where $b_n = g(n+1)Q(n+1)a_n$, with

$$Q(n) = (f(1)f(2)\cdots f(n-1))/(g(1)g(2)\cdots g(n)).$$

b) Use part (a) to solve the original recurrence relation to obtain

$$a_n = \frac{C + \sum_{i=1}^{n} Q(i)h(i)}{g(n+1)Q(n+1)}.$$

***49.** Use Exercise 48 to solve the recurrence relation $(n+1)a_n = (n+3)a_{n-1} + n$, for $n \geq 1$, with $a_0 = 1$.

50. It can be shown that C_n, the average number of comparisons made by the quick sort algorithm (described in preamble to Exercise 50 in Section 5.4), when sorting n elements in random order, satisfies the recurrence relation

$$C_n = n + 1 + \frac{2}{n}\sum_{k=0}^{n-1} C_k$$

for $n = 1, 2, \ldots$, with initial condition $C_0 = 0$.

a) Show that $\{C_n\}$ also satisfies the recurrence relation $nC_n = (n+1)C_{n-1} + 2n$ for $n = 1, 2, \ldots$.

b) Use Exercise 48 to solve the recurrence relation in part (a) to find an explicit formula for C_n.

****51.** Prove Theorem 4.

****52.** Prove Theorem 6.

53. Solve the recurrence relation $T(n) = nT^2(n/2)$ with initial condition $T(1) = 6$ when $n = 2^k$ for some integer k. [*Hint:* Let $n = 2^k$ and then make the substitution $a_k = \log T(2^k)$ to obtain a linear nonhomogeneous recurrence relation.]

8.3 Divide-and-Conquer Algorithms and Recurrence Relations

Introduction

Links

Many recursive algorithms take a problem with a given input and divide it into one or more smaller problems. This reduction is successively applied until the solutions of the smaller problems can be found quickly. For instance, we perform a binary search by reducing the search for an element in a list to the search for this element in a list half as long. We successively apply this reduction until one element is left. When we sort a list of integers using the merge sort, we split the list into two halves of equal size and sort each half separately. We then merge the two sorted halves. Another example of this type of recursive algorithm is a procedure for multiplying integers that reduces the problem of the multiplication of two integers to three multiplications of pairs of integers with half as many bits. This reduction is successively applied until integers with one bit are obtained. These procedures follow an important algorithmic paradigm known as **divide-and-conquer**, and are called **divide-and-conquer algorithms**, because they *divide* a problem into one or more instances of the same problem of smaller size and they *conquer* the problem by using the solutions of the smaller problems to find a solution of the original problem, perhaps with some additional work.

"Divide et impera" (translation: *"Divide and conquer"* - Julius Caesar

In this section we will show how recurrence relations can be used to analyze the computational complexity of divide-and-conquer algorithms. We will use these recurrence relations to estimate the number of operations used by many different divide-and-conquer algorithms, including several that we introduce in this section.

Divide-and-Conquer Recurrence Relations

Suppose that a recursive algorithm divides a problem of size n into a subproblems, where each subproblem is of size n/b (for simplicity, assume that n is a multiple of b; in reality, the smaller problems are often of size equal to the nearest integers either less than or equal to, or greater than or equal to, n/b). Also, suppose that a total of $g(n)$ extra operations are required in the conquer step of the algorithm to combine the solutions of the subproblems into a solution of the original problem. Then, if $f(n)$ represents the number of operations required to solve the problem of size n, it follows that f satisfies the recurrence relation

$$f(n) = af(n/b) + g(n).$$

This is called a **divide-and-conquer recurrence relation**.

We will first set up the divide-and-conquer recurrence relations that can be used to study the complexity of some important algorithms. Then we will show how to use these divide-and-conquer recurrence relations to estimate the complexity of these algorithms.

EXAMPLE 1 **Binary Search** We introduced a binary search algorithm in Section 3.1. This binary search algorithm reduces the search for an element in a search sequence of size n to the binary search for this element in a search sequence of size $n/2$, when n is even. (Hence, the problem of size n has been reduced to *one* problem of size $n/2$.) Two comparisons are needed to implement this reduction (one to determine which half of the list to use and the other to determine whether any terms of the list remain). Hence, if $f(n)$ is the number of comparisons required to search for an element in a search sequence of size n, then

$$f(n) = f(n/2) + 2$$

when n is even. ◀

EXAMPLE 2 **Finding the Maximum and Minimum of a Sequence** Consider the following algorithm for locating the maximum and minimum elements of a sequence $a_1, a_2, \ldots, a_n$. If $n = 1$, then a_1 is the maximum and the minimum. If $n > 1$, split the sequence into two sequences, either where both have the same number of elements or where one of the sequences has one more element than the other. The problem is reduced to finding the maximum and minimum of each of the two smaller sequences. The solution to the original problem results from the comparison of the separate maxima and minima of the two smaller sequences to obtain the overall maximum and minimum.

