$ if, and only if, u and v belong to the same component (tree) in F . Checking whether u and v belong to the same component can be done efficiently using the union and find algorithms for disjoint sets discussed in Chapter 4, where the collection of disjoint sets in the current context is the collection of vertex sets of each tree in forest F .

Key Fact

When implementing Kruskal's algorithm, note that an edge uv can be added to the current forest if, and only if, u and v belong to different trees in the forest. To efficiently test this condition, we use the disjoint set ADT, where the disjoint sets correspond to vertex sets of the trees in the forest, and the forest is maintained using the procedures *Union* and *Find2* discussed in Chapter 4.

In the following pseudocode for procedure *Kruskal*, the forest F is maintained using its set of edges.

```

procedure Kruskal( $G, w, MCSTree$ )
Input:  $G$  (a connected graph with vertex set  $V$  of cardinality  $n$ 
           and edge set  $E = \{e_i = u_iv_i \mid i \in \{1, \dots, m\}\}$ )
            $w$  (a weight function on  $E$ )
Output:  $MSTree$  (a minimum spanning tree)

Sort the edges in nondecreasing order of weights  $w(e_1) \leq \dots \leq w(e_m)$ 
 $Forest \leftarrow \emptyset$ 
 $Size \leftarrow 0$ 
for  $i \leftarrow 0$  to  $n - 1$  do //initialize a disjoint collection of single vertices
     $Parent[i] \leftarrow -1$ 
endfor
 $j \leftarrow 0$ 
while  $Size \leq n - 1$  .and.  $j < m$  do
     $j \leftarrow j + 1$ 
     $Find2(Parent[0:n - 1], u_j, r)$       // $e_j = u_jv_j$ 
     $Find2(Parent[0:n - 1], v_j, s)$ 
    if  $r \neq s$  then //add edge  $e_j$  to  $Forest$ 
         $Forest \leftarrow Forest \cup \{e_j\}$ 
         $Size \leftarrow Size + 1$ 
         $Union(Parent[0:n - 1], r, s)$  //combine sets containing  $u_j$  and  $v_j$ 
    endif
endwhile
 $MSTree \leftarrow Forest$ 
end Kruskal

```

Since the procedure *Kruskal* does $n - 1$ unions and at most m finds, it follows for the results given in Chapter 4 that checking for cycles has a worst-case complexity that is almost linear in m . In procedure *Kruskal*, we assumed as a precondition that the edges were sorted in increasing order of their weights. Since presorting the edges can be done in time $O(m \log m) = O(m \log n)$, the overall order of the worst-case complexity is $\Theta(m \log n)$.

Instead of sorting the edges, we could more efficiently select the next minimum weight edge by utilizing a priority queue of edges of the graph, implemented, for example, with a min-heap. *CreatePriorityQueue* (*MakeMinHeap*) and

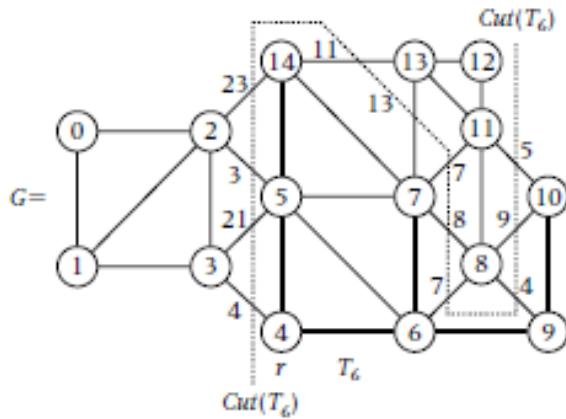
RemovePriorityQueue (*RemoveMinHeap*) then have worst-case complexities belonging to $O(m)$ and $O(\log m) = O(\log n)$, respectively. Hence, the selection process remains $O(m \log n)$ in the worst case. However, improvement over procedure *Kruskal* occurs for inputs where the tree is completed early (that is, not many of the selected edges form cycles). For example, in the best case, the first $n - 1$ edges selected form a tree, yielding best-case complexity belonging to $O(m + n \log n)$.

6.5.2 Prim's Algorithm

Prim's algorithm for generating a minimum spanning tree differs from Kruskal's algorithm in that it maintains a tree at each stage instead of a forest. The initial tree T_0 may be taken to be any single vertex r of the graph G . Prim's algorithm builds a sequence of n trees (rooted at r) T_0, T_1, \dots, T_{n-1} , where T_{i+1} is obtained from T_i by adding a single edge e_{i+1} , $i = 0, \dots, n-2$. The edge e_{i+1} is selected (greedily) to be a minimum weight edge among all edges having *exactly* one vertex in T_i . The set of all edges in G having exactly one vertex in T_i is denoted by $Cut(T_i)$. With this notation, at the i^{th} step Prim's algorithm chooses an edge e_i of minimum weight among all the edges in $Cut(T_i)$ (see Figure 6.9). That we can restrict attention to edges in $Cut(T_i)$ is a consequence of the following key fact.

Key Fact

Given any tree T and edge e in a graph G , $T \cup e$ is a tree if, and only if, $e \in Cut(T)$.



Sample tree T_6 of Prim's algorithm with $V(T_6) = \{4, 5, 6, 7, 9, 10, 14\}$ and $r = 4$.

$$Cut(T) = \{\{3,4\}, \{3,5\}, \{2,5\}, \{2,14\}, \{13,14\}, \{7,13\}, \{7,11\},$$

$$\{7,8\}, \{6,8\}, \{8,9\}, \{8,10\}, \{10,11\}\}.$$

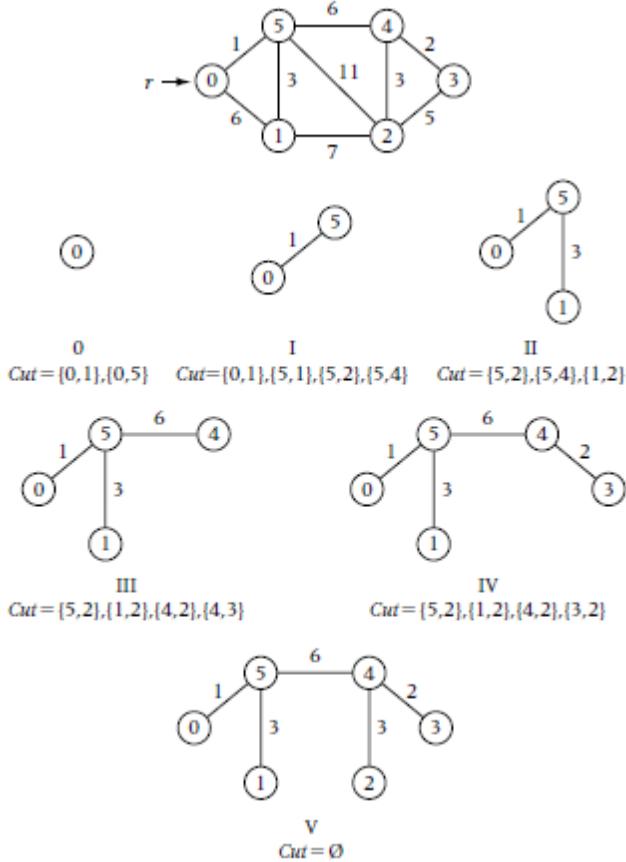
At stage 6 Prim's algorithm adds edge $e_6 = \{2, 5\}$ to T_6 .

since $\{2,5\}$ has minimum weight 3 in $Cut(T_6)$.

Sample tree T_6 of Prim's algorithms

Figure 6.9

A proof that Prim's algorithm yields the minimum spanning tree is left to the exercises. Prim's algorithm is illustrated in Figure 6.10 using the same sample graph as in Figure 6.8.



Tree T and $Cut(T)$ at each stage in Prim's algorithm, with $r = 0$

Figure 6.10

As with Kruskal's algorithm, the complexity of Prim's algorithm is dependent on specific implementation details. A straightforward implementation of Prim's algorithm based on explicitly implementing $Cut(T)$ at each stage is inefficient. A more efficient implementation of Prim's algorithm is based on the following observations. For convenience, we set $w(uv) = \infty$ (or some sufficiently large number) whenever u and v are nonadjacent vertices in G . For each vertex $v \in V(G) - V(T)$, we let $\eta_T(v)$ denote a vertex u of T that is “nearest” to v ; that is, $w(uv)$ equals the minimum of $w(xv)$ over all $x \in V(T)$. For example, in Figure 6.9 with $T = T_6$, $V(G) - V(T)$ consists of the vertices $v = 3, 4, 9, 12, 14$, with $\eta_T(v) = 6, 5, 10, 11, 15$, respectively. Clearly, the following equality holds:

$$\min\{w(\{v, \eta_T(v)\}) \mid v \in V(G) - V(T)\} = \min\{w(e) \mid e \in Cut(T)\}. \quad (6.5.1)$$

Thus, when implementing Prim's algorithm, we can maintain η_T at each stage rather than explicitly implementing $Cut(T)$.

When implementing Prim's algorithm, the tree T is dynamically maintained using an array $Parent[0:n - 1]$. The final state of $Parent[0:n - 1]$ yields the parent array implementation of a minimum spanning tree T . Guided by (6.5.1), at any stage of the algorithm a subarray of $Parent[0:n - 1]$ gives the parent array of the current tree T ,

whereas another subarray of $\text{Parent}[0:n - 1]$ maintains the nearest weight vertices $\eta_T(v)$. More precisely, $\text{Parent}[v]$ has the following dynamic interpretation. If $v \in V(T)$, then $\text{Parent}[v]$ is the parent of vertex v in T . If $v \notin V(T)$ but is adjacent to at least one vertex of T , then $\text{Parent}[v] = \eta_T(v)$. Otherwise, $\text{Parent}[v]$ is undefined. Initially, $\text{Parent}[r] = 0$.

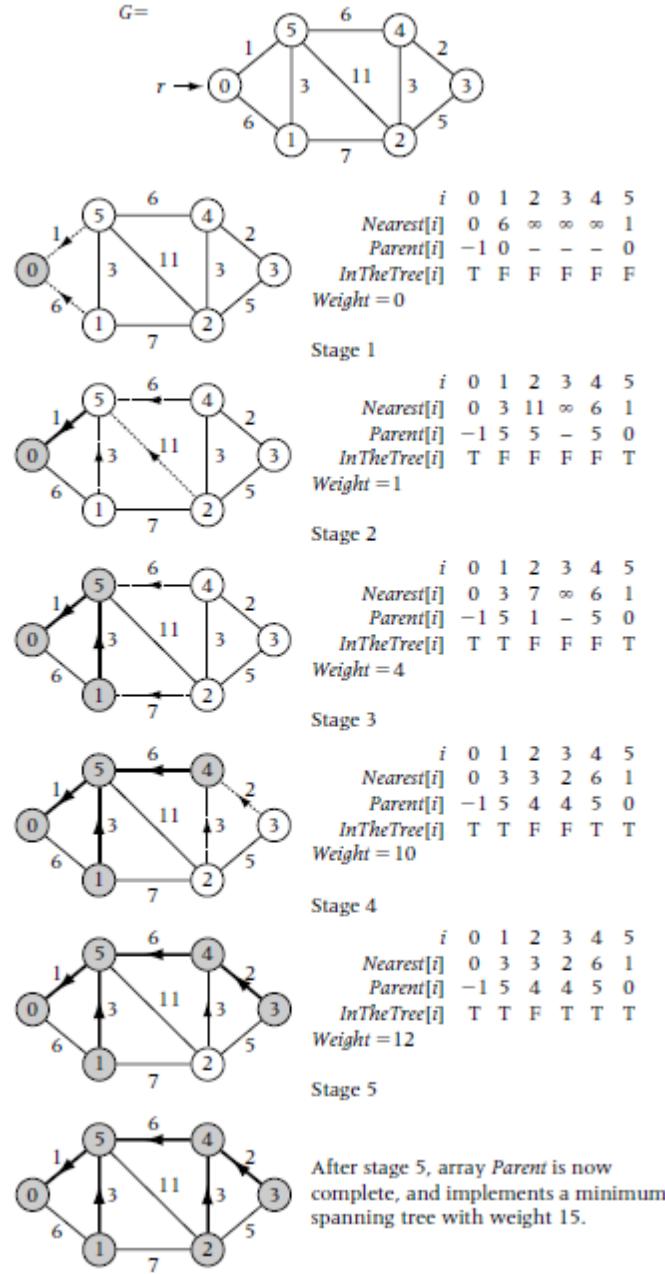
In the following pseudocode for procedure Prim , we also maintain a local array $\text{Nearest}[0:n - 1]$, which is initialized to ∞ 's, except for $\text{Nearest}[r]$ which is set to 0. At each stage, if $\text{Parent}[v]$ is defined, then $\text{Nearest}[v] = w(\{v, \text{Parent}[v]\})$, otherwise $\text{Nearest}[v] = \infty$. The next vertex u to be added to the tree T is the one such that $\text{Nearest}[u]$ is minimized over all $u \notin V(T)$. The arrays $\text{Parent}[0:n - 1]$ and $\text{Nearest}[0:n - 1]$ are examined for possible alteration at each stage for each vertex v not belonging to T that is adjacent to u . If $w(uv) < \text{Nearest}[v]$, then we set $\text{Parent}[v] = u$ and $\text{Nearest}[v] = w(uv)$. The procedure Prim also maintains a Boolean array InTheTree to keep track of the vertices not in T .

```

procedure Prim( $G, w, \text{Parent}[0:n - 1]$ )
Input:  $G$  (a connected graph with vertex set  $V$  and edge set  $E$ )
         $w$  (a weight function on  $E$ )
Output:  $\text{Parent}[0:n - 1]$  (parent array of a minimum spanning tree)
for  $v \leftarrow 0$  to  $n - 1$  do
     $\text{Nearest}[v] \leftarrow \infty$ 
     $\text{InTheTree}[v] \leftarrow \text{.false.}$ 
endfor
 $\text{Parent}[r] \leftarrow 0$                                 // root the tree at an arbitrary vertex  $r$ 
 $\text{Nearest}[r] \leftarrow 0$ 
for Stage  $\leftarrow 1$  to  $n - 1$  do
    Select vertex  $u$  that minimizes  $\text{Nearest}[u]$  over all  $u$  such that
     $\text{InTheTree}[u] = \text{.false.}$ 
     $\text{InTheTree}[u] \leftarrow \text{.true.}$                       // add  $u$  to  $T$ 
    for each vertex  $v$  such that  $uv \in E$  do          // update  $\text{Nearest}[v]$  and
        if .not.  $\text{InTheTree}[v]$  then                  //  $\text{Parent}[v]$  for all  $v \notin V(T)$ 
            if  $w(uv) < \text{Nearest}[v]$  then           // that are adjacent to  $u$ 
                 $\text{Nearest}[v] \leftarrow w(uv)$ 
                 $\text{Parent}[v] \leftarrow u$ 
            endif
        endif
    endfor
endfor
end Prim

```

The action of procedure Prim is illustrated in Figure 6.11 for the same graph given in Figure 6.10.



Action of procedure *Prim* for the same graph as in Figure 6.10

Figure 6.11

Remark

Note that the procedure *Prim* terminates after only $n - 1$ stages, even though there are n vertices in the final minimum spanning tree. The reason is simple: After stage $n - 1$ has been completed, $\text{InTheTree}[0:n - 1]$ is **false**, for only one vertex w . Thus, another iteration of the **for** loop controlled by *Stage* would result in no change to the arrays

$\text{Nearest}[0:n - 1]$ and $\text{Parent}[0:n - 1]$. In other words, the last vertex and edge in the minimum spanning tree come in for free.

Clearly, the procedure *Prim* has $\Theta(n^2)$ complexity, where we choose the comparison used in updating $\text{Nearest}[0:n - 1]$ in the inner **for** loop as the basic operation. The question arises as to whether procedure *Prim* is more or less efficient than the procedure *Kruskal*, which has $\Theta(m \log m)$ worst-case complexity. The answer depends on how many edges G has in comparison to the number of vertices. When the ratio m/n is large, then procedure *Prim* is more efficient, whereas when m/n is small, procedure *Kruskal* is better. For example, when the number of edges has the same order as the number of vertices (such as for graphs whose vertex degrees are bounded above by some fixed constant), procedure *Kruskal* has $\Theta(n \log n)$ worst-case complexity, which is significantly better than the $\Theta(n^2)$ worst-case complexity of procedure *Prim*. At the other extreme, when the number of edges is quadratic in the number of vertices (such as for the complete graph K_n , so that $m = n(n - 1)/2$), procedure *Kruskal* has worst-case complexity $\Theta(n^2 \log n)$, as compared to the $\Theta(n^2)$ worst-case complexity of procedure *Prim*.

Note that in procedure *Prim*, the operations performed on the array $\text{Nearest}[u]$ are essentially those associated with a priority queue. Thus, procedure *Prim* could be rewritten (procedure *Prim2*) using a priority queue Q of the vertices not in the current tree T . The priority of a vertex u is the weight of the edge joining u to its nearest neighbor $\eta_T(u)$ in T . After removing a vertex u from the priority queue and placing it in T , then we must reduce the priorities of each vertex v adjacent to u that is now “nearer” to the tree. If the priority queue Q is implemented as a min-heap, then both the operation of removing an element from Q and the operation of changing the priority value of an element in Q have $O(\log n)$ complexity. Since there are n removals of an element from Q and the number of times that the priority of a vertex v is lowered is no more than its degree, it follows that the worst-case complexity of *Prim2* belongs to $O(n \log n + 2m \log n) = O(m \log n)$. We leave the pseudocode for procedure *Prim2* as an exercise.

6.6 Shortest Paths in Weighted Graphs and Digraphs

In the previous chapter we saw that breadth first search solves the problem of finding shortest paths from a given vertex r in a graph or digraph to every vertex v reachable from r , where the weight of a path is simply the number of edges in a path. In a network setting this is often referred to as minimizing the number of *hops* from r to v . Often in applications, it is not just the weight of a path as measured by number of hops, but its total weight obtained by summing the weights of its edges, where each edge e has been assigned a nonnegative real weight $w(e)$. For example, a graph might model a network of cities, and the weight might be the distance or driving time between adjacent cities. As another example, a digraph might model a network of (direct) airplane flights, and the weight of a flight might be its cost or its flying time.

We now discuss an algorithm due to Dijkstra that grows a shortest path tree using the greedy method. For undirected graphs Dijkstra’s algorithm follows a similar strategy to Prim’s algorithm, differing only in the way the next edge is selected. At each stage in Dijkstra’s algorithm, instead of selecting an edge of minimum weight in $\text{Cut}(T)$, we select an edge $uv \in \text{Cut}(T)$, $u \in V(T)$, so that the path from r to v in the augmented tree $T \cup$

$\{uv\}$ is shortest; that is, such that $Dist[u] + w(uv)$ is minimized over all edges $uv \in Cut(T)$, where $Dist[u]$ is the weight of a path from r to u in T . The following pseudocode is a high-level description of Dijkstra's algorithm.

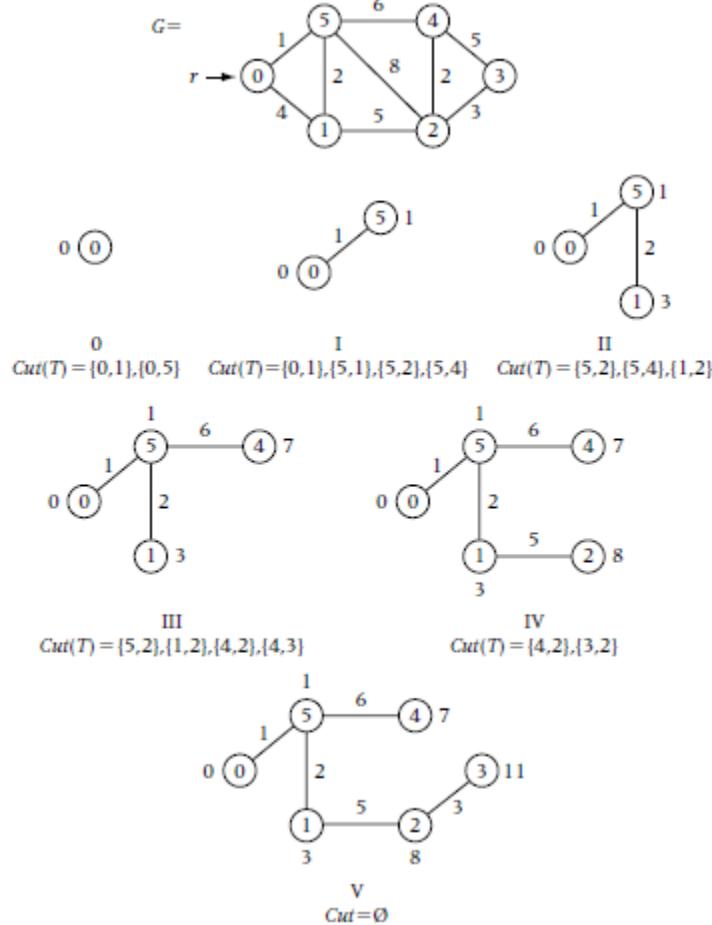
```

procedure Dijkstra( $G, a, w, T$ )
Input:  $G$  (a graph with vertex set  $V = \{0, \dots, n - 1\}$  and edge set  $E$ )
     $r$  (a vertex of  $G$ )
     $w$  (a nonnegative weighting on  $E$ )
Output:  $T$  (a shortest path tree rooted at  $r$ )
dcl  $Dist[0:n - 1]$  (an array initialized to  $\infty$ )
     $T$  is initialized to be the tree consisting of the single vertex  $r$ 
     $Dist[r] \leftarrow 0$ 

while  $Cut(T) \neq \emptyset$  do
    select an edge  $uv \in Cut(T)$  such that  $Dist[u] + w(uv)$  is minimized
    add vertex  $v$  and edge  $uv$  to  $T$ 
endwhile
end Dijkstra

```

Figure 6.12 shows the tree T at each stage of Dijkstra's algorithm for a sample weighted graph G . In this figure the label on the outside of each vertex v already included in the growing tree T is the weight of the path from v to a in T .



Tree T and $Cut(T)$ at each stage of Dijkstra's algorithms

Figure 6.12

Let T_i be the tree generated by the procedure *Dijkstra* after i iterations of the **while** loop, $i = 0, 1, \dots, t$, where t is the number of iterations of the loop ($t \leq n - 1$). For $v \in V(T_i)$, let $P_i^{(v)}$ denote the path from r to v in T_i , $i = 0, 1, \dots, t$. We prove correctness of procedure *Dijkstra* by establishing the following lemma.

Lemma 6.6.1 For each vertex $v \in V(T_i)$, $P_i^{(v)}$ is a shortest path in G from r to v , $i = 0, 1, \dots, t$.

Proof. We prove the lemma using mathematical induction on i .

Basis step: Clearly, the lemma is true for $i = 0$, since T_0 consists of the single vertex r .

Induction step: Assume that the lemma holds for T_i . The tree T_{i+1} is obtained from T_i by adding a single edge uv that minimizes $Dist[u] + w(uv)$ over all edges $uv \in Cut(T_i)$, where $Dist[u]$ is the weight of $P_u^{(i)}$. By induction assumption, for all $x \in V(T_i)$, $P_x^{(i+1)} = P_x^{(i)}$ is a shortest path in G from r to x . It remains to show that $P_v^{(i+1)}$ is a shortest path in

G from r to v . Consider any path Q in G from r to v . Since the deletion of the edges in $Cut(T_i)$ disconnects G with vertices r and v lying in different components, Q must contain edge xy from $Cut(T_i)$, $x \in V(T_i)$, such that the subpath Q_x of Q from r to x lies entirely in T_i . Then we have

$$\begin{aligned} weight(Q) &\geq weight(Q_x) + w(xy) \quad (\text{since the weight function is nonnegative}) \\ &\geq weight(P_x^{(i)}) + w(xy) \quad (\text{since } P_x^{(i)} \text{ is a shortest path in } G \text{ from } r \text{ to } x \text{ in } G) \\ &\geq weight(P_u^{(i)}) + w(uv) \quad (\text{by the greedy choice Dijkstra makes}) \\ &= weight(P_u^{(i+1)}) \end{aligned}$$

Since Q was chosen to be an arbitrary path in G from r to v , it follows that $P_u^{(i+1)}$ is a shortest path in G from r to v . This completes the induction step and the correctness proof of procedure *Dijkstra*. \blacksquare

As with Kruskal's and Prim's algorithms, the complexity of Dijkstra's algorithm is dependent on specific implementation details. We now implement a procedure *Dijkstra* that is similar to the procedure *Prim* in that the tree T is dynamically grown using an array $Parent[0:n - 1]$. We also maintain a Boolean array *InTheTree* to keep track of the vertices not in T . Again, rather than having to explicitly maintain the sets $Cut(T_i)$, we can efficiently select the next edge uv to be added to the tree by maintaining an array $Dist[0:n - 1]$, where $Dist[v]$ is the distance from r to v in T and $Dist[v] = \infty$ if v is not in T . Upon completion of *Dijkstra*, the distance from the root r to all the vertices of G is contained in the array $Dist[0:n - 1]$. At each intermediate stage $Dist[v]$ has the following dynamic interpretation. If $v \in V(T)$, then $Dist[v]$ is the weight of a path in T from r to v . If $v \notin V(T)$ and v is adjacent to a vertex of T , then $Dist[v]$ is the minimum value of $Dist[u] + w(uv)$ over all $uv \in Cut(T)$. If $v \notin V(T)$ and v is not adjacent to any vertex of T , then $Dist[v] = \infty$. $Dist[v]$ is initialized to ∞ for each vertex $v \neq r$ and $Dist[r]$ is initialized to 0. Each time a vertex u is added to the tree T , we need only update $Parent[v]$ and $Dist[v]$ for those vertices v that are adjacent to u and do not belong to the tree T . If $Dist[v] > Dist[u] + w(uv)$, then we set $Parent[v] = u$ and $Dist[v] = Dist[u] + w(uv)$.

```

procedure Dijkstra( $G, w, r, Parent[0:n - 1], Dist$ )
Input:  $G$  (a connected graph with vertex set  $V$  and edge set  $E$ )
         $r$  (a vertex of  $G$ )
         $w$  (a weight function on  $E$ )
Output:  $Parent[0:n - 1]$  (parent array of a shortest path spanning tree)
         $Dist[0:n - 1]$  (array of weights of shortest paths from  $a$ )
for  $v \leftarrow 1$  to  $n$  do // initialize  $Dist[0:n - 1]$  and InTheTree[0:n - 1]
     $Dist[v] \leftarrow \infty$ 
    InTheTree[v]  $\leftarrow$  .false.
endfor
 $Dist[r] \leftarrow 0$ 
 $Parent[r] \leftarrow 0$ 
for Stage  $\leftarrow 1$  to  $n - 1$  do
    Select vertex  $u$  that minimizes  $Dist[u]$  over all  $u$  such that

```

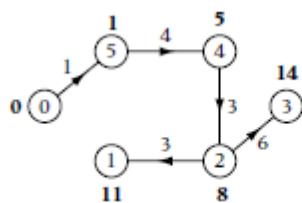
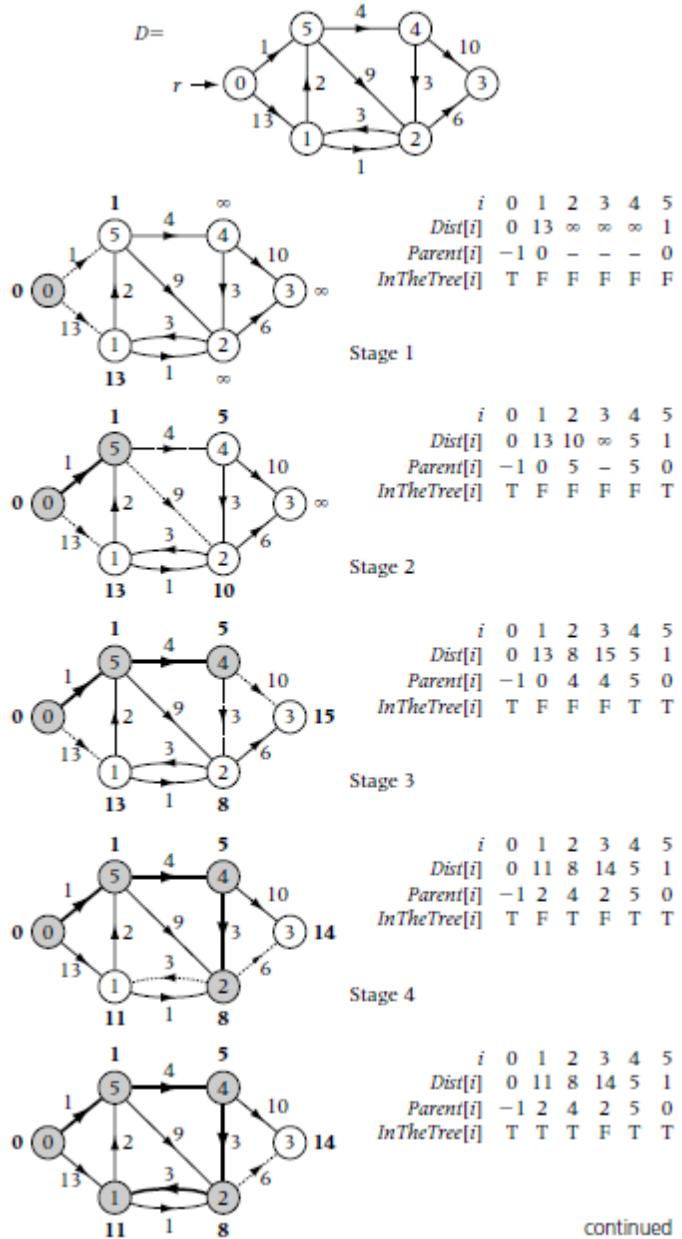
InTheTree[u] = .false.

```
InTheTree[ $u$ ] ← .true.  
for each vertex  $v$  such that  $uv \in E$  do// update  $Dist[v]$  and  $Parent[v]$  arrays}  
  if .not. InTheTree[ $v$ ] then  
    if  $Dist[u] + w(uv) < Dist[v]$  then  
       $Dist[v] \leftarrow Dist[u] + w(uv)$   
       $Parent[v] \leftarrow u$   
    endif  
  endif  
endfor  
endfor  
end Dijkstra
```

Remarks

1. As with procedure Prim, procedure Dijkstra terminates after only $n - 1$ stages, even though there are n vertices in the final shortest path spanning tree. Another iteration of the for loop controlled by *Stage* would result in no change to the arrays $Dist[0:n - 1]$ and $Parent[0:n - 1]$.
2. In spite of the similarity between Prim's algorithm and Dijkstra's algorithm and the fact that they both generate spanning trees, they solve different problems. Simple examples show that a shortest path spanning tree can fail to be a minimum spanning tree (see Exercise 6.x).

The shortest path algorithms presented in this section for (undirected) graphs generalize naturally to digraphs. In fact, the code for the procedure *Dijkstra* applies essentially unchanged to digraphs. We merely have to be careful in the case of digraphs to note that uv corresponds to a directed edge $uv = (u,v)$ from u to v (as opposed to the undirected edge $uv = \{u,v\}$). We illustrate the action of Dijkstra's algorithm for a sample digraph in Figure 6.13, where the current value of $Dist[v]$ is shown outside each vertex v .



Resulting shortest path spanning out-tree rooted at vertex $r = 0$

Action of procedure *Dijkstra* for a sample digraph D

Figure 6.13

As with procedure *Prim*, the worst-case complexity of procedure *Dijkstra* belongs to $\Theta(n^2)$.

6.5 Closing Remarks

The greedy method is a powerful tool to use, when it applies, since solutions generated by this method are often elegant and very efficient. However, as we have seen, potential solutions generated by the greedy method must always be verified for correctness. Often, greedy decisions that are locally optimal do not lead in the end to solutions that are globally optimal.

The area of data compression is a very active area of research today. Because of limited bandwidth and the vast amount of visual and audio text files that are being transported over the Internet, data compression is absolutely necessary in order to handle these transmissions efficiently. Some of these data compression algorithms such as jpeg not only use the Huffman code algorithm, but also use transformations closely related to the Fast Fourier Transform that we discuss in Chapter 7.

In this chapter we discussed Dijkstra's algorithm for computing shortest paths based on the greedy method. In Chapter 8 we will discuss two other algorithms for computing shortest paths, the Floyd-Warshall and Bellman-Ford algorithms, based on the design strategy Dynamic Programming.

Exercises

Section 6.1 General Description

- 6.1 Given the standard U.S. coins, pennies (1¢), nickels (5¢), dimes (10¢), quarters (25¢), and half-dollars (50¢), show that the greedy method of making change always yields a solution using the fewest coins.
- 6.2 For a given integer base $b > 2$, suppose you have (an unlimited number of) coins of each of the denominations b^0, b^1, \dots, b^{n-1} . Show that the greedy method of making change for these denominations always yields a solution using the fewest coins.

Section 6.2 Optimal Sequential Storage Order

- 6.3 Design and analyze a greedy algorithm for the optimal sequential storage problem when m tapes are available.
- 6.4 For the sequential storage order problem, consider the general case of minimizing $AveCost(\pi)$ as given by (6.2.1), where p_i denotes the probability that the object to be processed is b_i , $i = 0, \dots, n - 1$.
 - a. Show that the greedy strategy used when all the probabilities are equal (that is, sorting the objects by increasing order of costs c_i) fails to give an optimal solution for a general set of probabilities p_i .

- b. Show that the greedy strategy of arranging the objects in decreasing order of probabilities also fails to give optimal solutions in general.
- c. Show that the greedy strategy of arranging the objects in increasing order of the ratios c_i/p_i yields an optimal solution.

Section 6.3 The Knapsack Problem

6.5 Solve the following instance of the knapsack problem for capacity $C = 30$:

i	0	1	2	3	4	5	6	7
v_i	60	50	40	30	20	10	5	1
w_i	30	100	10	10	8	8	1	1

6.6 For the Knapsack problem with weights w_0, w_1, \dots, w_{n-1} and values v_0, v_1, \dots, v_{n-1} , give a single example where both the greedy solution based on placing the objects in the knapsack in the order $w_0 \leq w_1 \leq \dots \leq w_{n-1}$ as well as the greedy solution based on placing the objects in the knapsack in the order $v_0 \geq v_1 \geq \dots \geq v_{n-1}$ yield suboptimal solutions.

- 6.7
- a. Give an example where the greedy method described in Section 6.3 for the 0/1 Knapsack problem yields a suboptimal solution.
 - b. Show that the greedy method yields arbitrarily bad solutions; that is, given any $\varepsilon > 0$, find an example where the ratio of the greedy solution to the optimal solution is less than ε .

Section 6.4 Huffman Codes

- 6.8 Prove Proposition 6.4.1
- 6.9 Trace the action of *HuffmanCode* for the letters a, b, c, d, e, f, g, h occurring with frequencies 10, 7, 3, 5, 9, 2, 3, 2.
- 6.10 Given the Huffman Code Tree in Figure 6.6, decode the string
100111100001101111001
- 6.11 Suppose the Huffman Tree has nodes with the structure

```

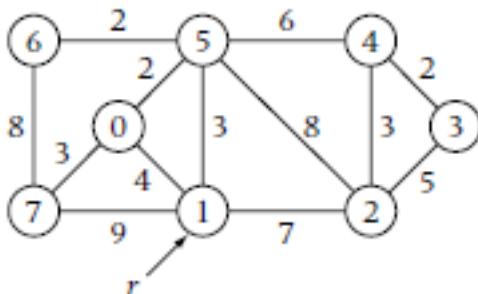
HuffmanTreeNode = record
    Symbol: character
    LeftChild: → HuffmanTreeNode
    RightChild: → HuffmanTreeNode
end HuffmanTreeNode

```

- 6.12 Given an arbitrary 2-tree T having n leaf nodes, find a set of frequencies such T is a Huffman tree for these frequencies.
- 6.13 Using a greedy algorithm to solve the *optimal merge pattern* problem. In this problem, we have n sorted files of lengths f_0, f_1, \dots, f_{n-1} , and we wish to merge them into a single file by a sequence of merges of pairs of files. To merge two files of lengths m_1 and m_2 takes $m_1 + m_2$ operations.

6.5 Minimum Spanning Tree

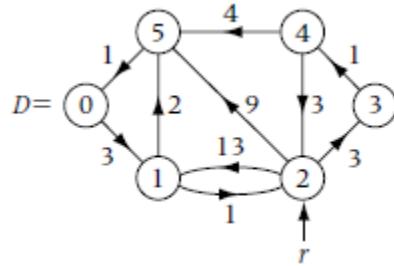
6.14 Consider the following weighted graph G .



- a. Trace the action of procedure *Kruskal* for G .
 - b. Trace the action of procedure *Prim* for G , with $r = 1$.
- 6.15 Give pseudocode for a version of the algorithm *Kruskal2* that maintains the weighted edges of the graph in a priority queue.
- 6.16 Prove the following two propositions used in the proof of the correctness of Kruskal's algorithm.
- a. Adding an edge to a tree creates a unique cycle.
 - b. Removing an edge from a cycle in a connected graph still leaves the resulting graph connected.
- 6.17a. Show how the problem of finding a minimum spanning tree in a weighted graph with repeated edge weights can be transformed into an equivalent problem with distinct edge weights
- b. Show that the minimum spanning tree is unique when the edge weights are distinct.
- 6.18 Modify procedure *Kruskal* to accept *any* graph G (connected or disconnected) and to output a minimum (weight) forest, each of whose trees is a minimum spanning tree for a component of G .
- 6.19 Prove that Prim's algorithm yields a minimum spanning tree.
- 6.20 Analyze the complexity of the variant of procedure *Prim* based on an explicit implementing of *Cut*(T) at each stage of the algorithm. Compare its complexity to that of procedure *Prim*.
- 6.21 Give pseudocode for procedure *Prim2* using a priority queue.
- 6.22 The problem of finding a minimum spanning tree in an undirected graph G generalizes naturally to the problem of finding a minimum spanning in-tree (or out-tree) rooted at a given vertex r in a strongly connected digraph D . Prim's algorithm directly generalizes to digraphs, with the modification that *Cut*(T) is replaced by *CutIn*(T) (or *CutOut*(T)) consisting of all edges of *Cut*(T) having head in T (or having tail in T). Show that this greedy method doesn't always yield a spanning in-tree (or out-tree) rooted at r .

6.6 Shortest Paths in Weighted Graphs and Digraphs

- 6.23 a) Trace the action of procedure *Dijkstra* for the following digraph with initial vertex $r = 2$.



b) repeat for $r = 3$.

- 6.24 a. Trace the action of procedure *Dijkstra* for the graph G of Figure 6.x with $r = 1$.
 b. Note that the shortest path tree generated in part (a) happens to coincide with the minimum spanning tree. Give an example of a weighted graph G with distinct weights and a vertex r such that the shortest spanning tree T generated by procedure *Dijkstra* is not a minimum spanning tree. In fact, find an example of a graph on n vertices where the shortest path tree has only one edge in common with the minimum spanning tree.
- 6.25 Given a weighted graph G and vertices a, b , consider the following greedy strategy that attempts to grow a shortest path from a to b . Choose an edge e having smallest weight among the remaining edges incident with the terminal vertex u of the path P previously generated, such that e contains no vertex of P other than u . These greedy choices continue until either b is reached or no choice for e exists. Give an example where the path so grown
 a. never reaches b ,
 b. reaches b , but is not a shortest path from a to b .
- 6.26 Prove that the tree generated by procedure *Dijkstra* spans all vertices in the component of the graph G containing a .
- 6.27 Give pseudocode for *Dijkstra2* using a priority queue.
- 6.28 Show that Dijkstra's algorithm can fail in the case where some of the weights are negative, even though no negative cycles exist. (In particular, why doesn't adding a sufficiently large positive constant to each edge always work?)

Divide and Conquer

The *divide-and-conquer* paradigm is one of the most powerful design strategies available in the theory of algorithms. The paradigm can be described in general terms as follows. A problem input (instance) is divided according to some criteria into a set of smaller inputs to the same problem. The problem is then solved for each of these smaller inputs, either recursively by further division into smaller inputs or by invoking an ad hoc or a priori solution. Finally, the solution for the original input is obtained by expressing it in some form as a combination of the solution for these smaller inputs. Ad hoc solutions are often invoked when the input size is smaller than some preassigned *threshold* value. Examples of a priori solutions (solutions known in advance) include sorting single element lists or multiplying single-digit binary numbers.

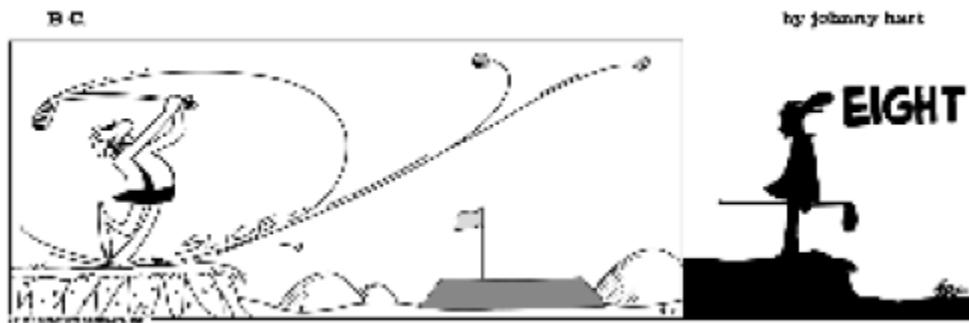
7.1 The Divide-and-Conquer Paradigm

The divide-and-conquer design strategy can be formalized as follows. Let *Known* denote the set of inputs to the problem whose solutions are known a priori or by ad hoc methods. The procedure *Divide_and_Conquer* calls two procedures *Divide* and *Combine*. *Divide* has input parameter I and output parameters I_1, \dots, I_m , where m may depend on I . The inputs I_1, \dots, I_m must be smaller or simpler inputs to the problem than I , but are not required to always be a *division* of I into subinputs. *Combine* has input parameters J_1, \dots, J_m and output parameter J . *Combine* is the procedure for obtaining a solution J to the problem with input I by combining the recursively obtained solutions J_1, \dots, J_m to the problem with inputs I_1, \dots, I_m . For convenience, we formulate *Divide_and_Conquer* as a procedure.

```
procedure Divide_and_Conquer( $I, J$ ) recursive
Input:  $I$  (an input to the given problem)
Output:  $J$  (a solution to the given problem corresponding to input  $I$ )
if  $I \in \text{Known}$  then
    assign the a priori or ad hoc solution for  $I$  to  $J$ 
else
    Divide( $I, I_1, \dots, I_m$ )      // $m$  may depend on the input  $I$ 
    for  $i \leftarrow 1$  to  $m$  do
        Divide_and_Conquer( $I_i, J_i$ )
    endfor
    Combine( $J_1, \dots, J_m, J$ )
endif
end Divide_and_Conquer
```

Often the work done by an algorithm based on *Divide_and_Conquer* resides solely in one of the two procedures *Divide* or *Combine*, but not both. For example, in the algorithm *QuickSort*, the divide step (call to *Partition*) is the heart of the algorithm and

the combine step requires no work at all. On the other hand, in the algorithm *MergeSort*, the divide step is trivial and the combine step (call to *Merge*) is the heart of the algorithm. Sometimes, both the divide and combine steps are difficult (see Figure 7.1).



An example of a hard divide – hard combine divide-and-conquer algorithm
(Reprinted by permission of Johnny Hart and Creators Syndicate, Inc.)

Figure 7.1

For most divide-and-conquer algorithms, the number m of subproblems is a constant (independent of any particular input I). Divide-and-conquer algorithms where m is a constant equal to one are referred to as *simplifications*. The binary search algorithm is an example of a simplification.

One useful technique to improve the efficiency of a divide-and-conquer algorithm is to employ a known ad hoc algorithm when the input size is smaller than some *threshold*. Of course, the ad hoc algorithm must be more efficient than the given divide-and-conquer algorithm for sufficiently small inputs.

For example, consider the sorting algorithm *MergeSort*, which has $\Theta(n \log n)$ average complexity. The sorting algorithm *InsertionSort*, which has $\Theta(n^2)$ average complexity, is much less efficient than *MergeSort* for large values of n . However, due to the constants involved, *InsertionSort* is more efficient than *MergeSort* for small values of n . Thus, we can improve the performance of *MergeSort* by calling *InsertionSort* for input lists of size not larger than a suitable threshold. Finding the optimal value for a threshold is often done empirically in practice. Empirical studies are needed since the best choice of a threshold depends on the constants associated with the implementation on a particular computer. For example, empirical studies have shown that for *MergeSort*, calling *InsertionSort* with a threshold of around $n = 16$ is usually optimal.

7.2 Symbolic Algebraic Operations on Polynomials

Algebraic manipulation of polynomials is an essential tool in many applications, and therefore the need arises to design efficient algorithms to carry out the basic arithmetic operations on polynomials, such as addition and multiplication, as efficiently as possible. Given a polynomial $P(x) = a_{m-1}x^{m-1} + \dots + a_1x + a_0$, it is important to differentiate between the pointwise and the symbolic representation of $P(x)$. The *pointwise representation* of $P(x)$ is simply a function that maps an input point x to the output point

$P(x)$. Thus, given two polynomials $P(x)$ and $Q(x)$, their *pointwise product* is the function $PtWiseMulT(P, Q)$ that maps an input point x to the output point $P(x) \times Q(x)$.

The *symbolic representation* of a polynomial $P(x) = a_{m-1}x^{m-1} + \dots + a_1x + a_0$ is its coefficient array $[a_0, a_1, \dots, a_{m-1}]$. Thus given two polynomials $P(x) = a_{m-1}x^{m-1} + \dots + a_1x + a_0$ and $Q(x) = b_{n-1}x^{n-1} + \dots + b_1x + b_0$, their symbolic product is the coefficient array $[c_0, c_1, \dots, c_{m+n-2}]$ of the product polynomial $P(x)Q(x)$ given by

$$c_k = \sum_{i+j=k} a_i b_j, \quad 0 \leq i \leq m-1, 0 \leq j \leq n-1, k = 0, \dots, m+n-2. \quad (7.2.1)$$

For example, the symbolic product of $P(x) = 3x^2 + 2x - 5$ having coefficient array $[-5, 2, 3]$ and $Q(x) = x^3 - x + 4$ having coefficient array $[4, -1, 0, 1]$ is the polynomial $3x^5 + 2x^4 - 8x^3 + 10x^2 + 13x - 20$ having coefficient array $[-20, 13, 10, -8, 2, 3]$.

The symbolic representation of the product is particularly important for determining various properties of the product polynomial. For example, the function *PtWiseMult* does not lend itself to taking the derivative of the product polynomial, whereas it is a simple matter to compute the symbolic representation of the derivative of a polynomial that is represented symbolically. Similar comments hold for other operations on polynomials such as root finding, integration, and so forth. Throughout the remainder of this chapter, when discussing algebraic operations on polynomials we implicitly assume that we are performing these operations symbolically.

7.2.1 Multiplication of Polynomials of the Same Input Size

An algorithm *DirectPolyMult* based on a straightforward calculation of (7.2.1) has complexity $\Theta(mn)$ (where we choose multiplication of coefficients as our basic operation). We now describe a more efficient algorithm for polynomial multiplication based on the divide-and-conquer paradigm. We first assume that $m = n$. Setting $d = \lceil n/2 \rceil$, we divide the set of coefficients of the polynomials in half, with the higher-order coefficients $a_{n-1}, a_{n-2}, \dots, a_d$ in one set and the lower-order coefficients of $a_{d-1}, a_{d-2}, \dots, a_0$ in the other. Setting $P_1(x) = a_{d-1}x^{d-1} + a_{d-2}x^{d-2} + \dots + a_1x + a_0$ and $P_2(x) = a_{n-1}x^{n-d-1} + a_{n-2}x^{n-d-2} + \dots + a_{d+1}x + a_d$ we obtain

$$P(x) = x^d P_2(x) + P_1(x).$$

A similar division of the set of coefficients of $Q(x)$ yields polynomials $Q_1(x)$ and $Q_2(x)$ having input size of at most d such that

$$Q(x) = x^d Q_2(x) + Q_1(x).$$

A straightforward application of the distributive law yields

$$P(x)Q(x) = x^{2d} P_2(x)Q_2(x) + x^d (P_1(x)Q_2(x) + P_2(x)Q_1(x)) + P_1(x)Q_1(x). \quad (7.2.2)$$

Note that polynomials $P_1(x)$ and $Q_1(x)$ both have input size d , and polynomials $P_2(x)$ and $Q_2(x)$ both have input size either d or $d - 1$. In the case when $P_2(x)$ and $Q_2(x)$ both have input size $d - 1$, we add a leading coefficient of zero to each so that they have input size d . Thus, the problem of multiplying $P(x)$ and $Q(x)$ has been reduced to the problem

of taking four products of polynomials of input sizes $d = \lceil n/2 \rceil$ together with two multiplications by powers of x and three additions. It turns out that the resulting divide-and-conquer algorithm based on (7.2.2) still has quadratic complexity. However, there is a clever way of combining the split polynomials that uses only *three* polynomial multiplications instead of four, based on the following simple identity

$$P(x)Q(x) = x^{2d} P_2(x)Q_2(x) + x^d ((P_1(x) + P_2(x))(Q_1(x) + Q_2(x)) - P_1(x)Q_1(x) - P_2(x)Q_2(x)) + P_1(x)Q_1(x). \quad (7.2.3)$$

Formula (7.2.3) yields the following divide-and-conquer algorithm *PolyMult1* for polynomial multiplication. *PolyMult1* calls a (split pea) procedure *Split*($P(x), P_1(x), P_2(x)$), which inputs a polynomial $P(x)$ (of input size n) and outputs the two polynomials $P_1(x)$ and $P_2(x)$. We write *PolyMult1* as a high-level recursive function whose inputs are the polynomials $P(x)$ and $Q(x)$ and whose output is the polynomial $P(x)Q(x)$. We will not be explicit about how the coefficients of the polynomials are maintained (linked lists, arrays, and so forth). We will also assume that we have well-defined procedures for multiplying a polynomial by x^i . For example, if the polynomial is maintained by an array of its coefficients, then this amounts to shifting indices by i and replacing the first i entries with zeros. We abuse the notation slightly by writing $x^i P(x)$ to mean the result of calling such a procedure. Also, for convenience we simply use the symbol $+$ to denote the addition of two polynomials.

```

function PolyMult1( $P, Q, n$ ) recursive
Input:  $P(x) = a_{n-1}x^{n-1} + \dots + a_1x + a_0$ ,  $Q(x) = b_{n-1}x^{n-1} + \dots + b_1x + b_0$ 
Output:  $P(x)Q(x)$  (the product polynomial)
if  $n = 1$  then
    return( $a_0b_0$ )
else
     $d \leftarrow \lceil n/2 \rceil$ 
    Split( $P(x), P_1(x), P_2(x)$ )
    Split( $Q(x), Q_1(x), Q_2(x)$ )
     $R(x) \leftarrow PolyMult1(P_2(x), Q_2(x), d)$ 
     $S(x) \leftarrow PolyMult1(P_1(x) + P_2(x), Q_1(x) + Q_2(x), d)$ 
     $T(x) \leftarrow PolyMult1(P_1(x), Q_1(x), d)$ 
    return( $x^{2d}R(x) + x^d(S(x) - R(x) - T(x)) + T(x)$ )
endif
end PolyMult1

```

When analyzing *PolyMult1*, we choose coefficient multiplication as our basic operation, and thus ignore the two multiplications by powers of x . However, the procedure referred to earlier for multiplying a polynomial by x^i has linear complexity and does not affect the order of complexity of *PolyMult1*. We also assume that n is a power of two, since we can interpolate asymptotic behavior using Θ -scalability (see Section 7.4.3). Since *PolyMult1* invokes itself three times with n replaced by $d = n/2$, the number of

coefficient multiplications $T(n)$ performed by *PolyMult1* satisfies the following recurrence relation

$$T(n) = 3T\left(\frac{n}{2}\right), \quad n > 1, \quad \text{init.cond. } T(1) = 1. \quad (7.2.4)$$

It follows from a simple unwinding of (7.2.4) that $T(n) \in \Theta(n^{\log_2 3})$ (see also the discussion following (3.3.12) in Chapter 3). Since $\log_2 3$ is approximately 1.59, we now have an algorithm for polynomial multiplication whose $\Theta(n^{\log_2 3})$ complexity is a significant improvement over the $\Theta(n^2)$ complexity of *DirectPolyMult*. In Chapter 21 we develop an even faster polynomial multiplication algorithm utilizing the powerful tool known as the Fast Fourier Transform.

7.2.2 Multiplication of Polynomials of Different Input Sizes

In practice, we often encounter the problem of multiplying two polynomials $P(x)$ and $Q(x)$ of different input sizes m and n , respectively. If $m < n$, then we could merely augment $P(x)$ with $n - m$ leading zeros, but this would be quite inefficient if n is significantly larger than m . It is better to partition $Q(x)$ into blocks of size m . For convenience, we assume n is a multiple of m , that is, $n = km$ for some positive integer k . We let $Q_i(x)$ be the polynomial of degree m given by

$$Q_i(x) = b_{im-1}x^{m-1} + b_{im-2}x^{m-2} + \cdots + b_{(i-1)m+1}x + b_{(i-1)m}, \quad i \in \{1, \dots, k\}.$$

Clearly,

$$Q(x) = Q_k(x)x^{m(k-1)} + Q_{k-1}(x)x^{m(k-2)} + \cdots + Q_2(x)x^m + Q_1(x).$$

It follows immediately from the distributive law that

$$P(x)Q(x) = P(x)Q_k(x)x^{m(k-1)} + P(x)Q_{k-1}(x)x^{m(k-2)} + \cdots + P(x)Q_1(x).$$

Applying the above ideas, we obtain the following algorithm *PolyMult2* for multiplying two polynomials $P(x)$ and $Q(x)$. *PolyMult2* is efficient even if the degree $m - 1$ of $P(x)$ is much less than the degree $n - 1$ of $Q(x)$. If n is not a multiple of m , we compute the largest integer k such that $n > km$ and augment $P(x)$ with $n - km$ leading zeros.

```
function PolyMult2( $P(x), Q(x), m, n$ )
Input:  $P(x) = a_{m-1}x^{m-1} + \dots + a_1x + a_0$ ,  $Q(x) = b_{n-1}x^{n-1} + \dots + b_1x + b_0$ 
          (polynomials)
           $n, m$  (positive integers) // $n = km$  for some integer  $k$ 
Output:  $P(x)Q(x)$  (the product polynomial)
ProdPoly( $x$ )  $\leftarrow 0$  {initialize all coefficients of ProdPoly( $x$ ) to be 0}
for  $i \leftarrow 1$  to  $k$  do
     $Q_i(x) = b_{im-1}x^{m-1} + b_{im-2}x^{m-2} + \dots + b_{(i-1)m}$ 
endfor
for  $i \leftarrow 1$  to  $k$  do
     $\text{ProdPoly}(x) \leftarrow \text{ProdPoly}(x) + x^{(i-1)m} \text{PolyMult1}(P(x), Q_i(x), m)$ 
endfor
```

```

return(ProdPoly(x))
end PolyMult2

```

The complexity of *PolyMult1* for multiplying two polynomials of degree $m - 1$ is $\Theta(m^{\log_2 3})$. Since *PolyMult2* invokes *PolyMult1* a total of $k = n/m$ times, each time with input polynomials of degree $m - 1$, it follows that the complexity of *PolyMult2* is

$$\Theta(km^{\log_2 3}) = \Theta\left(\frac{nm^{\log_2 3}}{m}\right) = \Theta(nm^{\log_2(3/2)})$$

7.3 Multiplication of Large Integers

Computers typically assign a fixed number of bits for storing integer variables. Arithmetic operations such as addition and multiplication of integers are often carried out by moving the integer operands into *fixed-length* registers and then invoking arithmetic operations built into the hardware. However, there are important applications, such as those that occur in cryptography, when the number of digits is too large to be handled in this way directly by the hardware. In such cases, it is necessary to perform these operations by storing the integers using an appropriate data structure (such as an array or a linked list) and then writing algorithms for carrying out the arithmetic operations. When analyzing the complexity of such algorithms, the size n of an integer A is taken to be the number of digits of A . Any base $b \geq 2$ can be chosen for representing an integer. We denote the i th least significant digit of U (base b) by u_i , $i = 0, 1, \dots, n - 1$; that is,

$$U = \sum_{i=0}^{n-1} u_i b^i. \quad (7.3.1)$$

The classic grade-school algorithm for multiplying two integers U and V of size n clearly involves n^2 multiplications of digits. Viewing the right-hand side of (7.3.1) as a polynomial in b leads us to a divide-and-conquer strategy for computing UV based on a formula analogous to (7.2.3). Note that although addition and multiplication of large integers are similar to multiplication of polynomials, there is the additional task of handling carry digits. The design details of the resulting $\Theta(n^{\log_2 3})$ algorithm *MultInt* are left to the exercises.

7.4 Multiplication of Matrices

Given two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$, $0 \leq i, j \leq n - 1$, recall that the product AB is defined to be the $n \times n$ matrix $C = (c_{ij})$, where

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}. \quad (7.4.1)$$

The straightforward algorithm based on this definition clearly performs n^3 (scalar) multiplications. Around 1970 Strassen devised a divide-and-conquer algorithm for matrix

multiplication of complexity $O(n^{\log_2 7})$ using certain algebraic identities for multiplying 2×2 matrices.

The classic method of multiplying 2×2 matrices performs 8 multiplications as follows:

$$AB = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{bmatrix}. \quad (7.4.2)$$

Strassen discovered a way to carry out the same matrix product AB using only the following seven multiplications:

$$\begin{aligned} m_1 &= (a_{00} + a_{11})(b_{00} + b_{11}) \\ m_2 &= (a_{10} + a_{11})b_{00} \\ m_3 &= a_{00}(b_{01} - b_{11}) \\ m_4 &= a_{11}(b_{10} - b_{00}) \\ m_5 &= (a_{00} + a_{01})(b_{11}) \\ m_6 &= (a_{10} - a_{00})(b_{00} + b_{01}) \\ m_7 &= (a_{01} - a_{11})(b_{10} + b_{11}) \end{aligned} \quad (7.4.3)$$

The matrix product AB is then given by

$$AB = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}. \quad (7.4.4)$$

Consider now the case of two $n \times n$ matrices, where for convenience we assume that $n = 2^k$. We first partition the matrices A and B into four $(n/2) \times (n/2)$ submatrices, as shown.

$$A = \begin{bmatrix} A_{00} & | & A_{01} \\ \hline \cdots & | & \cdots \\ A_{10} & | & A_{11} \end{bmatrix}, \quad B = \begin{bmatrix} B_{00} & | & B_{01} \\ \hline \cdots & | & \cdots \\ B_{10} & | & B_{11} \end{bmatrix} \quad (7.4.5)$$

The product AB can be expressed in terms of eight matrix products as follows.

$$AB = \begin{bmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{bmatrix}. \quad (7.4.6)$$

Thus, in complete analogy with the 2×2 case, we can carry out the matrix product AB using only the following seven matrix multiplications.

$$\begin{aligned} M_1 &= (A_{00} + A_{11})(B_{00} + B_{11}) \\ M_2 &= (A_{10} + A_{11})B_{00} \\ M_3 &= A_{00}(B_{01} - B_{11}) \\ M_4 &= A_{11}(B_{10} - B_{00}) \\ M_5 &= (A_{00} + A_{01})(B_{11}) \\ M_6 &= (A_{10} - A_{00})(B_{00} + B_{01}) \\ M_7 &= (A_{01} - A_{11})(B_{10} + B_{11}) \end{aligned} \quad (7.4.7)$$

As in the case of 2×2 matrices, the matrix product AB is then given by

$$AB = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}. \quad (7.4.8)$$

Equations (7.4.7) and (7.4.8) immediately yield a divide-and-conquer algorithm (Strassen) based on expressing the product of two $n \times n$ matrices in terms of seven products of $(n/2) \times (n/2)$ matrices. The complexity of Strassen's algorithm clearly satisfies the recurrence relation

$$T(n) = 7T\left(\frac{n}{2}\right), \quad n > 1, \quad \text{init.cond. } T(1) = 1. \quad (7.4.9)$$

By unwinding (7.4.9) we see that see also the discussion following (3.3.12 in Chapter 3) $T(n) \in \Theta(n^{\log_2 7})$. Since $\log_2 7$ is approximately 2.81, we now have an algorithm for matrix multiplication whose $\Theta(n^{\log_2 7})$ complexity is a significant improvement over the $\Theta(n^3)$ complexity of the classical algorithm for matrix multiplication.

