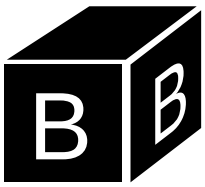


Chapter *20*

Recursive Searching and Sorting

Any algorithm that uses a loop to perform its processing can be written without a loop using recursion. In particular, the iterative searching and sorting algorithms we learned in Chapter 17 can all be written recursively. This chapter presents a recursive version of the binary search and a taxonomy of sorting algorithms based on recursion.

Recursive binary search

Figure 20.1 shows the input window and dialog box for a program that tests a recursive version of the binary search. As far as the user is concerned, there is no difference between a recursive and an iterative version of the search. The user loads a vector of values into a one-dimensional array, enters a number to search for, and clicks the LookUp button. The dialog box responds with the position of the number in the vector.

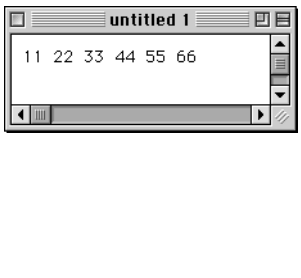


Figure 20.1
The input window and dialog box for testing a binary search algorithm.

The idea of a recursive solution to a problem is to assume that the procedure can call itself to solve a smaller problem. With the binary search, the smaller problem is the same search carried out on either the first half or second half of the list. The parameter list of the procedure, therefore, must include the indices between which the search for the smaller problem is carried out. The complete procedure heading is

```
PROCEDURE Search (IN v: ARRAY OF INTEGER; first, last, srchNum: INTEGER;
      OUT i: INTEGER; OUT fnd: BOOLEAN);
```

v is the array to search, first and last are the indices in v between which the search is to take place, and srchNum is the number to be searched. If srchNum is in the list, the procedure sets fnd to true and i to its location. Otherwise, it sets fnd to false. Figure 20.2 shows the procedure in a module that implements the dialog box in Figure 20.1.

Procedure Search contains no loop. The test for the basis is whether first is greater than last. If it is, the indices have crossed as shown in Figure 16.11(d), and the item is not in the list. Further recursive calls are not needed and fnd can be set to false. Otherwise, the midpoint between first and last is computed, and srchNum is compared to v[mid]. Depending on the result of this comparison, a recursive call is made on the first half of the list, or on the last half of the list, or srchNum was found, in which case fnd is set to true and no further recursive calls are necessary.

```
MODULE Pbox20A;
   IMPORT Dialog, TextModels, TextControllers, PboxMappers, PboxStrings;
   VAR
      d*: RECORD
         numItems-: INTEGER;
         searchNumber*: INTEGER;
         indexString-: ARRAY 16 OF CHAR;
      END;
      list: ARRAY 1024 OF INTEGER;

   PROCEDURE LoadList*;
      VAR
         md: TextModels.Model;
         cn: TextControllers.Controller;
         sc: PboxMappers.Scanner;
   BEGIN
      cn := TextControllers.Focus();
      IF cn # NIL THEN
         md := cn.text;
         sc.ConnectTo(md);
         sc.ScanIntVector(list, d.numItems)
      END;
      Dialog.Update(d)
   END LoadList;
```

Figure 20.2
A recursive version of the binary search algorithm.

```
    PROCEDURE Search (IN v: ARRAY OF INTEGER; first, last, srchNum: INTEGER;
          OUT i: INTEGER; OUT fnd: BOOLEAN);
       VAR
          mid: INTEGER;
    BEGIN
       ASSERT((0 <= first) & (last < LEN(v)), 20);
       IF first > last THEN
          fnd := FALSE
       ELSE
          mid := (first + last) DIV 2;
          IF srchNum < v[mid] THEN
             Search(v, first, mid - 1, srchNum, i, fnd)
          ELSIF srchNum > v[mid] THEN
             Search(v, mid + 1, last, srchNum, i, fnd)
          ELSE
             fnd := TRUE;
             i := mid
          END
       END
    END Search;

    PROCEDURE LookUp*;
       VAR
          j: INTEGER;
          found: BOOLEAN;
    BEGIN
       Search(list, 0, d.numItems - 1, d.searchNumber, j, found);
       IF found THEN
          PboxStrings.IntToString(j, 1, d.indexString)
       ELSE
          d.indexString := "No entry"
       END;
       Dialog.Update(d)
    END LookUp;

BEGIN
    d.numItems := 0;
    d.searchNumber := 0; d.indexString := ""
END Pbox20A.
```

Figure 20.2
Continued.

The Merritt sort taxonomy

The idea of a recursive sort is to sort a large list assuming you can recursively sort a smaller part of the list. Figure 20.3 shows the general approach. Suppose you have a list of elements, *L*. To sort the list, you split it into two sublists, *L*1 and *L*2. The sublists are each smaller than the original list, *L*. The recursive idea lets you assume that you have the solution to the problem of sorting the smaller lists. So you recursively sort *L*1, producing the sorted sublist *L*1′. Then you recursively sort *L*2, producing the sorted sublist *L*2′. The last step is to join the two sorted sublists, *L*1′ and *L*2′, into the final sorted list, *L*′.

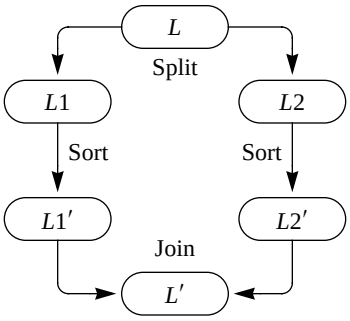
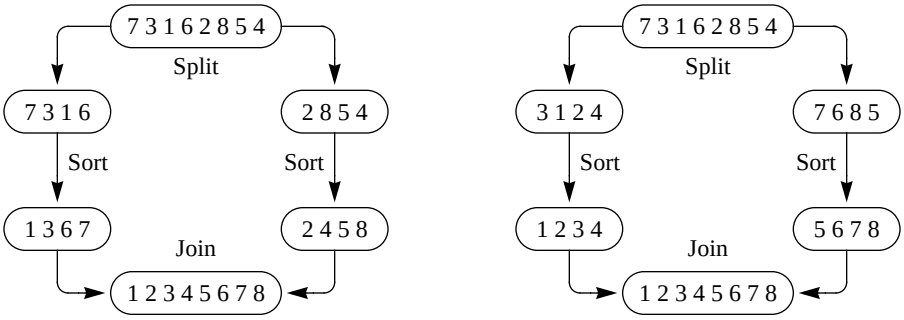


Figure 20.3
The general sort algorithm in the Merritt sort taxonomy.

Merge sort and quick sort

There are two basic sort algorithms, which differ in the methods they use to perform the split and the join. The two are the merge sort algorithm and the quick sort algorithm, shown in Figure 20.4. The classification of sort algorithms into these two families is known as the Merritt taxonomy after Susan Merritt, who proposed it in 1985.



(a) The merge sort algorithm.

(b) The quick sort algorithm.

Figure 20.4
The two basic sort algorithms in the Merritt sort taxonomy.

The figure shows the original unsorted list, L, as the eight values

7 3 1 6 2 8 5 4

for both algorithms. The final list for both is

1 2 3 4 5 6 7 8

which is the sorted list, L'.

The idea behind merge sort

The merge sort algorithm performs a simple split. It takes $L1$ as the first half of the list

7 3 1 6

and $L2$ as the second half of the list

2 8 5 4

It recursively sorts the sublists, producing the sorted sublist $L1'$ as

1 3 6 7

and the sorted sublist $L2'$ as

2 4 5 8

The last step is to merge these two sublists into a single sorted list, L'. You can see that the split of L into $L1$ and $L2$ is easy. You simply take the left half of L as $L1$ and the right half as $L2$. On the other hand, the join is hard. It requires a loop to cycle through the sublists, selecting the smallest number at each step to place in the merged list.

The idea behind quick sort

The quick sort algorithm splits the original list, L, such that every element in the sublist $L1$ is at most the median value, and every element in the sublist $L2$ is at least the median value. It follows that every element in $L1$ will be less than or equal to every element in $L2$. The sublist $L1$ is

3 1 2 4

and the sublist $L2$ is

7 6 8 5

The algorithm sorts $L1$ recursively into the list $L1'$

1 2 3 4

and $L2$ recursively into the list $L2'$

5 6 7 8

Then it joins $L1'$ and $L2'$ into the final sorted list, L'. You can see that the split of L into $L1$ and $L2$ is hard. It requires a loop that somehow compares the elements in the list with each other and moves the smaller elements to the left and the larger ones to the right. On the other hand, the join is easy. It does not require any further comparisons in a loop, the way the join in the merge sort does.

