2.3 The Divide-and-Conquer Approach

Having studied two divide-and-conquer algorithms in detail, you should now better understand the following general description of this approach.

The ***divide-and-conquer*** design strategy involves the following steps:

1. *Divide* an instance of a problem into one or more smaller instances.
2. *Conquer* (solve) each of the smaller instances. Unless a smaller instance is sufficiently small, use recursion to do this.
3. If necessary, *combine* the solutions to the smaller instances to obtain the solution to the original instance.

The reason we say "if necessary" in Step 3 is that in algorithms such as Binary Search Recursive (Algorithm 2.1) the instance is reduced to just one smaller instance, so there is no need to combine solutions.

More examples of the divide-and-conquer approach follow. In these examples we will not explicitly mention the steps previously outlined. It should be clear that we are following them.

2.4 Quicksort (Partition Exchange Sort)

Next we look at a sorting algorithm, called "Quicksort," that was developed by Hoare (1962). Quicksort is similar to Mergesort in that the sort is accomplished by dividing the array into two partitions and then sorting each partition recursively. In Quicksort, however, the array is partitioned by placing all items smaller than some pivot item before that item and all items larger than the pivot item after it. The pivot item can be any item, and for convenience we will simply make it the first one. The following example illustrates how Quicksort works.

Example 2.3 Suppose the array contains these numbers in sequence:

$$\overset{\overset{\text{Pivot item}}{\downarrow}}{15}\ \ 22\ \ 13\ \ 27\ \ 12\ \ 10\ \ 20\ \ 25$$

1. Partition the array so that all items smaller than the pivot item are to the left of it and all items larger are to the right:

$$\underbrace{\underline{10\ \ 13\ \ 12}}_{\text{All smaller}}\ \ \overset{\overset{\text{Pivot item}}{\downarrow}}{15}\ \ \underbrace{\underline{22\ \ 27\ \ 20\ \ 25}}_{\text{All larger}}$$

2. Sort the subarrays:

$$\begin{array}{c} \text{Pivot item} \\ \downarrow \\ \underbrace{10\ \ 12\ \ 13}_{\text{Sorted}}\ \ 15\ \ \underbrace{20\ \ 22\ \ 25\ \ 27}_{\text{Sorted}} \end{array}$$

After the partitioning, the order of the items in the subarrays is unspecified and is a result of how the partitioning is implemented. We have ordered them according to how the partitioning routine, which will be presented shortly, would place them. The important thing is that all items smaller than the pivot item are to the left of it, and all items larger are to the right of it. Quicksort is then called recursively to sort each of the two subarrays. They are partitioned, and this procedure is continued until an array with one item is reached. Such an array is trivially sorted. Example 2.3 shows the solution at the problem-solving level. Figure 2.3 illustrates the steps done by a human when sorting with Quicksort. The algorithm follows.

▶ Algorithm 2.6

Quicksort

Problem: Sort n keys in nondecreasing order.

Inputs: positive integer n, array of keys S indexed from 1 to n.

Outputs: the array S containing the keys in nondecreasing order.

```
void quicksort (index low, index high)
{
  index pivotpoint;

  if (high > low){
     partition(low, high, pivotpoint);
     quicksort(low, pivotpoint − 1);
     quicksort(pivotpoint + 1, high);
  }
}
```

Following our usual convention, n and S are not parameters to procedure *quicksort*. If the algorithm were implemented by defining S globally and n was the number of items in S, the top-level call to *quicksort* would be as follows:

```
quicksort(1, n);
```