Let $f(n)$ be the total number of comparisons needed to find the maximum and minimum elements of the sequence with n elements. We have shown that a problem of size n can be reduced into two problems of size $n/2$, when n is even, using two comparisons, one to compare the maxima of the two sequences and the other to compare the minima of the two sequences. This gives the recurrence relation

$$f(n) = 2f(n/2) + 2$$

when n is even. ◀

EXAMPLE 3 **Merge Sort** The merge sort algorithm (introduced in Section 5.4) splits a list to be sorted with n items, where n is even, into two lists with $n/2$ elements each, and uses fewer than n comparisons to merge the two sorted lists of $n/2$ items each into one sorted list. Consequently, the number of comparisons used by the merge sort to sort a list of n elements is less than $M(n)$, where the function $M(n)$ satisfies the divide-and-conquer recurrence relation

$$M(n) = 2M(n/2) + n.$$ ◀

EXAMPLE 4 **Fast Multiplication of Integers** Surprisingly, there are more efficient algorithms than the conventional algorithm (described in Section 4.2) for multiplying integers. One of these algorithms, which uses a divide-and-conquer technique, will be described here. This fast multiplication algorithm proceeds by splitting each of two $2n$-bit integers into two blocks, each with n bits. Then, the original multiplication is reduced from the multiplication of two $2n$-bit integers to three multiplications of n-bit integers, plus shifts and additions.

Suppose that a and b are integers with binary expansions of length $2n$ (add initial bits of zero in these expansions if necessary to make them the same length). Let

$$a = (a_{2n-1}a_{2n-2}\cdots a_1a_0)_2 \quad \text{and} \quad b = (b_{2n-1}b_{2n-2}\cdots b_1b_0)_2.$$

Let

$$a = 2^n A_1 + A_0, \quad b = 2^n B_1 + B_0,$$

where

$$A_1 = (a_{2n-1} \cdots a_{n+1} a_n)_2, \qquad A_0 = (a_{n-1} \cdots a_1 a_0)_2,$$
$$B_1 = (b_{2n-1} \cdots b_{n+1} b_n)_2, \qquad B_0 = (b_{n-1} \cdots b_1 b_0)_2.$$

The algorithm for fast multiplication of integers is based on the fact that ab can be rewritten as

$$ab = (2^{2n} + 2^n)A_1 B_1 + 2^n (A_1 - A_0)(B_0 - B_1) + (2^n + 1)A_0 B_0.$$

The important fact about this identity is that it shows that the multiplication of two $2n$-bit integers can be carried out using three multiplications of n-bit integers, together with additions, subtractions, and shifts. This shows that if $f(n)$ is the total number of bit operations needed to multiply two n-bit integers, then

$$f(2n) = 3f(n) + Cn.$$

The reasoning behind this equation is as follows. The three multiplications of n-bit integers are carried out using $3f(n)$-bit operations. Each of the additions, subtractions, and shifts uses a constant multiple of n-bit operations, and Cn represents the total number of bit operations used by these operations. ◀

EXAMPLE 5 **Fast Matrix Multiplication** In Example 7 of Section 3.3 we showed that multiplying two $n \times n$ matrices using the definition of matrix multiplication required n^3 multiplications and $n^2(n-1)$ additions. Consequently, computing the product of two $n \times n$ matrices in this way requires $O(n^3)$ operations (multiplications and additions). Surprisingly, there are more efficient divide-and-conquer algorithms for multiplying two $n \times n$ matrices. Such an algorithm, invented by Volker Strassen in 1969, reduces the multiplication of two $n \times n$ matrices, when n is even, to seven multiplications of two $(n/2) \times (n/2)$ matrices and 15 additions of $(n/2) \times (n/2)$ matrices. (See [CoLeRiSt09] for the details of this algorithm.) Hence, if $f(n)$ is the number of operations (multiplications and additions) used, it follows that