Insertion sort and selection sort

Figure 20.5 shows the special cases of the merge sort and quick sort when the split of n elements subdivides the list such that $L1$ has $n - 1$ elements and $L2$ has one element.

When $L2$ has a single element, the merge sort algorithm simply picks the rightmost element in the list during the split operation. In Figure 20.5(a), the rightmost

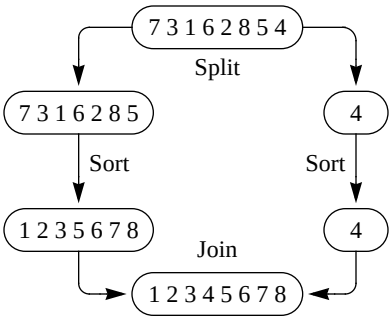
element is 4 which is split off into $L2$. $L1$ is the sublist

7 3 1 6 2 8 5

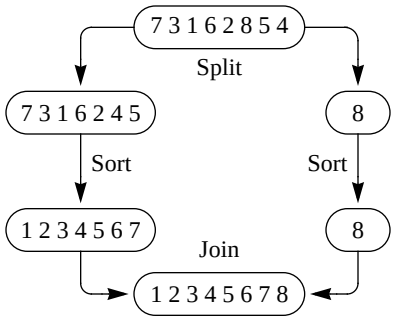
The algorithm recursively sorts the sublist $L1$ into

1 2 3 5 6 7 8

but does not need to sort $L2$ because $L2$ has only one element. Then, it joins $L1'$ and



(a) The insertion sort algorithm.



(b) The selection sort algorithm.

Figure 20.5
Sorting with a split of one element.

$L2'$ by inserting the single element from $L2'$ into $L1'$. The insertion process requires a simple loop to shift the lower elements down one slot to make room for the element from $L2$. The merge sort with a split of one element is called the insertion sort. The insert operation is really a merge of two lists where one of the lists has a single element.

When $L2$ has a single element, the quick sort algorithm must select the largest value from L to put in $L2$. Figure 20.5(b) shows that the largest element from the original list is 8. After it is selected from the original list, $L1$ is left as

7 3 1 6 2 4 5

The selection process requires a simple loop to find the index of the largest value. After the index is computed, an exchange puts the largest value in $L2$. In this example, the largest value 8 is exchanged with 4. The algorithm sorts the sublist $L1$ into

1 2 3 4 5 6 7

but it does not need to sort $L2$ because $L2$ has only one element. The quick sort with a split of one element is called the selection sort.

Nonrecursive versions of the insertion sort and selection sort

You can program the selection and insertion sorts recursively or nonrecursively. Figure 20.6 is a nonrecursive trace of the single-element sort algorithms with the same original unsorted list, L, as in the previous figure. The shaded areas are those regions that are guaranteed to be in order after each pass of the algorithm.

Initial list	7	3	1	6	2	8	5	4
Pass 1, insert 3	3	7	1	6	2	8	5	4
Pass 2, insert 1	1	3	7	6	2	8	5	4
Pass 3, insert 6	1	3	6	7	2	8	5	4
Pass 4, insert 2	1	2	3	6	7	8	5	4
Pass 5, insert 8	1	2	3	6	7	8	5	4
Pass 6, insert 5	1	2	3	5	6	7	8	4
Pass 7, insert 4	1	2	3	4	5	6	7	8

(a) The insertion sort algorithm.

Initial list	7	3	1	6	2	8	5	4
Pass 1, select 8	7	3	1	6	2	4	5	8
Pass 2, select 7	5	3	1	6	2	4	7	8
Pass 3, select 6	5	3	1	4	2	6	7	8
Pass 4, select 5	2	3	1	4	5	6	7	8
Pass 5, select 4	2	1	3	4	5	6	7	8
Pass 6, select 3	2	1	3	4	5	6	7	8
Pass 7, select 2	1	2	3	4	5	6	7	8

(b) The selection sort algorithm.

Figure 20.6
Nonrecursive traces of the single element sort algorithms.

The nonrecursive version of the insertion sort begins by inserting 3 into the sublist

7

producing the sorted sublist

3 7

Then it inserts 1 into this list, producing the sorted sublist

1 3 7

and so on.

Procedure Sort in Figure 16.12 is a nonrecursive implementation of the selection sort, which is traced in Figure 16.13. Figure 20.6(b) is also a trace of the nonrecursive selection sort, but with the same list as in Figure 20.6(a). The nonrecursive sort begins by selecting 8 and exchanging it with the last element of the list. Then, it selects 7 and exchange it with the penultimate element, and so on.

Figure 20.7 summarizes the four basic sort algorithms. Many other sort algorithms have been invented, but most fall into one of the two basic families, either merge sort or quick sort.

Quick sort

Figure 20.4(b) shows the ideal quick sort split. That figure had an original list, L, of eight items. The algorithm split L exactly in half, with four items in $L1$ and four in $L2$. The median value of a list of items is that value, m, such that there are as many items less than m as greater than m. If you knew the median value, you could split the list exactly in half. Unfortunately, the only way to determine the median value is

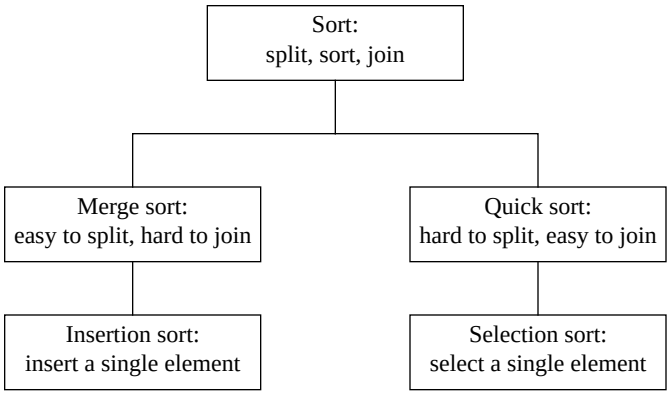
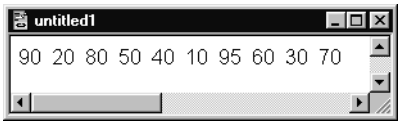
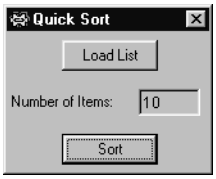


Figure 20.7
Summary of the sort algorithms.

to sort the list and pick the middle item. But you need the median value to sort the list in the first place. The only thing you can do in the face of this dilemma is to be satisfied with a less-than-ideal split.

Figure 20.8 shows the dialog box and input focus window for the quick sort algorithm. As with the recursive version of the binary search, it is impossible for the user to tell whether the sort is done iteratively or recursively. Figure 20.9 is an implementation of the quick sort algorithm.





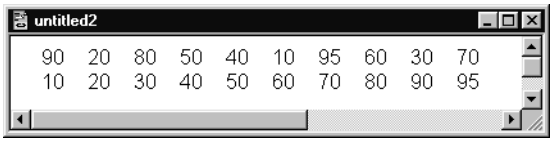


Figure 20.8
The input and output for the quick sort algorithm.

```
MODULE Pbox20B;
   IMPORT Dialog, TextModels, TextViews, Views, TextControllers, PboxMappers;
   VAR
      d*: RECORD
         numItems-: INTEGER;
      END;
      list: ARRAY 1024 OF INTEGER;
```

Figure 20.9
An implementation of the quick sort algorithm.

