5

Divide-and-Conquer

Whatever man prays for, he prays for a miracle. Every prayer reduces itself to this—Great God, grant that twice two be not four.

—Ivan Turgenev (1818–1883), Russian novelist and short-story writer

Divide-and-conquer is probably the best-known general algorithm design technique. Though its fame may have something to do with its catchy name, it is well deserved: quite a few very efficient algorithms are specific implementations of this general strategy. Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

The divide-and-conquer technique is diagrammed in Figure 5.1, which depicts the case of dividing a problem into two smaller subproblems, by far the most widely occurring case (at least for divide-and-conquer algorithms designed to be executed on a single-processor computer).

As an example, let us consider the problem of computing the sum of n numbers $a_0, \ldots, a_{n-1}$. If $n > 1$, we can divide the problem into two instances of the same problem: to compute the sum of the first $\lfloor n/2 \rfloor$ numbers and to compute the sum of the remaining $\lceil n/2 \rceil$ numbers. (Of course, if $n = 1$, we simply return a_0 as the answer.) Once each of these two sums is computed by applying the same method recursively, we can add their values to get the sum in question:

$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1}).$$

Is this an efficient way to compute the sum of n numbers? A moment of reflection (why could it be more efficient than the brute-force summation?), a

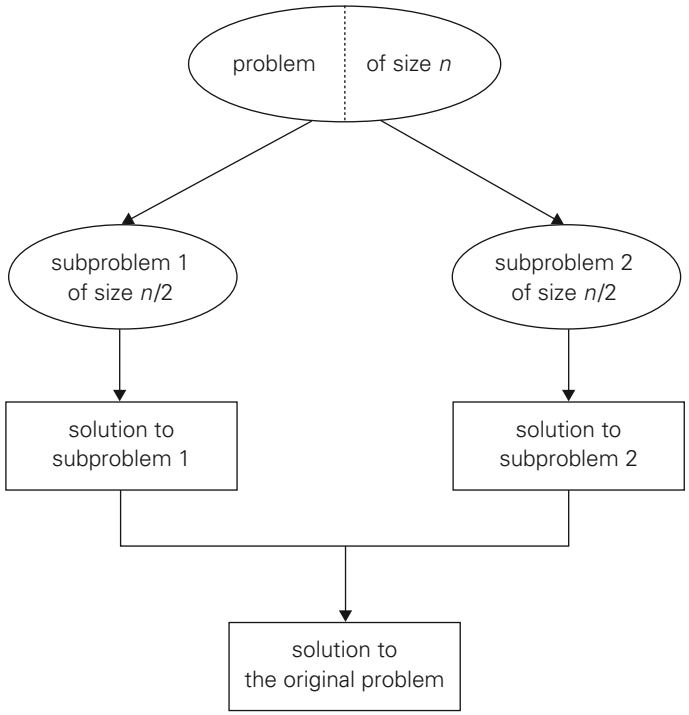


FIGURE 5.1 Divide-and-conquer technique (typical case).

small example of summing, say, four numbers by this algorithm, a formal analysis (which follows), and common sense (we do not normally compute sums this way, do we?) all lead to a negative answer to this question.[1]

Thus, not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution. But often our prayers to the Goddess of Algorithmics—see the chapter's epigraph—*are* answered, and the time spent on executing the divide-and-conquer plan turns out to be significantly smaller than solving a problem by a different method. In fact, the divide-and-conquer approach yields some of the most important and efficient algorithms in computer science. We discuss a few classic examples of such algorithms in this chapter. Though we consider only sequential algorithms here, it is worth keeping in mind that the divide-and-conquer technique is ideally suited for parallel computations, in which each subproblem can be solved simultaneously by its own processor.

1. Actually, the divide-and-conquer algorithm, called the ***pairwise summation***, may substantially reduce the accumulated round-off error of the sum of numbers that can be represented only approximately in a digital computer [Hig93].

As mentioned above, in the most typical case of divide-and-conquer a problem's instance of size n is divided into two instances of size $n/2$. More generally, an instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.) Assuming that size n is a power of b to simplify our analysis, we get the following recurrence for the running time $T(n)$:

$$T(n) = aT(n/b) + f(n), \tag{5.1}$$

where $f(n)$ is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions. (For the sum example above, $a = b = 2$ and $f(n) = 1$.) Recurrence (5.1) is called the ***general divide-and-conquer recurrence***. Obviously, the order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$. The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem (see Appendix B).

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

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

Analogous results hold for the O and Ω notations, too.

For example, the recurrence for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm (see above) on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2$, $b = 2$, and $d = 0$; hence, since $a > b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Note that we were able to find the solution's efficiency class without going through the drudgery of solving the recurrence. But, of course, this approach can only establish a solution's order of growth to within an unknown multiplicative constant, whereas solving a recurrence equation with a specific initial condition yields an exact answer (at least for n's that are powers of b).

It is also worth pointing out that if $a = 1$, recurrence (5.1) covers decrease-by-a-constant-factor algorithms discussed in the previous chapter. In fact, some people consider such algorithms as binary search degenerate cases of divide-and-conquer, where just one of two subproblems of half the size needs to be solved. It is better not to do this and consider decrease-by-a-constant-factor and divide-and-conquer as different design paradigms.

5.1 Mergesort

Mergesort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0..\lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor..n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

ALGORITHM *Mergesort*$(A[0..n-1])$

```
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌊n/2⌋ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)    //see below
```

The ***merging*** of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in Figure 5.2.

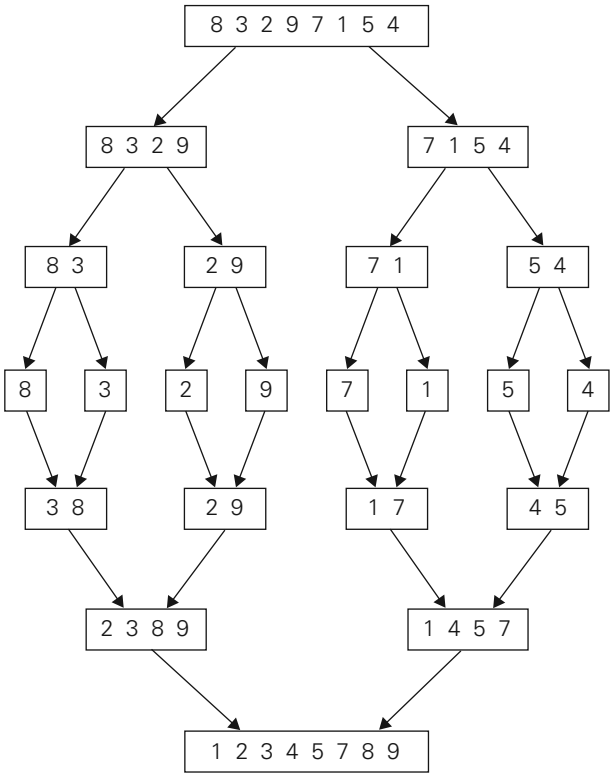


FIGURE 5.2 Example of mergesort operation.

How efficient is mergesort? Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons $C(n)$ is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

Hence, according to the Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$ (why?). In fact, it is easy to find the exact solution to the worst-case recurrence for $n = 2^k$:

$$C_{worst}(n) = n \log_2 n - n + 1.$$

The number of key comparisons made by mergesort in the worst case comes very close to the theoretical minimum[2] that any general comparison-based sorting algorithm can have. For large n, the number of comparisons made by this algorithm in the average case turns out to be about $0.25n$ less (see [Gon91, p. 173]) and hence is also in $\Theta(n \log n)$. A noteworthy advantage of mergesort over quicksort and heapsort—the two important advanced sorting algorithms to be discussed later—is its stability (see Problem 7 in this section's exercises). The principal shortcoming of mergesort is the linear amount of extra storage the algorithm requires. Though merging can be done in-place, the resulting algorithm is quite complicated and of theoretical interest only.