$$f(n) = 7f(n/2) + 15n^2/4$$

when n is even. ◀

As Examples 1–5 show, recurrence relations of the form $f(n) = af(n/b) + g(n)$ arise in many different situations. It is possible to derive estimates of the size of functions that satisfy such recurrence relations. Suppose that f satisfies this recurrence relation whenever n is divisible by b. Let $n = b^k$, where k is a positive integer. Then

$$\begin{aligned} f(n) &= af(n/b) + g(n) \\ &= a^2 f(n/b^2) + ag(n/b) + g(n) \\ &= a^3 f(n/b^3) + a^2 g(n/b^2) + ag(n/b) + g(n) \\ &\vdots \\ &= a^k f(n/b^k) + \sum_{j=0}^{k-1} a^j g(n/b^j). \end{aligned}$$

Because $n/b^k = 1$, it follows that

$$f(n) = a^k f(1) + \sum_{j=0}^{k-1} a^j g(n/b^j).$$

We can use this equation for $f(n)$ to estimate the size of functions that satisfy divide-and-conquer relations.

THEOREM 1

Let f be an increasing function that satisfies the recurrence relation

$$f(n) = af(n/b) + c$$

whenever n is divisible by b, where $a \geq 1$, b is an integer greater than 1, and c is a positive real number. Then

$$f(n) \text{ is } \begin{cases} O(n^{\log_b a}) & \text{if } a > 1, \\ O(\log n) & \text{if } a = 1. \end{cases}$$

Furthermore, when $n = b^k$ and $a \neq 1$, where k is a positive integer,

$$f(n) = C_1 n^{\log_b a} + C_2,$$

where $C_1 = f(1) + c/(a-1)$ and $C_2 = -c/(a-1)$.

Proof: First let $n = b^k$. From the expression for $f(n)$ obtained in the discussion preceding the theorem, with $g(n) = c$, we have

$$f(n) = a^k f(1) + \sum_{j=0}^{k-1} a^j c = a^k f(1) + c \sum_{j=0}^{k-1} a^j.$$

When $a = 1$ we have

$$f(n) = f(1) + ck\,.$$

Because $n = b^k$, we have $k = \log_b n$. Hence,

$$f(n) = f(1) + c \log_b n\,.$$

When n is not a power of b, we have $b^k < n < b^{k+1}$, for a positive integer k. Because f is increasing, it follows that $f(n) \leq f(b^{k+1}) = f(1) + c(k+1) = (f(1) + c) + ck \leq (f(1) + c) + c \log_b n$. Therefore, in both cases, $f(n)$ is $O(\log n)$ when $a = 1$.

Now suppose that $a > 1$. First assume that $n = b^k$, where k is a positive integer. From the formula for the sum of terms of a geometric progression (Theorem 1 in Section 2.4), it follows that

$$\begin{aligned} f(n) &= a^k f(1) + c(a^k - 1)/(a-1) \\ &= a^k [f(1) + c/(a-1)] - c/(a-1) \\ &= C_1 n^{\log_b a} + C_2, \end{aligned}$$

because $a^k = a^{\log_b n} = n^{\log_b a}$ (see Exercise 4 in Appendix 2), where $C_1 = f(1) + c/(a-1)$ and $C_2 = -c/(a-1)$.

Now suppose that n is not a power of b. Then $b^k < n < b^{k+1}$, where k is a nonnegative integer. Because f is increasing,

$$\begin{aligned} f(n) &\leq f(b^{k+1}) = C_1 a^{k+1} + C_2 \\ &\leq (C_1 a) a^{\log_b n} + C_2 \\ &= (C_1 a) n^{\log_b a} + C_2, \end{aligned}$$

because $k \leq \log_b n < k+1$.

Hence, we have $f(n)$ is $O(n^{\log_b a})$. ◁

Examples 6–9 illustrate how Theorem 1 is used.

EXAMPLE 6 Let $f(n) = 5f(n/2) + 3$ and $f(1) = 7$. Find $f(2^k)$, where k is a positive integer. Also, estimate $f(n)$ if f is an increasing function.

Extra Examples

Solution: From the proof of Theorem 1, with $a = 5$, $b = 2$, and $c = 3$, we see that if $n = 2^k$, then

$$\begin{aligned} f(n) &= a^k[f(1) + c/(a-1)] + [-c/(a-1)] \\ &= 5^k[7 + (3/4)] - 3/4 \\ &= 5^k(31/4) - 3/4. \end{aligned}$$

Also, if $f(n)$ is increasing, Theorem 1 shows that $f(n)$ is $O(n^{\log_b a}) = O(n^{\log 5})$. ◀

We can use Theorem 1 to estimate the computational complexity of the binary search algorithm and the algorithm given in Example 2 for locating the minimum and maximum of a sequence.