Figure 20.9
Continued.

```
PROCEDURE LoadList*;
    VAR
        md: TextModels.Model;
        cn: TextControllers.Controller;
        sc: PboxMappers.Scanner;
BEGIN
    cn := TextControllers.Focus();
    IF cn # NIL THEN
        md := cn.text;
        sc.ConnectTo(md);
        sc.ScanIntVector(list, d.numItems);
        Dialog.Update(d)
    END;
END LoadList;

PROCEDURE QuickSort (VAR v: ARRAY OF INTEGER; first, last: INTEGER);
    (* Sorts the items of array v between v[first] and v[last]. *)
    VAR
        i, j: INTEGER;
        key: INTEGER;
        temp: INTEGER;
BEGIN
    ASSERT((0 <= first) & (first <= last) & (last < LEN(v)) OR (last < 0) & (0 <= first), 20);
    IF first < last THEN
        key := v[(first + last) DIV 2];
        i := first;
        j := last;
        (* Invariant 1: key <= v[j + 1..last]. *)
        (* Invariant 2: v[first..i - 1] <= key. *)
        (* Invariant 3: if i <= j, there exists k in [first..j] such that v[k] <= key. *)
        (* Invariant 4: if i <= j, there exists k in [i..last] such that key <= v[k]. *)
        WHILE i <= j DO
            WHILE v[i] < key DO
                INC(i)
            END;
            WHILE key < v[j] DO
                DEC(j)
            END;
            IF i <= j THEN
                temp := v[j];
                v[j] := v[i];  (* Establish invariant 4. *)
                v[i] := temp;  (* Establish invariant 3. *)
                INC(i);  (* Establish invariant 2. *)
                DEC(j)  (* Establish invariant 1. *)
            END
        END;
        QuickSort (v, first, i - 1);
        QuickSort (v, i, last)
    END
END QuickSort;
```

Figure 20.9
Continued.

```
    PROCEDURE SortList*;
        VAR
            md: TextModels.Model;
            vw: TextViews.View;
            fm: PboxMappers.Formatter;
    BEGIN
        md := TextModels.dir.New();
        fm.ConnectTo(md);
        fm.WriteIntVector(list, d.numItems, 4); fm.WriteLn;
        QuickSort(list, 0, d.numItems - 1);
        fm.WriteIntVector(list, d.numItems, 4); fm.WriteLn;
        vw := TextViews.dir.New(md);
        Views.OpenView(vw)
    END SortList;

BEGIN
    d.numItems := 0
END Pbox20B.
```

The precondition for procedure QuickSort has two disjuncts. The first disjunct

```
(0 <= first) & (first <= last) & (last < LEN(v))
```

is for the case of a nonempty array, and the second disjunct

```
(last < 0) & (0 <= first)
```

is for the case of an empty array. If there are no items in the focus window d.numItems gets 0 when list is scanned. Because the call from procedure SortList is

```
QuickSort(list, 0, d.numItems - 1)
```

formal parameter first gets 0 and formal parameter last gets –1. The second disjunct is true, which allows procedure QuickSort to handle the case of an empty array.

Procedure QuickSort picks the middle item of the unsorted list and hopes it is close to the median value. The value it picks is called the *key*. If the key is less than the true median, list $L1$ will contain fewer items than list $L2$. If the key is greater than the true median, $L1$ will contain more items.

You could be extremely unlucky and have the key be the smallest value in the list, in which case $L1$ will have only one value. Or if the key is the largest value, $L2$ will have only one value. On the other hand, you could be extremely lucky and have the key be the true median. You must be content to let the key be what it will be, and accept the average behavior of the algorithm.

Figure 20.10 is a trace of the first call to procedure QuickSort in Figure 20.9. Procedure SortList calls the QuickSort with a value of 0 for first and 9 for last. As Figure 20.10(a) shows, QuickSort initializes i to first and j to last. It computes key as 40.

The WHILE loop splits the list into sublists $L1$ and $L2$. The two nested WHILE loops increase i and decrease j until i finds the value 90, which is greater than key, and j finds the value 30, which is less than key. Because i is less than or equal to j, 90

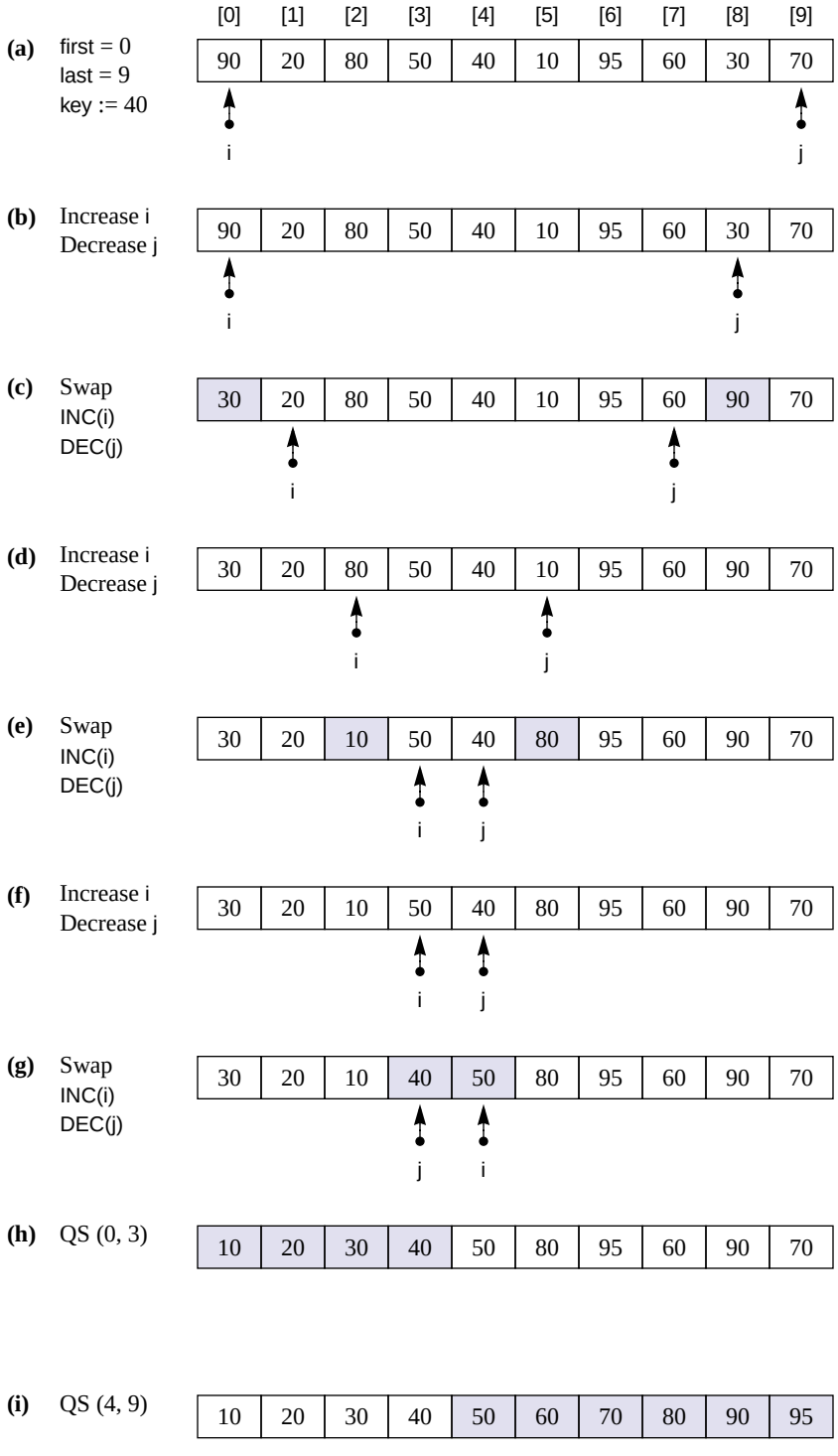


Figure 20.10
A trace of the first call to QuickSort in Figure 20.9.

is to the left of 30. So they need to be exchanged. Figure 20.10(c) shows the result of the exchange. Afterward, QuickSort increments i by 1 and decrements j by 1.

Because i is still to the left of j, the loop repeats. Figure 20.10(d) shows i increasing to find 80 and j decreasing to find 10. j skips over 60 and 95 because they are greater than key. i is still to the left of j. Figure 20.10(e) shows the exchange of 10 and 80, the increment of i, and the decrement of j.

The algorithm has four loop invariants as shown in the comments. They are described more formally in the next section. The net effect of the four invariants is to establish the single invariant:

- Every element between v[first] and v[i - 1] is less than or equal to every element between v[j + 1] and v[last].

In Figure 20.10(e), the loop invariant means that each of the values (30, 20, 10) is less than or equal to each of the values (80, 95, 60, 90, 70).

The initializing statements make the loop invariant true the first time. Because they initialize i to first, there are no elements between v[first] and v[first - 1]. Because they initialize j to last, there are no elements between v[last + 1] and v[last]. Because there are no elements in the left interval and no elements in the right interval, every element in the left interval is less than or equal to every element in the right interval.

The statements in the body of the WHILE loop keep the invariant true. They increase i and/or decrease j, in effect widening the left and right intervals. When i finds a value greater than or equal to key and j finds a value less than or equal to key, you know that i's value is greater than or equal to j's value. The exchange keeps the invariant true.

Figure 20.10(g) shows the last exchange. QuickSort swaps 50 and 40. After it increments i and decrements j, i has the value 5 and j has the value 4. So j is to the left of i, and the loop terminates.

