Introduction to Algorithms

Third Edition

DISCLAIMER:

ReadMe website is intended for academic and demonstration purposes only. We’re only showing a preview of the book to respect the author’s copyright. Thank you for your understanding!

- Group 4: The Classified

Floors and ceilings

For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the floor of x") and the least integer greater than or equal to x by $\lceil x \rceil$ (read "the ceiling of x"). For all real x,

$$x - 1 \ < \ \lfloor x \rfloor \ \leq \ x \ \leq \ \lceil x \rceil \ < \ x + 1 \ . \tag{3.3}$$

For any integer n,

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n \ ,$$

and for any real number $x \geq 0$ and integers $a, b > 0$,

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil \ = \ \left\lceil \frac{x}{ab} \right\rceil \ , \tag{3.4}$$

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor \ = \ \left\lfloor \frac{x}{ab} \right\rfloor \ , \tag{3.5}$$

$$\left\lceil \frac{a}{b} \right\rceil \ \leq \ \frac{a + (b-1)}{b} \ , \tag{3.6}$$

$$\left\lfloor \frac{a}{b} \right\rfloor \ \geq \ \frac{a - (b-1)}{b} \ . \tag{3.7}$$

The floor function $f(x) = \lfloor x \rfloor$ is monotonically increasing, as is the ceiling function $f(x) = \lceil x \rceil$.

Modular arithmetic

For any integer a and any positive integer n, the value $a \bmod n$ is the ***remainder*** (or ***residue***) of the quotient a/n:

$$a \bmod n = a - n \lfloor a/n \rfloor \ . \tag{3.8}$$

It follows that

$$0 \leq a \bmod n < n \ . \tag{3.9}$$

Given a well-defined notion of the remainder of one integer when divided by another, it is convenient to provide special notation to indicate equality of remainders. If $(a \bmod n) = (b \bmod n)$, we write $a \equiv b \pmod{n}$ and say that a is ***equivalent*** to b, modulo n. In other words, $a \equiv b \pmod{n}$ if a and b have the same remainder when divided by n. Equivalently, $a \equiv b \pmod{n}$ if and only if n is a divisor of $b - a$. We write $a \not\equiv b \pmod{n}$ if a is not equivalent to b, modulo n.

that $\Pr\{E_2 \mid E_1\} = 1/(n-1)$ because given that element $A[1]$ has the smallest priority, each of the remaining $n-1$ elements has an equal chance of having the second smallest priority. In general, for $i = 2, 3, \ldots, n$, we have that $\Pr\{E_i \mid E_{i-1} \cap E_{i-2} \cap \cdots \cap E_1\} = 1/(n-i+1)$, since, given that elements $A[1]$ through $A[i-1]$ have the $i-1$ smallest priorities (in order), each of the remaining $n-(i-1)$ elements has an equal chance of having the ith smallest priority. Thus, we have

$$\begin{aligned}\Pr\{E_1 \cap E_2 \cap E_3 \cap \cdots \cap E_{n-1} \cap E_n\} &= \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right)\cdots\left(\frac{1}{2}\right)\left(\frac{1}{1}\right) \\ &= \frac{1}{n!}\,,\end{aligned}$$

and we have shown that the probability of obtaining the identity permutation is $1/n!$.

We can extend this proof to work for any permutation of priorities. Consider any fixed permutation $\sigma = \langle\sigma(1), \sigma(2), \ldots, \sigma(n)\rangle$ of the set $\{1, 2, \ldots, n\}$. Let us denote by r_i the rank of the priority assigned to element $A[i]$, where the element with the jth smallest priority has rank j. If we define E_i as the event in which element $A[i]$ receives the $\sigma(i)$th smallest priority, or $r_i = \sigma(i)$, the same proof still applies. Therefore, if we calculate the probability of obtaining any particular permutation, the calculation is identical to the one above, so that the probability of obtaining this permutation is also $1/n!$. ■

You might think that to prove that a permutation is a uniform random permutation, it suffices to show that, for each element $A[i]$, the probability that the element winds up in position j is $1/n$. Exercise 5.3-4 shows that this weaker condition is, in fact, insufficient.