There are two main ideas leading to several variations of mergesort. First, the algorithm can be implemented bottom up by merging pairs of the array's elements, then merging the sorted pairs, and so on. (If n is not a power of 2, only slight bookkeeping complications arise.) This avoids the time and space overhead of using a stack to handle recursive calls. Second, we can divide a list to be sorted in more than two parts, sort each recursively, and then merge them together. This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called ***multiway mergesort***.

Exercises 5.1

1. a. Write pseudocode for a divide-and-conquer algorithm for finding the position of the largest element in an array of n numbers.

b. What will be your algorithm's output for arrays with several elements of the largest value?

c. Set up and solve a recurrence relation for the number of key comparisons made by your algorithm.

d. How does this algorithm compare with the brute-force algorithm for this problem?

2. a. Write pseudocode for a divide-and-conquer algorithm for finding values of both the largest and smallest elements in an array of n numbers.

b. Set up and solve (for $n = 2^k$) a recurrence relation for the number of key comparisons made by your algorithm.

c. How does this algorithm compare with the brute-force algorithm for this problem?

3. a. Write pseudocode for a divide-and-conquer algorithm for the exponentiation problem of computing a^n where n is a positive integer.

b. Set up and solve a recurrence relation for the number of multiplications made by this algorithm.

2. As we shall see in Section 11.2, this theoretical minimum is $\lceil \log_2 n! \rceil \approx \lceil n \log_2 n - 1.44n \rceil$.

c. How does this algorithm compare with the brute-force algorithm for this problem?

4. As mentioned in Chapter 2, logarithm bases are irrelevant in most contexts arising in analyzing an algorithm's efficiency class. Is this true for both assertions of the Master Theorem that include logarithms?

5. Find the order of growth for solutions of the following recurrences.

a. $T(n) = 4T(n/2) + n$, $T(1) = 1$

b. $T(n) = 4T(n/2) + n^2$, $T(1) = 1$

c. $T(n) = 4T(n/2) + n^3$, $T(1) = 1$

6. Apply mergesort to sort the list *E, X, A, M, P, L, E* in alphabetical order.

7. Is mergesort a stable sorting algorithm?

8. **a.** Solve the recurrence relation for the number of key comparisons made by mergesort in the worst case. You may assume that $n = 2^k$.

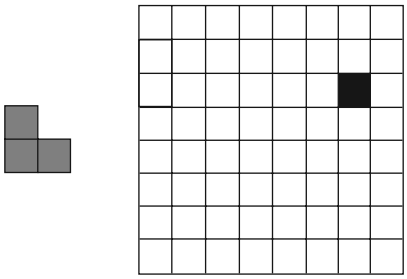
b. Set up a recurrence relation for the number of key comparisons made by mergesort on best-case inputs and solve it for $n = 2^k$.

c. Set up a recurrence relation for the number of key moves made by the version of mergesort given in Section 5.1. Does taking the number of key moves into account change the algorithm's efficiency class?

9. Let $A[0..n-1]$ be an array of n real numbers. A pair $(A[i], A[j])$ is said to be an ***inversion*** if these numbers are out of order, i.e., $i < j$ but $A[i] > A[j]$. Design an $O(n \log n)$ algorithm for counting the number of inversions.

10. Implement the bottom-up version of mergesort in the language of your choice.

11. *Tromino puzzle* A tromino (more accurately, a right tromino) is an L-shaped tile formed by three 1×1 squares. The problem is to cover any $2^n \times 2^n$ chessboard with a missing square with trominoes. Trominoes can be oriented in an arbitrary way, but they should cover all the squares of the board except the missing one exactly and with no overlaps. [Gol94]



Design a divide-and-conquer algorithm for this problem.

5.2 Quicksort

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value. We already encountered this idea of an array partition in Section 4.5, where we discussed the selection problem. A partition is an arrangement of the array's elements so that all the elements to the left of some element $A[s]$ are less than or equal to $A[s]$, and all the elements to the right of $A[s]$ are greater than or equal to it:

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are} \ge A[s]}$$

Obviously, after a partition is achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of $A[s]$ independently (e.g., by the same method). Note the difference with mergesort: there, the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions; here, the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

Here is pseudocode of quicksort: call $Quicksort(A[0..n-1])$ where

ALGORITHM $Quicksort(A[l..r])$

```
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right
//       indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l < r
    s ←Partition(A[l..r]) //s is a split position
    Quicksort(A[l..s − 1])
    Quicksort(A[s + 1..r])
```

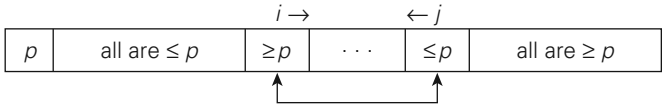
As a partition algorithm, we can certainly use the Lomuto partition discussed in Section 4.5. Alternatively, we can partition $A[0..n-1]$ and, more generally, its subarray $A[l..r]$ $(0 \le l < r \le n-1)$ by the more sophisticated method suggested by C.A.R. Hoare, the prominent British computer scientist who invented quicksort.[3]

3. C.A.R. Hoare, at age 26, invented his algorithm in 1960 while trying to sort words for a machine translation project from Russian to English. Says Hoare, "My first thought on how to do this was bubblesort and, by an amazing stroke of luck, my second thought was Quicksort." It is hard to disagree with his overall assessment: "I have been very lucky. What a wonderful way to start a career in Computing, by discovering a new sorting algorithm!" [Hoa96]. Twenty years later, he received the Turing Award for "fundamental contributions to the definition and design of programming languages"; in 1980, he was also knighted for services to education and computer science.

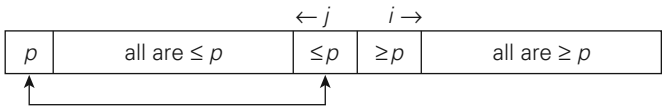
As before, we start by selecting a pivot—an element with respect to whose value we are going to divide the subarray. There are several different strategies for selecting a pivot; we will return to this issue when we analyze the algorithm's efficiency. For now, we use the simplest strategy of selecting the subarray's first element: $p = A[l]$.

Unlike the Lomuto algorithm, we will now scan the subarray from both ends, comparing the subarray's elements to the pivot. The left-to-right scan, denoted below by index pointer i, starts with the second element. Since we want elements smaller than the pivot to be in the left part of the subarray, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot. The right-to-left scan, denoted below by index pointer j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the right part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot. (Why is it worth stopping the scans after encountering an element equal to the pivot? Because doing this tends to yield more even splits for arrays with a lot of duplicates, which makes the algorithm run faster. For example, if we did otherwise for an array of n equal elements, we would have gotten a split into subarrays of sizes $n - 1$ and 0, reducing the problem size just by 1 after scanning the entire array.)

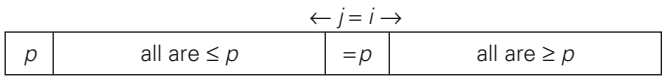
After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume the scans by incrementing i and decrementing j, respectively:



If the scanning indices have crossed over, i.e., $i > j$, we will have partitioned the subarray after exchanging the pivot with $A[j]$:



Finally, if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p (why?). Thus, we have the subarray partitioned, with the split position $s = i = j$:



We can combine the last case with the case of crossed-over indices ($i > j$) by exchanging the pivot with $A[j]$ whenever $i \geq j$.

Here is pseudocode implementing this partitioning procedure.

