11 Advanced Sorting Concepts

One of the most common applications in computer science is **sorting,** the process through which data are arranged according to their values. We are surrounded by data. If the data were not ordered, we would spend hours trying to find a single piece of information. Imagine the difficulty of finding someone's telephone number in a telephone book that had no internal order.

The history of sorting dates back to the roots of computing, Herman Hollerith's electric tabulating machine, which was used to tally the 1890 U.S. Census and was one of the first modern sorting machines. Sorting was also on the scene when general-purpose computers first came into use. According to Knuth, "There is evidence that a sorting routine was the first program ever written for a stored program computer."[1] Although computer scientists have not developed a major new algorithm in more than 30 years (the newest algorithm in this book is heap sort, which was developed in 1964), sorting is still one of the most important concepts in computing today.

1. Donald E. Knuth, *The Art of Computer Programming*, vol. 3, *Sorting and Searching* (Reading, MA: Addison-Wesley, 1973), 384.

11-1 GENERAL SORT CONCEPTS

We discuss six internal sorts in this chapter: insertion sort, bubble sort, selection sort, shell sort, heap sort, and quick sort. The first three are useful only for sorting very small lists, but they form the basis of the last three, which are all useful general-purpose sorting concepts. After discussing the internal sorts, we will introduce the basic concepts used in external sorts.

Note

> Sorting is one of the most common data-processing applications.

Sorts are generally classified as either internal or external sorts. An **internal sort** is a sort in which all of the data are held in primary memory during the sorting process. An **external sort** uses primary memory for the data currently being sorted and secondary storage for any data that will not fit in primary memory. For example, a file of 20,000 records may be sorted using an array that holds only 1000 records. During the sorting process, only 1000 records are therefore in memory at any one time; the other 19,000 are kept in a file in secondary storage.

Note

> Sorting algorithms are classified as either internal or external.

Internal sorting algorithms have been grouped into several different classifications depending on their general approach to sorting. Knuth identified five different classifications[2]: insertion, selection, exchanging, merging, and distribution sorts. In this text we cover the first three. Distribution sorts, although interesting, have minimal use in computers. The different sorts are shown in Figure 11-1.

Sort Order

Data may be sorted in either ascending sequence or descending sequence. The **sort order** identifies the sequence of the sorted data, ascending or descending. If the order of the sort is not specified, it is assumed to be ascending. Examples of common data sorted in ascending sequence are the dictionary and the telephone book. Examples of descending data are percentages of games won in a sporting event such as baseball or grade point averages for honor students.

---

2. See Donald E. Knuth, *The Art of Computer Programming*, vol. 3, *Sorting and Searching* (Reading, MA: Addison-Wesley, 1973), 73–180.

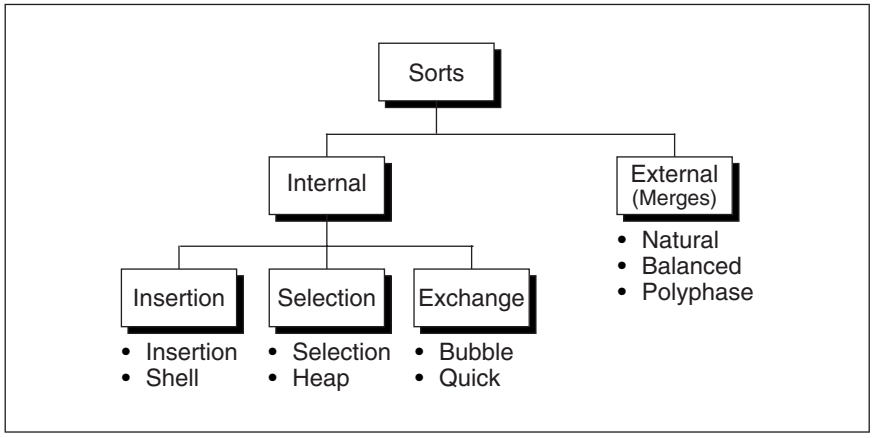

Figure 11-1 Sort classifications

Sort Stability

Sort stability is an attribute of a sort indicating that data with equal keys maintain their relative input order in the output. Stability is best seen in an example. In Figure 11-2(a) the unsorted data contain three entries with identical keys (212). If the data are sorted with a stable sort, the order in Figure 11-2(b) is guaranteed. That is, 212 *green* is guaranteed to be the first of the three in the output, 212 *yellow* is guaranteed to be the second, and 212 *blue* is guaranteed to be the last. If the sort is not stable, records with identical keys may occur in any order, including the stable order shown in Figure 11-2(b). Figure 11-2(c) is one example of the six different sequences that could occur in an unstable sort. Note that in this example, blue comes out first even though it was the last of the equal keys. Of the sort algorithms we will discuss in this text, the `insertionSort` (page 507), `selectionSort` (page 519), and `bubbleSort` (page 528) are stable; the others are unstable.

Sort Efficiency

Sort efficiency is a measure of the relative efficiency of a sort. It is usually an estimate of the number of comparisons and moves required to order an unordered list. We discuss the sort efficiency of each of the internal sorts we cover in this chapter. Generally speaking, however, the best possible sorting algorithms are on the order of $n\log_2 n$; that is, they are $O(n\log_2 n)$ sorts.[3] Three of the sorts we study are $O(n^2)$. The best, quick sort, is $O(n\log_2 n)$.

3. As a project, we discuss an interesting sort, radix sort, which is $O(n)$. However, its extensive overhead limits its general use.

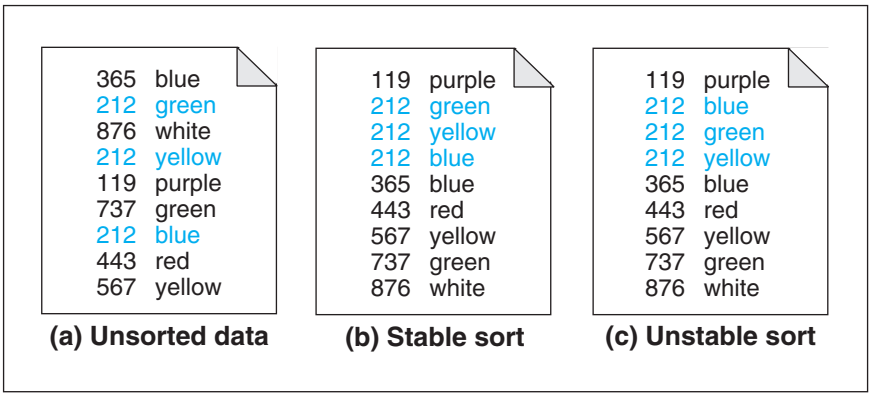

Figure 11-2 Sort stability

Passes

During the sorting process, the data are traversed many times. Each traversal of the data is referred to as a **sort pass.** Depending on the algorithm, the sort pass may traverse the whole list or just a section of the list. Also, characteristic of a sort pass is the placement of one or more elements in a sorted list.

11-2 INSERTION SORTS

Insertion sorting is one of the most common sorting techniques used by card players. As they pick up each card, they insert it into the proper sequence in their hand. (As an aside, card sorting is an example of a sort that uses two pieces of data to sort: suit and rank.) The concept extends well into computer sorting. In each pass of an insertion sort, one or more pieces of data are inserted into their correct location in an ordered list. In this section we study two insertion sorts, the straight insertion sort and the shell sort.

Straight Insertion Sort

In the **straight insertion sort,** the list is divided into two parts: sorted and unsorted. In each pass, the first element of the unsorted sublist is transferred to the sorted sublist by inserting it at the appropriate place. If we have a list of n elements, it will take at most $n - 1$ passes to sort the data. This concept is shown in Figure 11-3. In this figure, we have placed a visual wall between the sorted and unsorted portions of the list.

Straight Insertion Sort Example

Figure 11-4 traces the insertion sort through a list of six numbers. Sorting these data requires five sort passes. Each pass moves the wall

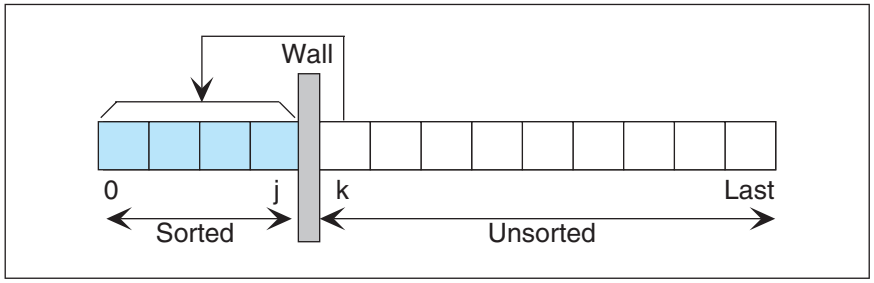

Figure 11-3 Insertion sort concept

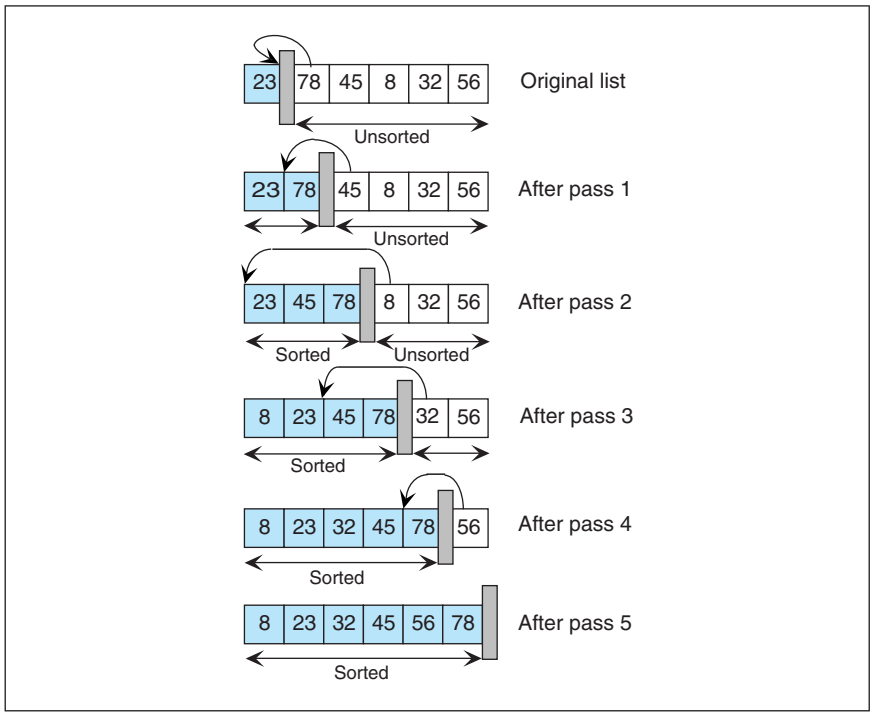

Figure 11-4 Insertion sort example

one element to the right as an element is removed from the unsorted sublist and inserted into the sorted sublist.