A better method for generating a random permutation is to permute the given array in place. The procedure RANDOMIZE-IN-PLACE does so in $O(n)$ time. In its ith iteration, it chooses the element $A[i]$ randomly from among elements $A[i]$ through $A[n]$. Subsequent to the ith iteration, $A[i]$ is never altered.

```
RANDOMIZE-IN-PLACE(A)
1  n = A.length
2  for i = 1 to n
3      swap A[i] with A[RANDOM(i, n)]
```

We shall use a loop invariant to show that procedure RANDOMIZE-IN-PLACE produces a uniform random permutation. A ***k-permutation*** on a set of n elements is a sequence containing k of the n elements, with no repetitions. (See Appendix C.) There are $n!/(n-k)!$ such possible k-permutations.

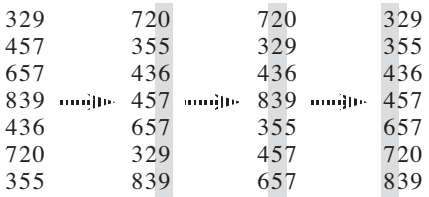

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

In a typical computer, which is a sequential random-access machine, we sometimes use radix sort to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day, next on month, and finally on year.

The code for radix sort is straightforward. The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

RADIX-SORT(A, d)

1 **for** $i = 1$ **to** d
2 use a stable sort to sort array A on digit i

Lemma 8.3

Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n + k))$ time if the stable sort it uses takes $\Theta(n + k)$ time.

Proof The correctness of radix sort follows by induction on the column being sorted (see Exercise 8.3-3). The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit is in the range 0 to $k-1$ (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes time $\Theta(n+k)$. There are d passes, and so the total time for radix sort is $\Theta(d(n + k))$. ■

When d is constant and $k = O(n)$, we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits.

no elements are stored outside the table, unlike in chaining. Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made; one consequence is that the load factor α can never exceed 1.

Of course, we could store the linked lists for chaining inside the hash table, in the otherwise unused hash-table slots (see Exercise 11.2-4), but the advantage of open addressing is that it avoids pointers altogether. Instead of following pointers, we *compute* the sequence of slots to be examined. The extra memory freed by not storing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

To perform insertion using open addressing, we successively examine, or ***probe***, the hash table until we find an empty slot in which to put the key. Instead of being fixed in the order $0, 1, \ldots, m-1$ (which requires $\Theta(n)$ search time), the sequence of positions probed *depends upon the key being inserted*. To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input. Thus, the hash function becomes

$$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\} \ .$$

With open addressing, we require that for every key k, the ***probe sequence***

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up. In the following pseudocode, we assume that the elements in the hash table T are keys with no satellite information; the key k is identical to the element containing key k. Each slot contains either a key or NIL (if the slot is empty). The HASH-INSERT procedure takes as input a hash table T and a key k. It either returns the slot number where it stores key k or flags an error because the hash table is already full.

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == NIL
5            T[j] = k
6            return j
7       else i = i + 1
8  until i == m
9  error "hash table overflow"
```

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. Therefore, the search can

