the algorithm completely and analyze the number of comparisons it requires.

4. In Algorithm 3.6, what happens if lines 7 to 17 are dropped? Does the resultant function still compute the maximum and minimum elements correctly?

3.4 MERGE SORT

As another example of divide-and-conquer, we investigate a sorting algorithm that has the nice property that in the worst case its complexity is $O(n \log n)$. This algorithm is called *merge sort.* We assume throughout that the elements are to be sorted in nondecreasing order. Given a sequence of n elements (also called keys) $a[1], \ldots, a[n]$, the general idea is to imagine them split into two sets $a[1], \ldots, a[\lfloor n/2 \rfloor]$ and $a[\lfloor n/2 \rfloor + 1], \ldots, a[n]$. Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of n elements. Thus we have another ideal example of the divide-and-conquer strategy in which the splitting is into two equal-sized sets and the combining operation is the merging of two sorted sets into one.

MergeSort (Algorithm 3.7) describes this process very succinctly using recursion and a function Merge (Algorithm 3.8) which merges two sorted sets. Before executing MergeSort, the n elements should be placed in $a[1 : n]$. Then MergeSort(1,n) causes the keys to be rearranged into nondecreasing order in a.

Example 3.7 Consider the array of ten elements $a[1 : 10]$ = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520). Algorithm MergeSort begins by splitting $a[\,]$ into two subarrays each of size five ($a[1 : 5]$ and $a[6 : 10]$). The elements in $a[1 : 5]$ are then split into two subarrays of size three ($a[1 : 3]$) and two ($a[4 : 5]$). Then the items in $a[1 : 3]$ are split into subarrays of size two ($a[1 : 2]$) and one ($a[3 : 3]$). The two values in $a[1 : 2]$ are split a final time into one-element subarrays, and now the merging begins. Note that no movement of data has yet taken place. A record of the subarrays is implicitly maintained by the recursive mechanism. Pictorially the file can now be viewed as

$$(310 \mid 285 \mid 179 \mid 652, 351 \mid 423, 861, 254, 450, 520)$$

where vertical bars indicate the boundaries of subarrays. Elements $a[1]$ and $a[2]$ are merged to yield

$$(285, 310 \mid 179 \mid 652, 351 \mid 423, 861, 254, 450, 520)$$

```
Algorithm MergeSort($low, high$)
// $a[low : high]$ is a global array to be sorted.
// Small($P$) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if ($low < high$) then  // If there are more than one element
    {
        // Divide $P$ into subproblems.
            // Find where to split the set.
                $mid := \lfloor (low + high)/2 \rfloor$;
        // Solve the subproblems.
            MergeSort($low, mid$);
            MergeSort($mid + 1, high$);
        // Combine the solutions.
            Merge($low, mid, high$);
    }
}
```

Algorithm 3.7 Merge sort

Then $a[3]$ is merged with $a[1 : 2]$ and

(179, 285, 310 | 652, 351 | 423, 861, 254, 450, 520)

is produced. Next, elements $a[4]$ and $a[5]$ are merged:

(179, 285, 310 | 351, 652 | 423, 861, 254, 450, 520)

and then $a[1 : 3]$ and $a[4 : 5]$:

(179, 285, 310, 351, 652 | 423, 861, 254, 450, 520)

At this point the algorithm has returned to the first invocation of MergeSort and is about to process the second recursive call. Repeated recursive calls are invoked producing the following subarrays:

(179, 285, 310, 351, 652 | 423 | 861 | 254 | 450, 520)