The assertion in the listing follows from the loop invariant and the termination condition. $L1$ is the sublist between v[first] and v[j]. $L2$ is the sublist between v[i] and v[last].

Figure 20.10(h) and (i) shows the result of the recursive calls to QuickSort. The abbreviation QS (0, 3) stands for the procedure call

```
QuickSort (v, first, i - 1)
```

when first has the value 0 and i has the value 4. Similarly, QS (4, 9) stands for the procedure call

```
QuickSort (v, i, last)
```

when i has the value 4 and last has the value 9.

Each recursive call to QuickSort splits a smaller list. The first recursive call splits the list

30 20 10 40

with a first of 0 and a last of 3. The second recursive call splits the list

```
50  80  95  60  90  70
```

with a first of 4 and a last of 9. Each of these executions produces a trace like that of Figure 20.10.

What is the structure of the call tree of QuickSort? The listing shows that the procedure makes either two recursive calls or no recursive calls, depending on the size of *L*. Therefore, each node in the call tree will have either two children or no children. For the values listed in Figure 20.8, you would need to do a trace of the split at each call. If you do the traces, you will see that the call tree is structured as shown in Figure 20.11.

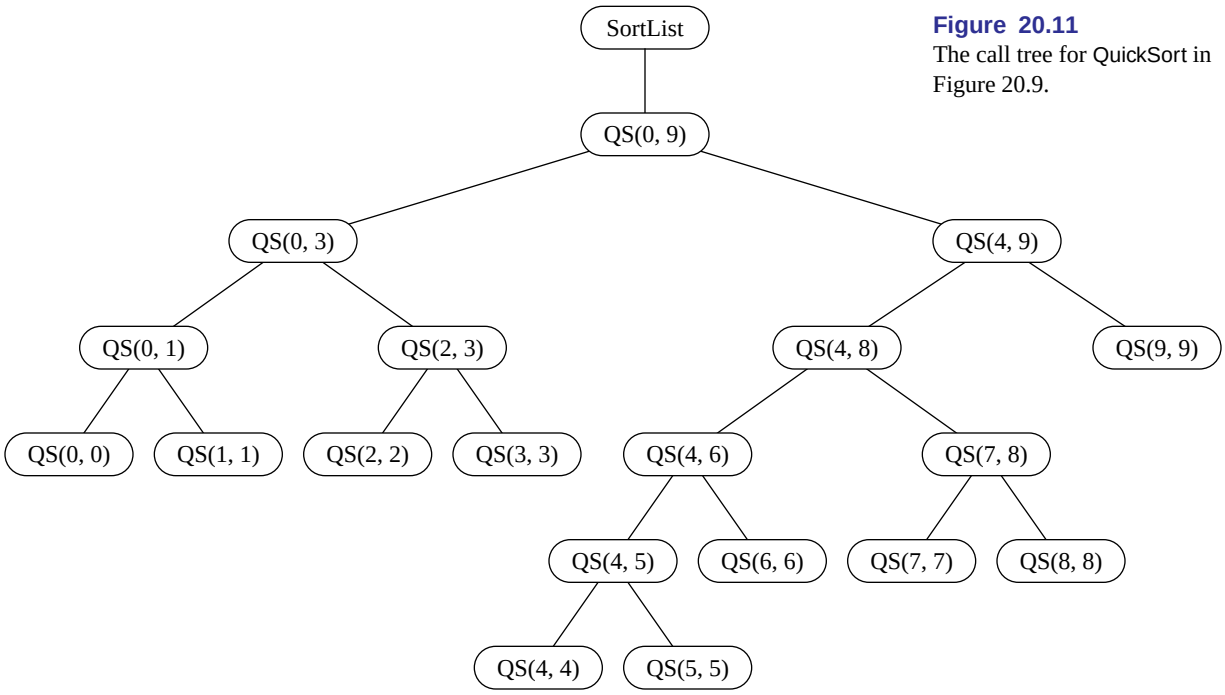


Figure 20.11
The call tree for QuickSort in Figure 20.9.

The program makes a total of 19 calls to QuickSort, including the initial call from SortList. The figure shows that QS (4, 9) splits the list of six elements into *L*1, with five elements, and *L*2, with one element. It does not call itself recursively for *L*2, but it does for *L*1. QS (4, 8) makes a more even split. Its list *L* has five items, from v[4] to v[8]. It splits it into sublist *L*1, from v[4] to v[6], and *L*2, from v[7] to v[8].

Using the technique of Figure 19.12, you can determine from the call tree that the order of calls and returns is as follows:

```
SortList
   Call QS (0, 9)
      Call QS (0, 3)
         Call QS (0, 1)
            Call QS (0, 0)
         Return to QS (0, 1)
            Call QS (1, 1)
         Return to QS (0, 1)
      Return to QS (0, 3)
         Call QS (2, 3)
            Call QS (2, 2)
         Return to QS (2, 3)
            Call QS (3, 3)
         Return to QS (2, 3)
      Return to QS (0, 3)
   Return to QS (0, 9)
      Call QS (4, 9)
         Call QS (4, 8)
            Call QS (4, 6)
               Call QS (4, 5)
                  Call QS (4, 4)
               Return to QS (4, 5)
                  Call QS (5, 5)
               Return to QS (4, 5)
            Return to QS (4, 6)
               Call QS (6, 6)
            Return to QS (4, 6)
         Return to QS (4, 8)
      Return to QS (4, 9)
         Call QS (9, 9)
      Return to QS (4, 9)
   Return to QS (0, 9)
Return to SortList
```

You can also see from Figure 20.11 that the maximum number of stack frames on the run-time stack is seven, including the stack frame for procedure SortList. The maximum occurs twice, once after the call to QS(4, 4) and then again after the call to QS(5, 5).

★ Correctness of quick sort

The quick sort procedure with the outer loop invariant is written in GCL as follows.

```
procedure QuickSort(v,first,last);
   if first < last →
      key := v[(first + last) div 2]; i := first; j := last;
      {(∀k | j + 1 ≤ k ≤ last : key ≤ v[k]) ∧
      (∀k | first ≤ k ≤ i – 1 : v[k] ≤ key) ∧
      (i ≤ j ⇒ (∃k | first ≤ k ≤ j : v[k] ≤ key)) ∧
      (i ≤ j ⇒ (∃k | i ≤ k ≤ last : key ≤ v[k]))}
      do i ≤ j →
         do v[i] < key → i := i + 1 od;
         do key < v[j] → j := j – 1 od;
         if i ≤ j → v[i], v[j] := v[j], v[i]; i, j := i + 1, j – 1
          [] i > j → skip
         fi
      od;
      QuickSort(v,first,i – 1);
      QuickSort(v,i,last)
   [] first ≥ last → skip
   fi
end QuickSort
```

To prove the correctness of the quick sort algorithm requires several steps. This section outlines the steps and leaves the details of the proof to the exercises.

Step 1: The quick sort algorithm is recursive. Therefore, the proof of its correctness is a proof by mathematical induction. The first step in a proof by mathematical induction is to prove the base case. The base case occurs when the segment of *v* to be sorted is empty or when it has one element. The complete formal specification for *QuickSort* including the precondition from the ASSERT statement in Figure 20.9 is

```
{(0 ≤ first ≤ last < len(v) ∨ last < 0 ≤ first) ∧ v = V}
v := ?
{perm(v,V,first,last) ∧ (∀i | first ≤ i < last – 1 : v[i] ≤ v[i + 1])}
```

The first disjunct in the precondition is for the case when the segment has one or more elements. The second disjunct is for the case when the original array to be sorted has no elements. Use the precondition with the base case to prove the postcondition.

Step 2: The second step in a proof by mathematical induction is to show that the correctness for a small number of elements implies the correctness for a larger number of elements. With *QuickSort*, you assume that the recursive call

```
QuickSort(v,first,i – 1)
```

will correctly sort *v* between *first* and *i* – 1, and that the recursive call

```
QuickSort(v,i,last)
```

will correctly sort *v* between *i* and *last*. For the recursion to terminate, these calls must be for segments that are smaller than the original segment [*first*..*last*]. If

$i = first$ then the second recursive call will be for a segment that is the same length as the original segment, and if $i = last + 1$ then the first recursive call will be for a segment that is the same length as the original segment. The second step is to prove that neither of these cases can happen, that is, that $first < i \leq last$, so the recursion will terminate.

