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

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