

order of growth. But for large enough inputs, a  $\Theta(n^2)$  algorithm, for example, will run more quickly in the worst case than a  $\Theta(n^3)$  algorithm.

### Exercises

#### 2.2-1

Express the function  $n^3/1000 - 100n^2 - 100n + 3$  in terms of  $\Theta$ -notation.

#### 2.2-2

Consider sorting  $n$  numbers stored in array  $A$  by first finding the smallest element of  $A$  and exchanging it with the element in  $A[1]$ . Then find the second smallest element of  $A$ , and exchange it with  $A[2]$ . Continue in this manner for the first  $n - 1$  elements of  $A$ . Write pseudocode for this algorithm, which is known as **selection sort**. What loop invariant does this algorithm maintain? Why does it need to run for only the first  $n - 1$  elements, rather than for all  $n$  elements? Give the best-case and worst-case running times of selection sort in  $\Theta$ -notation.

#### 2.2-3

Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in  $\Theta$ -notation? Justify your answers.

#### 2.2-4

How can we modify almost any algorithm to have a good best-case running time?

---

## 2.3 Designing algorithms

We can choose from a wide range of algorithm design techniques. For insertion sort, we used an **incremental** approach: having sorted the subarray  $A[1 \dots j - 1]$ , we inserted the single element  $A[j]$  into its proper place, yielding the sorted subarray  $A[1 \dots j]$ .

In this section, we examine an alternative design approach, known as “divide-and-conquer,” which we shall explore in more detail in Chapter 4. We’ll use divide-and-conquer to design a sorting algorithm whose worst-case running time is much less than that of insertion sort. One advantage of divide-and-conquer algorithms is that their running times are often easily determined using techniques that we will see in Chapter 4.

### 2.3.1 The divide-and-conquer approach

Many useful algorithms are *recursive* in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a *divide-and-conquer* approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

**Divide** the problem into a number of subproblems that are smaller instances of the same problem.

**Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

**Combine** the solutions to the subproblems into the solution for the original problem.

The *merge sort* algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

**Divide:** Divide the  $n$ -element sequence to be sorted into two subsequences of  $n/2$  elements each.

**Conquer:** Sort the two subsequences recursively using merge sort.

**Combine:** Merge the two sorted subsequences to produce the sorted answer.

The recursion “bottoms out” when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the “combine” step. We merge by calling an auxiliary procedure  $\text{MERGE}(A, p, q, r)$ , where  $A$  is an array and  $p, q$ , and  $r$  are indices into the array such that  $p \leq q < r$ . The procedure assumes that the subarrays  $A[p..q]$  and  $A[q + 1..r]$  are in sorted order. It *merges* them to form a single sorted subarray that replaces the current subarray  $A[p..r]$ .

Our  $\text{MERGE}$  procedure takes time  $\Theta(n)$ , where  $n = r - p + 1$  is the total number of elements being merged, and it works as follows. Returning to our card-playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto

the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile. Computationally, each basic step takes constant time, since we are comparing just the two top cards. Since we perform at most  $n$  basic steps, merging takes  $\Theta(n)$  time.

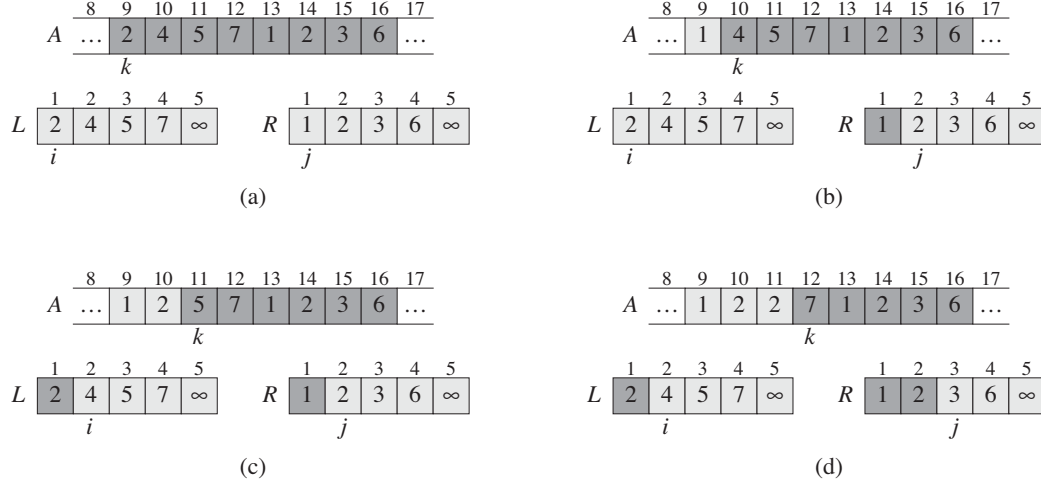
The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. We place on the bottom of each pile a *sentinel* card, which contains a special value that we use to simplify our code. Here, we use  $\infty$  as the sentinel value, so that whenever a card with  $\infty$  is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly  $r - p + 1$  cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

```

MERGE( $A, p, q, r$ )
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 