Step 3: The idea behind the quick sort algorithm is to partition the segment $[first..last]$ into two subsegments $[first..i-1]$ and $[i..last]$ such that every element in $[first..i-1]$ is less than or equal to every element in $[i..last]$. Assuming that the recursive calls correctly sort the subsegments the entire segment will be sorted. The third step is to prove that every element in the first subsegment is less than or equal to every element in the second subsegment. That is, you must prove that

$$(\forall k \mid first \leq k < i : (\forall l \mid i \leq l < last : v[k] \leq v[l]))$$

before the recursive calls are made. The recursive calls are made just after the outer **do** loop. So, for this part of the proof you may assume the four loop invariants and the negation of the loop condition.

Step 4: The previous step assumed the loop invariants. The fourth step is to prove the loop invariants from the precondition and the initialization statements just before the outer **do** loop.

Step 5: The fifth step is to prove that one execution of the outer **do** loop maintains the loop invariants.

Step 6: The sixth step is to prove that the outer **do** loop terminates. You can prove that it terminates with the help of the loop invariants.

Merge sort

Unlike the quick sort algorithm, merge sort splits L exactly in half every time. Figure 20.12 shows the split part of the merge sort algorithm. The parameters are the same as those for procedure QuickSort and the procedure is called the same way. The merge part of the algorithm is left as a problem for the student.

```
PROCEDURE MergeSort (VAR v: ARRAY OF INTEGER; first, last: INTEGER);
   (* Sorts the items of array v between v[first] and v[last]. *)
   VAR
      i, j, k, mid: INTEGER;
      temp: ARRAY 128 OF INTEGER;
BEGIN
   ASSERT((0 <= first) & (first <= last) & (last < LEN(v)) OR (last < 0) & (0 <= first), 20);
   IF first < last THEN
      mid := (first + last) DIV 2;
      MergeSort(v, first, mid);
      MergeSort(v, mid + 1, last);
      (* Problem for the student to join v[first..mid] and v[mid + 1..last] *)
   END
END MergeSort;
```

Figure 20.12
An implementation of the merge sort algorithm.

Merge sort has a problem that quick sort does not have. To merge two parts of one list into a second list requires storage for the second list. To perform the merge you must allocate storage for the second list as a local array variable, which is the purpose of temp in Figure 20.12. You then merge the two sublists into the second list and copy the second list back into the original.

Figure 20.13 shows a trace of the top-level call to MergeSort of Figure 20.12. The split is the simple computation of local variable mid. The recursive calls are at the beginning of the algorithm in contrast to the recursive calls of QuickSort, which are at the end.

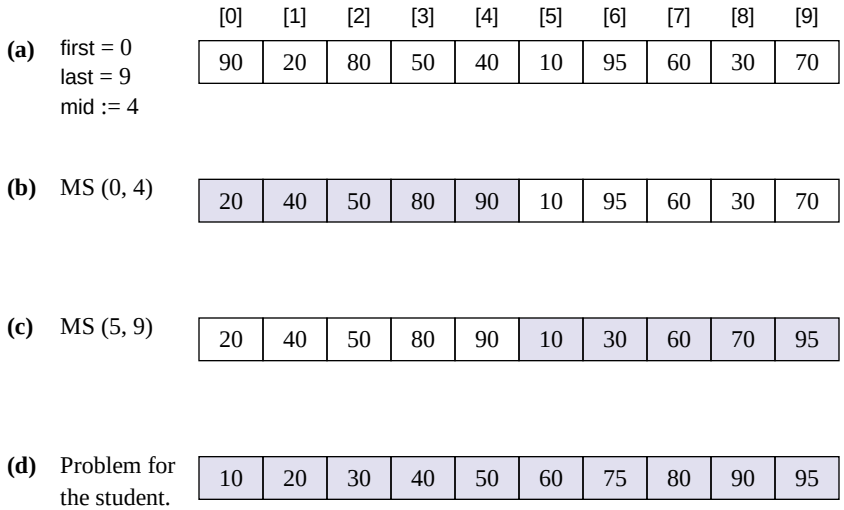


Figure 20.13
A trace of the top-level call to MergeSort in Figure 20.12.

Figure 20.14 shows a trace of the join operation that is left as a problem for the student. Each part of the figure shows the v array located above the temp array. The idea is to have one FOR loop with control variable k that increments from first to last and denotes the index of temp that receives a value from v. At each iteration of the loop, temp[k] will get either v[i], after which i is incremented, or v[j], after which j is incremented. The figure does not show seven steps in the loop that occur between parts (c) and (d). At the conclusion of the merge from array v to array temp, the elements from temp[first] to temp[last] must be copied back into array v from v[first] to v[last].

Figure 20.15 is the call tree for the values in Figure 20.13(a). The values in parentheses are the values of parameters first and last. The leaf nodes have first equal to last, indicating that a subarray of one element needs to be sorted. The algorithm does no processing in those cases, but just returns to the calling procedure.

You should be able to visualize the order in which the merges are performed from Figure 20.15. Figure 20.16 show the merges in the order they occur for the left half of the array.

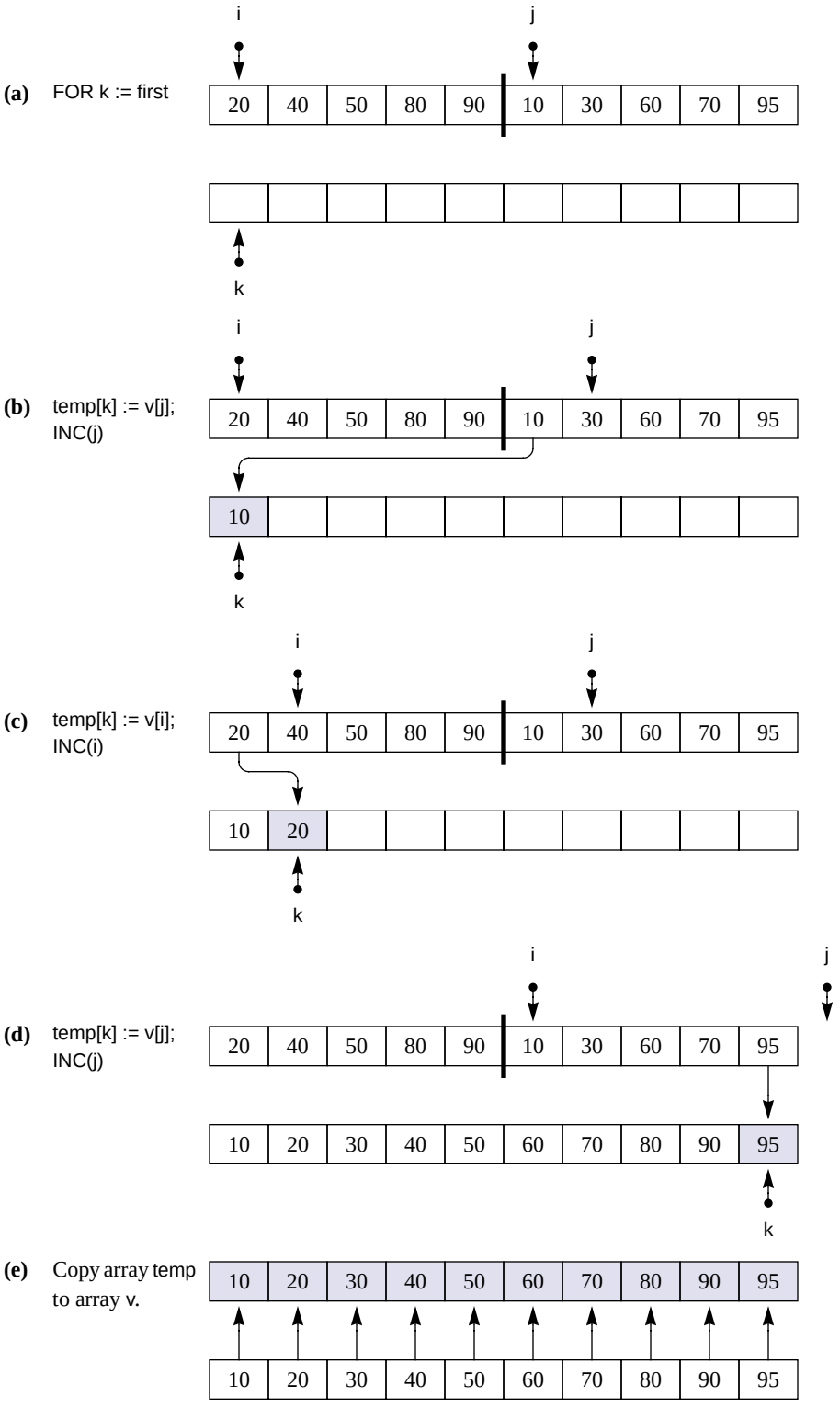


Figure 20.14
A trace of the top-level call to MergeSort in Figure 20.12. Seven steps are not shown between parts (c) and (d).