Note that Strassen's identities (7.4.3) and (7.4.4) involve a total of 18 additions (or subtractions). Winograd discovered the following set of identities, which leads to a method of multiplying 2×2 matrices using only 15 additions or subtractions, but still doing only seven multiplications.

$$\begin{aligned} m_1 &= (a_{10} + a_{11} - a_{00})(b_{11} - b_{01} + b_{00}) \\ m_2 &= a_{00}b_{00} \\ m_3 &= a_{01}b_{10} \\ m_4 &= (a_{00} - a_{10})(b_{11} - b_{01}) \\ m_5 &= (a_{10} + a_{11})(b_{01} - b_{00}) \\ m_6 &= (a_{01} - a_{10} + a_{00} - a_{11})b_{11} \\ m_7 &= a_{11}(b_{00} - b_{11} - b_{01} + b_{10}) \end{aligned} \quad (7.4.10)$$

The matrix product AB is then given by

$$AB = \begin{bmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{bmatrix}. \quad (7.4.11)$$

Although (7.4.10) and (7.4.11) involve a total of 24 additions or subtractions, it can be verified that a total of only 15 distinct additions or subtractions are needed. Thus, we obtain an improvement of three less additions or subtractions over Strassen's identities, but still use only seven multiplications.

7.5 The Discrete Fourier Transform

One of the most celebrated divide-and-conquer algorithms is the Fast Fourier Transform (*FFT*) implementing the Discrete Fourier Transform (*DFT*). The (possibly confusing) name *FFT* was coined by Cooley and Tukey and in their 1965 seminal paper describing

the algorithm. Note that the *FFT* is *not* a transformation, but rather is an efficient algorithm that makes the *DFT* useful in practice. In fact, the *FFT* has been referred to as the “most important numerical algorithm in our lifetime.” The *DFT* has a wide range of important applications to such fields as signal processing, image processing (MPEG), music processing (MP3), medical imaging (CT, MRI), weather prediction and statistics, coding theory, and a host of others. We begin here with a discussion of the *DFT* as applied to polynomials show how the *FFT* can be used to obtain an $O(n \log n)$ algorithm for polynomial multiplication. Then we show how the *FFT* can be used to multiply two polynomials efficiently by transforming the problem to one of simply multiplying n numbers.

7.5.1 The Discrete Fourier Transform of a Polynomial

The Discrete Fourier Transform DFT_ω of a polynomial of degree n is defined relative to a primitive n^{th} root of unity ω . A primitive n^{th} root of unity (over the field of complex numbers) is a complex number ω such that $\omega^n = 1$ and $\omega^k \neq 1$ for $0 < k < n$ (see Appendix A).

Definition 7.5.1

Given a primitive n^{th} root of unity ω , the Discrete Fourier Transform DFT_ω transforms a polynomial $P(x) = a_{n-1}x^{n-1} + \dots + a_1x + a_0$ with *real* or *complex* coefficients into the polynomial $b_{n-1}x^{n-1} + \dots + b_1x + b_0$, where $b_i = P(\omega^i)$, $i = 0, 1, \dots, n-1$. That is,

$$DFT_\omega(P(x)) = P(\omega^{n-1})x^{n-1} + \dots + P(\omega)x + P(1). \quad (7.5.1)$$

For convenience, we (equivalently) regard DFT_ω as transforming the coefficient array $\mathbf{a} = [a_0, a_1, \dots, a_{n-1}]$ of the polynomial $P(x)$ into the coefficient array $\mathbf{b} = [b_0, b_1, \dots, b_{n-1}]$ of the polynomial $DFT_\omega(P(x))$. Then \mathbf{a} and \mathbf{b} are related by the following matrix equation.

$$\begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & \omega^{n-1} & (\omega^{n-1})^2 & \cdots & (\omega^{n-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} \quad (7.5.2)$$

We denote the $n \times n$ matrix in (7.5.2) by $F_\omega = (f_{ij})_{n \times n}$, so that

$$f_{ij} = \omega^{(i-1)(j-1)}, \quad i, j = 1, \dots, n. \quad (7.5.3)$$

Remarks. In extending the applications of the *DFT* from polynomials to myriad and diverse domains, the numbers a_0, a_1, \dots, a_{n-1} are typically data points sampled over an appropriate time or space interval. For example, the amplitude of a sound wave might be measured at discrete points in a time interval, and the *DFT* of these values computed to obtain the contributions to the wave of the sinusoidal waves whose frequencies are

determined by $1, \omega, \dots, \omega^{n-1}$. Thus, the *DFT* transforms the time (or space) domain into the frequency (and phase) domain. The n (complex) numbers output by the *DFT* determine both the amplitudes and phases of these component sinusoidal waves. These n values can then be transmitted over an appropriate medium and the original wave (or a good enough approximation to it) can be reconstructed from these values (assuming an appropriately large value of n). We discuss this further in the Closing Remarks at the end of this chapter.

7.5.2 The Fast Fourier Transform

A straightforward way to compute $DFT_\omega(P(x))$ is simply to evaluate $P(x)$ at the points $1, \omega, \dots, \omega^{n-1}$ using Horner's rule, yielding an $O(n^2)$ algorithm. In fact, for evaluating $P(x)$ at any old set of n points, repeated applications of Horner's rule is the best that we can do. However, the n points, $1, \omega, \dots, \omega^{n-1}$ are very special beyond the mere fact that they are powers of a given number ω . The special nature of these points can be utilized to design an $O(n \log n)$ divide-and-conquer algorithm called the *Fast Fourier Transform (FFT)* for computing $DFT_\omega(P(x))$. Indeed, we now motivate the choice of points $1, \omega, \dots, \omega^{n-1}$ by showing how a divide-and-conquer algorithm to evaluate $P(x)$ at n points might proceed if the n points have certain simple relationships with one another. We assume in the following that $n = 2^k$ for a given integer $k \geq 1$.

Suppose we choose the n points so that second half of the points are the negatives of the other half,

$$z_0, z_1, \dots, z_{n/2-1}, -z_0, -z_1, \dots, -z_{n/2-1}. \quad (7.5.4)$$

Evaluating a polynomial $P(x) = a_{n-1}x^{n-1} + \dots + a_1x + a_0$ at such a set of points requires basically half the number of multiplications required for an arbitrary set of n points. To see this, we merely have to write $P(x)$ as the sum of two polynomials of degree $n/2 - 1$ by gathering up the even and odd powers of x .

$$P(x) = Even(x^2) + xOdd(x^2) \quad (7.5.5)$$

where $Even(x) = a_{n-2}x^{n/2-1} + \dots + a_2x + a_0$ and $Odd(x) = a_{n-1}x^{n/2-1} + \dots + a_3x + a_1$. Now if we replace x by $-x$ in (7.5.5), we obtain

$$P(-x) = Even(x^2) - xOdd(x^2). \quad (7.5.6)$$

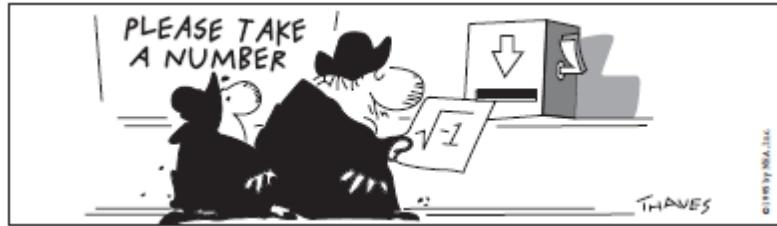
Thus, to evaluate $P(x)$ at the n points given by (7.5.4), we need only evaluate the two polynomials *Even* and *Odd* at the $n/2$ points $(z_0)^2, (z_1)^2, \dots, (z_{n/2-1})^2$, and then perform the $n/2$ additions, $n/2$ subtractions, and n multiplications as described in (7.5.5) and (7.5.6). Since we have divided our evaluation problem into two problems of size $n/2$, together with a combine step of $O(n)$ complexity, this looks like a promising start to a divide-and-conquer strategy that satisfies the familiar recurrence relation $t(n) = 2t(n/2) + O(n)$, and therefore yields an algorithm having $O(n \log n)$ complexity.

The same strategy can be applied to evaluate *Even* and *Odd* at $(z_0)^2, (z_1)^2, \dots, (z_{n/2-1})^2$ if we choose $z_0, z_1, \dots, z_{n/2-1}$ to satisfy the relation

$$(z_{n/4+j})^2 = -(z_j)^2, \quad j = 0, \dots, n/4 - 1. \quad (7.5.7)$$

Now (7.5.7) presents a *real* problem (not a *complex* problem, see Figure 7.2). More precisely while we cannot find *real* (nonzero) numbers satisfying (7.5.7), there is no problem finding *complex* numbers that do the job. We merely have to take

$$z_{n/4+j} = iz_j, \quad j = 0, \dots, \frac{n}{4} - 1, \quad i = \sqrt{-1}.$$



Taking a number that leads to numbers satisfying Formula (7.5.7)

Figure 7.2

In general, we continue the divide-and-conquer strategy at each step by dividing the range of the evaluation points in half (taking the first half of the points in the previous step), and with the new evaluation points being the squares of those in this new first half. What is needed to continue the strategy is that the points in the second half of the new range are the negatives of the points in the first half. More precisely, we need to find a set of points z_0, z_1, \dots, z_{n-1} with the property

$$(z_{n/2p+j})^p = -(z_j)^p, \quad j = 0, \dots, n/p - 1, \quad p = 2^q, \quad q = 0, \dots, k - 1, \quad n = 2^k. \quad (7.5.8)$$

Formula (7.5.9) is a bit of indices soup, but it merely quantifies the simple strategy described in words above.

If ω is a primitive n^{th} root of unity, then it follows from the properties described in Appendix A that the points $1, \omega, \dots, \omega^{n-1}$ satisfies the properties of (7.5.9). To see this analytically, if ω is a primitive n^{th} root of unity, it follows easily that $\omega^{n/2} = -1$, so that $\omega^{n/2+j} = \omega^{n/2}\omega^j = -\omega^j$, $j = 0, \dots, n/2 - 1$. Thus, (7.5.9) is satisfied for $p = 1$. Since ω^2 is a primitive $(n/2)^{\text{th}}$ root of unity, (7.5.9) also holds for $p = 1, \dots, n/2$. Note that when $\omega = e^{2\pi i/n}$ or $\omega = e^{-2\pi i/n}$, then (7.5.9) follows easily from the geometric interpretation of the multiplication of complex numbers (see Appendix A for the figures).

When writing the pseudocode for the algorithm *FFTRec*, it is convenient to use the coefficient array $\mathbf{a} = [a_0, a_1, \dots, a_{n-1}]$ to represent the polynomial $P(x) = a_{n-1}x^{n-1} + \dots + a_1x + a_0$. *FFTRec* evaluates $P(1), P(\omega), \dots, P(\omega^{n-1})$ by first splitting $P(x)$ into even- and odd-degree terms $Even(x) = a_{n-2}x^{n/2-1} + \dots + a_2x + a_0$ and $Odd(x) = a_{n-1}x^{n/2-1} + \dots + a_3x + a_1$. Then *FFTRec* recursively evaluates $Even(1), Even(\omega^2), \dots, Even((\omega^2)^{(n/2)-1})$ and $Odd(1), Odd(\omega^2), \dots, Odd((\omega^2)^{(n/2)-1})$. Finally, *FFTRec* uses (7.5.5) and (7.5.6) to evaluate $P(1), P(\omega), \dots, P(\omega^{n-1})$.

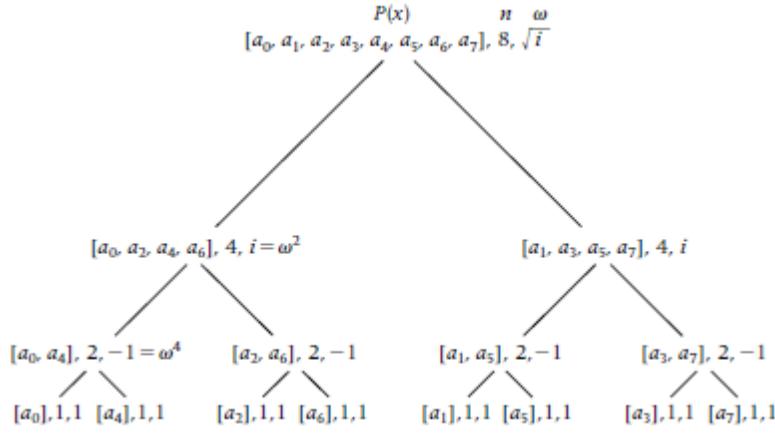
For *FFTRec* to be a valid recursive procedure, we must verify that it is recursively invoked with the proper type of input. It is immediate that when n is a power of two, so is $n/2$. As noted above, since ω is a primitive n^{th} root of unity and n is a power of two, ω^2 is a primitive $(n/2)^{\text{th}}$ root of unity. Thus, $n/2$ and ω^2 are appropriate input parameters for a recursive invocation of *FFTRec*. In our pseudocode for *FFTRec* and other procedures in this chapter, we avoid using i for a loop index, to save possible confusion with $i = \sqrt{-1}$.

However, we still find it convenient to use i for an integer when talking, for example, about the i^{th} leaf node in the tree of recursive calls to *FFTRec*.

```
procedure FFTRec( $a[0:n-1]$ ,  $n$ ,  $\omega$ ,  $b[0:n-1]$  recursive
Input:  $a[0:n-1]$  (an array of coefficients of the polynomial  $P(x) =$ 

$$a_{n-1}x^{n-1} + \dots + a_1x + a_0$$
)
 $n$  (a power of two) // $n = 2^k$ 
 $\omega$  (a primitive  $n^{\text{th}}$  root of unity)
Output:  $b[0:n-1]$  (an array of values  $b[j] = P(\omega^j)$ ,  $j = 0, \dots, n-1$ )
if  $n = 1$  then
     $b[0] \leftarrow a[0]$ 
else
    //divide into even-indexed and odd-indexed coefficients
    for  $j \leftarrow 0$  to  $n/2 - 1$  do
         $Even[j] \leftarrow a[2*j]$  // $[a_0, a_2, \dots, a_{n-2}]$ 
         $Odd[j] \leftarrow a[2*j + 1]$  // $[a_1, a_3, \dots, a_{n-1}]$ 
    endfor
    FFTRec( $Even[0:n/2-1], n/2, \omega^2, e[0:n/2-1]$ )
    FFTRec( $Odd[0:n/2-1], n/2, \omega^2, d[0:n/2-1]$ )
    //now combine according to (7.5.5) and (7.5.6)
    for  $j \leftarrow 0$  to  $n/2 - 1$  do
         $b[j] \leftarrow e[j] + \omega^j * d[j]$ 
         $b[j + n/2] \leftarrow e[j] - \omega^j * d[j]$ 
    endfor
endif
end FFTRec
```

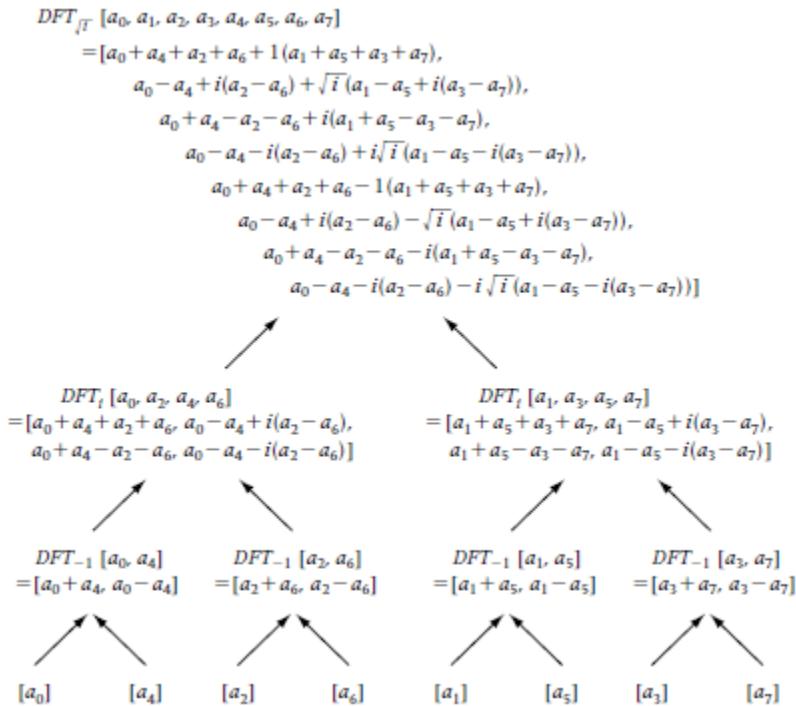
Figure 7.4 illustrates the tree of recursive calls to *FFTRec* resulting from an initial call with $n = 7$. As usual, the left (right) child of a node corresponds to the first (second) recursive call in the code. Each node in the tree lists the relevant **input** parameters for the given invocation. That is, we first list the input polynomial by listing its coefficient array, and we then list the current values of n and associated n^{th} root of unity for the given node.



Tree of recursive calls to *FFTRec* with $n = 8$

Figure 7.4

Resolution of the tree of recursive calls made by *FFTRec* with $n = 8$ is shown in Figure 7.5.



Resolutions of the recursive calls made by *FFTRec* with $n = 8$

Figure 7.5

7.5.3 An Iterative version of FFT

An iterative version *FFT* of *FFTRec* can be designed based on the bottom-up resolution of the tree T of recursive calls shown in Figure 7.5. Upon examining Figure 7.5, it is not immediately obvious for a general $n = 2^k$ which coefficient a_j corresponds to the i^{th} leaf node, $i = 0, \dots, n - 1$, in the tree of recursive calls for *FFTRec*. Fortunately, there is a simple procedure for computing the permutation $j = \pi_k(i)$ such that a_j is the coefficient corresponding to the i^{th} leaf note of T .

Proposition 7.5.1

Suppose $n = 2^k$, and $\pi_k: \{0, \dots, n - 1\} \Rightarrow \{0, \dots, n - 1\}$ is the permutation such that a_j is the coefficient corresponding to the i^{th} leaf node of T in the tree of recursive calls of *FFTRec* with input coefficient array $a[0:n - 1]$. Then the k -digit binary representation of $\pi_k(i)$ is obtained from the k -digit binary representation of i by simply reversing the digits, where leading zeros are included (if necessary).

To illustrate Proposition 7.5.1, suppose $k = 4$ ($n = 16$) and $i = 3$. The 4-digit binary representation of 3 is 0011, so that $\pi_k(3) = 1100$ in binary (in decimal, $\pi_k(3) = 12$).

Proof of Proposition 7.5.1

We prove Proposition 7.5.1 using induction on k .

Basis step: $k = 1$. Trivial.

Induction step: Assume that Proposition 7.5.3.1 is true for k . For $k + 1 = \log_2 n$, note that the leaves of the left subtree of the tree of recursive calls with input coefficients $a_0, a_1, \dots, a_{2^{k+1}-1}$, and n^{th} root of unit ω , consists of the coefficients $a_0, a_2, \dots, a_{2^{k+1}-2}$. Moreover, this left subtree corresponds exactly to the tree of recursive calls of *FFTRec* with input coefficients $\tilde{a}_0, \tilde{a}_1, \dots, \tilde{a}_{2^k-1}$, and $(n/2)^{\text{th}}$ root of unit ω^2 , where

$$\tilde{a}_i = a_{2i}, \quad i = 0, \dots, 2^k - 1. \quad (7.5.9)$$

It follows from (7.5.9) that

$$\pi_{k+1}(i) = 2\pi_k(i), \quad i = 0, \dots, 2^k - 1. \quad (7.5.10)$$

By induction hypothesis, for input coefficients \tilde{a}_i , $i = 0, \dots, 2^k - 1$, to *FFTRec*, the k -digit binary representation of $\pi_k(i)$ is obtained from the k -digit binary representation of i by simply reversing the digits. Using (7.5.12), the $(k + 1)$ -digit binary representation of $\pi_{k+1}(i)$ is obtained from $\pi_k(i)$ by adding a zero on the right. Since the $(k + 1)$ -digit binary representation of i is obtained by from the k -digit binary representation of i by adding a leading zero, we see that Proposition 7.5.1 holds for $\pi_{k+1}(i)$, $i = 0, \dots, 2^k - 1$. The proof

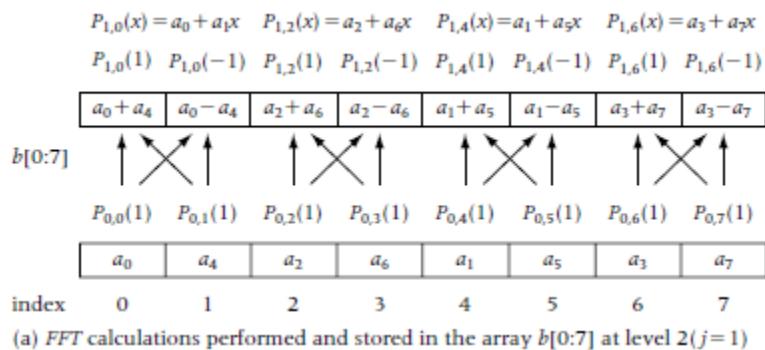
that Proposition 7.5.1 holds for $\pi_{k+1}(i)$, $i = 2^k, \dots, 2^{k+1} - 1$ is similar (using the right subtree) and is left as an exercise. ■

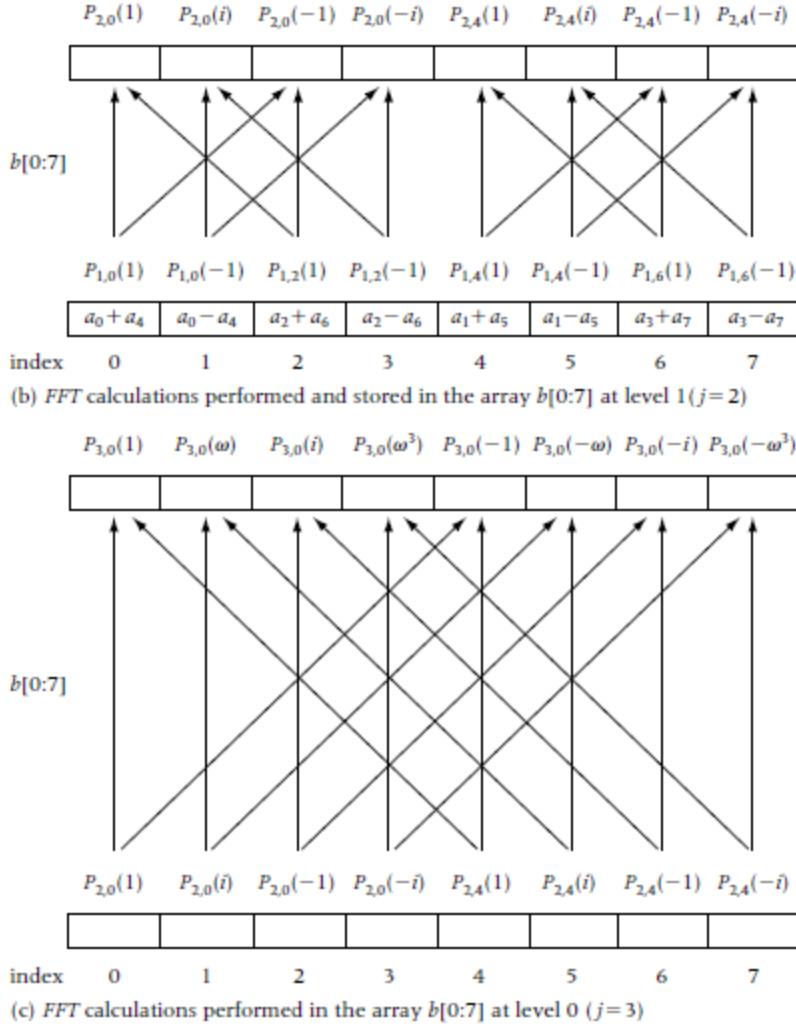
The pseudocode for a procedure $ReverseBinPerm(R[0:n-1])$, which computes $R[i] = \pi_k(i)$, $i = 0, \dots, n-1$, is easy to write. An iterative version of FFT begins by loading an array $b[0:n-1]$ with the values $b[i] = a_{\pi_k(i)} = a[R[i]]$, $i = 0, \dots, n-1$. The values $b[i]$ correspond to the leaves in the tree T of recursive calls to $FFTRec$, and the iterative version of FFT proceeds by resolving these calls level-by-level in a bottom-up fashion. With the aid of Figure 7.5, you might want to proceed directly to the pseudocode for FFT given at the end of this subsection. However, it is instructive to describe the polynomial evaluations that take place at each level. The leaves are at level k in T and correspond to evaluating the constant polynomials $P_{0,i}(x) = a_{\pi_k(i)}$, $i = 0, \dots, n-1$, at $(\omega^n)^0 = 1$. At level $k-j$, $j \in \{1, \dots, k\}$, the iterative version of FFT evaluates $n/2^j$ polynomials $P_{j,s}$, $s = 0, 2^j, \dots, n-2^j$ of degree 2^j-1 at the points $1, \omega^{n/2^j}, \dots, (\omega^{n/2^j})^{2^{j-1}-1}, -1, -\omega^{n/2^j}, \dots, -(\omega^{n/2^j})^{2^{j-1}-1}$. In this notation, the splitting of the polynomial $P_{j,s}$ into even and odd powers corresponds precisely to the two polynomials $P_{j-1,s}$ and $P_{j-1,s+2^{j-1}}$, respectively, at the next lower level in T . Hence, (7.5.5) and (7.5.6) become (see Figure 7.6)

$$P_{j,s}(x) = P_{j-1,s}(x^2) + xP_{j-1,s+2^{j-1}}(x^2), \quad (7.5.11)$$

$$P_{j,s}(-x) = P_{j-1,s}(x^2) - xP_{j-1,s+2^{j-1}}(x^2), \quad (7.5.12)$$

$$j = 1, \dots, k, \quad s = 0, 2^j, \dots, n-2^j, \quad x = 1, \omega^{n/2^j}, \dots, (\omega^{n/2^j})^{2^{j-1}-1}.$$





Level-by-level FFT computations for $n = 8$

Figure 7.6

The iterative version of FFT proceeds by computing (7.5.11) and (7.5.12) with three nested **for-do** loops, the outer loop controlled by $j = 1, \dots, k$, the middle loop controlled by $s = 0, 2^j, \dots, n - 2^j$, and the innermost loop controlled by $x = 1, \omega^{n/2^j}, \dots, (\omega^{n/2^j})^{2^{j-1}-1}$. The calculations at level $k-j$ begin with $b[0:n-1]$ containing the previously calculated values.

$$b[s+m] = P_{j-1,s}((\omega^{n/2^{j-1}})^m), \quad (7.5.13)$$

$$s = 0, 2^{j-1}, \dots, n - 2^{j-1}, \quad m = 0, 1, \dots, 2^{j-1} - 1$$

The values corresponding to level $k-j$ are then calculated by first using an auxiliary array $Temp[0:n-1]$ to receive the results of calculations in the right-hand sides of (7.5.11) and (7.5.12).

$$Temp[s+m] = b[s+m] + (\omega^{n/2^j})^m b[s+m+2^{j-1}], \quad (7.5.14)$$

$$Temp[s + m + 2^{j-1}] = b[s + m] - (\omega^{n/2^j})^m b[s + m + 2^{j-1}], \quad (7.5.15)$$

$$j = 1, \dots, k, \quad s = 0, 2^j, \dots, n - 2^j, \quad m = 0, 1, \dots, 2^j - 1$$

Formula (7.5.15) follows from (7.5.12) and the fact that $-b[s + m] = b[s + m + 2^{j-1}]$. Then the results in $Temp[0:n - 1]$ are copied back to $b[0:n - 1]$.

The level-by-level bottom-up calculations are illustrated in Figure 7.6, where we show using arrows that the new value of $b[s + m]$ is determined from the previously calculated values of $b[s + m]$ and $b[s + m + 2^{j-1}]$. Due to space limitations, we only show the initial values of $b[0:7]$ and its values after one iteration. However, all the values at each level can be determined by examining Figure 7.5.

The pseudocode for the iterative version *FFT* is based on the bottom-up strategy described above.

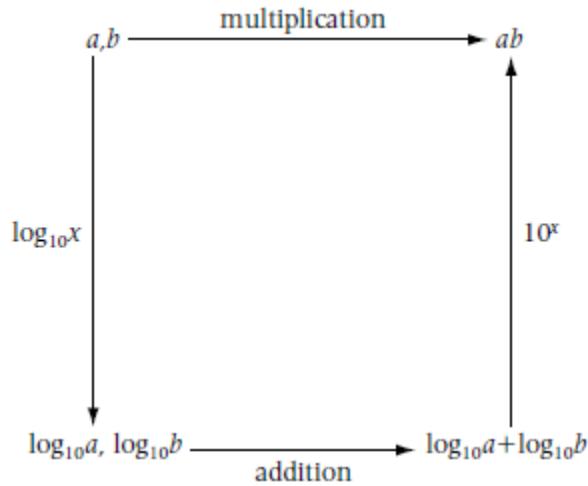
```
procedure FFT( $a[0:n - 1]$ ,  $n$ ,  $\omega$ ,  $b[0:n - 1]$ )
Input:  $a[0:n - 1]$  (an array of coefficients of the polynomial  $P(x) =$ 

$$a_{n-1}x^{n-1} + \dots + a_1x + a_0$$
)
 $n$  (a positive integer) //  $n = 2^k$ 
 $\omega$  (a primitive  $n^{\text{th}}$  root of unity)
Output:  $b[0:n - 1]$  (an array of values  $b[j] = P(\omega^j)$ ,  $j = 0, \dots, n - 1$ )
call ReverseBinPerm( $R[0:n - 1]$ )
for  $j \leftarrow 0$  to  $n - 1$ 
     $b[j] \leftarrow a[R[j]]$ 
endfor
for  $j \leftarrow 1$  to  $k$  do
    for  $s \leftarrow 0$  to  $n - 2^j$  by  $2^j$  do
        for  $m \leftarrow 0$  to  $2^{j-1} - 1$ 
             $Temp[s + m] \leftarrow b[s + m] + (\omega^{n/2^j})^m * b[s + m + 2^{j-1}]$ 
             $Temp[s + m + 2^{j-1}] \leftarrow b[s + m] - (\omega^{n/2^j})^m * b[s + m + 2^{j-1}]$ 
        endfor
    endfor
    for  $j \leftarrow 0$  to  $n - 1$  do
         $b[j] \leftarrow Temp[j]$ 
    endfor
endfor
end FFT
```

7.5.4 Transforming the Problem Domain

In secondary school mathematics, you were introduced to the idea of transforming problems into equivalent problems that are simpler or easier to solve. For example, it is often convenient to deal with large numbers by transforming them using logarithms to a certain base, say, base 10. The problem of multiplying two large numbers a and b is then transformed into the simpler but equivalent problem of adding their logarithms and then

using the inverse transformation (exponentiation) to compute the given product. The idea behind this transformation can be nicely captured by the commutative diagram shown in Figure 7.7.



Commutative diagram for transforming multiplication via logarithms

Figure 7.7

The diagram given in Figure 7.7 is called commutative since the product ab can be computed either directly by following the top arrow or indirectly by following the three arrows along the sides and bottom of the diagram. The mathematical expression of this commutativity is simply the equation

$$ab = 10^{\log_{10}a + \log_{10}b}$$

The utility of this transformation resides in the fact that addition can be carried out more quickly than multiplication. It is precisely to make the above transformations useful in practice that extensive tables of logarithms and antilogs have been generated.

The general idea behind transforming a given problem can be captured by the generic commutative diagram shown in Figure 7.8. In order for this transformation to be worthwhile, the transformation to the new problem domain and its inverse must both be done more efficiently than solving the problem in the original domain. Also, the problem in the transformed domain must be easier to solve than the problem in the original domain.

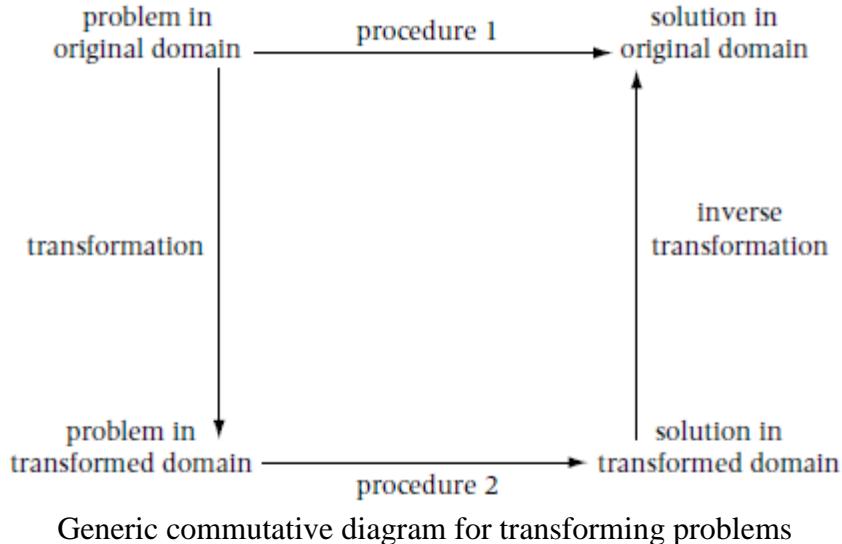


Figure 7.8

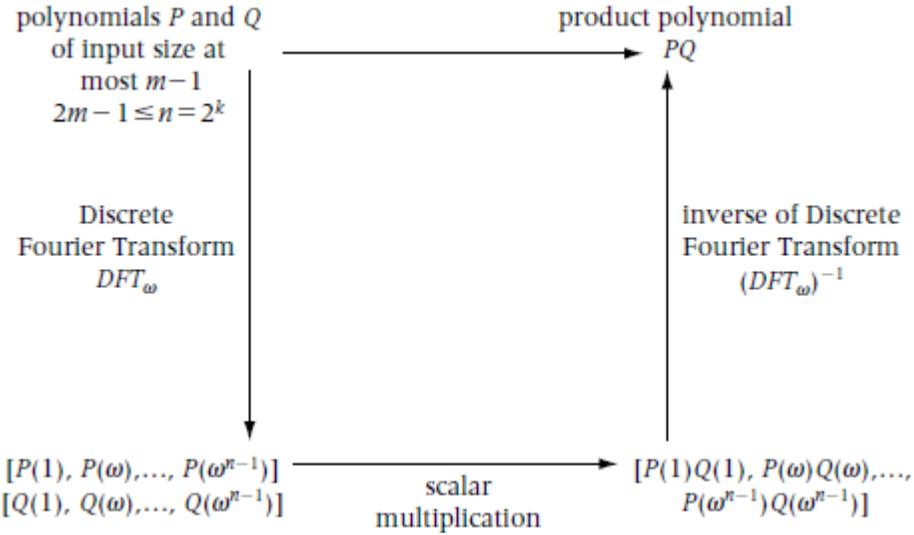
There is a host of important applications, such as signal and image processing, where the Discrete Fourier Transform (*DFT*) can be used to transform the problem domain into an equivalent, but (algorithmically) simpler, problem domain. However, in order for this transformation to be effective, the *DFT* and its inverse need to be computed efficiently. As we will see, the inverse of a *DFT* is actually (except for a multiplicative factor) just another *DFT*! Fortunately, the *FFT* provides an efficient algorithm to compute *DFTs*, and therefore efficiently transform the problem to the new domain and transform back to the original domain. The problem that we will use to illustrate *DFT* transformation of the domain is (symbolic) polynomial multiplication. This problem will be transformed into the simpler problem on multiplying n numbers.

7.5.5 The Inverse Fourier Transform and Fast Polynomial Multiplication

We have mentioned that the *FFT* can be used to design an $O(n \log n)$ algorithm for polynomial multiplication. Of course, *FFT* worked on polynomials whose input size is a power of two. The product of two polynomials both of input size m is a polynomial of input size $n = 2m - 1$. Thus, for convenience in discussing polynomial multiplication, we assume that n is a power of two and that we are multiplying two polynomials of input size m , where $n = 2m - 1$. In practice, when multiplying any two polynomials we can arrange for these conditions by adding leading terms with zero coefficients, if necessary.

Figure 7.9 shows how the Fourier Transform can be utilized to obtain the product polynomial $P(x)Q(x)$. We first apply the Fourier Transform to find the coefficient arrays $[P(1), P(\omega), \dots, P(\omega^{n-1})]$ and $[Q(1), Q(\omega), \dots, Q(\omega^{n-1})]$ of both $P(x)$ and $Q(x)$. Then we simply compute the coefficient array of the product polynomial $P(x)Q(x)$ by forming the n scalar products $P(1)Q(1), P(\omega)Q(\omega), \dots, P(\omega^{n-1})Q(\omega^{n-1})$. Thus by following the left side and bottom of the commutative diagram in Figure 7.9 we have actually computed the Fourier Transform of the product polynomial $P(x)Q(x)$. Hence, to recover $P(x)Q(x)$ we need to be able to perform the inverse to the Fourier Transform. The inverse

certainly exists, since it amounts to finding the polynomial interpolating the n points $(1, P(1)Q(1)), (\omega, P(\omega)Q(\omega)), \dots, (\omega^{n-1}, P(\omega^{n-1})Q(\omega^{n-1}))$.



Commutative diagram for computing product of two polynomials using DFT_ω

Figure 7.9

The left side of the diagram in Figure 7.9 can be computed using FFT with $O(n\log n)$ complexity. The bottom of the diagram can be computed with $O(n)$ complexity. However, the most straightforward interpolation algorithms have complexity $O(n^2)$. Hence, to arrive at an $O(n\log n)$ algorithm for polynomial multiplication we need to look for an $O(n\log n)$ algorithm to compute the inverse Discrete Fourier Transform DFT_ω^{-1} . Fortunately, we have the following useful key fact about the inverse Discrete Fourier Transform.

Key Fact

To compute the inverse of a Discrete Fourier Transform we simply need to compute another Discrete Fourier Transform.

The key fact results from the following elegant formula for the inverse namely $DFT_{\omega^{-1}}$:

$$DFT_{\omega^{-1}} = \left(\frac{1}{n} \right) DFT_{\omega^{-1}}. \quad (7.5.16)$$

To verify (7.5.16), it is useful to utilize the array and matrix notation discussed earlier for the Discrete Fourier Transform DFT_ω . Then (7.5.18) is equivalent to verifying that the inverse of the matrix A_ω defined by (7.5.19) is given by the formula

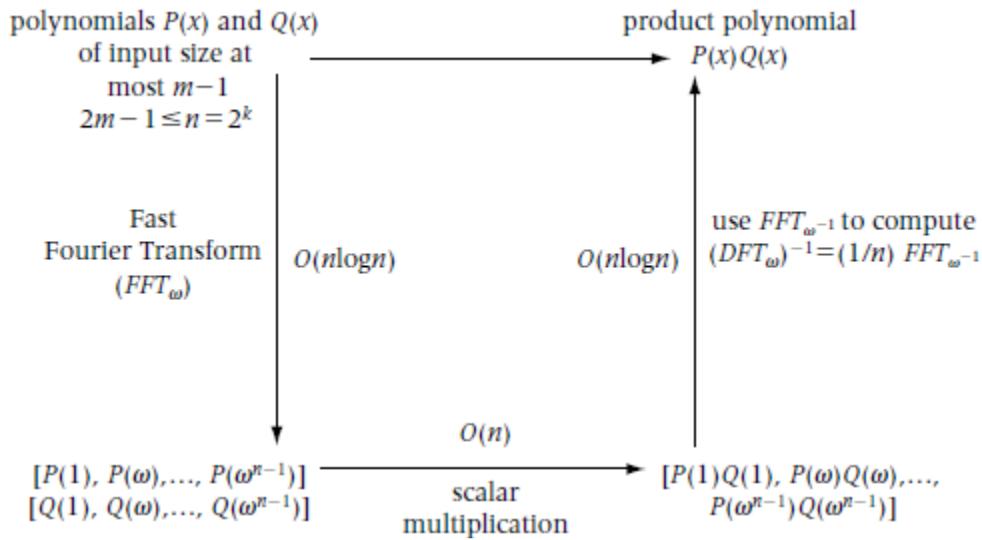
$$F_\omega^{-1} = \left(\frac{1}{n} \right) f_{\omega^{-1}}. \quad (7.5.17)$$

Equivalently, we must verify that

$$F_{\omega}^{-1} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & (\omega^{-2})^{n-1} & \dots & (\omega^{-(n-1)})^{n-1} \end{bmatrix} \quad (7.5.18)$$

We leave the proof of (7.5.10) to the exercises.

The commutative diagram in Figure 7.10 summarizes the application of *FFT* for $O(n \log n)$ polynomial multiplication. In Figure 7.10, we denote the application of *FFT* with input parameter ω by FFT_{ω} .



Commutative diagram summarizing $O(n \log n)$ polynomial multiplication using FFT_{ω}

Figure 7.10

We illustrate the actions of the commutative diagram in Figure 7.10 to compute the product of the polynomials $P(x) = x^3 - x + 2$ and $Q(x) = 2x^2 - 1$. The product polynomial has degree 5, so we take $n = 7$. Then, $\omega = \sqrt{i} = (1+i)/\sqrt{2}$. Using *FFT* yields the evaluations

$$\begin{aligned} [P(1), P(\omega), \dots, P(\omega^7)] &= [2, 2 - \omega + \omega^3, 2 - 2i, 2 + \omega - \omega^3, 2, 2 + \omega - \omega^3, \\ &\quad 2 + 2i, 2 - \omega + \omega^3] \\ [Q(1), Q(\omega), \dots, Q(\omega^7)] &= [1, 2i - 1, -3, -2i - 1, 1, 2i - 1, -3, -2i - 1]. \end{aligned}$$

Next we form the pointwise products $P(i)Q(i)$, $i = 1, \omega, \dots, \omega^7$ and acquire the coefficient array for $DFT_\omega(P(x)Q(x))$ given by

$$[2, 4i - 3\omega^3 - \omega - 2, 6i - 6, -4i - \omega^3 - 3\omega - 2, 2, 4i + 3 + \omega - 2, -6i - 6, \\ -4i + \omega^3 + 3\omega - 2]. \quad (7.5.19)$$

To complete the illustration of computing the product polynomial $P(x)Q(x)$ using the commutative diagram, we use *FFT* to compute $(DFT_\omega)^{-1} = (1/8) DFT_{\omega^{-1}}$ for the polynomial (coefficient array) given by (7.5.21). This yields the coefficient array $[-2, 1, 4, -3, 0, 2, 0, 0]$ of the product polynomial

$$P(x)Q(x) = 2x^5 - 3x^3 + 4x^2 + x - 2.$$

7.6 Two Classical Problems in Computational Geometry

We now use the divide-and-conquer technique to solve two fundamental problems in computational geometry, the *closest-pair* problem, and the *convex hull* problem. While both problems can be posed for points in Euclidean d -dimensional space for any $d \geq 1$, we limit our discussion of the solution to these problems to the line and the plane ($d = 1$ and 2, respectively).

7.6.1 The Closest-Pair Problem

In the closest-pair problem, we are given n points P_1, \dots, P_n in Euclidean d -space, and we wish to determine a pair of points P_i and P_j such that the distance between P_i and P_j is minimized over all pairs of points drawn from P_1, \dots, P_n . Of course, a brute-force quadratic complexity algorithm solving this problem is obtained by simply computing the distance between each of the $n(n - 1)/2$ pairs and recording the minimum value so computed. Actually to avoid round-off problems associated with taking square roots, it would be best to work with the square of the distance, which we assume throughout the discussion (and which, by abuse of language and notation, we still refer to as simply the distance d).

Using divide-and-conquer, we now design a $O(n \log n)$ algorithm for the closest-pair problem. This turns out to be order optimal, since there is a $\Omega(n \log n)$ lower bound for the problem. We describe the algorithm informally, since the actual pseudocode, while straightforward, is somewhat messy to fully describe.

To motivate the solution to the problem in the plane, we first consider the problem for points on the real line. Of course, we can solve the problem on the line by sorting the points using an $O(n \log n)$ algorithm, and then simply making a linear scan of these sorted points. However, this method does not generalize to the plane. To find a method that does generalize, let m be the median value of the n points. Using the algorithm *Select2* described in Section 7.3, m can be computed in linear time. Divide the points x_1, \dots, x_n into two subsets X_1, X_2 of equal size, those less than or equal to the median, and the remaining points (a linear scan determines the division). Let d_1 and d_2 be the minimum distances between pairs of points in X_1 and X_2 , respectively. Now either d

$= \min\{d_1, d_2\}$ is also the smallest distance between any pair drawn from x_1, \dots, x_n , or there is a pair $x_i \in X_1$ and $x_j \in X_2$ having a strictly smaller distance than d . However, it is easy to check (exercise) that the interval $(m - d, m]$ contains at most one point of X_1 , and the interval $(m, m + d]$ contains at most one point of X_2 . Hence, in linear time it can be determined if the only possible pair have closer distance than d actually exists. Thus, for $n = 2^k$ the complexity $W(n)$ of the algorithm satisfies the recurrence

$$W(n) = 2W(n/2) + n, \quad \text{init. cond. } W(1) = 0,$$

which unwinds to yield $W(n) \in O(n \log n)$.

The above divide-and-conquer solution for points on the real line generalizes naturally to a divide-and-conquer solution in the plane by dividing the n points $(x_1, y_1), \dots, (x_n, y_n)$ into sets X_1, X_2 on either side of line $x = m$, where m is the median of the x -coordinates of the points. The sets X_1, X_2 can be determined with $O(n \log n)$ complexity by sorting the points by their x -coordinates. We then recursively find the minimum distances d_1 and d_2 between pairs of points in the sets X_1 and X_2 , respectively. Now again, either $d = \min\{d_1, d_2\}$ is also the smallest distance between any pair drawn from $(x_1, y_1), \dots, (x_n, y_n)$, or there is a pair $(x_i, y_i) \in X_1$ and $(x_j, y_j) \in X_2$ having a strictly smaller distance than d . Also, if there is such a pair, then (x_i, y_i) lies in the strip S_1 determined by the lines $x = m - d$ and $x = d$, whereas (x_j, y_j) lies in the strip S_2 determined by the lines $x = d$ and $x = m + d$. However, unlike the case for the line, we can no longer be sure that there is at most a single pair of points to examine. Indeed, *all* the points $(x_1, y_1), \dots, (x_n, y_n)$ might be in the strip $S = S_1 \cup S_2$ between the lines $x = m - d$ and $x = m + d$, so that we would have to examine a quadratic number of pairs, which is no better than the brute force solution! However, the following proposition, whose proof we leave as an exercise, comes to our rescue.

Proposition 7.6.1 Consider a rectangle R in the plane of width d and height $2d$. There can only be at most six points in R such that the distance between each pair of these points is at least d . \square

The following Key Fact follows from Proposition 7.6.1 and the definition of d .

Key Fact

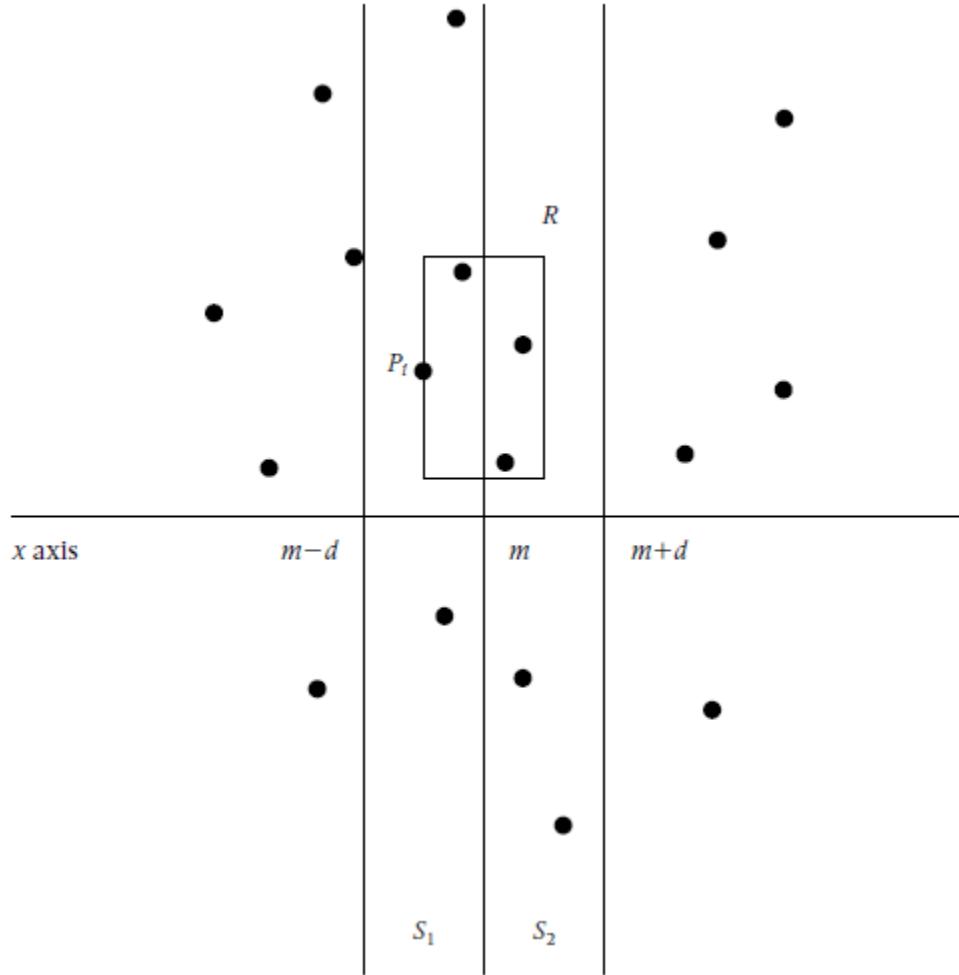
For each point $(x, y) \in X_1 \cap S_1$ there are at most five points in $X_2 \cap S_2 \cap R$, where R is the rectangle with corner points $(x, y - d), (x, y + d), (x + d, y - d), (x + d, y + d)$ (see Figure 7.11).

The key fact allows us to check less than $5n$ pairs to determine whether or not there is a pair $(x_i, y_i) \in X_1 \cap S_1$ and $(x_j, y_j) \in X_2 \cap S_2$ having distance less than d . To see this, note that we can require our recursive calls determining X_1, d_1 , and X_2, d_2 , to return X_1 and X_2 sorted by their y -coordinates, which then can be merged using no more than $n - 1$ comparisons by the procedure *Merge* discussed in Chapter 2. Thus, as we scan through the points in $X_1 \cap S_1$ in increasing order of their y -coordinates, a corresponding pointer can also scan the points in $X_2 \cap S_2$ in (slightly oscillatory) increasing order of their y -

coordinates checking at most $5n$ pairs. More precisely, suppose P_1, \dots, P_m (respectively, Q_1, \dots, Q_k) are the points in $X_1 \cap S_1$ (respectively, in $X_2 \cap S_2$). We first scan $X_2 \cap S_2$ until we find a point (if any) such that d added to its y -coordinate is at least as large as P_1 's y -coordinate. If we find such a point Q_i , then we leave a pointer Q at Q_i , and using Proposition 7.6.1 we need only check Q_i and the next four points $X_2 \cap S_2$ in order to see if d needs updating. We then move to point P_2 , and resume the scan of $X_2 \cap S_2$ starting at Q_i , this time looking for a point whose y -coordinate is at least as large as P_2 's y -coordinate. We repeat this process until all of $X_1 \cap S_1$ is scanned, and less than $5n$ pairs of points will be examined for updates to the current smallest distance. Since we make most $n - 1$ comparisons when merging X_1 and X_2 , we see that the combine step in our divide-and-conquer algorithm performs less than $6n$ comparisons altogether. Hence, the worst case $W(n)$ of the algorithm satisfies

$$W(n) < 2W(n/2) + 6n, \quad \text{init. cond. } W(1) = 0,$$

which shows that $W(n) \in O(n \log n)$.



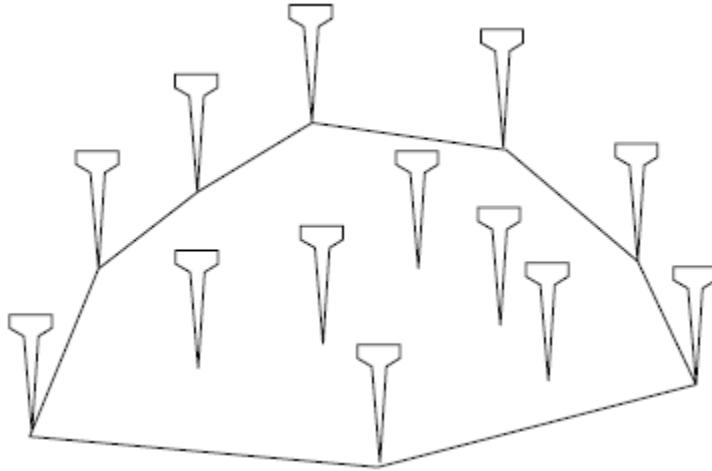
For $P_i \in X_1 \cap S_1$, rectangle R of width d and height $2d$ is shown where distances from P_i to points in $X_2 \cap S_2 \cap R$ need to be checked

Figure 7.11

When implementing the above algorithm, there are various degenerate cases that have to be handled. For example, it might happen that some (or even all) of the points are on the line $x = m$. We leave the implementation details to the exercises.

7.6.2 The Convex Hull Problem

Given any subset S of the Euclidean plane, S is called convex if for each pair of points $P_1, P_2 \in S$, the line segment joining P_1 and P_2 lies entirely within the set S . Given a set of n points $(x_1, y_1), \dots, (x_n, y_n)$ in the plane, the *convex hull* of these points is the smallest convex set containing them. In other words, the convex hull of the points is contained in any convex set containing the points. There is a nice physical interpretation of the convex hull. Consider placing pegs (or golf tees) at each of the n points $(x_1, y_1), \dots, (x_n, y_n)$. Stretching a small rubber band about the entire set of points and releasing the band determines the boundary of the convex hull (see Figure 7.12).



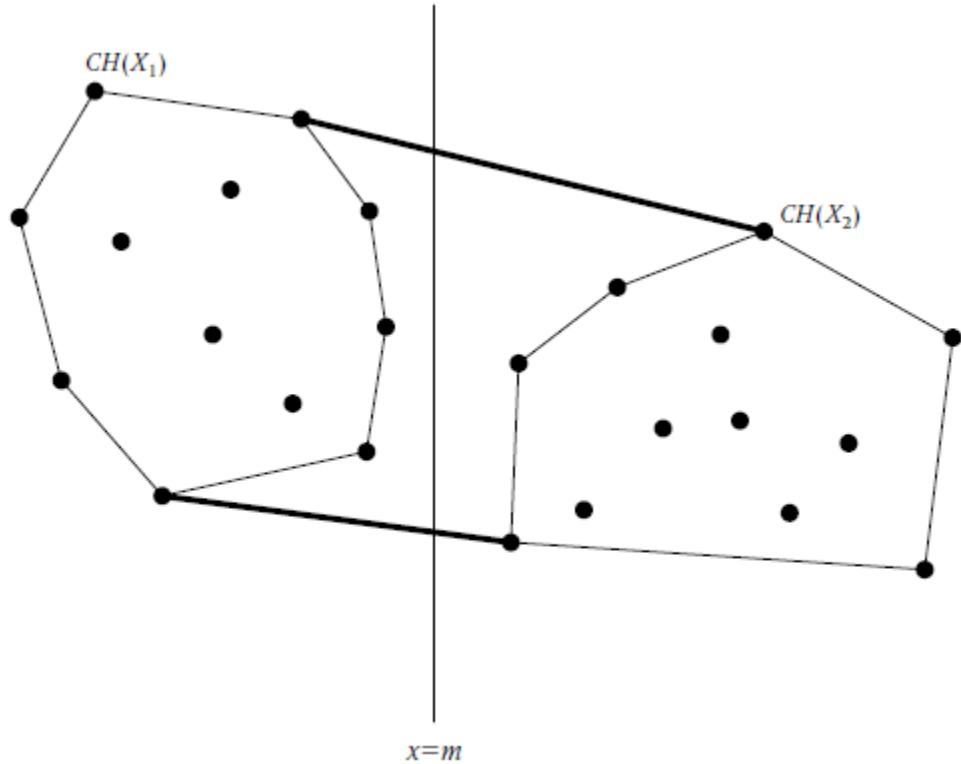
A small rubber band stretched and released around pegs at points in the plane determines convex hull of these points

Figure 7.12

Given the points $(x_1, y_1), \dots, (x_n, y_n)$ in the plane, the convex hull problem is to determine a subset of these points $P_1, \dots, P_k, P_{k+1} = P_1$ such that the boundary of the convex hull of the points $(x_1, y_1), \dots, (x_n, y_n)$ consists of the line segments joining P_i and P_{i+1} , $i = 1, \dots, k$. We assume that no three consecutive points in the (circular) list $P_1, \dots, P_k, P_{k+1} = P_1$ are collinear.