ALGORITHM *HoarePartition*$(A[l..r])$

```
//Partitions a subarray by Hoare's algorithm, using the first element
//        as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and right
//        indices l and r (l < r)
//Output: Partition of A[l..r], with the split position returned as
//        this function's value
p ← A[l]
i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])   //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

Note that index i can go out of the subarray's bounds in this pseudocode. Rather than checking for this possibility every time index i is incremented, we can append to array $A[0..n-1]$ a "sentinel" that would prevent index i from advancing beyond position n. Note that the more sophisticated method of pivot selection mentioned at the end of the section makes such a sentinel unnecessary.

An example of sorting an array by quicksort is given in Figure 5.3.

We start our discussion of quicksort's efficiency by noting that the number of key comparisons made before a partition is achieved is $n + 1$ if the scanning indices cross over and n if they coincide (why?). If all the splits happen in the middle of corresponding subarrays, we will have the best case. The number of key comparisons in the best case satisfies the recurrence

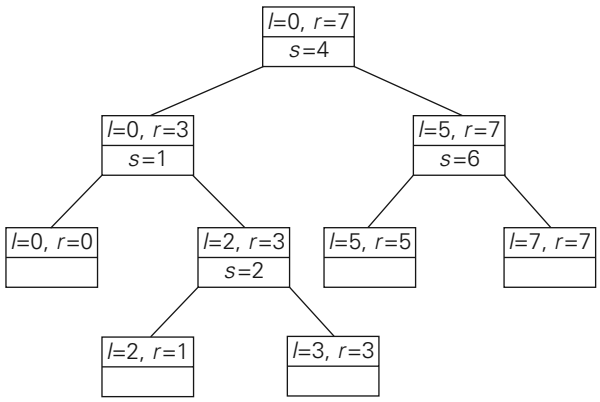
$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. This unfortunate situation will happen, in particular, for increasing arrays, i.e., for inputs for which the problem is already solved! Indeed, if $A[0..n-1]$ is a strictly increasing array and we use $A[0]$ as the pivot, the left-to-right scan will stop on $A[1]$ while the right-to-left scan will go all the way to reach $A[0]$, indicating the split at position 0:

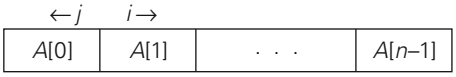
0	1	2	3	4	5	6	7
	i						*j*
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	2	8	9	7
				j	*i*		
5	3	1	4	2	8	9	7
2	3	1	4	**5**	8	9	7
	i		*j*				
2	3	1	4				
	i	*j*					
2	3	1	4				
	i	*j*					
2	1	3	4				
	j	*i*					
2	1	3	4				
1	**2**	3	4				
1							
			i j				
		3	4				
		j	*i*				
		3	4				
			4				
						i	*j*
					8	9	7
						i	*j*
					8	7	9
						j	*i*
					8	7	9
					7	**8**	9
					7		
							9

(a)



(b)

FIGURE 5.3 Example of quicksort operation. (a) Array's transformations with pivots shown in bold. (b) Tree of recursive calls to *Quicksort* with input values l and r of subarray bounds and split position s of a partition obtained.



So, after making $n + 1$ comparisons to get to this partition and exchanging the pivot $A[0]$ with itself, the algorithm will be left with the strictly increasing array $A[1..n - 1]$ to sort. This sorting of strictly increasing arrays of diminishing sizes will

continue until the last one $A[n-2..n-1]$ has been processed. The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Thus, the question about the utility of quicksort comes down to its average-case behavior. Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n. A partition can happen in any position s ($0 \le s \le n-1$) after $n+1$ comparisons are made to achieve the partition. After the partition, the left and right subarrays will have s and $n-1-s$ elements, respectively. Assuming that the partition split can happen in each position s with the same probability $1/n$, we get the following recurrence relation:

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$
$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

Its solution, which is much trickier than the worst- and best-case analyses, turns out to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$

Thus, on the average, quicksort makes only 39% more comparisons than in the best case. Moreover, its innermost loop is so efficient that it usually runs faster than mergesort (and heapsort, another $n \log n$ algorithm that we discuss in Chapter 6) on randomly ordered arrays of nontrivial sizes. This certainly justifies the name given to the algorithm by its inventor.

Because of quicksort's importance, there have been persistent efforts over the years to refine the basic algorithm. Among several improvements discovered by researchers are:

- better pivot selection methods such as ***randomized quicksort*** that uses a random element or the ***median-of-three*** method that uses the median of the leftmost, rightmost, and the middle element of the array
- switching to insertion sort on very small subarrays (between 5 and 15 elements for most computer systems) or not sorting small subarrays at all and finishing the algorithm with insertion sort applied to the entire nearly sorted array
- modifications of the partitioning algorithm such as the three-way partition into segments smaller than, equal to, and larger than the pivot (see Problem 9 in this section's exercises)

According to Robert Sedgewick [Sed11, p. 296], the world's leading expert on quicksort, such improvements in combination can cut the running time of the algorithm by 20%–30%.

Like any sorting algorithm, quicksort has weaknesses. It is not stable. It requires a stack to store parameters of subarrays that are yet to be sorted. While

the size of this stack can be made to be in $O(\log n)$ by always sorting first the smaller of two subarrays obtained by partitioning, it is worse than the $O(1)$ space efficiency of heapsort. Although more sophisticated ways of choosing a pivot make the quadratic running time of the worst case very unlikely, they do not eliminate it completely. And even the performance on randomly ordered arrays is known to be sensitive not only to implementation details of the algorithm but also to both computer architecture and data type. Still, the January/February 2000 issue of *Computing in Science & Engineering,* a joint publication of the American Institute of Physics and the IEEE Computer Society, selected quicksort as one of the 10 algorithms "with the greatest influence on the development and practice of science and engineering in the 20th century."

Exercises 5.2

1. Apply quicksort to sort the list E, X, A, M, P, L, E in alphabetical order. Draw the tree of the recursive calls made.

2. For the partitioning procedure outlined in this section:

a. Prove that if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p.

b. Prove that when the scanning indices stop, j cannot point to an element more than one position to the left of the one pointed to by i.

3. Give an example showing that quicksort is not a stable sorting algorithm.

4. Give an example of an array of n elements for which the sentinel mentioned in the text is actually needed. What should be its value? Also explain why a single sentinel suffices for any input.

5. For the version of quicksort given in this section:

a. Are arrays made up of all equal elements the worst-case input, the best-case input, or neither?

b. Are strictly decreasing arrays the worst-case input, the best-case input, or neither?

6. a. For quicksort with the median-of-three pivot selection, are strictly increasing arrays the worst-case input, the best-case input, or neither?

b. Answer the same question for strictly decreasing arrays.

7. a. Estimate how many times faster quicksort will sort an array of one million random numbers than insertion sort.

b. True or false: For every $n > 1$, there are n-element arrays that are sorted faster by insertion sort than by quicksort?

8. Design an algorithm to rearrange elements of a given array of n real numbers so that all its negative elements precede all its positive elements. Your algorithm should be both time efficient and space efficient.

9. a. The ***Dutch national flag problem*** is to rearrange an array of characters R, W, and B (red, white, and blue are the colors of the Dutch national flag) so that all the R's come first, the W's come next, and the B's come last. [Dij76] Design a linear in-place algorithm for this problem.

b. Explain how a solution to the Dutch national flag problem can be used in quicksort.

10. Implement quicksort in the language of your choice. Run your program on a sample of inputs to verify the theoretical assertions about the algorithm's efficiency.

11. *Nuts and bolts* You are given a collection of n bolts of different widths and n corresponding nuts. You are allowed to try a nut and bolt together, from which you can determine whether the nut is larger than the bolt, smaller than the bolt, or matches the bolt exactly. However, there is no way to compare two nuts together or two bolts together. The problem is to match each bolt to its nut. Design an algorithm for this problem with average-case efficiency in $\Theta(n \log n)$. [Raw91]