EXAMPLE 7 Give a big-O estimate for the number of comparisons used by a binary search.

Solution: In Example 1 it was shown that $f(n) = f(n/2) + 2$ when n is even, where f is the number of comparisons required to perform a binary search on a sequence of size n. Hence, from Theorem 1, it follows that $f(n)$ is $O(\log n)$. ◀

EXAMPLE 8 Give a big-O estimate for the number of comparisons used to locate the maximum and minimum elements in a sequence using the algorithm given in Example 2.

Solution: In Example 2 we showed that $f(n) = 2f(n/2) + 2$, when n is even, where f is the number of comparisons needed by this algorithm. Hence, from Theorem 1, it follows that $f(n)$ is $O(n^{\log 2}) = O(n)$. ◀

We now state a more general, and more complicated, theorem, which has Theorem 1 as a special case. This theorem (or more powerful versions, including big-Theta estimates) is sometimes known as the master theorem because it is useful in analyzing the complexity of many important divide-and-conquer algorithms.

THEOREM 2 **MASTER THEOREM** Let f be an increasing function that satisfies the recurrence relation

$$f(n) = af(n/b) + cn^d$$

whenever $n = b^k$, where k is a positive integer, $a \geq 1$, b is an integer greater than 1, and c and d are real numbers with c positive and d nonnegative. Then

$$f(n) \text{ is } \begin{cases} O(n^d) & \text{if } a < b^d, \\ O(n^d \log n) & \text{if } a = b^d, \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

The proof of Theorem 2 is left for the reader as Exercises 29–33.

EXAMPLE 9 **Complexity of Merge Sort** In Example 3 we explained that the number of comparisons used by the merge sort to sort a list of n elements is less than $M(n)$, where $M(n) = 2M(n/2) + n$. By the master theorem (Theorem 2) we find that $M(n)$ is $O(n \log n)$, which agrees with the estimate found in Section 5.4. ◀

EXAMPLE 10 Give a big-O estimate for the number of bit operations needed to multiply two n-bit integers using the fast multiplication algorithm described in Example 4.

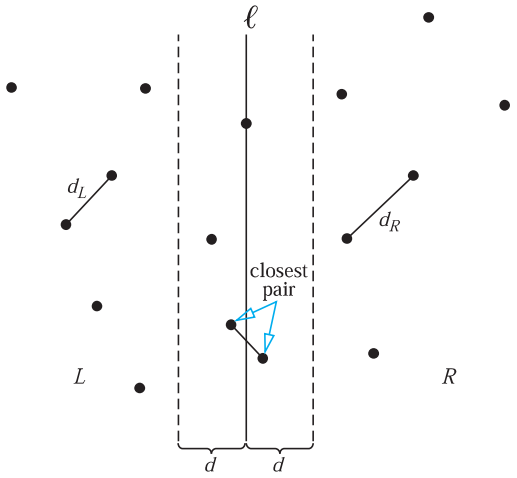
Solution: Example 4 shows that $f(n) = 3f(n/2) + Cn$, when n is even, where $f(n)$ is the number of bit operations required to multiply two n-bit integers using the fast multiplication algorithm. Hence, from the master theorem (Theorem 2), it follows that $f(n)$ is $O(n^{\log 3})$. Note that $\log 3 \sim 1.6$. Because the conventional algorithm for multiplication uses $O(n^2)$ bit operations, the fast multiplication algorithm is a substantial improvement over the conventional algorithm in terms of time complexity for sufficiently large integers, including large integers that occur in practical applications. ◀

EXAMPLE 11 Give a big-O estimate for the number of multiplications and additions required to multiply two $n \times n$ matrices using the matrix multiplication algorithm referred to in Example 5.