The divide-and-conquer algorithm for computing the convex hull that we now describe begins in the same way as the closest pair algorithm; namely, we divide the n points $(x_1, y_1), \dots, (x_n, y_n)$ into sets X_1, X_2 on either side of line $x = m$, where m is the median of the x -coordinates of the points. We then recursively determine the convex hulls $CH(X_1)$ and $CH(X_2)$, of X_1 and X_2 , respectively. The initial condition for the recursion is

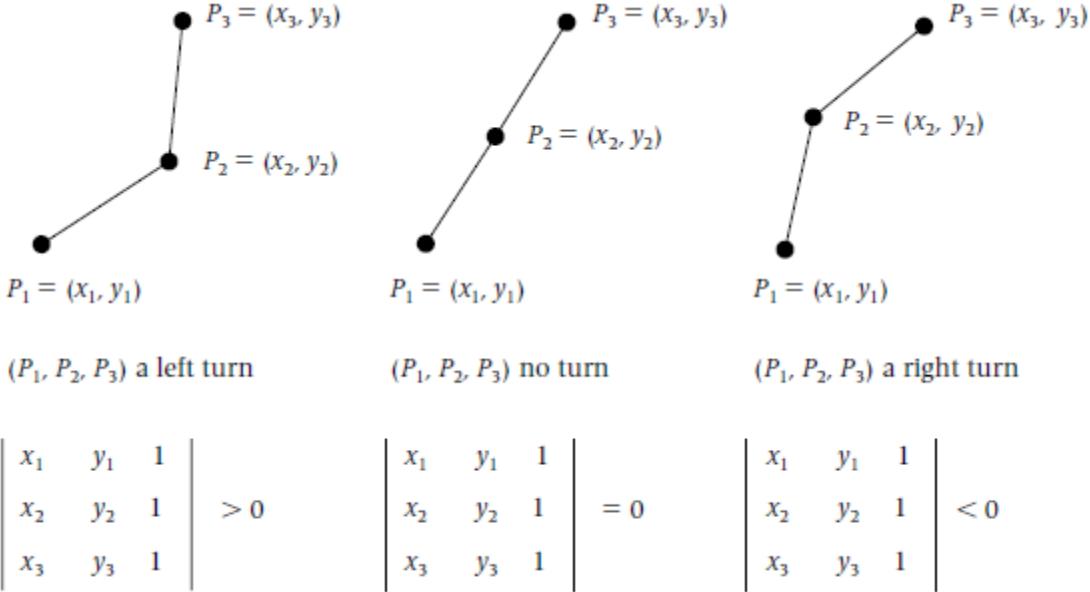
a set of one, two, or three points, since the convex hull in these cases is a point, a line segment connecting the two points, and a triangle connecting the three points (unless they are collinear), respectively. It then becomes a question of how to merge the convex hulls $CH(X_1)$, $CH(X_2)$ into the convex hull $CH(X_1 \cup X_2)$. The answer is to determine the upper and lower support line segments of $CH(X_1) \cup CH(X_2)$, as illustrated in Figure 7.13.



Upper and lower support line segments of $CH(X_1) \cup CH(X_2)$

Figure 7.13

In order to determine the support line segments, we introduce the notion of a turn determined by an ordered triple of points (P_1, P_2, P_3) in the plane. We have a left turn, no turn, or a right turn, respectively, as determined by the determinant shown in Figure 7.14. This determinant corresponds to twice the “signed area” determined by the triple of points (P_1, P_2, P_3) .



The indicated determinant determines whether the triple (P_1, P_2, P_3) is a left turn, no turn, or a right turn; that is, the turn depends on whether this determinant is positive, zero, or negative, respectively.

Figure 7.14

We now use the notion of turns to determine the upper support line segment. Suppose $CH(X_1)$ is given by $P_1, \dots, P_k, P_{k+1} = P_1$ and $CH(X_2)$ is given by $Q_1, \dots, Q_m, Q_{m+1} = Q_1$ where both sequences are in clockwise order (that is, you make right turns as you pass through the sequences). Let P_i be the point in X_1 with largest x -coordinate (if there are two such points, take the point with the smaller y -coordinate). Consider the sequence of turns determined by (P_i, Q_j, Q_{j+1}) , $j = 1, \dots, m$, where we set $Q_{m+1} = Q_1$ for convenience. Then the right-hand endpoint of the upper support line segment is the point Q_r where the turns go from left turns or no turn to a right turn; that is, (P_i, Q_{r-1}, Q_r) is a left turn or no turn, and (P_i, Q_r, Q_{r+1}) is a right turn. Now consider the sequence of turns (Q_r, P_j, P_{j-1}) , $j = 1, \dots, k$, where we set $P_0 = P_m$ for convenience (that is, we go around $CH(X_1)$ in counter-clockwise order). Then the left-hand endpoint of the upper support line segment is the point P_s where the turns go from right turns or no turn to a left turn. This determines the upper support segment. The lower support segment is obtained similarly.

Given the two support line segments, it is a simple matter to eliminate the points in $CH(X_1) \cup CH(X_2)$ that are not in $CH(X_1 \cup X_2)$, and to return the sequence of points in $CH(X_1 \cup X_2)$ in clockwise order. It is also easy to verify that the algorithm has $O(n \log n)$ complexity. As with the closest pair problem, when implementing the algorithm, various degenerate cases have to be handled. We leave the implementation details to the exercises.

There are other $O(n \log n)$ algorithms for determining the convex hull that are not based on divide-and-conquer. Two of the most commonly used of these are the Graham scan and the Jarvis march. We refer you to the references for a discussion of these and other convex hull algorithms. We point out there is a $\Omega(n \log n)$ lower bound for the

convex hull problem in the plane, since sorting can be reduced to this problem (see Exercise 7.32). Hence, the above divide-and-conquer algorithm, and the algorithms of Graham and Javis, all have optimal order of complexity.

7.7 Closing Remarks

Since Strassen's result for matrix multiplication appeared in 1970, researchers have been trying to improve the basic method. Identities are sought that perform fewer multiplications when multiplying matrices of a certain fixed size m . Then this reduction is used as the basis of a divide-and-conquer algorithm using decomposition into m^2 blocks of size n/m , where we assume for convenience that $n = m^k$ for some positive integer k . It has been shown that for 2×2 matrices, at least seven multiplications are *required*. Thus, divide-and-conquer algorithms, which use as their starting point a method of multiplying two $m \times m$ matrices using less than the number of multiplications used by Strassen's method, require that m be larger than 2.

Nearly ten years after Strassen discovered his identities, Pan found a way to multiply two 70×70 matrices that involves only 143,640 multiplications (compared to over 150,000 multiplications used by Strassen's method), yielding an algorithm that performs $O(n^{2.795})$ multiplications. Improvements over Pan's algorithm have been discovered, and the best result currently known (due to Coppersmith and Winograd) can multiply two $n \times n$ matrices using $O(n^{2.376})$ multiplications. However, all of these methods require n to be quite large before improvements over Strassen's method are significant. Moreover, they are very complicated due to the large number of identities required to achieve the savings in the number of multiplications. Hence, the currently known order-of-complexity improvements over Strassen's algorithm are mostly of theoretical rather than of practical interest.

Applications of the DFT and FFT

Almost a century after the Taylor series expansion of a (infinitely) differentiable function $f(x)$ was discovered (actually by Gregory), in a paper published in 1807 dealing with heat transfer problems, Fourier suggested expanding a function in terms of sines and cosines. Unlike the Taylor series, the Fourier series does not involve the powers of sines and cosines, but rather uses multiples of a fundamental frequency of these sinusoids. We will assume in our discussion that $x(t)$ is a continuous periodic signal with period T , in which case the Fourier series becomes

$$x(t) = \sum_{n=0}^{\infty} (a_n \cos(2\pi \frac{n}{T} t) + b_n \sin(2\pi \frac{n}{T} t)) \quad (7.7.1)$$

Thus, (7.7.1) is an infinite sum of scaled cosine and sine waves with frequencies $0, 1/T, 2/T, 3/T, \dots$.

Using the orthogonality properties of the sine and cosine, it can readily be shown that the coefficients a_n and b_n in (7.7.1) are given by

$$a_0 = \frac{1}{T} \int_0^T x(t) dt, \quad a_n = \frac{2}{T} \int_0^T x(t) \cos\left(2\pi \frac{n}{T} t\right) dt, \quad b_n = \frac{2}{T} \int_0^T x(t) \sin\left(2\pi \frac{n}{T} t\right) dt \quad (7.7.2)$$

Of course, when storing and sending signals with computers we cannot store more than a finite number of the terms in (7.7.2). Moreover, in practice a signal $x(t)$ such as a sound wave is captured and processed by measurements taken over time intervals, and a closed form explicit formula for $x(t)$ is not available making the integrals in (7.7.2) impossible to compute. Instead, for an appropriately small time interval $[0, T]$ the signal $x(t)$ is measured using a sufficiently large number N of sample points equally spaced in $[0, T]$. These values of $x(t)$, denoted by $x[n] = x(nT/N)$, are then correlated with the sines and cosines having the associated frequencies $0, 1, 2, \dots, N - 1$. More precisely, we have the following transformation, called (as with the *DFT* applied to polynomials) the *Discrete Fourier Transform (DFT)* of $x(t)$:

$$X[k] = \sum_{n=0}^{N-1} (x[n] \cos\left(2\pi \frac{nk}{N}\right) - x[n] \sin\left(2\pi \frac{nk}{N}\right)), \quad k = 0, 1, \dots, N - 1. \quad (7.7.3)$$

Thus we have transformed the signal $x(t)$ from the time domain into the frequency domain. The minus sign in (7.7.3) is taken because of the standard rewriting (see (7.7.5)) of the equation in terms of the complex exponentials $e^{-i2\pi \frac{nk}{N}}$ using the Euler formula $e^{i\theta} = \cos(\theta) + i \sin(\theta)$, namely

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-i2\pi \frac{nk}{N}} = \sum_{n=0}^{N-1} (x[n] \cos(2\pi \frac{nk}{N}) - ix[n] \sin(2\pi \frac{nk}{N})), \quad k = 0, 1, \dots, N - 1. \quad (7.7.4)$$

This rewriting of $X[k]$ using the complex exponentials is more than a convenience, it facilitates using the *FFT* to compute $X[k]$ and its inverse

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{i2\pi \frac{nk}{N}}, \quad n = 0, 1, \dots, N - 1. \quad (7.7.5)$$

Letting $\omega = e^{-2\pi i/N}$, we see that for a given k , ω^k is a $(N/k)^{\text{th}}$ primitive root of unity, and

$$X[k] = \sum_{n=0}^{N-1} x[n] (\omega^k)^n. \quad (7.7.6)$$

For $k > 0$, the k^{th} frequency $X[k]$ is thereby seen as measuring value of a sinusoid that complete k cycles as n runs from 0 to $N - 1$, that is a sinusoid of frequency k . When $k = 1$ we have what is known as the *fundamental frequency*. The value of sinusoid having frequency $X[k]$ corresponds to the assumed constituent sinusoid in $x(t)$ having frequency k/T . The complex number $X[k]$ encodes both the amplitude and phase of this constituent sinusoid by the respective formulas:

$$\text{amplitude} = 2|X[k]| / N = 2(\sqrt{\text{Real}(X[k])^2 + \text{Imag}(X[k])^2}) / N$$

$$\text{phase} = \arctan(\text{Imag}(X[k]) / \text{Real}(X[k])).$$

For $k = 0$, $X[0]$ is the sum of the measured values $x[0] + x[1] + \dots + x[N-1]$.

Remarks: It is important to note that n , k , and N are dimensionless, so that the time period T has been lost in the equations for $X[k]$ and $x[n]$. Thus, when sending the values of $X[k]$ to an

appropriate receiver, the value of T must be explicitly known by the receiver in order to reconstruct $x(t)$ (or to reconstruct a suitable approximation to $x(t)$) via the mapping $n \rightarrow nT/N$.

It is important to sample $x(t)$ at a sufficiently large number of points N in order to capture sinusoids of the appropriate frequencies. A theorem of Nyquist states that only frequencies less than $N/2$ can be accurately measured, and higher frequencies will tend to “alias” lower frequencies since they will be measured at times that are a subset of the measurement times of a lower frequencies. In fact, when the values of $x(t)$ are real (as is usual in applications), the frequencies higher than $N/2$ actually alias specific lower frequencies. Indeed, using the symbol $*$ to denote the conjugate of a complex number, we have $\omega^k = (\omega^{N-k})^*$ for a primitive n^{th} root of unity ω , so that it is easily verified for real functions $x(t)$ the following relation holds:

$$X[k] = X^*[N - k], \quad x(t) \text{ real}, \quad k = 1, 2, \dots, N/2. \quad (7.7.7)$$

Hence, the sinusoids having frequency greater than $N/2$ do not give any new information, and due to aliasing are ultimately dropped in the reconstructed $x(t)$.

Remarks: Sampling a continuous function at a sufficiently high sample rate is analogous to capturing a continuous in time movie scene using still snapshots (frames) taken rapidly (usually 24 frames per second). The human eye sees the rapid succession of still frames as continuous motion. If $x(t)$ is a sound wave, then in practice the wave is sampled by at least 44000 times per second. The discretized sound wave consisting of a discrete set of sinusoids can also be altered by the receiver using sound modifiers to delete or modify the intensity of the various sinusoids in order to enhance the listening experience. In particular, sinusoids at frequencies not detectable by human ears can be deleted. These considerations are additional advantages to breaking up the wave into its constituent sinusoids.

We illustrate the use of the DFT by two simple examples. Of course it is not feasible to illustrate examples from actual practice due to their high frequencies (see above remarks).

As our first example, let $x(t) = \sin(\pi t + \pi/4)$, $T = 2$, and $N = 4$. Then we are sampling at the four points $0, \frac{1}{2}, 1, \frac{3}{2}$ and $\omega = -i$. The DFT is then given by the matrix product

$$X = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \\ -1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix} = \begin{pmatrix} 0 \\ 2/\sqrt{2} - 2/\sqrt{2}i \\ 0 \\ 2/\sqrt{2} + 2/\sqrt{2}i \end{pmatrix} \quad (7.7.8)$$

Hence $X[1]$ has nonzero terms for both the cosine and sine, which might be surprising until we notice that $\sin(\pi t + \pi/4) = \sin(\pi t) \cos(\pi/4) + \cos(\pi t) \sin(\pi/4) = 1/\sqrt{2}(\cos(\pi t) + \sin(\pi t))$. Moreover, in our example the graph of our function is the graph of the ordinary sine function sifted by $\pi/4$ radians to the left, which is consistent with the phase angle $\arctan(\text{Real}(X[1])/\text{Imag}(X[1])) = \arctan(-1)$. The amplitude of X of our result is $2(|X[1]|)/N = 2(2)/4 = 1$ as expected.

As our second example, consider the function $x(t) = \sin(2\pi t) + 2\cos(\pi t)$ sampled on $[0, 2]$ as before. If we took $N = 4$, the theorem of Nyquist tells us that we should only expect correct results for sinusoids of frequency $< N/2 = 2$. However, $\sin(2\pi t)$ repeats twice on $[0, 2]$, so we

shouldn't expect to properly capture $\sin(2\pi t)$ with only 4 measuring points. In fact, $\sin(2\pi t)$ is 0 at each of our measuring points $t = 0, 1/2, 1, 3/2$, so that $\sin(2\pi t)$ is aliasing the identically zero function and is lost completely by the DFT. What remains is only the sinusoid $2\cos(\pi t)$ that is correctly captured. By the theorem of Nyquist $N = 8$ should suffice. The value of ω for $N = 8$ is $\omega = (1 - i)/\sqrt{2}$, and the DFT is given by

$$X = \begin{array}{cccccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 0 \\ 1 & \omega & -i & -\omega^* & -1 & -\omega & i & \omega^* & 1 + \sqrt{2} & 8 \\ 1 & -i & -1 & i & 1 & -i & -1 & i & 0 & -4i \\ 1 & -\omega^* & i & \omega & -1 & \omega^* & -i & -\omega & -1 - \sqrt{2} & 0 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & -2 & 0 \\ 1 & -\omega & -i & \omega^* & -1 & \omega & i & -\omega^* & 1 - \sqrt{2} & 0 \\ 1 & i & -1 & -i & 1 & i & -1 & -i & 0 & 4i \\ 1 & \omega^* & i & -\omega & -1 & -\omega^* & -i & \omega & -1 + \sqrt{2} & 8 \end{array} = \begin{array}{c} 0 \\ 8 \\ -4i \\ 0 \\ 0 \\ 0 \\ 4i \\ 8 \end{array} \quad (7.7.9)$$

$$X = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

We see that (7.7.9) yields $X[1] = 8$ corresponding to the $2\cos(\pi t)$ term, and has the correct amplitude $2|X[1]|/8 = 2$. Also, $X[2] = -4i$ corresponding to the $\sin(2\pi t)$ term, with the correct amplitude $2|X[2]|/8 = 1$. The phases are 0 in both cases. In addition, $X[0] = 0$ and $X[3] = 0$ as expected.

Of course as opposed to our two simple examples, in practice as we discussed we only know $x(t)$ from sample measurements, the whole point being to use the *DFT* to discover the amplitude and phases of the constituent sinusoids whose sum yields $x(t)$ (or a good enough approximation to it).

The *DFT* is utilized in a wide variety of applications, including all areas of signal processing, audio compression, digital image processing, video compression, coding, speech processing, speech recognition, digital communications, sonar, radar, seismology, biomedicine, and so forth. There are a number of other transformations related to the *DFT* that are also often used in practice. For example, an important transform known as the *Discrete Cosine Transformation (DCT)* uses only cosines, instead of both sines and cosines, and is used in JPEG compression of graphics files in combination with Huffman coding. A two dimensional

picture is broken up into one dimensional slices, and the *DCTs* are applied to each slice. In this case time sampling is replaced by space sampling.

In all of the applications of the *DFT* it is the *FFT* used to implement the *DFT* that makes the computations efficient enough to take place in real time by reducing the number of computations from n^2 to $O(n \log n)$.

Exercises

Section 7.1 The Divide-and-Conquer Paradigm

- 7.1 Design and analyze a divide-and-conquer algorithm for finding the maximum element in a list $L[0:n - 1]$.
- 7.2 Design and analyze a divide-and-conquer algorithm for finding the maximum and minimum elements in a list $L[0:n - 1]$.
- 7.3 Suppose we have a list $L[0:n - 1]$ representing the results of an election, so that $L[i]$ is the candidate voted for by person i , $i = 0, \dots, n - 1$. Design a linear algorithm to determine whether a candidate got a majority of the votes, that is, whether there exists a list element that occurs more than $n/2$ times in the list.
- 7.4 Design a version of *QuickSort* that uses a threshold. Do some empirical testing to determine a good threshold value.

Section 7.2 Symbolic Algebraic Operations on Polynomials

- 7.5 Give pseudocode and analyze the procedure *DirectPolyMult*, which is based directly on (7.2.1).
- 7.6 Repeat Exercise 7.5 when the polynomials are implemented by storing the coefficients in
 - a) an array
 - b) a linked list
 - c) consider the sparse case
- 7.7 Give pseudocode for a version of *PolyMult1* when the polynomials are implemented by storing the coefficients and the associated powers of the polynomials in a linked list.
- 7.8 When analyzing *PolyMult1*, we choose coefficient multiplication as our basic operation. This ignores the two multiplications by powers of x . Show how the procedure for multiplying a polynomial by x^i can be implemented with linear complexity. Taking this operation into account, verify that the order of complexity of *PolyMult1* remains unchanged.
- 7.9 Show that the divide-and-conquer algorithm for multiplying two polynomials based on the recurrence relation (7.2.2) has quadratic complexity.

Section 7.3 Multiplication of Large Integers

- 7.10 Design and analyze an algorithm *MultInt* for multiplying large integers.
- 7.11 a. Design an algorithm for adding two large integers implemented using arrays.
b. Repeat part (a) for linked list implementations.
- 7.12 Give pseudocode for the algorithm *MultInt* you designed in Exercise 7.10 for the following two implementations of large integers.
a. arrays
b. linked lists

Section 7.4 Multiplication of Matrices

- 7.13 Verify formula (7.4.4).
- 7.14 Verify formula (7.4.11).
- 7.15 Verify that the evaluation of formulas (7.4.10) and (7.4.11) requires 15 distinct additions or subtractions, which is three less than what is performed when using Strassen's identities.
- 7.16 Give pseudocode for the procedure *Strassen*, which implements Strassen's matrix multiplication algorithm. Assume that the input matrices A and B are both $n \times n$ matrices, where n is a power of 2.
- 7.17 Demonstrate the action of the procedure *Strassen* in Exercise 7.16 for the following input matrices.

$$A = \begin{bmatrix} 2 & 0 & 1 & 1 \\ 3 & 0 & 0 & 4 \\ 3 & -4 & 5 & 7 \\ 0 & 1 & 2 & 3 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 2 & 1 & 1 \\ 0 & 1 & 0 & -1 \\ 3 & 0 & 5 & 0 \\ 6 & 1 & 4 & 0 \end{bmatrix}$$

Section 7.5 The Discrete Fourier Transform

- 7.18 Verify formula (7.5.2).
- 7.19 Show that when ω is a primitive n th root of unity and n is a power of two, then ω^2 is a primitive $(n/2)$ th root of unity.
- 7.20 Give the tree of recursive calls to *FFTRec* for $n = 16$.
- 7.21 Write a program implementing *FFTRec*, and run for various inputs.
- 7.22 For $i = 1, \dots, n = 2^k$, consider the permutation $\pi_k(i) = j$ such that a_j is the coefficient corresponding to the i th leaf node of the tree of recursive calls to *FFTRec*. Complete the inductive proof that the k -digit binary representation of $\pi_k(i)$ is obtained from the k -digit binary representation of i by simply reversing the digits, where leading zeros are included (if necessary).
- 7.23 Design and analyze the procedure *ReverseBinPerm*($R[0:n - 1]$), which computes $R[i] = \pi_k(i)$, $i = 0, \dots, n - 1$.
- 7.24 Write a program implementing the iterative version of *FFT*, and run for various inputs.

Section 7.5.5 The Inverse Fourier Transformation and Fast Polynomial Multiplication

7.25 Using $x \cdot x$ verify that $DFT^{-1} =$

7.5.8 Given the polynomials $P(x) = x^3 - x + 2$, $Q(x) = 2x^2 - 1$,

- a. Show that going along the left-hand side of the commutative diagram in Figure 7.10 and using *FFT* yields the evaluations:

$$[P(1), P(\omega), \dots, P(\omega^7)] = [2, 2 - \omega + \omega^3, 2 - 2i, 2 + \omega - \omega^3,$$

$$2, 2 + \omega - \omega^3, 2 + 2i, 2 - \omega + \omega^3]$$

$$[Q(1), Q(\omega), \dots, Q(\omega^7)] = [1, 2i - 1, -3, -2i - 1, 1, 2i - 1, -3, -2i - 1].$$

- b. Verify that the polynomial $P(x)Q(x)$ results from going along the right-hand side of the commutative diagram in Figure 7.10 by using *FFT* to compute $(DFT_{\omega})^{-1} = (1/8)DFT_{\omega^{-1}}$.

Additional Exercises from Appendix A

7.54 Prove Proposition A.1 from Appendix A.

7.55 Prove Proposition A.2 from Appendix A.

7.56 Prove Proposition A.3 from Appendix A.

7.57 Give a complex number $z = x + iy$, its *complex conjugate* \bar{z} is defined by

$$\bar{z} = x - iy.$$

- a. Give a geometric interpretation of \bar{z} .

- b. Show that the complex conjugate has the following properties:

$$\text{i. } \overline{-z} = -\bar{z},$$

$$\text{ii. } \overline{z_1 + z_2} = \overline{z_1} + \overline{z_2},$$

$$\text{iii. } \overline{z_1 z_2} = \overline{z_1} \overline{z_2}.$$

- c. Use part (b) to show that the complex roots of a polynomial $P(x)$ with real coefficients occur in conjugate pairs; that is, if $P(z) = 0$, then $P(\bar{z}) = 0$.

7.58 Verify that the n^{th} roots of z defined by (A.31) from Appendix A are all distinct.

Section 7.6 Two Classical Problems in Computational Geometry

8.27 Prove that a rectangle of width d and height $2d$ can contain at most 6 points whose pair-wise distances are at least d .

8.28 Write a computer program implementing the closest pair algorithm.

8.29 Verify that the divide-and-conquer algorithm for the convex hull problem discussed in Section 7.6. has $O(n \log n)$ complexity.

8.30 Show that the convex hull of a set of points P_1, P_2, \dots, P_n in the plane consists of the set of all points $\lambda_1 P_1 + \lambda_2 P_2 + \dots + \lambda_n P_n$, where $\lambda_1, \lambda_2, \dots, \lambda_n$ are all non-negative real numbers such that $\lambda_1 + \lambda_2 + \dots + \lambda_n = 1$.

8.31 Write a computer program implementing the divide-and-conquer algorithm for the convex hull problem discussed in Section 7.6.

8.32 Show that a lower bound in $\Omega(n \log n)$ exists for the convex hull problem in the plane. Hint: Let x_1, \dots, x_n be n real numbers. Consider the convex hull P_1, \dots, P_m of

the n points $(x_1, (x_1)^2), \dots, (x_n, (x_n)^2)$, listed in counter-clockwise order, and where P_1 has minimum x -coordinate. Show that $m = n$ and the x -coordinates of P_1, \dots, P_m (in that order) is a sorting of x_1, \dots, x_n in increasing order.

Section 7.7 Closing Remarks

- 7.33 Pan's divide-and-conquer matrix multiplication algorithm is based on a partitioning scheme that assumes n is a power of 70. The complexity $T(n)$ of Pan's divide-and-conquer algorithm satisfies the recurrence relation

$$T(n) = 143,640T(n/70), \quad n = 70^i, \quad i \geq 1, \quad \text{init.cond. } T(1) = 1.$$

Show that this implies that $T(n)$ is approximately $n^{2.795}$.

8

Dynamic Programming

Dynamic programming is a design strategy that involves constructing a solution S to a given problem by building it up dynamically from solutions S_1, S_2, \dots, S_m to smaller (or simpler) instances of the problem. The solution S_i to any given smaller problem instance is itself built up from the solutions to even smaller (simpler) problem instances, and so forth. We start with the known solutions to the smallest (simplest) problem instances and build from there in a bottom-up fashion. To be able to reconstruct S from S_1, S_2, \dots, S_m , some additional information is usually required. We let *Combine* denote the function that combines S_1, S_2, \dots, S_m , using the additional information to obtain S , so that

$$S = \text{Combine}(S_1, S_2, \dots, S_m).$$

Dynamic programming is similar to divide-and-conquer in the sense that it is based on a recursive division of a problem instance into smaller or simpler problem instances. However, whereas divide-and-conquer algorithms often utilize a top-down resolution method, dynamic programming algorithms invariably proceed by solving all the simplest problem instances before combining them into more complicated problem instances in a bottom-up fashion. Also, unlike many instances of divide-and-conquer, dynamic programming algorithms typically never consider a given problem instance than once.

Dynamic programming algorithms for optimization problems also can avoid generating suboptimal problem instances when the *Principle of Optimality* holds, thereby leading to increased efficiency. In this chapter we consider a number of well-known problems that are amenable to the method of dynamic programming.

8.1 Optimization Problems and the Principle of Optimality

The method of dynamic programming is most effective in solving optimization problems when the Principle of Optimality holds. Consider the set of all *feasible* solutions S to the given optimization problem; that is, solutions S satisfying the constraints of the problem. An *optimal* solution S is a solution that optimizes (minimizes or maximizes) the objective function. If we wish to obtain an optimal solution S to the given problem instance, then we must optimize (minimize or maximize) over *all* solutions S_1, S_2, \dots, S_m such that $S = \text{Combine}(S_1, S_2, \dots, S_m)$. For many problems it is computationally infeasible to examine all such solutions, because often there are exponentially many possibilities. Fortunately we can drastically reduce the number of problem instances that we need to consider if the Principle of Optimality holds.

Definition 8.1.1

Given an optimization problem and an associated function *Combine*, the *Principle of Optimality* holds if the following is always true: If $S = \text{Combine}(S_1, S_2, \dots, S_m)$ and S is an

optimal solution to the problem instance, then S_1, S_2, \dots, S_m , are *optimal* solutions to their associated problem instances.

Key Fact

The efficiency of dynamic programming solutions based on a recurrence relation expressing the principle of optimality results from (i) the bottom-up resolution of the recurrence, thereby eliminating redundant recalculations, and (ii) eliminating suboptimal solutions to subproblems as we build up optimal solutions to larger problems; that is, we use only optimal solution “building blocks” in constructing our optimal solution.

We first illustrate the Principle of Optimality for the problem of finding a parenthesization of a matrix product of matrices M_0, \dots, M_{n-1} that minimizes the total number of (scalar) multiplications over all possible parenthesizations. If $(M_0 \dots M_k)(M_{k+1} \dots M_{n-1})$ is the “first cut” set of parentheses (and the last product performed), then the matrix products $M_0 \dots M_k$ and $M_{k+1} \dots M_{n-1}$ must both be parenthesized in such a way as to minimize the number of multiplications required to carry out the respective products. As a second example, consider the problem of finding optimal binary search trees for a set of distinct keys. Recall that a binary search tree T for keys $K_0 < \dots < K_{n-1}$ is a binary tree on n nodes each containing a key, such that the following property is satisfied: Given any node v in the tree, each key in the left subtree rooted at v is no larger than the key in v , and each key in the right subtree rooted at v is no smaller than the key in v (see Figure 8.5). If K_i is the key in the root, then the left subtree L of the root contains K_0, \dots, K_{i-1} and the right subtree R of the root contains K_{i+1}, \dots, K_{n-1} . Given a binary search tree T for keys K_0, \dots, K_{n-1} , let K_i denote the key associated with the root of T , and let L and R denote the left and right subtrees (of the root) of T , respectively. Again it follows that L (solution S_1) is a binary search tree for keys K_0, \dots, K_{i-1} and R (solution S_2) is a binary search tree for keys K_{i+1}, \dots, K_{n-1} . Given L and R , the function $\text{Combine}(L, R)$ merely reconstructs the tree T using K_i as the root. In the next section, we show that the Principle of Optimality holds for this problem by showing that if T is an optimal binary search tree, then so are L and R .

8.2 Optimal Parenthesization for Computing a Chained Matrix Product

Our first example of dynamic programming is an algorithm for the problem of parenthesizing a chained matrix product so as to minimize the number of (scalar) multiplications performed when computing the product. When solving this problem we will assume the straightforward method of matrix multiplication. If A and B are matrices of dimensions $p \times q$ and $q \times r$, then the matrix product AB involves pqr multiplications. Given a sequence (chain) of matrices M_0, M_1, \dots, M_{n-1} , consider the product $M_0 M_1 \dots M_{n-1}$, where the matrix M_i has dimension $d_i \times d_{i+1}$, $i = 0, \dots, n$, for a suitable sequence of positive integers d_0, d_1, \dots, d_n . Since matrix product is an associative operation, there are a number of ways to evaluate the chained product depending on how we choose to parenthesize the expression. It turns out that the manner in which the expression is parenthesized can make a major difference in the total number of multiplications performed when computing the chained product. In this section we consider the problem of finding an

optimal parenthesization, that is, a parenthesization that minimizes the total number of multiplications performed using ordinary matrix products.

We illustrate the problem with an example that commonly occurs in multivariate calculus. Suppose A and B are $n \times n$ matrices, X is an $n \times 1$ column vector, and we wish to evaluate ABX . The product ABX can be parenthesized in two ways, namely, $(AB)X$ and $A(BX)$, resulting in $n^3 + n^2$ and $2n^2$ multiplications, respectively. Thus, the two ways of parenthesizing make a rather dramatic difference in the number of multiplications performed (order $\Theta(n^3)$ versus order $\Theta(n^2)$).

The following is a formal (recursive) definition of a fully parenthesized chained matrix product and its associated first cut.

definition 8.2.1

Given the sequence of matrices M_0, M_1, \dots, M_{n-1} , P is a *fully-parenthesized* matrix product of M_0, M_1, \dots, M_{n-1} , which for convenience we simply call a *parenthesization of $M_0M_1\dots M_{n-1}$* , if P satisfies

$$\begin{aligned} P &= M_0, \quad n = 1, \\ P &= (P_1P_2), \quad n > 1, \end{aligned}$$

where for some k , P_1 and P_2 are parenthesizations of the matrix products $M_0M_1\dots M_k$ and $M_{k+1}M_{k+2}\dots M_{n-1}$, respectively. We call P_1 and P_2 the *left* and *right* parenthesizations of P , respectively. We call the index k the *first cut index of P* .

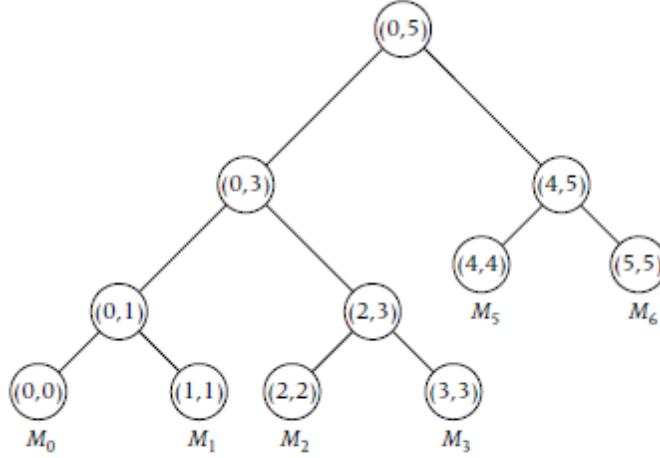
The table in Figure 8.1 shows all the different parenthesizations of the matrices M_0, M_1, M_2 , and M_3 having dimensions 20×10 , 10×50 , 50×5 , and 5×30 , respectively, with the optimal parenthesizations highlighted.

no. mult.	no. mult.	no. mult.	no. mult.
M_0	(M_0M_1) 10000	$(M_0(M_1M_2))$ 3500	$(M_0(M_1(M_2M_3)))$ 28500
		$((M_0M_1)M_2)$ 15000	$(M_0((M_1M_2)M_3))$ 10000
			$((M_0M_1)(M_2M_3))$ 47500
			$((M_0(M_1M_2))M_3)$ 6500
			$((M_0(M_1M_2))M_3)$ 18000

Number of multiplications performed for each full parenthesization are shown for matrices M_0, M_1, M_2 , and M_3 having dimensions 20×10 , 10×50 , 50×5 , and 5×30 , respectively. The optimal parenthesizations are shaded.

Figure 8.1

There is one-to-one correspondence between parenthesizations of $M_0M_1\dots M_{n-1}$ and 2-trees having n leaf nodes. Given a parenthesization P of $M_0M_1\dots M_{n-1}$, if $n = 1$ its associated 2-tree $T(P)$ consists of a single node corresponding to the matrix M_0 ; Otherwise, $T(P)$ has left subtree $T(P_1)$ and right subtree $T(P_2)$, where P_1 and P_2 are the left and right parenthesizations of P . The 2-tree $T(P)$ is the *expression tree* for P (see Figure 8.2).



Associated expression 2-tree for parenthesization $((M_0M_1)(M_2M_3))(M_4M_5)$. The label (i,j) inside each node indicates that the matrix product associated with the node involves matrices M_i, M_{i+1}, \dots, M_j

Figure 8.2

Thus, the number of parenthesizations p_n equals the number t_n of 2-trees having n leaf nodes, so that by Exercise 4.14 we have

$$p_n = \frac{1}{n} \binom{2n-2}{n-1} \geq \frac{4^{n-1}}{2n^2 - n} \in \Omega\left(\frac{4^n}{n^2}\right). \quad (8.2.1)$$

Hence, a brute-force algorithm that examines all possible parenthesizations is computationally infeasible.

We are led to consider a dynamic programming solution to our problem by noting that the Principle of Optimality holds for optimal parenthesizing. Indeed, consider any optimal parenthesization P for $M_0M_1\dots M_{n-1}$. Clearly, both the left and right parenthesizations P_1 and P_2 of P must be optimal for P to be optimal.

For $0 \leq i \leq j \leq n-1$, let m_{ij} denote the number of multiplications performed using an optimal parenthesization of $M_iM_{i+1}\dots M_j$. By the Principle of Optimality, we have the following recurrence for the numbers m_{ij} based on making an optimal choice for the first cut index

$$\begin{aligned} m_{ij} &= \min_k \{m_{ik} + m_{k+1,j} + d_i d_{k+1} d_{j+1} : 0 \leq i \leq k < j \leq n-1\} \\ \text{init. cond. } m_{ii} &= 0, \quad i = 0, \dots, n-1. \end{aligned} \quad (8.2.2)$$

The value $m_{0,n-1}$ corresponds to the minimum number of multiplications performed when computing $M_0M_1\dots M_{n-1}$. A divide-and-conquer algorithm *ParenthesizeRec* can be based directly on a top-down implementation of the recurrence relation (8.2.2). Unfortunately, a great many recalculations are performed by *ParenthesizeRec*, and it ends

up doing $\Omega(3^n)$ multiplications to compute the minimum number $m_{0,n-1}$ corresponding to an optimal parenthesization.

A straightforward dynamic programming algorithm proceeds by computing the values m_{ij} , $0 \leq i \leq j \leq n - 1$ in a bottom-up fashion using (8.2.2) (and thereby avoiding recalculations). Note that the values m_{ij} , $0 \leq i \leq j \leq n - 1$, occupy the upper-right triangular portion of an $n \times n$ table. Our bottom-up resolution proceeds throughout the upper-right triangular portion diagonal by diagonal, starting from the bottom diagonal consisting of the elements $m_{ii} = 0$, $i = 0, \dots, n - 1$. The q th diagonal consists of the elements $m_{i,i+q}$, $q = 0, \dots, n - 1$. In Figure 8.3 we illustrate the computation of the m_{ij} for the example given in Figure 8.1. When computing m_{ij} , we also generate a table c_{ij} of indices k where the minimum in (8.2.2) occurs, that is c_{ij} is where the first cut in $M_i M_{i+1} \dots M_j$ is made in an optimal parenthesization. The values c_{ij} can then be used to actually compute the matrix product according to the optimal parenthesization.

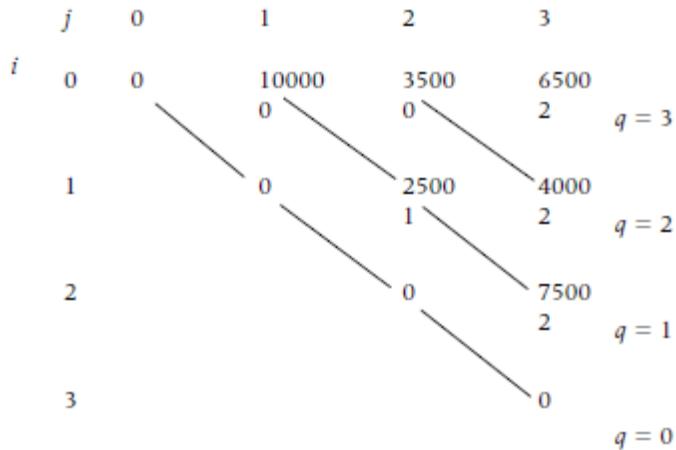


Table showing values m_{ij} , $0 \leq i \leq j \leq 3$, computed diagonal by diagonal from $q = 0$ to $q = 3$ using the bottom-up resolution of (8.2.2) for matrices M_0, M_1, M_2 , and M_3 having dimensions 20×10 , 10×50 , 50×5 , and 5×30 , respectively. The values of c_{ij} are shown underneath each m_{ij} , $0 \leq i \leq j \leq 3$.

Figure 8.3

The following procedure *OptimalParenthesization* accepts as input the *dimension sequence* $d[0:n]$, where matrix M_i has dimension $d_i \times d_{i+1}$, $i = 0, \dots, n - 1$. Procedure *OptimalParenthesization* outputs the matrix $m[0:n - 1, 0:n - 1]$, where $m[i,j] = m_{ij}$, $0 \leq i \leq j \leq n - 1$, is defined by recurrence (8.2.2). *OptimalParenthesization* also outputs the matrix *FirstCut*[0:n - 1, 0:n - 1], where $\text{FirstCut}[i,j] = c_{ij}$, $0 \leq i \leq j \leq n - 1$, which is the first-cut index in an optimal parenthesization for $M_i \dots M_j$.

```

procedure OptimalParenthesization( $d[0:n], m[0:n-1,0:n-1], FirstCut[0:n-1,0:n-1]$ )
Input:  $d[0:n]$  (dimension sequence for matrices  $M_0, M_1, \dots, M_{n-1}$ )
Output:  $m[0:n-1,0:n-1]$  ( $m[i,j] =$  number of multiplications performed in an optimal
parenthesization for computing  $M_i \dots M_j$ ,  $0 \leq i \leq j \leq n-1$ )
 $FirstCut[0:n-1,0:n-1]$  (index of first cut in optimal parenthesization of
 $M_i \dots M_j$ ,  $0 \leq i \leq j \leq n-1$ )
for  $i \leftarrow 0$  to  $n-1$  do // initialize  $M[i,i]$  to zero
     $m[i,i] \leftarrow 0$ 
endfor
for  $diag \leftarrow 1$  to  $n-1$  do
    for  $i \leftarrow 0$  to  $n-1-diag$  do
         $j \leftarrow i + diag$  // compute  $m_{ij}$  according to (8.2.2)
         $Min \leftarrow m[i+1,j] + d[i]*d[i+1]*d[j+1]$ 
         $TempCut \leftarrow i$ 
        for  $k \leftarrow i+1$  to  $j-1$  do
             $Temp \leftarrow m[i,k] + m[k+1,j] + d[i]*d[k+1]*d[j+1]$ 
            if  $Temp < Min$  then
                 $Min \leftarrow Temp$ 
                 $TempCut \leftarrow k$ 
            endif
        endfor
         $m[i,j] \leftarrow Min$ 
         $FirstCut[i,j] \leftarrow TempCut$ 
    endfor
endfor
end OptimalParenthesization

```

A simple loop counting shows that the complexity of *OptimalParenthesization* is in $\Theta(n^3)$.

It is now straightforward to write pseudocode for a recursive function *ChainMatrixProd* for computing the chained matrix product $M_0 \dots M_{n-1}$ using an optimal parenthesization. We assume that the matrices M_0, \dots, M_{n-1} and the matrix *FirstCut* $[0:n-1,0:n-1]$ are global variables to the procedure *ChainMatrixProd*. The chained matrix product $M_0 \dots M_{n-1}$ is computed by initially invoking the function *ChainMatrixProd* with $i = 0$ and $j = n - 1$. *ChainMatrixProd* invokes a function *MatrixProd*, which computes the matrix product of two input matrices.

```

function ChainMatrixProd( $i,j$ ) recursive
Input:  $i,j$  (indices delimiting matrix chain  $M_i, \dots, M_j$ )
 $M_0, \dots, M_{n-1}$  (global matrices)
 $FirstCut[0:n-1,0:n-1]$  (global matrix computed by
OptimalParenthesization)
Output:  $M_i \dots M_j$  (matrix chain)
if  $j > i$  then
     $X \leftarrow ChainMatrixProd(i, FirstCut[i,j])$ 
     $Y \leftarrow ChainMatrixProd(FirstCut[i,j] + 1, j)$ 

```

```

        return(MatrixProd(X,Y))
else
    return(Mi)
endif
end ChainMatrixProd

```

For the example given in Figure 8.1, invoking *ChainMatrixProd* with M_0, M_1, M_2, M_3 computes the chained matrix product $M_0M_1M_2M_3$ according to the parenthesization $((M_0(M_1M_2))M_3)$.

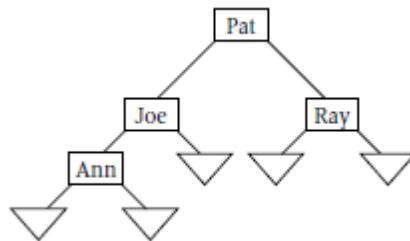
8.3 Optimal Binary Search Trees

We now use dynamic programming and the Principle of Optimality to generate an algorithm for the problem of finding optimal binary search trees. Given a search tree T and a search element X , the following recursive strategy finds an occurrence (if any) of a key X . First, X is compared to the key K associated with the root. If X is found there, then we are done. Otherwise, if X is less than K , then we search the left subtree, else we search the right subtree.

Consider, for example, the binary search tree given in Figure 8.4 involving the four keys ‘Ann’, ‘Joe’, ‘Pat’, ‘Ray’. The internal nodes correspond to the successful searches $X = \text{‘Ann’}$, $X = \text{‘Joe’}$, $X = \text{‘Pat’}$, $X = \text{‘Ray’}$, and the leaf nodes correspond to the unsuccessful searches $X < \text{‘Ann’}$, $\text{‘Ann’} < X < \text{‘Joe’}$, $\text{‘Joe’} < X < \text{‘Pat’}$, $\text{‘Pat’} < X < \text{‘Ray’}$, $\text{‘Ray’} < X$. Suppose, for example that $X = \text{‘Ann’}$. Then *SearchBinSrchTree* makes three comparisons, first comparing X to ‘Pat’, then comparing X to ‘Joe’, and finally comparing X to ‘Ann’. Now suppose that $X = \text{‘Pete’}$. Then *SearchBinSrchTree* makes two comparisons, first comparing X to ‘Pat’ and then comparing X to ‘Ray’.

SearchBinSrchTree implicitly) branches to the left child of the node containing the key ‘Ray’, that is, to the leaf (implicit node) corresponding to the interval ‘Pat’ < X < ‘Ray’. Let p_0, p_1, p_2, p_3 be the probability that $X = \text{‘Ann’}$, $X = \text{‘Joe’}$, $X = \text{‘Pat’}$, $X = \text{‘Ray’}$, respectively, and let q_0, q_1, q_2, q_3, q_4 , denote the probability that $X < \text{‘Ann’}$, $\text{‘Ann’} < X < \text{‘Joe’}$, $\text{‘Joe’} < X < \text{‘Pat’}$, $\text{‘Pat’} < X < \text{‘Ray’}$, $\text{‘Ray’} < X$, respectively. Then, the average number of comparisons made by *SearchBinSrchTree* for the tree T of Figure 8.4 is given by

$$3p_0 + 2p_1 + p_2 + 2p_3 + 3q_0 + 3q_1 + 2q_2 + 2q_3 + 2q_4.$$



Search tree with leaf nodes drawn representing unsuccessful searches

Figure 8.4

Now consider a general binary search tree T whose internal nodes correspond to a fixed set of n keys K_0, K_1, \dots, K_{n-1} with associated probabilities $\mathbf{p} = (p_0, p_1, \dots, p_{n-1})$, and whose $n + 1$ leaf (external) nodes correspond to the $n + 1$ intervals $I_0: X < K_0, I_1: K_0 < X < K_1, \dots, I_{n-1}: K_{n-2} < X < K_{n-1}, I_n: X > K_{n-1}$ with associated probabilities $\mathbf{q} = (q_0, q_1, \dots, q_n)$. (When implementing T , the leaf nodes need not actually be included. However, when discussing the average behavior of *SearchBinSrchTree*, it is useful to include them.) We now derive a formula for the average number of comparisons $A(T, n, \mathbf{p}, \mathbf{q})$ made by *SearchBinSrchTree*. Let d_i denote the depth of the internal node corresponding to K_i , $i = 0, \dots, n - 1$. Similarly, let e_i denote the depth of the leaf node corresponding to the interval I_i , $i = 0, 1, \dots, n$. If $X = K_i$, then *SearchBinSrchTree* traverses the path from the root to the internal node corresponding to K_i . Thus, it terminates after performing $d_i + 1$ comparisons. On the other hand, if X lies in I_i , then *SearchBinSrchTree* traverses the path from the root to the leaf node corresponding to I_i and terminates after performing e_i comparisons. Thus, we have

$$A(T, n, \mathbf{p}, \mathbf{q}) = \sum_{i=0}^{n-1} p_i(d_i + 1) + \sum_{i=0}^n q_i e_i. \quad (8.3.1)$$

We now consider the problem of determining an *optimal* binary search tree T , optimal in the sense that T minimizes $A(T, n, \mathbf{p}, \mathbf{q})$ over all binary search trees T . This problem is solved by a complete tree in the case where all the p_i 's are equal and all the q_i 's are equal. Here we use dynamic programming to solve the problem for general probabilities p_i and q_i . In fact, we solve the slightly more general problem, where we relax the condition that p_0, \dots, p_{n-1} and q_0, \dots, q_n are probabilities by allowing them to be arbitrary nonnegative real numbers. One could regard these numbers as frequencies, as we did when discussing Huffman codes in Chapter 6. That is, we solve the problem

$$\underset{T}{\text{minimize}} \ A(T, n, \mathbf{p}, \mathbf{q}) \quad (8.3.2)$$

over all binary search trees T of size n , where p_0, \dots, p_{n-1} and q_0, \dots, q_n are given fixed nonnegative real numbers. For convenience we sometimes refer to $A(T, n, \mathbf{p}, \mathbf{q})$ as the *cost* of T . We define $\sigma(\mathbf{p}, \mathbf{q})$ by:

$$\sigma(\mathbf{p}, \mathbf{q}) = \sum_{i=0}^{n-1} p_i + \sum_{i=0}^n q_i. \quad (8.3.3)$$

Note that we have removed the probability constraint that $\sigma(\mathbf{p}, \mathbf{q}) = 1$. Similar to our discussion of chained matrix products, we could obtain an optimal search tree by enumerating all binary search trees on the given identifiers and choosing the one with minimum $A(T, n, \mathbf{p}, \mathbf{q})$. However, the number of different binary search trees on n identifiers is the same as the number of binary trees on n nodes, which is given by the n th Catalan number

$$b_n + \frac{1}{n+1} \binom{2n}{n} \in \Omega\left(\frac{4^n}{n^2}\right).$$

Thus, a brute force algorithm for determining an optimal binary search tree using simple enumeration is computationally infeasible. Fortunately, the Principle of Optimality holds for the optimal binary search tree problem, so we look for a solution using dynamic programming.

Let K_i denote the key associated with the root of T , and let L and R denote the left and right subtrees (of the root) of T , respectively. As we remarked earlier, L is a binary search tree for the keys K_0, \dots, K_{i-1} , and R is a binary search tree for the keys K_{i+1}, \dots, K_{n-1} . For convenience, let $A(T) = A(T, n, \mathbf{p}, \mathbf{q})$, $A(L) = A(L, i, p_0, \dots, p_{i-1}, q_0, \dots, q_i)$, and $A(R) = A(R, n-i-1, p_{i+1}, \dots, p_{n-1}, q_{i+1}, \dots, q_n)$. Clearly, each node of T different from the root corresponds to exactly one node in either L or R . Further, if N is a node in T corresponding to a node N' in L , then the depth of N in T is exactly one greater than the depth of N' in L . A similar result holds if N corresponds to a node in R . Thus, it follows immediately from (8.3.1) that

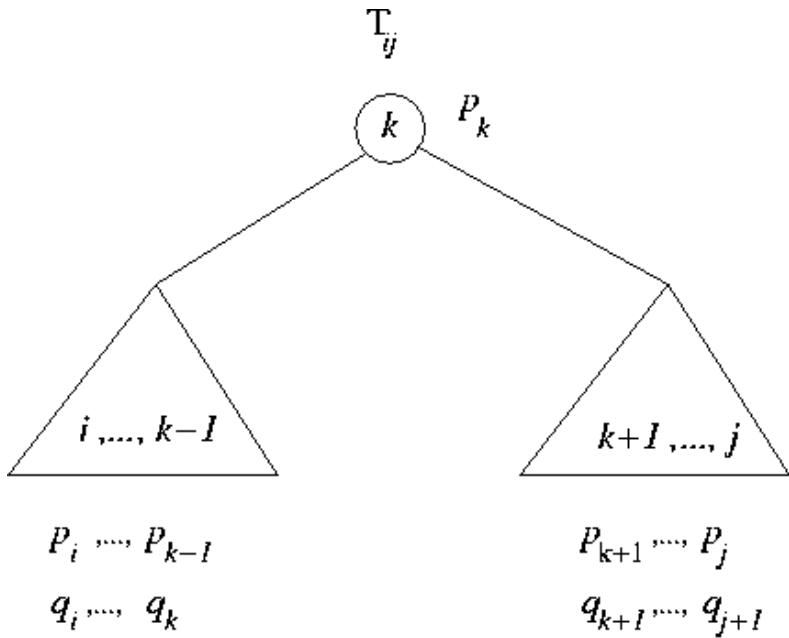
$$A(T) = A(L) + A(R) + \sigma(\mathbf{p}, \mathbf{q}). \quad (8.3.4)$$

We now employ (8.3.4) to show that the Principle of Optimality holds for the problem of finding an optimal search tree. Suppose that T is an optimal search tree; that is, T minimizes $A(T)$. We must show that L and R are also optimal search trees. Suppose there exists a binary search tree L' with $i-1$ nodes involving the keys K_0, \dots, K_{i-1} such that $A(L') < A(L)$. Clearly, the tree T' obtained from T by replacing L with L' is a binary search tree. Further, it follows from (8.3.4) that $A(T') < A(T)$, contradicting the assumption that T is an optimal binary search tree. Hence, L is an optimal binary search tree. By symmetry, R is also an optimal binary search tree, which establishes that the Principle of Optimality holds for the optimal binary search tree problem.

Since the Principle of Optimality holds, when constructing an optimal search tree T we need only consider binary search trees L and R , both of which are optimal. This observation, together with recurrence relation (8.3.4), is the basis of the following dynamic programming algorithm for constructing an optimal binary search tree.

For $i, j \in \{0, \dots, n-1\}$, we let T_{ij} denote an *optimal* search tree involving the consecutive keys K_i, K_{i+1}, \dots, K_j , where T_{ij} is the null tree if $i > j$. Thus, if K_k is the root key, then the left subtree L is $T_{i,k-1}$ and the right subtree R is $T_{k+1,j}$ (see Figure 8.5). Also, note that $T = T_{0,n-1}$ is an optimal search tree involving all n keys. For convenience, we define

$$\sigma(i, j) = \sum_{k=i}^j p_k + \sum_{k=i}^{j+1} q_k.$$



Principle of Optimality: if T_{ij} is optimal, then L and R must be optimal; that is, $L = T_{i,j-1}$ and $R = T_{k+1,j}$

Figure 8.5

We define $A(T_{ij})$ by:

$$A(T_{ij}) = A(T_{ij}, j-i+1, p_i, p_{i+1}, \dots, p_j, q_i, q_{i+1}, \dots, q_{j+1}). \quad (8.3.5)$$

Since the keys are sorted in nondecreasing order, it follows from the Principle of Optimality and (8.3.4) that

$$A(T_{ij}) = \min_k \{A(T_{i,k-1}) + A(T_{k+1,j})\} + \sigma(i, j), \quad (8.3.6)$$

where the minimum is taken over all $k \in \{i, i+1, \dots, j\}$.