Insertion Sort Algorithm

The design of the insertion sort follows the pattern in the example. Each execution of the outer loop inserts the first element from the unsorted list into the sorted list. The inner loop steps through the sorted list, starting at the high end, looking for the correct insertion location. The pseudocode is shown in Algorithm 11-1.

```
algorithm insertionSort (ref list  <array>,
                         val last  <index>)
Sort list[0...last] using insertion sort. The array is divided
into sorted and unsorted lists. With each pass, the first
element in the unsorted list is inserted into the sorted list.
   Pre    list must contain at least one element
          last is an index to last element in the list
   Post   list has been rearranged
 1 current = 1
 2 loop (current <= last)
   1   hold   = list[current]
   2   walker = current - 1
   3   loop (walker >= 0 AND hold.key < list[walker].key)
       1   list[walker + 1]  = list[walker]
       2   walker            = walker - 1
   4   end loop
   5   list[walker + 1] = hold
   6   current = current + 1
 3 end loop
 4 return
end insertionSort
```

Algorithm 11-1 Straight insertion sort

Algorithm 11-1 Analysis

Two design concepts need to be explored in this simple algorithm. At some point in their execution, all sort algorithms must exchange two elements. Each exchange takes three statements, which can greatly impact the sort efficiency when many elements need to be exchanged. To improve the efficiency, therefore, we use a hold area. The beginning of the exchange moves the data currently being sorted to the hold area. This typical first move in any exchange is shown in Statement 2.1. The inner loop (Statement 2.3) shifts elements to the right until it finds the correct insertion location. Each of the shifts is one exchange move. Finally, when the correct location is found, the hold area is moved back to the array (Statement 2.5). This is the third statement in the exchange logic and completes the exchange.

Note

In the straight insertion sort, the list at any moment is divided into sorted and unsorted sublists. In each pass, the first element of the unsorted sublist is inserted into the sorted sublist.

Another point you should study is the workings of the inner loop. To make the sort as efficient as possible, we start with the high end of the sorted list and work toward the beginning of the sorted area. For the first insertion, this approach requires a maximum of one element to be shifted. For the second, it requires a maximum of two elements. The result is that only a portion of the list is examined in each sort phase.

Shell Sort

The **shell sort** algorithm, named after its creator, Donald L. Shell, is an improved version of the straight insertion sort. It was one of the first fast sorting algorithms.[4]

Note

The shell sort is an improved version of the straight insertion sort in which diminishing partitions are used to sort the data.

In the shell sort, a list of *N* elements is divided into *K* **segments,** where *K* is known as the **increment.** Each segment contains *N*/*K* or more elements. Figure 11-5 contains a graphic representation of the segments in a shell sort. Note that the segments are dispersed throughout the list. In Figure 11-5, the increment is 3; the first, fourth, seventh, and tenth elements make up segment 1; the second, fifth, and eighth elements make up segment 2; and the third, sixth, and ninth elements make up segment 3. After each pass through the data, the data in each segment are ordered. Thus, if there are three segments, as we see in Figure 11-5, there are three different ordered lists. If there are two segments, there are two ordered lists; if there is only one segment, then the list is sorted.

Shell Sort Algorithm

Each pass through the data starts with the first element in the array and progresses through the array, comparing adjacent elements in each segment. In Figure 11-5, we begin by comparing elements 1 and 4 from the first segment, then 2 and 5 from the second segment, and then 3 and 6 from the third segment. We then compare 4 and 7 from the first segment, 5 and 8 from the second segment, and so forth until finally we compare elements 7 and 10. If the elements are out of sequence, they are exchanged. Study Figure 11-5 carefully until you see each of these comparisons. Be sure to note also that the first segment has four elements, whereas the other two have only three. The number

4. Shell's algorithm was first published in *Communications of the ACM* 2, no. 7 (1959): 30–32.

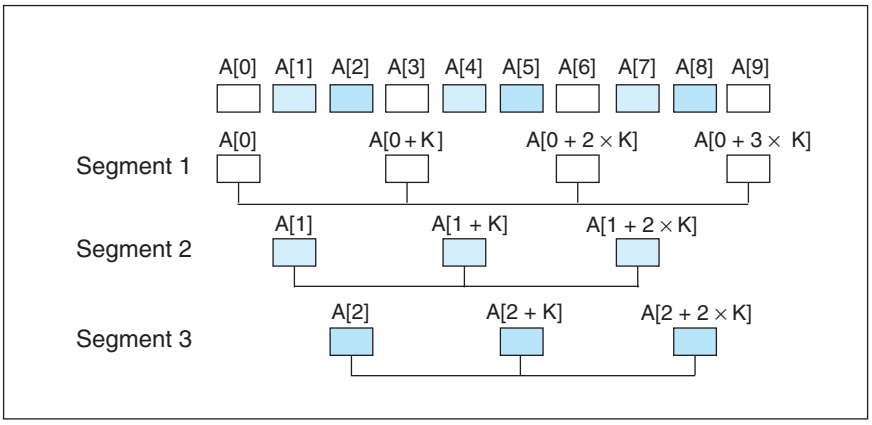

Figure 11-5 A segmented array

of elements in the segments varies because the list size (10) is not evenly divisible by the increment (3).

To compare adjacent keys in a segment, we add the increment to the current index, as shown below.

```
list[cur] : list[cur + incre]
```

After each pass through the data, the increment is reduced until, in the final pass, it is 1. Although the only absolute is that the last increment must be 1, the size of the increments influences the efficiency of the sort. We will discuss this issue separately later. The diminishing increment[5] is shown for an array of ten elements and increments of 5, 2, and 1 in Figure 11-6.

After each pass through the data, the elements in each segment are ordered. To ensure that each segment is ordered at the end of the pass, whenever an exchange is made, we drop back one increment and test the adjacent elements. If they are out of sequence, we exchange them and drop back again. If necessary, we keep exchanging and dropping back until we find two elements are ordered. We now have all of the elements of the shell sort. Its pseudocode is shown in Algorithm 11-2.

5. Knuth gave this sort the name "diminishing increment sort," but it is better known simply as the shell sort.

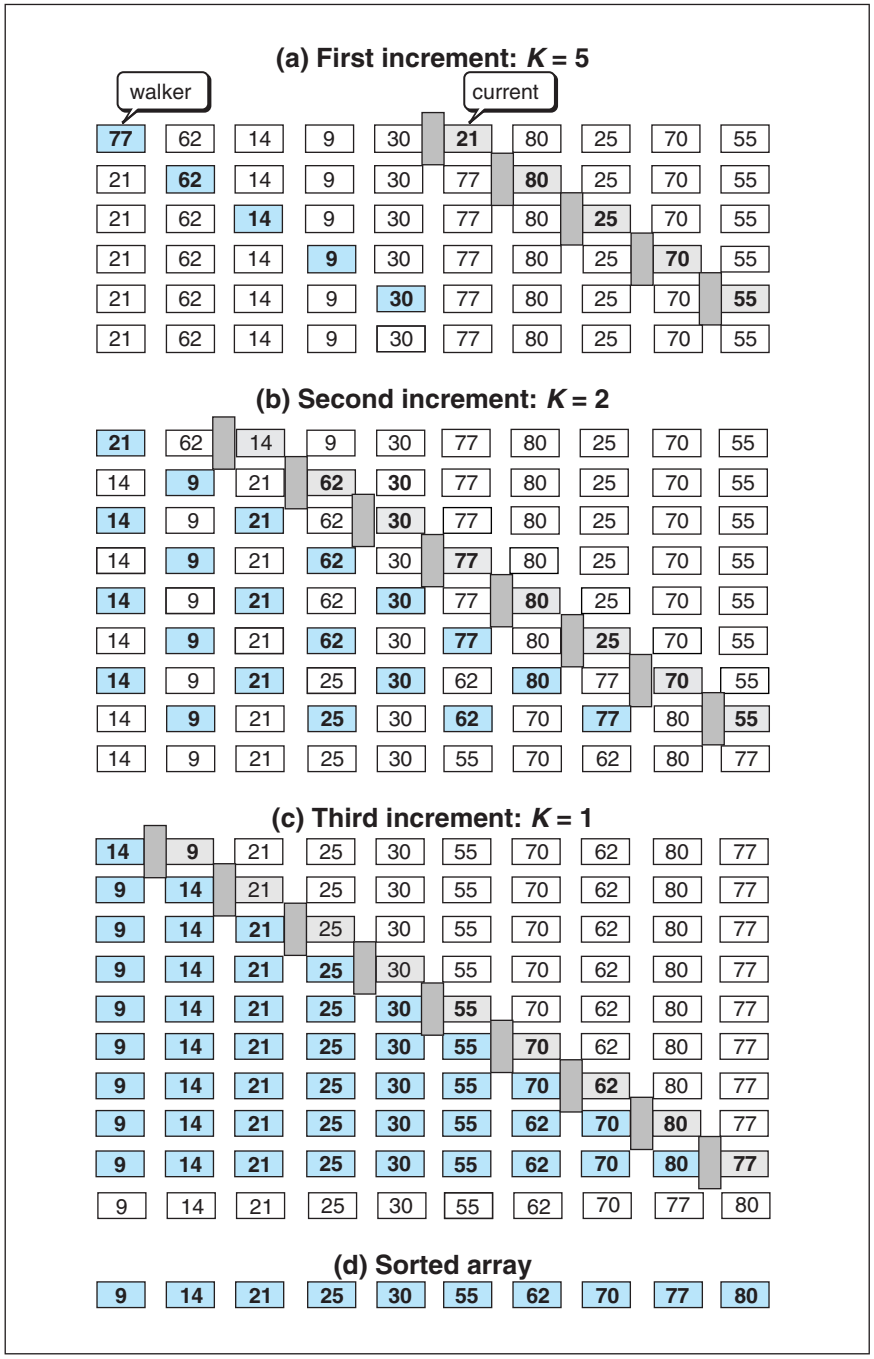

Figure 11-6 Diminishing increments in shell sort

```
algorithm shellSort (ref list  <array>,
                     val last  <integer>)
Data in list[0], list[1], ..., list[last] are sorted in place.
After the sort, their keys will be in order, list[0] ≤
list[1] ≤ ... ≤ list[last].
   Pre   list is an unordered array of records
         last is index to last record in array
   Post  list is ordered on list[i].key
 1 incre = last/2
Compare keys "increment" elements apart.
 2 loop (incre not 0)
   1   current = incre
   2   loop (current <= last)
       1   hold   = list[current]
       2   walker = current - incre
       3   loop (walker >= 0 AND hold.key < list[walker].key)
           Move larger element up in list.
           1   list [walker + incre] = list [walker]
           Fall back one partition.
           2   walker = walker - incre
       4   end loop
       Insert hold record in proper relative location.
       5   list [walker + incre] = hold
       6   current = current + 1
   3   end loop
   End of pass--calculate next increment.
   4   incre = incre/2
 3 end loop
 4 return
end shellSort
```