5.3 Binary Tree Traversals and Related Properties

In this section, we see how the divide-and-conquer technique can be applied to binary trees. A ***binary tree*** T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R called, respectively, the left and right subtree of the root. We usually think of a binary tree as a special case of an ordered tree (Figure 5.4). (This standard interpretation was an alternative definition of a binary tree in Section 1.4.)

Since the definition itself divides a binary tree into two smaller structures of the same type, the left subtree and the right subtree, many problems about binary trees can be solved by applying the divide-and-conquer technique. As an example, let us consider a recursive algorithm for computing the height of a binary tree. Recall that the height is defined as the length of the longest path from the root to a leaf. Hence, it can be computed as the maximum of the heights of the root's left

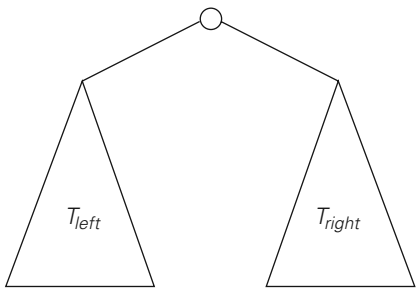


FIGURE 5.4 Standard representation of a binary tree.

and right subtrees plus 1. (We have to add 1 to account for the extra level of the root.) Also note that it is convenient to define the height of the empty tree as -1. Thus, we have the following recursive algorithm.

ALGORITHM *Height*(T)

//Computes recursively the height of a binary tree
//Input: A binary tree T
//Output: The height of T
if $T = \varnothing$ **return** -1
else return $\max\{Height(T_{left}), Height(T_{right})\} + 1$

We measure the problem's instance size by the number of nodes $n(T)$ in a given binary tree T. Obviously, the number of comparisons made to compute the maximum of two numbers and the number of additions $A(n(T))$ made by the algorithm are the same. We have the following recurrence relation for $A(n(T))$:

$$A(n(T)) = A(n(T_{left})) + A(n(T_{right})) + 1 \quad \text{for } n(T) > 0,$$
$$A(0) = 0.$$

Before we solve this recurrence (can you tell what its solution is?), let us note that addition is not the most frequently executed operation of this algorithm. What is? Checking—and this is very typical for binary tree algorithms—that the tree is not empty. For example, for the empty tree, the comparison $T = \varnothing$ is executed once but there are no additions, and for a single-node tree, the comparison and addition numbers are 3 and 1, respectively.

It helps in the analysis of tree algorithms to draw the tree's extension by replacing the empty subtrees by special nodes. The extra nodes (shown by little squares in Figure 5.5) are called ***external***; the original nodes (shown by little circles) are called ***internal***. By definition, the extension of the empty binary tree is a single external node.

It is easy to see that the *Height* algorithm makes exactly one addition for every internal node of the extended tree, and it makes one comparison to check whether

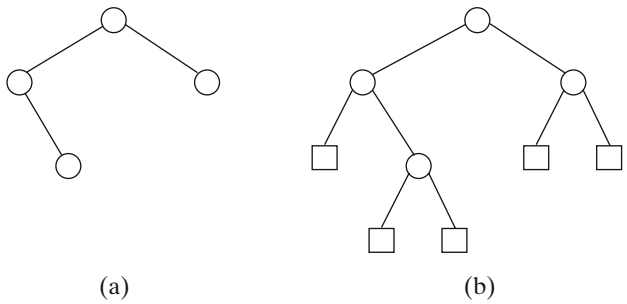


FIGURE 5.5 Binary tree (on the left) and its extension (on the right). Internal nodes are shown as circles; external nodes are shown as squares.

the tree is empty for every internal and external node. Therefore, to ascertain the algorithm's efficiency, we need to know how many external nodes an extended binary tree with n internal nodes can have. After checking Figure 5.5 and a few similar examples, it is easy to hypothesize that the number of external nodes x is always 1 more than the number of internal nodes n:

$$x = n + 1. \tag{5.2}$$

To prove this equality, consider the total number of nodes, both internal and external. Since every node, except the root, is one of the two children of an internal node, we have the equation

$$2n + 1 = x + n,$$

which immediately implies equality (5.2).

Note that equality (5.2) also applies to any nonempty ***full binary tree***, in which, by definition, every node has either zero or two children: for a full binary tree, n and x denote the numbers of parental nodes and leaves, respectively.

Returning to algorithm *Height,* the number of comparisons to check whether the tree is empty is

$$C(n) = n + x = 2n + 1,$$

and the number of additions is

$$A(n) = n.$$

The most important divide-and-conquer algorithms for binary trees are the three classic traversals: preorder, inorder, and postorder. All three traversals visit nodes of a binary tree recursively, i.e., by visiting the tree's root and its left and right subtrees. They differ only by the timing of the root's visit:

In the ***preorder traversal***, the root is visited before the left and right subtrees are visited (in that order).

In the ***inorder traversal***, the root is visited after visiting its left subtree but before visiting the right subtree.

In the ***postorder traversal***, the root is visited after visiting the left and right subtrees (in that order).

These traversals are illustrated in Figure 5.6. Their pseudocodes are quite straightforward, repeating the descriptions given above. (These traversals are also a standard feature of data structures textbooks.) As to their efficiency analysis, it is identical to the above analysis of the *Height* algorithm because a recursive call is made for each node of an extended binary tree.

Finally, we should note that, obviously, not all questions about binary trees require traversals of both left and right subtrees. For example, the search and insert operations for a binary search tree require processing only one of the two subtrees. Accordingly, we considered them in Section 4.5 not as applications of divide-and-conquer but rather as examples of the variable-size-decrease technique.

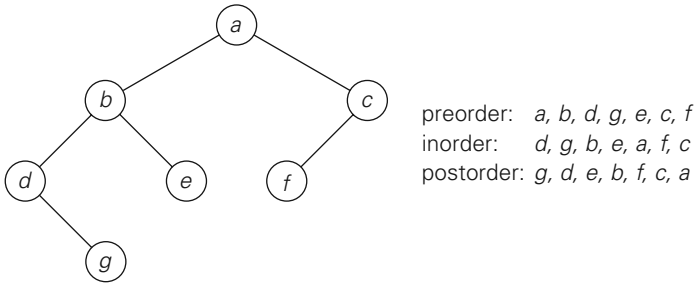


FIGURE 5.6 Binary tree and its traversals.

Exercises 5.3

1. Design a divide-and-conquer algorithm for computing the number of levels in a binary tree. (In particular, the algorithm must return 0 and 1 for the empty and single-node trees, respectively.) What is the time efficiency class of your algorithm?

2. The following algorithm seeks to compute the number of leaves in a binary tree.

ALGORITHM *LeafCounter(T)*

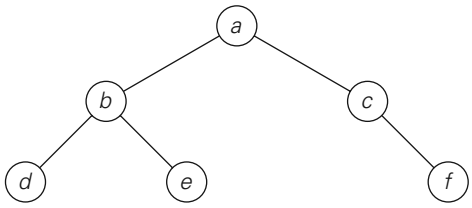
//Computes recursively the number of leaves in a binary tree
//Input: A binary tree T
//Output: The number of leaves in T
if $T = \varnothing$ **return** 0
else return $LeafCounter(T_{left}) + LeafCounter(T_{right})$

Is this algorithm correct? If it is, prove it; if it is not, make an appropriate correction.

3. Can you compute the height of a binary tree with the same asymptotic efficiency as the section's divide-and-conquer algorithm but without using a stack explicitly or implicitly? Of course, you may use a different algorithm altogether.

4. Prove equality (5.2) by mathematical induction.

5. Traverse the following binary tree
a. in preorder.
b. in inorder.
c. in postorder.



6. Write pseudocode for one of the classic traversal algorithms (preorder, inorder, and postorder) for binary trees. Assuming that your algorithm is recursive, find the number of recursive calls made.

7. Which of the three classic traversal algorithms yields a sorted list if applied to a binary search tree? Prove this property.

8. **a.** Draw a binary tree with 10 nodes labeled 0, 1, . . . , 9 in such a way that the inorder and postorder traversals of the tree yield the following lists: 9, 3, 1, 0, 4, 2, 7, 6, 8, 5 (inorder) and 9, 1, 4, 0, 3, 6, 7, 5, 8, 2 (postorder).

b. Give an example of two permutations of the same n labels 0, 1, . . . , $n - 1$ that cannot be inorder and postorder traversal lists of the same binary tree.

c. Design an algorithm that constructs a binary tree for which two given lists of n labels 0, 1, . . . , $n - 1$ are generated by the inorder and postorder traversals of the tree. Your algorithm should also identify inputs for which the problem has no solution.

9. The ***internal path length*** I of an extended binary tree is defined as the sum of the lengths of the paths—taken over all internal nodes—from the root to each internal node. Similarly, the ***external path length*** E of an extended binary tree is defined as the sum of the lengths of the paths—taken over all external nodes—from the root to each external node. Prove that $E = I + 2n$ where n is the number of internal nodes in the tree.

10. Write a program for computing the internal path length of an extended binary tree. Use it to investigate empirically the average number of key comparisons for searching in a randomly generated binary search tree.