```

In detail, the MERGE procedure works as follows. Line 1 computes the length  $n_1$  of the subarray  $A[p..q]$ , and line 2 computes the length  $n_2$  of the subarray  $A[q + 1..r]$ . We create arrays  $L$  and  $R$  (“left” and “right”), of lengths  $n_1 + 1$  and  $n_2 + 1$ , respectively, in line 3; the extra position in each array will hold the sentinel. The **for** loop of lines 4–5 copies the subarray  $A[p..q]$  into  $L[1..n_1]$ , and the **for** loop of lines 6–7 copies the subarray  $A[q + 1..r]$  into  $R[1..n_2]$ . Lines 8–9 put the sentinels at the ends of the arrays  $L$  and  $R$ . Lines 10–17, illus-



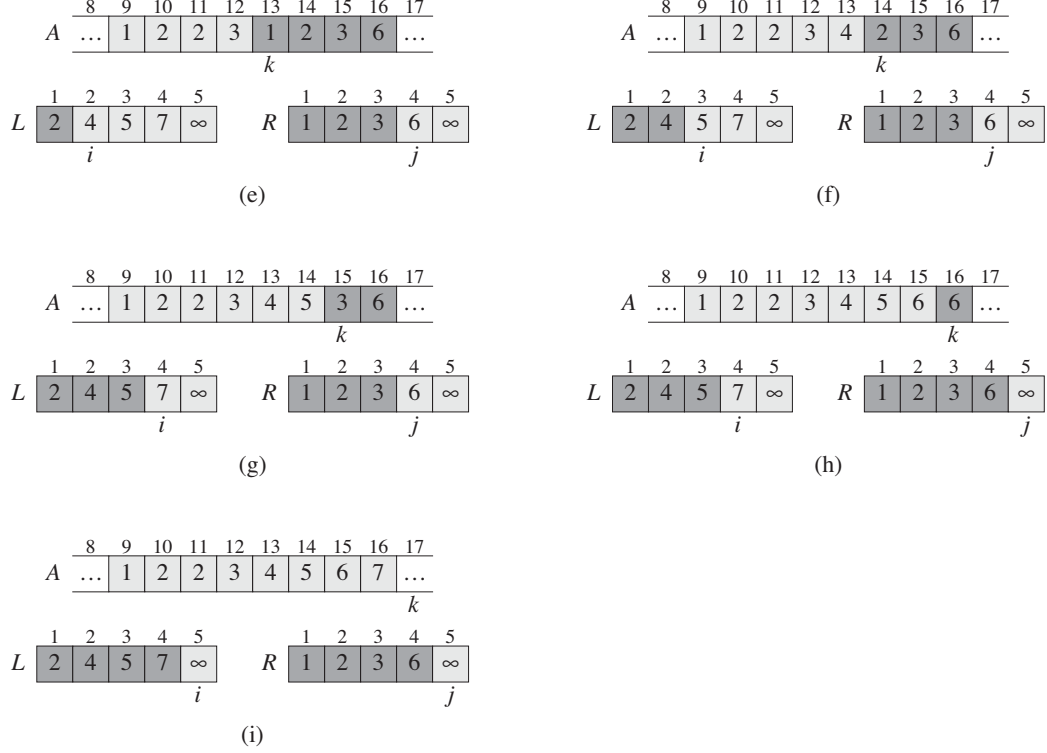
**Figure 2.3** The operation of lines 10–17 in the call `MERGE( $A, 9, 12, 16$ )`, when the subarray  $A[9..16]$  contains the sequence  $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$ . After copying and inserting sentinels, the array  $L$  contains  $\langle 2, 4, 5, 7, \infty \rangle$ , and the array  $R$  contains  $\langle 1, 2, 3, 6, \infty \rangle$ . Lightly shaded positions in  $A$  contain their final values, and lightly shaded positions in  $L$  and  $R$  contain values that have yet to be copied back into  $A$ . Taken together, the lightly shaded positions always comprise the values originally in  $A[9..16]$ , along with the two sentinels. Heavily shaded positions in  $A$  contain values that will be copied over, and heavily shaded positions in  $L$  and  $R$  contain values that have already been copied back into  $A$ . (a)–(h) The arrays  $A$ ,  $L$ , and  $R$ , and their respective indices  $k$ ,  $i$ , and  $j$  prior to each iteration of the loop of lines 12–17.