Recurrence relation (8.3.6) yields an algorithm for computing an optimal search tree T . The algorithm begins by generating all single-node binary search trees, which are trivially optimal. Namely, $T_{00}, T_{11}, \dots, T_{n-1,n-1}$. Using (8.3.6), the algorithm can then generate optimal search trees $T_{01}, T_{12}, \dots, T_{n-2,n-1}$. In general, at the k th stage in the algorithm, the recurrence relation (8.3.6) is applied to construct the optimal search trees $T_{0,k-1}, T_{1,k}, \dots, T_{n-k,n-1}$, using the previously generated optimal search trees as building blocks. Figure 8.6 illustrates the algorithm for a sample instance involving $n = 4$ keys. Note that there are two possible choices for T_{02} in Figure 8.6, each having a minimum cost of 1.1. The tree with the smaller root key was selected.

i	0	1	2	3	4
p_i	.15	.1	.2	.3	
q_i	.05	.05	0	.05	.1

$$T_{00} = \textcircled{0} \quad T_{11} = \textcircled{1} \quad T_{22} = \textcircled{2} \quad T_{33} = \textcircled{3}$$

$$A(T_{00}) = .25 \quad A(T_{11}) = .15 \quad A(T_{22}) = .25 \quad A(T_{33}) = .45$$

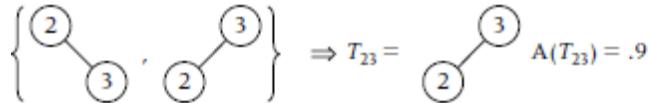
$$\left\{ \begin{array}{c} \textcircled{0} \\ \textcircled{1} \end{array}, \begin{array}{c} \textcircled{0} \\ \textcircled{1} \end{array} \right\} \Rightarrow T_{01} = \begin{array}{c} \textcircled{0} \\ \textcircled{1} \end{array} \quad A(T_{01}) = .5$$

$$\min \{0 + .15 = .15, .25 + 0 = .25\} + .35 = .5$$

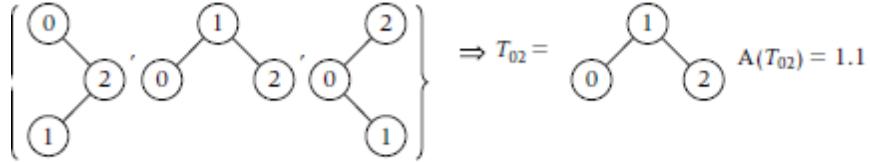
$$\left\{ \begin{array}{c} \textcircled{1} \\ \textcircled{2} \end{array}, \begin{array}{c} \textcircled{1} \\ \textcircled{2} \end{array} \right\} \Rightarrow T_{12} = \begin{array}{c} \textcircled{1} \\ \textcircled{2} \end{array} \quad A(T_{12}) = .55$$

$$\min \{0 + .25 = .25, .15 + 0 = .15\} + .4 = .55$$

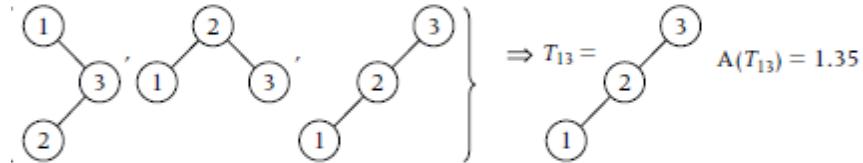
Continued



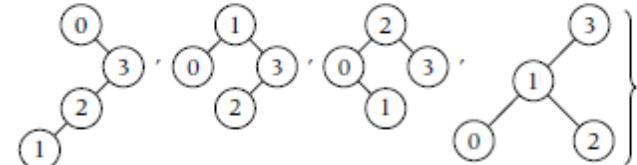
in $[0 + .45 = .45, .25 + 0 = .25] + .65 = .9$



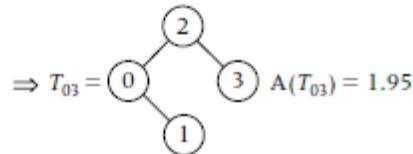
in $[0 + .55 = .55, .25 + .25 = .5, .5 + 0 = .5] + .6 = 1.1$



in $[0 + .9 = .9, .15 + .45 = .6, .55 + 0 = .55] + .8 = 1.35$



in $[0 + 1.35 = 1.35, .25 + .9 = 1.15, .5 + .45 = .95, 1.1 + 0 = 1.1] + 1 = 1.95$



Action of algorithm to find an optimal binary search tree using dynamic programming

Figure 8.6

We now give pseudocode for the algorithm *OptimalSearchTree*, which implements the preceding strategy. *OptimalSearchTree* computes the root, $\text{Root}[i,j]$, of each tree T_{ij} , and the cost, $A[i,j]$, of T_{ij} . An optimal binary search tree T for all the keys, namely $T_{0,n-1}$, can be easily constructed using recursion from the two-dimensional array $\text{Root}[0:n-1,0:n-1]$.

```

procedure OptimalSearchTree( $P[0:n - 1], Q[0:n], Root[0:n - 1, 0:n - 1]$ ,
 $A[0:n - 1, 0:n - 1]$ )
Input:  $P[0:n - 1]$  (an array of probabilities associated with successful
searches)
 $Q[0:n]$  (an array of probabilities associated with unsuccessful
searches)
Output:  $Root[0:n - 1, 0:n - 1]$  ( $Root[i,j]$  is the key of the root node of  $T_{ij}$ )
 $A[0:n - 1, 0:n - 1]$  ( $A[i,j]$  is the cost  $A(T_{ij})$  of  $T_{ij}$ )
for  $i \leftarrow 0$  to  $n - 1$  do
     $Root[i,i] \leftarrow i$ 
     $Sigma[i,i] \leftarrow p[i] + q[i] + q[i+1]$ 
     $A[i,i] \leftarrow Sigma[i,i]$ 
endfor
for  $Pass \leftarrow 1$  to  $n - 1$  do //  $Pass$  is one less than the size of the optimal
// trees  $T_{ij}$  being constructed in the given pass.
    for  $i \leftarrow 0$  to  $n - 1 - Pass$  do
         $j \leftarrow i + Pass$ 
        //Compute  $\sigma(p_i, \dots, p_j, q_i, \dots, q_{j+1})$ 
         $Sigma[i,j] \leftarrow Sigma[i,j - 1] + p[j] + q[j+1]$ 
         $Root[i,j] \leftarrow i$ 
         $Min \leftarrow A[i + 1,j]$ 
        for  $k \leftarrow i + 1$  to  $j$  do
             $Sum \leftarrow A[i,k - 1] + A[k + 1,j]$ 
            if  $Sum < Min$  then
                 $Min \leftarrow Sum$ 
                 $Root[i,j] \leftarrow k$ 
            endif
        endfor
         $A[i,j] \leftarrow Min + Sigma[i,j]$ 
    endfor
endfor
end OptimalSearchTree

```

In Figure 8.7, we illustrate the action of *OptimalSearchTree* for the optimal binary search tree described above. For convenience, we will change all the probabilities to frequencies, which, in the case we are considering, result by multiplying each probability by 100 to change all the numbers to integers. As we have remarked, working with frequencies instead of probabilities can always be done. In fact, when we are constructing optimal subtrees, it is actually frequencies that we are dealing with instead of probabilities. The optimal subtrees T_{ij} are built up starting from the base case T_{ii} , $i = 0, 1, 2, 3$. The Figure shows how the tables are built during each pass for $A(T_{ij}) = A[i,j]$, $Root(T_{ij}) = Root[i,j]$, and $Sigma[i,j] = p_i + \dots + p_j + q_i + \dots + q_{j+1} = Sigma[i,j - 1] + p_j + q_{j+1}$:

i	0	1	2	3	4
p_i	15	10	20	30	
q_i	5	5	0	5	10

$A[i,j]$				$Root[i,j]$				$Sigma[i,j]$									
i	j	0	1	2	3	i	j	0	1	2	3	i	j	0	1	2	3
0	25	*	*	*	*	0	0	*	*	*	*	0	25	*	*	*	*
1	15	*	*	*	*	1	1	*	*	*	*	1	15	*	*	*	*
2	25	*	*	25	*	2	2	*	2	2	*	25	*	25	*	*	
3	45					3	3		4	3							45
i	j	0	1	2	3	i	j	0	1	2	3	i	j	0	1	2	3
0	25	50	*	*	*	0	1	1	*	*	*	0	25	35	*	*	*
1	15	55	*	*	*	1	2	3	*	*	*	1	15	40	*	*	
2	25	90				2	2	3	4	2	2	25	65				
3	45					3	3		4	3							45
i	j	0	1	2	3	i	j	0	1	2	3	i	j	0	1	2	3
0	25	50	110	*	*	0	1	1	2	*	*	0	25	35	60	*	
1	15	55	110			1	2	3	4			1	15	40	60		
2	25	90				2	2	3	4			2	25	65			
3	45					3	3		4			3					45
i	j	0	1	2	3	i	j	0	1	2	3	i	j	0	1	2	3
0	25	50	110	195		0	1	1	2	3		0	25	35	60	100	
1	15	55	110			1	2	3	4			1	15	40	60		
2	25	90				2	2	3	4			2	25	65			
3	45					3	3		4			3					45

Building arrays $A[i,j]$, $Root[i,j]$, $Sigma[i,j]$ from inputs $P[0:3] = (15,10,20,30)$ and $Q[0:4] = (5,5,0,5,10)$ to *OptimalSearchTree*

Figure 8.7

Since *OptimalSearchTree* does the same amount of work for any input $P[0:n - 1]$ and $Q[0:n]$, the best-case, worst-case, and average complexities are all equal. Clearly, the number of additions made in computing *Sum* has the same order as the total number of additions made by *OptimalSearchTree*. Therefore, we choose the addition made in computing *Sum* as the basic operation. Since *Pass* varies from 1 to $n - 1$, i varies from 0 to $n - 1 - \text{Pass}$, and k varies from $i + 1$ to $i + \text{Pass}$, it follows that the total number of additions made in computing *Sum* is given by

$$\begin{aligned}
& \sum_{t=1}^{n-1} \sum_{i=0}^{n-1-t} \sum_{k=i+1}^{i+t} 1 \\
&= \sum_{t=1}^{n-1} (n-t)t \\
&= n \sum_{t=1}^{n-1} t - \sum_{t=1}^{n-1} t^2 \\
&= n \left[(n-1) \frac{n}{2} \right] - \left[(n-1)n \frac{(2n-1)}{6} \right] \in \Theta(n^3).
\end{aligned}$$

8.4 Longest Common Subsequence and Edit Distance

In this section we consider the problem of determining how close two given character strings are to one another. For example, in a spell checking situation, one of the strings might be a string contained in a text being created on a word processor, and another string might be a pattern string from a stored dictionary. If there is no exact match between the text string and any pattern string, then a number of pattern strings might be presented to the operator that are fairly close, in some sense, to the text string. As another example, we might be comparing two DNA strings to measure how close they match. There are a number of ways to define closeness of two strings. The *edit distance* is commonly used by search engines to find approximate matchings for a given text string entered by the user in a query in situations where an exact match is not found. The edit distance is also used by spell checkers. Roughly speaking, the edit distance between two strings is the minimum number of changes that need to be made (adding, deleting, or changing characters) to transform one string to the other. In this section we consider another closeness measure, namely, the longest common subsequence (LCS) contained in a text string and a particular pattern string. Both computing LCS and computing the edit distance are optimization problems satisfying the Principle of Optimality, and both can be solved using dynamic programming. However, the solution to the LCS problem is easier to understand since it has a simpler recurrence relation, which we now describe.

8.4.1 Longest Common Subsequence

Suppose $T = T_0 T_1 \dots T_{n-1}$ is a text string that we wish to compare to a pattern string $P = P_0 P_1 \dots P_{m-1}$, where we assume that the characters in each string are drawn from some fixed alphabet A . A *subsequence* of T is a string of the form $T_{i_1} T_{i_2} \dots T_{i_k}$, where $0 \leq i_1 < i_2 < \dots < i_k \leq n-1$. Note that a *substring* of T is a special case of a subsequence of T in which the subscripts making up the subsequence increase by one. For example, consider the pattern string *Cincinnati* and the text string *Cincinatti* (a common misspelling). You can easily check that the longest common subsequence of the pattern string and the text string has length 9 (just one less than the common length of both strings), whereas it takes two changes to transform the text string to the pattern string (so that the edit distance between the two strings is two).

We now describe a dynamic programming algorithm to determine the length of longest common subsequence of T and P . For simplicity of notation, we will assume that the strings are stored in arrays $T[0:n-1]$ and $P[0:m-1]$, respectively. For integers i

and j , we define $LCS[i,j]$ to be the length of the longest common subsequence of the substrings $T[0:i-1]$ and $P[0:j-1]$ (so that $LCS[n,m]$ is the length of the longest common subsequence of T and P). For convenience, we set $LCS[i,j] = 0$ if $i = 0$ or $j = 0$ (corresponding to empty strings). Note that $LCS[1,1] = 1$ if $T[0] = P[0]$, otherwise $LCS[1,1] = 0$. This initial condition is actually a special case of the following recurrence relation for $LCS[i,j]$

$$\begin{aligned} LCS[i,j] &= LCS[i-1,j-1] + 1 && \text{if } T[i-1] = P[j-1], \\ &= \max\{LCS[i,j-1], LCS[i-1,j]\} && \text{otherwise.} \end{aligned} \quad (8.4.1)$$

To verify (8.4.1), note first that if $T[i-1] \neq P[j-1]$, then a longest common subsequence of $T[0:i-1]$ and $P[0:j-1]$ might end in $T[i-1]$ or $P[j-1]$, but certainly not both. In other words, if $T[i-1] \neq P[j-1]$, then a longest common subsequence of $T[0:i-1]$ and $P[0:j-1]$ must be drawn from either the pair $T[0:i-2]$ and $P[0:j-1]$ or from the pair $T[0:i-1]$ and $P[0:j-2]$. Moreover, such a longest common subsequence must be a longest common subsequence of the pair of substrings from which it is drawn (that is, the principle of optimality holds). This verifies that

$$LCS[i,j] = \max\{LCS[i,j-1], LCS[i-1,j]\} \quad \text{if } T[i-1] \neq P[j-1]. \quad (8.4.2)$$

On the other hand, if $T[i-1] = P[j-1] = C$, then a longest common subsequence must end either at $T[i-1]$ in $T[0:i-1]$ or at $P[j-1]$ in $P[0:j-1]$ (or both), otherwise by adding this common value C to a given subsequence we would increase the length of the subsequence by one. Also, if the last term of a longest common subsequence ends at an index $k < i-1$ in $T[0:i-1]$ (so that $T[k] = C$), then clearly we achieve an equivalent longest common subsequence by swapping $T[i-1]$ for $T[k]$ in the subsequence. By a similar argument involving $P[0:j-1]$, when $T[i-1] = P[j-1]$, we can assume without loss of generality that a longest common subsequence in $T[0:i-1]$ and $P[0:j-1]$ ends at $T[i-1]$ and $P[j-1]$. But then removing these end points from the subsequence clearly must result in a longest common subsequence in $T[0:i-2]$ and $P[0:j-2]$, respectively (that is, the principle of optimality again holds). It follows that:

$$LCS[i,j] = LCS[i-1,j-1] + 1 \quad \text{if } T[i-1] = P[j-1]. \quad (8.4.3)$$

which, together with (8.4.2), completes the verification of (8.4.1).

The following algorithm is the straightforward row-by-row computation of the array $LCS[0:n, 0:m]$ based on the recurrence (8.4.1).

```
procedure LongestCommonSubseq( $T[0:n-1], P[0:m-1], LCS[0:n,0:m]$ )
Input:  $T[0:n-1], P[0:m-1]$  (strings)
Output:  $LCS[0:n,0:m]$  (array such that  $LCS[i,j]$  is length of the longest common
subsequence of  $T[0:i-1]$  and  $P[0:j-1]$ )
for  $i \leftarrow 0$  to  $n$  do //initialize for boundary conditions
   $LCS[i,0] \leftarrow 0$ 
endfor
for  $j \leftarrow 0$  to  $m$  do //initialize for boundary conditions
```

```

 $LCS[0,j] \leftarrow 0$ 
endfor
for  $i \leftarrow 1$  to  $n$  do // compute the row index by  $i$  of  $LCS[0:n,0:m]$ 
    for  $j \leftarrow 1$  to  $m$  do // compute  $LCS[i,j]$  using (8.4.1)
        if  $T[i - 1] = P[j - 1]$  then
             $LCS[i,j] \leftarrow LCS[i - 1,j - 1] + 1$ 
        else
             $LCS[i,j] \leftarrow \max(LCS[i,j - 1], LCS[i - 1,j])$ 
        endif
    endfor
endfor
end LongestCommonSubseq

```

Using the comparison of text characters as our basic operation, we see that *LongestCommonSubseq* has complexity in $\Theta(nm)$, which is a rather dramatic improvement over the exponential complexity $O(2^n m)$ brute-force algorithm that would examine each of the 2^n subsequences of $T[0:n - 1]$ and determine the longest subsequence that also occurs in $P[0:m - 1]$.

In Figure 8.8 we show the array $LCS[0:8, 0:11]$ output by *LongestCommonSubseq* for $T[0:7] = usbeeune$ and $P[0:10] = subsequence$. Note that *LongestCommonSubseq* determines the length of the longest common subsequence of $T[0:n - 1]$ and $P[0:m - 1]$, but does not output the actual subsequence itself. In the previous problem of finding the optimal parenthesization of a chained matrix product, it was not of much value to know the minimum number of multiplications required without determining the actual parenthesization that did the job. This is why we needed to compute the array *FirstCut* $[0:n - 1,0:n - 1]$ in order to be able to construct the optimal parenthesization. Similarly, in the optimal binary search tree problem, it was not of much value to know the average search complexity of the optimal search tree without actually determining the optimal search tree itself. That is why we kept track of the key $Root[i,j]$ in the root of the optimal binary search tree containing the keys $K_i < \dots < K_j$. However, in the LCS problem, knowing the actual common subsequence is not as important as knowing its length. For example, in a situation like spell checking, the subsequence is not as important as its length, since typically one would be given a list of pattern strings for correcting a misspelled word in the text that share subsequences exceeding a threshold length (dependent of the length of the strings), as opposed to exhibiting common subsequences. Nevertheless, it is interesting that a longest common sequence can be determined just from the array $LCS[0:n - 1,0:m - 1]$ (and $T[0:n - 1]$ and $P[0:m - 1]$) without the need to maintain any additional information.

One way to generate a longest common subsequence is to start at the bottom right-hand corner position (n,m) of the array LCS , and work your way backwards through the array to build the subsequence in reverse order. The moves are dictated by looking at how you get the value assigned to a given position when you used (8.4.1) to build the array LCS . More precisely, if you currently at position (i,j) in LCS , and $T[i - 1] = P[j - 1]$, then this common value is appended to the beginning of the string already generated (starting with the null string), and you move to position $(i - 1,j - 1)$ in LCS . On the other hand, if $T[i - 1] \neq P[j - 1]$, then you move to position $(i - 1,j)$ or $(i,j - 1)$ depending on

whether $LCS[i - 1, j]$ is greater than $LCS[i, j - 1]$ or not. In the case where $LCS[i - 1, j]$ is equal to $LCS[i, j - 1]$, then either move can be made. In the latter case, the two different choices might not only generate different longest common subsequences, but may also yield different longest common strings corresponding to these subsequences. For example, in Figure 8.8a we show the path generated using the go-left rule, where we always move to position $(i - 1, j)$ when $T[i - 1] \neq P[j - 1]$, whereas Figure 8.8b shows the path resulting by always moving up to position $(i, j - 1)$. The darker shaded positions (i, j) in these paths correspond to where $T[i - 1] = P[j - 1]$. These two paths yield the longest common strings *sbeune* and *useune*, respectively. When generating a path in LCS , a longest common subsequence is obtained when you reach a position where $i = 0$ or $j = 0$.

<i>LCS</i>	<i>S</i>	<i>u</i>	<i>b</i>	<i>s</i>	<i>e</i>	<i>q</i>	<i>u</i>	<i>e</i>	<i>n</i>	<i>c</i>	<i>e</i>
0	0	0	0	0	0	0	0	0	0	0	0
<i>u</i>	1	0	0	1	1	1	1	1	1	1	1
<i>s</i>	2	0	1	1	1	2	2	2	2	2	2
<i>b</i>	3	0	1	1	2	2	2	2	2	2	2
<i>e</i>	4	0	1	1	2	2	3	3	3	3	3
<i>e</i>	5	0	1	1	2	2	3	3	4	4	4
<i>u</i>	6	0	1	2	2	2	3	3	4	4	4
<i>n</i>	7	0	1	2	2	2	3	3	4	5	5
<i>e</i>	8	0	1	2	2	2	3	3	4	5	6

The matrix $LCS[0:8, 0:11]$ for the strings $T[0:7] = usbeeune$ and $P[0:10] = subsequence$, with the path in LCS generating the longest common string *sbeune* using the move-left on ties rule

Figure 8.8a

	<i>S</i>	<i>u</i>	<i>b</i>	<i>s</i>	<i>e</i>	<i>q</i>	<i>u</i>	<i>e</i>	<i>n</i>	<i>c</i>	<i>e</i>	
<i>LCS</i>	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
<i>u</i>	1	0	0	1	1	1	1	1	1	1	1	1
<i>s</i>	2	0	1	1	1	2	2	2	2	2	2	2
<i>b</i>	3	0	1	1	2	2	2	2	2	2	2	2
<i>e</i>	4	0	1	1	2	2	3	3	3	3	3	3
<i>e</i>	5	0	1	1	2	2	3	3	3	4	4	4
<i>u</i>	6	0	1	2	2	3	3	4	4	4	4	4
<i>n</i>	7	0	1	2	2	3	3	4	4	5	5	5
<i>e</i>	8	0	1	2	2	3	3	4	5	5	5	6

The path in LCS generating the longest common string *useune* the move-up on ties rule

Figure 8.8b

8.4.2 Edit Distance and Approximate String Matching

In practice, there are often misspellings in the text and it is useful when searching for a pattern string P in a text to find words that are approximately the same as P . In this section we formulate and solving this problem using dynamic programming. We will first consider the problem of determining whether a pattern string $P = p_1 \dots p_p$ is an k -approximation of a text string $T = t_1 \dots t_t$. Later, we will look at the problem of finding occurrences of substrings of T for which P is a k -approximation. The pattern string P is a k -approximate matching of the text string T if T can be converted to P using at most k operations involving one of

- (i) changing a character of T (substitution)
- (ii) adding a character to T (insertion)
- (iii) removing a character of T (deletion)

For example suppose P is the string “algorithm”

- (i) elgortihm \rightarrow algorithm (*substitution* of *e* with *a*)
- (ii) algorthm \rightarrow algorithm (*insertion* of letter *i*)
- (iii) lagorithm \rightarrow algorithm (*deletion* of letter *l*)

In the above example each string T differs from P in at most one character. Unfortunately, in practice more serious mistakes are made where the difference may involve multiple characters. We define the *edit distance*, $D(P, T)$, between P and T to be the minimum number of operations of substitution, deletion and insertion needed to convert T to P . For example, the string “algorithm” and “logarithm” have edit distance 3
 $\text{logarithm} \rightarrow \text{alogarithm} \rightarrow \text{algorithim} \rightarrow \text{algorithm}$.

Let $D[i, j]$ denote the editing distance between the substring $P[1:i]$ consisting of the first i characters of the pattern string P and $T[1:j]$ consisting of the first j characters of

the text string T . If $p[i] = t[j]$, then, $D[i,j] = D[i - 1, j - 1]$. Otherwise, consider an optimal intermixed sequence involving the three operations substitution, insertion and deletion that converts $T[1:j]$ into $P[1:i]$. The number of such operations is the edit distance. Note that in transforming T to P inserting a character into T is equivalent to deleting a character from P . For convenience we will perform the equivalent operation of deleting characters from P rather than adding characters to T . We can assume without loss of generality that the sequence of operations involving the first $i - 1$ characters of P and the first $j - 1$ characters of T are operated on first. To obtain a recurrence relation for $D[i,j]$, we examine the last operation. If the last operation is substitution of $t[j]$ with $p[i]$ in T , then $D[i,j] = D[i - 1, j - 1] + 1$. If the last operation is the deletion of $p[i]$ from P , then $D[i,j] = D[i - 1, j] + 1$. Finally, if the last operation is deletion of $t[j]$ from T , then $D[i,j] = D[i, j - 1] + 1$. The edit distance is realized by computing the minimum of these three possibilities. Observing that the edit distance between a string of size i and the null string is i , we obtain the following recurrence relation for the edit distance:

$$D[i,j] = \begin{cases} D[i-1, j-1], & p[i] = t[j], \\ \min \{D[i-1, j-1] + 1, D[i-1, j] + 1, D[i, j-1] + 1\}, & \text{otherwise.} \end{cases} \quad (8.4.4)$$

init. cond. $D[0,i] = D[0,i] = i$.

The design of a dynamic programming algorithm based on this recurrence and its analysis is similar to that given for the longest common subsequence problem, and we leave it to the exercises. We also leave it to the exercise to design an algorithm for finding the first occurrence or all occurrence of a substring of the text string T that is a k -approximation of the pattern string P .

8.5 Floyd's Algorithm

Dijkstra's shortest-path algorithm find paths from a single source vertex r to all other vertices of a weighted digraph $D = (V,E)$ and has complexity $\Theta(n^2)$. By repeated calls to either of Dijkstra's algorithm we obtain an algorithm for a shortest path between every pair of vertices u and v , having $\Theta(n^3)$ complexity. We now use dynamic programming to obtain a $\Theta(n^3)$ algorithm, Floyd's algorithm, (also known as the Floyd-Warshall algorithm), for finding a shortest path between every pair of vertices of a weighted digraph D . Unlike the all-pairs shortest paths algorithm based on repeated applications of Dijkstra's algorithm, Floyd's algorithm works even if some of the edges have negative weights, provided that there is no negative cycle (sum of the weights of the edges in the cycle is negative).

Consider a digraph D with vertex set $V = \{0, \dots, n - 1\}$ and edge set E , whose edges have been assigned a weighting w . For k a nonnegative integer $k \leq n - 1$, let V_k denote the subset of vertices $\{0, \dots, k - 1\}$ (by convention, $V_0 = \emptyset$). Let $S_k(i,j)$ denote the weight of a shortest path from i to j , whose interior vertices (if any) all lie in V_k (if no such path exists, then $S_k(i,j) = \infty$). (An *interior* vertex is a vertex of P different from either i or j .) Since, by definition, $S_0(i,j)$ is the weight of a shortest path from i to j containing no interior vertices, S_0 is equal to the *weight matrix* W defined as follows: $W(i,j) = w(e)$ if D contains an edge e from vertex i to vertex j , 0 if $i = j$, and ∞ otherwise. Note also that

$S_n(i,j)$ is the weight of a shortest path in G from i to j .

Let P be a path joining two distinct vertices, $i, j \in V$ whose interior vertices belong to V_{k+1} and suppose v is an interior vertex of P . It is easily verified that the Principle of Optimality holds, so that the subpath P_1 from i to v is a shortest path from i to v whose interior vertices belong V_{k+1} and the subpath P_2 from v to j is a shortest path from v to j whose interior vertices belong to V_{k+1} .

Now consider $S_{k+1}(i,j)$ and let P be a path that realizes this distance, i.e., P is a shortest path from i to j whose interior vertices belong to $V_{k+1} = \{0, \dots, k\}$. If k is not an interior vertex of P , then P is a shortest path from i to j whose interior vertices lie in V_{k-1} , so that $S_{k+1}(i,j) = S_k(i,j)$. On the other hand, if k is an interior vertex of P , then the interior vertices (if any) of the path P_1 from i to k and the interior vertices (if any) of the path P_2 from k to j all lie in V_{k-1} . Since P_1 is a shortest path from i to k and P_2 is a shortest path from k to j , we have $S_{k+1}(i,j) = S_k(i,k) + S_k(k,j)$. Thus, either $S_{k+1}(i,j) = S_k(i,j)$ or $S_{k+1}(i,j) = S_k(i,k) + S_k(k,j)$, depending on whether or not P contains vertex k . Since P is a shortest path i to j , it follows that the weight of P is equal to the minimum of these two values, yielding the following recurrence relation for $S_k(i,j)$.

$$S_{k+1}(i,j) = \min\{S_k(i,j), S_k(i,k) + S_k(k,j)\} \quad (8.5.1)$$

init. cond. $S_0(i,j) = W(i,j)$ for all $i, j \in V$.

Floyd's algorithm is based on recurrence relation (8.5.1). The matrix $S_{k+1}(i,j)$ keeps track of the weight of the shortest paths and not the shortest paths themselves. To keep track of the paths, we maintain another matrix $P_k(i,j)$ defined by

$$P_{k+1}(i,j) = \begin{cases} P_k(i,j) & \text{if } S_{k+1}(i,j) = S_k(i,j) \\ k & \text{otherwise} \end{cases}, \quad (8.5.2)$$

init. cond. $P_0(i,j) = 0$ for all $i, j \in V$.

Floyd's algorithm is illustrated in Figure 8.9 for a sample digraph D and weight function w .

A shortest path from i to j in G can then be reconstructed from P_n as follows: If $P_n(i,j) = 0$, then a shortest path from i to j is directly along the edge (i,j) ; otherwise, if $P_n(i,j) = k$, then k is an intermediate vertex of a shortest path from i to j and any other intermediate vertices in this shortest path can be obtained by recursively examining $P_n(i,k)$ and $P_n(k,j)$. For example, in Figure 8.9, a shortest path S from vertex 2 to vertex 4 can be reconstructed from the final matrix P_4 as follows. Since $P_4(2,4) = 1$, vertex 1 is an internal vertex of S . Now, $P_4(1,4) = -1$, so that there are no intermediate vertices of S between 1 and 4. Since $P_4(2,1) = 0$, vertex 0 is an intermediate vertex of S between 1 and 1. Now, $P_4(2,0) = -1$ and $P_4(0,1) = -1$, so that there are no intermediate vertices between 2 and 0 or between 0 and 1. Thus, a shortest path S from 2 to 4 is given by 2, 0, 1, 4. We now give pseudocode for Floyd's algorithm.

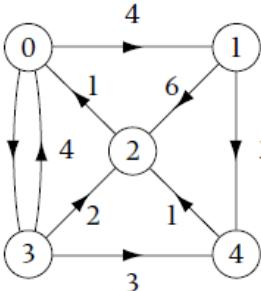
```

procedure Floyd( $W[0:n - 1,0:n - 1], P[0:n - 1,0:n - 1], S[0:n - 1,0:n - 1]$ )
Input:  $W[0:n - 1,0:n - 1]$  (weight matrix for a weighted digraph  $D$ )
Output:  $P[0:n - 1,0:n - 1]$  (matrix implementing shortest paths)
 $S[0:n - 1,0:n - 1]$  (distance matrix where  $S[u,v]$  is the weight of a
shortest path from  $u$  to  $v$  in  $G$ )
for  $i \leftarrow 0$  to  $n - 1$  do // initialize  $P$  and  $S$ 
    for  $j \leftarrow 0$  to  $n - 1$  do
         $P[i,j] \leftarrow -1$ 
         $S[i,j] \leftarrow W[i,j]$ 
    endfor
endfor
for  $k \leftarrow 0$  to  $n - 1$  do // update  $S$  and  $P$  using (8.5.1) and (8.5.2)
    for  $i \leftarrow 0$  to  $n - 1$  do
        for  $j \leftarrow 0$  to  $n - 1$  do
            if  $S[i,j] > S[i,k] + S[k,j]$  then
                 $P[i,j] \leftarrow k$ 
                 $S[i,j] \leftarrow S[i,k] + S[k,j]$ 
            endif
        endfor
    endfor
endfor
end Floyd

```

Remark

An alternate way to maintain the shortest paths is to store in $P[i,j]$ the first vertex encountered on the current path from i to j (see Exercise 8.57).



$$S_0 = W = \begin{pmatrix} 0 & 4 & \infty & 3 & \infty \\ \infty & 0 & 6 & \infty & 2 \\ 1 & \infty & 0 & \infty & \infty \\ 4 & \infty & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & 0 \end{pmatrix}$$

$$S_1 = \begin{pmatrix} 0 & 4 & \infty & 3 & \infty \\ \infty & 0 & 6 & \infty & 2 \\ 1 & 5 & 0 & 4 & \infty \\ 4 & 8 & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & 0 \end{pmatrix} \quad P_1 = \begin{pmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & 0 & -1 & 0 & -1 \\ -1 & 0 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

$$S_2 = \begin{pmatrix} 0 & 4 & 10 & 3 & 6 \\ \infty & 0 & 6 & \infty & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 4 & 8 & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & 0 \end{pmatrix} \quad P_2 = \begin{pmatrix} -1 & -1 & 1 & -1 & 1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & 0 & -1 & 0 & 1 \\ -1 & 0 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

$$S_3 = \begin{pmatrix} 0 & 4 & 10 & 3 & 6 \\ 7 & 0 & 6 & 10 & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 3 & 7 & 2 & 0 & 3 \\ 2 & 6 & 1 & 5 & 0 \end{pmatrix} \quad P_3 = \begin{pmatrix} -1 & -1 & 1 & -1 & 1 \\ 2 & -1 & -1 & 2 & -1 \\ -1 & 0 & -1 & 0 & 1 \\ 2 & 2 & -1 & -1 & -1 \\ 2 & 2 & -1 & 2 & -1 \end{pmatrix}$$

$$S_4 = \begin{pmatrix} 0 & 4 & 5 & 3 & 6 \\ 7 & 0 & 6 & 10 & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 3 & 7 & 2 & 0 & 3 \\ 2 & 6 & 1 & 5 & 0 \end{pmatrix} \quad P_4 = \begin{pmatrix} -1 & -1 & 3 & -1 & 1 \\ 2 & -1 & -1 & 2 & -1 \\ -1 & 0 & -1 & 0 & 1 \\ 2 & 2 & -1 & -1 & -1 \\ 2 & 2 & -1 & 2 & -1 \end{pmatrix}$$

$$S_5 = \begin{pmatrix} 0 & 4 & 5 & 3 & 6 \\ 4 & 0 & 3 & 7 & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 3 & 7 & 2 & 0 & 3 \\ 2 & 6 & 1 & 5 & 0 \end{pmatrix} \quad P_5 = \begin{pmatrix} -1 & -1 & 3 & -1 & 1 \\ 4 & -1 & 4 & 4 & -1 \\ -1 & 0 & -1 & 0 & 1 \\ 2 & 2 & -1 & -1 & -1 \\ 2 & 2 & -1 & 2 & -1 \end{pmatrix}$$

Stages of Floyd's algorithm for a sample weighted digraph G

Figure 8.6

8.6 The Bellman-Ford Algorithm

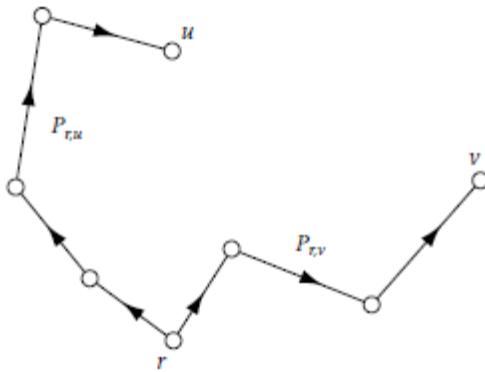
The Bellman-Ford algorithm, as with Dijkstra's algorithm discussed in Chapter 6, finds the best (smallest-weight) path in a directed weighted graph from a given vertex r to all other vertices. While Dijkstra's algorithm has better worst-case performance, the Bellman-Ford algorithm does not require that the weights on the edges are non-negative. The Bellman-Ford algorithm is not directly based on the dynamic programming paradigm, but has a similar flavor. The algorithm is based on making successive scans through all of the edges of the digraph, keeping track of the distance $Dist[v]$ from r to v using the best (smallest cost) path from r to v generated so far. After the k th stage of the algorithm, it turns out that $Dist[v]$ is no greater than the value of the smallest-cost path from r to v that has at most k edges. This optimality condition mirrors the Principle of Optimality that characterizes dynamic program solutions.

We initialize the array $Dist[0:n - 1]$ by setting $Dist[r] = 0$, and $Dist[v] = \infty$ for $v \neq r$. We also keep track of the shortest paths so far generated using a parent array implementation $Parent[0:n - 1]$ initialized by $Parent[r] = 0$, and $Parent[v] = \infty$ for $v \neq r$. The following key idea is the basis for the Bellman-Ford algorithm.

Key Fact

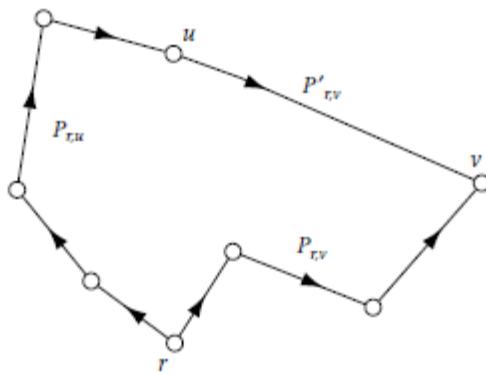
During a given scan through the edges, when we examine the edge uv , $Dist[v]$ is updated by checking whether or not a shorter path from r to v is obtained by replacing the current path by the path consisting of the current path from r to u together with the edge uv . This process of updating $Dist[v]$ is called *relaxing* the edge uv .

The relaxing of an edge is illustrated by Figure 8.7. In Figure 8.7a, we show the currently shortest path $P_{r,u}$ from r to u having cost $Dist[u]$, and the currently shortest path $P_{r,v}$ from r to v having cost $Dist[v]$, just before we consider the directed edge uv . In Figure 8.7b, by considering the edge uv , we now have a second possible path from r to v , namely, the path $P'_{r,v}$ obtained by adding the edge uv to the path $P_{r,u}$.



In Bellman–Ford Algorithm just before considering edge uv

(a)



In Bellman–Ford Algorithm after considering edge uv .
We compare weight of P_{rv} with weight of P'_{rv}
and update $Dist[v]$ with smaller of the two

(b)

Relaxing the edge uv in the Bellman-Ford algorithm

Figure 8.7

Note that the cost of $P'_{r,v}$ is the cost $Dist[u]$ of the path $P_{r,u}$ plus the cost $c(uv)$ of the edge uv . Hence, to see if we now have a shorter path from r to v , we simply need to check whether the cost of the path $P'_{r,v}$ is smaller than the cost of the path $P_{r,v}$, that is, we simply set

$$Dist[v] = \min\{\text{cost}(P_{r,v}), \text{cost}(P'_{r,v})\} = \min\{Dist[v], Dist[u] + c(uv)\}.$$

If we have now found a shorter path, that is, if $\text{cost}(P'_{r,v}) < \text{cost}(P_{r,v})$, then we set $Dist[v] = \text{cost}(P'_{r,v}) = Dist[u] + c(uv)$, and $Parent[v] = u$. As usual, the path from r to v is given (in reverse order) by the sequence

$$v, Parent[v], Parent[Parent[v]], \dots, Parent^k[v] = r.$$

Note that the current path from r to v may be updated many times during the same pass, since there may be many edges uv in D . If D has no negative cycles, it turns out that shortest paths will have been computed after $n - 1$ scans. After completion of the first $n - 1$ scans, procedure *BellmanFord* makes a final scan of the edges to check for negative cycles. If there is any edge uv such that $Dist[v] > Dist[u] + c(uv)$, then a negative cycle exists. In the case where D has negative cycles, that is, a cycle such that the total cost of the directed paths making up the cycle is negative, then by repeatedly going around the cycle we can make the cost of paths from r to vertices in this cycle as small as we desire. The following pseudocode for the Bellman-Ford algorithm follows directly from our discussion.

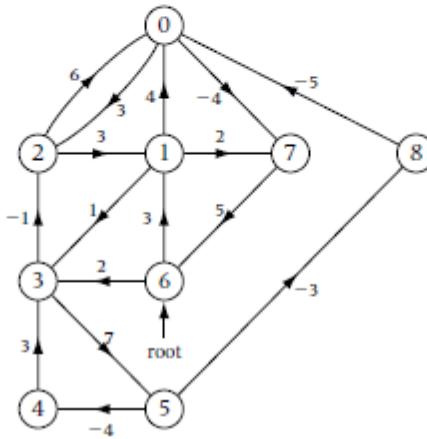
```

procedure BellmanFord( $D, r, c, Dist[0:n - 1], Parent[0:n - 1], NegativeCycle$ )
Input:  $D$  (a digraph with vertex set  $V = \{0, \dots, n - 1\}$  and edge set  $E$ )
          $c$  (a weighting of the edges with real numbers)
          $r$  (a vertex of  $D$ )
Output:  $Parent[0:n - 1]$  (an array implementing a shortest path tree rooted at  $r$ )
           $Dist[0:n - 1]$  (an array of distances from  $r$ )
           $NegativeCycle$  (a Boolean variable having the value true, if, and
                           only if, there exists a negative cycle)
for  $i \leftarrow 0$  to  $n - 1$  do           {initialize  $Dist[0:n - 1]$  and  $Parent[0:n - 1]$ }
     $Dist[i] \leftarrow \infty$ 
     $Parent[i] \leftarrow \infty$ 
endfor
 $Dist[r] \leftarrow 0$ 
 $Parent[r] \leftarrow 0$ 
for  $Pass \leftarrow 1$  to  $n - 1$  do           // update  $Dist[0:n - 1]$  and  $Parent[0:n - 1]$ 
    for each edge  $uv \in E$  do           // by scanning all the edges
        if  $Dist[u] + c(uv) < Dist[v]$  then
             $Parent[v] \leftarrow u$ 
             $Dist[v] \leftarrow Dist[u] + c(uv)$ 
        endif
    endfor
endfor
 $NegativeCycle \leftarrow \text{false.}$  {check for negative cycles}
for each edge  $uv \in E$  do
    if  $Dist[v] > Dist[u] + c(uv)$  then
         $NegativeCycle \leftarrow \text{true.}$ 
    endif
endfor
end BellmanFord

```

In procedure *BellmanFord* we did not specify the order in which the edges are scanned. In fact, any order will do, but the efficiency can be dramatically different for different orders. In Figure 8.8, during each pass, we scan the edges (u,v) in lexicographical order.

	Index	0	1	2	3	4	5	6	7	8	0	1	2	3	4	5	6	7	8
Pass																			
0	Dist:	∞	∞	∞	∞	∞	∞	0	∞	-1	∞	∞							
1	Dist:	∞	3	∞	2	∞	∞	0	∞	∞	∞	6	∞	6	∞	∞	-1	∞	∞
2	Dist:	1	3	1	2	5	9	0	5	6	∞	8	6	3	6	5	3	-1	1
3	Dist:	1	3	1	2	5	9	0	-3	6	∞	8	6	3	6	5	3	-1	0
4	Dist:	1	3	1	2	5	9	0	-3	6	6	∞	8	6	3	6	5	3	-1
5	Dist:	1	3	1	2	5	9	0	-3	6	6	6	∞	8	6	3	6	5	3
6	Dist:	1	3	1	2	5	9	0	-3	6	6	6	6	∞	8	6	3	5	3
7	Dist:	1	3	1	2	5	9	0	-3	6	6	6	6	5	∞	8	6	3	5
8	Dist:	1	3	1	2	5	9	0	-3	6	6	6	6	5	3	-1	∞	0	5



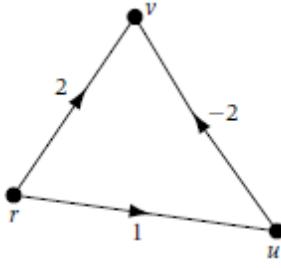
Action of procedure *BellmanFord* for a sample weighted digraph and root vertex $r = 6$, where edges (u,v) are scanned in lexicographical order.

Figure 8.8

Since procedure *BellmanFord* examines each edge n times, it has (best-case, average, and worst-case) complexity belonging to $\Theta(nm)$. The best-case and average performance of procedure *BellmanFord* can be improved by adding a flag to test whether any changes to *Dist* occur during a given pass. If no change occurs in any particular pass, then the algorithm can terminate. For example, for the digraph illustrated in Figure 8.8, only four passes are required. By adding the flag, the performance of procedure *BellmanFord* can be drastically different for different orderings of passing through the edges. To see why, consider the 3-vertex digraph shown in Figure 8.8, and note the difference between *Dist*[v] after the first pass for the two different orderings of the edges. In fact, it is not hard to give any example of a weighted digraph D on n vertices and two orderings of the edges such that procedure *BellmanFord* performs only two passes in one ordering, and the full n passes in the other ordering.

Key Fact

The correctness of either version (flag or no flag) of the procedure *BellmanFord* does not depend on the order in which the edges are scanned. However, with the flag added, the efficiency can vary drastically depending on the order in which the edges are scanned.



First pass of procedure *BellmanFord*, where edges are considered in the order rv , uv , ru would yield $Dist[v] = 2$, but the order ru , uv , rv would yield $Dist[u,v] = -1$.

Figure 8.9

The correctness of procedure *BellmanFord* involves establishing the loop invariant stated in the following lemma.

Lemma 8.6.1

After completing k iterations of the *Pass for* loop, for each vertex v , $Dist[v]$ is no larger than the minimum length $L_{v,k}$ of a path from r to v having at most k edges (where $L_{v,k} = \infty$ if no such path exists), $k = 0, \dots, n - 1$.

Proof We prove the lemma by induction on the number k of passes that have been completed. Clearly, the lemma is true before any pass has been completed, so that the basis step is established. Now assume that after k passes have been completed, for each vertex v , $Dist[v]$ is no larger than the minimum length $L_{v,k}$ of a path from r to v having at most k edges. Consider any vertex v for which there is a path from r to v having at most $k + 1$ edges (the lemma is trivially true if there is no such path), and let P be such a path of minimum length $L_{v,k}$. Let uv be the terminal edge of P , and let Q be the subpath of P from r to u . Since Q has at most k edges, the length of Q is at least $L_{u,k}$. Thus, the length of P is at least $L_{u,k} + c(uv)$. But, by induction assumption, after the k th pass, $Dist[u]$ is at most $L_{u,k}$. Therefore, after the $(k + 1)$ st pass, we have $Dist[v] \leq Dist[u] + c(uv) \leq L_{u,k} + c(uv) \leq \text{length of } P = L_{v,k}$. Thus, the loop invariant holds after $k + 1$ passes. ■

When there are no negative cycles, a shortest path contains at most $n - 1$ edges. Hence, the correctness of *BellmanFord* when there are no negative cycles follows immediately from Lemma 8.6.1. We leave the correctness proof of *BellmanFord* when there are negative cycles as an exercise.

8.7 Closing Remarks

Floyd's algorithm is also known as the Roy–Warshall algorithm, the Roy–Floyd algorithm, or the WFI algorithm. Floyd published his algorithm in 1962, but it was

essentially the same as algorithms published by Bernard Roy in 1959 and by Stephen Warshall's in 1962 for finding the transitive closure of a graph.

The dynamic programming paradigm was first formulated by Richard Bellman. His interest was in solving optimization problems that involved making a sequence of decisions (resulting in problem states x_0, x_1, \dots) that were guaranteed to yield an optimal solution (highest value) V for a value function F associated with a given problem. He stated the following condition that formed the basis for a recursive solution to the problem.

Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

The *Bellman Equation* results from expressing the optimal value function at a given initial state x_0 in terms of its value at the state resulting from making the initial decision. Suppose $D(x_i)$ denotes the set of possible decisions available at stage x_i , $i = 0, 1, \dots$, and let T denote that transition function that determines the next stage x_{i+1} resulting from making the decision $d_i \in D(x_i)$ at stage x_i , $i = 0, 1, \dots$. Then the principle of optimality results in the following recursive Bellman Equation for the optimal value V

$$V(x_0) = \max\{ F(x_0, d_0) + V(T(x_0, d_0)) \} \text{ over all } d_0 \in D(x_0). \quad (8.7.1)$$

The recursive expression then is solved in a bottom-up resolution of the recursion in the manner that we have described in this chapter.

The Bellman Equation has been applied in a wide variety of optimization problems. In many applications, especially in financial problems, the term $V(T(x_0, d_0))$ in (8.7.1) is multiplied by a so-called *discount factor* β .

In Exercise 8.16 we develop a dynamic programming solution to the Traveling Salesman Problem (TSP). TSP involves finding a minimum length tour of n cities, starting and ending at a given city. TSP is a rather famous problem, and a good deal of research has been on this problem. Unfortunately, no polynomial algorithm has been found for its solution, and in fact it is NP-hard (so that most researchers believe no polynomial solution for TSP will ever be found). The dynamic programming solution discussed in Exercise 8.16 has complexity in $\Theta(n2^n)$, which at least is rather better than the brute force method of enumerating the $(n - 1)!$ possible tours.

Exercises

Section 8.1 Optimization Problems and the Principle of Optimality

- 8.1 Suppose the matrix $C[0:n - 1 0:n - 1]$ contains the cost of $C[i,j]$ of flying directly from airport i to airport j . Consider the problem of finding the cheapest flight from i to j where we may fly to as many intermediate airports as desired. Verify that the Principle of Optimality holds for the minimum cost flight. Derive a recurrence relation based on the principle of optimality.

- 8.2 Does the Principle of Optimality hold for costliest trips (no revisiting of airports, please)? Discuss.
- 8.3 Does the Principle of Optimality hold for coin changing? Discuss with various interpretations of the *Combine* function.

Section 8.2 Optimal Parenthesization for Computing a Chained Matrix Product

- 8.4 Given the matrix product $M_0M_1\dots M_{n-1}$ and a 2-tree T with n leaves, show that there is a unique parenthesization P such that $T = T(P)$.
- 8.5 Give pseudocode for *ParenthesizeRec* and analyze its complexity.
- 8.6 Show that the complexity of *OptimalParenthesization* is in $\Theta(n^3)$.
- 8.7 Using *OptimalParenthesization*, find an optimal parenthesization for the chained product of five matrices with dimensions $6 \times 7, 7 \times 8, 8 \times 3, 3 \times 10, 10 \times 6$.
- 8.8 Write a program implementing *OptimalParenthesization* and run it for some sample inputs.

Section 8.3 Optimal Binary Search Trees

- 8.9 Use dynamic programming to find an optimal binary search tree for the following probabilities, where we assume that the search key is in the search tree, that is, $q_i = 0, i = 0, \dots, n$:

keys	i	0	1	2	3
probabilities	p_i	.4	.3	.2	.1

- 8.10 Use dynamic programming to find an optimal search tree for the following probabilities:

i	0	1	2	3	4	5
p_i	.2	.1	.2	.05	.05	
q_i	.05	0	.25	0	.1	0

- 8.11 a. Design and analyze a recursive algorithm that computes an optimal binary search tree T for all the keys from the two-dimensional array $Root[0:n - 1, 0:n - 1]$ generated by *OptimalSearchTree*.
- b. Show the action of your algorithm from part (a) for the instance given in Exercise 8.9.
- 8.12 a. Give a set of probabilities p_0, \dots, p_{n-1} (assume a successful search so that $q_0 = q_1 = \dots = q_n = 0$), such that a completely right-skewed search tree T (the left child of every node is nil) is an optimal search tree with respect to these probabilities.
- b. More generally, prove the following induction on n : If T is any given binary search tree with n nodes, then there exists a set of probabilities p_0, \dots, p_{n-1} such that T is the (unique) optimal binary search tree with respect to these probabilities.

8.4 Longest Common Subsequence

- 8.13 Show the array $LCS[0:9, 0:10]$ that is built by *LongestCommonSubseq* for $T[0:8] = \text{alligator}$ and $P[0:9] = \text{algorithms}$.

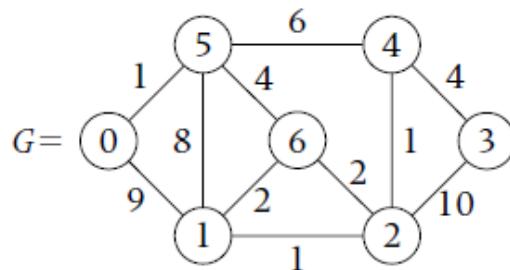
8.14 By following various paths in the array $LCS[0:8,0:11]$ given in Figure 8.8, find all longest common subsequences of the strings *usbeeune* and *subsequence*.

8.15 Design and analyze an algorithm that generates all longest common subsequences given the input array $LCS[0:n - 1, 0:m - 1]$.

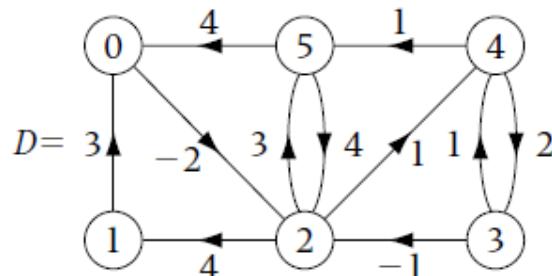
8.16 Write a program that implements *LongestCommonSubseq* and run it for some sample inputs.

8.5 Floyd's Algorithm

8.17 Show the action of Floyd's algorithm for the following graph.



8.18 Given the following weighted digraph D , show the matrices $S_k[0:5,0:5]$, $P_k[0:5,0:5]$, $k = 0, \dots, 5$ as computed by Floyd's algorithm. Describe how the matrix P_5 can be used, for example, to find the shortest path from vertex 1 to vertex 3.



8.19 Give pseudocode for an algorithm that computes the shortest path from i to j from the matrix $P[0:n - 1, 0:n - 1]$ generated by Floyd's algorithm.

8.20 Alter the procedure Floyd to store in $P[i,j]$ the first vertex encountered on the current path from i to j . Then repeat the previous exercise for this altered matrix $P[0:n - 1, 0:n - 1]$.

8.21 Prove that Floyd's algorithm works even if some of the edges have negative weights, provided that there is no negative weight cycle.

8.22 Show how Floyd's algorithm can be used to obtain longest paths between every pair of vertices in a dag.

8.6 The Bellman-Ford Algorithm

8.23 a. Redo the action of procedure *BellmanFord* shown in Figure 8.8 when $r = 2$.

- b. Redo the action of procedure *BellmanFord* shown in Figure 8.8 when $r = 6$ and where the cost of the edge $(3,2)$ is changed to -2 .
- 8.24 a. Design a modification of procedure *BellmanFord* that improves the best-case and average performance by terminating if no changes to *Dist* occur during a pass.
- b. Analyze the best-case and worst-case complexity of your algorithm in part (a) in terms of the parameters n and m . Explicitly exhibit input digraphs achieving the best-case and worst-case complexities, respectively.
- c. Repeat part (b) in terms of the single parameter n .
- 8.25 A high-level description of procedure *BellmanFord* is given in Section 8.6. Give a description of procedure *BellmanFord* for the following implementations of the digraph.
- adjacency matrix
 - adjacency lists
- 8.26 Show that if a weighted digraph contains no negative or zero cycles, then a shortest path contains at most $n - 1$ edges. Conclude the correctness proof of procedure *BellmanFord* when there are no negative cycles.
- 8.27 Give the correctness proof of procedure *BellmanFord* when there are negative cycles.
- 8.28 Modify procedure *BellmanFord* to output a negative cycle, when one exists.
- 8.29 Consider the version of procedure *BellmanFord*, which checks each pass for any change in $Dist[0:n - 1]$. For this version of procedure *BellmanFord*, the order in which the edges are scanned can significantly affect the worst-case performance of the algorithm. Give an example of a weighted digraph, and two orderings *A* and *B* of the edges, such that if ordering *A* is used during each pass, procedure *BellmanFord* performs only two passes, but if ordering *B* is used during each pass, procedure *BellmanFord* performs n passes.

Additional Problems

- 8.30 Consider a sequence of n distinct integers. Design and analyze a dynamic programming algorithm to find the length of the longest increasing subsequence. For example, consider the sequence:
- | | | | | | | | | | | | |
|----|----|---|---|----|-----|----|----|----|----|----|---|
| 45 | 23 | 9 | 3 | 99 | 108 | 76 | 12 | 77 | 16 | 18 | 4 |
|----|----|---|---|----|-----|----|----|----|----|----|---|
- The longest increasing subsequence is 3 12 16 18, having length 4.
- 8.31 The 0/1 Knapsack problem is NP-hard when the input is measured in binary. However, when the input is measured in unary. Design and analyze a dynamic programming solution to the 0/1 Knapsack problem, with positive integer capacity and weights which is quadratic in $C + n$, where C is the capacity and n is the number of objects. HINT: Let $V[i,j]$ denote the maximum value that can be placed in a knapsack of capacity j using objects drawn from $\{b_0, \dots, b_{i-1}\}$. Use the principle of optimality to find a recurrence relation for $V[i,j]$.
- 8.32 Design and analyze a dynamic programming solution to coin-changing problem under similar assumptions to that in the previous exercise.
- 8.33 Given n integers, the partition problem is to find a bipartition of the integers into two subsets having the same sum or determine that no such bipartition exists. Design and analyze a dynamic programming algorithm for solving the partition problem.

10

NP-Complete Problems

It is a curious phenomenon that the (worst-case) complexity of known sequential algorithms for most of the commonly encountered problems in computer science fall into the following two main categories:

1. bounded above by low degree polynomials,
2. bounded below by super-polynomial functions. (A function f is *super-polynomial* if $f \in \Omega(n^k)$ for all integers $k \geq 1$.)

Problems in graph theory illustrate this situation well. For example, we have seen $O(n^2)$ and $O(n^3)$ algorithms, respectively, for the problems of finding minimum cost spanning trees and finding the shortest paths between every pair of vertices in weighted graphs, whereas the best-known algorithms for the Hamiltonian cycle problem and the vertex-coloring problem have super-polynomial complexity. On the other hand, no one has been able to show that super-polynomial lower bounds for these problems exist (that is, lower bounds on the complexity of *any* algorithms for these problems). Problems such as the Hamiltonian cycle and graph coloring problems (or, more precisely, their decision versions as defined in the next section) belong to a large class of fundamental decision problems called *NP-complete problems* (see Section 10.3). It turns out that if any NP-complete problem is solvable by a polynomial (time) algorithm, then they all are so solvable. It has long been conjectured that no polynomial algorithm exists for any of the NP-complete problems. This conjecture is arguably the most important open question in theoretical computer science.

In this chapter we give a brief introduction to the class of NP-complete problems. Our treatment will be somewhat intuitive and informal. In particular, we will not establish the formal machinery necessary to prove the famous result of Cook that NP-complete problems exist, but refer the interested student to the references for a completely rigorous treatment.

We also expand our earlier discussion of the class NC and introduce the notion of P-completeness. P-completeness can be viewed as the analog for parallel computation of NP-completeness for sequential computation.

10.1 The Classes P and NP

We consider a problem to be *tractable* if it can be solved by a worst-case polynomial (time) algorithm. Problems having superpolynomial worst-case complexity are considered *intractable*. Throughout this chapter when we use the term *computing time* (or *complexity*) we will always mean the worst-case computing time (complexity).

Superpolynomial algorithms are computationally infeasible to implement in the worst case. Even though we have called a problem tractable if it can be solved by a polynomial algorithm, such an algorithm may still be computationally infeasible if it has complexity $\Omega(n^k)$ where k is large. For example, an algorithm having complexity n^{64} will not finish in

our lifetime even for $n = 2$. Nevertheless, it is standard in the theory of algorithms to regard problems solved by polynomial algorithms as tractable, and it becomes important to identify those problems. Besides, if a polynomial algorithm has been shown to exist for a problem (even one of high degree), then there may be hope that a more practical polynomial algorithm of relatively small degree can be found for the problem.

In the theory of complexity, it is convenient and useful to restrict attention to decision problems, that is, problems whose solutions simply output yes or no (0 or 1). Let P denote the class of decision problems that are solvable by algorithms having polynomial (worst-case) complexity.

10.1.1 Input Size

When considering membership in P , we must be very careful about how input size is measured. For example, when considering a decision problem having input parameters that are numeric quantities, the size of these numeric quantities is taken into account when measuring input size. We usually use the number of digits in the binary representation of the sum of the appropriate sizes of these numeric quantities. For example, consider an input $(w_0, w_1, \dots, w_{n-1}), (v_0, v_1, \dots, v_{n-1})$ and C to the 0/1 Knapsack problem. If we let $m = \sum_{i=0}^{n-1} (w_i + v_i)$ and d denote the number of digits of m , then the size of this input is $n + d \in \Theta(n + \log m)$.

10.1.2 Optimization Problems and Their Decision Versions

Decision problems occur naturally in many contexts. For example, an important decision problem in graph theory is to determine whether a clique of size k occurs in a given input graph. In mathematical logic, a fundamental decision problem is whether the Boolean variables in a given Boolean formula can be assigned truth values that make the entire formula true. In addition to such decision problems that are of interest in their own right, optimization problems usually give rise to associated decision problems. Moreover, restricting attention to the decision version of an optimization problem sometimes can be done without loss of generality (up to polynomial factors). In other words, the decision version of an optimization problem may have the property that a polynomial solution to the optimization problem exists if and only if a polynomial solution to the associated decision problem exists (the “only if” part is always true).

To illustrate, consider the optimization problem of determining the chromatic number $\chi = \chi(G)$ of a graph G on n vertices, that is, finding the minimum number of colors necessary to (properly) color the vertices of a graph G such that no two adjacent vertices get the same color. Given an integer k the associated decision problem asks whether a proper k -coloring of G exists. Clearly, a solution to the optimization problem immediately implies a solution to the decision problem: A proper k -coloring exists if, and only if, $k \geq \chi(G)$. On the other hand, a polynomial solution to the decision problem also implies a polynomial solution to the original problem: Simply call the decision problem for $k = 1, 2, \dots$ until a yes is returned (which will happen after at most n calls).

As another example, consider the 0/1 Knapsack problem with input weights w_0, w_1, \dots, w_{n-1} , values v_0, v_1, \dots, v_{n-1} , and capacity C . The decision version of this problem adds an integer input parameter k and asks whether a collection of the objects can be placed in the knapsack whose total value is at least k . Here again, it is obvious that

the ordinary version of the 0/1 Knapsack problem yields a solution of the decision version with no extra work. However, in the case of noninteger (rational) weights and values, there is no obvious polynomial number of decision problems whose answers yield the optimal solution to the 0/1 Knapsack problem. The situation improves if we restrict attention to the case of *integer* weights and values. Then each possible solution has an integer value, with the largest possible value equal to $m = v_0 + v_1 + \dots + v_{n-1}$. Unlike the graph-coloring problem, we cannot simply call the decision problem for $k = m, m-1, \dots$ until a *yes* is returned, since this will take an exponential number (with respect to input size) of calls in general. However, using a binary search strategy, by making $\log_2 m$ calls to the decision problem for suitable values of k we can determine the optimal solution in polynomial time (see Exercise 10.1).

Remark

It is not known in general whether a polynomial solution to the 0/1 Knapsack decision problem implies a polynomial solution to the 0/1 Knapsack (optimization) problem.

10.1.3 The Class NP

The class NP (Nondeterministic Polynomial) consists of those decision problems for which *yes* instances can be solved in polynomial time by a nondeterministic algorithm. Formal treatments of the class NP and nondeterministic algorithms are usually given in terms of Turing Machines and formal-languages. We will proceed less formally, and base our discussion on the intuitive notion of “guessing and verifying.” In this context, we give the following high level pseudocode for a generic Nondeterministic Polynomial algorithm *NPAAlgorithm*. Step 1 in *NPAAlgorithm* is the nondeterministic step, and step 2 is the deterministic step.

function *NPAAlgorithm*(*A,I*)

Input: *A* (a decision problem), *I* (an instance of problem *A*)

Output: “yes” or “don’t know”

1. In polynomial time, guess a candidate certificate *C* for the problem *A*
2. In polynomial time, use *C* to deterministically verify that *I* is a yes instance.

if a yes instance is verified in step 2 **then**

return (“yes”)

else

return (“don’t know”)

endif

end *NPAAlgorithm*

NPAAlgorithm is consider correct if whenever *I* is a yes instance, then a certificate can be produced in step 1 (by “perfect” guessing) that verifies that *I* is a yes instance, whereas if *I* is a no instance, then “don’t know” is always returned. Be careful to note that a given running of *NPAAlgorithm* might return “don’t know” even on a yes instance. The correctness requirement says that if *I* is a yes instance, then a certificate verifying *can be* produced in step 2 assuming the appropriate guess is made.