11. *Chocolate bar puzzle* Given an $n \times m$ chocolate bar, you need to break it into nm 1×1 pieces. You can break a bar only in a straight line, and only one bar can be broken at a time. Design an algorithm that solves the problem with the minimum number of bar breaks. What is this minimum number? Justify your answer by using properties of a binary tree.

5.4 Multiplication of Large Integers and Strassen's Matrix Multiplication

In this section, we examine two surprising algorithms for seemingly straightforward tasks: multiplying two integers and multiplying two square matrices. Both

achieve a better asymptotic efficiency by ingenious application of the divide-and-conquer technique.

Multiplication of Large Integers

Some applications, notably modern cryptography, require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer, they require special treatment. This practical need supports investigations of algorithms for efficient manipulation of large integers. In this section, we outline an interesting algorithm for multiplying such numbers. Obviously, if we use the conventional pen-and-pencil algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications. (If one of the numbers has fewer digits than the other, we can pad the shorter number with leading zeros to equalize their lengths.) Though it might appear that it would be impossible to design an algorithm with fewer than n^2 digit multiplications, this turns out not to be the case. The miracle of divide-and-conquer comes to the rescue to accomplish this feat.

To demonstrate the basic idea of the algorithm, let us start with a case of two-digit integers, say, 23 and 14. These numbers can be represented as follows:

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0 \quad \text{and} \quad 14 = 1 \cdot 10^1 + 4 \cdot 10^0.$$

Now let us multiply them:

$$\begin{aligned} 23 * 14 &= (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0) \\ &= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0. \end{aligned}$$

The last formula yields the correct answer of 322, of course, but it uses the same four digit multiplications as the pen-and-pencil algorithm. Fortunately, we can compute the middle term with just one digit multiplication by taking advantage of the products $2 * 1$ and $3 * 4$ that need to be computed anyway:

$$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - 2 * 1 - 3 * 4.$$

Of course, there is nothing special about the numbers we just multiplied. For any pair of two-digit numbers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0,$$

where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0.

Now we apply this trick to multiplying two n-digit integers a and b where n is a positive even number. Let us divide both numbers in the middle—after all, we promised to take advantage of the divide-and-conquer technique. We denote the first half of the a's digits by a_1 and the second half by a_0; for b, the notations are b_1 and b_0, respectively. In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$ and $b = b_1b_0$ implies that $b = b_1 10^{n/2} + b_0$. Therefore, taking advantage of the same trick we used for two-digit numbers, we get

$$\begin{aligned} c = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\ &= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\ &= c_2 10^n + c_1 10^{n/2} + c_0, \end{aligned}$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,

$c_0 = a_0 * b_0$ is the product of their second halves,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

If $n/2$ is even, we can apply the same method for computing the products c_2, c_0, and c_1. Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. In its pure form, the recursion is stopped when n becomes 1. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

How many digit multiplications does this algorithm make? Since multiplication of n-digit numbers requires three multiplications of $n/2$-digit numbers, the recurrence for the number of multiplications $M(n)$ is

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Solving it by backward substitutions for $n = 2^k$ yields

$$\begin{aligned} M(2^k) &= 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2 M(2^{k-2}) \\ &= \cdots = 3^i M(2^{k-i}) = \cdots = 3^k M(2^{k-k}) = 3^k. \end{aligned}$$

Since $k = \log_2 n$,

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}.$$

(On the last step, we took advantage of the following property of logarithms: $a^{\log_b c} = c^{\log_b a}$.)

But what about additions and subtractions? Have we not decreased the number of multiplications by requiring more of those operations? Let $A(n)$ be the number of digit additions and subtractions executed by the above algorithm in multiplying two n-digit decimal integers. Besides $3A(n/2)$ of these operations needed to compute the three products of $n/2$-digit numbers, the above formulas

require five additions and one subtraction. Hence, we have the recurrence

$$A(n) = 3A(n/2) + cn \quad \text{for } n > 1, \quad A(1) = 1.$$

Applying the Master Theorem, which was stated in the beginning of the chapter, we obtain $A(n) \in \Theta(n^{\log_2 3})$, which means that the total number of additions and subtractions have the same asymptotic order of growth as the number of multiplications.

The asymptotic advantage of this algorithm notwithstanding, how practical is it? The answer depends, of course, on the computer system and program quality implementing the algorithm, which might explain the rather wide disparity of reported results. On some machines, the divide-and-conquer algorithm has been reported to outperform the conventional method on numbers only 8 decimal digits long and to run more than twice faster with numbers over 300 decimal digits long—the area of particular importance for modern cryptography. Whatever this outperformance "crossover point" happens to be on a particular machine, it is worth switching to the conventional algorithm after the multiplicands become smaller than the crossover point. Finally, if you program in an object-oriented language such as Java, C++, or Smalltalk, you should also be aware that these languages have special classes for dealing with large integers.

Discovered by 23-year-old Russian mathematician Anatoly Karatsuba in 1960, the divide-and-conquer algorithm proved wrong the then-prevailing opinion that the time efficiency of any integer multiplication algorithm must be in $\Omega(n^2)$. The discovery encouraged researchers to look for even (asymptotically) faster algorithms for this and other algebraic problems. We will see such an algorithm in the next section.

Strassen's Matrix Multiplication

Now that we have seen that the divide-and-conquer approach can reduce the number of one-digit multiplications in multiplying two integers, we should not be surprised that a similar feat can be accomplished for multiplying matrices. Such an algorithm was published by V. Strassen in 1969 [Str69]. The principal insight of the algorithm lies in the discovery that we can find the product C of two 2×2 matrices A and B with just seven multiplications as opposed to the eight required by the brute-force algorithm (see Example 3 in Section 2.3). This is accomplished by using the following formulas:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$
$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

where

$$
\begin{aligned}
m_1 &= (a_{00} + a_{11}) * (b_{00} + b_{11}),\\
m_2 &= (a_{10} + a_{11}) * b_{00},\\
m_3 &= a_{00} * (b_{01} - b_{11}),\\
m_4 &= a_{11} * (b_{10} - b_{00}),\\
m_5 &= (a_{00} + a_{01}) * b_{11},\\
m_6 &= (a_{10} - a_{00}) * (b_{00} + b_{01}),\\
m_7 &= (a_{01} - a_{11}) * (b_{10} + b_{11}).
\end{aligned}
$$

Thus, to multiply two 2×2 matrices, Strassen's algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions. These numbers should not lead us to multiplying 2×2 matrices by Strassen's algorithm. Its importance stems from its *asymptotic* superiority as matrix order n goes to infinity.

Let A and B be two $n \times n$ matrices where n is a power of 2. (If n is not a power of 2, matrices can be padded with rows and columns of zeros.) We can divide A, B, and their product C into four $n/2 \times n/2$ submatrices each as follows:

$$
\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right].
$$

It is not difficult to verify that one can treat these submatrices as numbers to get the correct product. For example, C_{00} can be computed either as $A_{00} * B_{00} + A_{01} * B_{10}$ or as $M_1 + M_4 - M_5 + M_7$ where M_1, M_4, M_5, and M_7 are found by Strassen's formulas, with the numbers replaced by the corresponding submatrices. If the seven products of $n/2 \times n/2$ matrices are computed recursively by the same method, we have Strassen's algorithm for matrix multiplication.