Solution: Let $f(n)$ denote the number of additions and multiplications used by the algorithm mentioned in Example 5 to multiply two $n \times n$ matrices. We have $f(n) = 7f(n/2) + 15n^2/4$, when n is even. Hence, from the master theorem (Theorem 2), it follows that $f(n)$ is $O(n^{\log 7})$. Note that $\log 7 \sim 2.8$. Because the conventional algorithm for multiplying two $n \times n$ matrices uses $O(n^3)$ additions and multiplications, it follows that for sufficiently large integers n, including those that occur in many practical applications, this algorithm is substantially more efficient in time complexity than the conventional algorithm. ◀

THE CLOSEST-PAIR PROBLEM We conclude this section by introducing a divide-and-conquer algorithm from computational geometry, the part of discrete mathematics devoted to algorithms that solve geometric problems.

EXAMPLE 12 Links **The Closest-Pair Problem** Consider the problem of determining the closest pair of points in a set of n points $(x_1, y_1), \ldots, (x_n, y_n)$ in the plane, where the distance between two points (x_i, y_i) and (x_j, y_j) is the usual Euclidean distance $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. This problem arises in many applications such as determining the closest pair of airplanes in the air space at a particular altitude being managed by an air traffic controller. How can this closest pair of points be found in an efficient way?



In this illustration the problem of finding the closest pair in a set of 16 points is reduced to two problems of finding the closest pair in a set of eight points *and* the problem of determining whether there are points closer than $d = \min(d_L, d_R)$ within the strip of width $2d$ centered at ℓ.

FIGURE 1 **The Recursive Step of the Algorithm for Solving the Closest-Pair Problem.**

It took researchers more than 10 year to find an algorithm with $O(n \log n)$ complexity that locates the closest pair of points among n points.

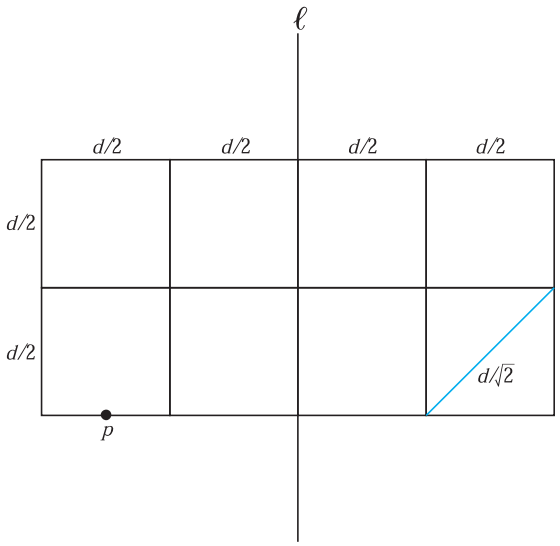
Solution: To solve this problem we can first determine the distance between every pair of points and then find the smallest of these distances. However, this approach requires $O(n^2)$ computations of distances and comparisons because there are $C(n, 2) = n(n-1)/2$ pairs of points. Surprisingly, there is an elegant divide-and-conquer algorithm that can solve the closest-pair problem for n points using $O(n \log n)$ computations of distances and comparisons. The algorithm we describe here is due to Michael Samos (see [PrSa85]).

For simplicity, we assume that $n = 2^k$, where k is a positive integer. (We avoid some technical considerations that are needed when n is not a power of 2.) When $n = 2$, we have only one pair of points; the distance between these two points is the minimum distance. At the start of the algorithm we use the merge sort twice, once to sort the points in order of increasing x coordinates, and once to sort the points in order of increasing y coordinates. Each of these sorts requires $O(n \log n)$ operations. We will use these sorted lists in each recursive step.

The recursive part of the algorithm divides the problem into two subproblems, each involving half as many points. Using the sorted list of the points by their x coordinates, we construct a vertical line ℓ dividing the n points into two parts, a left part and a right part of equal size, each containing $n/2$ points, as shown in Figure 1. (If any points fall on the dividing line ℓ, we divide them among the two parts if necessary.) At subsequent steps of the recursion we need not sort on x coordinates again, because we can select the corresponding sorted subset of all the points. This selection is a task that can be done with $O(n)$ comparisons.

There are three possibilities concerning the positions of the closest points: (1) they are both in the left region L, (2) they are both in the right region R, or (3) one point is in the left region and the other is in the right region. Apply the algorithm recursively to compute d_L and d_R, where d_L is the minimum distance between points in the left region and d_R is the minimum distance between points in the right region. Let $d = \min(d_L, d_R)$. To successfully divide the problem of finding the closest two points in the original set into the two problems of finding the shortest distances between points in the two regions separately, we have to handle the conquer part of the algorithm, which requires that we consider the case where the closest points lie in different regions, that is, one point is in L and the other in R. Because there is a pair of points at distance d where both points lie in R or both points lie in L, for the closest points to lie in different regions requires that they must be a distance less than d apart.