For example, if the decision problem is whether a graph can be properly colored using at most *k* colors for a given instance graph *G*, a nondeterministic algorithm would simply

guess a color from $\{1, 2, \dots, k\}$ for each vertex. Perfect guessing would produce a proper k -coloring for a graph G if it exists, and this coloring would then be a certificate that can be used to deterministically verify that G is a yes instance.

A decision problem is in the class NP if it can be solved by a nondeterministic algorithm. In the k -coloring problem, clearly a certificate coloring can be verified to be a proper coloring in polynomial time.

Note that $P \subseteq NP$, since if A is a polynomial algorithm for a decision problem, then a nondeterministic algorithm for the problem is obtained by guessing any succinct (computable in polynomial time) certificate S for a given input I , and then using A to solve the problem instance I completely ignoring S .

10.1.4 Some Sample Problems in NP

We illustrate the definition of NP with some sample problems in NP. In each of these examples, it will be obvious that a certificate can be produced and verified for yes instances in polynomial time. A *clique* Q of size k in a graph G is a subset of k vertices that are pair-wise adjacent.

Clique

Given the graph $G = (V, E)$ and the integer k , does there exist a clique of size k in G ?

A nondeterministic algorithm guesses a candidate solution by choosing a set of k vertices. The verification that the set of vertices is distinct and forms a clique can be done in polynomial time.

Sum of Subsets

Given the input integers $\{a_0, a_1, \dots, a_{n-1}\}$, and the target sum C , is there a subset of the integers whose sum equals C ?

The nondeterministic algorithm guesses a candidate solution by choosing a subset of the integers $\{a_{i_1}, a_{i_2}, \dots, a_{i_m}\}$. The verification that $a_{i_1} + a_{i_2} + \dots + a_{i_m} = C$ can be done in polynomial time.

Hamiltonian Cycle

Given the graph $G = (V, E)$, does G have a Hamiltonian cycle?

A nondeterministic algorithm guesses a Hamiltonian cycle by choosing a sequence of vertices $v_0, v_1, \dots, v_{n-1}, v_0$. Verification that the sequence of vertices is a Hamiltonian cycle can be done in polynomial time.

Showing that the problems in this section were in NP was relatively easy. However, there are problems whose membership in NP is quite difficult to establish. A notable example is prime testing, which asks whether a given integer n is prime. Note that a *yes* instance means that the answer is *no* to the question of whether n has factors, and the nonexistence of factors has no obvious certificate. However, it was shown by Pratt that a succinct certificate can be produced by a nondeterministic algorithm that can be verified

in polynomial time (see Exercise 10.2). In 2002, it was shown that primality testing is actually in P.

The question of whether P = NP can be viewed as asking whether adding nondeterminism allows us to solve a class of problems in polynomial time that are not otherwise so solvable. It seems reasonable that adding the luxury of making good guesses should really help, and most computer scientists believe that P \neq NP. For example, the question of whether or not a search element occurs in a list of size n can be solved in constant time by a nondeterministic algorithm, whereas any deterministic algorithm solving this problem has linear worst-case complexity. Reinforcement of the belief that P \neq NP can be found in the fact that there are thousands of varied problems in NP that have resisted all attempts to find polynomial solutions.

10.2 Reducibility

One reason that polynomial complexity is such a convenient measure is that polynomials have nice closure properties. For example, the composition $f(x) = g(h(x))$ of two polynomials g and h is yet another polynomial; in other words, a polynomial of a polynomial is a polynomial. Polynomials have other nice closure properties: The sum (difference, product) of a polynomial and a polynomial is a polynomial. These closure properties of polynomials, together with the fact that polynomial complexity is the standard measure of efficiency in the theory of complexity, allows a nice description of when a given decision problem A is not harder to solve than another decision problem B ; namely, the solution to B should yield the solution to A with at most a polynomial factor of additional work. We make this precise in the following definition.

Definition 10.2.1

Given two decision problems A and B , we say that A is (polynomially) *reducible* to B , denoted $A \propto B$, if there is a mapping f from the inputs to problem A to the inputs to problem B , such that

1. f can be computed in polynomial time (that is, there is a polynomial algorithm that for input I to problem A outputs the input $f(I)$ to problem B), and
2. the answer to a given input I to problem A is *yes* if, and only if, the answer to the input $f(I)$ to problem B is *yes*.

In particular, condition (1) of Definition 10.2.1 implies that there exists a polynomial $p(n)$ such that $|f(I)| \in O(p(|I|))$, for any input I to algorithm A , where $|I|$ denotes the size of I .

The following two propositions are easy consequences of the closure properties of polynomials.

Proposition 10.2.1

The relation \propto is transitive; that is, if $A \propto B$ and $B \propto C$ then $A \propto C$. □

Proposition 10.2.2

If $A \propto B$ and B has polynomial complexity, then so does A . □

We now illustrate the notion of reducibility by several examples. Further examples are given in Section 10.4.

Example 10.2.1: Hamiltonian Cycles \propto Traveling Salesman

Given the graph $G = (V, E)$, $|V| = n$, define the following weighting w on K_n on the complete graph on n vertices: $w(e) = 1$ if $e \in E$, otherwise $w(e) = 2$. Clearly, w can be computed in polynomial time. We now map G to the instance $f(G) = (K_n, w, k = n)$ of Traveling Salesman (so that $p(n) = n^2$). Clearly, G has a Hamiltonian cycle if, and only if, K_n with weighting w has a tour of cost no more than n .

Example 10.2.2: Clique \propto Vertex Cover \propto Independent Set \propto Clique

A *vertex cover* C in a graph G is a set of vertices with the property that every edge in G is incident to at least one vertex in C . The vertex cover problem asks whether a graph G has a vertex cover of size k . An *independent set* of vertices in a graph G is a set of vertices no two of which are adjacent. The independent set problem asks whether a graph G has an independent set of size k . Since the relation \propto is transitive, Example 10.2.2 says that the three problems, clique, vertex cover, and independent set, are each polynomial reducible to one another. Given the graph $G = (V, E)$, recall that the complement $\bar{G} = (V, \bar{E})$ is the graph with the same vertex set V , such that $e \in \bar{E}$ if, and only if, $e \notin E$. Given G , clearly the complement \bar{G} can be computed in polynomial time. The polynomial reductions are consequences of the following proposition, whose proof is left as an exercise.

Proposition 10.2.3

Given the graph $G = (V, E)$, then the following three conditions are mutually equivalent conditions on a subset of vertices $U \subseteq V$

1. U is a clique.
2. U is an independent set of vertices in the complement $\bar{G} = (V, \bar{E})$.
3. The set $V - U$ is a vertex cover in the complement $\bar{G} = (V, \bar{E})$. □

The reducibility illustrated in the previous examples was easy to show and not very surprising, since the problems involved were highly related. In Section 10.4 we give examples of disparate problems A and B where $A \propto B$.

The following definition makes precise the notion of two problems being polynomially equivalent to one another.

Definition 10.2.2

Two decision problems A and B are *polynomially equivalent to one another* if $A \propto B$ and $B \propto A$.

Since the relation \propto is clearly reflexive, it follows from Proposition 10.2.1 that the relation of polynomial equivalence is an equivalence relation on the set of decision problems.

10.3 NP-Complete Problems: Cook's Theorem

S. A. Cook raised a fundamental question: Is there a problem in NP that is as hard (up to polynomial factors) as every other problem in NP? In other words, is there a problem C in NP such that if A is *any* problem in NP, then $A \leq C$? If such a problem exists, then it is called *NP-complete*. (Note that all NP-complete problems are automatically polynomially equivalent to one another.) In one of the most celebrated results in theoretical computer science, Cook showed in 1971 that the satisfiability problem for Boolean expressions (as defined presently) is NP-complete. About the same time, Levin showed that a certain tiling problem was NP-complete. The following proposition follows easily from the transitivity of the relation \leq .

Proposition 10.3.1

Suppose A is in NP, and B is NP-complete. If $B \leq A$, then A is NP-complete. □

In the time since Cook's result appeared, hundreds of NP-complete problems have been identified. Despite considerable effort, no polynomial algorithm has been found for any NP-complete problem. This lack of success reinforces the belief that $P \neq NP$. The following corollary of Proposition 10.3.1 shows that it is unlikely that a polynomial algorithm for any NP-complete problem will ever be found.

Corollary 10.3.2

If any NP-complete problem A also belongs to P, then $P = NP$. □

A problem A (not necessarily a decision problem) is *NP-hard* if it is as hard to solve as any problem in NP. More precisely, A is NP-hard if a polynomial time solution for A would imply $P = NP$. For example, if the decision version of an optimization problem is NP-complete, then the optimization problem itself is NP-hard. Note that the NP-complete problems are precisely those problems in NP that are NP-hard.

To describe Cook's result, we need to recall some terms from first-order logic (see also Exercise 10.23). As with our pseudocode conventions, a *Boolean* (logical) variable is a variable that can only take on two values, true (T) or false (F). Given a Boolean variable x we denote by \bar{x} the variable has that the value T if, and only if, x has the value F . Boolean variables can be combined using logical operators *and* (denoted by \wedge and called *conjunction*), *or* (denoted by \vee and called *disjunction*), *not* (denoted by \neg and called *negation*), and parenthesization, to form Boolean expressions (formulas). A *literal* in a Boolean formula is of the form x or \bar{x} , where x is a Boolean variable. Note that $\neg x$ has the same truth value as \bar{x} . A Boolean formula is said to be in *conjunctive normal form* (CNF) if it has the form $C_1 \wedge C_2 \wedge \dots \wedge C_m$, where each clause C_i is a disjunction of $n(i)$ literals, $i = 1, \dots, m$.

A Boolean formula is called *satisfiable* if there is an assignment of truth values to the literals occurring in the formula that makes the formula evaluate to true. For example, the CNF formula

$$(x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2) \wedge (x_1 \vee x_3) \wedge (\bar{x}_1 \vee x_3)$$

is satisfiable ($x_1 = x_2 = x_3 = T$), whereas the CNF formula

$$(x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge (x_1 \vee \bar{x}_2) \wedge (x_1 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$$

is not satisfiable.

The *CNF-Satisfiability* problem (CNF SAT) is the problem of determining whether a given CNF formula is satisfiable. CNF SAT is a fundamental problem in mathematical logic and computer science, with numerous applications.

The input size of a CNF formula can be defined as the total number of literals (counting repetitions) occurring in the formula. Clearly, CNF SAT \in NP, since a linear scan of the formula determines whether a candidate assignment to the literals occurring in the formula results in the formula evaluating to true. CNF SAT was the first example shown to be NP-complete.

Theorem 10.3.3 (Cook's Theorem)

CNF SAT is NP-complete. □

Given any (decision) problem A in NP, there is a nondeterministic algorithm that produces a candidate solution for A that can be checked in polynomial time. Cook proved Theorem 10.3.3 by constructing a CNF formula that, roughly speaking, modeled the action of the nondeterministic algorithm. The length of the formula constructed by Cook is polynomial in the size of an instance I of A , and is satisfiable if, and only if, the problem A is true for I .

10.4 Some Sample NP-Complete Problems

As soon as it was established that CNF SAT is NP-complete, the door was opened to show that a host of other NP-complete problems exist. Proposition 10.3.1 gives a recipe for proving that a problem A is NP-complete, which we state as a key fact.

Key Fact

To show that a problem A is NP complete,

1. show that A is in NP, and
2. find *any* NP-complete problem B such that $B \propto A$.

Condition 1 is usually easy to show, so that condition 2 becomes the issue. However, as we have already seen, there are cases where condition 1 is difficult to verify.

Thousands of classical and practical problems have been shown to be NP-complete. If you are working on a problem that you think might be NP-complete, you can check the vast list of NP-complete problems for one that might resemble your problem. The following subsections provide a small sample list of NP-complete problems. We have already shown that the clique problem and the sum of subsets problems are in NP. For each of the other examples, we will leave it as an exercise to show the problem is in NP. For each of the sample problems in this section, we will give a reduction that shows the problem is NP-complete.

Example 10.4.1: 3-CNF SAT

An instance of the 3-CNF SAT problem is a CNF Boolean formula in which every clause contains *exactly* three literals. Since general CNF SAT is in NP, 3-CNF SAT is also in NP. Surprisingly, 3-CNF SAT is NP-complete, even though 2-CNF SAT (each clause consists of two literals) is in P (see Exercise 10.9).

We show that CNF SAT \propto 3-CNF SAT by showing how any CNF formula $I = C_1 \wedge C_2 \wedge \dots \wedge C_m$ can be mapped onto a 3-CNF formula $f(I)$ with no more $12|I|$ literals, such that I is satisfiable if, and only if, $f(I)$ is satisfiable. More precisely, we construct $f(I)$ by replacing each clause $C_i = i = 1, \dots, m$, by conjunctions of three-literal clauses according to the following three cases

1. C_i has exactly three clauses,
2. C_i has more than three clauses,
3. C_i has less than three clauses.

In case 1, we leave C_i unaltered. In case 2, suppose C_i contains $k > 3$ literals x_1, x_2, \dots, x_k , so that $C_i = x_1 \vee x_2 \vee \dots \vee x_k$. We then replace C_i by the $k - 2$ clauses

$$(x_1 \vee x_2 \vee y_1) \wedge (\bar{y}_1 \vee x_3 \vee y_2) \wedge (\bar{y}_2 \vee x_4 \vee y_3) \wedge \dots \wedge (\bar{y}_{k-3} \vee x_{k-1} \vee x_k),$$

where y_1, y_2, \dots, y_{k-3} are new Boolean variables. We leave it as an exercise to show that C_i is satisfiable if, and only if, the conjunction of the $k - 2$ clauses replacing C_i is satisfiable.

Now consider case 3. If C_i consists of a single literal x , then we can replace x by $(x \vee y \vee z) \wedge (x \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee z) \wedge (x \vee \bar{y} \vee \bar{z})$, where y and z are new Boolean variables. If C_i consists of the two-literal clause $x_1 \vee x_2$, then we replace C_i by $(x_1 \vee x_2 \vee y) \wedge (x_1 \vee x_2 \vee \bar{y})$, where y is a new Boolean variable. In both cases, it follows easily that C_i is satisfiable if, and only if, the conjunction of the clauses replacing it are satisfiable.

Example 10.4.2: Clique

This example establishes a connection between Boolean formulas and graph theoretic problems, thereby extending the realm of NP-complete problems into the latter domain.

We show that CNF SAT \propto Clique. Consider any CNF formula $I = C_1 \wedge C_2 \wedge \dots \wedge C_m$.

Writing clause C_i as $y_1 \vee \dots \vee \underline{y_j} \vee \underline{y_{j+1}} \vee \dots \vee \underline{y_k}$ (the j and k depend i), where

$y_1, \dots, y_j \in \{x_1, \dots, x_n\}$ and $\underline{y_{j+1}}, \dots, \underline{y_k} \in \{\bar{x}_1, \dots, \bar{x}_n\}$, let

$V_i = \{(y_1, i), \dots, (y_j, i), (\bar{y}_{j+1}, i), \dots, (\bar{y}_k, i)\}$, $i = 1, \dots, m$. We map I to the graph $G = f(I)$ as follows. The vertex set V of G is the union of the V_i 's, $i = 1, \dots, m$. We define two vertices (a, i) and (b, j) to be adjacent if, and only if, $i \neq j$ and the literal a is not the negative of the literal b . For example, if $(x_1, 3), (x_5, 7), (\bar{x}_8, 2), (x_2, 4), (\bar{x}_3, 1), (\bar{x}_8, 1), (x_6, 2)$, and $(\bar{x}_6, 5)$ are vertices of G , then $\{(x_1, 3), (x_5, 7)\}$ and $\{(\bar{x}_8, 2), (x_2, 4)\}$ are edges of G but $\{(\bar{x}_3, 1), (\bar{x}_8, 1)\}$ and $\{(x_6, 2), (\bar{x}_6, 5)\}$ are not. Note that G can be constructed in polynomial time.

Claim

I is satisfiable if, and only if, G has a clique of size m .

Proof Suppose that G has a clique W of size m . By definition of adjacency in G , it follows that, for each clause C_i in I , there is exactly one vertex in W of the form (a,i) . If a is the positive literal x_q , then assign x_q the value T . If a is the negative literal \bar{x}_q , then assign \bar{x}_q the value T (that is, x_q is F). This assignment is well defined, since by definition of adjacency in G , none of the literals (in the second component) of a vertex is the negative of another. Moreover, this assignment makes each of the clauses C_i evaluate to T , so that I evaluates as true.

Now suppose that I is satisfiable. Let $Z = \{z_1, \dots, z_n\}$ denote the n literals from $\{x_1, \dots, x_n\} \cup \{\bar{x}_1, \dots, \bar{x}_n\}$ that have the value T . Since each clause C_i must have the value T , $i = 1, \dots, m$, C_i contains at least one literal z_i from Z . But, by the definition of G , (z_i, i) is a vertex of G . Further, the set of vertices $\{(z_i, i) : i \in \{1, \dots, m\}\}$ forms a clique of size m . ■

Example 10.4.3: Vertex Cover and Independent Set

That these problems are both NP-complete follows from Example 10.4.2 and Proposition 10.2.3.

Example 10.4.4: Three-Dimensional Matching

Recall that the perfect matching problem in a bipartite graph interprets the classical Marriage problem, which asks whether marriages can be arranged between a set of n boys and a set of n girls so that each boy marries a girl that he knows, and no girl marries more than one boy. The three-dimensional matching problem generalizes the Marriage problem by adding n houses to the mix. Not only do we seek to arrange marriages, we also wish to match each married couple with a house from a specified subcollection of the houses that are available to the couple, and so that no two couples share the same house. Adding houses turns a problem solved by an $O(n^3)$ algorithm into an NP-complete problem.

More formally, given three sets H , B , and G (houses, boys, and girls, respectively), each having the same cardinality $|H|$, and a subset $M \subseteq H \times B \times G$, the three-dimensional matching problem asks whether there is a subset (matching) $M' \subseteq M$ of cardinality $|M'| = |H|$, such that none of the triples in M' agree in any coordinate. We refer to the triples in M' as *households*. The three-dimensional matching problem is in NP, since a nondeterministic algorithm guesses a set of $|H|$ triples, and the verification that no two triples agree in any coordinate can be done in polynomial time. We now show that the three-dimensional matching problem is NP-complete by showing that CNF SAT reduces to it.

Suppose $I = C_1 \wedge C_2 \wedge \dots \wedge C_m$ is any instance of CNF SAT, and suppose I contains the Boolean variables x_1, x_2, \dots, x_n . We will construct an instance of the three-dimensional matching problem $H, B, G, M \subseteq H \times B \times G$, where $|H| = |B| = |G| = 2mn$, such that a matching $M' \subseteq M$ exists if, and only if, I is satisfiable. H consists of m copies of the literals $\{x_1, x_2, \dots, x_n\}, \{\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\}$. We use subscripts and superscripts to denote

the elements in H , where subscripts reference literals, and superscripts reference clauses in I ; that is

$$H = \{x_i^{(j)}, \bar{x}_i^{(j)} : 1 \leq i \leq n, 1 \leq j \leq m\}.$$

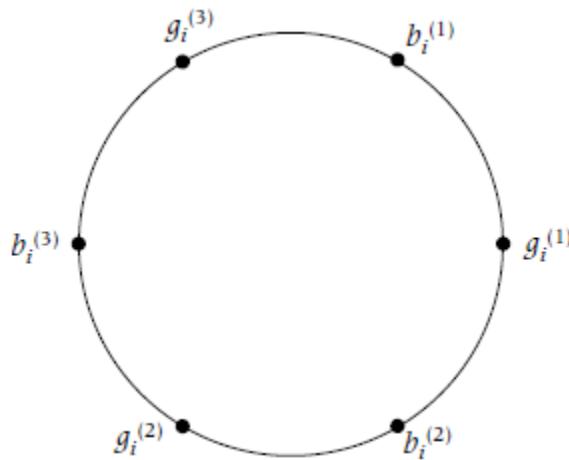
We now create sets B and G of $2mn$ boys and $2mn$ girls, respectively. The sets B and G will be divided into three classes, and marriages are restricted to take place only between boys and girls in the same class. Thus, the division of the boys and girls induces a division of M into three classes of households, which will be called “truth setting,” “satisfaction testing,” and “remaining” households, respectively. As the names imply, truth setting households force an assignment of truth values to the Boolean variables x_1, x_2, \dots, x_n , whereas satisfaction testing households determine whether this truth assignment makes the formula I true. The remaining households in M are made up of a set of additional couples created merely to occupy the remaining houses not covered by the households in the first two classes. We now describe the division of the boys and girls into three classes together with the resulting three classes of households.

Truth Setting Households

The truth setting households will be divided into n components $T_i, i = 1, \dots, n$, where each component holds m households. For each $i, 1 \leq i \leq n$, we create a set of m boys and m girls

$$B_{T_i} = \{b_i^{(1)}, b_i^{(2)}, \dots, b_i^{(m)}\}, G_{T_i} = \{g_i^{(1)}, g_i^{(2)}, \dots, g_i^{(m)}\}, i = 1, \dots, n.$$

Marriages are restricted so that a boy in the i th component B_{T_i} must marry a girl in the i th component G_{T_i} . The marriages are further restricted within each component T_i to take place between people who are adjacent when arranged in a circle, as in Figure 10.1.

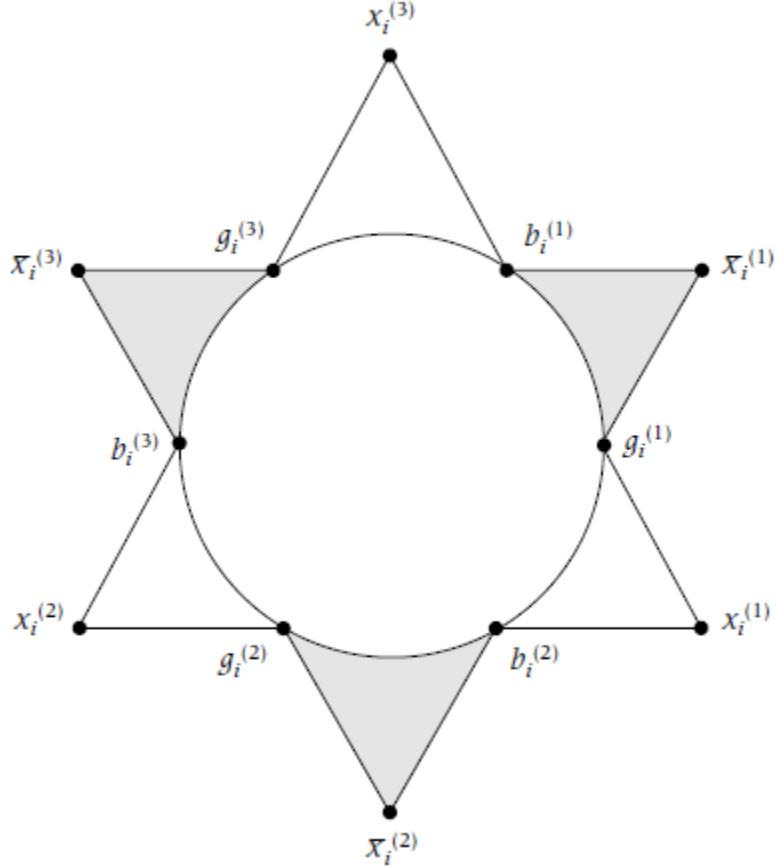


In each component, marriages must take place between adjacent people in the given circle arrangement, as illustrate for $m = 3$.

Figure 10.1

In particular, there are only two marriage schemes possible in T_i , a clockwise scheme, $\{b_i^{(1)}, g_i^{(1)}\}, \{b_i^{(2)}, g_i^{(2)}\}, \dots, \{b_i^{(m)}, g_i^{(m)}\}$, and a counterclockwise scheme, $\{b_i^{(1)}, g_i^{(m)}\}, \{b_i^{(m)}, g_i^{(m-1)}\}, \{b_i^{(m-1)}, g_i^{(m-2)}\}, \dots, \{b_i^{(2)}, g_i^{(1)}\}$.

In each component T_i , only one house will be made available to a given couple. If girl $g_i^{(j)}$ marries boy $b_i^{(j)}$, then she must live in house $\bar{x}_i^{(j)}$. Otherwise (girl $g_i^{(j)}$ marries boy $b_i^{(j+1)}$, where we set $b_i^{(m+1)} = b_i^{(1)}$), she must live in house $x_i^{(j)}$. The housing restrictions are illustrated in Figure 10.2, which helps explain the “truth setting” nomenclature. We see from Figure 10.2 that any matching M' must intersect the households in the i th component T_i in precisely one of two ways: Either $M' \cap T_i$ consists of the set $T_i(t)$ of shaded households, or $M' \cap T_i$ consists of the set $T_i(f)$ of unshaded households. Thus, the component T_i forces a matching M' to set a truth value for $x_i, i = 1, \dots, n$, where $M' \cap T_i = T_i(t)$ is considered as setting x_i to T , whereas $M' \cap T_i = T_i(f)$ is considered as setting x_i to F .



The household of T_i corresponding to $m = 3$. Any matching must intersect T_i precisely in either the shaded households $T_i(t)$ or the unshaded household $T_i(f)$.

Figure 10.2

In summary, the truth setting households in M consist of n components T_i , $i = 1, \dots, n$, where each T_i is the union of two sets of triples

$$T_i(t) = \{(\bar{x}_i^{(j)}, b_i^{(j)}, g_i^{(j)}): 1 \leq i \leq n, 1 \leq j \leq m\},$$

$$T_i(f) = \{(x_i^{(j)}, b_i^{(j+1)}, g_i^{(j)}): 1 \leq i \leq n, 1 \leq j < m\} \cup (x_i^{(j)}, b_i^{(1)}, g_i^{(m)}).$$

Moreover, no household in M other than T_i contains a boy $b_i^{(j)}$ or a girl $g_i^{(j)}$, $1 \leq i \leq n$, $1 \leq j \leq m$.

Satisfaction Testing Households

We now create m boys $s_b^{(j)}$ and m girls $s_g^{(j)}$, respectively, $j = 1, \dots, m$, and match them by prearranged marriages $\{s_b^{(j)}, s_g^{(j)}\}: j = 1, \dots, m\}$. Moreover, the only houses available to couple $\{s_b^{(j)}, s_g^{(j)}\}$ are literals (x_i or \bar{x}_i) that appear in clause C_j . Thus, the satisfaction testing households in M consist of the triples:

$$S_j = \{(x_i^{(j)}, s_b^{(j)}, s_g^{(j)}): x_i \in C_j, i = 1, \dots, n\} \cup \{(\bar{x}_i^{(j)}, s_b^{(j)}, s_g^{(j)}): x_i \in C_j, i = 1, \dots, n\}.$$

No other households in M contain a boy $s_b^{(j)}$ or a girl $s_g^{(j)}$, $j = 1, \dots, m$. Hence, any matching $M' \subseteq M$ must contain exactly one household from each S_j , $j = 1, \dots, m$. Moreover, if the household chosen in S_j uses a house corresponding to the variable x_i , then that house must not be included in $M' \cap T_i$. In other words, we must choose a house whose truth setting for x_i as determined by M' satisfies clause C_j . Thus, a matching $M' \subseteq M$ cannot exist unless I is satisfiable.

Remaining Households

Any matching M' uses nm houses to accommodate the truth setting households and m houses to accommodate the satisfaction testing households. Thus, there are $m(n - 1)$ houses that are not covered by the truth setting or satisfaction testing households making up a *partial matching* M' . We create a set of $m(n - 1)$ boys $r_b^{(j)}$ and $m(n - 1)$ girls $r_g^{(j)}$, respectively, $j = 1, \dots, m(n - 1)$, so that they can occupy the remaining $m(n - 1)$ houses not covered by such a partial matching M' , and thereby extend M' to a matching. These boys and girls are matched by prearranged marriages consisting of couples $\{r_b^{(j)}, r_g^{(j)}\}: j = 1, \dots, m(n - 1)\}$. While these couples had no choice in their marriages, the set M places no housing restrictions on them. Thus, the remaining households R in M consist of the triples

$$R = \{(x_i^{(j)}, r_b^{(k)}, r_g^{(k)}): i = 1, \dots, n, j = 1, \dots, m, k = 1, \dots, m(n - 1)\}$$

$$\cup \{(\bar{x}_i^{(j)}, r_b^{(k)}, r_g^{(k)}): i = 1, \dots, n, j = 1, \dots, m, k = 1, \dots, m(n - 1)\}.$$

Given the instance I of CNF SAT, the set $H \times B \times G$ and the subset of households $M \subseteq H \times B \times G$ can clearly be constructed in polynomial time. We have already noted that a matching $M' \subseteq M$ cannot exist unless I is satisfiable. We complete the proof that CNF SAT \propto three-dimensional matching by showing that if I is satisfiable, then a matching $M' \subseteq M$ exists. Consider a truth assignment to the variables x_1, x_2, \dots, x_n that makes I true.

For each clause C_j , we choose a literal I_j occurring in C_j having the value T . Such a choice must be possible since I is assumed true with the given truth assignment. We then set

$$M' = \left(\bigcup_{x_i=T} T_i(t) \right) \cup \left(\bigcup_{\bar{x}_i=T} T_i(t) \right) \cup \left(\bigcup_{j=1}^m \{(l_j, s_b^{(j)}, s_g^{(j)})\} \right) \cup R',$$

where R' is a suitably constructed collection of remaining households that includes all the couples $\{r_b^{(k)}, r_g^{(k)}\}$, $k = 1, \dots, m(n-1)$ and all the remaining houses that were not covered by the truth setting or satisfaction testing households in M' . We leave it as an exercise to verify that R' can be constructed and that the resulting set M' is a matching.

Example 10.4.5: 3-Exact Cover

Given a family $F = \{S_1, S_2, \dots, S_n\}$ of n subsets of $S = \{x_1, x_2, \dots, x_{3m}\}$, each of cardinality three, the *3-exact cover* problem asks if there is a subfamily $\{S_{i_1}, S_{i_2}, \dots, S_{i_m}\}$ of m subsets of F that covers S , that is, $S = \bigcup_{j=1}^m S_{i_j}$. The 3-exact cover problem is in NP, since a nondeterministic algorithm guesses a subfamily, and the verification that the subfamily covers S can be done in linear time. (We can use characteristic bit vectors of length $3m$ to represent subsets of F .)

Note that three-dimensional matching \propto 3-exact cover. In fact, three-dimensional matching is actually just a special case of 3-exact cover where we ignore the ordering of the subsets. More precisely, we simply associate the instance $M \subseteq H \times B \times G$ of the three-dimensional matching problem with the instance $S = H \cup B \cup G$ and $F = \{\{h, b, g\} : (h, b, g) \in M\}$ of the 3-exact cover problem.

Example 10.4.6: Sum of Subsets

We now show that 3-exact cover \propto sum of subsets. Given an instance $F = \{S_1, S_2, \dots, S_n\}$, $S = \{x_1, x_2, \dots, x_{3m}\}$ of 3-exact cover, as noted previously we can consider the sets in F as represented by their characteristic bit vectors of length $3m$. For example, if $m = 3$ and $S_1 = \{x_2, x_3, x_8\}$, then S_1 is represented by $(0, 1, 1, 0, 0, 0, 1, 0)$. We map an instance of 3-exact cover to an instance of sum of subsets by interpreting each characteristic vector as an integer in the base- $(n+1)$ system. Thus, for each $S_j \in F$, we associate with S_j the integer $a_{j-1} = \sum_{x_i \in S_j} (n+1)^{i-1}$. We let C be the integer corresponding to the bit vector of all ones; that is, $C = \sum_{i=0}^{3m-1} (n+1)^i$. We leave it as an exercise to show that F, S is a yes instance of 3-exact cover if, and only if, a_0, a_1, \dots, a_{n-1} and target sum C is a yes instance of sum of subsets.

Example 10.4.7: Graph Coloring

We show that 3-CNF SAT \propto graph coloring. Suppose $I = C_1 \wedge C_2 \wedge \dots \wedge C_m$ is an instance of 3-CNF SAT involving the Boolean variables $\{x_1, x_2, \dots, x_n\}$. We may assume that $n \geq 4$, since otherwise the satisfiability of I can be checked in polynomial time. We construct an instance $G = (V, E)$, $k = n + 1$, of graph coloring as follows

$$V = \{x_1, x_2, \dots, x_n\} \cup \{\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\} \cup \{y_1, y_2, \dots, y_n\} \cup \{c_1, c_2, \dots, c_m\},$$

$$\begin{aligned}
E = & \{x_i \bar{x}_i : 1 \leq i \leq n\} \cup \{y_i y_j : 1 \leq i < j \leq n\} \cup \{x_i y_j : 1 \leq i \neq j \leq n\} \\
& \cup \{\bar{x}_i y_j : 1 \leq i \neq j \leq n\} \cup \{x_i c_j : x_i \notin C_j, \quad 1 \leq i \leq n, 1 \leq j \leq m\} \\
& \cup \{\bar{x}_i c_j : \bar{x}_i \notin C_j, \quad 1 \leq i \leq n, \quad 1 \leq j \leq m\}.
\end{aligned}$$

Clearly, G can be constructed from I in polynomial time. We leave it as an exercise to show that I is satisfiable if, and only if, the graph G is $(n + 1)$ -colorable.

10.5 The Class co-NP

Given a decision problem Π , the complementary problem $\bar{\Pi}$ is the decision problem with the same instances, but where an instance I of the problem Π is true if, and only if, I is false for $\bar{\Pi}$. For example, suppose $\bar{\Pi}$ is the decision problem “Does a graph G contain a Hamiltonian cycle?” The complementary decision problem $\bar{\Pi}$ asks: “Is there *no* Hamiltonian cycle in G ?”. A certificate for a Hamiltonian cycle is easy to verify, but what would a certificate be for the complementary problem? One would be an enumeration of all possible Hamiltonian cycles, but there are far too many candidates to check in polynomial time. In fact, it is believed that the non-Hamiltonian problem does not belong to NP.

Clearly, the complement $\bar{\Pi}$ of a problem in P also belongs to P. Indeed, a polynomial algorithm for Π also serves as a polynomial algorithm for $\bar{\Pi}$ by outputting *yes* for an instance I of $\bar{\Pi}$ if, and only if, the algorithm outputs *no* for instance I of problem Π . However, the same argument does not work for problems in NP. Given a problem $\Pi \in$ NP, a *yes* instance for Π corresponds to a *no* instance for $\bar{\Pi}$, and we can't be assured of the existence of certificates that can be verified in polynomial time for *no* instances of problems in NP.

The class *co-NP* is the class of problems that are complements of problems in NP. As an example, consider again the problem of prime testing, which asks whether a given integer n is prime. Prime testing is in co-NP, since its complement belongs to NP. A nondeterministic algorithm merely guesses a factor, and it can be checked in polynomial time whether the factor divides n evenly. Recall that it has recently been shown that prime testing is actually in P.

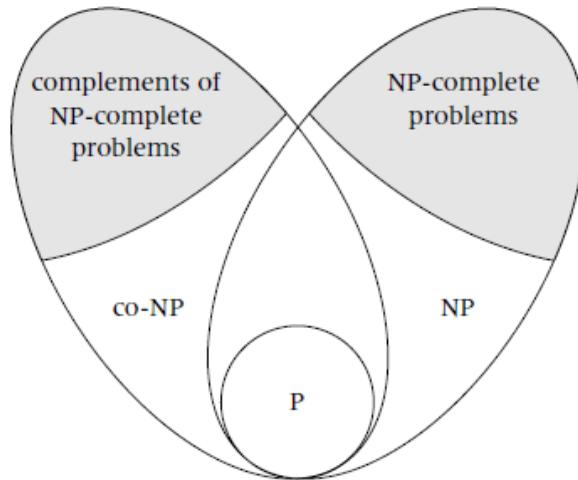
A natural question is whether $\text{co-NP} \neq \text{NP}$. Most computer scientists believe the answer is yes and conjecture that any NP-complete problem provides the answer. The following theorem provides the motivation for this belief.

Theorem 10.5.1

Let Π be any NP-complete problem. If the complement $\bar{\Pi}$ belongs to NP, then $\text{NP} = \text{co-NP}$. □

We leave the proof of Theorem 10.5.1 as an exercise.

The Venn diagram in Figure 10.3 shows the conjectured relationships between P, NP, and co-NP. However, there still is the (very unlikely) possibility that $P = NP$, in which case the whole diagram would collapse to P.



Conjecture relationships between P, NP, and Co-NP

Figure 10.3

10.6 Closing Remarks

Cobham introduced the class P in 1964. Edmonds, who independently proposed the class P in 1965, also introduced the class NP and conjectured that $P \neq NP$. Cook, who introduced the notion of NP -completeness in 1971, showed that CNF SAT and 3-CNF SAT are NP -complete. Shortly thereafter, Levin independently discovered the notion and also proved the existence of NP -complete problems. Karp popularized the notion of NP -completeness by establishing that a wide variety of important and practical problems are NP -complete.

Our discussions of the notions of NP and NP -completeness were somewhat informal. A more formal treatment requires the establishment of a formal model of computation such as a Turing Machine.

Exercises

Sections 10.1 and 10.2 The Classes P and NP and Reducibility

- 10.1 Suppose that the weights and values of the objects in the 0/1 Knapsack problem are all integers. Show that 0/1 Knapsack problem and its associated decision version are polynomially equivalent. (*Hint:* Use a binary search technique.)
- 10.2 Given an integer n , it was shown by Pratt that n is prime if, and only if, there exists an integer a such that
 - a) $a^{n-1} \equiv 1 \pmod{n}$, and

b) $a^{(n-1)/p} \not\equiv 1 \pmod{n}$ for all prime divisors p of $n - 1$.

Based on this result, show that the prime testing problem is in NP.

- 10.3 Prove a result for TSP analogous to that of Exercise 10.1.
- 10.4 Prove Proposition 10.2.1.
- 10.5 Prove Proposition 10.2.2.
- 10.6 Prove Proposition 10.2.3.

Sections 10.3 and 10.4 NP-Complete Problems

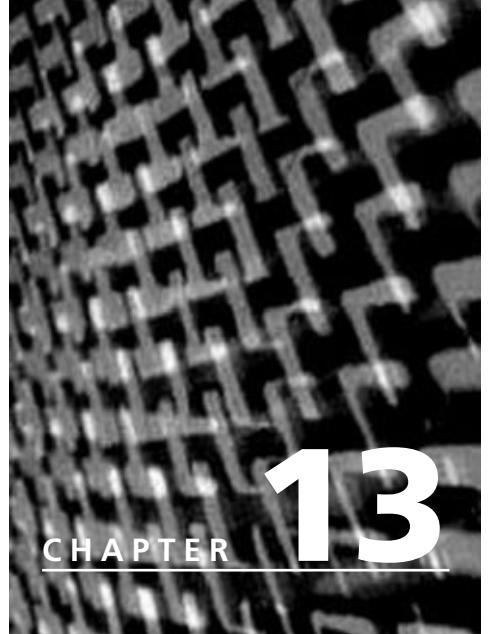
- 10.7 Verify that each of the NP-complete problems in Section 10.4 belong to NP.
- 10.8 Prove Proposition 10.3.1
- 10.9 Show that 2-CNF SAT is in P.
- 10.10 This exercise refers to the mapping f from CNF SAT to 3-CNF SAT described in Example 10.4.1.
 - a. Show that the input size of $f(I)$ is at most 12 times the input size of I for any instance I of CNF SAT.
 - b. Show in cases 2 and 3 that each clause C_i of I is satisfiable if, and only if, the conjunction of the clauses replacing C_i is satisfiable.
- 10.11 In Example 10.4.4, verify that the set R' of remaining households can be constructed and that the resulting set M' is a matching.
- 10.12 Show that the instance F, S of the 3-exact cover problem is a yes instance if, and only if, the instance s_1, s_2, \dots, s_n and C constructed in Example 10.4.6 is a yes instance of the sum of subsets problem.
- 10.13 Show that the problem of determining whether a graph G is bipartite (has a proper 2-coloring) is in P.
- 10.14 In Example 10.4.7, show that the graph G constructed from the instance I of 3-CNF SAT is $(n + 1)$ -colorable if, and only if, I is satisfiable.
- 10.15 Given a multiset $S = \{s_1, s_2, \dots, s_n\}$ of positive integers, the *partition problem* asks whether S can be partitioned into two subsets having the same sum. Show that the partition problem is NP-complete. (*Hint:* Show that sum of subsets \propto partition.)
- 10.16 Show that the 0/1 Knapsack problem is NP-complete. (*Hint:* Show that partition \propto 0/1 Knapsack [partition is defined in previous exercise].)
- 10.17 Suppose we have a set of n objects $\{x_1, x_2, \dots, x_n\}$ of sizes $\{s_1, s_2, \dots, s_n\}$, where $1 < s_i < 1$, $i = 1, \dots, n$. The *bin packing* optimization problem is to place the objects into bins having unit size using the minimum number of bins. Each bin can contain any subset of objects whose total size does not exceed one. The decision version asks for a given integer k whether we can pack the objects using no more than k bins. Show that the bin problem is NP-complete. (*Hint:* Show that sum of subsets \propto bin packing.)

Section 10.5 The class co-NP

- 10.18 Prove Theorem 10.5.1.

Section 10.6 The Classes NC and P-Complete

- 10.19 Prove Proposition 10.6.1 (DeMorgan's Laws).
- 10.20 Use induction to prove the following generalization of DeMorgan's Laws. For $x_1, x_2, \dots, x_k \in \{T, F\}$,
- $\neg(x_1 \wedge x_2 \wedge \dots \wedge x_k) = \neg x_1 \vee \neg x_2 \vee \dots \vee \neg x_k$,
 - $\neg(x_1 \vee x_2 \vee \dots \vee x_k) = \neg x_1 \wedge \neg x_2 \wedge \dots \wedge \neg x_k$.
- 10.21 Let f be the transformation that maps an instance I of the SLC problem to the instance $f(I)$ of the SLCWD problem, where $f(I)$ is constructed from I by concurrently replacing each occurrence of $x \vee y$, where x and y are Boolean expressions, with $\neg(\neg x \wedge \neg y)$. Design an NC algorithm for computing $f(I)$ from I .
- 10.22
 - Design an NC algorithm for performing the replacement operations given in (10.6.2) to obtain $g(I)$ from an instance I of the SLCWD problem.
 - Design an NC algorithm for converting $g(I)$ to an instance I of the NOR SLC problem.
- 10.23
 - Give pseudocode for the (sequential) procedure *SequentialLFMIS* for computing the lexicographically first maximal independent set of a graph G .
 - Analyze the complexity of procedure *SequentialLFMIS*.
- 10.24
 - Design an NC algorithm for performing the reduction from an instance I of the NOR SLC problem to the instance $G = f(I)$ of the LFMIS problem shown in Figure 10.6.
 - Show that x_i has the value T in instance I of the NOR SLC problem if, and only if, vertex i is in the lexicographically maximal independent set in $G = f(I)$.



13

CHAPTER

GRAPH CONNECTIVITY AND FAULT-TOLERANCE OF NETWORKS

In many applications modeled using graphs and digraphs, it is important to determine whether certain connectivity properties hold. For example, in a communication or computer network, it is important to find an alternate path joining two nodes in the presence of node failures. In this chapter we present algorithms for some classical graph connectivity problems, such as finding the strongly connected components of a digraph, finding articulation points of a graph, and computing the open ear decomposition of a biconnected graph. We design $O(m + n)$ algorithms for these problems based on depth-first search numbering techniques introduced by Tarjan. We also discuss fault-tolerant routing schemes based on open ear decompositions and s - t orderings.

13.1 Strongly Connected Components

We can easily solve the problem of finding the components of an undirected graph G using depth-first traversal. The vertex set of each tree of a depth-first traversal forest is a component of G . Thus, we can compute the components of G in time $O(m + n)$ using *DFTForest* (see Chapter 11), where n and m denote the number of vertices and edges of G , respectively.

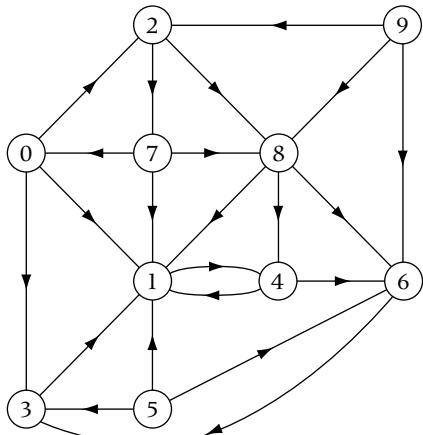
The notion of a strongly connected component in a digraph $D = (V, E)$ is a generalization of the notion of a component in a graph. Two vertices u and v in a digraph D are *strongly connected* if there is a directed path from u to v and a directed path from v to u . Clearly, the relation *strongly connected* is an equivalence relation on the set $V = V(D)$ of vertices of D . The subdigraphs induced by these equivalence classes are called *strongly connected components of D* (see Figure 13.1).

REMARK

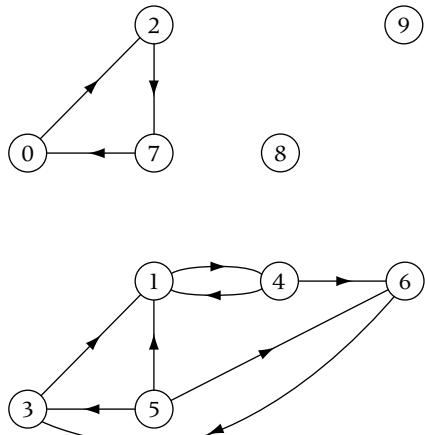
Although the connected components of a graph partition the vertex set as well as the edge set, in general, the strongly connected components of a digraph only partition the vertex set. An edge can be in at most one strongly connected component, but it might not be in any.

FIGURE 13.1

Strongly connected components in a digraph D .



Digraph D



Strongly connected components

The depth-first search out-tree $T_{\text{out}, v}$ of digraph D rooted at v is an out-tree that contains a path from v to every vertex u for which there exists a path in D from v to u . The depth-first search in-tree $T_{\text{in}, v}$ of D rooted at v contains a path from u to v for every vertex u for which there exists a path in D from u to v . Thus, a vertex u belongs to the strongly connected component containing v if and only if u belongs to both $T_{\text{out}, v}$ and $T_{\text{in}, v}$. Computing $T_{\text{out}, v}$ and $T_{\text{in}, v}$ using DFSOutTree and DFSInTree , and then determining the vertices common to both, yields an $O(m + n)$ algorithm for finding the strongly connected component containing vertex v . Calling this algorithm for each vertex v yields an $O(n(m + n))$ algorithm for obtaining all the strongly connected components of D .

We now discuss a more efficient ($O(m + n)$) algorithm *StrongComponents* for obtaining the set of strongly connected components of a digraph, which is based on the notion of the postnumbering of the vertices of a digraph. For convenience, we assume that $V(D) = \{0, 1, \dots, n - 1\}$. The postnumbering of a digraph D is determined by performing an out-directed depth-first traversal of D . For definiteness in our illustrations, we assume that the depth-first traversal always chooses the smallest vertex whenever there is a choice. Recall that a vertex u of D becomes *explored* when the traversal accesses u , having visited all vertices in the out-neighborhood of u . The *postnumber* of u , denoted $\text{PostNum}(u)$, is the integer i , where u is the $(i + 1)^{\text{st}}$ vertex to be explored in an out-directed depth-first traversal of D , $i = 0, \dots, n - 1$ (see Figure 13.2a).

REMARK

Vertex u has postnumber i if u is the $(i + 1)^{\text{st}}$ vertex to be visited in a postorder traversal of the DFT out-forest, $i = 0, \dots, n - 1$.

Clearly, PostNum is a permutation of $0, 1, \dots, n - 1$. The inverse permutation of PostNum , which we denote by PostNumInv , can be computed in linear time. Note that $\text{PostNumInv}(0), \text{PostNumInv}(1), \dots, \text{PostNumInv}(n - 1)$ lists the vertices in the order in which they have been explored. We can easily modify the depth-first traversal DFTOut to compute arrays $\text{PostNum}[0:n - 1]$ and $\text{PostNumInv}[0:n - 1]$, representing the permutations PostNum and PostNumInv , respectively. In Figure 13.2, we illustrate the values of the arrays $\text{PostNum}[0:n - 1]$ and $\text{PostNumInv}[0:n - 1]$ for a given digraph D .

The following two propositions are the basis for the algorithm *StrongComponents* for computing the strongly connected components of a digraph.

Proposition 13.1.1 Let C be the vertex set of a strongly connected component of a digraph D . Suppose u is a vertex such that $\text{PostNum}(u) < \text{PostNum}(r)$ for some vertex $r \in C$. If u is joined with an edge uv to some vertex v of C (not necessarily equal to r), then u belongs to C .

PROOF

We give a proof by contradiction. Assume that the hypothesis of the proposition holds, but u does not belong to C . It follows that there is no path from any vertex in C to u . Clearly, u does not become explored by the out-directed depth-first traversal until after v is accessed. Let x be the first vertex in C to be accessed by the out-directed depth-first traversal. Since C is the vertex set of a strongly connected component, it follows that every vertex of C different from x will be accessed and eventually explored before x becomes explored. Because there is no path from x to u , after having accessed x , the out-directed depth-first search will not access u until after x has been explored. Thus, every vertex in C is explored before u , so that every vertex in C has a smaller post-number than u . However, this contradicts the fact that r has a greater post-number than u . ■

Proposition 13.1.2

Given a digraph D , suppose we perform an in-directed depth-first traversal in which we scan the vertices in the order $\text{PostNumInv}[n - 1], \text{PostNumInv}[n - 2], \dots, \text{PostNumInv}[0]$. Then the vertex sets of the in-trees T_1, T_2, \dots, T_k of the associated depth-first traversal in-forest coincide with the vertex sets that induce the strongly connected components of D . More precisely, given the tree T_i , let r_i denote the root vertex of T_i and let V_i denote the vertex set of T_i . Then the vertex set C of the strongly connected component containing r_i is precisely $V_i, i = 1, \dots, k$.

PROOF

By proving that $C \supseteq V_i$, we show inductively that the vertices of V_i having depth d in T_i belong to C , $d = 0, \dots, \text{depth}(T_i)$.

Basis step: Since r_i belongs to C , this result is true for $d = 0$.

Induction step: Assume that all vertices of T_i having depth d belong to C and consider a vertex u of T_i having depth $d + 1$. Because we choose r_i to be the vertex not in trees T_1, \dots, T_{i-1} having the largest postnumber, it follows that $\text{PostNum}(u) < \text{PostNum}(r_i)$. Further, since u has depth $d + 1$ in T_i , it is joined to some vertex v of T_i at depth d . By induction hypothesis, v belongs to C , so that Proposition 13.1.1 implies that u belongs to C . Thus, by induction, all the vertices of T_i belong to C . ■

The following algorithm, *StrongComponents*, finds the strongly connected components D in two stages. In the first stage, *StrongComponents* performs an out-directed depth-first traversal to compute the array $PostNumInv[0:n - 1]$. In the second stage, *StrongComponents* performs an in-directed depth-first traversal, where we scan in the order $PostNumInv[n - 1], \dots, PostNumInv[0]$. In the pseudocode for *StrongComponents*, we use the array $InForest[0:n - 1]$ to keep track of the depth-first traversal in-forest (see Figure 13.2). *StrongComponents* has the same order of complexity, $O(m + n)$, as the depth-first traversal.



```

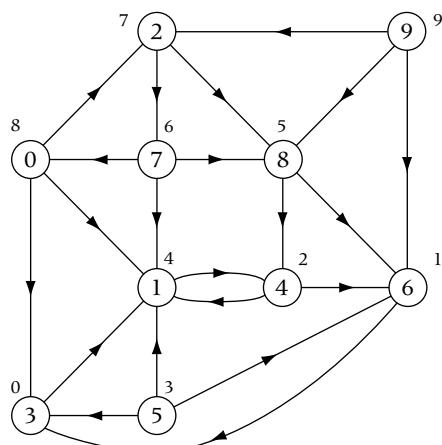
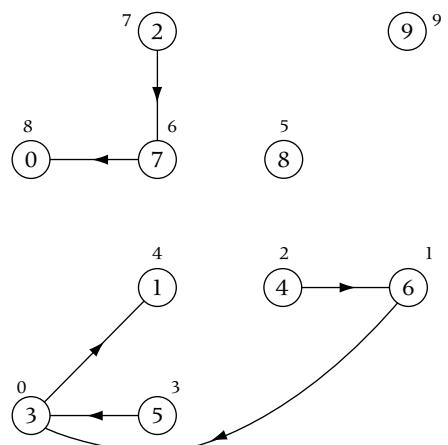
procedure StrongComponents( $D, PostNumInv[0:n - 1]$ )
Input:  $D$  (a digraph with  $n$  vertices and  $m$  edges)
         $PostNumInv[0:n - 1]$  ( $PostNumInv[i]$  (the vertex  $u$  where  $PostNum[u] = i$ )
Output:  $InForest[0:n - 1]$  (an array giving parent representation of the forest of in-
        trees  $T_1, T_2, \dots, T_k$ )
         $Mark[0:n - 1]$  //a 0–1 array
         $InForest[0:n - 1]$ 
for  $v \leftarrow 0$  to  $n - 1$  do
         $Mark[v] \leftarrow 0$ 
         $InForest[v] \leftarrow -1$ 
endfor
for  $i \leftarrow n - 1$  downto 0 do      //perform an in-directed depth-first traversal
         $v \leftarrow PostNumInv[i]$            //according to reverse order of postnumbers
        if  $Mark[v] = 0$  then
            DFSInTree( $D, v, InForest$ ) //perform an in-directed depth-first search rooted
                                //at vertex  $v$  and store associated depth-first in-tree
                                //as part of  $InForest[0:n - 1]$ 
        endif
endfor
end StrongComponents

```

The action of *StrongComponents* is illustrated in Figure 13.2.

FIGURE 13.2

Arrays
 $\text{PostNum}[0:9]$,
 $\text{PostNumInv}[0:9]$,
and
 $\text{InForest}[0:9]$ for
the digraph D given
in Figure 13.1.

Digraph D with the post number shown outside each nodeIn-forest generated by
StrongComponents

i	0	1	2	3	4	5	6	7	8	9
$\text{PostNum}[i]$	8	4	7	0	2	3	1	6	5	9
$\text{PostNumInv}[i]$	3	6	4	5	1	8	7	2	0	9
$\text{InForest}[i]$	-1	-1	7	1	6	3	3	0	-1	-1

13.2 Articulation Points and Biconnected Components

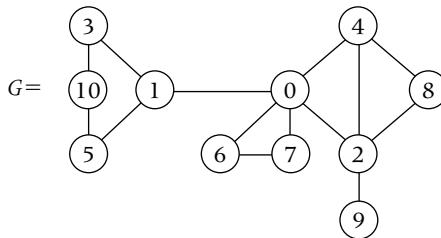
In many applications of graphs, it is useful to know that stronger connectivity properties hold than merely being connected. For example, suppose G models a *communication network*, where the vertices of G might correspond to people, processors, computers, and so forth. Two vertices u and v in such a network are joined with an edge whenever direct communication is possible between them. A pair of vertices u and v that cannot communicate directly can communicate by relaying the information along a u - v path—that is, a path joining u and v . In practice, having more than one path connecting vertices u and v makes the communication network more tolerant of certain failures. For example, sometimes a vertex in the network becomes “inoperable,” meaning it can no longer relay information. This situation calls for finding communication paths that bypass the inoperable vertex.

For some vertices in a graph, finding bypass paths might not be possible in all cases. For example, if deleting a certain vertex v disconnects the graph, then vertices u, w in two different components of $G - v$ can no longer communicate with one another because all paths connecting u and w in G must pass through v . In this section, we consider algorithms for identifying such troublesome vertices v , called *articulation points*.

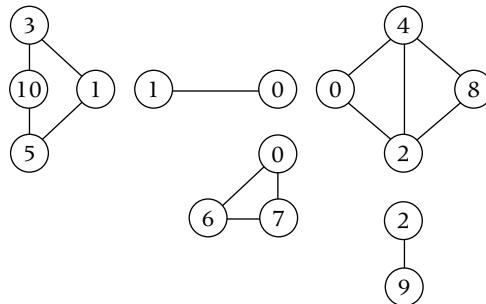
Throughout this section, we assume that the graph G is connected. A vertex v of a connected graph G is an *articulation point* (also called a *cut vertex*) if the graph $G - v$ obtained from G by deleting vertex v and all incident edges is not connected. A *biconnected component* of G (also called a *block* of G) is a maximal subgraph B such that B has no articulation points (see Figure 13.3).

FIGURE 13.3

Articulation points and biconnected components of a connected graph G .



Graph G with articulation points $\{0, 1, 2\}$



Biconnected components of G

There are many equivalent conditions for a graph to be biconnected. Six of these conditions are given in the following theorem.

Theorem 13.2.1 Let G be a connected graph with at least three vertices. Then the following statements are equivalent:

1. G is biconnected.
2. Every two vertices u and v belong to a common cycle. Equivalently, there exist two internal disjoint $u-v$ paths.
3. Every vertex and edge belongs to a common cycle.
4. Every two edges belong to a common cycle.
5. Given two vertices u, v and an edge e , there is a $u-v$ path containing e .
6. Given three distinct vertices u, v, w of G , there exists a $u-v$ path containing w .
7. Given three distinct vertices u, v, w of G , there exists a $u-v$ path not containing w . \square

The proof of Theorem 13.2.1 is left to the exercises. Since a biconnected component is a maximal subgraph that is biconnected, the subgraph $G[U]$ induced by a set of vertices U is a biconnected component if and only if U is a maximal set such that $G[U]$ satisfies the conditions of Theorem 13.2.1.

We now consider the problem of finding the articulation points and the biconnected components of a connected graph G . First we design an algorithm for computing the articulation points, which we then use to compute the biconnected components. The algorithms we discuss for finding the articulation points of G are based on the DFS-tree T_v rooted at a vertex $v \in V(G)$. The following two easily verified propositions are useful in finding articulation points.