```
OS-RANK(T, x)
1  r = x.left.size + 1
2  y = x
3  while y ≠ T.root
4       if y == y.p.right
5            r = r + y.p.left.size + 1
6       y = y.p
7  return r
```

The procedure works as follows. We can think of node x's rank as the number of nodes preceding x in an inorder tree walk, plus 1 for x itself. OS-RANK maintains the following loop invariant:

> At the start of each iteration of the **while** loop of lines 3–6, r is the rank of $x.key$ in the subtree rooted at node y.

We use this loop invariant to show that OS-RANK works correctly as follows:

Initialization: Prior to the first iteration, line 1 sets r to be the rank of $x.key$ within the subtree rooted at x. Setting $y = x$ in line 2 makes the invariant true the first time the test in line 3 executes.

Maintenance: At the end of each iteration of the **while** loop, we set $y = y.p$. Thus we must show that if r is the rank of $x.key$ in the subtree rooted at y at the start of the loop body, then r is the rank of $x.key$ in the subtree rooted at $y.p$ at the end of the loop body. In each iteration of the **while** loop, we consider the subtree rooted at $y.p$. We have already counted the number of nodes in the subtree rooted at node y that precede x in an inorder walk, and so we must add the nodes in the subtree rooted at y's sibling that precede x in an inorder walk, plus 1 for $y.p$ if it, too, precedes x. If y is a left child, then neither $y.p$ nor any node in $y.p$'s right subtree precedes x, and so we leave r alone. Otherwise, y is a right child and all the nodes in $y.p$'s left subtree precede x, as does $y.p$ itself. Thus, in line 5, we add $y.p.left.size + 1$ to the current value of r.

Termination: The loop terminates when $y = T.root$, so that the subtree rooted at y is the entire tree. Thus, the value of r is the rank of $x.key$ in the entire tree.

As an example, when we run OS-RANK on the order-statistic tree of Figure 14.1 to find the rank of the node with key 38, we get the following sequence of values of $y.key$ and r at the top of the **while** loop:

iteration	$y.key$	r
1	38	2
2	30	4
3	41	4
4	26	17

16 Greedy Algorithms

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. A ***greedy algorithm*** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. This chapter explores optimization problems for which greedy algorithms provide optimal solutions. Before reading this chapter, you should read about dynamic programming in Chapter 15, particularly Section 15.3.

Greedy algorithms do not always yield optimal solutions, but for many problems they do. We shall first examine, in Section 16.1, a simple but nontrivial problem, the activity-selection problem, for which a greedy algorithm efficiently computes an optimal solution. We shall arrive at the greedy algorithm by first considering a dynamic-programming approach and then showing that we can always make greedy choices to arrive at an optimal solution. Section 16.2 reviews the basic elements of the greedy approach, giving a direct approach for proving greedy algorithms correct. Section 16.3 presents an important application of greedy techniques: designing data-compression (Huffman) codes. In Section 16.4, we investigate some of the theory underlying combinatorial structures called "matroids," for which a greedy algorithm always produces an optimal solution. Finally, Section 16.5 applies matroids to solve a problem of scheduling unit-time tasks with deadlines and penalties.

The greedy method is quite powerful and works well for a wide range of problems. Later chapters will present many algorithms that we can view as applications of the greedy method, including minimum-spanning-tree algorithms (Chapter 23), Dijkstra's algorithm for shortest paths from a single source (Chapter 24), and Chvátal's greedy set-covering heuristic (Chapter 35). Minimum-spanning-tree algorithms furnish a classic example of the greedy method. Although you can read

from the spindle. When a given head is stationary, the surface that passes underneath it is called a ***track***. Multiple platters increase only the disk drive's capacity and not its performance.