Let us evaluate the asymptotic efficiency of this algorithm. If $M(n)$ is the number of multiplications made by Strassen's algorithm in multiplying two $n \times n$ matrices (where n is a power of 2), we get the following recurrence relation for it:

$$
M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.
$$

Since $n = 2^k$,

$$
\begin{aligned}
M(2^k) &= 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2 M(2^{k-2}) = \cdots\\
&= 7^i M(2^{k-i}) \cdots = 7^k M(2^{k-k}) = 7^k.
\end{aligned}
$$

Since $k = \log_2 n$,

$$
M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},
$$

which is smaller than n^3 required by the brute-force algorithm.

Since this savings in the number of multiplications was achieved at the expense of making extra additions, we must check the number of additions $A(n)$ made by Strassen's algorithm. To multiply two matrices of order $n > 1$, the algorithm needs to multiply seven matrices of order $n/2$ and make 18 additions/subtractions of matrices of size $n/2$; when $n = 1$, no additions are made since two numbers are

simply multiplied. These observations yield the following recurrence relation:

$$A(n) = 7A(n/2) + 18(n/2)^2 \quad \text{for } n > 1, \quad A(1) = 0.$$

Though one can obtain a closed-form solution to this recurrence (see Problem 8 in this section's exercises), here we simply establish the solution's order of growth. According to the Master Theorem, $A(n) \in \Theta(n^{\log_2 7})$. In other words, the number of additions has the same order of growth as the number of multiplications. This puts Strassen's algorithm in $\Theta(n^{\log_2 7})$, which is a better efficiency class than $\Theta(n^3)$ of the brute-force method.

Since the time of Strassen's discovery, several other algorithms for multiplying two $n \times n$ matrices of real numbers in $O(n^\alpha)$ time with progressively smaller constants α have been invented. The fastest algorithm so far is that of Coopersmith and Winograd [Coo87] with its efficiency in $O(n^{2.376})$. The decreasing values of the exponents have been obtained at the expense of the increasing complexity of these algorithms. Because of large multiplicative constants, none of them is of practical value. However, they are interesting from a theoretical point of view. On one hand, they get closer and closer to the best theoretical lower bound known for matrix multiplication, which is n^2 multiplications, though the gap between this bound and the best available algorithm remains unresolved. On the other hand, matrix multiplication is known to be computationally equivalent to some other important problems, such as solving systems of linear equations (discussed in the next chapter).

Exercises 5.4

1. What are the smallest and largest numbers of digits the product of two decimal n-digit integers can have?

2. Compute $2101 * 1130$ by applying the divide-and-conquer algorithm outlined in the text.

3. a. Prove the equality $a^{\log_b c} = c^{\log_b a}$, which was used in Section 5.4.

b. Why is $n^{\log_2 3}$ better than $3^{\log_2 n}$ as a closed-form formula for $M(n)$?

4. a. Why did we not include multiplications by 10^n in the multiplication count $M(n)$ of the large-integer multiplication algorithm?

b. In addition to assuming that n is a power of 2, we made, for the sake of simplicity, another, more subtle, assumption in setting up the recurrences for $M(n)$ and $A(n)$, which is not always true (it does not change the final answers, however). What is this assumption?

5. How many one-digit additions are made by the pen-and-pencil algorithm in multiplying two n-digit integers? You may disregard potential carries.

6. Verify the formulas underlying Strassen's algorithm for multiplying 2×2 matrices.

7. Apply Strassen's algorithm to compute

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

exiting the recursion when $n = 2$, i.e., computing the products of 2×2 matrices by the brute-force algorithm.

8. Solve the recurrence for the number of additions required by Strassen's algorithm. Assume that n is a power of 2.

9. V. Pan [Pan78] has discovered a divide-and-conquer matrix multiplication algorithm that is based on multiplying two 70×70 matrices using 143,640 multiplications. Find the asymptotic efficiency of Pan's algorithm (you may ignore additions) and compare it with that of Strassen's algorithm.

10. Practical implementations of Strassen's algorithm usually switch to the brute-force method after matrix sizes become smaller than some crossover point. Run an experiment to determine such a crossover point on your computer system.

5.5 The Closest-Pair and Convex-Hull Problems by Divide-and-Conquer

In Section 3.3, we discussed the brute-force approach to solving two classic problems of computational geometry: the closest-pair problem and the convex-hull problem. We saw that the two-dimensional versions of these problems can be solved by brute-force algorithms in $\Theta(n^2)$ and $O(n^3)$ time, respectively. In this section, we discuss more sophisticated and asymptotically more efficient algorithms for these problems, which are based on the divide-and-conquer technique.

The Closest-Pair Problem

Let P be a set of $n > 1$ points in the Cartesian plane. For the sake of simplicity, we assume that the points are distinct. We can also assume that the points are ordered in nondecreasing order of their x coordinate. (If they were not, we could sort them first by an efficeint sorting algorithm such as mergesort.) It will also be convenient to have the points sorted in a separate list in nondecreasing order of the y coordinate; we will denote such a list Q.

If $2 \le n \le 3$, the problem can be solved by the obvious brute-force algorithm. If $n > 3$, we can divide the points into two subsets P_l and P_r of $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ points, respectively, by drawing a vertical line through the median m of their x coordinates so that $\lceil n/2 \rceil$ points lie to the left of or on the line itself, and $\lfloor n/2 \rfloor$ points lie to the right of or on the line. Then we can solve the closest-pair problem

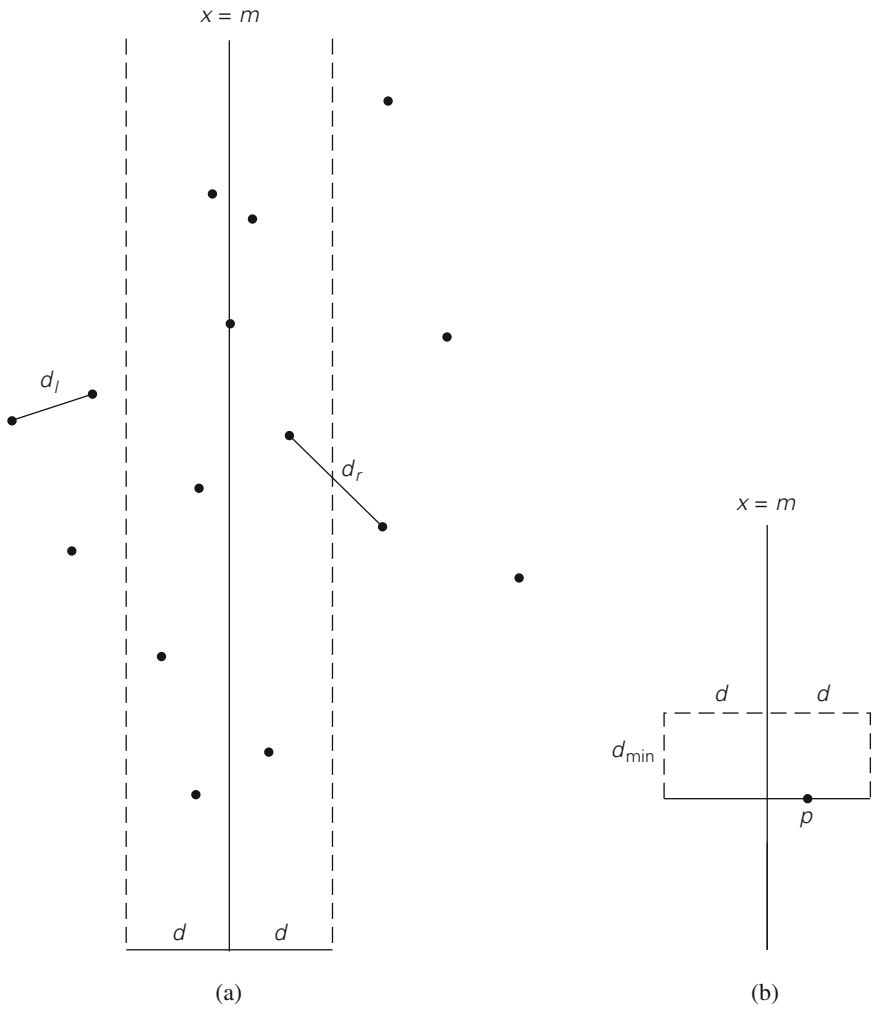


FIGURE 5.7 (a) Idea of the divide-and-conquer algorithm for the closest-pair problem. (b) Rectangle that may contain points closer than $d_{\min}$ to point p.

recursively for subsets P_l and P_r. Let d_l and d_r be the smallest distances between pairs of points in P_l and P_r, respectively, and let $d = \min\{d_l, d_r\}$.

Note that d is not necessarily the smallest distance between all the point pairs because points of a closer pair can lie on the opposite sides of the separating line. Therefore, as a step combining the solutions to the smaller subproblems, we need to examine such points. Obviously, we can limit our attention to the points inside the symmetric vertical strip of width $2d$ around the separating line, since the distance between any other pair of points is at least d (Figure 5.7a).

Let S be the list of points inside the strip of width $2d$ around the separating line, obtained from Q and hence ordered in nondecreasing order of their y coordinate. We will scan this list, updating the information about $d_{\min}$, the minimum distance seen so far, if we encounter a closer pair of points. Initially, $d_{\min} = d$, and subsequently $d_{\min} \leq d$. Let $p(x, y)$ be a point on this list. For a point $p'(x', y')$ to have a chance to be closer to p than $d_{\min}$, the point must follow p on list S and the difference between their y coordinates must be less than $d_{\min}$ (why?). Geometrically, this means that p' must belong to the rectangle shown in Figure 5.7b. The principal insight exploited by the algorithm is the observation that the rectangle can contain just a few such points, because the points in each half (left and right) of the rectangle must be at least distance d apart. It is easy to prove that the total number of such points in the rectangle, including p, does not exceed eight (Problem 2 in this section's exercises); a more careful analysis reduces this number to six (see [Joh04, p. 695]). Thus, the algorithm can consider no more than five next points following p on the list S, before moving up to the next point.

Here is pseudocode of the algorithm. We follow the advice given in Section 3.3 to avoid computing square roots inside the innermost loop of the algorithm.

ALGORITHM *EfficientClosestPair*(P, Q)