Proposition 13.2.2 Given a vertex r of a connected graph $G = (V, E)$, consider the DFS-tree T_r rooted at vertex r . Then for any edge $e \in E$, one end point of e is an ancestor of the other in T_r . More precisely, if $vw \in E$, and v is accessed before w in the depth-first search rooted at r , then v is an ancestor of w in T_r . \square

Proposition 13.2.3 Given a vertex r of a connected graph $G = (V, E)$, consider the DFS-tree T_r rooted at vertex r . Then r is an articulation point in G if and only if r has two or more children in T_r . \square

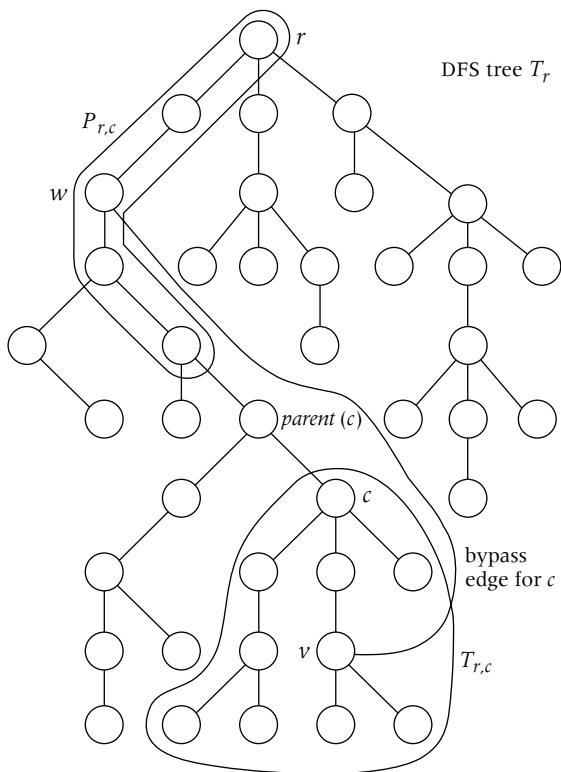
Using Proposition 13.2.3, computing T_r for each vertex r in the graph G yields an $O(nm)$ algorithm for determining the set of all articulation points.

We now discuss an $O(m)$ algorithm *ArticulationPoints* for computing the articulation points. As with the algorithm *StrongComponents*, *ArticulationPoints* is based on computing (two) numberings of the vertices based on a single depth-first search. The first numbering, *DFSNum*, is simply the order in which the vertices are (first) accessed by a depth-first search rooted at vertex r . The second numbering, *Lowest*, can be computed recursively at the same time as the depth-first search tree T_r rooted at vertex r (and associated numbering *DFSNum*) is determined. *Lowest* is defined in terms of the notion of a bypass edge.

Given a vertex u in T_r different from r , we show that u is an articulation point if and only if u has a child c in T_r that has no bypass edge defined as follows. Let $T_{r,c}$ denote the subtree of T_r having root vertex c , and let $P_{r,c}$ denote the path in T_r joining r to $\text{parent}(\text{parent}(c))$. A *bypass edge for c* is an edge $vw \in E$ such $v \in T_{r,c}$ and $w \in P_{r,c}$ (see Figure 13.4).

FIGURE 13.4

DFS-tree T_r , vertex c , subtree $T_{r,c}$, path $P_{r,c}$, and bypass edge vw for c .



The following proposition and its corollary are easily verified.

Proposition 13.2.4 Given a vertex r of a connected graph $G = (V, E)$, consider the DFS-tree T_r rooted at vertex r . Then for any vertex c at depth at least two in T_r , there exists a path in G not containing $\text{parent}(c)$ that joins c and r if and only if there exists a bypass edge for c . \square

Corollary 13.2.5 Given a vertex r of a connected graph $G = (V, E)$, consider the DFS-tree T_r rooted at vertex r . Then a vertex $v \neq r$ is an articulation point in G if and only if there exists a child c of v in T_r having no bypass edge for c . \square

An algorithm for finding the articulation points in G is based on Corollary 13.2.5 and the fact that the numbering Lowest gives a simple test for the existence of bypass edges. We define $\text{Lowest}[c]$ to be the minimum DFS number of c and all vertices w for which there is a nontree edge vw , where v is either c or a descendant of c in T_r and w is an ancestor of c in T_r . Clearly, c has a bypass edge if and only if $\text{Lowest}[c] < \text{DFSNumb}(\text{parent}(c))$. Therefore, by Corollary 13.2.5, we have the following proposition.

Proposition 13.2.6 Let u be any vertex of G different from the root of T_r . Then, u is an articulation point if and only if u has at least one child c such that

$$\text{Lowest}[c] \geq \text{DFSNumb}(u). \quad (13.2.1) \quad \square$$

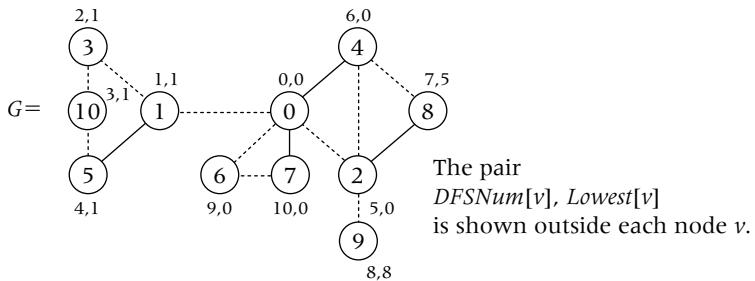
For a given $v \in V(G)$, it is easily verified that $\text{Lowest}[v]$ satisfies the following recurrence relation (see Exercise 13.11).

$$\text{Lowest}[v] = \min \begin{cases} \min \{\text{Lowest}[c] \mid c \text{ is a child of } v \text{ in } T_r\}, \\ \min \{\text{DFSNumb}[w] \mid vw \in E, w \neq \text{parent}(v)\}, \\ \text{DFSNumb}[v]. \end{cases} \quad (13.2.2)$$

- REMARKS**
1. We assume that $\min\{\text{DFSNumb}[w] : vw \in E, w \neq \text{parent}(v)\} = \infty$ if no such w exists. Similarly, $\min\{\text{Lowest}[c] \mid c \text{ is a child of } v \text{ in } T_r\} = \infty$ if v is a leaf node of T_r .
 2. We can compute Lowest using a depth-first search. When computing $\text{Lowest}[v]$ during such a search, if $vw \in E$, and $w \neq \text{parent}(v)$, then by Proposition 13.2.2, either w is an ancestor of v in T_r , so that $\text{DFSNumb}[w]$ is defined, or $w \in T_{r,v}$ and $\text{DFSNumb}[w]$ will be defined during the recursive resolution of $\text{Lowest}[c]$ for an appropriate child c of v . This makes $\min\{\text{DFSNumb}[w] \mid vw \in E, w \neq \text{parent}(v)\}$ well defined during our depth-first search.

FIGURE 13.5

Values of $DFSNumb$
and $Lowest$ for G
and the array
 $TreeDFS[0:10]$ for
 T_0 .



i	0	1	2	3	4	5	6	7	8	9	10
$TreeDFS[i]$	-1	0	0	1	2	10	0	6	4	2	3

Figure 13.5 illustrates the values of $DFSNumb[v]$ and $Lowest[v]$ for the same graph as shown in Figure 13.3, where we use root vertex $r = 0$. The pair $DFSNumb[v], Lowest[v]$ is shown outside each node v , and the edges of the depth-first search tree T rooted at vertex 0 are shown as dotted.

We can compute the $Lowest$ and $DFSNumb$ s numberings by performing what amounts to a depth-first search as given by the following $O(m)$ recursive algorithm:

→

```

procedure DFSNumberings( $G, v$ ) recursive
Input:  $G$  (a connected graph with  $n$  vertices and  $m$  edges)
 $v$  (a vertex of  $G$ ) // $v$  is root vertex  $r$  of  $DFS$  on initial call
Mark[ $0:n - 1$ ] (global array initialized to zeros)
TreeDFS[ $0:n - 1$ ] (global array initialized to zeros)
DFSNumb[ $0:n - 1$ ] (global array)
Lowest[ $0:n - 1$ ] (global array)
i (global integer initialized to -1)
Output: the  $DFS$  and  $Lowest$  numberings of the vertices of  $G$ , and the parent
implementation of the depth-first search tree  $TreeDFS(T_r)$  rooted at vertex  $r$ 
i  $\leftarrow i + 1$ 
Mark[ $v$ ]  $\leftarrow 1$ 
DFSNumb[ $v$ ]  $\leftarrow i$ 
for each  $u$  adjacent to  $v$  do
    if Mark[ $u$ ] = 0 then
        TreeDFS[ $u$ ]  $\leftarrow v$ 
        DFSNumberings( $G, u$ )
    endif
endfor
Min1  $\leftarrow \{Lowest[c] \text{ for each child } c \text{ of } v\}$ 
Min2  $\leftarrow \min \{DFSNumb[w] \mid w \text{ adjacent to } v \text{ but } w \text{ is not the parent of } v\}$ 
Lowest[ $v$ ]  $\leftarrow \min \{DFSNumb[v], Min1, Min2\}$ 
end DFSNumberings

```

REMARK

For simplicity, in our description of the output of *DFSNumberings*, we only described the output for the initial (nonrecursive) call to the procedure. The output from recursive calls also provides the proper updating of *Mark*, *DFSNumb*, *TreeDFS*, and *Lowest*, thereby ensuring that the initial call returns the desired *DFS* and *Lowest* numberings of G .

With the aid of *DFSNumberings*, we now can present the algorithm *ArticulationPoints*, which outputs the set of articulation points of a graph G . For simplicity of pseudocode, we assume that the root vertex for the depth-first search is $r = 0$.



```

procedure ArticulationPoints( $G$ ,  $Art$ )
Input:  $G$  (a connected graph with  $n$  vertices and  $m$  edges)
Output:  $Art$  (the set of articulation points)
     $TreeDFS[0:n - 1]$ ,  $DFSNumb[0:n - 1]$ ,  $Lowest[0:n - 1]$ , and
     $Mark[0:n - 1]$  arrays of size  $n$ 
    for  $v \leftarrow 0$  to  $n - 1$  do //initialize Mark and TreeDFS
         $Mark[v] \leftarrow 0$ 
         $TreeDFS[v] \leftarrow 0$ 
    endfor
     $i \leftarrow 0$ 
     $Art \leftarrow \emptyset$ 
     $DFSNumberings(G, 0)$  //compute  $TreeDFS[u]$ ,  $DFSNumb[u]$ ,  $Lowest[u]$ 
    //for each vertex  $u \in V$  use Proposition 13.2.3
    //to test whether root  $r = 0$  is an articulation point; that is,
    //check to see if vertex 0 has more than one child in  $T_1$ 
     $Number \leftarrow 0$ 
     $c \leftarrow 1$ 
    while  $Number < 2$  .and.  $c \leq n$  do
        if  $TreeDFS[c] = 1$  then // $c$  is a child of root vertex  $r = 1$ 
             $Number \leftarrow Number + 1$ 
        endif
         $c \leftarrow c + 1$ 
    endwhile
    if  $Number = 2$  then //root vertex is an articulation point
         $Art \leftarrow Art \cup \{0\}$ 
    endif
    //use Proposition 13.2.4 and Corollary 13.2.5 to determine nonroot articulation
    //points: whenever a child  $c$  has a parent  $v \neq r$  such that  $Lowest[c] \geq DFSNum[v]$ ,
    //then  $v$  is an articulation point

```

```

for  $c \leftarrow 1$  to  $n - 1$  do
     $v \leftarrow TreeDFS[c]$ 
    if  $Lowest[c] \geq DFSNum[v]$  .and.  $v \neq 0$  then
         $Art \leftarrow Art \cup \{v\}$ 
    endif
endfor
end ArticulationPoints

```

ArticulationPoints outputs $Art = \{0, 1, 2\}$ for the input graph G of Figure 13.6. Note that these are precisely the articulation points of G .

Algorithm *ArticulationPoints* uses the *Lowest* and *DFSTime* numberings to compute the articulation points of a connected graph G . We now describe how we can use these numberings to compute the biconnected components of G . The following key fact shows that we need to do more than just determine the articulation points.

Key Fact

Because vertices can belong to more than one biconnected component, simply knowing the articulation points in a graph G may give very little information about the biconnected components. (Indeed, starting from any graph H , consider the graph G obtained by adding a new vertex v' for each vertex v in H , together with an edge vv' . Then the articulation points of G are simply the vertex set of H , which gives no information about the biconnected components of G belonging to H .) However, each edge belongs to exactly one biconnected component, so that the biconnected components partition the edge set of G .

In addition to maintaining the usual stack of vertices associated with the depth-first search, we maintain a second stack of edges. Initially, this second stack is empty. When an edge uv is first examined during the depth-first search, we push uv on the second stack. Note that we also include edges uv where v has already been visited; that is, edges uv that do not belong to the depth-first search tree. When the depth-first search backtracks along a tree edge uv from v to u , where $Lowest[v] \geq DFSNum[u]$, we then perform a sequence of pops from the second stack until the edge uv is popped. It is easily verified (see Exercise 13.12) that the set of edges popped in this manner are precisely the edge set of a biconnected component of G . Because every edge of G is eventually scanned and pushed on the second stack, all the biconnected components are generated in this fashion.

We now describe of a nonrecursive algorithm for computing both the articulation points and the biconnected components by performing a single depth-first search. Again, two stacks are involved: a DFS-stack of vertices for implementing the depth-first search and a second stack of edges for computing the biconnected components. Initially both stacks are empty. When a vertex u is first visited, we compute the depth-first search number $DFSNumb[u]$ (by incrementing the previously computed depth-first search number and assigning it to $DFSNumb[u]$), and initialize $Lowest[u]$ to $DFSNumb[u]$. When an edge uv is accessed during the depth-first search, we push uv onto the second stack of edges. If uv is not a tree edge, we replace $Lowest[v]$ by the minimum of $Lowest[v]$ and $DFSNumb[v]$. When backtracking along a tree edge uv (that is, when v is popped from the DFS-stack having been being pushed on the stack when exploring u), we replace $Lowest[u]$ by the minimum of $Lowest[u]$ and $Lowest[v]$. If $Lowest[v] \geq DFSNum[u]$, then we pop all the edges down to and including edge uv from the second stack, generating the next biconnected component. We leave it as an exercise to verify the correctness of this algorithm.



13.3 Fault-Tolerant Routing Schemes

A fundamental network problem is to route data from a given node u in the network to another node v . We refer to u as the *source node* and v as the *destination node*. The condition of the network being connected ensures that a packet of data can be routed from u to v along a path from u to v . However, with many possible paths joining u and v , we need a routing scheme to move the packet along a specific path. One such scheme uses a routing table associated with the spanning tree T rooted at the destination node v . For each node u different from v , the routing table stores the parent node of u in T . A packet at any node u can then be rooted to v by successively moving to parent nodes until v is reached. To optimize the length of the path generated, often T is chosen to be a shortest-path tree. In the previous two chapters, we have discussed algorithms for computing such shortest-path trees. By associating a routing table (a spanning tree rooted at v) with each node v of the network, we can route packets from any source node u to any destination node v .

In this section, we discuss a routing scheme that moves data successfully from any vertex u to a destination vertex v in the presence of a single vertex fault (not at u or v) in the network. The routing scheme we design is based on computing two independent spanning trees, T_1 and T_2 , rooted at v .

DEFINITION 13.3.1 We say that two spanning trees, T_1 and T_2 , rooted at a vertex v are *independent* if, for any vertex u , the path in T_1 from u to v and the path in T_2 from u to v have no interior vertices in common.

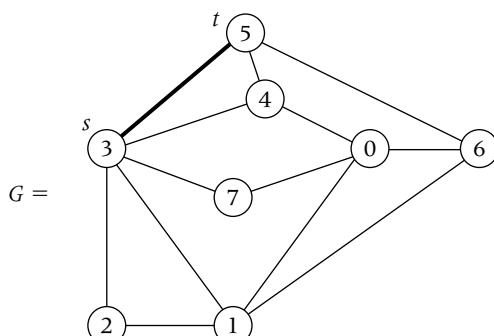
Note that if two spanning trees T_1 and T_2 rooted at a vertex v are independent and a single vertex w (different from u and v) fails, then the path from u to v in at least one of the trees avoids w . In this section, using the notion of an open ear decomposition, we design an algorithm for computing two independent trees rooted at v .

13.3.1 Open Ear Decompositions

The concept of an open ear decomposition, first introduced by Whitney, provides a useful characterization of biconnected graphs and has many algorithmic applications. Given any two adjacent vertices s and t , an *ear decomposition* of G is a sequence of paths P_0, P_1, \dots, P_{q-1} , with P_0 consisting of the single edge st , and where P_i has only its end vertices in common with the subgraph G_i whose vertex set and edge set are the union of the vertex sets and edges sets, respectively, of the simple paths P_0, P_1, \dots, P_{i-1} , $i = 1, \dots, q$. Moreover, the set of paths partition the edges of G , so that $G = G_q$. The term *ear decomposition* derives from the image of P_i being an ear added to the graph G_i (see Figure 13.6). If the two end vertices of each path P_i are distinct—that is, if P_i is not a closed path—then we say that the ear decomposition is open.

FIGURE 13.6

An open ear decomposition P_0, \dots, P_6 .



$$P_0: 3,5 \quad P_1: 3,2,1,6,5 \quad P_2: 3,1 \quad P_3: 1,0,6 \quad P_4: 0,4,5 \quad P_5: 3,4 \quad P_6: 3,7,0$$

We now discuss an algorithm *OpenEarDecomposition* for computing an open ear decomposition of a biconnected graph G , starting with the edge st . We assume that the graph G is implemented using adjacency lists. Recall that the adjacency list for u consists of the set of vertices u_1, u_2, \dots, u_k adjacent to u , so that uu_1, uu_2, \dots, uu_k are the edges incident with u . Initially, the only restriction we place on these lists is that t occurs first in the adjacency list of s .

Algorithm *OpenEarDecomposition* is performed in three stages. In the first stage, we perform a depth-first search starting at vertex s (and first traversing edge st). Similar to the algorithm *ArticulationPoints*, we compute the DFS numbers $DFSNum[0:n - 1]$ and the lowest numbers $Lowest[0:n - 1]$ during this pass.

The second stage consists of a simple rearrangement of the adjacency lists so that, for each vertex u , the first edge uv is either an edge of the DFS-tree such that $Lowest[v] = Lowest[u]$, or a nontree edge such that $DFSNum[v] = Lowest[u]$. In the third stage, we perform a second depth-first search of G starting with edge st using the rearranged adjacency list resulting from the second stage. The sequence of edges generated by this second depth-first search can be partitioned into simple paths, where the first path is the edge st , and each subsequent path begins immediately after a nontree edge has been traversed and ends with a nontree edge. The set of paths generated in this way determines an open ear decomposition.

Pseudocode and the verification of the correctness of the algorithm *OpenEarDecomposition* are developed in the exercises. Because depth-first search has $O(m)$ worst-case complexity, it follows that *OpenEarDecomposition* has $O(m)$ worst-case complexity.

We have just shown how algorithm *EarDecomposition* constructs an open ear decomposition beginning with any edge st in a biconnected graph G . It is not difficult to verify that a graph G that has an open ear decomposition beginning with an edge st is biconnected (see Exercise 13.18). This yields the following classical characterization of biconnected graphs due to Whitney.

Theorem 13.3.1. A graph G is biconnected if and only if for any edge st of G there exists an ear decomposition beginning with ear st . ■

13.3.2 s - t Orderings and Independent Spanning Trees

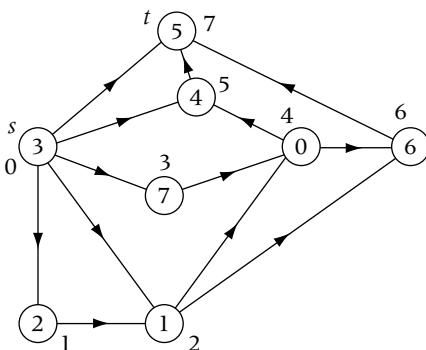
We now show how to compute two independent trees. The first step is to compute an s - t ordering. An s - t ordering of a graph G is an ordering of the vertices such that s is first in the order, t is last, and every other vertex v has at least one vertex u in its neighborhood having a smaller s - t order and one having a larger s - t order.

Associated with an s - t ordering is the notion of an s - t orientation of the edges of G , defined as follows. An *orientation* of G assigns a direction to each edge. An s - t orientation (also called a *bipolar orientation*) of G is an orientation such that the resulting directed graph is acyclic, and s and t are the only source and sink vertices, respectively. (Recall that a *source vertex* is a vertex such that all incident edges are directed out of that vertex, and a *sink vertex* is a vertex such that all incident edges are directed into that vertex.) An s - t orientation can be obtained from an s - t ordering by directing each edge e from the end vertex of e having lower s - t order to the end having higher order. Conversely, given an s - t orientation, we obtain an s - t ordering by performing a topological sort of an associated acyclic digraph. It is easily verified that the topological ordering is an s - t ordering (see Exercise 13.19). Figure 13.7 illustrates an s - t orientation and an associated s - t ordering of a sample biconnected graph for the same graph as in Figure 13.6, where the $(i+1)^{\text{st}}$ vertex in the s - t -ordering is labeled i , $i = 0, \dots, n-1$.

Given an open ear decomposition of a biconnected graph we can compute an s - t orientation, and thus an s - t ordering as follows. Given an ear $P_i = u_1, u_2, \dots, u_p$, we orient the edges of P_i such that it forms a directed path from u_1 to u_p , in which case we say that it is directed out of u_1 , or it forms a directed path from u_p to u_1 , in which case we will say that it is directed into u_1 . We begin by orienting the ears that have one end at s so that they are all directed out of s and push each of them on a stack. While the stack is not empty, we pop an ear P , and

FIGURE 13.7

An s - t orientation and an associated s - t ordering for the graph in Figure 13.6.



we scan P in the direction it is oriented performing the following operations. For each vertex u that is encountered, we orient the ears that are not already oriented and have one end at u so that they are all directed out of u , and we push each of these ears on the stack. It is easily verified that the resulting orientation is an s - t orientation. The s - t orientation of the graph in Figure 13.8 was obtained from the open ear decomposition given in Figure 13.7 using this algorithm. The algorithm just described constructs an s - t orientation from any open ear decomposition with st as the initial ear. Thus, it follows from Theorem 13.3.1 that a biconnected graph has an s - t orientation and s - t ordering for any given edge st . It is easily verified that the converse is true (see Exercise 13.21), yielding the following theorem.

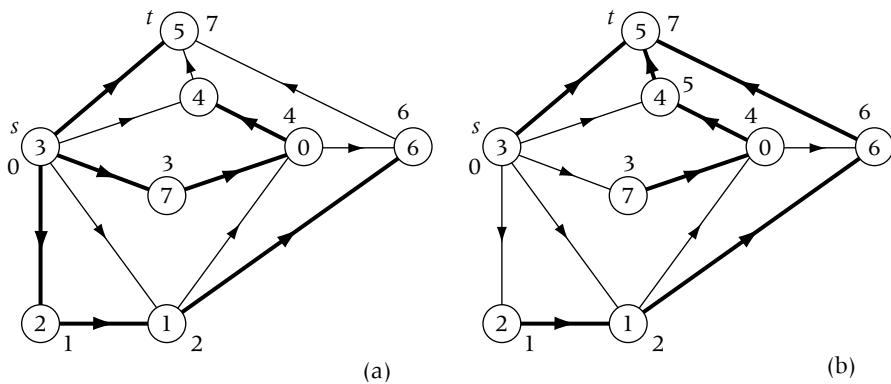
Theorem 13.3.2.

A graph G is biconnected if and only if for any edge st , there exists an s - t orientation (s - t ordering). ■

Given an s - t ordering, we can immediately generate two independent spanning trees rooted at s . For each vertex v different from s , we let $\text{lower}(v)$ be any vertex in the neighborhood of v that has lower s - t order; and for each vertex different from t , we let $\text{higher}(v)$ be any vertex in the neighborhood of v having higher s - t order, respectively. We construct a tree T_1 such that the parent of v is $\text{lower}(v)$ for each vertex different from s and t , and we construct a tree T_2 such that the parent of v is $\text{higher}(v)$ for each vertex different from s and t . In both trees, we make s the parent of t . Equivalently, given an s - t orientation we can generate two independent trees T_1 and T_2 rooted at s by choosing one vertex in the in-neighborhood and out-neighborhood, respectively, of each vertex v different from s and t and making it the parent of v . Again, in both trees, we make s the parent of t . Figure 13.8 illustrates two independent trees constructed using the method just described for the same graph and the same s - t orientation and s - t ordering given in Figure 13.7.

FIGURE 13.8

Two independent spanning trees constructed using the s - t orientation (s - t ordering) of the graph given in Figure 13.7. Part (a) shows the spanning tree T_1 , and part (b) shows the spanning tree T_2 .



We have just described an $O(m + n)$ algorithm *IndependentTrees* for computing two independent trees T_1 and T_2 rooted at a given vertex s of a biconnected graph G . We leave the design of pseudocode for the algorithm *IndependentTrees* as an exercise. It is easily verified that if a graph has two independent trees rooted at each vertex s , the graph is biconnected, which leads to the following theorem.

Theorem 13.3.3. A graph is biconnected if and only if it has two independent spanning trees rooted at any vertex v . \square



13.4 Closing Remarks

Tarjan and Hopcroft in the early 1970s popularized the use of depth-first search in graph algorithms. The concept of biconnectivity generalizes to the concept of k -connectivity. A graph is said to be (vertex) k -connected if it remains connected after the removal of any set of $k - 1$ vertices and their incident edges. In the next chapter, we will consider the problem of determining k -connectivity from the point of view of network flow theory. In the present chapter, we have shown that a biconnected graph has two independent trees rooted at any vertex s . It is an open question whether a k -connected graph has k independent trees rooted at any vertex. The question has been answered affirmatively for $k = 3$.

References and Suggestions for Further Reading

Two references containing the early work of Tarjan and Hopcroft on various applications of depth-first search generated numberings:

Tarjan, R. E. *Data Structures and Network Algorithms*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1983.

Hopcroft, J. E., and R. E. Tarjan. "Efficient Algorithms for Graph Manipulation." *Communications of the ACM* 16, no. 6 (1973): 372–378.

Two early books on algorithmic graph theory:

Even, S. *Graph Algorithms*. Potomac, MD: Computer Science Press, 1979.

Gibbons, A. M. *Algorithmic Graph Theory*. Cambridge, England: Cambridge University Press, 1985.

Chartrand, G., and O. R. Oellerman. *Applied and Algorithmic Graph Theory*. New York: McGraw-Hill, 1993. A more recent book on graph algorithms.

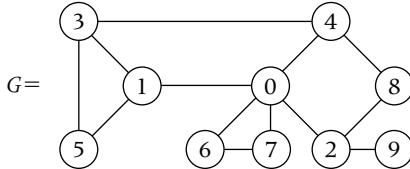
EXERCISES**Section 13.1** Strongly Connected Components

- 13.1 Without using the notion of postnumbering, design an $O(m + n)$ algorithm to test whether a digraph is strongly connected.
- 13.2 Let F be the depth-first out-forest of a digraph D consisting of trees T_1, \dots, T_k , listed in the order they are generated by the out-directed depth-first traversal. Show that a vertex u has postnumber i if u is the $(i + 1)^{\text{st}}$ vertex to be visited in the *postorder traversal* of F , $i = 0, \dots, n - 1$ —that is, the traversal of F consisting of successive postorder traversals of T_j , $j = 1, \dots, k$.
- 13.3 Give a linear-time algorithm for computing the array $\text{PostNumInv}[0:n - 1]$ from the array $\text{PostNum}[0:n - 1]$.
- 13.4 Modify the depth-first traversal procedure DFTOut to compute arrays $\text{PostNum}[0:n - 1]$ and $\text{PostNumInv}[0:n - 1]$.
- 13.5 Show that an out-directed depth-first search starting at a vertex u of a digraph D visits all the vertices in the strongly connected component containing u .

Section 13.2 Articulation Points and Biconnected Components

- 13.6 Prove Theorem 13.2.1.
- 13.7 Prove Proposition 13.2.2.
- 13.8 Prove Proposition 13.2.3.
- 13.9 Prove Proposition 13.2.4.
- 13.10 Prove Proposition 13.2.6.
- 13.11 Verify recurrence relation (13.2.2).
- 13.12 Prove the correctness of the algorithm described for computing the biconnected components of a graph G given its articulation points.
- 13.13 Prove the correctness of the nonrecursive algorithm described for computing the articulation points and the biconnected components of a graph G .

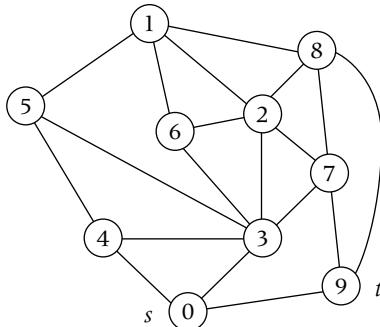
13.14 Consider the following graph G :



- a. Using a depth-first search starting at vertex $r = 3$, compute the global arrays $\text{TreeDFS}[0:9]$, $\text{DFSNumb}[0:9]$, and $\text{Lowest}[0:9]$ output by DFSNumberings .
 - b. Demonstrate how the articulation points are obtained from the arrays computed in part (a).
- 13.15 Let G be graph that is not biconnected, and suppose B_1 and B_2 are any two biconnected components of G . Show that B_1 and B_2 have at most one vertex in common.

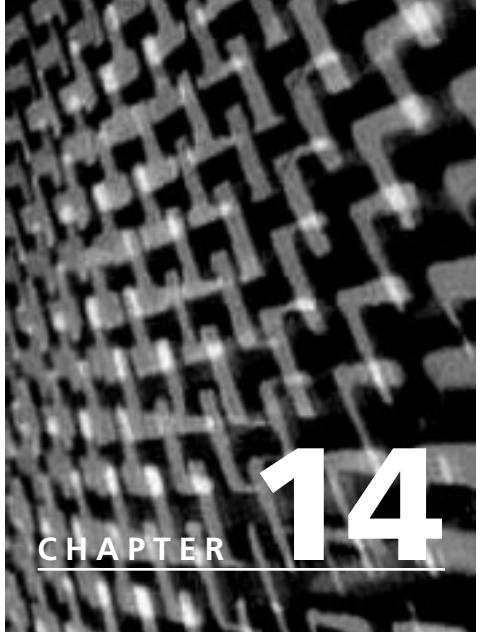
Section 13.3 Fault-Tolerant Routing Schemes

13.16 Consider the following example graph G and vertices s and t .



- a. Demonstrate the action of the algorithm $\text{OpenEarDecomposition}$ for computing an open ear decomposition of G with respect to the edge st . (Assume for definiteness that the adjacency lists are given in increasing order of vertex labels.)
- b. Using the open ear decomposition you computed in part (a), demonstrate the action of the algorithm given in Section 13.3 for computing an $s-t$ orientation.
- c. Using the $s-t$ orientation ($s-t$ ordering) you computed in part (b), compute two independent trees rooted at vertex s .

- 13.17 Verify the correctness of the algorithm *OpenEarDecomposition*.
 - 13.18 Verify that a graph G that has an open ear decomposition beginning with an edge st is biconnected.
 - 13.19 Verify that a topological sorted ordering of an $s-t$ oriented graph is an $s-t$ ordering.
 - 13.20 Verify the correctness of the algorithm for obtaining an $s-t$ orientation from an open ear decomposition.
 - 13.21 Show that a graph G is biconnected if for any edge st , G has an $s-t$ orientation ($s-t$ ordering).
 - 13.22 Show that the trees T_1 and T_2 obtained from an $s-t$ ordering of a biconnected graph G are independent.
 - 13.23 Show that the trees T_1 and T_2 obtained from an $s-t$ orientation of a biconnected graph G are independent.
 - 13.24 Give pseudocode for the algorithm *IndependentTrees*.
-



CHAPTER

14

MATCHING AND NETWORK FLOW ALGORITHMS

Finding matchings and maximum flows in graphs and networks are two fundamental problems with myriad practical applications. We begin this chapter with an algorithm for finding a perfect matching in a bipartite graph and a maximum-weighted perfect matching in a weighted complete bipartite graph. Some of the techniques we use to solving the matching problems, such as finding augmenting paths when constructing a perfect matching in a bipartite graph, can be generalized to apply to the maximum flow problem.

There are many natural interpretations of flows in networks, such as fluid flow through a network of pipelines, data flow through a computer network, traffic flow through a network of highways, current flow through an electrical network, and so forth. Usually, each edge in a network has a certain flow capacity (a *capacitated* network), and the problem arises of finding the maximum flow subject to the capacity constraints of the edges. One of the most celebrated theorems about capacitated networks is the max-flow min-cut theorem of Ford and Fulkerson. In this chapter, we present this theorem and an associated algorithm

for finding a maximum flow in a capacitated network. Finding a maximum flow is useful in solving the well-known marriage problem, which is equivalent to the problem of finding a perfect matching in a bipartite graph.



14.1 Perfect Matchings in Bipartite Graphs

An independent set of edges, or *matching*, in a graph G is a set of edges that are pairwise vertex disjoint. A matching that spans the vertices is called a *perfect matching*. In this section, we discuss an algorithm known as the Hungarian algorithm for finding a perfect matching or determining it does not exist. We also discuss the Kuhn-Munkres algorithm, which employs the Hungarian algorithm to find a maximum-weight perfect matching in an edge-weighted complete bipartite graph. We begin by discussing the marriage problem, which is a colorful interpretation of the problem of finding a perfect matching in a bipartite graph. In Chapter 24, we will give a probabilistic algorithm for determining whether or not a bipartite graph has a perfect matching.

14.1.1 The Marriage Problem

Suppose we have a set of n boys b_1, b_2, \dots, b_n and a set of n girls g_1, g_2, \dots, g_n , where each boy knows some of the girls. The classical *marriage problem* is to determine a necessary and sufficient condition so that each boy can marry a girl that he knows (no two boys can marry the same girl). More formally, the marriage problem is finding the conditions under which the permutation $\sigma: \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$ exists such that boy b_i knows girl $g_{\sigma(i)}$, $i = 1, 2, \dots, n$. If such a permutation σ exists, then the corresponding set of matches $P_\sigma = \{\{b_i, g_{\sigma(i)}\} \mid i = 1, 2, \dots, n\}$ is called a perfect matching.

Let K_i denote the set of girls that boy b_i knows, $i = 1, \dots, n$. For example, suppose that $n = 5$ and the sets K_i , $i = 1, \dots, 5$, are given by

$$K_1 = \{g_1, g_2, g_4, g_5\}, K_2 = \{g_1, g_3\}, K_3 = \{g_2, g_3\}, K_4 = \{g_3, g_4, g_5\}, K_5 = \{g_3\}.$$

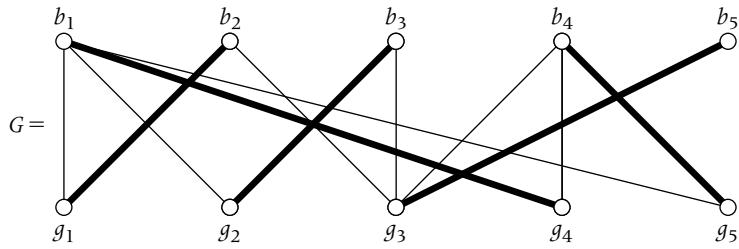
Then, the following set of pairs determines a perfect matching:

$$\{b_1, g_4\}, \{b_2, g_1\}, \{b_3, g_2\}, \{b_4, g_5\}, \{b_5, g_3\}.$$

The marriage problem is naturally modeled using a bipartite graph G , where one set X of the vertex bipartition consists of the set of n boys and the other set Y consists of the set of n girls. A vertex b_i in X is joined to a vertex g_j in Y whenever the boy b_i knows the girl g_j (see Figure 14.1). The marriage problem then becomes determining a necessary and sufficient condition for a bipartite graph to contain a perfect matching—that is, a set of n edges no two of which have a vertex in common.

FIGURE 14.1

The bipartite graph G corresponding to sets
 $K_1 = \{g_1, g_2, g_4, g_5\}$,
 $K_2 = \{g_1, g_3\}$,
 $K_3 = \{g_2, g_3\}$,
 $K_4 = \{g_3, g_4, g_5\}$,
 $K_5 = \{g_3\}$.
 G contains the perfect matching
 $\{b_1, g_4\}, \{b_2, g_1\},$
 $\{b_3, g_2\}, \{b_4, g_5\},$
 $\{b_5, g_3\}$.



Suppose in the previous example that K_1 is replaced with the set

$$K'_1 = \{g_1, g_2, g_3\}.$$

Because the four boys b_1, b_2, b_3, b_5 collectively know only the three girls g_1, g_2, g_3 , a perfect matching cannot exist. More generally, if there exists a set of k boys who collectively know strictly less than k girls, then a perfect matching does not exist. Surprisingly, the converse is also true, by a theorem of Hall, which states that a solution to the marriage problem exists if and only if every set of k boys collectively knows at least k girls, $k \in \{1, 2, \dots, n\}$.

We can restate Hall's theorem for bipartite graphs as follows: For S , a set of vertices of G , let $\Gamma(S)$ denote the set of all vertices that are adjacent to those in S ; that is,

$$\Gamma(S) = \{v \in V \mid \text{there exists a vertex } u \text{ in } S \text{ such that } uv \in E\}. \quad (14.1.1)$$

Theorem 14.1.1

Hall's Theorem

A bipartite graph with vertex bipartition $V = X \cup Y$ contains a perfect matching if and only if, for every subset S of X ,

$$|S| \leq |\Gamma(S)|. \quad (14.1.2)$$

PROOF

First, suppose that there exists a perfect matching M in G . For $u \in V$, let $M(u)$ denote the *mate* of u in M —that is, the unique vertex v such that $uv \in M$. For $S \subseteq X$, let

$$M(S) = \{M(x) \mid x \in S\}. \quad (14.1.3)$$

Clearly, $M(S) \subseteq \Gamma(S)$. Hence,

$$|\Gamma(S)| \geq |M(S)| = |S|. \quad \blacksquare$$

We complete the proof of Theorem 14.1.1 in the next subsection by presenting a “matchmaker” algorithm, called Hungarian, which accepts an initial matching M (possibly empty) and repeatedly augments M until either a perfect matching is generated or a set $S \subseteq X$ is found such that $|\Gamma(S)| < |S|$.

14.1.2 The Hungarian Algorithm

The Hungarian algorithm uses the notion of an M -alternating path. Given a matching M , we say that a vertex v of G is *M -matched* if it is incident with an edge of M ; otherwise, we say that v is *M -unmatched*. An *M -alternating path* is one where the edges alternately belong to $E(G) \setminus M$ and M . An *M -augmenting path* is an alternating path in which the initial and terminal vertices are both M -unmatched.

Key Fact

Given an M -augmenting path P , the size of the matching M can be increased by 1 by removing the edges of M belonging to P , and adding to M the remaining edges of P .

The larger matching described in this key fact can be expressed as the *symmetric difference* $M \oplus E(P)$ of M and the edges $E(P)$ of P ; that is

$$M \oplus E(P) = (M \cup E(P)) \setminus (M \cap E(P)). \quad (14.1.4)$$

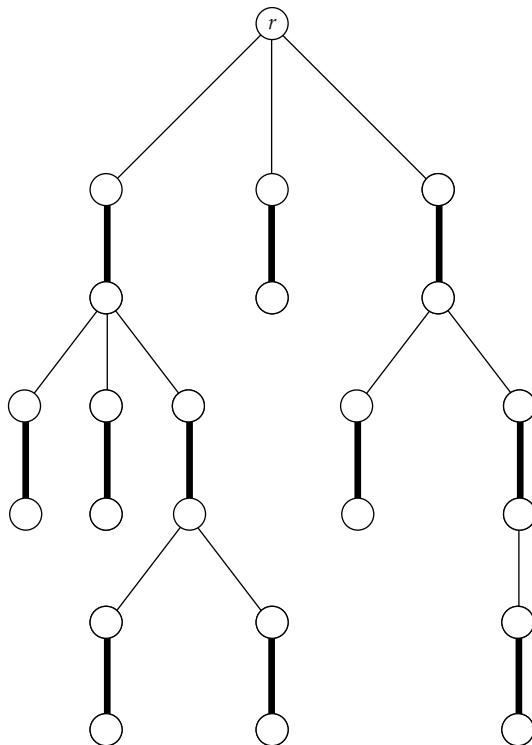
The Hungarian algorithm searches for augmenting paths. To find an M -augmenting path in G , the algorithm grows a tree T , called an *M -alternating tree*, having the following properties:

1. The root r of T is an M -unmatched vertex belonging to X .
2. For each odd integer i less than the depth of T , each edge of T joining a vertex at level i to a vertex at level $i + 1$ belongs to M .
3. The leaf nodes of T all belong to X .

An M -alternating tree is illustrated in Figure 14.2. Clearly, all the paths in an M -alternating tree T are M -alternating paths, and X_T and Y_T denote the subset of vertices of X and Y , respectively, belonging to T .

FIGURE 14.2

An M -alternating tree T rooted at vertex r .



The proofs of the next three lemmas are straightforward and left as exercises.

Lemma 14.1.2

If M is a matching and P is an M -augmenting path, then $M \oplus E(P)$ is a matching of size one greater than M . \square

Lemma 14.1.3

If T is an M -alternating tree, then

$$|X_T| = |Y_T| + 1.$$

(14.1.5) \square

Lemma 14.1.4

Given an M -alternating tree T , if there exists a vertex $x \in X_T$ that is adjacent to an M -unmatched vertex y (not in T), then the path from the root r of T to x , together with the edge xy , determine an M -augmenting path. \square

If all the vertices in X are M -matched, then M is a perfect matching. In this case, the Hungarian algorithm returns the perfect matching M and terminates. In the case where the vertices in X are not all matched, the Hungarian algorithm looks for an augmenting path by growing an M -alternating tree T . The tree T may be initialized to consist of the single vertex r , where r is chosen arbitrarily from the set of unmatched vertices in X . At each stage of growing the tree T , we encounter one of the following three cases:

1. $\Gamma(X_T) - Y_T$ is empty. (Action: algorithm terminates.)

Then by Lemma 14.1.3, $|\Gamma(X_T)| = |Y_T| = |X_T| - 1$. The Hungarian algorithm then returns $S = X_T$ and terminates, because by Theorem 14.1.1, no perfect matching exists.

In the next two cases, $\Gamma(X_T) - Y_T$ is nonempty, and we choose y to be any vertex in $\Gamma(X_T) - Y_T$ and x to be any vertex in X_T adjacent to y .

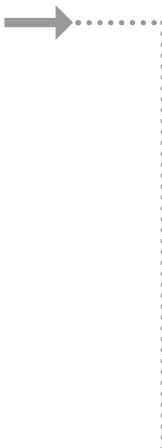
2. Vertex y is matched. (Action: T is augmented.)

Then we augment T to obtain a new M -alternating tree by adding the edges xy and yz , where z is the mate of y in M .

3. Vertex y is unmatched. (Action: M is augmented.)

Then an augmenting path P in T from r to x together with the edge xy has been found. We then replace M by the augmented matching $M \oplus E(P)$ containing one more edge.

In case 3, we either have found a perfect matching or we look for another augmenting path by growing another M -alternating tree rooted at a new unmatched vertex in X . Clearly, M can be augmented at most n times. Because at each stage the alternating tree T can be grown in time $O(n^2)$, the worst-case complexity of the following procedure *Hungarian* is $O(n^3)$. Pseudocode for *Hungarian* follows.



```

procedure Hungarian( $G, M, S$ )
Input:  $G$  (a bipartite graph with vertex bipartition  $(X, Y)$ )
         $M$  (an initial matching, possibly empty)
Output:  $M$  (a perfect matching if one exists)
         $S$  (a set of vertices with the property that  $|\Gamma(S)| < |S|$  if no perfect matching
        exists)

     $AugmentingM \leftarrow .true.$ 
    while  $AugmentingM$  do
        if all the vertices in  $X$  are  $M$ -matched then //  $M$  is a perfect matching
             $AugmentingM \leftarrow .false.$ 
        else // Grow  $M$ -alternating tree  $T$ 
             $r \leftarrow$  any  $M$ -unmatched vertex in  $X$ 
             $T \leftarrow$  tree consisting of the single vertex  $r$ 
             $GrowingTree \leftarrow .true.$ 

```

```

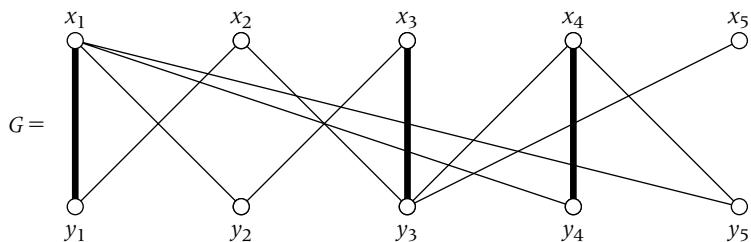
while GrowingTree do
  if  $\Gamma(X_T) = Y_T$  then
     $S \leftarrow X_T$                                 // $|\Gamma(S)| < |S|$ 
    GrowingTree  $\leftarrow$  .false.
    AugmentingM  $\leftarrow$  .false.
  else
     $y \leftarrow$  any vertex in  $\Gamma(X_T) - Y_T$ 
     $x \leftarrow$  any vertex in  $X_T$  adjacent to adjacent to  $y$ 
    if  $y$  is  $M$ -matched then                  //augment  $T$ 
       $z \leftarrow M(y)$                           // $z$  is the mate of  $y$  in  $M$ 
       $T \leftarrow T \cup xy \cup yz$ 
    else                                         //an  $M$ -augmenting path has been
      found
       $P \leftarrow$  path in  $T$  from  $r$  to  $x$  together with the edge  $xy$ 
       $M = M \oplus E(P)$                       //augment  $M$ 
      GrowingTree  $\leftarrow$  .false.
    endif
  endif
  endwhile
endif
endwhile
end Hungarian

```

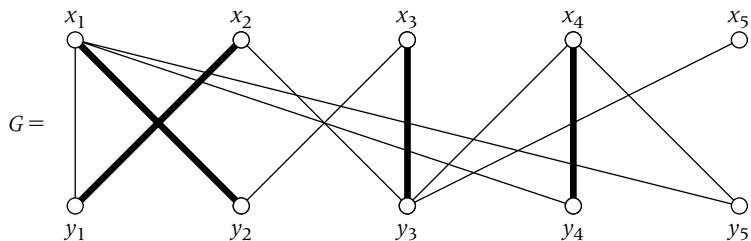
Figure 14.3 illustrates the action of procedure *Hungarian* for a sample bipartite graph. For definiteness, in Figure 14.3 we show the perfect matching generated by always choosing vertices in order of their vertex number. More precisely, the phrase “any ... vertex” occurring in three statements in the pseudocode for *Hungarian* is replaced by the phrase “the ... vertex of smallest index” when generating the perfect matching in Figure 14.3. We leave the intermediate steps in finding the augmenting paths as an exercise.

FIGURE 14.3

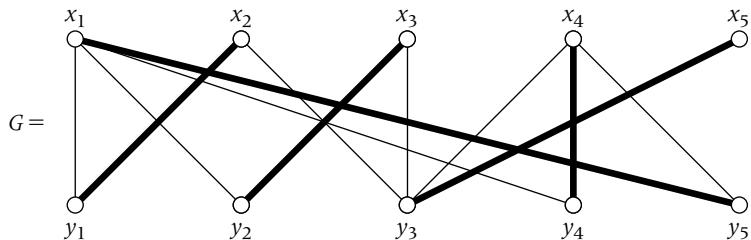
Action of procedure *Hungarian* for a sample bipartite graph with initial matching $\{x_1, y_1\}$, $\{x_3, y_3\}$, $\{x_4, y_4\}$. Vertices are considered in the order of their indices.



Sample bipartite graph G and initial matching $\{x_1, y_1\}, \{x_3, y_3\}, \{x_4, y_4\}$



A first M -augmenting path $P: x_2y_1x_1y_2$, found by *Hungarian* with starting M -unmatched vertex x_2 yielding augmented matching $M \oplus E(P) = \{x_1y_2, x_2y_1, x_3y_3, x_4y_4\}$



A second M -augmenting path $P: x_5y_3x_3y_2x_1y_5$, found by *Hungarian* with starting M -unmatched vertex x_5 yielding augmented perfect matching $M \oplus E(P) = \{x_1y_5, x_2y_1, x_3y_2, x_4y_4, x_5y_3\}$

14.1.3 Maximum Perfect Matching in a Weighted Bipartite Graph

Suppose n workers x_1, x_2, \dots, x_n are to be assigned to n jobs y_1, y_2, \dots, y_n , where each worker is qualified to perform any of the jobs. Associated with each worker-job pair (x_i, y_j) is a weight ω_{ij} measuring how effectively worker x_i can

perform job y_j . The natural problem arises of finding assignments of workers to jobs so that the total effectiveness of each workers is optimized. This problem can be modeled using a weighted complete bipartite graph $G = (V, E)$, with vertex bipartition $X = \{x_1, \dots, x_n\}$ and $Y = \{y_1, \dots, y_n\}$. Each edge $x_i y_j$ of G is assigned the weight ω_{ij} , $i, j \in \{1, \dots, n\}$. Clearly, a perfect matching in G corresponds to an assignment of each worker to a job so that no two workers are assigned the same job. We define the *weight* of a perfect matching M , denoted by $\omega(M)$, to be the sum of the weights of its edges.

$$\omega(M) = \sum_{e \in M} \omega(e). \quad (14.1.6)$$

A *maximum perfect matching* is a perfect matching of maximum weight over all perfect matchings of G . Because G is complete, any permutation π of $\{1, 2, \dots, n\}$ determines a perfect matching $M_\pi = \{x_i y_{\pi(i)} \mid i \in \{1, \dots, n\}\}$ and conversely. Thus, a brute-force algorithm that enumerates all $n!$ perfect matchings and chooses one of maximum weight is hopelessly inefficient.

We now describe an $O(n^3)$ algorithm due to Kuhn and Munkres for finding a maximum perfect matching in a weighted complete bipartite graph. The Kuhn-Munkres algorithm uses the Hungarian algorithm, together with the notion of a feasible vertex weighting.

DEFINITION 14.1.1 A *feasible vertex weighting* is a mapping ϕ from V to the real numbers such that for each edge $xy \in E$ (each $x \in X$ and $y \in Y$),

$$\phi(x) + \phi(y) \geq \omega(xy). \quad (14.1.7)$$

Any sufficiently large ϕ is a feasible vertex weighting. For example, the following vertex weighting is feasible:

$$\phi(v) = \begin{cases} \max \{\omega(vy) \mid vy \in E(G)\} & \text{if } v \in X, \\ 0 & \text{otherwise} \end{cases} \quad (14.1.8)$$

The following proposition is easily verified.

Proposition 14.1.5 Let ϕ be any feasible vertex weighting and M any perfect matching of G . Then,

$$\omega(M) \leq \sum_{v \in V} \phi(v). \quad (14.1.9) \quad \square$$

Given a vertex weighting ϕ , the *equality subgraph* $G_\phi = (V_\phi, E_\phi)$ is a subgraph of G such that $V_\phi = V(G)$ and E_ϕ consists of all the edges xy such that $\omega(xy) = \phi(x) + \phi(y)$. Note that it is possible for some of the vertices of G to be isolated vertices in G_ϕ .

proposition 14.1.6 Let ϕ be any feasible vertex weighting. If the equality subgraph G_ϕ contains a perfect matching M , then M is a maximum perfect matching in G . \square

Observe that if M is a perfect matching in G_ϕ , then $\omega(M) = \sum_{v \in V} \phi(v)$. Hence, Proposition 14.1.6 follows from Proposition 14.1.5.

Starting with an initial feasible vertex weighting, such as the one given by Formula (14.1.8), the Kuhn-Munkres algorithm applies the Hungarian algorithm to the subgraph G_ϕ . If a perfect matching M_ϕ is found, then by Proposition 14.1.5, M_ϕ is a maximum perfect matching in G . On the other hand, suppose the Hungarian algorithm terminates by finding a matching M and an M -alternating tree T such that $\Gamma(X_T) - Y_T$ is empty. Then we have found a set $S = X_T$ such that $|\Gamma_\phi(S)| = |S| - 1 < |S|$. Instead of terminating at this point, we replace ϕ with a new feasible weighting ϕ' and continue the Hungarian algorithm in the equality graph $G_{\phi'}$ of ϕ' . The new feasible weighting ϕ' is constructed as follows. Set

$$\varepsilon = \min\{\phi(x) + \phi(y) - \omega(xy) \mid x \in S, y \in Y - \Gamma_\phi(S)\}. \quad (14.1.10)$$

For each $v \in V(G)$, the weighting ϕ' is defined by

$$\phi'(v) = \begin{cases} \phi(v) - \varepsilon & v \in S, \\ \phi(v) + \varepsilon & v \in \Gamma_\phi(S), \\ \phi(v) & \text{otherwise.} \end{cases} \quad (14.1.11)$$

It is easily verified that the vertex weighting ϕ' given by (14.1.11) is a feasible vertex weighting. Moreover, the following key fact allows us to continue growing the M -alternating tree T .

Key Fact

The equality subgraph $G_{\phi'}$ contains the M -alternating tree T . Further, $G_{\phi'}$ contains at least one edge $\{x, y\}$, where $x \in S$ and $y \in Y - \Gamma_{\phi'}(S)$. Since y is unmatched by M , the path in $G_{\phi'}$ consisting of the path in T from its root to x together with edge xy is an M -augmenting path.

Thus, by continuing the Hungarian algorithm in $G_{\phi'}$, the matching M will be augmented by at least one edge. After at most $n/2$ steps in which ϕ is replaced with ϕ' , we obtain a perfect matching M and a feasible weighting ϕ^* such that M is contained in the equality subgraph G_{ϕ^*} of ϕ^* . By Proposition 14.1.6, such a perfect matching will necessarily be a maximum perfect matching in G .

In the following pseudocode for the Kuhn-Munkres algorithm, we call the procedure *Hungarian2*, which is identical to *Hungarian* except that the alternating tree T is added as an input/output parameter.



```

procedure KuhnMunkres( $G, \phi, M$ )
Input:  $G$  (a weighted complete bipartite graph with vertex bipartition  $(X, Y), |X| = |Y| = n$ )
 $\phi$  (a positive edge weighting of  $G$ )
Output:  $M$  (a maximum perfect matching)
 $\phi \leftarrow$  any feasible vertex weighting //in particular, the one given by
 $\phi$  //in particular, the one given by
 $G_\phi \leftarrow$  equality subgraph for  $\phi$ 
 $M \leftarrow$  any matching in  $G_\phi$  //in particular,  $M$  can be chosen to be
empty
PerfectMatchingFound  $\leftarrow$  .false.
while .not. PerfectMatchingFound do
    Hungarian2( $G_{\phi'}, M, S, T$ )
    if  $M$  is a perfect matching then
        PerfectMatchingFound  $\leftarrow$  .true.
    else
         $\varepsilon \leftarrow \min \{ \phi(x) + \phi(y) - \omega(xy) \mid x \in S, y \in Y - \Gamma_\phi(S) \}$ 
        for all  $x \in S$  do
             $\phi(x) = \phi(x) - \varepsilon$ 
        endfor
        for all  $y \in \Gamma_\phi(S)$  do
             $\phi(y) = \phi(y) + \varepsilon$ 
        endfor
    endif
endwhile
end KuhnMunkres

```

We leave it as an exercise to show that the worst-case complexity of procedure *KuhnMunkres* is $O(n^3)$

14.2 Maximum Flows in Capacitated Networks

The problem of finding a maximum flow in a network from a source s to a sink t , where each link in the network has a given capacity, and the dual problem of finding a minimum cut in such a network, is a classical optimization problem with many applications. Our study of flows in networks begins by formally defining the notion of a flow in a digraph (an *uncapacitated* network) and establishing some elementary results about flows.

REMARK

The theory of flows in networks modeled on digraphs includes as an important special case flows modeled on graphs by associating with the graph its combinatorially equivalent (symmetric) digraph.

14.2.1 Flows in Digraphs

Let $D = (E, V)$ be a digraph with vertex set V and directed edge set E . A *real weighting* ω of the edges of D is a mapping from E to the set \mathbb{R} of real numbers. We refer to $\omega(e)$ as the ω -*weight* of edge e . For $v \in V$, we let $\sigma_{\text{in}}(\omega, v)$ and $\sigma_{\text{out}}(\omega, v)$ denote the sum of the ω -weights over all the edges having head v and tail v , respectively, so that

$$\sigma_{\text{in}}(\omega, v) = \sum_{uv \in E} \omega(uv),$$

$$\sigma_{\text{out}}(\omega, v) = \sum_{vw \in E} \omega(vw).$$

By convention, $\sigma_{\text{in}}(\omega, v) = 0$ if there are no edges having head v . Similarly, $\sigma_{\text{out}}(\omega, v) = 0$ if there are no edges having tail v .

The following proposition is easily verified.

Proposition 14.2.1 Given any real weighting ω of the edges in a digraph D ,

$$\sum_{v \in V} (\sigma_{\text{in}}(\omega, v) - \sigma_{\text{out}}(\omega, v)) = 0.$$

□

Now suppose we are given two vertices s and t such that there are no edges having head s or tail t . A *flow* f from s to t is a weighting of the edges such that

$$\sigma_{\text{in}}(f, v) = \sigma_{\text{out}}(f, v), \quad \forall v \in V \setminus \{s, t\}. \quad (\text{14.2.1})$$

Given any vertex v in D , we refer to $\sigma_{\text{in}}(f, v)$ and $\sigma_{\text{out}}(f, v)$ as the *flow into* v and the *flow out of* v , respectively. Formula (14.2.1) is called a *flow conservation equation*. The *value of flow* f , denoted by $\text{val}(f)$, is defined to be the flow out of s . It follows easily from Proposition 14.2.1 that the flow out of s equals the flow into t . Hence,

$$\text{val}(f) = \sigma_{\text{out}}(f, s) = \sigma_{\text{in}}(f, t). \quad (\text{14.2.2})$$

A *unit flow* is a flow f such that $\text{val}(f) = 1$. Figure 14.4 illustrates a flow f of value 55 on a sample digraph D .

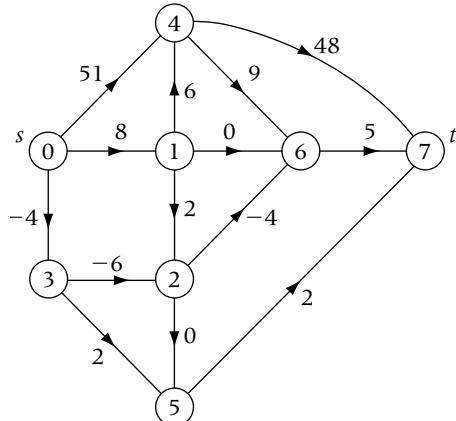
The following proposition is easily verified.

Proposition 14.2.2 The set of all flows is closed under linear combinations; that is, for any flows f_1 and f_2 and real numbers λ_1 and λ_2 , $\lambda_1 f_1 + \lambda_2 f_2$ is a flow. Moreover,

$$\text{val}(\lambda_1 f_1 + \lambda_2 f_2) = \lambda_1 \text{val}(f_1) + \lambda_2 \text{val}(f_2).$$

FIGURE 14.4

A flow of value 55.



Now consider a directed path $P = se_1u_1e_2u_2 \dots u_{p-1}e_pt$ from s to t (not necessarily a simple path). Associated with P is the flow χ_P from s to t given by

$$\chi_P(e) = \begin{cases} 1 & e \in E(P) \\ 0 & e \notin E(P), \quad \forall e \in E. \end{cases}$$

We call χ_P the *characteristic flow* of P . Note that the characteristic flow is a *unit flow*.