For a point in the left region and a point in the right region to lie at a distance less than d apart, these points must lie in the vertical strip of width $2d$ that has the line ℓ as its center. (Otherwise, the distance between these points is greater than the difference in their x coordinates, which exceeds d.) To examine the points within this strip, we sort the points so that they are listed in order of increasing y coordinates, using the sorted list of the points by their y coordinates. At



At most eight points, including p, can lie in or on the $2d \times d$ rectangle centered at ℓ because at most one point can lie in or on each of the eight $(d/2) \times (d/2)$ squares.

FIGURE 2 Showing That There Are at Most Seven Other Points to Consider for Each Point in the Strip.

each recursive step, we form a subset of the points in the region sorted by their y coordinates from the already sorted set of all points sorted by their y coordinates, which can be done with $O(n)$ comparisons.

Beginning with a point in the strip with the smallest y coordinate, we successively examine each point in the strip, computing the distance between this point and all other points in the strip that have larger y coordinates that could lie at a distance less than d from this point. Note that to examine a point p, we need only consider the distances between p and points in the set that lie within the rectangle of height d and width $2d$ with p on its base and with vertical sides at distance d from ℓ.

We can show that there are at most eight points from the set, including p, in or on this $2d \times d$ rectangle. To see this, note that there can be at most one point in each of the eight $d/2 \times d/2$ squares shown in Figure 2. This follows because the farthest apart points can be on or within one of these squares is the diagonal length $d/\sqrt{2}$ (which can be found using the Pythagorean theorem), which is less than d, and each of these $d/2 \times d/2$ squares lies entirely within the left region or the right region. This means that at this stage we need only compare at most seven distances, the distances between p and the seven or fewer other points in or on the rectangle, with d.

Because the total number of points in the strip of width $2d$ does not exceed n (the total number of points in the set), at most $7n$ distances need to be compared with d to find the minimum distance between points. That is, there are only $7n$ possible distances that could be less than d. Consequently, once the merge sort has been used to sort the pairs according to their x coordinates and according to their y coordinates, we find that the increasing function $f(n)$ satisfying the recurrence relation

$$f(n) = 2f(n/2) + 7n,$$

where $f(2) = 1$, exceeds the number of comparisons needed to solve the closest-pair problem for n points. By the master theorem (Theorem 2), it follows that $f(n)$ is $O(n \log n)$. The two sorts of points by their x coordinates and by their y coordinates each can be done using $O(n \log n)$ comparisons, by using the merge sort, and the sorted subsets of these coordinates at each of the $O(\log n)$ steps of the algorithm can be done using $O(n)$ comparisons each. Thus, we find that the closest-pair problem can be solved using $O(n \log n)$ comparisons. ◀