```
//Solves the closest-pair problem by divide-and-conquer
//Input: An array P of n ≥ 2 points in the Cartesian plane sorted in
//       nondecreasing order of their x coordinates and an array Q of the
//       same points sorted in nondecreasing order of the y coordinates
//Output: Euclidean distance between the closest pair of points
if n ≤ 3
    return the minimal distance found by the brute-force algorithm
else
    copy the first ⌈n/2⌉ points of P to array P_l
    copy the same ⌈n/2⌉ points from Q to array Q_l
    copy the remaining ⌊n/2⌋ points of P to array P_r
    copy the same ⌊n/2⌋ points from Q to array Q_r
    d_l ← EfficientClosestPair(P_l, Q_l)
    d_r ← EfficientClosestPair(P_r, Q_r)
    d ← min{d_l, d_r}
    m ← P[⌈n/2⌉ − 1].x
    copy all the points of Q for which |x − m| < d into array S[0..num − 1]
    dminsq ← d^2
    for i ← 0 to num − 2 do
        k ← i + 1
        while k ≤ num − 1 and (S[k].y − S[i].y)^2 < dminsq
            dminsq ← min((S[k].x − S[i].x)^2 + (S[k].y − S[i].y)^2, dminsq)
            k ← k + 1
return sqrt(dminsq)
```

The algorithm spends linear time both for dividing the problem into two problems half the size and combining the obtained solutions. Therefore, assuming as usual that n is a power of 2, we have the following recurrence for the running time of the algorithm:

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

where $f(n) \in \Theta(n)$. Applying the Master Theorem (with $a = 2$, $b = 2$, and $d = 1$), we get $T(n) \in \Theta(n \log n)$. The necessity to presort input points does not change the overall efficiency class if sorting is done by a $O(n \log n)$ algorithm such as mergesort. In fact, this is the best efficiency class one can achieve, because it has been proved that any algorithm for this problem must be in $\Omega(n \log n)$ under some natural assumptions about operations an algorithm can perform (see [Pre85, p. 188]).

Convex-Hull Problem

Let us revisit the convex-hull problem, introduced in Section 3.3: find the smallest convex polygon that contains n given points in the plane. We consider here a divide-and-conquer algorithm called ***quickhull*** because of its resemblance to quicksort.

Let S be a set of $n > 1$ points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$ in the Cartesian plane. We assume that the points are sorted in nondecreasing order of their x coordinates, with ties resolved by increasing order of the y coordinates of the points involved. It is not difficult to prove the geometrically obvious fact that the leftmost point p_1 and the rightmost point p_n are two distinct extreme points of the set's convex hull (Figure 5.8). Let $\overrightarrow{p_1 p_n}$ be the straight line through points p_1 and p_n directed from p_1 to p_n. This line separates the points of S into two sets: S_1 is the set of points to the left of this line, and S_2 is the set of points to the right of this line. (We say that point q_3 is to the left of the line $\overrightarrow{q_1 q_2}$ directed from point q_1 to point q_2 if $q_1 q_2 q_3$ forms a counterclockwise cycle. Later, we cite an analytical way to check this condition, based on checking the sign of a determinant formed by the coordinates of the three points.) The points of S on the line $\overrightarrow{p_1 p_n}$, other than p_1 and p_n, cannot be extreme points of the convex hull and hence are excluded from further consideration.

The boundary of the convex hull of S is made up of two polygonal chains: an "upper" boundary and a "lower" boundary. The "upper" boundary, called the ***upper hull***, is a sequence of line segments with vertices at p_1, some of the points in S_1 (if S_1 is not empty) and p_n. The "lower" boundary, called the ***lower hull***, is a sequence of line segments with vertices at p_1, some of the points in S_2 (if S_2 is not empty) and p_n. The fact that the convex hull of the entire set S is composed of the upper and lower hulls, which can be constructed independently and in a similar fashion, is a very useful observation exploited by several algorithms for this problem.

For concreteness, let us discuss how quickhull proceeds to construct the upper hull; the lower hull can be constructed in the same manner. If S_1 is empty, the

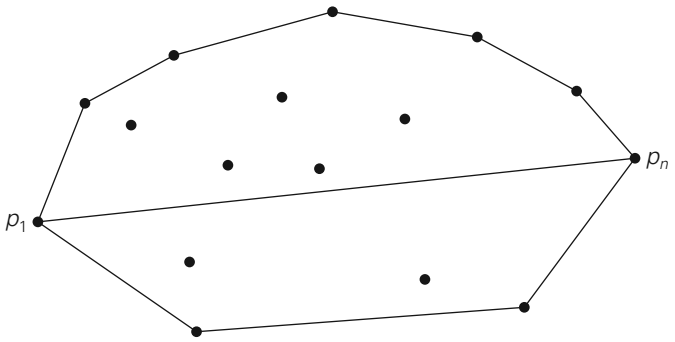


FIGURE 5.8 Upper and lower hulls of a set of points.

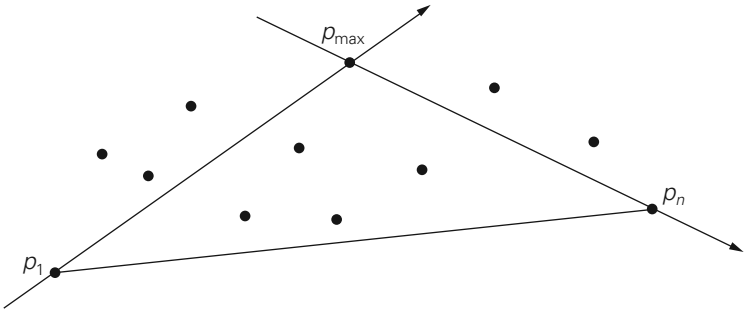


FIGURE 5.9 The idea of quickhull.

upper hull is simply the line segment with the endpoints at p_1 and p_n. If S_1 is not empty, the algorithm identifies point $p_{\max}$ in S_1, which is the farthest from the line $\overrightarrow{p_1 p_n}$ (Figure 5.9). If there is a tie, the point that maximizes the angle $\angle p_{\max} p p_n$ can be selected. (Note that point $p_{\max}$ maximizes the area of the triangle with two vertices at p_1 and p_n and the third one at some other point of S_1.) Then the algorithm identifies all the points of set S_1 that are to the left of the line $\overrightarrow{p_1 p_{\max}}$; these are the points that will make up the set $S_{1,1}$. The points of S_1 to the left of the line $\overrightarrow{p_{\max} p_n}$ will make up the set $S_{1,2}$. It is not difficult to prove the following:

- $p_{\max}$ is a vertex of the upper hull.
- The points inside $\triangle p_1 p_{\max} p_n$ cannot be vertices of the upper hull (and hence can be eliminated from further consideration).
- There are no points to the left of both lines $\overrightarrow{p_1 p_{\max}}$ and $\overrightarrow{p_{\max} p_n}$.

Therefore, the algorithm can continue constructing the upper hulls of $p_1 \cup S_{1,1} \cup p_{\max}$ and $p_{\max} \cup S_{1,2} \cup p_n$ recursively and then simply concatenate them to get the upper hull of the entire set $p_1 \cup S_1 \cup p_n$.

Now we have to figure out how the algorithm's geometric operations can be actually implemented. Fortunately, we can take advantage of the following very useful fact from analytical geometry: if $q_1(x_1, y_1)$, $q_2(x_2, y_2)$, and $q_3(x_3, y_3)$ are three arbitrary points in the Cartesian plane, then the area of the triangle $\triangle q_1q_2q_3$ is equal to one-half of the magnitude of the determinant

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3,$$