Elements $a[6]$ and $a[7]$ are merged. Then $a[8]$ is merged with $a[6 : 7]$:

```
Algorithm Merge($low, mid, high$)
// $a[low : high]$ is a global array containing two sorted
// subsets in $a[low : mid]$ and in $a[mid + 1 : high]$. The goal
// is to merge these two sets into a single set residing
// in $a[low : high]$. $b[\,]$ is an auxiliary global array.
{
    $h := low$; $i := low$; $j := mid + 1$;
    while (($h \le mid$) and ($j \le high$)) do
    {
        if ($a[h] \le a[j]$) then
        {
            $b[i] := a[h]$;$h := h + 1$;
        }
        else
        {
            $b[i] := a[j]$; $j := j + 1$;
        }
        $i := i + 1$;
    }
    if ($h > mid$) then
        for $k := j$ to $high$ do
        {
            $b[i] := a[k]$; $i := i + 1$;
        }
    else
        for $k := h$ to $mid$ do
        {
            $b[i] := a[k]$; $i := i + 1$;
        }
    for $k := low$ to $high$ do $a[k] := b[k]$;
}
```

Algorithm 3.8 Merging two sorted subarrays using auxiliary storage

$$(179, 285, 310, 351, 652 \mid 254, 423, 861 \mid 450, 520)$$

Next $a[9]$ and $a[10]$ are merged, and then $a[6:8]$ and $a[9:10]$:

$$(179, 285, 310, 351, 652 \mid 254, 423, 450, 520, 861)$$

At this point there are two sorted subarrays and the final merge produces the fully sorted result

$$(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)$$

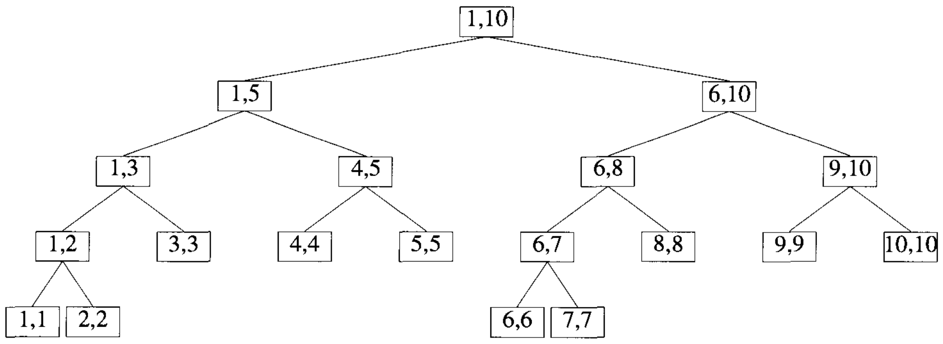

Figure 3.3 Tree of calls of MergeSort$(1, 10)$

Figure 3.3 is a tree that represents the sequence of recursive calls that are produced by MergeSort when it is applied to ten elements. The pair of values in each node are the values of the parameters *low* and *high*. Notice how the splitting continues until sets containing a single element are produced. Figure 3.4 is a tree representing the calls to procedure Merge by MergeSort. For example, the node containing 1, 2, and 3 represents the merging of $a[1:2]$ with $a[3]$. □

If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

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

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$

It is easy to see that if $2^k < n \le 2^{k+1}$, then $T(n) \le T(2^{k+1})$. Therefore