Algorithm 11-2 Shell sort

Algorithm 11-2 Analysis

Let's look at the shell sort carefully to see how it qualifies as an insertion sort. Recall that insertion sorts insert the new data into their correct location in the ordered portion of the list. This concept is found in the shell sort: the ordered portion of the list is a segment with its members separated by the increment size. To see this more clearly, look at Figure 11-7. In this figure, the segment is seen as the shaded elements in an array. The new element, *A*[0 + 3 * *K*], is being inserted into its correct position in the ordered portion of the segment.

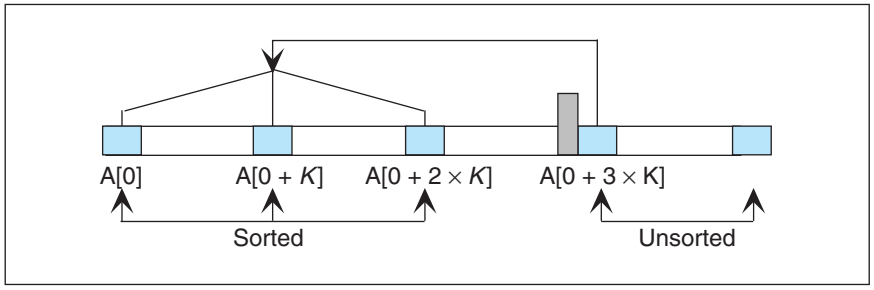

Figure 11-7 Ordered segment in a shell sort

Furthermore, if you compare the code in Algorithm 11-2 with the code in Algorithm 11-1, "Straight insertion sort," on page 507, you will see that, other than the increment, the code is identical.

One of the most important parts of the shell sort is falling back to ensure that the segment is ordered. This logic takes place in Statement 2.2.3.2. If this logic is not included, the resulting list will not be completely sorted.

Selecting the Increment Size

First, recognize that no increment size is best for all situations. The overriding considerations in the sort are to complete the sort with the minimum number of passes (increments) and to minimize the number of elements that appear in more than one segment. One method to eliminate completely an element being in more than one list is to use prime numbers. Unfortunately, the dynamic calculation of prime numbers is a relatively slow process.

Most texts use the simple series we proposed in Algorithm 11-2, setting the increment to half the list size and dividing by 2 each pass. Knuth suggests, however, that you should not start with an increment greater than one-third of the list size.[6] Other computer scientists have suggested that the increments be a power of 2 minus 1 or a Fibonacci series. These variations may result in a slightly more efficient sort, but they are relatively complex. One simple variation of the division-by-2 approach would be to add 1 whenever the increment is even. Doing so would tend to reduce the number of elements that appear in multiple segments.

Although you can use more complex increment-setting algorithms, the efficiency of a shell sort will never approach that of a quick sort. Therefore, if the objective is to obtain the most efficient sort, the solution is to use the quick sort rather than trying to optimize the increment size in the shell sort. On the other hand, the shell sort is a much simpler sort and at the same time is reasonably efficient.

6. Donald E. Knuth, *The Art of Computer Programming*, vol. 3, *Sorting and Searching* (Reading, MA: Addison-Wesley, 1973), 93.

Insertion Sort Algorithms

Sort algorithms determine the **sort effort** for a given sort. **Sort effort** is defined as the relative efficiency of a sort. It can be determined in several ways, but we will use the number of loops in the sort. Another common measure is the number of moves and compares needed to sort the list. Of course, the best measure would be the time it takes to actually run the sort. Time, however, varies by the efficiency of the program implementation and the speed of the computer being used. For analyzing different sorts, therefore, the first two measures are more meaningful. Let's now analyze the straight insertion and shell sort algorithms to determine their relative efficiency.

Straight Insertion Sort

Referring to Algorithm 11-1, "Straight insertion sort," on page 507, we find the following two loops:

```
2 loop (current <= last)
    ...
    3 loop (walker >= 0 AND hold.key < list[walker].key)
```

The outer loop executes $n - 1$ times, from 1 through `last`. For each outer loop, the inner loop executes from 0 to `current` times, depending on the relationship between `hold.key` and `list[walkerl.key`. On the average, we can expect the inner loop to process through the data in half of the sorted list. Because the inner loop depends on the outer loop's setting for `current`, we have a dependent quadratic loop, which is mathematically stated as

$$f(n) = n\left(\frac{n+1}{2}\right)$$

In big-O notation, the dependent quadratic loop is $O(n^2)$.

Note

The straight insertion sort efficiency is $O(n^2)$.

Shell Sort

Now let's look at Algorithm 11-2, "Shell sort," on page 511. This algorithm contains the nested loops shown below.

```
2 loop (incre not 0)
       ...
       2 loop (current <= last)
           ...
           3 loop (walker >= 0 AND hold.key < list[walker].key)
```

Because we are dividing the increment by 2 in each loop, the outer loop is logarithmic; it is executed $\log_2 n$ times. The first inner loop executes n – `increment` times for each of the outer loops; the first time it loops through 50% of the array ($n = (n/2)$), the second time it loops through 75% of the array ($n - (n/4)$), and so forth until it loops through all of the elements. The total number of iterations for the outer loop and the first inner loop is shown below.

$$\log_2 n \times \left[\left(n - \frac{n}{2}\right) + \left(n - \frac{n}{4}\right) + \left(n - \frac{n}{8}\right) + \cdots + 1\right] = n\log_2 n$$

The innermost loop is the most difficult to analyze. The first limit keeps us from falling off the beginning of the array. The second limit determines whether we have to loop at all: We loop only when the data are out of order. Sometimes the inner loop is executed zero times, sometimes it is executed anywhere from one to `incre` times. If we were able to derive a formula for the third factor, the total sort effort would be the product of the three loops. The first two loops have a combined efficiency of $O(n\log_2 n)$. However, we still need to include the third loop. We can see, therefore, that the result is something greater than $O(n\log_2 n)$.

Knuth[7] tells us that the sort effort for the shell sort cannot be derived mathematically. He estimates from his empirical studies that the average sort effort is $15n^{1.25}$. Reducing Knuth's analysis to a big-O notation, we see that the shell sort is $O(n^{1.25})$.

Note

> **The shell sort efficiency is $O(n^{1.25})$.**

Summary

Our analysis indicates that the shell sort is more efficient than the straight insertion sort. Table 11-1 shows the number of loops for each sort with different array sizes.

	Number of Loops	
n	Straight Insertion	Shell
25	625	55
100	10,000	316
500	250,000	2364
1000	1,000,000	5623
2000	4,000,000	13,374

Table 11-1 Comparison of straight insertion and shell sorts

7. Donald E. Knuth, *The Art of Computer Programming*, vol. 3, *Sorting and Searching* (Reading, MA: Addison-Wesley, 1973), 381.

One important point to note in analyzing Table 11-1 is that big-O notation is only an approximation. At small array sizes, it may not accurately indicate the relative merit of one sort over another. For example, heuristic studies indicate that the straight insertion sort is more efficient than the shell sort for small lists.

Insertion Sort Implementation

In this section we write C++ implementations of the straight insertion sort and the shell sort.

Straight Insertion Sort

The straight insertion sort's implementation follows the pseudocode very closely. To test the sort, we created an array of random integers. The parameter is therefore changed to integers and the comparisons reference list without a key qualification. The code is shown in Program 11-1.

```
 1  /* ===================== insertionSort =======================
 2     Sort list using insertion sort. The list is divided into
 3     sorted and unsorted lists. With each pass, first element
 4     in unsorted list is inserted into sorted list.
 5        Pre   list must contain at least one element
 6              last contains index to last element in the list
 7        Post  list has been rearranged
 8  */
 9  void insertionSort (int list[],
10                      int last)
11  {
12  // Local Definitions
13     int current;
14     int hold;
15     int walker;
16
17  // Statements
18     for (current = 1; current <= last; current++)
19        {
20         hold = list[current];
21         for (walker = current - 1;
22              walker >= 0 && hold < list[walker];
23              walker--)
24            list[walker + 1] = list[walker];
25         list [walker + 1] = hold;
26        } // for current
27
28     return;
29  } // insertionSort
```

Program 11-1 Insertion sort

Program 11-1 Analysis

We implement both loops using a *for* statement. When we begin the sort, the sorted list contains the first element. Therefore, we set the

loop start to position 1 for the unsorted list (Statement 18). In the inner loop, the limiting condition is the beginning of the array, as seen in Statement 22.

Shell Sort

The shell sort implementation, which also uses an array of integers, is shown in Program 11-2.

```
 1  /* ==================== shellSort ====================
 2     List[0], list[1], ..., list[last] are sorted in place.
 3     After the sort, their keys will be in order, list[0].key
 4     <= list[1].key <= ... <= list[last].key.
 5        Pre   list is an unordered array of integers
 6              last is index to last element in array
 7        Post  list is ordered
 8  */
 9  void shellSort (int list [],
10                  int last)
11  {
12  // Local Definitions
13     int hold;
14     int incre;
15     int curr;
16     int walker;
17
18  // Statements
19     incre = last / 2;
20     while (incre != 0)
21        {
22         for (curr = incre; curr <= last; curr++)
23            {
24             hold = list [curr];
25             walker = curr - incre;
26             while (walker >= 0 && hold < list [walker])
27                {
28                 // Move larger element up in list
29                 list [walker + incre] = list [walker];
30                 //  Fall back one partition
31                 walker = (walker - incre);
32                } //  while
33             // Insert hold in proper relative position
34             list [walker + incre] = hold;
35            } // for
36          //  End of pass--calculate next increment.
37          incre = incre / 2;
38        } // while
39     return;
40  } //  shellSort
```

Program 11-2 Shell sort

Program 11-2 Analysis

Although more complex than the straight insertion sort, the shell sort is by no means a difficult algorithm. There are only two additional complexities over the straight insertion sort. First, rather than a single array we have an array of partitions. Second, whenever an exchange is made, we must fall back and verify the order of the partition.