trated in Figure 2.3, perform the  $r - p + 1$  basic steps by maintaining the following loop invariant:

At the start of each iteration of the **for** loop of lines 12–17, the subarray  $A[p..k-1]$  contains the  $k-p$  smallest elements of  $L[1..n_1+1]$  and  $R[1..n_2+1]$ , in sorted order. Moreover,  $L[i]$  and  $R[j]$  are the smallest elements of their arrays that have not been copied back into  $A$ .

We must show that this loop invariant holds prior to the first iteration of the **for** loop of lines 12–17, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

**Initialization:** Prior to the first iteration of the loop, we have  $k = p$ , so that the subarray  $A[p..k-1]$  is empty. This empty subarray contains the  $k-p = 0$  smallest elements of  $L$  and  $R$ , and since  $i = j = 1$ , both  $L[i]$  and  $R[j]$  are the smallest elements of their arrays that have not been copied back into  $A$ .



**Figure 2.3, continued** (i) The arrays and indices at termination. At this point, the subarray in  $A[9..16]$  is sorted, and the two sentinels in  $L$  and  $R$  are the only two elements in these arrays that have not been copied into  $A$ .

**Maintenance:** To see that each iteration maintains the loop invariant, let us first suppose that  $L[i] \leq R[j]$ . Then  $L[i]$  is the smallest element not yet copied back into  $A$ . Because  $A[p..k-1]$  contains the  $k-p$  smallest elements, after line 14 copies  $L[i]$  into  $A[k]$ , the subarray  $A[p..k]$  will contain the  $k-p+1$  smallest elements. Incrementing  $k$  (in the **for** loop update) and  $i$  (in line 15) reestablishes the loop invariant for the next iteration. If instead  $L[i] > R[j]$ , then lines 16–17 perform the appropriate action to maintain the loop invariant.

**Termination:** At termination,  $k = r + 1$ . By the loop invariant, the subarray  $A[p..k-1]$ , which is  $A[p..r]$ , contains the  $k-p = r-p+1$  smallest elements of  $L[1..n_1+1]$  and  $R[1..n_2+1]$ , in sorted order. The arrays  $L$  and  $R$  together contain  $n_1 + n_2 + 2 = r - p + 3$  elements. All but the two largest have been copied back into  $A$ , and these two largest elements are the sentinels.

To see that the MERGE procedure runs in  $\Theta(n)$  time, where  $n = r - p + 1$ , observe that each of lines 1–3 and 8–11 takes constant time, the **for** loops of lines 4–7 take  $\Theta(n_1 + n_2) = \Theta(n)$  time,<sup>7</sup> and there are  $n$  iterations of the **for** loop of lines 12–17, each of which takes constant time.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT( $A, p, r$ ) sorts the elements in the subarray  $A[p..r]$ . If  $p \geq r$ , the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index  $q$  that partitions  $A[p..r]$  into two subarrays:  $A[p..q]$ , containing  $\lceil n/2 \rceil$  elements, and  $A[q+1..r]$ , containing  $\lfloor n/2 \rfloor$  elements.<sup>8</sup>

MERGE-SORT( $A, p, r$ )

```

1  if  $p < r$ 
2       $q = \lfloor (p + r)/2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )

```

To sort the entire sequence  $A = \langle A[1], A[2], \dots, A[n] \rangle$ , we make the initial call MERGE-SORT( $A, 1, A.length$ ), where once again  $A.length = n$ . Figure 2.4 illustrates the operation of the procedure bottom-up when  $n$  is a power of 2. The algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length  $n/2$  are merged to form the final sorted sequence of length  $n$ .