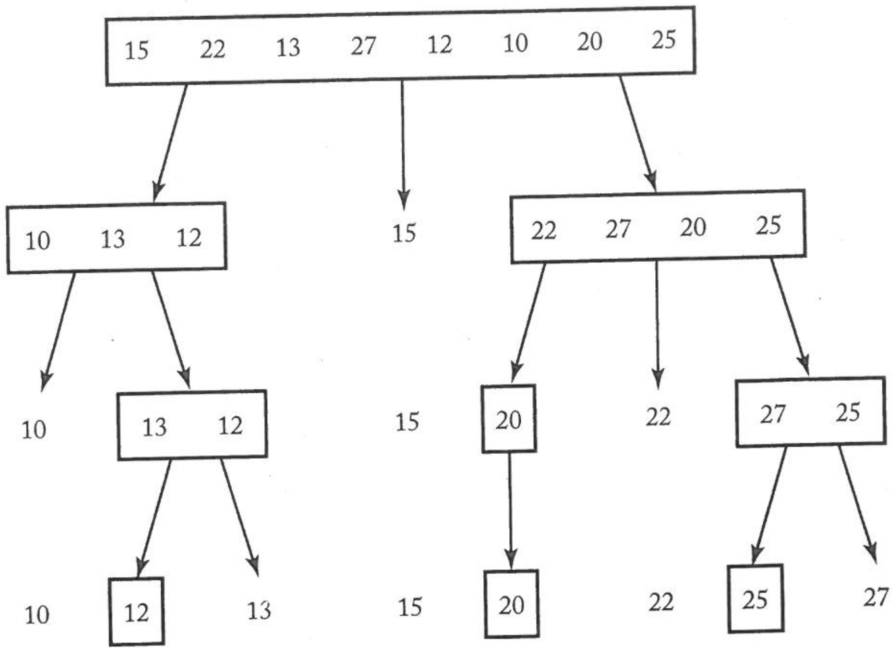

Figure 2.3
The steps done by a human when sorting with Quicksort. The subarrays are enclosed in rectangles whereas the pivot points are free.

The partitioning of the array is done by procedure *partition*. Next we show an algorithm for this procedure.

▶ Algorithm 2.7

Partition

Problem: Partition the array S for Quicksort.

Inputs: two indices, *low* and *high*, and the subarray of S indexed from *low* to *high*.

Outputs: *pivotpoint*, the pivot point for the subarray indexed from *low* to *high*.

```
void partition (index low, index high,
                index& pivotpoint)
{
   index i, j;
   keytype pivotitem;

   pivotitem = S[low];                          // Choose first item for
   j = low;                                     // pivotitem.
   for (i = low + 1; i <= high; i++)
      if (S[i] < pivotitem){
         j++;
         exchange S[i] and S[j];
      }
   pivotpoint = j;
   exchange S[low] and S[pivotpoint];  // Put pivotitem at pivotpoint.
}
```

● **Table 2.2** An example of procedure *partition**

i	j	$S[1]$	$S[2]$	$S[3]$	$S[4]$	$S[5]$	$S[6]$	$S[7]$	$S[8]$	
—	—	15	22	13	27	12	10	20	25	← Initial values
2	1	**15**	**22**	13	27	12	10	20	25	
3	2	**15**	22	**13**	27	12	10	20	25	
4	2	**15**	$\boxed{13}$	$\boxed{22}$	**27**	12	10	20	25	
5	3	**15**	13	22	27	**12**	10	20	25	
6	4	**15**	13	$\boxed{12}$	27	$\boxed{22}$	**10**	20	25	
7	4	**15**	13	12	$\boxed{10}$	22	$\boxed{27}$	**20**	25	
8	4	**15**	13	12	10	22	27	20	**25**	
—	4	$\boxed{10}$	13	12	$\boxed{15}$	22	27	20	25	← Final values

*Items compared are in boldface. Items just exchanged appear in squares.

Procedure *partition* works by checking each item in the array in sequence. Whenever an item is found to be less than the pivot item, it is moved to the left side of the array. Table 2.2 shows how *partition* would proceed on the array in Example 2.3.

Next we analyze Partition and Quicksort.

Analysis of Algorithm 2.7

▶ **Every-Case Time Complexity (Partition)**

Basic operation: the comparison of $S[i]$ with *pivotitem*.

Input size: $n = high - low + 1$, the number of items in the subarray.

Because every item except the first is compared,

$$T(n) = n - 1.$$

We are using n here to represent the size of the subarray, not the size of the array S. It represents the size of S only when *partition* is called at the top level.

Quicksort does not have an every-case complexity. We will do worst-case and average-case analyses.

Analysis of Algorithm 2.6

▶ **Worst-Case Time Complexity (Quicksort)**

Basic operation: the comparison of $S[i]$ with *pivotitem* in *partition*.

Input size: n, the number of items in the array S.

Oddly enough, it turns out that the worst case occurs if the array is already sorted in nondecreasing order. The reason for this should become