Chapter 3

DIVIDE-AND-CONQUER

3.1 GENERAL METHOD

Given a function to compute on n inputs the *divide-and-conquer* strategy suggests splitting the inputs into k distinct subsets, $1 < k \leq n$, yielding k subproblems. These subproblems must be solved, and then a method must be found to combine subsolutions into a solution of the whole. If the subproblems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied. Often the subproblems resulting from a divide-and-conquer design are of the *same* type as the original problem. For those cases the reapplication of the divide-and-conquer principle is naturally expressed by a recursive algorithm. Now smaller and smaller subproblems of the same kind are generated until eventually subproblems that are small enough to be solved without splitting are produced.

To be more precise, suppose we consider the divide-and-conquer strategy when it splits the input into two subproblems of the same kind as the original problem. This splitting is typical of many of the problems we examine here. We can write a control abstraction that mirrors the way an algorithm based on divide-and-conquer will look. By a *control abstraction* we mean a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. DAndC (Algorithm 3.1) is initially invoked as DAndC(P), where P is the problem to be solved.

Small(P) is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this is so, the function S is invoked. Otherwise the problem P is divided into smaller subproblems. These subproblems $P_1, P_2, \ldots, P_k$ are solved by recursive applications of DAndC. Combine is a function that determines the solution to P using the solutions to the k subproblems. If the size of P is n and the sizes of the k subproblems are $n_1, n_2, \ldots, n_k$, respectively, then the

```
1   Algorithm DAndC(P)
2   {
3       if Small(P) then return S(P);
4       else
5       {
6           divide P into smaller instances P_1, P_2, ..., P_k, k ≥ 1;
7           Apply DAndC to each of these subproblems;
8           return Combine(DAndC(P_1),DAndC(P_2),...,DAndC(P_k));
9       }
10  }
```

Algorithm 3.1 Control abstraction for divide-and-conquer

computing time of DAndC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases} \tag{3.1}$$

where $T(n)$ is the time for DAndC on any input of size n and $g(n)$ is the time to compute the answer directly for small inputs. The function $f(n)$ is the time for dividing P and combining the solutions to subproblems. For divide-and-conquer-based algorithms that produce subproblems of the same type as the original problem, it is very natural to first describe such algorithms using recursion.

The complexity of many divide-and-conquer algorithms is given by recurrences of the form

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases} \tag{3.2}$$

where a and b are known constants. We assume that $T(1)$ is known and n is a power of b (i.e., $n = b^k$).

One of the methods for solving any such recurrence relation is called the *substitution method.* This method repeatedly makes substitution for each occurrence of the function T in the right-hand side until all such occurrences disappear.

Example 3.1 Consider the case in which $a = 2$ and $b = 2$. Let $T(1) = 2$ and $f(n) = n$. We have

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \end{aligned}$$

In general, we see that $T(n) = 2^i T(n/2^i) + in$, for any $\log_2 n \geq i \geq 1$. In particular, then, $T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + n \log_2 n$, corresponding to the choice of $i = \log_2 n$. Thus, $T(n) = nT(1) + n \log_2 n = n \log_2 n + 2n$. □

Beginning with the recurrence (3.2) and using the substitution method, it can be shown that

$$T(n) = n^{\log_b a}[T(1) + u(n)]$$

where $u(n) = \sum_{j=1}^{k} h(b^j)$ and $h(n) = f(n)/n^{\log_b a}$. Table 3.1 tabulates the asymptotic value of $u(n)$ for various values of $h(n)$. This table allows one to easily obtain the asymptotic value of $T(n)$ for many of the recurrences one encounters when analyzing divide-and-conquer algorithms. Let us consider some examples using this table.

$h(n)$	$u(n)$
$O(n^r),\ r < 0$	$O(1)$
$\Theta((\log n)^i),\ i \geq 0$	$\Theta((\log n)^{i+1}/(i+1))$
$\Omega(n^r),\ r > 0$	$\Theta(h(n))$

Table 3.1 $u(n)$ values for various $h(n)$ values

Example 3.2 Look at the following recurrence when n is a power of 2:

$$T(n) = \begin{cases} T(1) & n = 1 \\ T(n/2) + c & n > 1 \end{cases}$$