$$T(n) = O(n \log n)$$

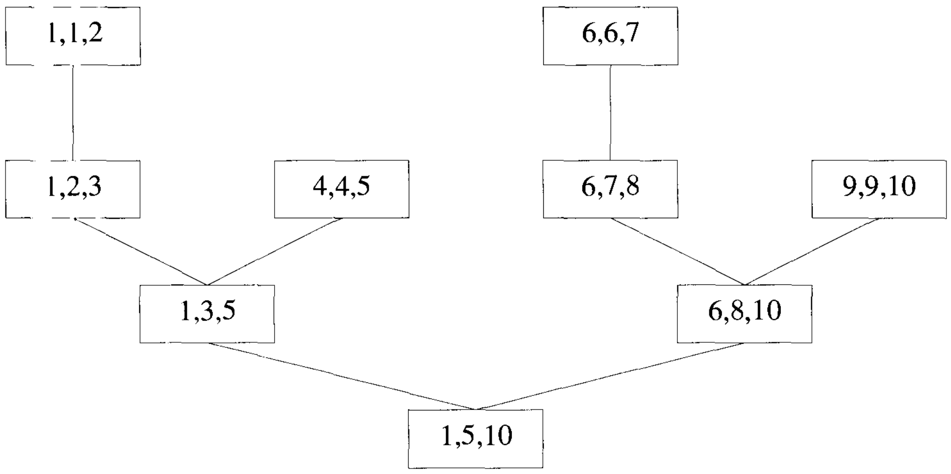

Figure 3.4 Tree of calls of Merge

Though Algorithm 3.7 nicely captures the divide-and-conquer nature of merge sort, there remain several inefficiencies that can and should be eliminated. We present these refinements in an attempt to produce a version of merge sort that is good enough to execute. Despite these improvements the algorithm's complexity remains $O(n \log n)$. We see in Chapter 10 that no sorting algorithm based on comparisons of entire keys can do better.

One complaint we might raise concerning merge sort is its use of $2n$ locations. The additional n locations were needed because we couldn't reasonably merge two sorted sets in place. But despite the use of this space the algorithm must still work hard and copy the result placed into $b[low : high]$ back into $a[low : high]$ on each call of Merge. An alternative to this copying is to associate a new field of information with each key. (The elements in $a[\ ]$ are called *keys*.) This field is used to link the keys and any associated information together in a sorted list (keys and related information are called *records*). Then the merging of the sorted lists proceeds by changing the link values, and no records need be moved at all. A field that contains only a link will generally be smaller than an entire record, so less space will be used.

Along with the original array $a[\ ]$, we define an auxiliary array $link[1 : n]$ that contains integers in the range $[0, n]$. These integers are interpreted as pointers to elements of $a[\ ]$. A list is a sequence of pointers ending with a zero. Below is one set of values for *link* that contains two lists: Q and R. The integer $Q = 2$ denotes the start of one list and $R = 5$ the start of the other.

link:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	6	4	7	1	3	0	8	0

The two lists are $Q = (2, 4, 1, 6)$ and $R = (5, 3, 7, 8)$. Interpreting these lists as describing sorted subsets of $a[1 : 8]$, we conclude that $a[2] \le a[4] \le a[1] \le a[6]$ and $a[5] \le a[3] \le a[7] \le a[8]$.

Another complaint we could raise about MergeSort is the stack space that is necessitated by the use of recursion. Since merge sort splits each set into two approximately equal-sized subsets, the maximum depth of the stack is proportional to log n. The need for stack space seems indicated by the top-down manner in which this algorithm was devised. The need for stack space can be eliminated if we build an algorithm that works bottom-up; see the exercises for details.

As can be seen from function MergeSort and the previous example, even sets of size two will cause two recursive calls to be made. For small set sizes most of the time will be spent processing the recursion instead of sorting. This situation can be improved by not allowing the recursion to go to the lowest level. In terms of the divide-and-conquer control abstraction, we are suggesting that when Small is true for merge sort, more work should be done than simply returning with no action. We use a second sorting algorithm that works well on small-sized sets.

Insertion sort works exceedingly fast on arrays of less than, say, 16 elements, though for large n its computing time is $O(n^2)$. Its basic idea for sorting the items in $a[1 : n]$ is as follows:

```
for $j := 2$ to $n$ do {
    place $a[j]$ in its correct position in the sorted set $a[1 : j - 1]$;
}
```

Though all the elements in $a[1:j-1]$ may have to be moved to accommodate $a[j]$, for small values of n the algorithm works well. Algorithm 3.9 has the details.

```
Algorithm InsertionSort($a, n$)
// Sort the array $a[1:n]$ into nondecreasing order, $n \geq 1$.
{
    for $j := 2$ to $n$ do
    {
        // $a[1:j-1]$ is already sorted.
        $item := a[j]$; $i := j-1$;
        while (($i \geq 1$) and ($item < a[i]$)) do
        {
            $a[i+1] := a[i]$; $i := i-1$;
        }
        $a[i+1] := item$;
    }
}
```

Algorithm 3.9 Insertion sort

The statements within the **while** loop can be executed zero up to a maximum of j times. Since j goes from 2 to n, the worst-case time of this procedure is bounded by

$$\sum_{2 \leq j \leq n} j = n(n+1)/2 - 1 = \Theta(n^2)$$

Its best-case computing time is $\Theta(n)$ under the assumption that the body of the **while** loop is never entered. This will be true when the data is already in sorted order.