In the next section, we compute maximum flows in a capacitated network by using the notion of a semipath, which is a path where the orientation of the edges is ignored. More precisely, a *semipath* S from s to t is an alternating sequence of vertices and edges $se_1u_1e_2u_2 \dots u_{p-1}e_pt$ such that either e_i has tail u_{i-1} and head u_i (e_i is a *forward edge of S*) or e_i has tail u_i and head u_{i-1} (e_i is a *backward edge of S*), $i = 1, \dots, p$, where $u_0 = s$ and $u_p = t$. Associated with semipath S is the flow χ_S , called the *characteristic flow* of S , given by

$$\chi_S(e) = \begin{cases} 1 & e \text{ is a forward edge of } S, \\ -1 & e \text{ is a backward edge of } S, \\ 0 & \text{otherwise,} \end{cases} \quad \forall e \in E. \quad (14.2.3)$$

It is easily verified that χ_S is a unit flow.

14.2.2 Flows in Capacitated Networks

A *capacitated network* N (sometimes called a *transportation* or *flow* network) consists of the 4-tuple (D, s, t, c) , where $D = (V, E)$ is a digraph; s and t are two distinguished vertices of D called the *source* and *sink*, respectively; and c is a positive real weighting of the edges, called the *capacity weighting* of D . We assume that all the edges incident with s are directed out of s , and all the edges incident with t are directed into t . We refer to $c(e)$ as the *capacity* of edge e .

A *flow* f in a capacitated network N is a flow in D from s to t such that for each $e \in E$, $f(e)$ is nonnegative and does not exceed the capacity of e ; that is

$$0 \leq f(e) \leq c(e), \quad \forall e \in E.$$

Figure 14.5 shows a flow of value 23 in a sample capacitated network N on eight vertices. In Figure 14.5 and all subsequent figures, when illustrating a flow f , all edges e such that $f(e) = 0$ are omitted.

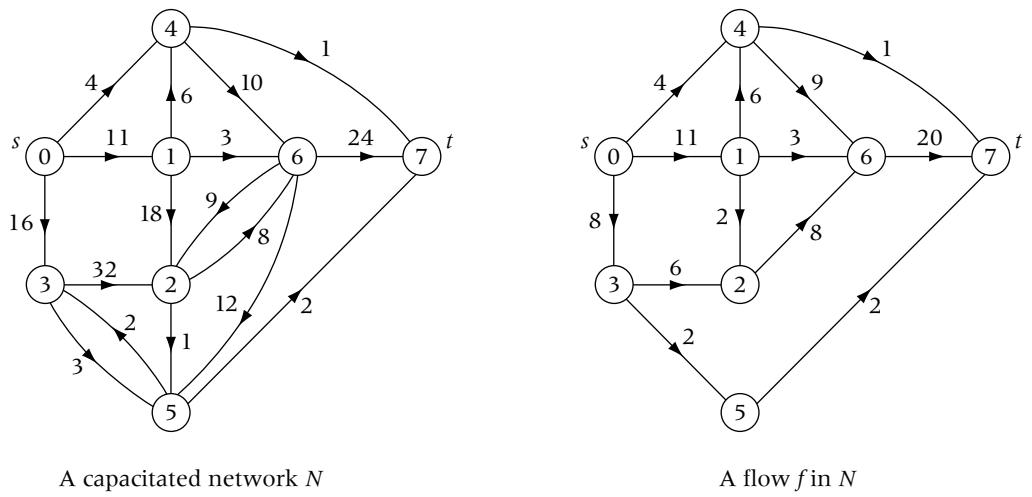


FIGURE 14.5
A sample capacitated network N and a flow f having value $\text{val}(f) = 23$.

A *maximum (value) flow* is a flow f in N whose value is maximum over all flows in N . A naive attempt to finding a maximum flow might proceed as follows. First we find a path P_1 from s to t in N using an algorithm such as a breadth-first search. Let λ_1 denote the minimum capacity among all the edges of P_1 . Then $f = \lambda_1 \chi_{P_1}$ is a flow in N having value λ_1 . Next, we adjust the capacities of the edges of N by subtracting λ_1 from each edge belonging to P_1 and deleting all the edges of N whose capacity becomes zero. We then find a path P_2 (if one exists) in the new network. We now augment the flow f by $\lambda_2 \chi_{P_2}$, where λ_2 is the minimum capacity (in the new network) among all the edges of P_2 . The current flow $f = \lambda_1 \chi_{P_1} + \lambda_2 \chi_{P_2}$ has value $\lambda_1 + \lambda_2$. Again, we subtract λ_2 from every edge belonging to P_2 and delete all the edges whose capacity becomes zero. Continuing in this way, we find paths P_3, \dots, P_k and associated real numbers $\lambda_3, \dots, \lambda_k$, respectively, until no paths are left from s to t in the final network.

We illustrate the action of our naive attempt in Figure 14.6 for a sample capacitated network. Unfortunately, the final flow f attained has value 14, whereas the maximum flow shown in Figure 14.7 has value 24. However, any flow generated by our naive attempt does have a maximality property—namely, if g is any flow different from f , then $g(e) < f(e)$ for some edge e .

FIGURE 14.6

Action of naive attempt to find maximum flow for a sample capacitated network N , yielding flow $f = 10\chi_{P_1} + \chi_{P_2} + 3\chi_{P_3}$ having value 14. This flow is suboptimal because a maximum flow f^* shown in Figure 14.7 has value 24.

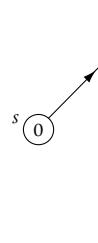
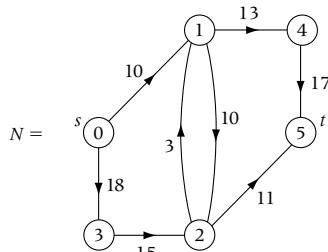
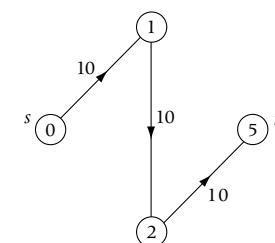
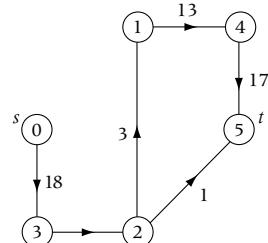
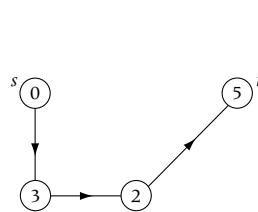
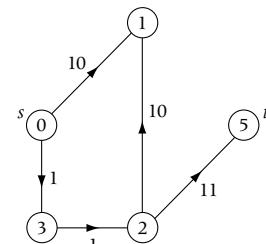
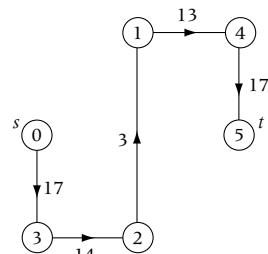
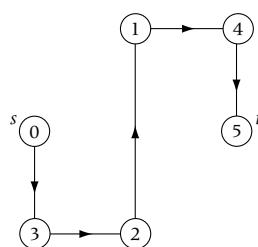
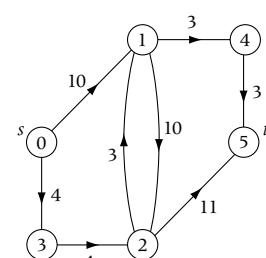
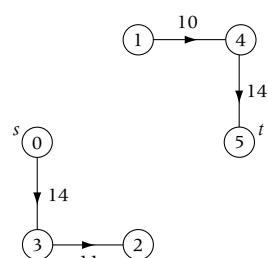
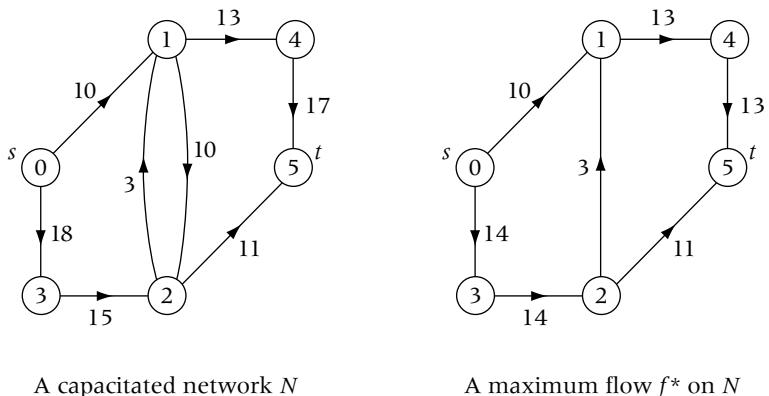
path $P_1, \lambda_1 = 10$  $f = 10\chi_{P_1}$ reduced N path $P_2, \lambda_2 = 1$  $f = 10\chi_{P_1} + \chi_{P_2}$ reduced N path $P_3, \lambda_3 = 3$  $f = 10\chi_{P_1} + \chi_{P_2} + 3\chi_{P_3}$ reduced N

FIGURE 14.7

A maximum flow f^* having value 24 in the capacitated network N of Figure 14.5.



The flow f of Figure 14.6 generated by our naive algorithm is a dead end in our search for a maximum flow in the sense that we can no longer augment f by simply finding directed paths from s to t . The following key fact using the notion of a semipath allows us to continue to augment the flow f .

Key Fact

Given a semipath with no forward edge used to capacity and nonzero flow in each backward edge, we can continue to augment f by adding flow to forward edges while removing flow from backward edges.

14.2.3 Finding an Augmenting Semipath

The *residual capacity* with respect to a flow f is $c(e) - f(e)$. An edge e is f -*saturated* if the residual capacity is zero; otherwise, edge e is f -*unsaturated*. A semipath S is an f -*augmenting semipath* if every forward edge of S is f -unsaturated and $f(e) > 0$ for every backward edge e of S . For $e \in E(S)$, we let

$$c_f(S, e) = \begin{cases} c(e) - f(e) & e \text{ is a forward edge of } S, \\ f(e) & e \text{ is a backward edge of } S. \end{cases}$$

Let $c_f(S)$ denote the minimum value of $c_f(S, e)$ among all the edges of S ; that is,

$$c_f(S) = \min\{c_f(S, e) \mid e \in E(S)\}.$$

Now consider the edge weighting \hat{f} given by

$$\hat{f} = f + c_f(S)\chi_S, \quad (14.2.4)$$

where χ_S is the characteristic flow of S given by Formula (14.2.3). Thus, for each $e \in E(G)$,

$$\hat{f}(e) = \begin{cases} f(e) + c_f(S) & e \text{ is a forward edge of } S, \\ f(e) - c_f(S) & e \text{ is a backward edge of } S, \\ f(e) & \text{otherwise.} \end{cases}$$

Proposition 14.2.2 implies that the edge weighting \hat{f} given by Formula (14.2.4) is a flow in the digraph D . Further, it is immediate from the definition of $c_f(S)$ that for all $e \in E$,

$$0 \leq \hat{f}(e) \leq c(e).$$

Hence, \hat{f} is a flow in the capacitated network N . Further, since χ_S is a unit flow, it follows from Proposition 14.2.2 that

$$val(\hat{f}) = val(f) + c_f(S),$$

so that \hat{f} has a strictly greater value than f .

A semipath S and the value $c_f(S)$ can be computed by finding a path from s to t in the f -derived network N_f constructed from the network N and the flow f . The network N_f is obtained by starting with vertex set V and adding edges as follows. For each edge uv of N that is f -unsaturated, we add an edge uv to N_f having weight $c(uv) - f(uv)$. For each edge uv of N such that $f(uv) > 0$, we add an edge vu to N_f having weight $f(uv)$. When uv and vu are both edges of N , it is possible for N_f to have two edges joining vertex u to vertex v . In our search for a maximum flow, allowing such pairs of edges does not present a problem. In fact, we can even eliminate the possibility of such pairs in N_f by replacing f with a new flow f' obtained by subtracting $\min\{f(uv), f(vu)\}$ from both $f(uv)$ and $f(vu)$ so that one of them becomes equal to zero. The latter operation does not affect the value of the flow f . Note that if $val(f) = 0$, then $N_f = N$. The f -derived network N_f for a sample network N and flow f is illustrated in Figure 14.8.

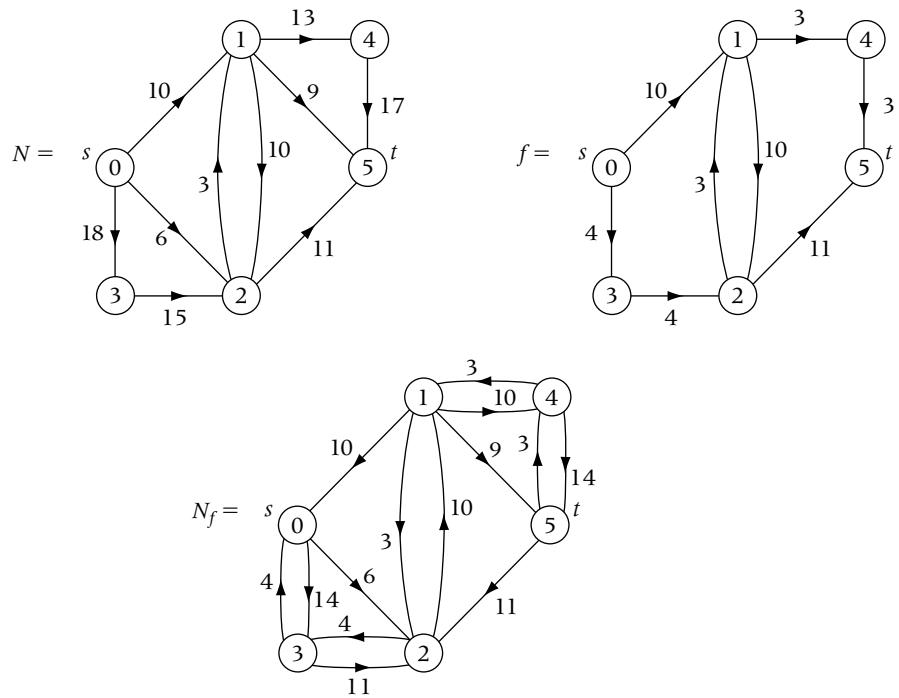
For P , a path from s to t in N_f , we define the weight $\mu(P)$ to be the minimum weight over all the edges of P ; that is,

$$\mu(P) = \min\{w(e) | e \in E(P)\}.$$

The following proposition follows immediately from the definitions of N_f and the f -augmenting semipath S .

FIGURE 14.8

The f -derived network N_f for a sample network N and flow f .



Proposition 14.2.3 If P is a path from s to t in N_f , then the corresponding semipath S in N is an f -augmenting semipath. Further,

$$c_f(S) = \mu(P).$$

It follows from Formula (14.2.4) that if there exists an f -augmenting semipath S , then we can find a flow whose value is strictly greater than f . The Ford-Fulkerson algorithm is based on the fact that the converse is also true; that is, *if there is no f -augmenting semipath, then f is a maximum flow*. To prove this important result, we introduce the concept of a cut.

14.2.4 Bounding Flow Values by Capacities of Cuts

Consider a bipartition of the vertex set V into two disjoint sets X and Y . We denote this bipartition by (X, Y) . The cut associated with the bipartition (X, Y) , denoted $\text{cut}(X, Y)$, is defined by

$$\text{cut}(X, Y) = \{xy \in E \mid x \in X, y \in Y\}.$$

We say a set of edges Γ is a *cut* if $\Gamma = \text{cut}(X, Y)$ for some bipartition (X, Y) of V . For $u, v \in V$, if $u \in X$ and $v \in Y$, then we say that Γ separates u and v . Unless otherwise stated, we assume that Γ separates the source s from the sink t . The *capacity* of Γ , denoted by $\text{cap}(\Gamma)$, is the sum of the capacities of all the edges in Γ ; that is,

$$\text{cap}(\Gamma) = \sum_{e \in \Gamma} c(e).$$

A *minimum capacity cut* (or simply *minimum cut*) is a cut Γ whose capacity is minimum over all cuts separating s and t .

Because deleting all the edges of a cut disconnects the source s from the sink t , intuitively we would expect that the value of any flow f is not greater than the capacity of any cut Γ . The following proposition affirms this intuition.

Proposition 14.2.4 Let f be any flow from s to t in N , and let $\Gamma = (X, Y)$ be any cut separating s and t . Then, the value of f is bounded above by the capacity of Γ ; that is,

$$\text{val}(f) \leq \text{cap}(\Gamma).$$

PROOF

Given a nonnegative weighting ω of E , we extend ω to a mapping of all $V \times V$ as follows:

$$\omega(u, v) = \begin{cases} \omega(uv) & uv \in E, \\ 0 & \text{otherwise.} \end{cases}$$

For $A, B \subseteq V$, let

$$\omega(A, B) = \sum_{a \in A} \sum_{b \in B} \omega(a, b). \quad (14.2.5)$$

In the notation of Formula (14.2.5), the flow conservation equation (14.2.1) can be rewritten as follows:

$$f(u, V) - f(V, u) = 0, \quad u \in V - \{s, t\}. \quad (14.2.6)$$

Formula (14.2.2) then becomes

$$f(s, V) = f(V, t) = \text{val}(f). \quad (14.2.7)$$

It follows immediately from Formulas (14.2.6) and (14.2.7) that

$$f(X, V) - f(V, X) = \text{val}(f). \quad (14.2.8)$$

Clearly, for U , which is any subset of V , we have

$$\begin{aligned} f(U, V) &= f(U, X) + f(U, Y), \\ f(V, U) &= f(X, U) + f(Y, U). \end{aligned} \quad (14.2.9)$$

Substituting $U = X$ in both parts of Formula (14.2.9), subtracting the second part from the first, and employing Formula (14.2.8) yields

$$f(X, Y) - f(Y, X) = f(X, V) - f(V, X) = \text{val}(f). \quad (14.2.10)$$

Using Formula (14.2.10), we have

$$\begin{aligned} \text{val}(f) &= f(X, Y) - f(Y, X) \\ &\leq f(X, Y) \\ &= \sum_{xy \in \Gamma} f(xy) \\ &\leq \sum_{xy \in \Gamma} c(xy) \quad (\text{since } f(e) \leq c(e) \text{ for all } e \in E) \\ &= \text{cap}(\Gamma). \end{aligned}$$

■

Corollary 14.2.5 If f is a flow from s to t and Γ is a cut separating s and t such that $\text{val}(f) = \text{cap}(\Gamma)$, then f is a maximum flow and Γ is a minimum cut.

PROOF

Let f' be any flow from s to t and let Γ' be any cut separating s and t . Then by Proposition 14.2.4, we have

$$\begin{aligned} \text{val}(f') &\leq \text{cap}(\Gamma) = \text{val}(f), \\ \text{cap}(\Gamma') &\geq \text{val}(f) = \text{cap}(\Gamma). \end{aligned}$$

■

Corollary 14.2.5 states a condition guaranteeing that f is a maximum flow and Γ is a minimum cut, but it does not tell us whether such a flow f and cut Γ actually exist. In fact, a seminal result in flow theory is that such a flow f and cut Γ always exist.

Theorem 14.2.6 Max-Flow Min-Cut Theorem

Let $N = (D, c, s, t)$ be a capacitated network. The maximum value of a flow from s to t equals the minimum capacity of a cut separating s and t . \square

To prove Theorem 14.2.6, we use Corollary 14.2.5 to establish it is sufficient to exhibit a flow f and cut Γ such that $\text{val}(f) = \text{cap}(\Gamma)$. In the next subsection, we give an algorithm for computing such a flow f and cut Γ .

14.2.5 The Ford-Fulkerson Algorithm

The following procedure computes a maximum flow and minimum cut by repeatedly augmenting the current flow f using an f -augmenting semipath.



```

procedure FordFulkerson( $N, f, \Gamma$ )
  Input:  $N = (D, s, t, c)$  (a capacitated network)
  Output:  $f$  (maximum flow)
             $\Gamma$  (minimum cut)

   $f \leftarrow 0$ 
   $N_f \leftarrow N$ 
  while there is a directed path from  $s$  to  $t$  in the  $f$ -derived network  $N_f$  do
     $P \leftarrow$  a path from  $s$  to  $t$  in  $N_f$ 
     $S \leftarrow$  the  $f$ -augmenting semipath in  $N$  corresponding to path  $P$  in  $N_f$ 
     $c_f(S) \leftarrow \mu(P)$            // $\mu(P)$  is the minimum weight over all the edges of  $P$ 
     $f \leftarrow f + c_f(S)\chi_S$       //augment  $f$ 
    update  $N_f$ 
  endwhile

   $X \leftarrow$  set of vertices that are accessible from  $s$  in  $N_f$            // $s \in X$ 
   $Y \leftarrow$  set of vertices that are not accessible from  $s$  in  $N_f$            // $t \in Y$ 
   $\Gamma \leftarrow \text{cut}(X, Y)$ 
end FordFulkerson

```

The correctness of procedure *FordFulkerson* is established with the aid of the following two lemmas, whose proofs are left as exercises.

Lemma 14.2.7

Let f and (X, Y) be the flow and vertex bipartition, respectively, generated by procedure *FordFulkerson*. Then every edge xy of the cut $\Gamma = \text{cut}(X, Y)$ is saturated; that is,

$$f(xy) = c(xy).$$

\square

Lemma 14.2.8 Let f and (X, Y) be the flow and vertex bipartition, respectively, generated by procedure *FordFulkerson*. Then for each edge $yx \in E(D)$, where $y \in Y$ and $x \in X$, we have

$$f(yx) = 0.$$

□

Lemmas 14.2.7 and 14.2.8 imply that

$$\begin{aligned} f(X, Y) &= \text{cap}(\Gamma), \\ f(Y, X) &= 0. \end{aligned}$$

Thus, by Formula (14.2.10) we have

$$\text{val}(f) = f(X, Y) - f(Y, X) = \text{cap}(\Gamma).$$

By Corollary 14.2.5, f is a maximum flow and Γ is a minimum cut, which completes the correctness proof of procedure *FordFulkerson*.

The Ford-Fulkerson augmenting semipath algorithm is a general method for computing a maximum flow and minimum cut. In procedure *FordFulkerson*, we did not specify how the path P is generated in the derived network N_f at each stage. In general, there may be many such paths, and the efficiency of *Ford-Fulkerson* is dependent on which augmenting semipath S is chosen at each stage.

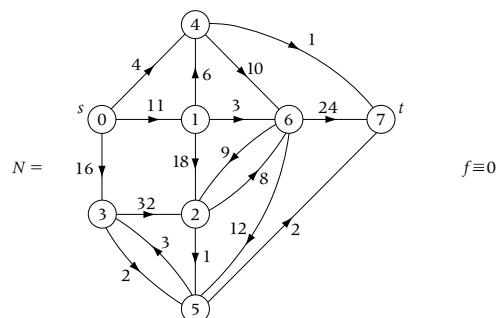
For a poor choice of augmenting semipaths S , procedure *FordFulkerson* may never terminate. However, in the case when the capacities on the edges are all integers, procedure *FordFulkerson* terminates after having performed at most $\text{val}(f)$ iterations of the **while** loop. Because each iteration can be performed in time $O(m)$, the worst-case complexity $W(n, m)$ of procedure *FordFulkerson* belongs to $O(m*\text{val}(f))$. Since $\text{val}(f)$ depends on the capacities of the edges, it can be arbitrarily large. It is not hard to find examples showing that for a poor choice of augmenting semipaths S , $W(n, m)$ can also be arbitrarily large.

Edmonds and Karp showed that a good choice for the augmenting semipath S at each stage of procedure *FordFulkerson* is the shortest one (with a minimum number of edges) over all such semipaths. At each stage, a shortest augmenting semipath S can be found by performing a breadth-first search of the f -derived network N_f to find a shortest path P from s to t . The Edmonds-Karp algorithm has worst-case complexity $W(n, m) \in O(nm^2)$. (The proof of this complexity result is beyond the scope of this book.) The Edmonds-Karp algorithm is illustrated in Figure 14.9 for a sample flow network N having eight vertices and 17 edges. The shortest path generated at each step is indicated with a dotted (as opposed to solid) line.

FIGURE 14.9

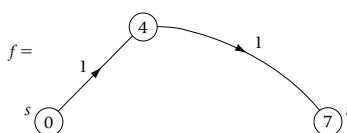
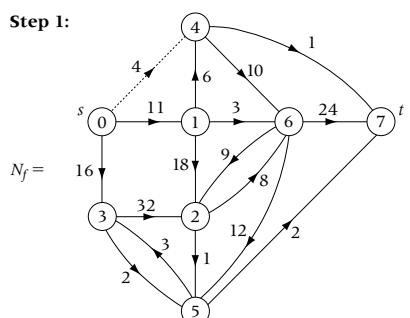
Action of the Edmonds-Karp algorithm for a sample capacitated network N .

Original flow network N with capacities c , and initial flow $f \equiv 0$:

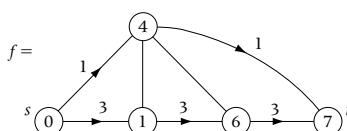
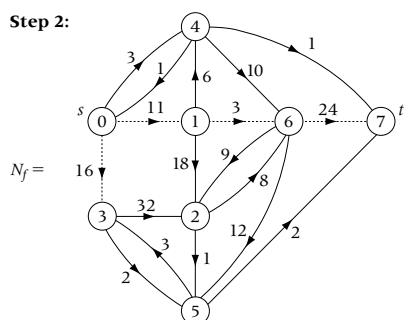


$$f \equiv 0$$

Step 1:



Step 2:



Step 3:

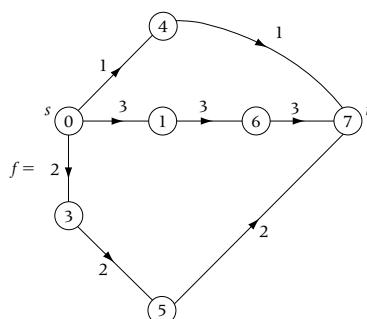
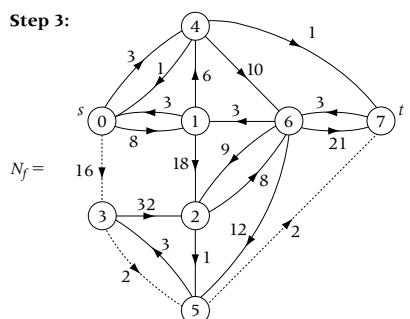
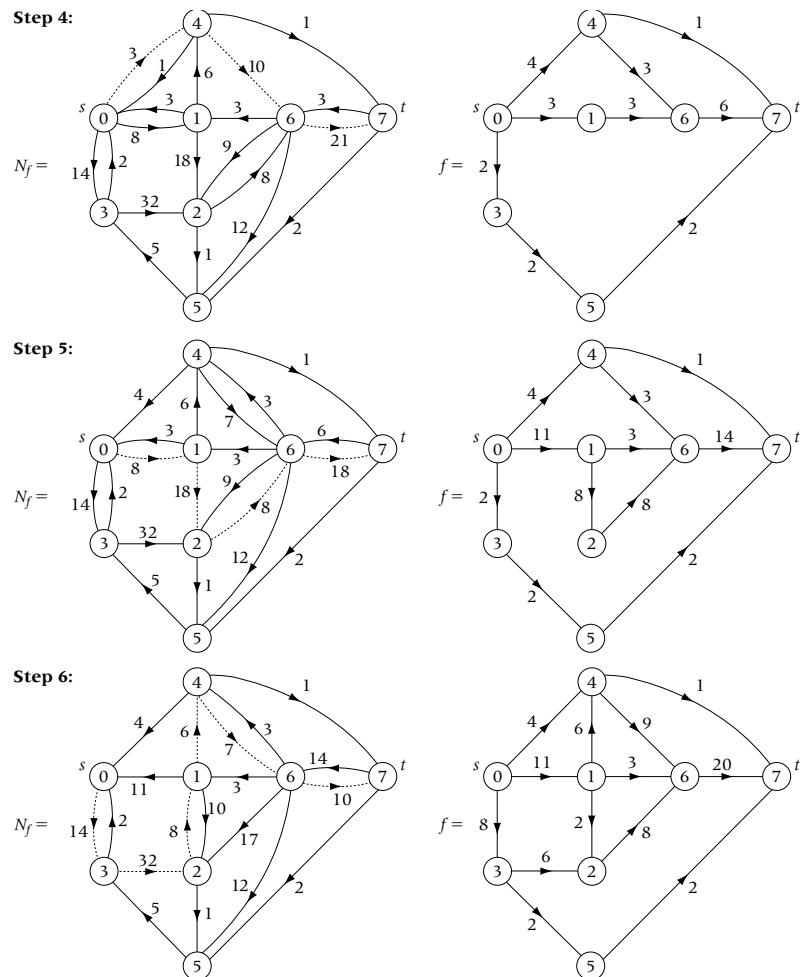
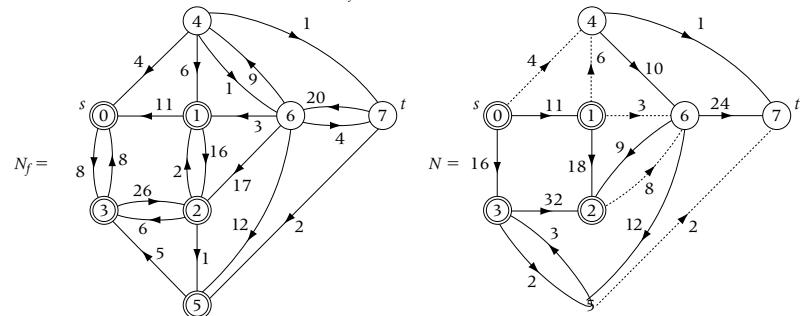


FIGURE 14.9
Continued



There are no more augmenting semipaths. The final flow f has value 23.

Step 7: Compute the f -derived network N_f and minimum cut $\text{cut}(X, Y)$.



The set $X = \{1, 2, 3, 4, 6\}$ of vertices that are accessible in N_f from the source s (marked with \bigcirc) and the set $Y = \{5, 7, 8\}$ of vertices that are not accessible from s determine a cut $\Gamma = \text{cut}(X, Y)$ of capacity $c(X, Y) = 4 + 6 + 3 + 8 + 2 = 23$. Hence, we have $\text{val}(f) = 23 = \text{cap}(\Gamma)$, so that f is a maximum flow Γ and is a minimum cut.

14.2.6 Maximum Flows in 0/1 Networks: Menger's Theorem

An *integer flow* in a digraph $D = (V, E)$ is a flow f such that $f(e)$ is an integer for every edge $e \in E$.

Proposition 14.2.9

Let $D = (V, E)$ be a digraph, and let f be any integer flow from s to t , where $s, t \in V$. Then there exists a set \mathcal{P} of simple paths from s to t and positive integers $\{\lambda_p \mid P \in \mathcal{P}\}$ such that the number of paths containing edge e is at most $c(e)$ and

$$\text{val}(f) = \sum_{P \in \mathcal{P}} \lambda_p. \quad (14.2.11) \square$$

We leave the proof of Proposition 14.2.9 as an exercise. A set of paths \mathcal{P} and associated positive integers $\{\lambda_p \mid P \in \mathcal{P}\}$ satisfying Formula (14.2.11) can be computed in time $O(m * \text{val}(f))$.

A *0/1 flow* is an integer flow such that, for each edge e , $f(e)$ is either 0 or 1. The following result is an immediate corollary of Proposition 14.2.9.

Corollary 14.2.10

Let f be any 0/1 flow from u to v . Then there exists a set \mathcal{P} of pairwise edge-disjoint paths from u to v such that

$$\text{val}(f) = |\mathcal{P}|. \quad \square$$

When the capacities are all integers, $c_f(S)$ is an integer at each stage of the procedure *FordFulkerson*. Thus, for integer capacities, *FordFulkerson* generates a maximum integer flow f . In particular, if each edge has unit capacity, then a maximum 0/1 flow f is generated. By Corollary 14.2.10, if f is a 0/1 flow, then there exists a set \mathcal{P} of pairwise edge-disjoint paths from u to v such that $\text{val}(f) = |\mathcal{P}|$. If \mathcal{P} is a set of pairwise edge-disjoint paths from u to v , then $\sum_{P \in \mathcal{P}} \chi_P$ is a 0/1 flow. Thus, a set \mathcal{P} of pairwise edge-disjoint paths from s to t has maximum cardinality over all such sets if and only if $\sum_{P \in \mathcal{P}} \chi_P$ is a maximum flow.

It follows that we can compute a maximum size set \mathcal{P} of pairwise edge-disjoint paths from s to t in a given digraph D by first applying procedure *FordFulkerson* to the capacitated network $N = (D, c, s, t)$, where each edge e has

capacity $c(e) = 1$ to obtain a maximum flow f , and then computing a set \mathcal{P} of pairwise edge-disjoint paths from s to t such that $\text{val}(f) = |\mathcal{P}|$. Now consider the cut Γ generated by *FordFulkerson*. Because every edge has unit capacity, the capacity of Γ equals the size of (number of edges in) Γ . Since the capacity of Γ equals the value of f , it follows that the size of Γ equals the size of \mathcal{P} , which yields the classical theorem of Menger.

Theorem 14.2.11 Menger's Theorem for Digraphs

Let $D = (V, E)$ be a digraph. Then for $s, t \in V$, the maximum size of a set \mathcal{P} of pairwise edge-disjoint paths from s to t equals the minimum size of a cut Γ separating s and t . \square

The following corollary is the analog of Theorem 14.2.11 for graphs.

Corollary 14.2.12

Menger's Theorem for Undirected Graphs

Let $G = (V, E)$ be an undirected graph. Then for $s, t \in V$, the maximum size of a set \mathcal{P} of pairwise edge-disjoint paths from s to t equals the minimum size of a cut Γ separating s and t . \square

14.2.7 Maximum Size Matching

Procedure *FordFulkerson* can be applied to obtain a maximum size matching in a bipartite graph G . Let (X, Y) denote the associated bipartition of the vertex set V of G . Construct a digraph D as follows. The vertex set of D consists of the vertex set of G together with two new vertices s and t ; that is,

$$V(D) = V(G) \cup \{s, t\}.$$

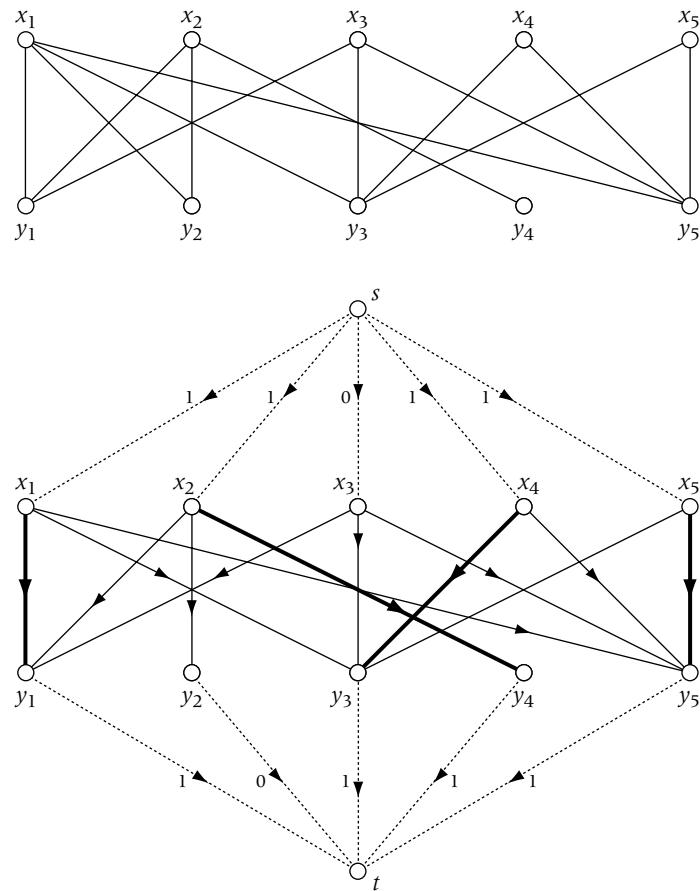
The edge set $E(D)$ of D consists of pairs xy such that $xy \in E(G)$, $x \in X$, and $y \in Y$, together with all pairs sx , $x \in X$, and all pairs yt , $y \in Y$; that is,

$$E(D) = \{xy \mid xy \in E(G), x \in X, y \in Y\} \cup \{sx \mid x \in X\} \cup \{yt \mid y \in Y\}.$$

Now consider the capacitated network $N = (D, c, s, t)$, where every edge e has capacity $c(e) = 1$. For f , a 0/1 flow from s to t , let $\mu(f)$ denote the set of all edges xy of G such that xy is an edge of D and $f(xy) = 1$. Clearly, $\mu(f)$ is a matching M in G , and $\text{val}(f)$ is equal to the size of M . Conversely, given any matching M of G there is a flow f in N such that $M = \mu(f)$. Let f^* be the flow generated by procedure *FordFulkerson*. Since f^* is a maximum-value flow in N , $\mu(f^*)$ is a maximum-size matching in G . The algorithm just described for finding a maximum-size matching is illustrated in Figure 14.10.

FIGURE 14.10

A maximum flow in an associated network where all edges have capacity 1 yielding a maximum matching in a bipartite graph G .





14.3 Closing Remarks

In addition to the Edmonds-Karp algorithm for finding a maximum flow, which uses the Ford-Fulkerson augmenting-path method, other efficient algorithms for finding maximum flows have been designed. These algorithms include the Dinic maximum flow algorithm and, more recently, the preflow push algorithm designed by Goldberg and Tarjan. The problem of finding a maximum flow from a single source s to a single sink t generalizes to the problem of finding a multi-commodity flow from a set of sources to a set of sinks. The theory of multicommodity flows is an active research area, with applications to such areas as routing in communication networks and VLSI layout.

In this chapter, we discussed the Hungarian algorithm for finding a perfect matching in bipartite graph and the Kuhn-Munkres algorithm for finding a maximum-weight perfect matching in a weighted bipartite graph. We also showed how a maximum flow can be used to find a maximum-size matching in a bipartite graph. Algorithms for finding maximum flows and perfect matchings in general graphs can be found in the references.

In Chapter 24, we will present a parallel probabilistic algorithm for determining whether or not a bipartite graph contains a perfect matching.

References and Suggestions for Further Reading

Good references for network optimization algorithms, including flow and matching algorithms:

Ahuja, R. K. *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

Gordon, M., and M. Minoux. *Graphs and Algorithms* (trans. by S. Vajda). New York: Wiley, 1984.

Lawler, E. L. *Combinatorial Optimization: Networks and Matroids*. New York: Holt, Rinehart and Winston, 1976.

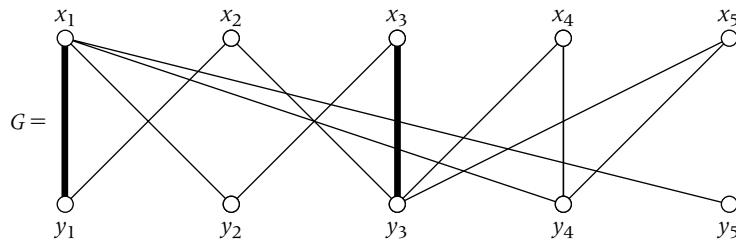
Papadimitriou, C. H., and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, NJ: Prentice-Hall, 1982.

Tarjan, R. E. *Data Structures and Network Algorithms*. Philadelphia: Society for Industrial and Applied Mathematics, 1983.

Lovász, L., and M. D. Plummer. *Matching Theory*. Amsterdam: North Holland, 1986. A nice reference to results and algorithms on matchings.

EXERCISES**Section 14.1** Perfect Matchings in Bipartite Graphs

- 14.1 Prove Lemma 14.1.2.
- 14.2 Prove Lemma 14.1.3.
- 14.3 Prove Lemma 14.1.4.
- 14.4 Show that at each stage of the procedure *Hungarian*, the alternating tree T can be grown in time $O(n^2)$, so the worst-case complexity of procedure *Hungarian* is $O(n^3)$.
- 14.5 Refine the pseudocode of the procedure *Hungarian* to include details for finding the augmenting paths.
- 14.6 Consider the following bipartite graph G :

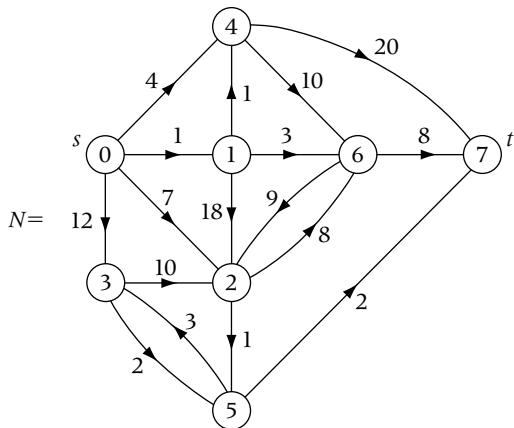


Starting with the given matching $M = \{x_1y_1, x_3y_3\}$, determine the perfect matching output by *Hungarian*. Show each augmenting path that is generated.

- 14.7 Prove Proposition 14.1.5.
- 14.8 Verify that the vertex weighting given by Formula (14.1.8) is a feasible vertex weighting.
- 14.9
 - a. Verify that the vertex weighting ϕ' given by Formula (14.1.11) is a feasible vertex weighting whose equality subgraph $G_{\phi'}$ contains the M -alternating tree T .
 - b. Show that $G_{\phi'}$ contains at least one edge $\{x, y\}$, where $x \in S$ and $y \in Y - \Gamma_{\phi'}(S)$.
- 14.10 Show that the worst-case complexity of procedure *KuhnMunkres* is $O(n^3)$.

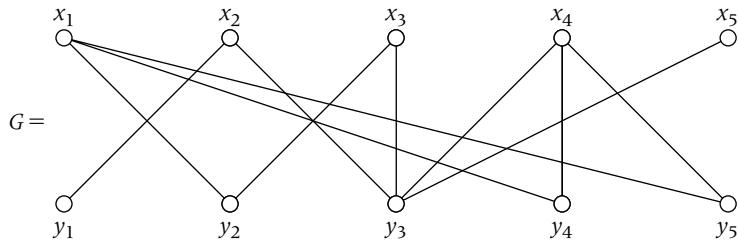
Section 14.2 Maximum Flows in Capacitated Networks

- 14.11 Let f be any flow from vertex s to vertex t in a digraph. Using Proposition 14.2.1, show that the flow out of s equals the flow into t .
- 14.12 Prove Proposition 14.2.2.
- 14.13 Verify that χ_S given by Formula (14.2.3) is a unit flow.
- 14.14 Verify that the final flow $f = \lambda_1\chi_{P_1} + \dots + \lambda_k\chi_{P_k}$ generated by the naive algorithm illustrated in Figure 14.5 is maximal in the sense that if g is any other flow, then $g(e) < f(e)$ for some edge e .
- 14.15 Prove Lemma 14.2.7.
- 14.16 Prove Lemma 14.2.8.
- 14.17 Show that for a poor choice of augmenting semipaths S , procedure *FordFulkerson* may never terminate.
- 14.18 Assuming that the capacities of the network are positive integers, show that for a poor choice of augmenting paths S , the worst-case complexity $W(n, m)$ of procedure *FordFulkerson* can also be arbitrarily large.
- 14.19 Using the Edmonds-Karp algorithm, find a maximum flow f and a minimum cut Γ in the following capacitated network N :



- 14.20 Design and analyze an algorithm for computing a set of paths P and associated positive integers $\{\lambda_P : P \in P\}$ satisfying (14.2.11).
- 14.21 Derive Menger's theorem for undirected graphs (Corollary 14.2.12) from Menger's theorem for digraphs (Theorem 14.2.11).

- 14.22 Using the Edmonds-Karp algorithm, find a maximum matching in the following bipartite graph G by computing a maximum flow in the associated capacitated network.



- 14.23 Derive Konig's theorem, which states that the size of a maximum matching in a bipartite graph G equals the size of a minimum cover (a *cover* is a set of vertices such that every edge of G is incident with at least one vertex in the cover) from Menger's theorem for digraphs (Theorem 14.2.11).



INTERNET ALGORITHMS

The advent of the Internet and the World Wide Web has opened up a vast new frontier of information that is readily available to anyone with a computer and an Internet connection. However, for the Internet to revolutionize the way in which information is gathered and disseminated, it is essential that relevant information be made easily accessible. Search engines have played a central role in facilitating the Internet revolution by allowing users to efficiently locate relevant information on the Web for a given query (a set of relevant keywords). The speed at which commercial search engines (AltaVista, Google, Lycos, and Yahoo, among others) present a list of Web pages relevant to a given query is indeed impressive. In this chapter, we discuss search engines and algorithms for Web page ranking. Another fundamental Internet problem is the efficient delivery of content. In this chapter, we discuss the solution to this problem using Web caching and consistent hashing. We also discuss hashing in general, which is an important data structure used in many Web algorithms. Security is becoming increasingly important in sharing documents and message passing on the Web. We discuss a commonly used algorithm for encrypting messages called the RSA cryptosystem.



17.1 Search Engines

A search engine builds a database of Web documents, which it uses to answer queries. This database is created by using a *spider* (also called a *Web crawler* or *software robot*), which traverses and surveys the Web for information. In addition, some search engines use listings sent in by authors to build their databases. The spider starts with a set of initial *Web addresses* or *URLs* (*Uniform Resource Locators*) and uses the hyperlinks of these Web pages to access more Web pages. The hyperlinks of the newly accessed Web pages are in turn used to access more Web pages. This process continues until a desired amount of the Web has been accessed.

Because the database created by the search engine is huge, the information needs to be indexed for efficient retrieval. An index that is used to find relevant Web pages (URLs), given a term (word) or set of terms from a *lexicon* (vocabulary) of terms, is called a *reverse* or *inverted index*. An index that is used to find a term in a Web document is called a *forward index*. When you enter a query in the search engine Web site, the search engine uses the reverse index to retrieve URLs that contain the terms associated with the query. There are several methods for building an index. One of the most effective ways is to build a *hash table*, as discussed in Section 17.3.

In practice, many commonly used terms (keywords) have a vast number of Web pages associated with them. For example there are millions of Web pages that contain the keyword *algorithm*. Thus, a search engine must not only find Web pages for the query but also use relevancy algorithms to associate a *ranking* or *relevancy value* with the Web pages and list the top-ranked Web pages first. In the next section, we discuss two algorithms for ranking Web pages that are based on the hyperlink structure of the Web. These algorithms exploit the fact that human intelligence is involved in the creation of Web pages; therefore, the hyperlink structure can reveal a great deal of useful information about the relevancy of Web documents.



17.2 Ranking Web Pages

The first ranking algorithm we discuss, called *PageRank*, was developed in 1996 at Stanford by Larry Page and Sergey Brin, the cofounders of Google. PageRank assigns a measure of “prestige” or ranking to each Web page, independent of any query. It is defined using a digraph based on the hyperlink structure of the Web called the *Web digraph*. More formally, the Web digraph W is the digraph whose vertex set $V(W)$ consists of all Web pages (or in practice, a large collection of Web pages), and whose edge set $E(W)$ corresponds to the hyperlinks; that is, an edge

is included from page p to page q whenever there is a hyperlink reference (`href`) in page p to page q .

The second algorithm, called *HITS* (Hyperlink Induced Topic Search), was developed by Jon Kleinberg at IBM in the same year. Unlike PageRank, HITS generates a ranking that is based on a particular query given by the user. Based on the query, HITS generates a subdigraph of the Web digraph called a *focused subdigraph*, which it uses to generate two weightings of the pages: an *authority weighting* and a *hub weighting*. These weightings are computed by using the mutual reinforcement relationship between authorities and hubs: A page has a high authority value, and can be identified as an *authoritative page*, if there are many pages with high hub values that link to that page. On the other hand, a page has high hub value, and can be identified as a *hub page*, if it includes many hyperlinks to authoritative pages.

17.2.1 PageRank

PageRank is an objective measure of people's subjective idea of the importance of the hyperlinks. The definition of PageRank has two distinct underlying motivations. The first motivation comes from the ranking academic citation literature, which existed long before the advent of the Web. The number of citations to a given publication gives some indication of the publication's importance and quality. The second motivation comes from performing a random walk on the Web digraph.

The number of Web pages q that include a hyperlink to p almost always gives a meaningful measure of the significance of Web page p . Thus, one possible ranking of the Web page p would be to compute the number of Web pages that belong to the in-neighborhood $N_{\text{in}}(p)$ of p in the Web digraph W —that is, the in-degree of p . However, intuitively, it makes sense to give more weight to Web pages q in $N_{\text{in}}(p)$ that have a high rank, so we define the rank of p to be the sum of the ranks of all q in the in-neighborhood of p . One drawback with this ranking is that it would allow pages q containing many hyperlinks to have too much influence on the ranking. Therefore, in defining the page rank, we divide by the number of hyperlinks that q contains, or equivalently by the out-degree $d_{\text{out}}(q)$ of q . This yields the following formula for assigning a rank $R[p]$ to a page p :

$$R[p] = \sum_{q \in N_{\text{in}}(p)} \frac{R[q]}{d_{\text{out}}(q)}. \quad (17.2.1)$$

The actual formula for PageRank given by Page and Brin incorporates a damping factor into the ranking function R . To simplify the discussion, we ignore the damping factor for now.

534 ■ PART IV: Parallel and Distributed Algorithms

Formula (17.2.1) can be expressed using matrix notation as follows. Let B denote the matrix whose rows and columns are indexed by $V(W)$ such that

$$B[p, q] = \begin{cases} 1/d_{out}(p), & pq \in E(W), \\ 0, & \text{otherwise.} \end{cases} \quad (17.2.2)$$

For convenience, we use R to also denote the column vector whose entry corresponding to p is $R[p]$. Then Formula (17.2.1) becomes

$$R = B^T R, \quad (17.2.3)$$

where B^T denotes the transpose of the matrix B . Thus, vector R is a so-called *stationary point* with respect to the linear transformation given by the matrix B^T .

Points in a vector space that remain stationary up to a scalar multiple with respect to a linear transformation are known as *eigenvectors*. The scalar multiple is called the *eigenvalue*. More precisely, an eigenvalue of a square matrix M over the real numbers is a real number λ such that $MX = \lambda X$ for some nonzero column vector X , called an eigenvector (see Appendix D). Thus, it follows from Formula (17.2.3) that R is an eigenvector for eigenvalue $\lambda = 1$. Further, it can be shown that 1 is the largest eigenvalue of B^T , called the *principal eigenvector* of B^T . There are numerical algorithms, beyond the scope of this book, for computing the principal eigenvector of a matrix. However, since the matrix B is so enormous, its size being the number of pages in the entire World Wide Web (or, in practice, a large portion of the Web), implementing such an algorithm on the matrix B would require a tremendous amount of computation.

We now describe a more practical algorithm that obtains a good approximation to R . This algorithm uses iteration and, starting with an initial vector R_0 , successively generates vectors $R_0, R_1, \dots, R_i, \dots$. The initial vector R_0 , can be taken to be any vector whose entries are positive and sum to 1. We then inductively compute R_i from R_{i-1} using the formula

$$R_i = B^T R_{i-1}, \quad i = 1, 2, \dots. \quad (17.2.4)$$

Letting $R_i[p]$ denote the ranking function associated with R_i (that is, $R_i[p]$ is the value of the component of R_i corresponding to page p), the matrix equation (17.2.4) is equivalent to

$$R_i[p] = \sum_{q \in N_{in}(p)} \frac{R_{i-1}[q]}{d_{out}(q)}, \quad i = 1, 2, \dots. \quad (17.2.5)$$

Using Formula (17.2.4) and iterating i times we obtain the formula

$$R_i = (B^T)^i R_0. \quad (17.2.6)$$

The vector R_i will not necessarily converge to the principal eigenvector R unless the digraph W satisfies certain conditions. For example, it would need to be strongly-connected; otherwise, for some choices of R_0 , $R_i[p]$ would equal 0 for all i , whereas for other choices, it would be nonzero for all i . In particular, suppose $R_0[p] = 1$ if $p = q$, and $R_0[p] = 0$ otherwise. Then if there is no path from q to p , we have $R_i[p] = 0$ for all i , whereas if there is a path, then $R_i[p]$ is nonzero for some i (exercise). However, it is not sufficient that W is strongly connected. We must assume the stronger condition that W is also *aperiodic*; that is, for each node p , there is a closed walk of length (number of edges) i containing p for all but a finite number of integers i , where a *walk* is defined similarly to a path, but where we allow both vertices and edges to be repeated. We state the following key fact whose proof is beyond the scope of this book.

Key Fact

If the digraph W is strongly connected and aperiodic, then R_i will converge to the principal eigenvector R of B^T as i goes to infinity. Further, this convergence is independent of the choice of the initial vector R_0 whose entries are positive and sum to 1. Because we wish to find the ordering of pages determined by R rather than its actual value, it is enough to iterate sufficiently often (approximately 50 iterations usually is enough, even for millions of pages).

The PageRank $R[p]$ has an interesting interpretation in terms of random walks. Note that the sum of the entries in row p of B equals 1. Thus, B can be thought of as the matrix for a random walk (see Appendix E) on the digraph W , where $B[p][q]$ is the probability that a random walker at page p (or in our case, an “aimless surfer” of the Web) will follow the hyperlink from page p to page q . It is not difficult to show that the i^{th} power matrix of the matrix B has the property that its pq^{th} entry $B^i[p][q]$ is the probability that an aimless surfer starting at page p reaches q in i steps by following a path of i hyperlinks (see Exercise 17.5). Equivalently, $(B^T)^i[p][q]$ is the probability that an aimless surfer starting at page q will reach p after i steps.

Because the entries of R_0 are positive and sum to 1, R_0 determines a probability distribution on the set of pages $V(W)$, where $R_0[q]$ is the probability that the aimless surfer begins surfing from page q . It follows from the previous result that $R_i = (B^T)^i R_0$ is then the probability that the surfer will end up at page p after i steps. In particular, choosing the probability distribution given by $R_0[q] = 1$ for a

536 ■ PART IV: Parallel and Distributed Algorithms

particular page q and $R_0[p] = 0$ for every other page p , $R_i[p]$ becomes the probability that the aimless surfer starting at the given page q will end up at p after i steps. As discussed earlier, if W is strongly connected and aperiodic, then the limit of R_i converges to R as i goes to infinity, independent of the choice of R_0 (The concept of aperiodic is defined more generally in a Markov chain. See Appendix D for more discussion on random walks and Markov chains.) It follows that we are free to choose any page q as the initial starting point for our aimless surfer, as long as we iterate enough times.

The aimless surfer interpretation gives us further intuition that PageRank is a meaningful ranking of Web pages. When randomly surfing the Web, we have a higher probability of ending up at a more significant page than a less significant one because more hyperlinks lead to significant pages.

To simplify our discussion of PageRank, we ignored the damping factor. However, in practice, to obtain better ranking results, a damping factor d , which is a value between 0 and 1, is included in the definition of PageRank.

DEFINITION 17.2.1 PageRank Let n denote the number of nodes of the Web digraph W . The PageRank $R[p]$ of a Web page p is given by

$$R[p] = (1 - d)/n + d \sum_{q \in N_{in}(p)} \frac{R[q]}{d_{out}(q)}, \quad (17.2.7)$$

where d is the damping factor. Taking the damping factor d to be between 0.8 and 0.9 has been found to work well in practice.

Formula (17.2.1) is the special case of Formula (17.2.7), where the damping factor d is 1. The more general Formula (17.2.7) also has an interpretation in terms of an aimless surfer. The difference is that now the surfer at page q can either follow a hyperlink from page q to page p as before (with probability d/d_{out}), when such a hyperlink exists, or jump to a random page p with small probability [with probability $(1 - d)/n$]. Because the Web digraph is not strongly connected, this jumping allows the aimless surfer to potentially reach pages of the Web that he or she could not reach by simply following hyperlinks. Thus, the underlying graph becomes complete (and aperiodic). Similar to the algorithmic solution we discussed earlier for the case $d = 1$, the formula for PageRank given in Definition 17.2.1 can be solved using iteration. As before, the initial ranking R_0 can be any nonzero vector whose entries sum to 1. Based on Formula (17.2.7), we obtain the following iteration formula:

$$R_i[p] = d/n + (1 - d) \sum_{q \in N_{in}(p)} \frac{R_{i-1}[q]}{d_{out}(q)}, i = 1, 2, \dots \quad (17.2.8)$$

R E M A R K S

1. Page and Brin actually published two different versions of PageRank. In their original version, they defined PageRank by

$$R[p] = d + (1 - d) \sum_{q \in N_{in}(p)} \frac{R[q]}{d_{out}(q)}.$$

Although the two versions do not differ fundamentally, the version given by Formula (17.2.8) has the interesting idle surfer interpretation we just described.

2. Although PageRank is the centerpiece of Google's ranking mechanism, other criteria are also used, such as keywords, phrase matches, match proximity, and so forth. In fact, the rank of a page in the commercial version of PageRank used by Google can be increased by paying an appropriate fee.

Again, because we are more interested in the ranking (order) induced by the ranking function R than its exact value, we can stop iterating when R_i is close enough to R to induce the same ranking or close to the same ranking as R . For example, empirical experiments have shown that acceptable ranking functions R_i are achieved in 52 iterations for about 322 million hyperlinks.

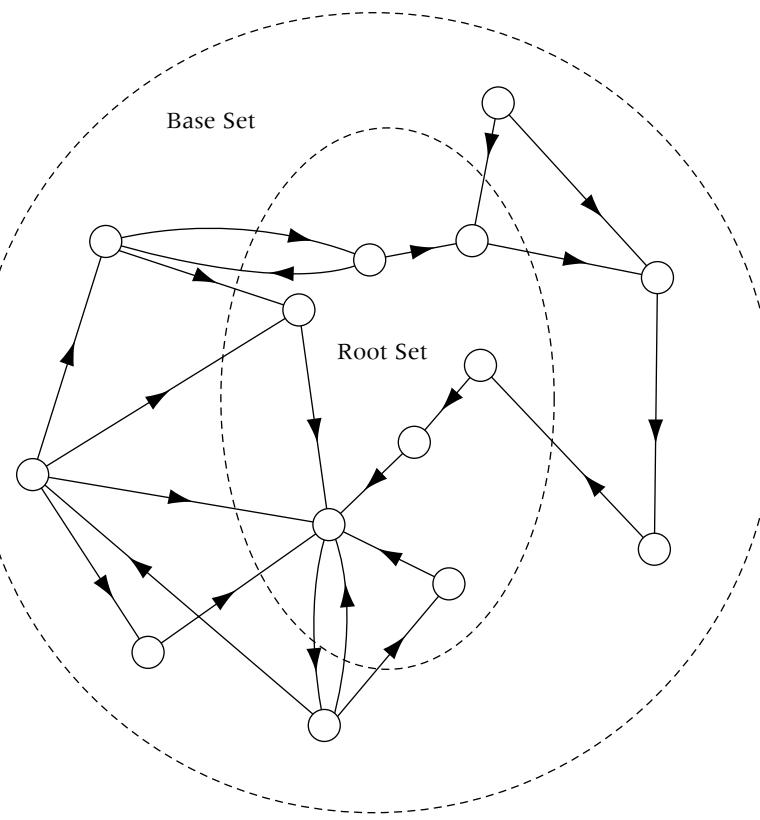
Computing the page rank is a good way to prioritize the results of Web keyword searches. It has been shown that for most popular subjects, a simple text-matching search that is restricted to Web page titles performs well when PageRank is used to rank the results. However, the computation of PageRank is independent of any particular query string Q . We now discuss the algorithm called *HITS*, which uses query string information to obtain an improved ranking.