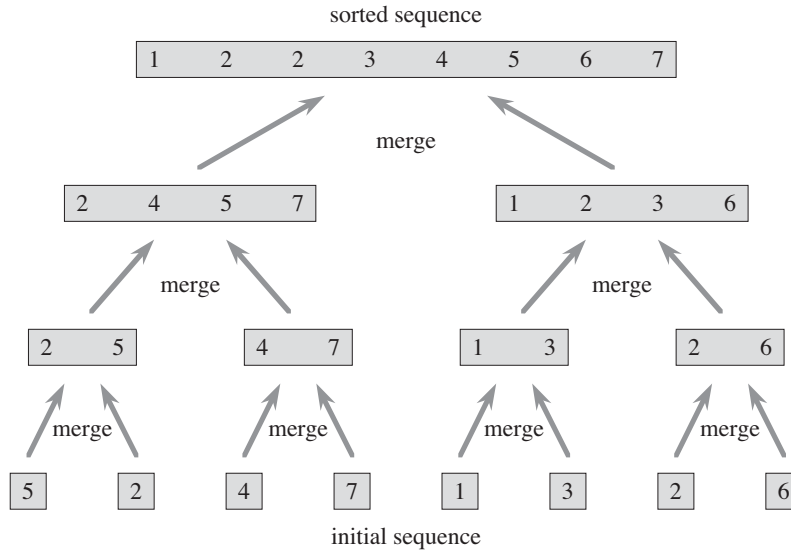
### 2.3.2 Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call to itself, we can often describe its running time by a **recurrence equation** or **recurrence**, which describes the overall running time on a problem of size  $n$  in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

---

<sup>7</sup>We shall see in Chapter 3 how to formally interpret equations containing  $\Theta$ -notation.

<sup>8</sup>The expression  $\lceil x \rceil$  denotes the least integer greater than or equal to  $x$ , and  $\lfloor x \rfloor$  denotes the greatest integer less than or equal to  $x$ . These notations are defined in Chapter 3. The easiest way to verify that setting  $q$  to  $\lfloor (p + r)/2 \rfloor$  yields subarrays  $A[p..q]$  and  $A[q + 1..r]$  of sizes  $\lceil n/2 \rceil$  and  $\lfloor n/2 \rfloor$ , respectively, is to examine the four cases that arise depending on whether each of  $p$  and  $r$  is odd or even.



**Figure 2.4** The operation of merge sort on the array  $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$ . The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic paradigm. As before, we let  $T(n)$  be the running time on a problem of size  $n$ . If the problem size is small enough, say  $n \leq c$  for some constant  $c$ , the straightforward solution takes constant time, which we write as  $\Theta(1)$ . Suppose that our division of the problem yields  $a$  subproblems, each of which is  $1/b$  the size of the original. (For merge sort, both  $a$  and  $b$  are 2, but we shall see many divide-and-conquer algorithms in which  $a \neq b$ .) It takes time  $T(n/b)$  to solve one subproblem of size  $n/b$ , and so it takes time  $aT(n/b)$  to solve  $a$  of them. If we take  $D(n)$  time to divide the problem into subproblems and  $C(n)$  time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

In Chapter 4, we shall see how to solve common recurrences of this form.

### Analysis of merge sort

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that

the original problem size is a power of 2. Each divide step then yields two subsequences of size exactly  $n/2$ . In Chapter 4, we shall see that this assumption does not affect the order of growth of the solution to the recurrence.

We reason as follows to set up the recurrence for  $T(n)$ , the worst-case running time of merge sort on  $n$  numbers. Merge sort on just one element takes constant time. When we have  $n > 1$  elements, we break down the running time as follows.

**Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus,  $D(n) = \Theta(1)$ .

**Conquer:** We recursively solve two subproblems, each of size  $n/2$ , which contributes  $2T(n/2)$  to the running time.

**Combine:** We have already noted that the MERGE procedure on an  $n$ -element subarray takes time  $\Theta(n)$ , and so  $C(n) = \Theta(n)$ .