We are now ready to present the revised version of merge sort with the inclusion of insertion sort and the links. Function MergeSort1 (Algorithm 3.10) is initially invoked by placing the keys of the records to be sorted in $a[1:n]$ and setting $link[1:n]$ to zero. Then one says MergeSort1$(1, n)$. A pointer to a list of indices that give the elements of $a[\ ]$ in sorted order is returned. Insertion sort is used whenever the number of items to be sorted is less than 16. The version of insertion sort as given by Algorithm 3.9 needs to be altered so that it sorts $a[low:high]$ into a linked list. Call the altered version InsertionSort1. The revised merging function, Merge1, is given in Algorithm 3.11.

```
Algorithm MergeSort1($low, high$)
// The global array $a[low : high]$ is sorted in nondecreasing order
// using the auxiliary array $link[low : high]$. The values in $link$
// represent a list of the indices $low$ through $high$ giving $a[\,]$ in
// sorted order. A pointer to the beginning of the list is returned.
{
    if (($high - low$) < 15) then
        return InsertionSort1($a, link, low, high$);
    else
    {
        $mid := \lfloor (low + high)/2 \rfloor$;
        $q :=$ MergeSort1($low, mid$);
        $r :=$ MergeSort1($mid + 1, high$);
        return Merge1($q, r$);
    }
}
```

Algorithm 3.10 Merge sort using links

Example 3.8 As an aid to understanding this new version of merge sort, suppose we simulate the algorithm as it sorts the eight-element sequence (50, 10, 25, 30, 15, 70, 35, 55). We ignore the fact that less than 16 elements would normally be sorted using InsertionSort. The *link* array is initialized to zero. Table 3.4 shows how the *link* array changes after each call of MergeSort1 completes. On each row the value of p points to the list in *link* that was created by the last completion of Merge1. To the right are the subsets of sorted elements that are represented by these lists. For example, in the last row $p = 2$ which begins the list of links 2, 5, 3, 4, 7, 1, 8, and 6; this implies $a[2] \leq a[5] \leq a[3] \leq a[4] \leq a[7] \leq a[1] \leq a[8] \leq a[6]$. □

EXERCISES

1. Why is it necessary to have the auxiliary array $b[low : high]$ in function Merge? Give an example that shows why in-place merging is inefficient.

2. The worst-case time of procedure MergeSort is $O(n \log n)$. What is its best-case time? Can we say that the time for MergeSort is $\Theta(n \log n)$?

3. A sorting method is said to be *stable* if at the end of the method, identical elements occur in the same order as in the original unsorted

```
Algorithm Merge1($q, r$)
// $q$ and $r$ are pointers to lists contained in the global array
// $link[0:n]$. $link[0]$ is introduced only for convenience and need
// not be initialized. The lists pointed at by $q$ and $r$ are merged
// and a pointer to the beginning of the merged list is returned.
{
    $i := q$; $j := r$; $k := 0$;
    // The new list starts at $link[0]$.
    while (($i \neq 0$) and ($j \neq 0$)) do
    { // While both lists are nonempty do
        if ($a[i] \leq a[j]$) then
        { // Find the smaller key.
            $link[k] := i$; $k := i$; $i := link[i]$;
            // Add a new key to the list.
        }
        else
        {
            $link[k] := j$; $k := j$; $j := link[j]$;
        }
    }
    if ($i = 0$) then $link[k] := j$;
    else $link[k] := i$;
    return $link[0]$;
}
```