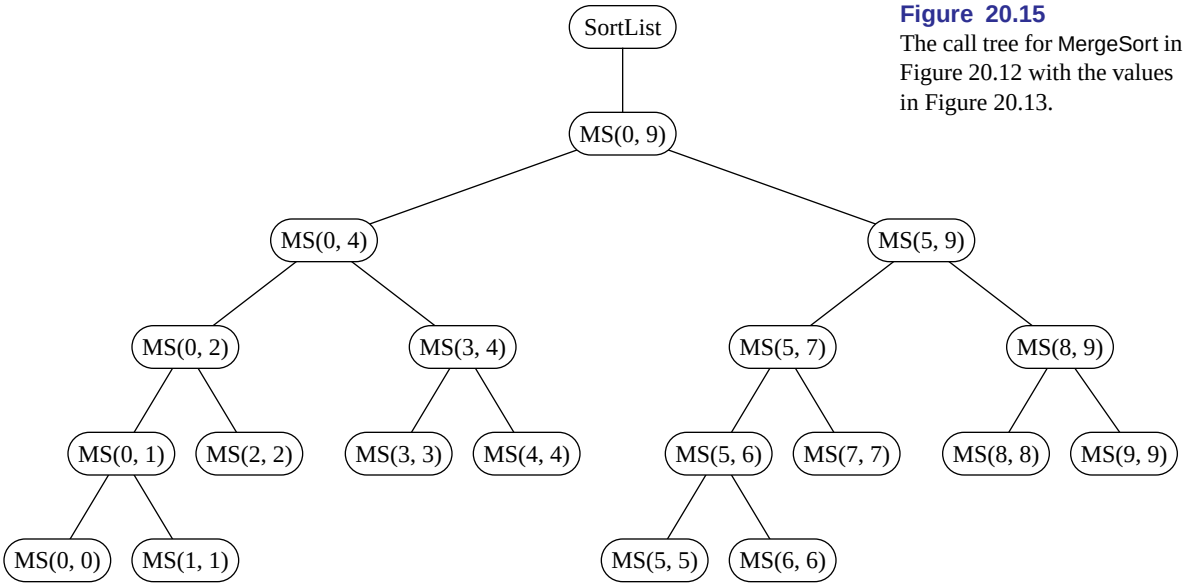


Figure 20.15
The call tree for MergeSort in Figure 20.12 with the values in Figure 20.13.

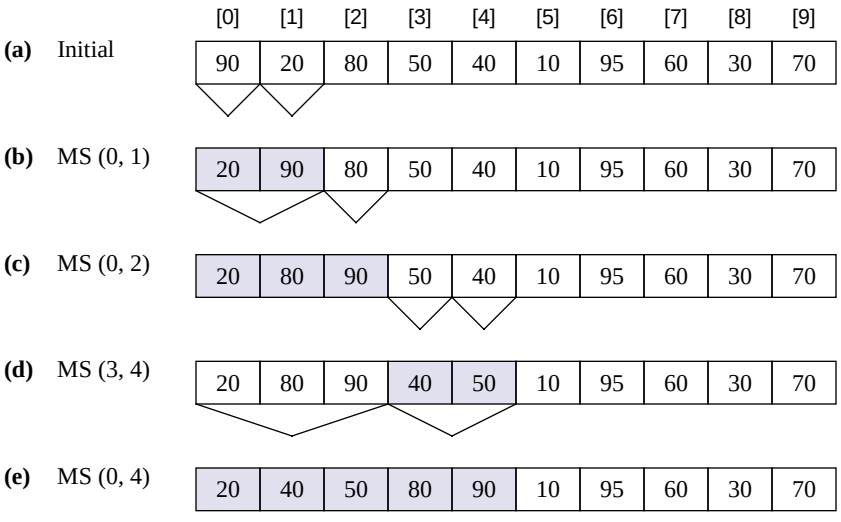


Figure 20.16
The merges in MergeSort in the order they occur for the left half of the array.

In-place merge sort

The program in Figure 20.17 is an implementation of the in-place merge sort. It is a more efficient implementation of the algorithm because it does not require extra storage for the second array or extra time for the copy operation in Figure 20.14(e). The program sorts an array of integers. It stores each value in a record with two parts, value and link. The array to be sorted is an array of records. The algorithm does not exchange any records in the array. Instead, it alters the link part of all the records in such a way that you can always determine the next higher number from the link field.

The input and output are identical to that for the quick sort in Figure 20.8. One difference from the previous sort algorithms is that the in-place merge sort cannot work with an empty array. Consequently, its precondition is weaker, and procedure SortList must test for the case of an empty array.

```
MODULE Pbox20C;
   IMPORT Dialog, TextModels, TextViews, Views, TextControllers, PboxMappers;
   TYPE
      Item = RECORD
         value: INTEGER;
         link: INTEGER
      END;
   VAR
      d*: RECORD
         numItems-: INTEGER;
      END;
      list: ARRAY 1024 OF Item;

   PROCEDURE LoadList*;
      VAR
         md: TextModels.Model;
         cn: TextControllers.Controller;
         sc: PboxMappers.Scanner;
         i: INTEGER;
   BEGIN
      cn := TextControllers.Focus();
      IF cn # NIL THEN
         md := cn.text;
         sc.ConnectTo(md);
         i := 0;
         sc.ScanInt(list[i].value); list[i].link := -1;
         WHILE ~sc.eot DO
            INC(i);
            sc.ScanInt(list[i].value); list[i].link := -1
         END;
         d.numItems := i;
         Dialog.Update(d)
      END;
   END LoadList;
```

Figure 20.17
An implementation of the in-place merge sort algorithm.

```
PROCEDURE MergeSort (VAR v: ARRAY OF Item; first, last: INTEGER; OUT start: INTEGER);
    (* Sorts the items of array v between v[first] and v[last]. *)
    VAR
        mid, loStart, hiStart: INTEGER;
        i, j, k: INTEGER;
BEGIN
    ASSERT((0 <= first) & (first <= last) & (last < LEN(v) - 1), 20);
    IF first = last THEN
        start := first
    ELSE
        mid := (first + last) DIV 2;
        MergeSort(v, first, mid, loStart);
        MergeSort(v, mid + 1, last, hiStart);
        i := loStart;
        j := hiStart;
        k := LEN(v) - 1; (* Temporary start of merged list *)
        WHILE (i # -1) & (j # -1) DO
            IF v[i].value <= v[j].value THEN
                v[k].link := i;
                k := i;
                i := v[i].link
            ELSE
                v[k].link := j;
                k := j;
                j := v[j].link
            END
        END;
        IF i = -1 THEN
            v[k].link := j  (* Attach remainder of last list *)
        ELSE
            v[k].link := i  (* Attach remainder of first list *)
        END;
        start := v[LEN(v) - 1].link
    END
END MergeSort;
```

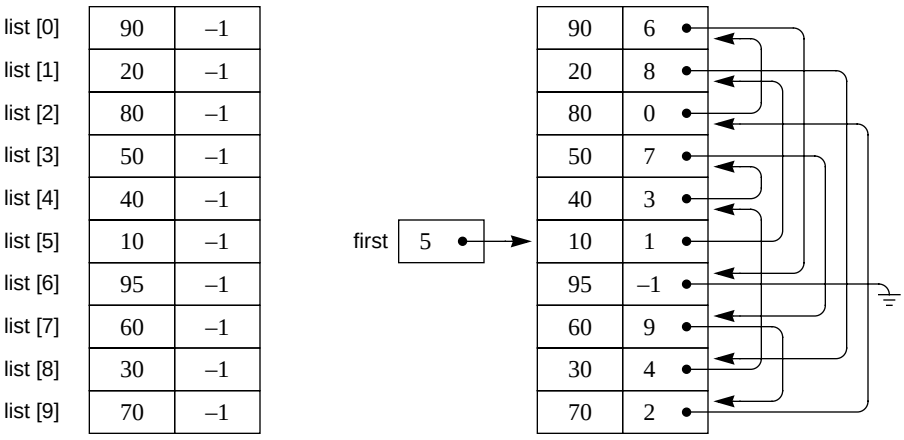
Figure 20.17
Continued.

```
    PROCEDURE SortList*;
        VAR
            md: TextModels.Model;
            vw: TextViews.View;
            fm: PboxMappers.Formatter;
            i: INTEGER;
            first: INTEGER;
    BEGIN
        md := TextModels.dir.New();
        fm.ConnectTo(md);
        FOR i := 0 TO d.numItems -1 DO
            fm.WriteInt(list[i].value, 4)
        END;
        fm.WriteLn;
        IF d.numItems > 0 THEN
            MergeSort(list, 0, d.numItems - 1, first);
            i := first;
            WHILE i # -1 DO
                fm.WriteInt(list[i].value, 4);
                i := list[i].link
            END
        END;
        vw := TextViews.dir.New(md);
        Views.OpenView(vw)
    END SortList;

BEGIN
    d.numItems := 0
END Pbox20C.
```

Figure 20.17
Continued.