Although disks are cheaper and have higher capacity than main memory, they are much, much slower because they have moving mechanical parts.[1] The mechanical motion has two components: platter rotation and arm movement. As of this writing, commodity disks rotate at speeds of 5400–15,000 revolutions per minute (RPM). We typically see 15,000 RPM speeds in server-grade drives, 7200 RPM speeds in drives for desktops, and 5400 RPM speeds in drives for laptops. Although 7200 RPM may seem fast, one rotation takes 8.33 milliseconds, which is over 5 orders of magnitude longer than the 50 nanosecond access times (more or less) commonly found for silicon memory. In other words, if we have to wait a full rotation for a particular item to come under the read/write head, we could access main memory more than 100,000 times during that span. On average we have to wait for only half a rotation, but still, the difference in access times for silicon memory compared with disks is enormous. Moving the arms also takes some time. As of this writing, average access times for commodity disks are in the range of 8 to 11 milliseconds.

In order to amortize the time spent waiting for mechanical movements, disks access not just one item but several at a time. Information is divided into a number of equal-sized ***pages*** of bits that appear consecutively within tracks, and each disk read or write is of one or more entire pages. For a typical disk, a page might be 2^{11} to 2^{14} bytes in length. Once the read/write head is positioned correctly and the disk has rotated to the beginning of the desired page, reading or writing a magnetic disk is entirely electronic (aside from the rotation of the disk), and the disk can quickly read or write large amounts of data.

Often, accessing a page of information and reading it from a disk takes longer than examining all the information read. For this reason, in this chapter we shall look separately at the two principal components of the running time:

- the number of disk accesses, and
- the CPU (computing) time.

We measure the number of disk accesses in terms of the number of pages of information that need to be read from or written to the disk. We note that disk-access time is not constant—it depends on the distance between the current track and the desired track and also on the initial rotational position of the disk. We shall

[1] As of this writing, solid-state drives have recently come onto the consumer market. Although they are faster than mechanical disk drives, they cost more per gigabyte and have lower capacities than mechanical disk drives.

c. Modify the VEB-TREE-INSERT procedure to produce pseudocode for the procedure RS-VEB-TREE-INSERT(V, x), which inserts x into the RS-vEB tree V, calling CREATE-NEW-RS-VEB-TREE as appropriate.

d. Modify the VEB-TREE-SUCCESSOR procedure to produce pseudocode for the procedure RS-VEB-TREE-SUCCESSOR(V, x), which returns the successor of x in RS-vEB tree V, or NIL if x has no successor in V.

e. Prove that, under the assumption of simple uniform hashing, your RS-VEB-TREE-INSERT and RS-VEB-TREE-SUCCESSOR procedures run in $O(\lg \lg u)$ expected time.

f. Assuming that elements are never deleted from a vEB tree, prove that the space requirement for the RS-vEB tree structure is $O(n)$, where n is the number of elements actually stored in the RS-vEB tree.

g. RS-vEB trees have another advantage over vEB trees: they require less time to create. How long does it take to create an empty RS-vEB tree?

20-2 y-fast tries

This problem investigates D. Willard's "y-fast tries" which, like van Emde Boas trees, perform each of the operations MEMBER, MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR on elements drawn from a universe with size u in $O(\lg \lg u)$ worst-case time. The INSERT and DELETE operations take $O(\lg \lg u)$ amortized time. Like reduced-space van Emde Boas trees (see Problem 20-1), y-fast tries use only $O(n)$ space to store n elements. The design of y-fast tries relies on perfect hashing (see Section 11.5).

As a preliminary structure, suppose that we create a perfect hash table containing not only every element in the dynamic set, but every prefix of the binary representation of every element in the set. For example, if $u = 16$, so that $\lg u = 4$, and $x = 13$ is in the set, then because the binary representation of 13 is 1101, the perfect hash table would contain the strings 1, 11, 110, and 1101. In addition to the hash table, we create a doubly linked list of the elements currently in the set, in increasing order.

a. How much space does this structure require?

b. Show how to perform the MINIMUM and MAXIMUM operations in $O(1)$ time; the MEMBER, PREDECESSOR, and SUCCESSOR operations in $O(\lg \lg u)$ time; and the INSERT and DELETE operations in $O(\lg u)$ time.

To reduce the space requirement to $O(n)$, we make the following changes to the data structure:

23.1-6

Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

23.1-7

Argue that if all edge weights of a graph are positive, then any subset of edges that connects all vertices and has minimum total weight must be a tree. Give an example to show that the same conclusion does not follow if we allow some weights to be nonpositive.