11-3 SELECTION SORTS

Selection sorts are among the most intuitive of all sorts. Given a list of data to be sorted, we simply select the smallest item and place it in a sorted list. These steps are then repeated until we have sorted all of the data. In this section we study two selection sorts, the straight selection sort and the heap sort.

Straight Selection Sort

In the straight selection sort, the list at any moment is divided into two sublists, sorted and unsorted, which are divided by an imaginary wall. We select the smallest element from the unsorted sublist and exchange it with the element at the beginning of the unsorted data. After each selection and exchange, the wall between the two sublists moves one element, increasing the number of sorted elements and decreasing the number of unsorted ones. Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed one sort pass. If we have a list of n elements, therefore, we need $n - 1$ passes to completely rearrange the data. The selection sort is graphically presented in Figure 11-8.

Note

In each pass of the selection sort, the smallest element is selected from the unsorted sublist and exchanged with the element at the beginning of the unsorted list.

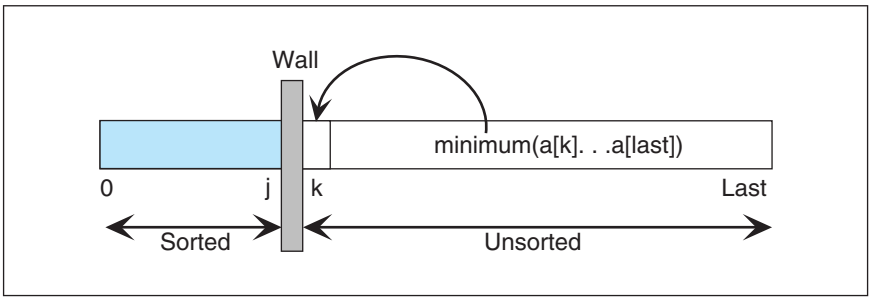

Figure 11-8 Selection sort concept

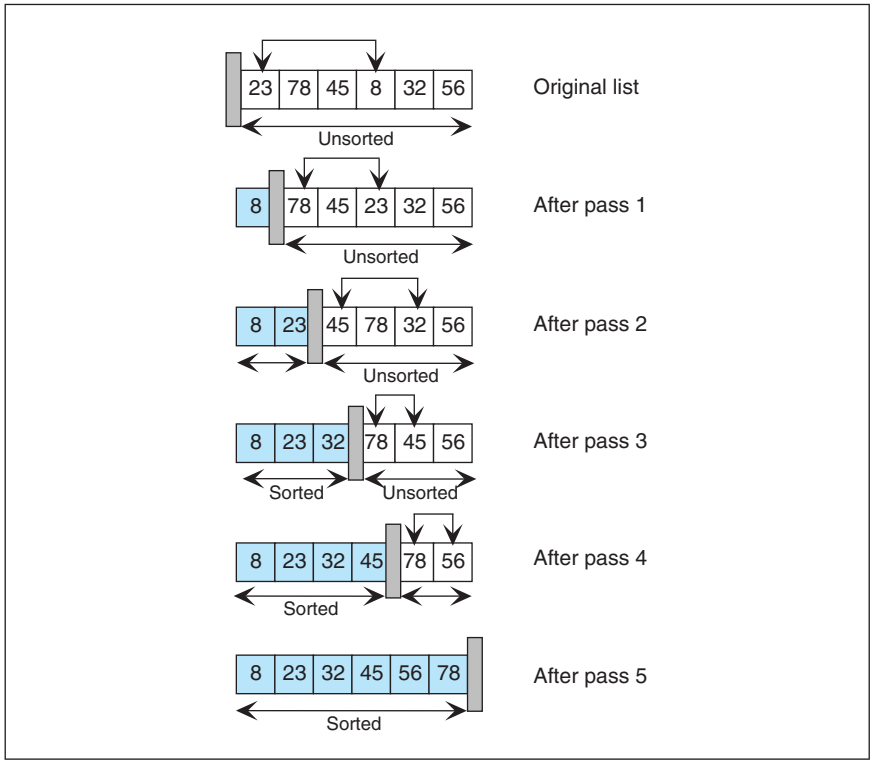

Figure 11-9 Selection sort example

Figure 11-9 traces our set of six integers as we sort them. It shows how the wall between the sorted and unsorted sublists moves in each pass. As you study the figure, you will see that the array is sorted after five passes, one less than the number of elements in the array. Thus, if we use a loop to control the sorting, our loop has one less iteration than the number of elements in the array.

Selection Sort Algorithm

If you knew nothing about sorting and were asked to sort a list on paper, you would undoubtedly scan the list to locate the smallest item and then copy it to a second list. You would then repeat the process of locating the smallest remaining item in the list and copying it to the new list until you had copied all items to the sorted list.

With the exception of using two areas, this is exactly how the selection sort works. Starting with the first item in the list, the algorithm scans the list for the smallest element and exchanges it with the item at the beginning of the list. Each selection and exchange is one sort pass. After advancing the index (wall), the sort continues until the list is completely sorted. The pseudocode is shown in Algorithm 11-3.

```
algorithm selectionSort (ref list  <array>,
                         val last  <index>)
Sorts list[0...last] by selecting smallest element in unsorted
portion of array and exchanging it with element at the
beginning of the unsorted list.
   Pre   list must contain at least one item
         last contains index to last element in the list
   Post  list has been rearranged smallest to largest
 1 current = 0
 2 loop (current < last)
   1   smallest = current
   2   walker = current + 1
   3   loop (walker <= last)
       1   if (list[walker] < list[smallest])
           1   smallest = walker
       2   walker = walker + 1
   4   end loop
   Smallest selected: exchange with current element.
   5   exchange (list, current, smallest)
   6   current = current + 1
 3 end loop
 4 return
end selectionSort
```

Algorithm 11-3 Selection sort

Heap Sort

In Chapter 9 we studied heaps. Recall that a heap is a tree structure in which the root contains the largest (or smallest) element in the tree. (You may want to review Chapter 9 before studying heap sort.) As a quick review, Figure 11-10 shows a heap in its tree form and in its array form.

The **heap sort** algorithm is an improved version of straight selection sort. The straight selection sort algorithm scans the unsorted elements and selects the smallest element. Finding the smallest element among the n elements requires $n - 1$ comparisons. This part of the selection sort makes it very slow.

The heap sort also selects an element from the unsorted portion of the list, but it is the largest element. Because the heap is a tree structure, however, we don't need to scan the entire list to locate the largest key. Rather, we reheap, which moves the largest element to the root by following tree branches. This ability to follow branches makes the heap sort much faster than the straight selection sort.

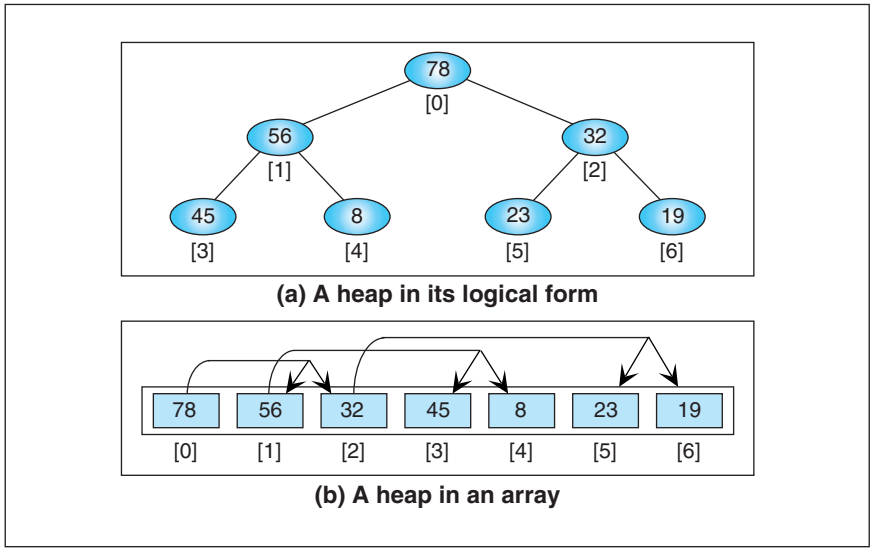

Figure 11-10 Heap representations

Heap sort begins by turning the array to be sorted into a heap. The array is turned into a heap only once for each sort. We then exchange the root, which is the largest element in the heap, with the last element in the unsorted list. This exchange results in the largest element being added to the beginning of the sorted list. We then reheap down to reconstruct the heap and exchange again. The reheap and exchange process continues until the entire list is sorted.

Note

The heap sort is an improved version of the selection sort in which the largest element (the root) is selected and exchanged with the last element in the unsorted list.

The heap sort process is shown in Figure 11-11. We first turn the unordered list into a heap. You should verify for yourself that the array is actually a heap.

Because the largest element (78) is at the top of the heap, we can exchange it with the last element in the heap and move the heap wall one element to the left. This exchange places the largest element in its correct location at the end of the array, but it destroys the heap. We therefore rebuild the heap. The smaller heap now has its largest element (56) at the top. We exchange 56 with the element at the end of the heap (23), which places 56 in its correct location in the sorted array. The reheap and exchange processing continues until we have sorted the entire array.