Figure 20.18(a) shows the array of records before SortList calls procedure MergeSort. Procedure LoadList sets the link field of every record to –1 before the call.



(a) Before the first MergeSort call. **(b)** After the top-level merge

Figure 20.18
The result of a MergeSort call from procedure SortList of Figure 20.17.

Figure 20.18(b) shows the array of records after the call to MergeSort. Procedure SortList has an integer variable, first. MergeSort sets first to 5 because list[5].value is the smallest item in the list. It sets list[5].link to 1 because list[1].value is the next larger item in the list. It sets list[1].link to 8 because list[1].value is the next larger item in the list, and so on.

For each record, i, list[i].link is the index of the record whose value part is the next larger item in the list. The second field links each item to the next larger item. The record with the largest value, record 6 in this list, has a Link of –1. That link points to nothing at all, which the figure indicates by the dashed triangle.

The WHILE loop in SortList outputs the list in order. It initializes i to 5 and outputs list[5].value. The assignment

```
i := list[i].link
```

gives i the value 1. The next time through the loop the WriteInt procedure outputs list[1].value. The loop continues to advance i through the linked list until it gets the value –1, when the loop terminates. Even though the program exchanged no values, the output is indistinguishable from the QuickSort program. In effect, the program sorted the list.

Procedure SortList calls MergeSort with a value of 0 for first and 9 for last. MergeSort splits the list in half with

```
mid := (first + last) DIV 2
```

which gives 4 to mid. It calls itself recursively to sort $L1$ as the list between v[0] and v[4], and $L2$ as the list between v[5] and v[9]. Figure 20.19 shows the list after these two recursive calls to MergeSort.

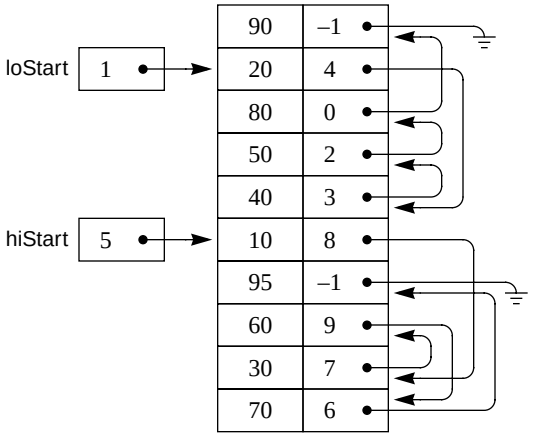
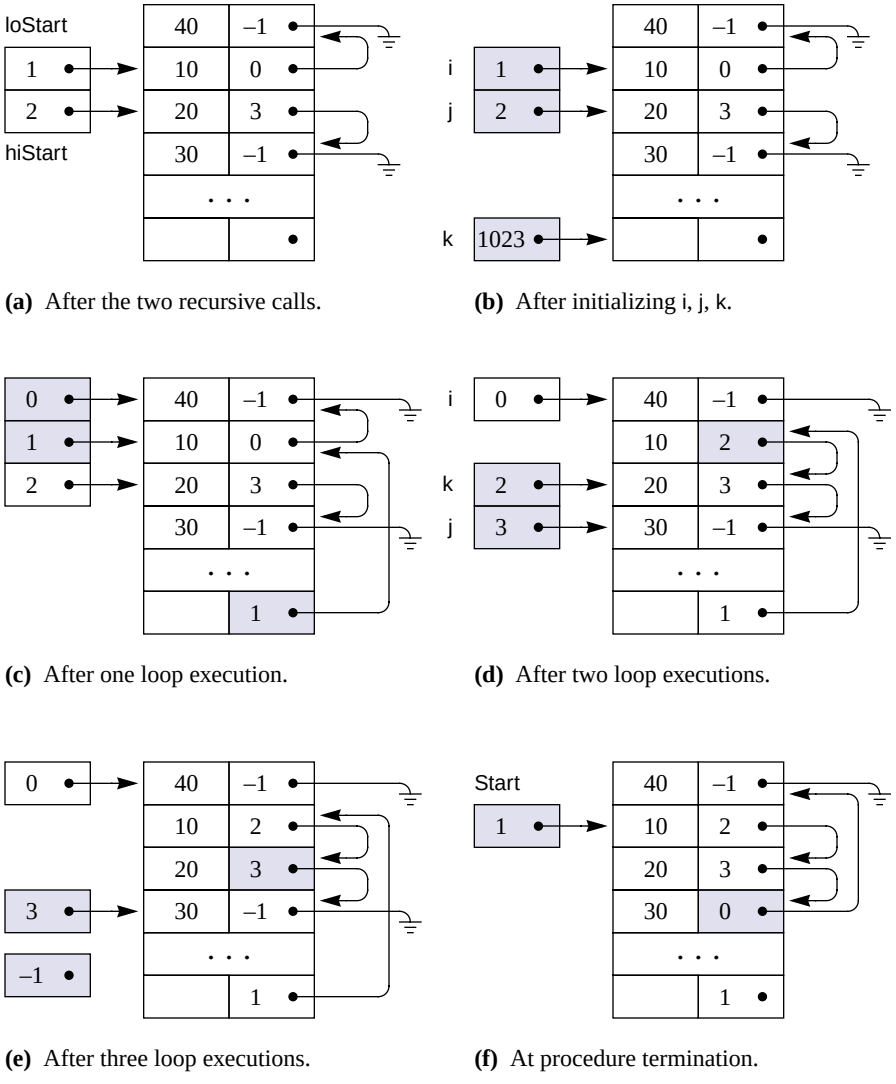


Figure 20.19
The list in the call to MergeSort(v, 0, 9, first) after the recursive calls to MergeSort(v, 0, 4, loStart) and MergeSort(v, 5, 9, hiStart).

The split was easy. The rest of MergeSort, that part in the WHILE loop, is the join. Given the values for loStart and hiStart, which point to the start of two ordered linked lists, the problems is to alter their link fields to make one ordered linked list with start pointing to the smallest element. Figure 20.20 shows a trace of the join operation for two short linked lists.



(a) After the two recursive calls.

(b) After initializing i, j, k.

(c) After one loop execution.

(d) After two loop executions.

(e) After three loop executions.

(f) At procedure termination.

Figure 20.20
A trace of the join operation in MergeSort for two short linked lists.

Figure 20.20(b) shows i initialized to loStart and j initialized to hiStart. MergeSort initializes k to LEN(v) - 1. It assumes that the list does not use the entire array and that the last record is available for temporary storage. i advances through the first list, j advances through the second list, and k advances through the merged list.

Each time the loop executes, it finds the next item from lists *L*1 and *L*2 to put in

the merged list. It changes the link in the last record of the merged list to point to the newly merged item from *L*1 or *L*2. The newly merged item is taken off the sublist. At the conclusion of the loop, all the items will be in one merged list with no physical exchanges.

Figure 20.20(c) shows the operation after one loop execution. The statements

```
v[k].link := i;
k := i;
i := v[i].link
```

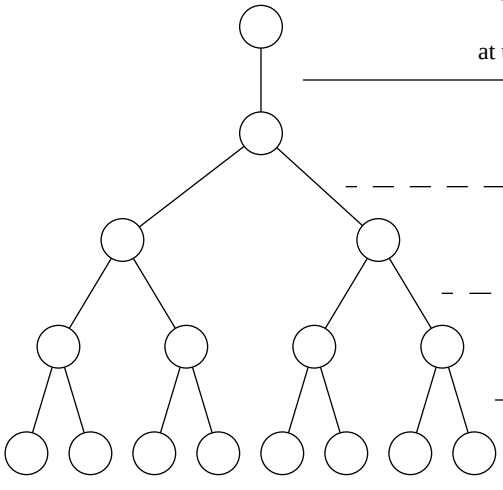
link the v[1] record into the new merged list and unlink it from the i list. Figure 20.20(d) and (e) show the same operation with both values from the j list.