When we add the functions  $D(n)$  and  $C(n)$  for the merge sort analysis, we are adding a function that is  $\Theta(n)$  and a function that is  $\Theta(1)$ . This sum is a linear function of  $n$ , that is,  $\Theta(n)$ . Adding it to the  $2T(n/2)$  term from the “conquer” step gives the recurrence for the worst-case running time  $T(n)$  of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases} \quad (2.1)$$

In Chapter 4, we shall see the “master theorem,” which we can use to show that  $T(n)$  is  $\Theta(n \lg n)$ , where  $\lg n$  stands for  $\log_2 n$ . Because the logarithm function grows more slowly than any linear function, for large enough inputs, merge sort, with its  $\Theta(n \lg n)$  running time, outperforms insertion sort, whose running time is  $\Theta(n^2)$ , in the worst case.

We do not need the master theorem to intuitively understand why the solution to the recurrence (2.1) is  $T(n) = \Theta(n \lg n)$ . Let us rewrite recurrence (2.1) as

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{if } n > 1, \end{cases} \quad (2.2)$$

where the constant  $c$  represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps.<sup>9</sup>

---

<sup>9</sup>It is unlikely that the same constant exactly represents both the time to solve problems of size 1 and the time per array element of the divide and combine steps. We can get around this problem by letting  $c$  be the larger of these times and understanding that our recurrence gives an upper bound on the running time, or by letting  $c$  be the lesser of these times and understanding that our recurrence gives a lower bound on the running time. Both bounds are on the order of  $n \lg n$  and, taken together, give a  $\Theta(n \lg n)$  running time.



Figure 2.5 shows how we can solve recurrence (2.2). For convenience, we assume that  $n$  is an exact power of 2. Part (a) of the figure shows  $T(n)$ , which we expand in part (b) into an equivalent tree representing the recurrence. The  $cn$  term is the root (the cost incurred at the top level of recursion), and the two subtrees of the root are the two smaller recurrences  $T(n/2)$ . Part (c) shows this process carried one step further by expanding  $T(n/2)$ . The cost incurred at each of the two sub-nodes at the second level of recursion is  $cn/2$ . We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1, each with a cost of  $c$ . Part (d) shows the resulting **recursion tree**.

Next, we add the costs across each level of the tree. The top level has total cost  $cn$ , the next level down has total cost  $c(n/2) + c(n/2) = cn$ , the level after that has total cost  $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$ , and so on. In general, the level  $i$  below the top has  $2^i$  nodes, each contributing a cost of  $c(n/2^i)$ , so that the  $i$ th level below the top has total cost  $2^i c(n/2^i) = cn$ . The bottom level has  $n$  nodes, each contributing a cost of  $c$ , for a total cost of  $cn$ .

The total number of levels of the recursion tree in Figure 2.5 is  $\lg n + 1$ , where  $n$  is the number of leaves, corresponding to the input size. An informal inductive argument justifies this claim. The base case occurs when  $n = 1$ , in which case the tree has only one level. Since  $\lg 1 = 0$ , we have that  $\lg n + 1$  gives the correct number of levels. Now assume as an inductive hypothesis that the number of levels of a recursion tree with  $2^i$  leaves is  $\lg 2^i + 1 = i + 1$  (since for any value of  $i$ , we have that  $\lg 2^i = i$ ). Because we are assuming that the input size is a power of 2, the next input size to consider is  $2^{i+1}$ . A tree with  $n = 2^{i+1}$  leaves has one more level than a tree with  $2^i$  leaves, and so the total number of levels is  $(i + 1) + 1 = \lg 2^{i+1} + 1$ .

To compute the total cost represented by the recurrence (2.2), we simply add up the costs of all the levels. The recursion tree has  $\lg n + 1$  levels, each costing  $cn$ , for a total cost of  $cn(\lg n + 1) = cn \lg n + cn$ . Ignoring the low-order term and the constant  $c$  gives the desired result of  $\Theta(n \lg n)$ .

## Exercises

### 2.3-1

Using Figure 2.4 as a model, illustrate the operation of merge sort on the array  $A = \{3, 41, 52, 26, 38, 57, 9, 49\}$ .

### 2.3-2

Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array  $L$  or  $R$  has had all its elements copied back to  $A$  and then copying the remainder of the other array back into  $A$ .