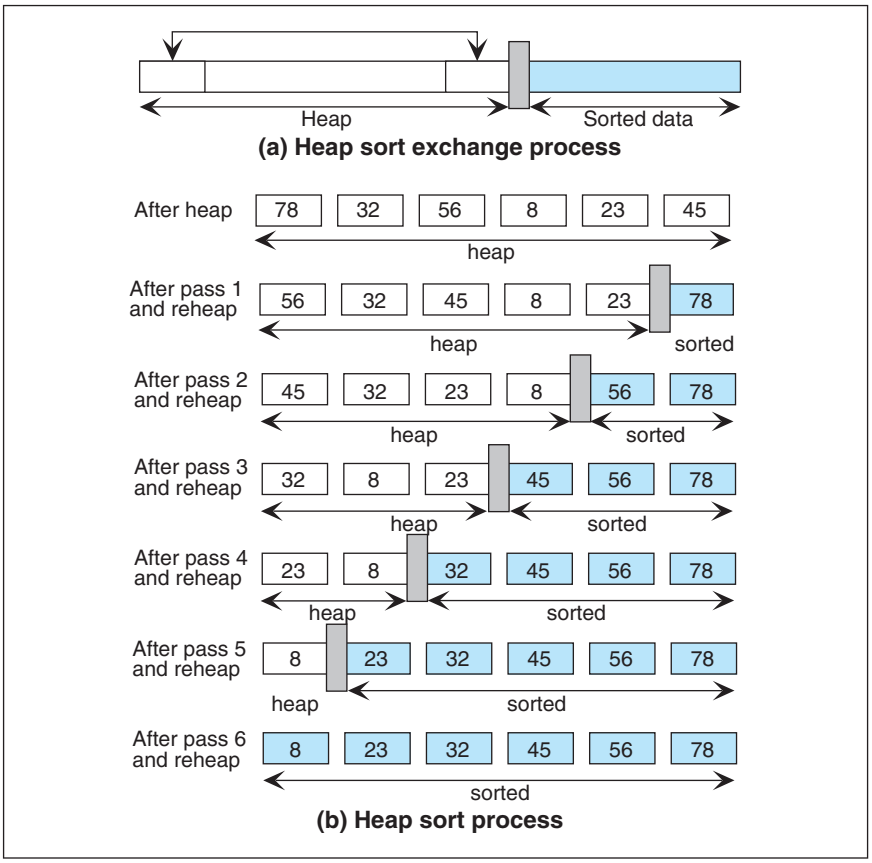

Figure 11-11 Heap sort process

Heap Sort Algorithm

Algorithm 11-4 contains the pseudocode for heap sort. It uses two algorithms defined in Chapter 9, Algorithm 9-1, "`reheapUp,`" on page 414, and Algorithm 9-2, "`reheapDown,`" on page 415.

```
algorithm heapSort (ref heap <array>,
                    val last <index>)
Sort an array, list[0 ... last], using a heap.
    Pre   array is filled
          last is index to last element in array
    Post  array has been sorted
Create heap
 1  walker = 1
 2  loop (walker <= last)
```

Algorithm 11-4 Heap sort

```
    1   reheapUp (heap, walker)
    2   walker = walker + 1
 3  end loop
Heap created. Now sort it.
 4  sorted = last
 5  loop (sorted > 0)
    1   exchange (heap, 0, sorted)
    2   sorted = sorted - 1
    3   reheapDown (heap, 0, sorted)
 6  end loop
 7  return
end heapSort
```

Algorithm 11-4 Heap sort *(continued)*

Algorithm 11-4 Analysis

Whereas the heap-array structure is relatively complex, the heap sort algorithm is deceptively simple. The algorithm begins by using `reheapUp` to turn the array into a heap. It then sorts the array by exchanging the element at the top of the heap with the element at the end of the heap and rebuilding the heap using `reheapDown`.

Selection Sort Algorithms

In this section, we examine the sort efficiency for the selection sorts.

Straight Selection Sort

The code for the straight selection sort is found in Algorithm 11-3, "Selection Sort," on page 519. It contains the two loops shown below.

```
2   loop (current < last)
    ...
    3   loop (walker <= last)
```

The outer loop executes $n - 1$ times. The inner loop also executes $n - 1$ times. This is a classic example of the quadratic loop. Its search effort, using big-O notation, is $O(n^2)$.

Note

The straight selection sort efficiency is $O(n^2)$.

Heap Sort

The heap sort code is shown in Algorithm 11-4. Ignoring the effort required to build the heap initially, the sort contains two loops. The first is a simple iterative loop; the second is a recursive loop:

```
5   loop (sorted > 0)
    ...
    3   reheapDown (heap, 0, sorted)
```

The outer loop starts at the end of the array and moves through the heap one element at a time until it reaches the first element. It therefore loops n times. The inner loop follows a branch down a binary tree from the root to a leaf or until the parent and child data are in heap order. The probability of the data being in order before we reach the leaf is very small so we will ignore it. The difficult part of this analysis is that for each of the outer loops, the heap becomes smaller, shortening the path from the root to a leaf. Again, except for the largest of heaps, this factor is rather minor and will be eliminated in big-O analysis; therefore, we will ignore it also.

Following the branches of a binary tree from a root to a leaf requires $\log_2 n$ loops. The sort effort, the outer loop times the inner loop, for the heap sort is therefore

$$n(\log_2 n)$$

When we include the processing to create the original heap, the big-O notation is the same. Creating the heap requires $n\log_2 n$ loops through the data. When factored into the sort effort, it becomes a coefficient, which is then dropped to determine the final sort effort.

Note

The heap sort efficiency is $O(n\log_2 n)$.

Summary

Our analysis leads to the conclusion that the heap sort is more efficient than the other sorts we have discussed. The straight insertion and straight selection sorts are both $O(n^2)$ sorts, the shell sort is $O(n^{1.25})$, and the heap sort is $O(n\log_2 n)$. Table 11-2 offers a comparison of the sorts. Note that according to a strict mathematical interpretation of the big-O notation, heap sort surpasses shell sort in efficiency as we approach 2000 elements to be sorted. Remember two points, however: First, big-O is a rounded approximation; all coefficients and many of the factors have been removed. Second, big-O is based on an analytical evaluation of the algorithms, not an evaluation of the code. Depending on the algorithm's implementation, the actual run time can be affected.

n	Number of Loops		
	Straight Insertion Straight Selection	Shell	Heap
25	625	55	116
100	10,000	316	664
500	250,000	2364	4482
1000	1,000,000	5623	9965
2000	4,000,000	13,374	10,965

Table 11-2 Comparison of insertion and selection sorts

Selection Sort Implementation

We now turn our attention to implementing the selection sort algorithms in C++. We first implement the straight selection sort and then the heap sort.

Selection Sort C++ Code

The code for the selection sort is shown in Program 11-3.

```
/* ================== selectionSort ====================
   Sorts list[0...last] by selecting smallest element in
   unsorted portion of array and exchanging it with element
   at the beginning of the unsorted list.
      Pre  list must contain at least one item
           last contains index to last element in the list
      Post list has been rearranged smallest to largest
*/
void selectionSort  (int list[],
                     int last)
{
// Local Definitions
   int current;
   int smallest;
   int holdData;
   int walker;

// Statements
   for (current = 0; current < last; current++)
      {
       smallest = current;
       for (walker = current + 1;
            walker <= last;
            walker++)
          if (list[walker] < list[smallest])
              smallest = walker;
```

Program 11-3 Selection sort

```
27
28          // Smallest selected: exchange with current
29          holdData        = list[current];
30          list[current]   = list[smallest];
31          list[smallest]  = holdData;
32        } //  for current
33     return;
34  } // selectionSort
```

Program 11-3 Selection sort *(continued)*

Heap Sort C++ Code

The heap sort program requires a driver function, which we call heap sort, and two functions we disccused earlier, reheap up and reheap down. We have tailored them here for the sorting process.

Heap Sort Function

The heap sort function accepts an array of unsorted data and an index to the last element in the array. It then creates a heap and sorts it using reheap up and reheap down. The heap sort algorithm is shown in Program 11-4.

```
1  /* ======================== heapSort ========================
2     Sort an array, list[0...last], using a heap.
3        Pre   list must contain at least one item
4              last contains index to last element in the list
5        Post  list has been rearranged smallest to largest
6  */
7  void heapSort  (int  list[],
8                  int  last)
9  {
10 // Local Definitions
11    int sorted;
12    int holdData;
13    int walker;
14
15 // Statements
16    // Create heap
17    for (walker = 1; walker <= last; walker++)
18        reheapUp (list, walker);
19
20 // Heap created. Now sort it.
21    sorted = last;
22    while (sorted > 0)
23       {
24        holdData      = list[0];
25        list[0]       = list[sorted];
26        list[sorted]  = holdData;
```

Program 11-4 Heap sort

```
27          sorted--;
28          reheapDown (list, 0, sorted);
29        } // while
30      return;
31    } // heapSort
```

Program 11-4 Heap sort *(continued)*

Program 11-4 Analysis

The heap sort implementation follows the pseudocode closely. Using an index, we walk through the array, calling reheap up to create the heap. Once the heap has been created, we exchange the element at the top of the heap with the last element in the heap and adjust the heap size down by 1. We then call reheap down to recreate the heap by moving the root element down the tree to its correct location.

11-4 EXCHANGE SORTS

The third category of sorts, exchange sorting, contains the most common sort taught in computer science, the bubble sort, and the most efficient general-purpose sort, quick sort. In exchange sorts, we exchange elements that are out of order until the entire list is sorted. Although virtually every sorting method uses some form of exchange, the sorts in this section use it extensively.

Bubble Sort

In the **bubble sort**, the list at any moment is divided into two sublists: sorted and unsorted. The smallest element is bubbled from the unsorted sublist and moved to the sorted sublist. After moving the smallest to the sorted list, the wall moves one element to the right, increasing the number of sorted elements and decreasing the number of unsorted ones (Figure 11-12). Each time an element moves from the unsorted sublist to the sorted sublist, one sort pass is completed. Given a list of n elements, bubble sort requires up to $n - 1$ passes to sort the data.