Exercises

1. How many comparisons are needed for a binary search in a set of 64 elements?
2. How many comparisons are needed to locate the maximum and minimum elements in a sequence with 128 elements using the algorithm in Example 2?
3. Multiply $(1110)_2$ and $(1010)_2$ using the fast multiplication algorithm.
4. Express the fast multiplication algorithm in pseudocode.
5. Determine a value for the constant C in Example 4 and use it to estimate the number of bit operations needed to multiply two 64-bit integers using the fast multiplication algorithm.
6. How many operations are needed to multiply two 32×32 matrices using the algorithm referred to in Example 5?
7. Suppose that $f(n) = f(n/3) + 1$ when n is a positive integer divisible by 3, and $f(1) = 1$. Find
   a) $f(3)$. **b)** $f(27)$. **c)** $f(729)$.
8. Suppose that $f(n) = 2f(n/2) + 3$ when n is an even positive integer, and $f(1) = 5$. Find
   a) $f(2)$. **b)** $f(8)$. **c)** $f(64)$. **d)** $f(1024)$.
9. Suppose that $f(n) = f(n/5) + 3n^2$ when n is a positive integer divisible by 5, and $f(1) = 4$. Find
   a) $f(5)$. **b)** $f(125)$. **c)** $f(3125)$.
10. Find $f(n)$ when $n = 2^k$, where f satisfies the recurrence relation $f(n) = f(n/2) + 1$ with $f(1) = 1$.
11. Give a big-O estimate for the function f in Exercise 10 if f is an increasing function.
12. Find $f(n)$ when $n = 3^k$, where f satisfies the recurrence relation $f(n) = 2f(n/3) + 4$ with $f(1) = 1$.
13. Give a big-O estimate for the function f in Exercise 12 if f is an increasing function.
14. Suppose that there are $n = 2^k$ teams in an elimination tournament, where there are $n/2$ games in the first round, with the $n/2 = 2^{k-1}$ winners playing in the second round, and so on. Develop a recurrence relation for the number of rounds in the tournament.
15. How many rounds are in the elimination tournament described in Exercise 14 when there are 32 teams?
16. Solve the recurrence relation for the number of rounds in the tournament described in Exercise 14.
17. Suppose that the votes of n people for different candidates (where there can be more than two candidates) for a particular office are the elements of a sequence. A person wins the election if this person receives a majority of the votes.
    a) Devise a divide-and-conquer algorithm that determines whether a candidate received a majority and, if so, determine who this candidate is. [*Hint:* Assume that n is even and split the sequence of votes into two sequences, each with $n/2$ elements. Note that a candidate could not have received a majority of votes without receiving a majority of votes in at least one of the two halves.]
    b) Use the master theorem to give a big-O estimate for the number of comparisons needed by the algorithm you devised in part (a).
18. Suppose that each person in a group of n people votes for exactly two people from a slate of candidates to fill two positions on a committee. The top two finishers both win positions as long as each receives more than $n/2$ votes.
    a) Devise a divide-and-conquer algorithm that determines whether the two candidates who received the most votes each received at least $n/2$ votes and, if so, determine who these two candidates are.
    b) Use the master theorem to give a big-O estimate for the number of comparisons needed by the algorithm you devised in part (a).
19. **a)** Set up a divide-and-conquer recurrence relation for the number of multiplications required to compute x^n, where x is a real number and n is a positive integer, using the recursive algorithm from Exercise 26 in Section 5.4.
    b) Use the recurrence relation you found in part (a) to construct a big-O estimate for the number of multiplications used to compute x^n using the recursive algorithm.
20. **a)** Set up a divide-and-conquer recurrence relation for the number of modular multiplications required to compute a^n **mod** m, where a, m, and n are positive integers, using the recursive algorithms from Example 4 in Section 5.4.
    b) Use the recurrence relation you found in part (a) to construct a big-O estimate for the number of modular multiplications used to compute a^n **mod** m using the recursive algorithm.
21. Suppose that the function f satisfies the recurrence relation $f(n) = 2f(\sqrt{n}) + 1$ whenever n is a perfect square greater than 1 and $f(2) = 1$.
    a) Find $f(16)$.
    b) Give a big-O estimate for $f(n)$. [*Hint:* Make the substitution $m = \log n$.]
22. Suppose that the function f satisfies the recurrence relation $f(n) = 2f(\sqrt{n}) + \log n$ whenever n is a perfect square greater than 1 and $f(2) = 1$.
    a) Find $f(16)$.
    b) Find a big-O estimate for $f(n)$. [*Hint:* Make the substitution $m = \log n$.]

****23.** This exercise deals with the problem of finding the largest sum of consecutive terms of a sequence of n real numbers. When all terms are positive, the sum of all terms provides

the answer, but the situation is more complicated when some terms are negative. For example, the maximum sum of consecutive terms of the sequence $-2, 3, -1, 6, -7, 4$ is $3 + (-1) + 6 = 8$. (This exercise is based on [Be86].) Recall that in Exercise 56 in Section 8.1 we developed a dynamic programming algorithm for solving this problem. Here, we first look at the brute-force algorithm for solving this problem; then we develop a divide-and-conquer algorithm for solving it.

a) Use pseudocode to describe an algorithm that solves this problem by finding the sums of consecutive terms starting with the first term, the sums of consecutive terms starting with the second term, and so on, keeping track of the maximum sum found so far as the algorithm proceeds.

b) Determine the computational complexity of the algorithm in part (a) in terms of the number of sums computed and the number of comparisons made.

c) Devise a divide-and-conquer algorithm to solve this problem. [*Hint:* Assume that there are an even number of terms in the sequence and split the sequence into two halves. Explain how to handle the case when the maximum sum of consecutive terms includes terms in both halves.]

d) Use the algorithm from part (c) to find the maximum sum of consecutive terms of each of the sequences: $-2, 4, -1, 3, 5, -6, 1, 2$; $4, 1, -3, 7, -1, -5, \quad 3, -2$; and $-1, 6, 3, -4, -5, 8, -1, 7$.

e) Find a recurrence relation for the number of sums and comparisons used by the divide-and-conquer algorithm from part (c).

f) Use the master theorem to estimate the computational complexity of the divide-and-conquer algorithm. How does it compare in terms of computational complexity with the algorithm from part (a)?