while the sign of this expression is positive if and only if the point $q_3 = (x_3, y_3)$ is to the left of the line $\overrightarrow{q_1q_2}$. Using this formula, we can check in constant time whether a point lies to the left of the line determined by two other points as well as find the distance from the point to the line.

Quickhull has the same $\Theta(n^2)$ worst-case efficiency as quicksort (Problem 9 in this section's exercises). In the average case, however, we should expect a much better performance. First, the algorithm should benefit from the quicksort-like savings from the on-average balanced split of the problem into two smaller subproblems. Second, a significant fraction of the points—namely, those inside $\triangle p_1p_{\max}p_n$ (see Figure 5.9)—are eliminated from further processing. Under a natural assumption that points given are chosen randomly from a uniform distribution over some convex region (e.g., a circle or a rectangle), the average-case efficiency of quickhull turns out to be linear [Ove80].

Exercises 5.5

1. **a.** For the one-dimensional version of the closest-pair problem, i.e., for the problem of finding two closest numbers among a given set of n real numbers, design an algorithm that is directly based on the divide-and-conquer technique and determine its efficiency class.
   b. Is it a good algorithm for this problem?
2. Prove that the divide-and-conquer algorithm for the closest-pair problem examines, for every point p in the vertical strip (see Figures 5.7a and 5.7b), no more than seven other points that can be closer to p than $d_{\min}$, the minimum distance between two points encountered by the algorithm up to that point.
3. Consider the version of the divide-and-conquer two-dimensional closest-pair algorithm in which, instead of presorting input set P, we simply sort each of the two sets P_l and P_r in nondecreasing order of their y coordinates on each recursive call. Assuming that sorting is done by mergesort, set up a recurrence relation for the running time in the worst case and solve it for $n = 2^k$.
4. Implement the divide-and-conquer closest-pair algorithm, outlined in this section, in the language of your choice.

5. Find on the Web a visualization of an algorithm for the closest-pair problem. What algorithm does this visualization represent?
6. The **Voronoi polygon** for a point p of a set S of points in the plane is defined to be the perimeter of the set of all points in the plane closer to p than to any other point in S. The union of all the Voronoi polygons of the points in S is called the ***Voronoi diagram*** of S.
   a. What is the Voronoi diagram for a set of three points?
   b. Find a visualization of an algorithm for generating the Voronoi diagram on the Web and study a few examples of such diagrams. Based on your observations, can you tell how the solution to the previous question is generalized to the general case?
7. Explain how one can find point $p_{\max}$ in the quickhull algorithm analytically.
8. What is the best-case efficiency of quickhull?
9. Give a specific example of inputs that make quickhull run in quadratic time.
10. Implement quickhull in the language of your choice.
11. *Creating decagons* There are 1000 points in the plane, no three of them on the same line. Devise an algorithm to construct 100 decagons with their vertices at these points. The decagons need not be convex, but each of them has to be simple, i.e., its boundary should not cross itself, and no two decagons may have a common point.
12. *Shortest path around* There is a fenced area in the two-dimensional Euclidean plane in the shape of a convex polygon with vertices at points $p_1(x_1, y_1)$, $p_2(x_2, y_2)$, $\ldots$, $p_n(x_n, y_n)$ (not necessarily in this order). There are two more points, $a(x_a, y_a)$ and $b(x_b, y_b)$ such that $x_a < \min\{x_1, x_2, \ldots, x_n\}$ and $x_b > \max\{x_1, x_2, \ldots, x_n\}$. Design a reasonably efficient algorithm for computing the length of the shortest path between a and b. [ORo98]

SUMMARY

- *Divide-and-conquer* is a general algorithm design technique that solves a problem by dividing it into several smaller subproblems of the same type (ideally, of about equal size), solving each of them recursively, and then combining their solutions to get a solution to the original problem. Many efficient algorithms are based on this technique, although it can be both inapplicable and inferior to simpler algorithmic solutions.
- Running time $T(n)$ of many divide-and-conquer algorithms satisfies the recurrence $T(n) = aT(n/b) + f(n)$. The *Master Theorem* establishes the order of growth of its solutions.
- *Mergesort* is a divide-and-conquer sorting algorithm. It works by dividing an input array into two halves, sorting them recursively, and then *merging* the two

sorted halves to get the original array sorted. The algorithm's time efficiency is in $\Theta(n \log n)$ in all cases, with the number of key comparisons being very close to the theoretical minimum. Its principal drawback is a significant extra storage requirement.

- *Quicksort* is a divide-and-conquer sorting algorithm that works by partitioning its input elements according to their value relative to some preselected element. Quicksort is noted for its superior efficiency among $n \log n$ algorithms for sorting randomly ordered arrays but also for the quadratic worst-case efficiency.
- The classic traversals of a binary tree—*preorder*, *inorder*, and *postorder*—and similar algorithms that require recursive processing of both left and right subtrees can be considered examples of the divide-and-conquer technique. Their analysis is helped by replacing all the empty subtrees of a given tree by special *external nodes*.
- There is a divide-and-conquer algorithm for multiplying two n-digit integers that requires about $n^{1.585}$ one-digit multiplications.
- *Strassen's algorithm* needs only seven multiplications to multiply two 2×2 matrices. By exploiting the divide-and-conquer technique, this algorithm can multiply two $n \times n$ matrices with about $n^{2.807}$ multiplications.
- The divide-and-conquer technique can be successfully applied to two important problems of computational geometry: the closest-pair problem and the convex-hull problem.