17.2.2 *HITS* Algorithm

In the first phase of *HITS*, given a query Q , the algorithm computes a root set A consisting of the top-ranked Web pages for Q based on a “first approximation” ranking. This ranking can be obtained using textual information such as the frequency of occurrence of the query string on the page, whether the query string occurs in bold or in a large font size, PageRank, and so forth. The number of such Web pages chosen is based on a predetermined number r —say, $r = 200$. *HITS* then augments the root set A to obtain a base set B as follows. For each page p in the root set A , all the pages in its out-neighborhood in W and all the pages in its in-neighborhood are added to A . Because, in practice, the in-neighborhood could be huge, *HITS* limits the number of pages it chooses to some reasonable number of pages chosen in an arbitrarily fashion from the in-neighborhood of p . Also, we may wish to delete pages from popular sites, such as <http://www.yahoo.com> or <http://www.google.com>, which would tend to be the most

FIGURE 17.1

Expanding the root set into a base set.



authoritative pages with respect to any query string they contain. The root set A and its expansion to a base set B is illustrated in Figure 17.1. After computing the base set B , HITS computes the induced subdigraph $W[B]$. However, hyperlinks (edges) between pages within the same host are considered navigational or nepotistic, rather than informational, so we remove them from $W[B]$ to obtain a subdigraph that we call the *focused subdigraph for query Q*, and denote by F_Q .

Note that the focused subdigraph F_Q may include authoritative pages for the query Q that do not contain even a single occurrence of the query string. For example, suppose hypothetically that Albert Einstein were still alive today and had a Web page. Although Einstein's Web page may not contain any occurrences of the string "genius," it is likely that at least one of the Web pages in the root set for the query "genius" would contain a hyperlink to Einstein's Web page. Thus, Einstein's Web page would be brought into the base set B and the focused subdigraph F_Q .

We define the hub value $h[p]$ of a page p to be the sum of the authority values over all pages q in its out-neighborhood $N_{\text{out}}(p)$ of the focused subdigraph F_Q and the authority value to be the sum of the hub values over all pages q in its in-neighborhood $N_{\text{in}}(p)$ of F_Q .

$$h[p] = \sum_{q \in N_{\text{out}}(p)} a[q], \quad (17.2.9)$$

$$a[p] = \sum_{q \in N_{\text{in}}(p)} h[q]. \quad (17.2.10)$$

Letting A denote the adjacency matrix of F_Q , and letting h and a denote the column vectors whose entries corresponding to page (vertex) p are $h[p]$ and $a[p]$, respectively, Formulas (17.2.9 and 17.2.10) become

$$a = A^T h, \quad (17.2.11)$$

$$h = A a. \quad (17.2.12)$$

Combining Formulas (17.2.11) and (17.2.12), we obtain

$$a = A^T h = A^T (A a) = A^T A a, \quad (17.2.13)$$

$$h = A a = A (A^T h) = A A^T h. \quad (17.2.14)$$

We normalize the vectors a and h by dividing by $\|a\| = \sum_{p \in V(F_Q)} a[p]$ and $\|h\| = \sum_{p \in V(F_Q)} h[p]$, respectively, so that $\|a\| = \|h\| = 1$.

We call $A^T A$ the *authority matrix* and $A A^T$ the *hub matrix*. As with PageRank, a and h are the principal eigenvectors of the matrices $A^T A$ and $A A^T$, respectively, and we can iterate to obtain approximations a_i and h_i to a and h , respectively. The only difference is that we normalize a_i and h_i after each iteration. Initially, we choose a_0 and h_0 to both be the column vector $(1, 1, \dots, 1)^T$.

The following is a high-level description of the HITS algorithm.

→.....

```

procedure HITS(Q, k, a, h)
Input: Q (query string), k (number of iterations to be performed)
Output: a (authority ranking function), h (hub ranking function)
        Compute the focused subdigraph  $F_Q$  for the query  $Q$ 
        Compute the adjacency matrix  $A$  of  $F_Q$ 
         $a \leftarrow h \leftarrow (1, \dots, 1)^T$ 
        for  $i \leftarrow 1$  to  $k$  do

```

```

    ...
    h ← Aa
    Compute  $\|h\| \leftarrow \sum_{p \in V(F_q)} h[p]$ 
    h ← h /  $\|h\|$ 
    a ←  $A^T h$ 
    Compute  $\|a\| \leftarrow \sum_{p \in V(F_q)} a[p]$ 
    a ← a /  $\|a\|$ 
endfor
end HITS
.....

```



17.3 Hashing

To efficiently retrieve information from a database, such as the huge database of Web documents created by a search engine, it is useful to build an index. One of the most effective ways of building an index is to use a *hash function* and a *hash table* implemented using an array. The hash function maps the keys to some reasonably large interval of integral indices $\{0, 1, \dots, n - 1\}$. Good choices of hash functions will avoid excessive “collisions” among the keys when they are mapped to the corresponding array $HashTable[0:n - 1]$. Operations performed on the hash table include inserting, deleting, and searching for keys. These operations are sometimes called *dictionary* operations. Other ADTs for implementing dictionary operations, such as balanced trees and B-trees, are given in Chapter 21.

Clearly, if we use an array for storing n keys whose index range is as large as the size of the ordered set S consisting of all possible keys, then we can reserve a unique position in the array for each key. In the so-called *direct-addressing scheme*, the ordered set S itself is used as the indexing for the array. For example, if S consists of integers between 0 and 99, and the actual keys in current use are 7, 23, 55, 61, and 80, then an array $DirAddr[0:99]$ supporting direct-addressing would have positions (slots) 7, 23, 55, 61, and 80 occupied with key values, and the remaining 95 slots would be unoccupied. If the keys are used to index records in a database file, then it would be assumed that the element in the i^{th} slot of $DirAddr[0:99]$, $i = 0, \dots, 99$, contains the address of the record having the key i in this file.

In a direct-addressing scheme, all the dictionary operations (inserting, deleting, and searching for a key) can be performed in constant time. Unfortunately, the direct-addressing scheme is not often feasible in practice because the size of S is simply too large. For example, suppose we have at most 10,000 records corresponding to students at UHash College, where the key is a student’s 9-digit Social Security number. We would need to maintain an array of size 999,999,999

to have enough positions for every possible Social Security number, even though at most 10,000 of these positions would actually be used.

The idea behind hashing is to define an appropriate mapping h from the set S of all possible keys to a much smaller set S' consisting of consecutive integers. For example, in our UHash College example, suppose h maps each Social Security number s onto the number determined by the last four digits of s . Then h maps the set S of all Social Security numbers onto the much smaller set S' of integers between 0 and 9999, inclusive. The associated hash table can then be an array $H[0:9999]$, and a nine-digit Social Security number s would be stored in position $h(s)$ of the array H .

Although hashing has the advantage of saving memory, it has the drawback that multiple keys in S are mapped by h to the same integer in S' . For example, two or more students from UHash College may have Social Security numbers with the same last four digits. After the first student has been inserted in the hash table, the attempted insertion of a second student whose Social Security number has the same last four digits results in a *collision*. In the next two subsections, we discuss the two standard methods used to resolve collisions, *chaining* and *open addressing*. Of course, it is best to avoid collisions as much as possible by a good choice of hash function. In our UHash College example, the choice of hash function was not very good because it completely ignored the first five digits of the Social Security number. It would be better to use a hash function h that depends on all the digits of s , because there might be some bias present in the last four digits.

The word *hashing* derives from the fact that $h(s)$ is often obtained, as in our UHash College example, by chopping off or otherwise altering some aspect of the key value. Also, good hash functions should mix things up pretty well, thereby keeping collisions to a minimum.

17.3.1 Hash Functions

As just mentioned, a good hash function is one that minimizes the number of collisions. Of course, a good hash function should also be easily computable. The number of collisions is minimized if each key k from S is equally likely to be mapped by the hash function h to any given index (*slot*) in the hash table. In other words, if there are m slots $0, \dots, m - 1$ in the hash table, and we independently draw keys from S according to a given probability distribution P , then the number of collisions is minimized if

$$P(h(k) = i) = \frac{1}{m}, \quad i = 0, 1, \dots, m - 1. \quad (17.3.1)$$

542 ■ PART IV: Parallel and Distributed Algorithms

A hash function satisfying Formula (17.3.1) is termed a *uniform hash function*, and such a hash function together with its associated hash table is called *uniform hashing*. Unfortunately, in most instances, the probability distribution P on S is not known. Even when P is known, uniform hash functions are difficult to obtain. However, heuristic techniques can be used to find hash functions that come close to satisfying the assumption of uniform hashing and therefore perform quite well in practice.

To simplify our discussion, from now on we assume that the keys are drawn from the set of integers. There is usually a natural way in which to satisfy this assumption in practice. For example, keys made up of alphabetical characters could be replaced by their ASCII (decimal) equivalents. Given integer keys, two standard hashing techniques are the division method and the multiplication method.

In the *division method*, h maps k onto $k \bmod m$, where m is the size of the hash table:

$$h(k) = k \bmod m. \quad (17.3.2)$$

We must be careful in the choice of m so as not to bias the hash function. For example, if m is divisible by two, then h would have the bias that even (odd) integers would be hashed to even (odd) positions in the hash table (goodbye uniform hashing). Similar biases would exist if m has other nontrivial factors. For instance, considering again our UHash College example, note that the hash function was precisely Formula (17.3.2) with $m = 10^4$. To avoid these types of biases, a good choice of m is a prime not too close to an exact integer power.

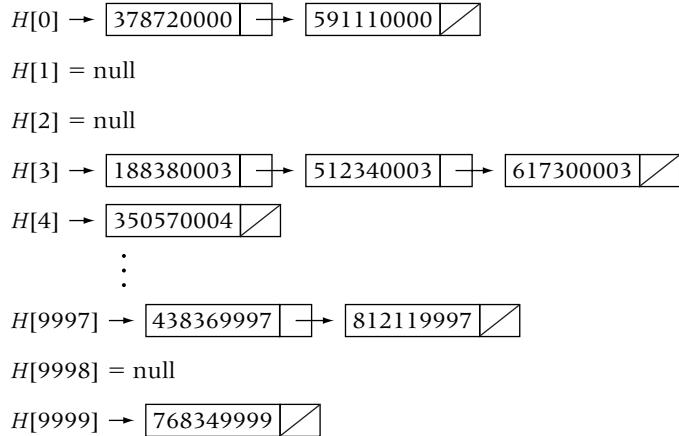
In the *multiplication method*, we map the keys to slots in $H[0:m - 1]$ by the formula

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor, \quad (17.3.3)$$

where A is a constant such that $0 < A < 1$. The term $(kA - \lfloor kA \rfloor)$ in Formula (17.3.3) is merely the fractional part of kA . While the method works for any choice of the constant A , for a particular set of keys, some choices of A will tend to yield a more uniform behavior for h than others. It turns out that the choice of A given by the inverse of the golden ratio $\phi^{-1} = (\sqrt{5} - 1)/2$ works well in many situations because of the especially uniform way that the points $0, \phi^{-1} - \lfloor \phi^{-1} \rfloor, 2\phi^{-1} - \lfloor 2\phi^{-1} \rfloor, 3\phi^{-1} - \lfloor 3\phi^{-1} \rfloor, \dots$ divide up the unit interval $[0, 1]$.

The choice of m in Formula (17.3.3) is guided, of course, by such things as the anticipated size of the set of keys to be stored in the hash table. Fortunately, the choice of m is not critical to the good behavior of the hashing function given by Formula (17.3.3). Thus, m is typically taken to be a power of two to make the implementation of $h(k)$ easier on most computers.

FIGURE 17.2
A chained hash function for student's social security numbers at UHash College, where the hash function h maps a social security number onto the number determined by its last four digits



17.3.2 Collision Resolution by Chaining

In the method of collision resolution by *chaining*, the i^{th} entry in the hash table $H[0:m - 1]$, $i = 0, \dots, m - 1$, does not contain a key but instead contains a pointer to a linked list of keys, all of which satisfy $h(k) = i$. We illustrate collision resolution by chaining in Figure 17.2 for our UHash College example. Recall that the keys are nine-digit Social Security numbers, and $h(k)$ is the integer represented by the last four digits.

When searching for a given key, we first hash the key to the slot that points to the appropriate linked list (*chain*), and then we perform a search of this chain. When inserting a key, we hash the key to the appropriate slot and then insert the key in the corresponding linked list. Deletion of a key also amounts to simple deletion in the appropriate linked list.

Searching, insertion, and deletion each have worst-case behavior bounded by the longest chain pointed to by the hash table. Of course, in the worst case, all the keys hash to the same list, so that the dictionary operations are basically identical to that performed on an ordinary linked list (with the additional overhead of computing the hash function). Thus, the dictionary operations for hashing with chaining have linear worst-case complexity, which is certainly unattractive. However, as we will see, the average behavior of hashing with chaining is much better.

To discuss the average behavior of hashing with chaining, we need to make some assumptions and provide some notation. We focus on searching, since insertion and deletion have the same complexity as searching. Given a hash table

544 ■ PART IV: Parallel and Distributed Algorithms

$H[0:m - 1]$ of m slots containing n keys, we measure the average complexity of searching a hash table with chaining in terms of the *load factor* $\alpha = n/m$. The ratio α can be thought of as the average number of keys in a chain. However, it is only when our hash function h is close to uniform that each chain holds approximately α keys.

We now analyze the average behavior of both a successful search and an unsuccessful search of a hash table with chaining under the assumption that h is a uniform hash function. Let U_n and S_n denote the average number of key comparisons done in an unsuccessful search and a successful search, respectively. Note that U_n is nothing more than the average length of a chain. Moreover, since h is a uniform hash function, the average length of a chain is given by α . Hence, under our assumptions, the average complexity of unsuccessful search is given by

$$U_n = \alpha = \frac{n}{m}. \quad (17.3.4)$$

We now consider a successful search, where we assume that any key in the hash table is equally likely to be the search element. Under our assumptions of uniform hashing, the expected number of key comparisons made when searching for a given key is one more than that made when the key was inserted into the hash table. Thus, S_n is equal to 1 plus the average number of key comparisons made when inserting the i^{th} key, $i = 1, \dots, n$. Since we have assumed uniform hashing and each key is equally likely, when the i^{th} key was inserted, each chain had average length given by $(i - 1)/m$. Thus, we have

$$\begin{aligned} S_n &= \frac{1}{n} \sum_{i=1}^n \left(1 + \frac{i-1}{m} \right) = 1 + \frac{1}{nm} \sum_{i=1}^n (i-1) \\ &= 1 + \frac{1}{nm} \left(\frac{(n-1)n}{2} \right) \quad (17.3.5) \\ &= 1 + \frac{\alpha}{2} - \frac{1}{2m}. \end{aligned}$$

From Formula (17.3.5), S_n is approximately equal to $1 + \alpha/2$ for large m , which is entirely reasonable since this is the average number of comparisons performed by a linear search on a list of size α .

What happens in the formulas for U_n and S_n as both m and n increase? If the number of slots m in the hash table is proportional to the number of keys n in the table, then $\alpha = n/m \in O(1)$. Then both successful search and unsuccessful search

(and the other dictionary operations as well) can be done in constant time, on average. The dramatic improvement over the linear worst-case behavior is the reason that finding good hash functions is so important.

17.3.3 Collision Resolution with Open Addressing

The second major method of resolving collisions in hash tables is to use the technique of *open addressing*. When a collision is detected upon inserting a key, a search of the hash table is made to find the first available unoccupied slot in the hash table. The key is then inserted at this open position. In the open addressing scheme, we can never store more keys than available slots in the hash table. Thus, in contrast to the chaining scheme, the loading factor $\alpha = n/m$ for hashing with open addressing is never more than 1.

There are various methods of determining the first available open position for inserting a key. Each method generates a certain *probe sequence* that, starting from the slot given by the hash function, visits the other slots in the hash table in some determined order. We require that the probe sequence generate a permutation of the slots, to guarantee that all m slots are eventually reached by unsuccessful searches. We discuss two commonly used probing methods, *linear probing* and *double hashing*.

When inserting a key, we follow the appropriate probe sequence until the first unoccupied slot is found (if any). The key is then inserted at this first available slot. Searching for a key also works by executing the appropriate probe sequence until either the key is found, an unoccupied slot is encountered, or all m slots have been examined without finding the key. Both of the latter cases correspond to unsuccessful searches. The reason that encountering unoccupied slots signals an unsuccessful search is that each probe sequence is completely determined by the key and the specific probing method. In particular, if an unoccupied position is ever encountered when searching for a given key, the key is not in the table because it would have been already found or originally inserted at the unoccupied position.

The previous discussion highlights a problem that arises with open addressing. Namely, deletions confound subsequent searches. How do we tell whether an unoccupied slot was vacated by a previous deletion? Clearly, we must be able to recognize this; otherwise, searches might end prematurely and erroneously signal unsuccessful searches. One way around this dilemma is to keep a bit field around to signal whether or not a slot was ever occupied. While this solves the problem, it tends to generate long searches in the presence of many deletions. Thus, hashing with chaining is preferred over hashing with open addressing when there is the likelihood of a large number of deletions being performed.

546 ■ PART IV: Parallel and Distributed Algorithms

slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
key	23			3	4	5	26	6	7	55						16	39	17				22	

FIGURE 17.3

Contents of $H[0:22]$ after inserting 3, 4, 5, 26, 6, 7, 23, 16, 39, 17, 22, and 55 in initially empty table using the hash function $h(k) = k \bmod 23$ and open addressing with linear probing

Linear Probing. Perhaps the simplest probing method is *linear probing*, where starting from the slot given by the hash function, we perform a linear search of the table. We assume that the indices are stored circularly (that is, modulo m), so that the probe resumes at the beginning of the table after it reaches the end. Thus, for a given k , the linear probing sequence corresponding to k is given by

$$(h(k) + i) \bmod m, \quad i = 0, \dots, m - 1. \quad (17.3.6)$$

In Figure 17.3, we show the contents of a hash table $H[0:22]$ after the insertion of the keys 3, 4, 5, 26, 6, 7, 23, 16, 39, 17, 22, and 55, where $h(k) = k \bmod 23$.

Unfortunately, while linear probing is easy to implement, it does not come very close to uniform hashing because long runs (called *primary clusters*) of occupied slots tend to build up even in relatively sparse tables. The reason for clustering is clear: An empty slot preceded by a nonempty slot is twice as likely to be filled by an insertion than is an empty slot preceded by an empty slot. In fact, if i slots are filled preceding a given open slot, then the empty slot has probability $(i + 1)/m$ of being filled by the next insertion, whereas it has probability of $1/m$ of being filled if its preceding slot is empty. (These calculations assume, as usual, that any slot is equally likely to be the hash value of the inserted key.)

Double Hashing. *Double hashing* gets its name from the fact that it uses two hash functions to generate its probing sequences. Given two hash functions h_1 and h_2 , the double hash function generated by h_1 and h_2 has a probe sequence given by

$$(h_1(k) + ih_2(k)) \bmod m, \quad i = 0, \dots, m - 1, \quad (17.3.7)$$

where it is required that $\gcd(h_2(k), m) = 1$.

slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
key	23																						

FIGURE 17.4

Contents of $H[0:22]$ after inserting 3, 4, 5, 26, 6, 7, 23, 16, 39, 17, 22, and 55 in initially empty table using the double hash function based on the hash functions $h_1(k) = k \bmod 23$ and $h_2(k) = 1 + (k \bmod 21)$ and open addressing

In particular, the double hash function has its initial value given by $h_1(k)$. The condition that $h_2(k)$ and m are relatively prime is needed to guarantee that the probe sequence generates a permutation of $0, \dots, m - 1$ (that is, all slots are reached). To ensure relative primality, we can define $h_2(k) = 1 + (k \bmod m')$, where m' is slightly smaller than m . Figure 17.4 illustrates a double hash function based on $h_1(k) = k \bmod 23$ and $h_2(k) = 1 + (k \bmod 21)$ for the same keys as in Figure 17.3.

Double hashing outperforms linear probing because it does not suffer from the problem of clustering. In practice, double hashing usually yields a rather good approximation to uniform hashing.

Average Behavior of Hashing with Open Addressing. When analyzing the average behavior of hashing with collision resolution by open addressing, a very strong form of uniform hashing is assumed. The strong form takes into account the entire probing sequence generated by the hash function, not just its initial value. A *strongly uniform hash function* h is a hash function with the property that each of the $m!$ permutations of $0, \dots, m - 1$ are equally likely to be the probe sequence generated by h for any key. In particular, a strongly uniform hash function is also a uniform hash function in our previous sense, because its initial value can be any of the slots $0, \dots, m - 1$ with equal probability.

Of course, strongly uniform hash functions are even harder to find than ordinary uniform hash functions. Double hashing is closer to being strongly uniform than linear probing. Linear probing generates a total of only m probe sequences because a linear probe is simply a cyclic shift of the identity permutation. On the other hand, double hashing generates m^2 probe sequences because each pair $(h_1(k), h_2(k))$ yields a distinct probe sequence.

548 ■ PART IV: Parallel and Distributed Algorithms

The following theorem expresses the average behavior of unsuccessful and successful searches for hashing with collision resolution by open addressing. The proof of Theorem 17.3.1 is left to the exercises. Theorem 17.3.1 expresses the average behavior in terms of the loading factor $\alpha = n/m < 1$.

Theorem 17.3.1 Suppose we have an open-address hash table with a loading factor $\alpha = n/m < 1$ and a strongly uniform hashing function. Let U_n and S_n denote the expected number of probes in an unsuccessful and successful search, respectively. Then we have

$$U_n \leq \frac{1}{1 - \alpha}, \quad (17.3.8)$$

and

$$S_n \leq \frac{1}{\alpha} \left(1 + \log_2 \left(\frac{1}{1 - \alpha} \right) \right). \quad (17.3.9) \blacksquare$$

Inserting an element has the same complexity as that for an unsuccessful search. Again, supposing that α is held relatively constant as m and n grow large, then Theorem 17.3.1 implies that both unsuccessful and successful searching can be done in constant time.



17.4 Caching, Content Delivery, and Consistent Hashing

In the first two sections of this chapter, we considered the fundamental problem of searching and retrieving information on the World Wide Web and the ranking of Web pages. In this section, we consider another fundamental problem, the efficient delivery of content on the Internet. We discuss an important strategy for solving this problem called *caching* and an efficient scheme for hashing content to caches called *consistent hashing*.

17.4.1 Web Caching

Web caching involves the storage of content at special servers called *caches*. Replicating information and storing it in caches alleviates a number of problems. First, it reduces network congestion by reducing the amount of redundant traffic that needs to cross the backbone of the Internet. Second, it reduces *latency*—the speed at which packets (data) need to travel to reach users. Third, caching

reduces load and helps eliminate “hot spots”—locations where many users are trying to access data from the same server because content is distributed among multiple caches.

REMARK

Often in practice, the pictures on a Web site are cached but the text is not. The advantage of this is that the original site will have a high hit count, which is usually desired by the content provider.

The most straightforward approach to caching is to have a single large cache to serve each region, such as a city. This is called the *monolithic approach*. All users from a particular region—say, New York—would go to the cache to access content from various content provider sites (URLs) throughout the Internet. When the first client (user) from New York wishes to access content from a particular URL, the content will not be available in the New York cache, and the client will need to go directly to the original site. However, the content will then be cached locally in New York, so that subsequent clients from New York who request the same content can access it directly from the New York cache.

The monolithic approach has some disadvantages. For example, if a regional cache fails, then all the users in that region are cut off. Thus, a large fault-tolerant machine is needed for each region, which is expensive and not very practical to build and maintain. Another disadvantage is that each item must be replicated r times, where r is the number of regions for which at least one user in the region has requested the item. Although, a certain amount of replication is needed if the item is popular, in general, the amount needed for the monolithic approach far exceeds the amount needed simply to avoid hot spots. Thus, caches become congested.

Another approach, called the *distributed approach*, is to place many smaller caches distributed among local neighborhoods to serve the users in the neighborhood. Although this eliminates the need for a large, highly fault tolerant cache, it has the major disadvantage that the hit rate is low because the caches are receiving requests from a smaller population. This means that users have to go to original sites much of the time to obtain content. Clearly, it would be unfeasible to replicate each item in each cache every time there is a miss—that is, every time a user goes to the neighborhood cache and fails to find the item. Also, because individual caches are small, they can only hold a small portion of the data from content providers.

A hybrid approach, called *harvest caching*, combines the monolithic and distributed approaches. This approach involves a hierarchy of caches, where nodes closer to the root serve a larger region, and the leaf nodes correspond to the neighborhood caches. If there is no hit in a neighborhood cache, the parent cache can be searched for the item. This approach still has the disadvantage of the monolithic caching model in that it requires larger fault-tolerant machines for the regional caches.

In the next subsection we discuss a better approach using consistent hashing.

17.4.2 Consistent Hashing

Before discussing consistent hashing, we first consider the problem of using an ordinary hash function to map content to caches. We refer to the content objects to be cached as *items* and the caches as *buckets*. Let $\mathcal{I} = \{I_0, I_1, \dots, I_{t-1}\}$ denote the set of items and $\mathcal{B} = \{B_0, \dots, B_{k-1}\}$ the set of buckets. Suppose we use a linear hash function $h(i) = ai + b \pmod{k}$, where a and b are randomly chosen integers and i is a unique identifying integer associated with item $I \in \mathcal{I}$. Because a and b were chosen at random, we can expect that the load will be fairly evenly distributed among the buckets.

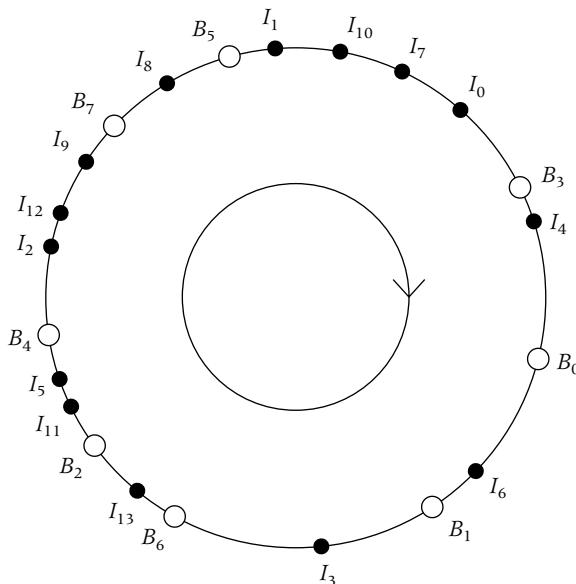
Using such a hash function has the advantage that no item needs to be replicated for clients to find it. Once an item is placed in a cache (bucket), subsequent clients go directly to the cache to access the item. However, this approach has a major drawback. In practice, the Internet is dynamic, with new caches constantly being added and caches going down or becoming full. Note that if we were to add a single new cache, the new hash function would become $h'(i) = ai + b \pmod{k+1}$. Thus, each item would now be mapped to a bucket that does not contain the item. Of course, when a client requests item I and it is not found in the cache (bucket) $h'(i)$, it can copy to $h'(i)$. However, in a worst-case scenario, after many additions of caches, it is possible that I might be replicated in every bucket.

We now describe a hash function, called a *consistent hash function*, with the property that, when a new bucket is added, exactly one of the old buckets has some of its items reassigned to the new bucket and all other assignments of items remain the same. In particular, we discuss a consistent hash function called UC_{random} . UC_{random} is based on first independently and randomly mapping the items and the buckets to the unit circle. This mapping can be achieved using randomly selected hash functions h_{items} and $h_{buckets}$, which map the set of items and set of buckets, respectively, to the integers $0, 1, \dots, N - 1$ for some large number N , where $0, 1, \dots, N - 1$ are identified with N points evenly distributed around the unit circle.

Once the items and buckets are mapped to the unit circle, the UC_{random} hash function h then maps each item to the clockwise nearest bucket. A sample UC_{random} hash function h involving 8 buckets and 14 items is shown in Figure 17.5.

FIGURE 17.5

Example UC_{random} hash function for $k = 8$ buckets and $t = 14$ items.



- $B_0 : I_4$
- $B_1 : I_6$
- $B_2 : I_{13}$
- $B_3 : I_0, I_1, I_7, I_{10}$
- $B_4 : I_5, I_{11}$
- $B_5 : I_8$
- $B_6 : I_3$
- $B_7 : I_2, I_9, I_{12}$

Note that if we were to add a new bucket B_8 between the points labeled I_{10} and I_7 in the unit circle in Figure 17.5, then only items I_1 and I_{10} would be reassigned to B_9 . The assignment of all other items to buckets would remain the same.

The consistent hash function given in Figure 17.5 assumes that all the caches are available to every client that requests an item. However, in practice, caches may be down at certain times. Also, clients from different regions may be restricted in the caches they can access. Thus, when different clients request item I , they may have different *views* of the available caches. Consistent hashing can be naturally generalized to take this into account. Instead of simply mapping each item I to a bucket B , we map each item-view pair (I, V) to a bucket B where, $V \subseteq \mathcal{B}$. Such hash functions, which map $\mathcal{I} \times 2^{\mathcal{B}}$ to \mathcal{B} , are called *ranged hash functions*. The function UC_{random} can easily be extended to a ranged hash function by mapping each item I requested by a client to the clockwise nearest cache (bucket) in the client's view V . For the same random hashing of 8 buckets and 14 items to the unit circle given in Figure 17.5, Figure 17.6 shows the UC_{random} hash function for three different views.

552 ■ PART IV: Parallel and Distributed Algorithms

FIGURE 17.6

UC_{random} hash function for three different views for the same hashing of 8 buckets and 14 items to the unit circle given in Figure 17.5.

$$V = \{B_0, B_2, B_5, B_6\}$$

$$B_0: I_0, I_1, I_4, I_7, I_{10}$$

$$B_2: I_{13}$$

$$B_5: I_2, I_5, I_8, I_9, I_{11}, I_{12}, I_{13}$$

$$B_6: I_3, I_6$$

$$V = \{B_5, B_6\}$$

$$B_5: I_2, I_5, I_8, I_9, I_{11}, I_{12}, I_{13}$$

$$B_6: I_0, I_1, I_3, I_4, I_6, I_7, I_{10}$$

$$V = \{B_2, B_3, B_7\}$$

$$B_2: I_3, I_4, I_6, I_{13}$$

$$B_3: I_0, I_1, I_7, I_8, I_{10}$$

$$B_7: I_2, I_5, I_9, I_{11}, I_{12}$$

The unit circle is equivalent to the unit interval $[0,1]$ in the sense that we can take any point on the unit circle and map it to the point 0 of the unit interval and then map the point that is distance $2\pi x$ in the clockwise direction around the unit circle to the point x of unit interval, $0 \leq x < 1$. Thus, each bucket and item is assigned an integer value between 0 and 1. It follows that, given an item I , we can compute the bucket B it is mapped to by UC_{random} in $O(\log|V|)$ time by performing a binary search on the sorted list of buckets in the given view V to find the position where I would be inserted in the list.

It is possible that the random hash function maps buckets to the unit circle in a nonuniform way, making the distance to the nearest counterclockwise bucket for some buckets disproportionately large. Because such buckets “control” a larger proportion of the unit circle, the expected number of items that would be hashed by the UC_{random} to these buckets may be much greater than it would be to other buckets. Figure 17.7(a) shows such a nonuniform distribution involving eight buckets, where bucket B_3 controls a disproportionately large portion of the unit circle; and Figure 17.7(b) shows a sample mapping of 13 items to the same unit circle and the associated unbalanced assignment of these items to buckets by UC_{random} . To avoid such unfortunate distributions, we duplicate each bucket m times by randomly hashing each bucket to m different points on the unit circle. An item is still mapped to the nearest clockwise bucket, but the likelihood of a more balanced assignment of items to a bucket is improved. For the same mapping of items as in Figure 17.7(b), in Figure 17.7(c) a duplicate of each bucket is also mapped to the unit circle illustrating the case $m = 2$. Note the improved distribution of items to buckets.

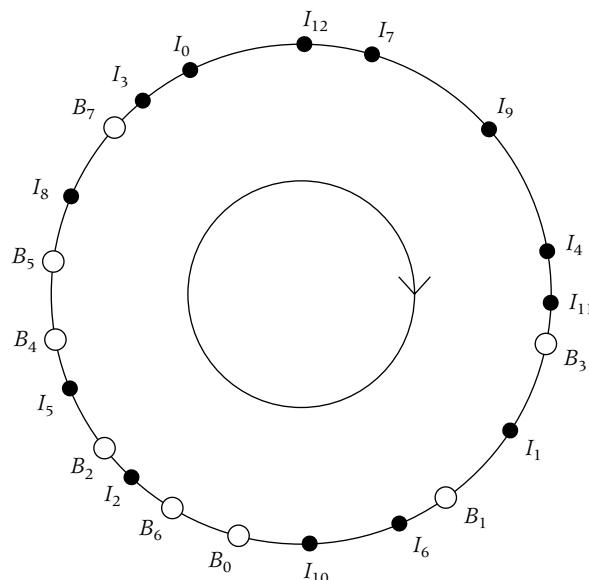
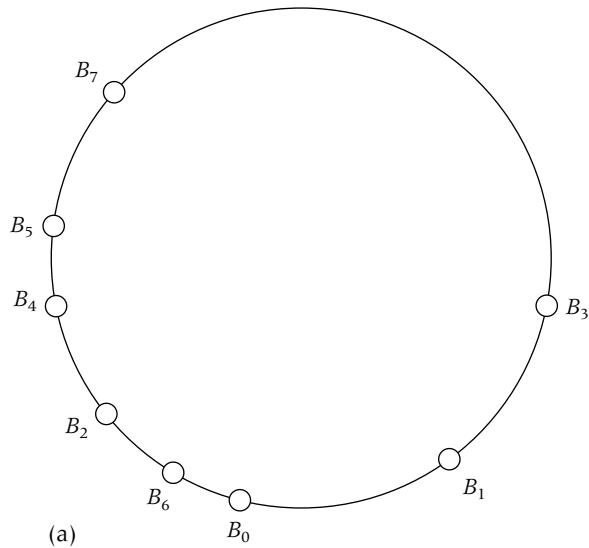
An easily verified key fact about the UC_{random} hash function is the following monotonicity property.

Key Fact

If a bucket is added to a view, the only items reassigned are those assigned to the added bucket. More generally, given a UC_{random} hash function f , two views V_1, V_2 , where $V_1 \subset V_2$, and an item I , $f(I, V_2) \in V_1$, implies that $f(I, V_1) = f(I, V_2)$.

FIGURE 17.7

- (a) Nonuniform distribution, where bucket B_3 controls a disproportionately large portion of the unit circle;
 (b) assignment of buckets to items for a sample mapping of 13 items to the unit circle;



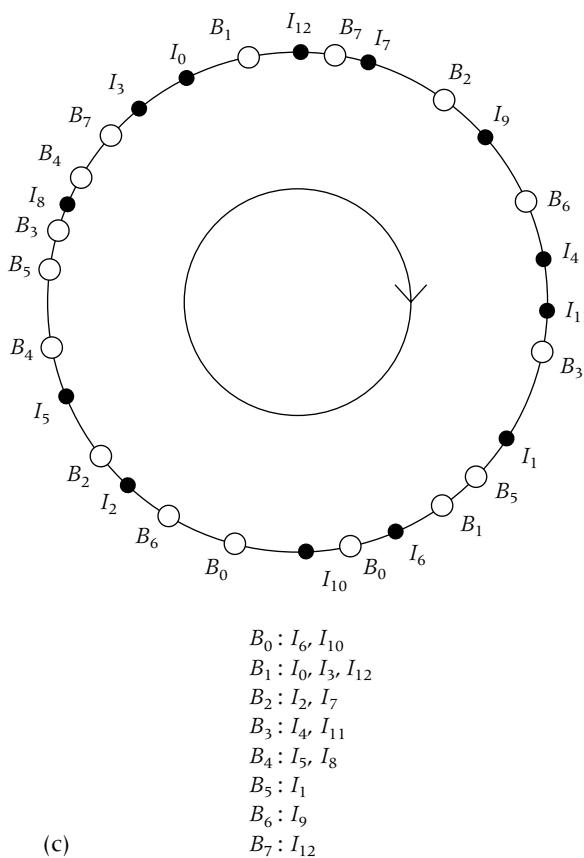
(b)

$B_0 : I_6, I_{10}$
 $B_1 : I_1$
 $B_2 : I_2$
 $B_3 : I_0, I_3, I_4, I_7, I_9, I_{11}, I_{12}$
 $B_4 : I_5$
 $B_5 : \text{empty}$
 $B_6 : \text{empty}$
 $B_7 : I_8$

continued

FIGURE 17.7

Continued.
 (c) for the same mapping of items as in (a), a duplicate of each bucket is also mapped to the unit circle illustrating the case $m = 2$. Note the improved distribution of items to buckets



In addition to being monotone, the UC_{random} function has several other desirable properties. Three important goodness measures associated with hashing items to caches are spread, load, and balance. Consider a set of c clients with views V_1, \dots, V_t . Let \mathcal{B}_t denote the union of the buckets over all these views; that is, $\mathcal{B}_t = V_1 \cup \dots \cup V_t$. Also, let b denote the number of such buckets; that is, $b = |\mathcal{B}_t|$. The *spread* of an item with respect to the set of views V_1, \dots, V_t is the number of buckets from \mathcal{B}_t containing the item. The *load* of a bucket $B \in \mathcal{B}_t$ is the number of items assigned to B over the set of views V_1, \dots, V_t . Given a view V , item I , and bucket $B \in V$, the *balance* is the probability that item I is mapped to B . It is desirable for the spread and load to be small and for the balance to be close to $1/|V|$. The following theorem (see the paper of Karger, et al. in the references),

whose proof is beyond the scope of this book, shows that the spread and load are small for UC_{random} , and items are uniformly distributed among the buckets in any given view.

Theorem 17.4.1

Consider a set of t views V_1, \dots, V_t , having the property that the number of buckets in each view contains at least a constant fraction of the buckets over all the views; that is, letting $\mathcal{B}_t = V_1 \cup \dots \cup V_t$ and $b = |\mathcal{B}_t|$, suppose $|V_i| \geq b/c$, for some constant c , $i = 1, \dots, t$. If m denotes the multiplicity with which each bucket is mapped to the unit circle by UC_{random} and r is a positive number representing a confidence factor, then the following three properties of UC_{random} hashing hold:

1. **Spread:** With probability at least $1 - 1/r$, any given item I is stored in $O(\log rt)$ buckets.
2. **Load:** Over all the views V_1, \dots, V_t , with probability at least $1 - 1/r$, no bucket $B \in \mathcal{B}_t$ is assigned more than $O(\log rmt)$ times the average number of items assigned to the buckets in \mathcal{B}_t .
3. **Balance:** If $m \in \Omega(\log b)$, then for any fixed view V and item I , the probability that item I is mapped to a given bucket $B \in V$ is $O(1/|V|)$. \square



17.5 Message Security Algorithms: RSA

To make messages sent over the Internet more secure, a cryptographic key can be used to encrypt the message. In Chapter 3, we discussed an algorithm for computing a cryptographic key known to only two parties, Alice and Bob. In this section, we consider the problem of computing both a public key E and secret (private) key D , where E is used to encrypt messages and D is used to decrypt messages that have been encrypted using E . These keys need to be chosen so that it is computationally infeasible to derive D from E . Anyone with the public key E is able to encrypt a message, but only someone knowing D is able (in real time) to decrypt an encrypted message. Such a system is called a *public-key cryptosystem*. Many public-key cryptosystems have been designed. In this section, we discuss the popular RSA public key cryptosystem, named for its creators, Rivest, Shamir, and Adleman.

17.5.1 Modular Arithmetic

Before describing the RSA algorithm, we need to establish some concepts and results from modular arithmetic. The set $Z_n = \{0, \dots, n - 1\}$ of integers (*residues*) mod n forms a commutative ring (see Appendix A), where the operations of addition and multiplication are defined the same as over the integers, but where the result of each operation is reduced mod n to an element in Z_n .

An element x of Z_n is *invertible* with respect to multiplication if there is an element y in Z_n , such that $xy \equiv 1 \pmod{n}$. We will refer to y as the *multiplicative inverse* of x . Integers x and n are *relatively prime* if and only if they have no divisor except 1 in common. Equivalently, x and n are relatively prime if and only if $\gcd(x, n) = 1$.

Proposition 17.5.1 An element x of Z_n is invertible with respect to multiplication if and only if it is relatively prime to n (that is, $\gcd(x, n) = 1$).

PROOF

Let $g = \gcd(x, n)$. It follows from Exercise 1.10 (see Chapter 1) that g can be expressed as an integer linear combination of x and n ; that is, there exist integers s and t such that $sx + tn = g$. Further, g is the smallest integer with this property. Now suppose that x and n are relatively prime. Then $sx + tn = g = 1$, so that $sx \equiv 1 \pmod{n}$. It follows that $s \pmod{n}$ is the inverse of $x \pmod{n}$. Conversely, suppose that x is invertible, so that there exists an integer y such $xy \equiv 1 \pmod{n}$. Then we have that $xy - 1 = mn$ for some integer m , or equivalently, $xy - mn = 1 \pmod{n}$. However, since g is the smallest integer that is expressible as a linear combination of x and n , it follows that $g = 1$. ■

Let Z_n^* denote the subset of all nonnegative integers less than n that are relatively prime to n . For example, for the integers mod 15 shown in Figure 17.9, $Z_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$. Then by Proposition 17.5.1, Z_n^* consists of all the elements from Z_n with a multiplicative inverse mod n . Given two integers x and y , the product xy is relatively prime with n if and only if both x and y are relatively prime with n . It follows that Z_n^* is closed under multiplication mod n . Thus, Z_n^* forms a group under multiplication mod n . The multiplication table for the group Z_{15}^* is shown in Figure 17.9. This table is obtained from the multiplication table for Z_{15} by deleting all rows and columns that do not contain a 1, which is precisely those rows and columns that are indexed by an integer having a divisor in common with 15.

FIGURE 17.8

The multiplication table for the group Z_{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

The inverse of an element x of Z_n^* can be computed efficiently using an extension of Euclid's algorithm (again, see Exercise 1.10) that computes integers g , s , and t for input integers a and b , where $g = \gcd(a, b)$ and $sa + tb = g$.

Let $\varphi(n)$ denote the number of elements less than n that are relatively prime to n , so that $\varphi(n)$ is the size of Z_n^* . The following useful theorem is due to Euler and generalizes an earlier result of Fermat known as Fermat's little theorem.

Theorem 17.5.2 Let x and n be relatively prime numbers. Then

$$x^{\varphi(n)} \equiv 1 \pmod{n}. \quad (17.5.1)$$

PROOF

Because x and n are relatively prime numbers, it follows that $x \in Z_n^*$. Let $a_1, a_2, \dots, a_{\varphi(n)}$ denote the elements of Z_n^* . Since x is invertible, $xa_1, xa_2, \dots, xa_{\varphi(n)}$ is a permutation of the elements $a_1, a_2, \dots, a_{\varphi(n)}$. Thus, we have that

$$(xa_1)(xa_2)\dots(xa_{\varphi(n)}) = x^{\varphi(n)}a_1a_2\dots a_{\varphi(n)} = a_1a_2\dots a_{\varphi(n)} \pmod{n} \quad (17.5.2)$$

Since the product $a_1a_2\dots a_{\varphi(n)}$ is also in Z_n^* , it is invertible. Multiplying both sides of Formula (17.5.2) by the inverse of $a_1a_2\dots a_{\varphi(n)}$, we obtain Formula (17.5.1). ■

558 ■ PART IV: Parallel and Distributed Algorithms

Observing that $\varphi(n) = n - 1$ when n is a prime number, the following corollary known as Fermat's little theorem, follows immediately from Theorem 17.5.2.

Corollary 17.5.3 Fermat's Little Theorem

Let n be prime number and let x be any number that is not divisible by n . Then

$$x^{n-1} \equiv 1 \pmod{n}. \quad (17.5.3)$$

□

The following proposition will be useful in the design of the RSA algorithm.

Proposition 17.5.4 Let n be an integer that is the product of two prime numbers p and q . Then

$$\varphi(n) = (p - 1)(q - 1).$$

PROOF

The elements of Z_n that have a divisor different from 1 in common with n , excluding n itself, are the $q - 1$ multiples of p —namely, $p, 2p, \dots, (q - 1)p$ —and the $p - 1$ multiples of q —namely, $q, 2q, \dots, (p - 1)q$. Thus, including pq , there are precisely $(p - 1) + (q - 1) + 1 = p + q - 1$ elements of Z_n that have a divisor different from 1 in common with n . Therefore, the number of elements of Z_n that do not have a divisor different from 1 in common with n —that is, the number of elements relatively prime to n —is given by $\varphi(n) = pq - (p + q - 1) = (p - 1)(q - 1)$. ■

As an illustration of Proposition 3.2.4, consider Z_{15} , whose addition and multiplication tables are illustrated in Figure 17.8. In this case, $p = 3$ and $q = 5$. As shown in Figure 17.8, the number of elements of Z_{15}^* is $(3 - 1)(5 - 1) = 8$.

The following theorem is the basis for the RSA algorithm described in the next subsection.

Theorem 17.5.5

Let $n = pq$, where p and q are two prime numbers; let e be an integer that is relatively prime with $\varphi(n)$; and let d be its multiplicative inverse mod $\varphi(n)$, that is, $ed \equiv 1 \pmod{\varphi(n)}$. Then for any integer m ,

$$m^{ed} \equiv m \pmod{n}.$$

PROOF

Since $ed \equiv 1 \pmod{\varphi(n)}$, it follows that $ed = \varphi(n)k + 1$ for some integer k . First suppose that $\gcd(m, n) = 1$. Then applying Theorem 17.5.2, we have

$$\begin{aligned} m^{ed} &= m^{\varphi(n)k+1} \\ &= (m^{\varphi(n)})^k m \\ &\equiv (1)^k m = m \pmod{n} \end{aligned}$$

Now consider the case when $\gcd(m, n) > 1$. If $\gcd(m, n) = n$, then n divides m and we have

$$m^{ed} \equiv 0^{ed} \equiv 0 \equiv m \pmod{n}.$$

Otherwise, since $n = pq$, either m is divisible by q but not p or n is divisible by p but not q . Assume without loss of generality that m is divisible by q but not p . Then by Fermat's little theorem (Corollary 17.5.3),

$$m^{p-1} \equiv 1 \pmod{p}. \quad (17.5.4)$$

Applying Formula (17.5.4), we obtain

$$\begin{aligned} m^{k\varphi(n)} &= m^{k(p-1)(q-1)} \\ &\equiv (m^{p-1})^{k(q-1)} \pmod{p} \\ &\equiv (1)^{k(q-1)} \equiv 1 \pmod{p}. \end{aligned}$$

It follows that

$$m^{k\varphi(n)} = jp + 1 \quad (17.5.5)$$

for some integer j . Multiplying both sides of Formula (17.5.5) by m , we obtain

$$m^{k\varphi(n)+1} = jpm + m. \quad (17.5.6)$$

However, because m is divisible by q , jpm is divisible by n . Therefore, we have

$$m^{k\varphi(n)+1} \equiv m \pmod{n}. \quad \blacksquare$$

17.5.2 RSA Public-Key Cryptosystem

We now describe the RSA algorithm for computing a public and secret key for encrypting a message M and decrypting an encrypted message, respectively. Because a string can be represented as a binary sequence (for example, if the message is a string of ASCII characters, then each character in the string can be replaced with an 8-bit binary string representing its ASCII value), we can assume without loss of generality that M is an integer. We begin by choosing two large prime numbers p and q . To achieve security—that is, so that it is infeasible to compute the secret key from the public key—these primes must be about 100 digits long or longer. (The problem of designing efficient probabilistic algorithms for computing large prime numbers is discussed in Chapter 24.) We then compute the product $n = pq$.

We now choose a small odd integer e that is relatively prime with $\varphi(n)$, and take the *public key* to be the pair (e, n) . The *secret key* is then chosen to be the pair (d, n) , where d is the multiplicative inverse of $e \bmod \varphi(n)$. A message M is encrypted by taking it to the power $e \bmod n$, and an encrypted message $C = M^e$ is decrypted by taking it to the power $d \bmod n$. By Theorem 17.5.5, this is valid because $C^d = (M^e)^d = M^{ed} \equiv M \pmod{n}$. Using the modular exponential algorithm discussed in Chapter 3, both the encryption and decryption of messages can be performed efficiently. In fact, they require $O(b^3)$ bit operations, where b is the number of digits of n in its binary representation (see Exercise 17.17).

For the RSA cryptosystem to be unbreakable, d must not be computable from e in real time. It is generally believed that to compute d from e would require the factorization of n into its prime factors p and q . In practice, it is surprisingly difficult to factor large numbers (with hundreds of digits), even knowing that n is the product of two prime numbers p and q . Although the problem of factoring a number into prime factors has not been proved to be NP-hard, mathematicians and computer scientists alike have worked on the factorization problem for many years, and it is generally believed to be difficult. However, small numbers n can be efficiently factored, so we need to choose primes p and q that are about 100 digits long or longer to make the RSA cryptosystem unbreakable.

R E M A R K S

1. The message M before encryption is often called **plaintext** and after encryption is called **ciphertext**.
2. Encrypting a message using RSA may be too slow if the message is long. For such messages, RSA is often used in conjunction with a faster nonpublic-key cryptosystem, where encryption and decryption are done using the same key K . The idea is that Alice selects a key K at random and encrypts M using K to obtain the encrypted message (ciphertext) C . She then encrypts K using Bob's public RSA key $E = (e, n)$ to obtain $E(K) = K^e$ and transmits $(C, E(K))$ to Bob. K^e can be computed efficiently because K is small. After receiving the message, Bob then uses his secret RSA key $D(K) = (d, n)$ to decrypt $P(K)$, obtaining K , after which he uses K to decrypt C , obtaining the message M .

17.5.3 Digital Signatures and Authentication

Public-key cryptosystems, and in particular the RSA cryptosystem, have another very important application, the *authentication* of messages using *digital signatures*. Suppose that Alice wishes to send a message to Bob together with a digital signature, so Bob can verify that the message is actually from her and not a forgery from someone else. To do this, Alice encrypts M using her secret key D , yielding the digital signature $S = D(M)$. The RSA signature for M is $S = M^d \pmod{n}$. Alice then sends the pair (M, S) to Bob. Bob can authenticate the message M using the digital signature S and Alice's public key E by checking whether $M = E(S)$. In the case of the RSA signature, Bob checks whether $M = S^e \pmod{n}$.



17.6 Closing Remarks

In this chapter, we discussed two algorithms *PageRank* and *HITS* for ranking Web pages. Both algorithms are effective for *broad-based queries*, where there is an abundance of pages containing the query keyword. A user can make several other types of queries. One other type of query is called a *specific query*, which involves asking a question, such as "What is a search engine?" Another type is called a *similar-page query*, which involves finding Web pages similar to a given Web page or document. Different types of queries require different handling techniques. Designing efficient algorithms for these problems is an important and challenging area of research in Internet algorithms.

562 ■ PART IV: Parallel and Distributed Algorithms

Search engines use hash functions extensively. In particular, they use perfect hash functions to map URLs to an index. For a given set of input keys S contained in some universal set U of keys, a function $h : U \rightarrow \{0, 1, \dots, m - 1\}$ is called a *perfect hash function for S* if h is one-to-one on S . Thus, a perfect hash function avoids collisions altogether on S . If $|S| = n \leq m$, then clearly there are many perfect hash functions for S . The problem is to compute a perfect hash function efficiently for a given S , m , and U . Another requirement is that the perfect hash function should evaluate at any point in U in constant time. Quite a bit of research has been done on the problem of finding perfect hash functions. Most of the recent work in this direction involves probabilistic algorithms, which will be the topic of Chapter 24.

Various variants of the RSA public-key cryptosystem have been developed that speed up the decryption process. One such variant, which uses the Chinese remainder theorem, is explored in the exercises.

Other algorithms with applications to the Internet will be discussed in later chapters, such as string matching in Chapter 20, the FFT in Chapter 22, and multicasting and the Steiner tree in Chapter 27.

References and Suggestions for Further Reading

Chakrabarti, S. *Mining the Web, Discovering Knowledge from Hypertext Data*. San Francisco, CA: Morgan Kaufman Publishers, 2003. A book on mining the World Wide Web, including a discussion of the ranking and relevancy of Web pages.

Schmeh, K. *Cryptography and Public Key Infrastructure on the Internet*. West Sussex, England: John Wiley & Sons, 2003. A book on cryptography related to the Internet.

Karger D., E. Lehman, T. Leighton, M. Levin, D. Lewin, R. Panigrahy “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.” *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, May 1997. Original paper on consistent hashing and the UC_{random} hash function.

Detailed discussions of hashing:

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.

Gonnet, G. H. *Handbook of Algorithms and Data Structures*. Reading, MA: Addison-Wesley, 1984.

Knuth, D. E. *The Art of Computer Programming*. Vol. 3, *Sorting and Searching*. 2nd ed. Reading, MA: Addison-Wesley, 1998.

Majewski, B. S., N. C. Wormahld, G. Havas, and A. J. Czech. "Graphs, Hypergraphs, and Hashing." *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '93)*, 1993. An excellent guide to perfect hashing schemes.

EXERCISES

Section 17.2 Ranking Web Pages

- 17.1 a. Verify that matrix formula (17.2.3) is the same as Formula (17.2.1).
 b. Verify that Formula (17.2.6) follows from Formula (17.2.4).
- 17.2 a. Describe a digraph on n vertices, which is strongly connected but not aperiodic.
 b. Does R_i converge to the R as i goes to infinity for the digraph you have given in part (a), where $R_0 = (1, 1, \dots, 1)$?
- 17.3 Suppose $R_0[p] = 1$ if $p = q$, and $R_0[q] = 0$ otherwise, for p and q two nodes of the digraph W . Show that, if there is no path from q to p , then $R_i[p] = 0$ for all i , whereas if there is a path, then $R_i[p]$ is nonzero for some i .
- 17.4 Write a program that computes R_i for input digraphs W , number of iterations i , damping factor d , and initial vector R_0 . Test how quickly R_i converges R or whether it converges at all, for various choices of R_0 and damping factors d . In particular, test for the following input digraphs and damping factors d :
 - a. Digraphs that are not strongly connected but not aperiodic, with $d = 0$.
 - b. Digraphs that are aperiodic with $d = 0$.
 - c. Digraphs with various nonzero values of d .
- 17.5 Show that the i^{th} power matrix of the matrix B has the property that its pq^{th} entry $B^i[p][q]$ is the probability that an aimless surfer starting at page p reaches q in i steps.
- 17.6 Obtain a matrix formula for PageRank given by Formula (17.2.8), which generalizes Formula (17.2.4).
- 17.7* Show that if the digraph W is aperiodic, then R_i will converge to the principal eigenvector R of B^T as i goes to infinity. Further, show that this convergence is independent of the choice of the initial vector R_0 whose entries are positive and sum to 1.

Section 17.3 Hashing

- 17.8 Show the contents of an open-address hash table $H[0:22]$ after the insertion of the keys 2, 11, 55, 23, 53, 9, 61, 46, 1, 3, 25, 19, 17, 34, 10, 38, and 48, where $h(k) = k \bmod 23$. Use linear probing to resolve collisions.
- 17.9 Repeat Exercise 17.8, using double hashing, with hash functions $h_1(k) = k \bmod 23$, and $h_2(k) = 1 + (k \bmod 21)$.
- 17.10 Consider the (not-so-good) hash function $h(i)$, which maps an integer i to its leading digit. Using the method of collision resolution by chaining, suppose the keys 23, 55, 123, 504, 211, 200, 88, 91, and 9 have been inserted in the hash table. Assuming each slot (leading digit) is equally likely for a given search k , determine the average behavior of a successful search.
- 17.11 Repeat Exercise 17.10 for an unsuccessful search.
- 17.12 Show that the probe sequence given by Formula (17.3.7) reaches all slots in the table if and only if $h_2(k)$ and m are relatively prime.
- 17.13 Prove Theorem 17.3.1.

Section 17.4 Caching, Content Delivery, and Consistent Hashing

- 17.14 Prove that the UC_{random} hash function h is monotone by showing that, given two views V_1, V_2 , where $V_1 \subset V_2$, and an item I , $h(I, V_2) \in V_1$ implies that $h(I, V_1) = h(I, V_2)$.
- 17.15 Associate with each item $I \in \mathcal{I}$ a permutation π_I of the buckets, and consider the ranged hash function h that maps the item-view pair (I, V) onto the bucket $\pi_I(j)$, where j is the smallest index such that $\pi_I(j) \in V$.
- Show that the function h is monotone.
 - Show that any monotone function can be constructed in this way.

Section 17.5 Message Security Algorithms: RSA

- 17.16 Give the multiplication table for the group Z_{23}^* .
- 17.17 Using the extended Euclid algorithm described in Exercise 1.10 (Chapter 1), compute the following:
- Inverse of 3 mod 1097.
 - Inverse of 46 mod 1097.
 - Inverse of 127 mod 19888.
- 17.18 Consider the problem of dividing a by b . Show that, using the straightforward algorithm for long division, the quotient q and remainder r can be computed using $O(\log b + \log q \log b)$ bit operations.
- 17.19 Using the result from Exercise 17.18, show that RSA encryption and decryption of messages can be performed using $O(\beta^3)$ bit operations, where β is the number of binary digits of n .
- 17.20 Prove the following result, known as the Chinese remainder theorem: If m_1, m_2, \dots, m_k are k pairwise relatively prime and greater than or equal to 2, and a_1, a_2, \dots, a_k are any k integers, then there is a solution to the following simultaneous congruences:

$$x \equiv a_1 \pmod{m_1}$$

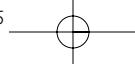
$$x \equiv a_2 \pmod{m_2}$$

$$\vdots$$

$$x \equiv a_k \pmod{m_k}.$$

In particular, $x = a_1 b_1 \frac{M}{m_1} + \dots + a_k b_k \frac{M}{m_k}$, where $M = m_1 m_2 \cdots m_k$ and b_i is determined from $b_i \frac{M}{m_i} \equiv 1 \pmod{m_i}$, $i = 1, \dots, k$.

- 17.21 Instead of directly decrypting the original message by computing $M \equiv C^d \pmod{n}$, as in the original RSA algorithm, we can achieve a speedup of about 4 by computing $M_p \equiv C^{d_p} \pmod{p}$ and $M_q \equiv C^{d_q} \pmod{q}$, where $d_p = d \pmod{p-1}$ and $d_q = d \pmod{q-1}$, and then computing $M = q(q^{-1} \pmod{p})M_p + p(p^{-1} \pmod{q})M_q$.

**566 ■ PART IV:** Parallel and Distributed Algorithms

a. Show that M satisfies the following two congruences:

$$M \equiv M_p \pmod{p}$$

$$M \equiv M_q \pmod{q}$$

b. Applying the Chinese remainder theorem, show that M satisfies

$$M = q(q^{-1} \pmod{p})M_p + p(p^{-1} \pmod{q})M_q.$$

- 17.22 a. A speedup of about 9 can be achieved if n is the product of three primes $n = pqr$. Extend the result in Exercise 17.21 to for this case.
b. Generalize the result in part (a) for n the product of k primes.