24. Apply the algorithm described in Example 12 for finding the closest pair of points, using the Euclidean distance between points, to find the closest pair of the points $(1, 3)$, $(1, 7)$, $(2, 4)$, $(2, 9)$, $(3, 1)$, $(3, 5)$, $(4, 3)$, and $(4, 7)$.

25. Apply the algorithm described in Example 12 for finding the closest pair of points, using the Euclidean distance between points, to find the closest pair of the points $(1, 2)$, $(1, 6)$, $(2, 4)$, $(2, 8)$, $(3, 1)$, $(3, 6)$, $(3, 10)$, $(4, 3)$, $(5, 1)$, $(5, 5)$, $(5, 9)$, $(6, 7)$, $(7, 1)$, $(7, 4)$, $(7, 9)$, and $(8, 6)$.

***26.** Use pseudocode to describe the recursive algorithm for solving the closest-pair problem as described in Example 12.

27. Construct a variation of the algorithm described in Example 12 along with justifications of the steps used by the algorithm to find the smallest distance between two points if the distance between two points is defined to be $d((x_i, y_i), (x_j, y_j)) = \max(|x_i - x_j|, |y_i - y_j|)$.

***28.** Suppose someone picks a number x from a set of n numbers. A second person tries to guess the number by successively selecting subsets of the n numbers and asking the first person whether x is in each set. The first person answers either "yes" or "no." When the first person answers each query truthfully, we can find x using $\log n$ queries by successively splitting the sets used in each query in half. Ulam's problem, proposed by Stanislaw Ulam in 1976, asks for the number of queries required to find x, supposing that the first person is allowed to lie exactly once.

a) Show that by asking each question twice, given a number x and a set with n elements, and asking one more question when we find the lie, Ulam's problem can be solved using $2 \log n + 1$ queries.

b) Show that by dividing the initial set of n elements into four parts, each with $n/4$ elements, 1/4 of the elements can be eliminated using two queries. [*Hint:* Use two queries, where each of the queries asks whether the element is in the union of two of the subsets with $n/4$ elements and where one of the subsets of $n/4$ elements is used in both queries.]

c) Show from part (b) that if $f(n)$ equals the number of queries used to solve Ulam's problem using the method from part (b) and n is divisible by 4, then $f(n) = f(3n/4) + 2$.

d) Solve the recurrence relation in part (c) for $f(n)$.

e) Is the naive way to solve Ulam's problem by asking each question twice or the divide-and-conquer method based on part (b) more efficient? The most efficient way to solve Ulam's problem has been determined by A. Pelc [Pe87].

In Exercises 29–33, assume that f is an increasing function satisfying the recurrence relation $f(n) = af(n/b) + cn^d$, where $a \geq 1$, b is an integer greater than 1, and c and d are positive real numbers. These exercises supply a proof of Theorem 2.

***29.** Show that if $a = b^d$ and n is a power of b, then $f(n) = f(1)n^d + cn^d \log_b n$.

30. Use Exercise 29 to show that if $a = b^d$, then $f(n)$ is $O(n^d \log n)$.

***31.** Show that if $a \neq b^d$ and n is a power of b, then $f(n) = C_1 n^d + C_2 n^{\log_b a}$, where $C_1 = b^d c/(b^d - a)$ and $C_2 = f(1) + b^d c/(a - b^d)$.

32. Use Exercise 31 to show that if $a < b^d$, then $f(n)$ is $O(n^d)$.

33. Use Exercise 31 to show that if $a > b^d$, then $f(n)$ is $O(n^{\log_b a})$.

34. Find $f(n)$ when $n = 4^k$, where f satisfies the recurrence relation $f(n) = 5f(n/4) + 6n$, with $f(1) = 1$.

35. Give a big-O estimate for the function f in Exercise 34 if f is an increasing function.

36. Find $f(n)$ when $n = 2^k$, where f satisfies the recurrence relation $f(n) = 8f(n/2) + n^2$ with $f(1) = 1$.

37. Give a big-O estimate for the function f in Exercise 36 if f is an increasing function.