23.1-8

Let T be a minimum spanning tree of a graph G, and let L be the sorted list of the edge weights of T. Show that for any other minimum spanning tree T' of G, the list L is also the sorted list of edge weights of T'.

23.1-9

Let T be a minimum spanning tree of a graph $G = (V, E)$, and let V' be a subset of V. Let T' be the subgraph of T induced by V', and let G' be the subgraph of G induced by V'. Show that if T' is connected, then T' is a minimum spanning tree of G'.

23.1-10

Given a graph G and a minimum spanning tree T, suppose that we decrease the weight of one of the edges in T. Show that T is still a minimum spanning tree for G. More formally, let T be a minimum spanning tree for G with edge weights given by weight function w. Choose one edge $(x, y) \in T$ and a positive number k, and define the weight function w' by

$$w'(u, v) = \begin{cases} w(u, v) & \text{if } (u, v) \neq (x, y) , \\ w(x, y) - k & \text{if } (u, v) = (x, y) . \end{cases}$$

Show that T is a minimum spanning tree for G with edge weights given by w'.

23.1-11 ⋆

Given a graph G and a minimum spanning tree T, suppose that we decrease the weight of one of the edges not in T. Give an algorithm for finding the minimum spanning tree in the modified graph.

Therefore, any path p from v_0 to v_k has $\hat{w}(p) = w(p) + h(v_0) - h(v_k)$. Because $h(v_0)$ and $h(v_k)$ do not depend on the path, if one path from v_0 to v_k is shorter than another using weight function w, then it is also shorter using $\hat{w}$. Thus, $w(p) = \delta(v_0, v_k)$ if and only if $\hat{w}(p) = \hat{\delta}(v_0, v_k)$.

Finally, we show that G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function $\hat{w}$. Consider any cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$. By equation (25.10),

$$\begin{aligned} \hat{w}(c) &= w(c) + h(v_0) - h(v_k) \\ &= w(c) , \end{aligned}$$

and thus c has negative weight using w if and only if it has negative weight using $\hat{w}$. ■

Producing nonnegative weights by reweighting

Our next goal is to ensure that the second property holds: we want $\hat{w}(u, v)$ to be nonnegative for all edges $(u, v) \in E$. Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, we make a new graph $G' = (V', E')$, where $V' = V \cup \{s\}$ for some new vertex $s \notin V$ and $E' = E \cup \{(s, v) : v \in V\}$. We extend the weight function w so that $w(s, v) = 0$ for all $v \in V$. Note that because s has no edges that enter it, no shortest paths in G', other than those with source s, contain s. Moreover, G' has no negative-weight cycles if and only if G has no negative-weight cycles. Figure 25.6(a) shows the graph G' corresponding to the graph G of Figure 25.1.

Now suppose that G and G' have no negative-weight cycles. Let us define $h(v) = \delta(s, v)$ for all $v \in V'$. By the triangle inequality (Lemma 24.10), we have $h(v) \leq h(u) + w(u, v)$ for all edges $(u, v) \in E'$. Thus, if we define the new weights $\hat{w}$ by reweighting according to equation (25.9), we have $\hat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$, and we have satisfied the second property. Figure 25.6(b) shows the graph G' from Figure 25.6(a) with reweighted edges.

Computing all-pairs shortest paths

Johnson's algorithm to compute all-pairs shortest paths uses the Bellman-Ford algorithm (Section 24.1) and Dijkstra's algorithm (Section 24.3) as subroutines. It assumes implicitly that the edges are stored in adjacency lists. The algorithm returns the usual $|V| \times |V|$ matrix $D = d_{ij}$, where $d_{ij} = \delta(i, j)$, or it reports that the input graph contains a negative-weight cycle. As is typical for an all-pairs shortest-paths algorithm, we assume that the vertices are numbered from 1 to $|V|$.

this model, as well how the underlying concurrency platform can schedule computations efficiently.

Our model for dynamic multithreading offers several important advantages:

- It is a simple extension of our serial programming model. We can describe a multithreaded algorithm by adding to our pseudocode just three "concurrency" keywords: **parallel**, **spawn**, and **sync**. Moreover, if we delete these concurrency keywords from the multithreaded pseudocode, the resulting text is serial pseudocode for the same problem, which we call the "serialization" of the multithreaded algorithm.
- It provides a theoretically clean way to quantify parallelism based on the notions of "work" and "span."