When the WHILE loop gets to the end of one of the lists, you know that all the remaining links of the other sublist do not need changing. The last IF statement links the tail of the other sublist to the end of the merged list, as Figure 20.20(f) shows.

Complexity of the sort algorithms

How fast are the sort algorithms that are described in this section? Remember from Chapter 19 that the selection sort is $O(n^2)$. The insertion sort is also $O(n^2)$. You can visualize in Figure 20.6 that each algorithm requires n passes through the list. Each pass requires a loop to do the insertion or selection. The doubly nested loops give the algorithms their $O(n^2)$ behavior.

How do quick sort and merge sort compare with the single-element sorts? In the best case with quick sort, you divide the list in half each time. Figure 20.21 shows the call tree for merge sort and the best case quick sort for a 16-element list.



Number of calls at this level	Number of executions of each loop	Number of comparisons
1	16	1 × 16 = 16
2	8	2 × 8 = 16
4	4	4 × 4 = 16
8	2	8 × 2 = 16
Total number of comparisons:		4 × 16 = 64

Figure 20.21
The call tree for merge sort and the best case quick sort with a 16-element list.

The list at the top level has 16 elements. The lists at the next lower recursive call have 8 elements. The lists at the next lower call have 4 elements, and the bottom level has 2-element lists.

A loop executes at each level. In merge sort, the WHILE loop performs the join. In quick sort, the REPEAT loop performs the split. In both algorithms, the loop passes through the list comparing items. The number of comparisons equals the number of items in the list.

For example, the top level has a list of 16 items, and the algorithm makes 16 comparisons. The next level has lists with 8 items. The loops at this level make 8 comparisons each. Because there are 2 recursive calls at the second level, the total number of comparisons at this level is also 16. Similarly, the number of comparisons at the next lower level is also 16. There are 4 lists, and each list requires 4 comparisons. In general, if the list has n elements, the algorithms make n comparisons at each level. So the total number of comparisons is n times the total number of levels.

How many levels are there for a list of n elements? The number of times you need to divide n in half to get it down to 1. You recognize this answer from the analysis of the binary search algorithm. It is $\lg n$. In Figure 20.21, the logarithm of 16 is 4, which corresponds to the 4 levels of recursive calls. The total number of comparisons is therefore 4 times 16, which is 64.

In general, the total number of comparisons is n times $\lg n$. Merge sort and the best-case quick sort are $O(n \lg n)$ algorithms. In the worst case, quick sort is $O(n^2)$ because in that case it is equivalent to the selection sort. In practice, quick sort is $O(n \lg n)$ on the average.

The five orders encountered thus far, starting with the fastest, are

- $O(\lg n)$ Example: the binary search
- $O(n)$ Example: the sequential search
- $O(n \lg n)$ Example: the merge sort and quick sort
- $O(n^2)$ Example: the single-element sorts
- $O(n^3)$ Example: matrix multiplication

If you are comparing two algorithms with different orders, the algorithm with an order farther down the list will be worse for large amounts of data, regardless of the coefficients. For example, an algorithm with a statement execution count of $5n \lg n$ will be faster than one with $2n^2$. Even though 5 is greater than 2, for large n the first expression will be smaller than the second. On the other hand, the coefficients are important when you compare two algorithms with the same order. If one algorithm has a statement execution count of $5n \lg n$, and the second has a count of $2n \lg n$, the second algorithm will be faster.

Some algorithms, not encountered in this book, are even worse than $O(n^3)$. They are $O(2^n)$, and form a class of difficult problems that computer scientists have spent a great deal of time investigating. They are interesting problems that you will learn about if you take more advanced computer science courses.

Exercises

1. Draw the ideal quick sort and merge sort traces corresponding to Figure 20.4(a) and (b) for the following lists:

(a) 4 7 5 2 3 8 1 6
(b) 8 7 6 5 4 3 2 1
(c) 8 1 2 3 4 5 6 7

2. Work Exercise 1 for the single-element sorts of Figure 20.5(a) and (b).

3. Work Exercise 1 for the nonrecursive single-element sorts of Figure 20.6(a) and (b).

4. Write the list of 10 integer values just after the following calls to QuickSort in Figure 20.11.

(a) QS (0, 3) **(b)** QS (2, 3) **(c)** QS (4, 9)
(d) QS (4, 8) **(e)** QS (4, 5) **(f)** QS (7, 8)

5. Draw the list and the elements that j and i point to corresponding to Figure 20.10(g), (h), and (i) for the initial lists that follow. Figure 20.10(g) represents the list just before the first recursive call to QuickSort.

(a) 10 60 40 80 30 90 20 70 25 50
(b) 10 20 25 30 40 50 60 70 80 90
(c) 90 80 70 60 50 40 30 25 20 10
(d) 80 90 50 50 50 50 50 50 10 20

6. Draw the QuickSort call tree as in Figure 20.11 for the following initial lists.

(a) 30 70 40 20 **(b)** 80 40 20 90 70
(c) 40 60 10 70 90 30 **(d)** 10 20 30 40 50 60 70
(e) 70 60 50 40 30 20 10

How many times is QuickSort called? What is the maximum number of QuickSort stack frames on the run-time stack during the execution? In what order does the program make the calls and returns?

7. The section on the correctness of the quick sort algorithm, page 446, gave an eight-step outline of the proof. **(a)** Prove step 1. **(b)** Prove step 2. **(c)** Prove step 3. **(d)** Prove step 4. **(e)** Prove step 5. **(f)** Prove step 6. **(g)** Prove step 7. **(h)** Prove step 8.

8. Draw the result of a MergeSort call corresponding to Figure 20.18 for the following lists of numbers.

(a) 30 50 10 80 40 70 20 60
(b) 10 20 30 40 50 60 70 80
(c) 80 70 60 50 40 30 20 10

9. Work Exercise 8 for the two top-level recursive MergeSort calls corresponding to Figure 20.19.

10. What is the total number of comparisons for the merge sort and the best-case quick sort with the following number of elements?

(a) 32 **(b)** 1024 **(c)** 65,536

What is the maximum number of stack frames allocated at one time for each of the lists?

Problems

11. Write the recursive version of the sequential search. Use the technique of inserting the number to be searched at the end of the list as shown in Figure 16.3 and Figure 16.4. Declare

```
PROCEDURE RecursiveSearch (VAR v: ARRAY OF INTEGER; numItm, srchNum: INTEGER;
        VAR i: INTEGER; OUT fnd: BOOLEAN);
BEGIN
    (* Problem for the student *)
END RecursiveSearch;

PROCEDURE Search (VAR v: ARRAY OF INTEGER; numItm, srchNum: INTEGER;
        OUT i: INTEGER; OUT fnd: BOOLEAN);
BEGIN
    ASSERT((0 <= numItm) & (numItm < LEN(v)), 20);
    i := 0;
    v[numItm] := srchNum;
    RecursiveSearch(v, numItm, srchNum, i, fnd)
END Search;
```

Note that i is called by reference in RecursiveSearch. Do not use a loop. Do not compare i with numItm. Test your program with a dialog box identical to that in Figure 20.1.

12. Write the recursive version of the insertion sort algorithm as shown in Figure 20.5(a). Make a recursive call to sort the left part of the list, and use a single loop, starting from the end of the sorted part of the list to insert the single element. Test your algorithm in a program similar to the one in Figure 20.9.

13. Write the recursive version of the selection sort algorithm as shown in Figure 20.5(b). Use a single loop to move the largest element to the end of the list, then make a recursive call to sort the remaining left part of the list. Test your algorithm in a program similar to the one in Figure 20.9.

14. Write the nonrecursive version of the insertion sort algorithm as shown in Figure 20.5(a). Use a nested loop. Test your algorithm in a program similar to the one in Figure 20.9.

15. Suppose an application uses the merge sort, but it needs to have the array elements physically in order, not just linked in order. Modify procedure SortList of Figure 20.17 to put the elements of list physically in order. Declare tempList to be an array of the same type as list. After the call to MergeSort, copy the elements from list into tempList in physical order. Then copy the elements from tempList to list and output list without using the link field to verify that its elements are physically in order.

16. Complete procedure MergeSort of Figure 20.12. Test it in a program identical to Figure 20.9 except that the call is to MergeSort instead of to QuickSort.