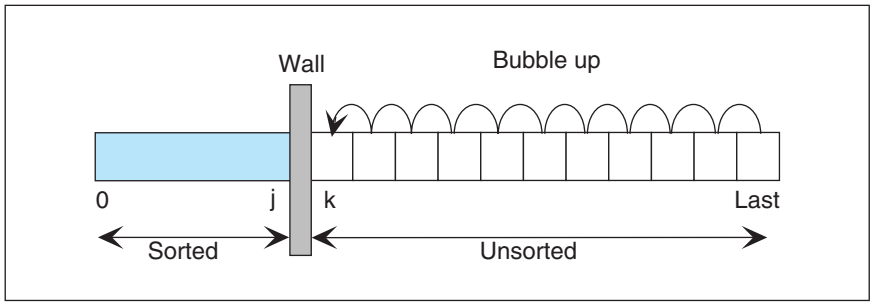

Figure 11-12 Bubble sort concept

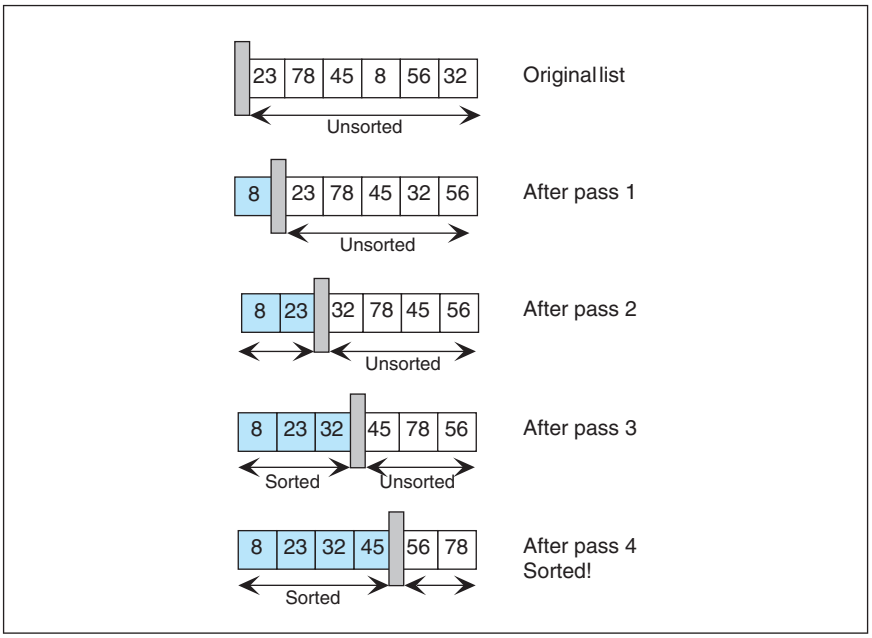

Figure 11-13 Bubble sort example

Figure 11-13 shows how the wall moves one element in each pass. Looking at the first pass, we start with 32 and compare it with 56. Because 32 is less than 56, we exchange the two and step down one element. We then compare 32 and 8. Because 32 is not less than 8, we do not exchange these elements. We step down one element and compare 45 and 8. They are out of sequence, so we exchange them and step down again. Because we moved 8 down, it is now compared with 78 and these two elements are exchanged. Finally, 8 is compared with 23 and exchanged. This series of exchanges places 8 in the first location and the wall is moved up one position.

Note

In each pass of the bubble sort, the smallest element is bubbled from the unsorted sublist and moved to the sorted sublist.

Bubble Sort Algorithm

Like the insertion and selection sorts, the bubble sort is quite simple. In each pass through the data, the smallest element is bubbled to the beginning of the unsorted segment of the array. In the process, adjacent elements that are out of order are exchanged, partially ordering the data. When the smallest element is encountered, it is automatically bubbled to the beginning of the unsorted list. The sort then continues by making another pass through the unsorted list. The code for bubble sort is shown in Algorithm 11-5.

```
algorithm bubbleSort (ref list <array>,
                      val last <index>)
Sort an array, list[0...last], using bubble sort. Adjacent
elements are compared and exchanged until list is
completely ordered.
   Pre    list must contain at least one item
          last contains index to last element in the list
   Post   list has been rearranged in sequence low to high
 1 current   = 0
 2 sorted    = false
 3 loop (current <= last AND sorted false)
   Each iteration is one sort pass.
   1   walker  = last
   2   sorted  = true
   3   loop (walker > current)
       1   if (list[walker] < list[walker - 1])
           Any exchange means list is not sorted.
           1   sorted = false
           2   exchange (list, walker, walker - 1)
       2   end if
       3   walker = walker - 1
   4   end loop
   5   current = current + 1
 4 end loop
 5 return
end bubbleSort
```

Algorithm 11-5 Bubble sort

Algorithm 11-5 Analysis

If you have studied other bubble sort algorithms, you may have noticed a slight improvement in this version of the sort. If an exchange is not made in a pass (Statement 3), then we know the list is already sorted and the sort can stop. At Statement 3.2, we set a Boolean, `sorted`, to true. If at any time during the pass an exchange is made, `sorted` is changed to false, indicating that the list was not sorted when the pass began.

Another difference you may have noticed is that we started from the high end and bubbled down. As a historical note, the bubble sort was originally written to "bubble up" the highest element in the list. From an efficiency point of view, it makes no difference whether the largest element is bubbled down or the smallest element is bubbled up. From a consistency point of view, however, comparisons between the sorts

are easier if all three of our basic sorts (insertion, selection, and exchange) work in the same manner. For that reason, we have chosen to bubble the lowest key in each pass.

Quick Sort

In the bubble sort, consecutive items are compared and possibly exchanged on each pass through the list, which means that many exchanges may be needed to move an element to its correct position. **Quick sort** is an exchange sort developed by C. A. R. Hoare in 1962. It is more efficient than the bubble sort because a typical exchange involves elements that are far apart, so fewer exchanges are required to correctly position an element.

Each iteration of the quick sort selects an element, known as **pivot,** and divides the list into three groups: a partition of elements whose keys are less than the pivot's key, the pivot element that is placed in its ultimately correct location in the list, and a partition of elements greater than or equal to the pivot's key. The sorting then continues by quick sorting the left partition followed by quick sorting the right partition. This partitioning is shown in Figure 11-14.

Hoare's original algorithm selected the pivot key as the first element in the list. In 1969, R. C. Singleton improved the sort by selecting the pivot key as the median value of three elements: left, right, and an element in the middle of the list. Once the median value is determined, it is exchanged with the left element. We implement Singleton's variation of quick sort.

Note

> **Quick sort is an exchange sort in which a pivot key is placed in its correct position in the array while rearranging other elements widely dispersed across the list.**

Knuth suggests that when the sort partition becomes small, a straight insertion sort be used to complete the sorting of the partition.[8] Although the mathematics to optimally choose the minimum partition size are quite complex, the partition size should be relatively small; we recommend 16.

Quick Sort Algorithm

We are now ready to develop the algorithm. As you may have anticipated from our discussion, we will use a recursive algorithm for quick sort. In addition to the basic algorithm, two supporting algorithms are required, one to determine the pivot element and one for the straight insertion sort. We discuss these two algorithms first.

---

8. Donald E. Knuth, *The Art of Computer Programming,* vol. 3, *Sorting and Searching* (Reading, MA: Addison-Wesley, 1973), 116.

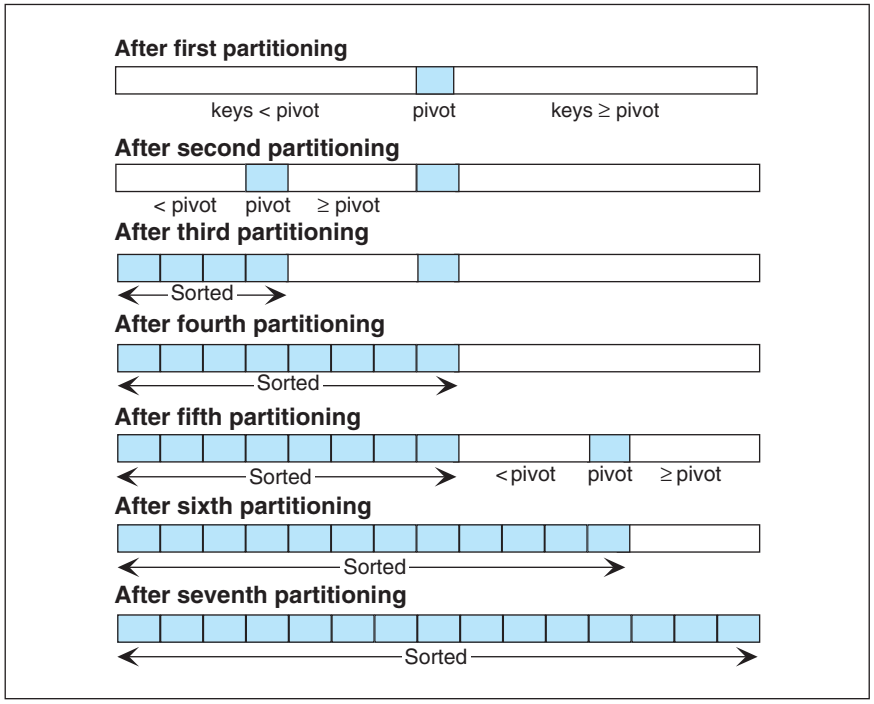

Figure 11-14 Quick sort partitions

Straight Insertion Module

The straight insertion sort is a minor variation on the algorithm developed earlier, which always sorted a list beginning at location 0. Because the partition to be sorted in the quick sort can be found anywhere in the array, we must be able to sort partitions beginning at locations other than 0. We therefore add a parameter that specifies the starting location of the location to be sorted in addition to its ending location. The modified code is shown in Algorithm 11-6.

```
algorithm quickInsertion (ref list   <array>,
                          val first  <index>,
                          val last   <index>)
Sort list[first...last] using insertion sort. The list is divided
into sorted and unsorted lists. With each pass, first element
in unsorted list is inserted into sorted list. This is a special
version of the insertion sort modified for use with quick sort.
    Pre    list must contain at least one element
           first is an index to first element in the list
           last is an index to last element in the list
```

Algorithm 11-6 Quick sort's straight insertion sort module

```
    Post   list has been rearranged
 1  current = first + 1
 2  loop (current <= last)
    1   hold    = list[current]
    2   walker = current - 1
    3   loop (walker >= first AND hold.key < list[walker].key)
        1   list[walker + 1]  = list[walker]
        2   walker            = walker - 1
    4   end loop
    5   list[walker + 1] = hold
    6   current = current + 1
 3  end loop
 4  return
end quickInsertion
```

Algorithm 11-6 Quick sort's straight insertion sort module *(continued)*

Determine Median of Three

The logic to select the median location requires three tests. First, we test the left and middle elements; if they are out of sequence, we exchange them. Then, we test the left and right elements; if they are out of sequence, we exchange them. Finally, we test the middle and right elements; if they are out of sequence, we exchange them. At this point, the three elements are in order.

```
array[left] ≤ array[middle] ≤ array[right]
```

We see, therefore, that this logic is based on the logical proposition that if a is less than b and b is less than c, then a is less than c. Finally, before we leave the algorithm, we exchange the left and middle elements, thus positioning the median valued element at the left of the array. The pseudocode is shown in Algorithm 11-7.

```
algorithm medianLeft (ref sortData  <array>,
                      val left      <index>,
                      val right     <index>)
Find the median value of an array, sortData [left ... right], and
place it in the location sortData[left].
    Pre    sortData is an array of at least three elements
           left and right are the boundaries of the array
    Post   median value located and placed at sortData[left]
Rearrange sortData so median value is in middle location.
```

Algorithm 11-7 Median left

```
 1  mid = (left + right)/2
 2  if (sortData[left].key > sortData[mid].key)
        1   exchange (sortData, left, mid)
 3  end if
 4  if (sortData[left].key > sortData[right].key)
        1   exchange (sortData, left, right)
 5  end if
 6  if (sortData[mid].key > sortData[right].key)
        1   exchange (sortData, mid, right)
 7  end if
Median is in middle location. Exchange with left.
 8  exchange (sortData, left, mid)
 9  return
end medianLeft
```

Algorithm 11-7 Median left *(continued)*

Algorithm 11-7 Analysis

The logic to determine a median value can become unintelligible very quickly. The beauty of this algorithm is its simplicity. It approaches the determination of the median value by performing a very simple sort on the three elements, placing the median in the middle location. It then exchanges the middle element with the left element.