Algorithm 3.11 Merging linked lists of sorted elements

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	
a:	-	50	10	25	30	15	70	35	55	
link:	0	0	0	0	0	0	0	0	0	
q r p										
1 2 2	2	0	1	0	0	0	0	0	0	(10, 50)
3 4 3	3	0	1	4	0	0	0	0	0	(10, 50), (25, 30)
2 3 2	2	0	3	4	1	0	0	0	0	(10, 25, 30, 50)
5 6 5	5	0	3	4	1	6	0	0	0	(10, 25, 30, 50), (15, 70)
7 8 7	7	0	3	4	1	6	0	8	0	(10, 25, 30, 50), (15, 70), (35, 55)
5 7 5	5	0	3	4	1	7	0	8	6	(10, 25, 30, 50) (15, 35, 55, 70)
2 5 2	2	8	5	4	7	3	0	1	6	(10, 15, 25, 30, 35, 50, 55, 70)

MergeSort1 applied to $a[1:8] = (50, 10, 25, 30, 15, 70, 35, 55)$

Table 3.4 Example of *link* array changes

set. Is merge sort a stable sorting method?

4. Suppose $a[1:m]$ and $b[1:n]$ both contain sorted elements in non-decreasing order. Write an algorithm that merges these items into $c[1:m+n]$. Your algorithm should be shorter than Algorithm 3.8 (Merge) since you can now place a large value in $a[m+1]$ and $b[n+1]$.

5. Given a file of n records that are partially sorted as $x_1 \leq x_2 \leq \cdots \leq x_m$ and $x_{m+1} \leq \cdots \leq x_n$, is it possible to sort the entire file in time $O(n)$ using only a small fixed amount of additional storage?

6. Another way to sort a file of n records is to scan the file, merge consecutive pairs of size one, then merge pairs of size two, and so on. Write an algorithm that carries out this process. Show how your algorithm works on the data set (100, 300, 150, 450, 250, 350, 200, 400, 500).

7. A version of insertion sort is used by Algorithm 3.10 to sort small subarrays. However, its parameters and intent are slightly different from the procedure InsertionSort of Algorithm 3.9. Write a version of insertion sort that will work as Algorithm 3.10 expects.

8. The sequences $X_1, X_2, \ldots, X_\ell$ are sorted sequences such that $\sum_{i=1}^{\ell} |X_i| = n$. Show how to merge these ℓ sequences in time $O(n \log \ell)$.

3.5 QUICKSORT

The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort. In merge sort, the file $a[1:n]$ was divided at its midpoint into subarrays which were independently sorted and later merged. In quicksort, the division into two subarrays is made so that the sorted subarrays do not need to be merged later. This is accomplished by rearranging the elements in $a[1:n]$ such that $a[i] \le a[j]$ for all i between 1 and m and all j between $m+1$ and n for some m, $1 \le m \le n$. Thus, the elements in $a[1:m]$ and $a[m+1:n]$ can be independently sorted. No merge is needed. The rearrangement of the elements is accomplished by picking some element of $a[\ ]$, say $t = a[s]$, and then reordering the other elements so that all elements appearing before t in $a[1:n]$ are less than or equal to t and all elements appearing after t are greater than or equal to t. This rearranging is referred to as *partitioning*.

Function Partition of Algorithm 3.12 (due to C. A. R. Hoare) accomplishes an in-place partitioning of the elements of $a[m:p-1]$. It is assumed that $a[p] \ge a[m]$ and that $a[m]$ is the partitioning element. If $m = 1$ and $p-1 = n$, then $a[n+1]$ must be defined and must be greater than or equal to all elements in $a[1:n]$. The assumption that $a[m]$ is the partition element is merely for convenience; other choices for the partitioning element than the first item in the set are better in practice. The function Interchange(a, i, j) exchanges $a[i]$ with $a[j]$.

Example 3.9 As an example of how Partition works, consider the following array of nine elements. The function is initially invoked as Partition$(a, 1, 10)$. The ends of the horizontal line indicate those elements which were interchanged to produce the next row. The element $a[1] = 65$ is the partitioning element and it is eventually (in the sixth row) determined to be the fifth smallest element of the set. Notice that the remaining elements are unsorted but partitioned about $a[5] = 65$. □

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

Using Hoare's clever method of partitioning a set of elements about a chosen element, we can directly devise a divide-and-conquer method for completely sorting n elements. Following a call to the function Partition, two sets S_1 and S_2 are produced. All elements in S_1 are less than or equal