- Many multithreaded algorithms involving nested parallelism follow naturally from the divide-and-conquer paradigm. Moreover, just as serial divide-and-conquer algorithms lend themselves to analysis by solving recurrences, so do multithreaded algorithms.
- The model is faithful to how parallel-computing practice is evolving. A growing number of concurrency platforms support one variant or another of dynamic multithreading, including Cilk [51, 118], Cilk++ [71], OpenMP [59], Task Parallel Library [230], and Threading Building Blocks [292].

Section 27.1 introduces the dynamic multithreading model and presents the metrics of work, span, and parallelism, which we shall use to analyze multithreaded algorithms. Section 27.2 investigates how to multiply matrices with multithreading, and Section 27.3 tackles the tougher problem of multithreading merge sort.

27.1 The basics of dynamic multithreading

We shall begin our exploration of dynamic multithreading using the example of computing Fibonacci numbers recursively. Recall that the Fibonacci numbers are defined by recurrence (3.22):

$$\begin{aligned} F_0 &= 0 , \\ F_1 &= 1 , \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2 . \end{aligned}$$

Here is a simple, recursive, serial algorithm to compute the nth Fibonacci number:

are ***linear inequalities***. We use the general term ***linear constraints*** to denote either linear equalities or linear inequalities. In linear programming, we do not allow strict inequalities. Formally, a ***linear-programming problem*** is the problem of either minimizing or maximizing a linear function subject to a finite set of linear constraints. If we are to minimize, then we call the linear program a ***minimization linear program***, and if we are to maximize, then we call the linear program a ***maximization linear program***.

The remainder of this chapter covers how to formulate and solve linear programs. Although several polynomial-time algorithms for linear programming have been developed, we will not study them in this chapter. Instead, we shall study the simplex algorithm, which is the oldest linear-programming algorithm. The simplex algorithm does not run in polynomial time in the worst case, but it is fairly efficient and widely used in practice.

An overview of linear programming

In order to describe properties of and algorithms for linear programs, we find it convenient to express them in canonical forms. We shall use two forms, ***standard*** and ***slack***, in this chapter. We will define them precisely in Section 29.1. Informally, a linear program in standard form is the maximization of a linear function subject to linear *inequalities*, whereas a linear program in slack form is the maximization of a linear function subject to linear *equalities*. We shall typically use standard form for expressing linear programs, but we find it more convenient to use slack form when we describe the details of the simplex algorithm. For now, we restrict our attention to maximizing a linear function on n variables subject to a set of m linear inequalities.

Let us first consider the following linear program with two variables:

$$\begin{aligned}
&\text{maximize} && x_1 + x_2 && && && && (29.11)\\
&\text{subject to} && && && && && \\
& && 4x_1 - x_2 && \le && 8 && (29.12)\\
& && 2x_1 + x_2 && \le && 10 && (29.13)\\
& && 5x_1 - 2x_2 && \ge && -2 && (29.14)\\
& && x_1, x_2 && \ge && 0\ . && (29.15)
\end{aligned}$$

We call any setting of the variables x_1 and x_2 that satisfies all the constraints (29.12)–(29.15) a ***feasible solution*** to the linear program. If we graph the constraints in the (x_1, x_2)-Cartesian coordinate system, as in Figure 29.2(a), we see

is a ***bit-reversal permutation***. That is, if we let $\text{rev}(k)$ be the $\lg n$-bit integer formed by reversing the bits of the binary representation of k, then we want to place vector element a_k in array position $A[\text{rev}(k)]$. In Figure 30.4, for example, the leaves appear in the order $0, 4, 2, 6, 1, 5, 3, 7$; this sequence in binary is $000, 100, 010, 110, 001, 101, 011, 111$, and when we reverse the bits of each value we get the sequence $000, 001, 010, 011, 100, 101, 110, 111$. To see that we want a bit-reversal permutation in general, we note that at the top level of the tree, indices whose low-order bit is 0 go into the left subtree and indices whose low-order bit is 1 go into the right subtree. Stripping off the low-order bit at each level, we continue this process down the tree, until we get the order given by the bit-reversal permutation at the leaves.

Since we can easily compute the function $\text{rev}(k)$, the BIT-REVERSE-COPY procedure is simple:

```
BIT-REVERSE-COPY(a, A)
1  n = a.length
2  for k = 0 to n − 1
3      A[rev(k)] = a_k
```

The iterative FFT implementation runs in time $\Theta(n \lg n)$. The call to BIT-REVERSE-COPY(a, A) certainly runs in $O(n \lg n)$ time, since we iterate n times and can reverse an integer between 0 and $n - 1$, with $\lg n$ bits, in $O(\lg n)$ time. (In practice, because we usually know the initial value of n in advance, we would probably code a table mapping k to $\text{rev}(k)$, making BIT-REVERSE-COPY run in $\Theta(n)$ time with a low hidden constant. Alternatively, we could use the clever amortized reverse binary counter scheme described in Problem 17-1.) To complete the proof that ITERATIVE-FFT runs in time $\Theta(n \lg n)$, we show that $L(n)$, the number of times the body of the innermost loop (lines 8–13) executes, is $\Theta(n \lg n)$. The **for** loop of lines 6–13 iterates $n/m = n/2^s$ times for each value of s, and the innermost loop of lines 8–13 iterates $m/2 = 2^{s-1}$ times. Thus,

$$\begin{aligned} L(n) &= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\ &= \sum_{s=1}^{\lg n} \frac{n}{2} \\ &= \Theta(n \lg n) \, . \end{aligned}$$

Note that the gap character may occur an arbitrary number of times in the pattern but not at all in the text. Give a polynomial-time algorithm to determine whether such a pattern P occurs in a given text T, and analyze the running time of your algorithm.

32.2 The Rabin-Karp algorithm

Rabin and Karp proposed a string-matching algorithm that performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching. The Rabin-Karp algorithm uses $\Theta(m)$ preprocessing time, and its worst-case running time is $\Theta((n-m+1)m)$. Based on certain assumptions, however, its average-case running time is better.

This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number. You might want to refer to Section 31.1 for the relevant definitions.

For expository purposes, let us assume that $\Sigma = \{0, 1, 2, \ldots, 9\}$, so that each character is a decimal digit. (In the general case, we can assume that each character is a digit in radix-d notation, where $d = |\Sigma|$.) We can then view a string of k consecutive characters as representing a length-k decimal number. The character string 31415 thus corresponds to the decimal number 31,415. Because we interpret the input characters as both graphical symbols and digits, we find it convenient in this section to denote them as we would digits, in our standard text font.

Given a pattern $P[1..m]$, let p denote its corresponding decimal value. In a similar manner, given a text $T[1..n]$, let t_s denote the decimal value of the length-m substring $T[s+1..s+m]$, for $s = 0, 1, \ldots, n-m$. Certainly, $t_s = p$ if and only if $T[s+1..s+m] = P[1..m]$; thus, s is a valid shift if and only if $t_s = p$. If we could compute p in time $\Theta(m)$ and all the t_s values in a total of $\Theta(n-m+1)$ time,[1] then we could determine all valid shifts s in time $\Theta(m) + \Theta(n-m+1) = \Theta(n)$ by comparing p with each of the t_s values. (For the moment, let's not worry about the possibility that p and the t_s values might be very large numbers.)

We can compute p in time $\Theta(m)$ using Horner's rule (see Section 30.1):

$$p = P[m] + 10\,(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots))\,.$$

Similarly, we can compute t_0 from $T[1..m]$ in time $\Theta(m)$.

[1] We write $\Theta(n-m+1)$ instead of $\Theta(n-m)$ because s takes on $n-m+1$ different values. The "+1" is significant in an asymptotic sense because when $m = n$, computing the lone t_s value takes $\Theta(1)$ time, not $\Theta(0)$ time.

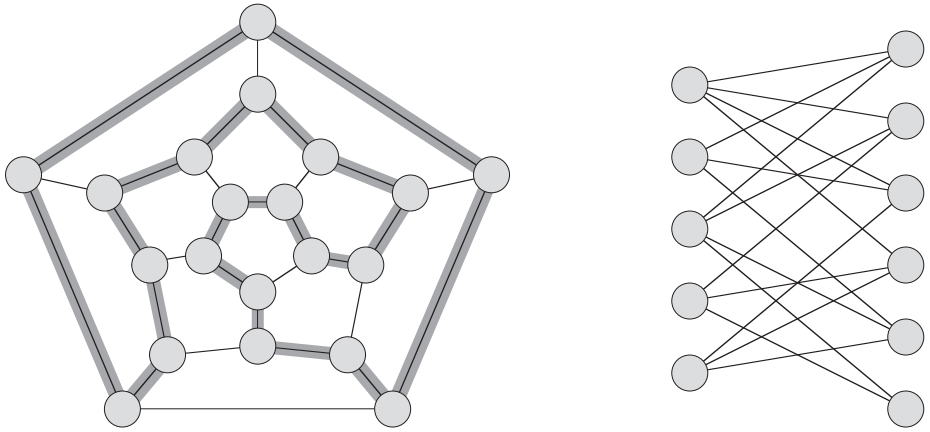

Figure 34.2 **(a)** A graph representing the vertices, edges, and faces of a dodecahedron, with a hamiltonian cycle shown by shaded edges. **(b)** A bipartite graph with an odd number of vertices. Any such graph is nonhamiltonian.

containing all the vertices.[7] The dodecahedron is hamiltonian, and Figure 34.2(a) shows one hamiltonian cycle. Not all graphs are hamiltonian, however. For example, Figure 34.2(b) shows a bipartite graph with an odd number of vertices. Exercise 34.2-2 asks you to show that all such graphs are nonhamiltonian.

We can define the ***hamiltonian-cycle problem***, "Does a graph G have a hamiltonian cycle?" as a formal language:

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ is a hamiltonian graph}\} .$$

How might an algorithm decide the language HAM-CYCLE? Given a problem instance $\langle G \rangle$, one possible decision algorithm lists all permutations of the vertices of G and then checks each permutation to see if it is a hamiltonian path. What is the running time of this algorithm? If we use the "reasonable" encoding of a graph as its adjacency matrix, the number m of vertices in the graph is $\Omega(\sqrt{n})$, where $n = |\langle G \rangle|$ is the length of the encoding of G. There are $m!$ possible permutations

[7] In a letter dated 17 October 1856 to his friend John T. Graves, Hamilton [157, p. 624] wrote, "I have found that some young persons have been much amused by trying a new mathematical game which the Icosion furnishes, one person sticking five pins in any five consecutive points ... and the other player then aiming to insert, which by the theory in this letter can always be done, fifteen other pins, in cyclical succession, so as to cover all the other points, and to end in immediate proximity to the pin wherewith his antagonist had begun."

35.5-4

How would you modify the approximation scheme presented in this section to find a good approximation to the smallest value not less than t that is a sum of some subset of the given input list?

35.5-5

Modify the APPROX-SUBSET-SUM procedure to also return the subset of S that sums to the value z^*.

Problems

35-1 Bin packing

Suppose that we are given a set of n objects, where the size s_i of the ith object satisfies $0 < s_i < 1$. We wish to pack all the objects into the minimum number of unit-size bins. Each bin can hold any subset of the objects whose total size does not exceed 1.

a. Prove that the problem of determining the minimum number of bins required is NP-hard. (*Hint:* Reduce from the subset-sum problem.)

The ***first-fit*** heuristic takes each object in turn and places it into the first bin that can accommodate it. Let $S = \sum_{i=1}^{n} s_i$.

b. Argue that the optimal number of bins required is at least $\lceil S \rceil$.

c. Argue that the first-fit heuristic leaves at most one bin less than half full.

d. Prove that the number of bins used by the first-fit heuristic is never more than $\lceil 2S \rceil$.

e. Prove an approximation ratio of 2 for the first-fit heuristic.

f. Give an efficient implementation of the first-fit heuristic, and analyze its running time.

35-2 Approximating the size of a maximum clique

Let $G = (V, E)$ be an undirected graph. For any $k \geq 1$, define $G^{(k)}$ to be the undirected graph $(V^{(k)}, E^{(k)})$, where $V^{(k)}$ is the set of all ordered k-tuples of vertices from V and $E^{(k)}$ is defined so that $(v_1, v_2, \ldots, v_k)$ is adjacent to $(w_1, w_2, \ldots, w_k)$ if and only if for $i = 1, 2, \ldots, k$, either vertex v_i is adjacent to w_i in G, or else $v_i = w_i$.

$$\begin{aligned}\frac{b(k;n,p)}{b(k-1;n,p)} &= \frac{\binom{n}{k}p^k q^{n-k}}{\binom{n}{k-1}p^{k-1}q^{n-k+1}} \\ &= \frac{n!(k-1)!(n-k+1)!p}{k!(n-k)!n!q} \\ &= \frac{(n-k+1)p}{kq} \\ &= 1+\frac{(n+1)p-k}{kq}\,.\end{aligned} \tag{C.41}$$

This ratio is greater than 1 precisely when $(n+1)p-k$ is positive. Consequently, $b(k;n,p) > b(k-1;n,p)$ for $k < (n+1)p$ (the distribution increases), and $b(k;n,p) < b(k-1;n,p)$ for $k > (n+1)p$ (the distribution decreases). If $k = (n+1)p$ is an integer, then $b(k;n,p) = b(k-1;n,p)$, and so the distribution then has two maxima: at $k = (n+1)p$ and at $k-1 = (n+1)p-1 = np-q$. Otherwise, it attains a maximum at the unique integer k that lies in the range $np-q < k < (n+1)p$.

The following lemma provides an upper bound on the binomial distribution.

Lemma C.1
Let $n \geq 0$, let $0 < p < 1$, let $q = 1-p$, and let $0 \leq k \leq n$. Then

$$b(k;n,p) \leq \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}\,.$$

Proof Using equation (C.6), we have

$$\begin{aligned}b(k;n,p) &= \binom{n}{k}p^k q^{n-k} \\ &\leq \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k} \\ &= \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}\,.\end{aligned}$$

■

Exercises

C.4-1
Verify axiom 2 of the probability axioms for the geometric distribution.

C.4-2
How many times on average must we flip 6 fair coins before we obtain 3 heads and 3 tails?