Quick Sort Algorithm

We now turn our attention to the quick sort algorithm itself. It contains an interesting program design that you will find useful in other array applications. To determine the correct position for the pivot element, we work from the two ends of the array toward the middle. Because we have used the median value of three elements to determine the pivot element, the pivot may end up near the middle of the array, although this is not guaranteed. Quick sort is most efficient when the pivot's location is the middle of the array. The technique of working from the ends to the middle is shown in Figure 11-15.

As you study Figure 11-15, note that before the exchanges start, the median element is in the middle position and the smallest of the three elements used to determine the median is in the right location. After calling the median left algorithm, the median is in the left position and the smallest is in the middle location. The pivot key is then moved to a hold area to facilitate the processing. This move is actually the first part of an exchange that will put the pivot key in its correct location.

To help follow the sort, envision two walls, one on the left just after the pivot and one on the right (see Figure 11-15). We start at the left wall and move right while looking for an element that belongs on the right of the pivot. We locate one at element 97. After finding an element on the left that belongs on the right, we start at the right wall and move left while looking for an element that belongs on the left of the pivot.

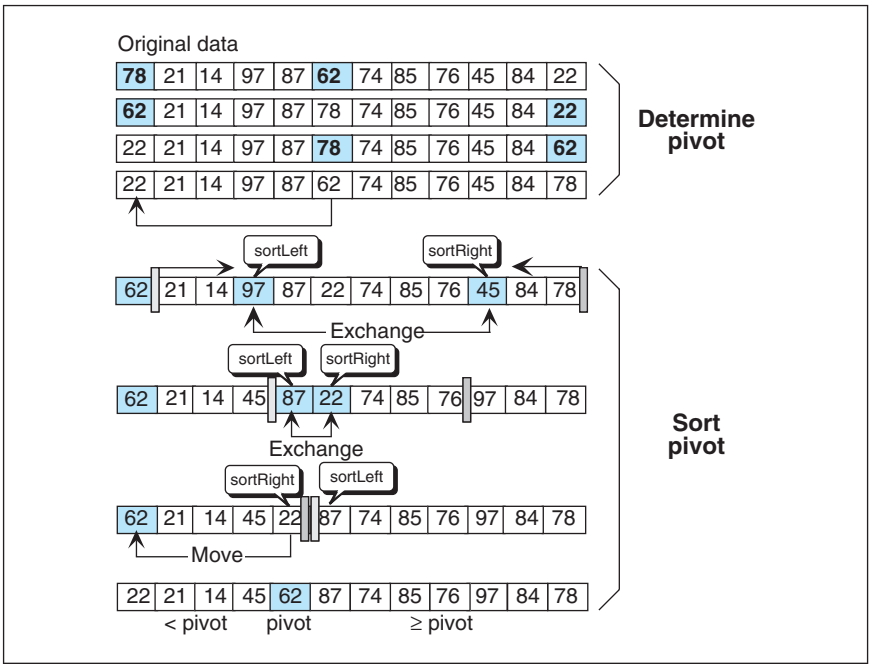

Figure 11-15 Quick sort pivot

We find one at element 45. At this point we exchange the two elements, move the walls to the left of element 87 and the right of element 76, and repeat the process.

This time, as we move to the right, the first left element, 87, belongs on the right. When we move down from the right, we find that element 22 belongs on the left. Again, we exchange the left and right elements and move the wall. At this point, we note that the walls have crossed. We have thus located the correct position for the pivot element. We move the data in the pivot's location to the first element of the array and then move the pivot back to the array, completing the exchange we started when we moved the pivot key to the hold area. The list has now been rearranged so that the pivot element (62) is in its correct location in the array relative to all of the other elements in the array. The elements on the left of the pivot are all less than the pivot, and the elements on the right of the pivot are all greater than the pivot. Any equal data would be found on the right of the pivot.

After placing the pivot key in its correct location, we recursively call quick sort to sort the left partition. When the left partition is completely sorted, we recursively call quick sort to sort the right partition. When both the left partition and the right partition have been sorted, the list is completely sorted. The data in Figure 11-15 are completely sorted using only quick sort in Figure 11-16. The pseudocode is shown in Algorithm 11-8.

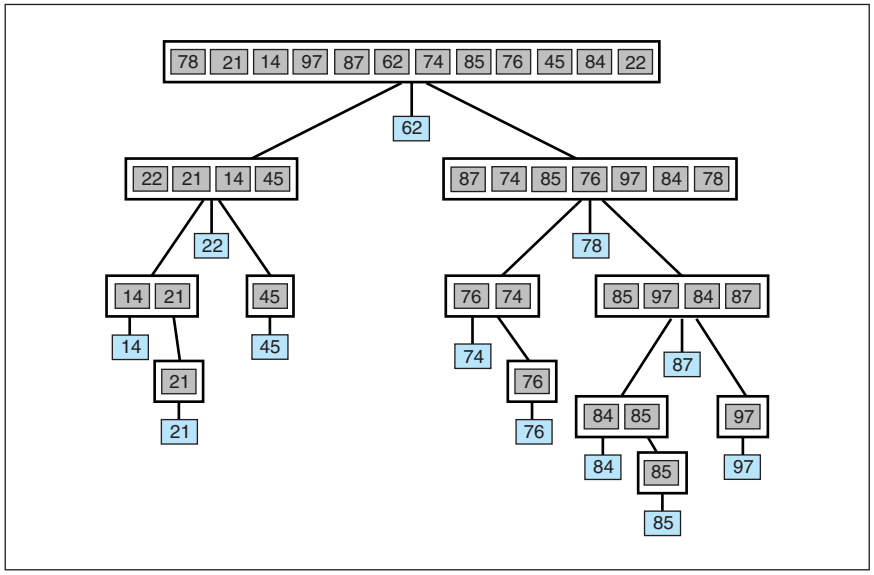

Figure 11-16 Quick sort operation

```
algorithm quickSort (ref list  <array>,
                     val left  <index>,
                     val right <index>)
An array, list [left...right] is sorted using recursion.
   Pre    list is an array of data to be sorted
          left and right identify the first and last elements
          of the list, respectively
   Post   list is sorted
 1 if ((right - left) > minSize)
   quick sort
   1   medianLeft (list, left, right)
   2   pivot             =  list [left]
   3   sortLeft          =  left + 1
   4   sortRight         =  right
   5   loop (sortLeft    <= sortRight)
       Find key on left that belongs on right
       1   loop (list[sortLeft].key < pivot.key)
           1   sortLeft = sortLeft + 1
       2   end loop
       Find key on right that belongs on left
       3   loop (list[sortRight].key >= pivot.key)
```

Algorithm 11-8 Quick sort

```
            1   sortRight = sortRight - 1
        4   end loop
        5   if (sortLeft <= sortRight)
            1   exchange(list, sortLeft, sortRight)
            2   sortLeft   =  sortLeft + 1
            3   sortRight  =  sortRight - 1
        6   end if
    6   end loop
    Prepare for next phase
    7   list [left]          = list [sortLeft - 1]
    8   list [sortLeft - 1]  = pivot
    9   if  (left < sortRight)
        1   quickSort (list, left, sortRight - 1)
    10  end if
    11  if (sortLeft < right)
        1   quickSort (list, sortLeft, right)
    12  end if
2   else
    1   insertionSort (list, left, right)
3   end if
4   end quickSort
```

Algorithm 11-8 Quick sort *(continued)*

Algorithm 11-8 Analysis

In addition to the design that works from both ends to the middle, two aspects of this algorithm merit further discussion. The loops used to determine the sort left and sort right elements (Statements 1.5.1 and 1.5.3) test only one condition. Most loops that process an array must also test for the end of the array, but we can omit the test in this algorithm based on selection of the pivot key. The pivot key is guaranteed to be the median value of three elements, the first, the last, and the one in the middle. Therefore, the median key cannot be less than the leftmost key nor greater than the rightmost key. At worst, it will be equal to the leftmost or rightmost key.

Assume that we had the worst case: all of the elements in a partition have the same key value. In this case, the pivot key would be equal to the leftmost and the rightmost keys. When we start on the left to move up the list, we stop immediately because the pivot key is equal to the `sortLeft` key. When we then start on the right and move down, we will move beyond the beginning of the list when all keys are equal. However, because the two elements have crossed, we will not exchange elements. Rather, we will move to Statement 1.7. In this case, we never use the sort right index that has moved off the beginning of the array.

Hoare's original algorithm was not recursive. Because he had to maintain a stack, he incorporated logic to ensure that the stack size was kept to a minimum. Rather than simply sort the left partition, he determined which partition was larger and put it in the stack while he sorted the smaller partition. We are not concerned with minimizing the stack size for two reasons. First, we have chosen the pivot key to be the median value. Therefore, the size of the two partitions should be generally the same, thus minimizing the number of recursive calls. More important, because we are using recursion, we do not need to determine the size of the stack in advance. We simply call on the system to provide stack space.

Exchange Sort Algorithms

In the exchange sorts we find what Knuth called the best general-purpose sort, quick sort. Let's determine their sort efforts to see why.

Bubble Sort

The code for the bubble sort is shown in Algorithm 11-5, "Bubble Sort," on page 528. As we saw with the straight insertion and straight selection sorts, it uses two loops to sort the data. The loops are shown below.

```
3   loop (current <= last AND sorted false)
    ...
    3   loop (walker > current)
```

The outer loop tests two conditions, the current index and a sorted flag. Assuming that the list is not sorted until the last pass, we loop through the array *n* times. The number of loops in the inner loop depends on the current location in the outer loop. It therefore loops through half the list on the average. The total number of loops is the product of both loops, making the bubble sort efficiency

$$n\left(\frac{n+1}{2}\right)$$

In big-O notation, the bubble sort efficiency is $O(n^2)$.

Note

> **The bubble sort efficiency is $O(n^2)$.**

Quick Sort

The code for the quick sort is shown in Algorithm 11-8, "Quick Sort," on page 534. A quick look at the algorithm reveals that there are five loops (three iterative loops and two recursive loops). The algorithm also contains the straight insertion sort as a subroutine. Because it is pos-

sible to use the quick sort to completely sort the data—that is, it is possible to eliminate the insertion sort—we analyze the quick sort portion as though the insertion sort weren't used. The quick sort loops are shown below.

```
5   loop (sortLeft <= sortRight)
    ...
    1   loop (list[sortLeft].key < pivot.key)
        ...
    2   end loop
    3   loop (list[sortRight].key >= pivot.key)
        ...
    4   end loop
    ...
6   end loop
9   if (left < sortRight)
    1   quickSort (list, left, sortRight - 1)
10  end if
11  if (sortLeft < right)
    1   quickSort (list, sortLeft, right)
12  if (sortLeft < right)
```

Recall that each pass in the quick sort divides the list into three parts, a list of elements smaller than the pivot key, the pivot key, and a list of elements greater than the pivot key. The first loop (Statement 5), in conjunction with the two nested loops (Statements 5.1 and 5.3), looks at each element in the portion of the array being sorted. Statement 5.1 loops through the left portion of the list; Statement 5.3 loops through the right portion of the list. Together, therefore, they loop through the list n times.

Similarly, the two recursive loops process one portion of the array each, either on the left or the right of the pivot key. The question is how many times they are called. Recall that we said that quick sort is most efficient when the pivot key is in the middle of the array. That's why we used a median value. Assuming that it is located relatively close to the center, we see that we have divided the list into two sublists of roughly the same size. Because we are dividing by 2, the number of loops is logarithmic. The total sort effort is therefore the product of the first loop times the recursive loops, or $n\log_2 n$.

Note

The quick sort efficiency is $O(n\log_2 n)$.

Algorithmics does not explain why we use the straight insertion sort when the list is small. The answer lies in the algorithm code and the

overhead of recursion. When the list becomes sufficiently small, it is simply more efficient to use a straight insertion sort.

11-5 SUMMARY

Our analysis leads to the conclusion that the quick sort's efficiency is the same as that of the heap sort. This is true because big-O notation is only an approximation of the actual sort efficiency. Although both are on the order of $n\log_2 n$, if we were to develop more accurate formulas to reflect their actual efficiency, we would see that the quick sort is actually more efficient. Table 11-3 summarizes the six sorts we discuss in this chapter.

	Number of Loops		
n	Straight Insertion Straight Selection Bubble Sorts	Shell	Heap and Quick
25	625	55	116
100	10,000	316	664
500	250,000	2364	4482
1000	1,000,000	5623	9965
2000	4,000,000	13,374	10,965

Table 11-3 Sort comparisons

We recommend that you use the straight insertion sort for small lists and the quick sort for large lists. Although the shell sort is an interesting sort that played an important role in the history of sorts, we would not recommend it in most cases. As you will see when we discuss external sorting, algorithms such as the heap sort play an important niche role in special situations and for that reason also belong in your sorting tool kit.

Exchange Sort Implementation

Let's look at the C++ code for the bubble sort and quick sort algorithms.

Bubble Sort Code

The bubble sort code is shown in Program 11-5.

```
1  /* ==================== bubbleSort ====================
2     Sort list using bubble sort. Adjacent elements are
3     compared and exchanged until list completely ordered.
4        Pre   list must contain at least one item
5              last contains index to last element in list
6        Post  list rearranged in sequence low to high
7  */
```

Program 11-5 Bubble sort

```
 8  void bubbleSort  (int list [],
 9                    int last)
10  {
11  // Local Definitions
12     int   current;
13     int   walker;
14     int   temp;
15     bool  sorted;
16
17  // Statements
18     // Each iteration is one sort pass
19     for (current = 0, sorted = false;
20          current <= last && !sorted;
21          current++)
22        for (walker = last, sorted = true;
23             walker > current;
24             walker--)
25           if (list[walker]  < list[walker - 1])
26              // Any exchange means list is not sorted
27              {
28               sorted           = false;
29               temp             = list[walker];
30               list[walker]     = list[walker - 1];
31               list[walker - 1] = temp;
32              } // if
33     return;
34  } // bubbleSort
```

Program 11-5 Bubble sort *(continued)*

Quick Sort Code

In this section we implement the three quick sort algorithms described earlier. The sort array is once again a simple array of integers.

The first function is quick sort, shown in Program 11-6. The modified version of the straight insertion sort is shown in Program 11-7, and the median function is shown in Program 11-8.

```
 1  /* Array sortData[left..right] is sorted using recursion.
 2        Pre   sortData is an array of data to be sorted
 3              left identifies the first element of sortData
 4              right identifies the last element of sortData
 5        Post  sortData array is sorted
 6  */
 7  void quickSort  (int  sortData[],
 8                   int  left,
 9                   int  right)
10  {
```

Program 11-6 Quick sort

```
#define MIN_SIZE 4

// Local Definitions
   int sortLeft;
   int sortRight;
   int pivot;
   int hold;

// Statements
   if ((right - left) > MIN_SIZE)
      // quick sort
      {
       medianLeft (sortData, left, right);
       pivot      = sortData [left];
       sortLeft   = left + 1;
       sortRight  = right;

       while (sortLeft <= sortRight)
          {
           // Find key on left that belongs on right
           while (sortData [sortLeft] < pivot)
              sortLeft++;
           // Find key on right that belongs on left
           while (sortData[sortRight] >= pivot)
              sortRight--;
           if (sortLeft <= sortRight)
              {
               hold  = sortData[sortLeft];
               sortData[sortLeft]   = sortData[sortRight];
               sortData[sortRight]  = hold;
               sortLeft++;
               sortRight--;
              } /* if */
          } // while
       // Prepare for next phase
       sortData [left] = sortData [sortLeft - 1];
       sortData [sortLeft - 1] = pivot;
       if  (left    < sortRight)
           quickSort (sortData, left, sortRight - 1);
       if (sortLeft < right)
           quickSort (sortData, sortLeft, right);
      } // if right > minimum
   else
       quickInsertion (sortData, left, right);
   return;
} // end quickSort
```

Program 11-6 Quick sort *(continued)*

```
/* Sort list[0...last] using insertion sort. The list is
   divided into sorted and unsorted lists. With each pass,
   first element in unsorted list is inserted into sorted
   list. This is a special version of the insertion sort
   modified for use with quick sort.
      Pre   list must contain at least one element
            first is an index to first element in the list
            last is an index to last element in the list
      Post  list has been rearranged
*/
void quickInsertion (int sortData[],
                     int first,
                     int last)
{
// Local Definitions
   int current;
   int hold;
   int walker;

// Statements
   for (current = first + 1;
        current <= last;
        current++)
       {
        hold    = sortData[current];
        walker  = current - 1;
        while (walker >= first
            && hold   <  sortData[walker])
           {
            sortData[walker + 1] = sortData[walker];
            walker--;
           } // while
        sortData[walker + 1] = hold;
       } // for
   return;
} // end quickInsertion
```

Program 11-7 Modified straight insertion sort for quick sort

```
/* Find the median value of an array, sortData[left..right],
   and place it in the location sortData[left].
      Pre  sortData is an array of at least three elements
           left and right are the boundaries of the array
      Post median value placed at sortData[left]
*/
```

Program 11-8 Median left for quick sort

```
 7  void medianLeft (int sortData[],
 8                   int left,
 9                   int right)
10  {
11  // Local Definitions
12     int mid;
13     int hold;
14
15  // Statements
16     //  Rearrange sortData so median is in middle location
17     mid = (left + right) / 2;
18     if (sortData[left] > sortData[mid])
19        {
20         hold            = sortData[left];
21         sortData[left]  = sortData[mid];
22         sortData[mid]   = hold;
23        } // if
24     if (sortData[left] > sortData[right])
25        {
26         hold             = sortData[left];
27         sortData[left]   = sortData[right];
28         sortData[right]  = hold;
29        } // if
30     if (sortData[mid]  > sortData[right])
31        {
32         hold             = sortData[mid];
33         sortData[mid]    = sortData[right];
34         sortData[right]  = hold;
35        } // if
36     // Median is in middle. Exchange with left.
37     hold            = sortData[left];
38     sortData[left]  = sortData[mid];
39     sortData[mid]   = hold;
40
41     return;
42  } // medianLeft
```

Program 11-8 Median left for quick sort *(continued)*

11-6 EXTERNAL SORTS

All of the algorithms we have studied so far have been internal sorts—that is, sorts that require the data to be entirely sorted in primary memory during the sorting process. We now turn our attention to external sorting, sorts that allow portions of the data to be stored in secondary memory during the sorting process.

The term *external sorting* is somewhat of a misnomer. Most of the work spent ordering large files is not sorting but actually merging. To understand external sorting, therefore, we must first understand the merge concept.

Merging Ordered Files

A **merge** is the process that combines two files ordered on a given key into one ordered file on the same given key. A simple example is shown in Figure 11-17.

File 1 and file 2 are to be merged into file 3. To merge the files, we compare the first record in file 1 with the first record in file 2 and write the smaller one, 1, to file 3. We then compare the second record in file 1 with the first record in file 2 and write the smaller one, 2, to file 3. The process continues until all data have been merged in order into file 3. The logic to merge these two files, which is relatively simple, is shown in Algorithm 11-9.

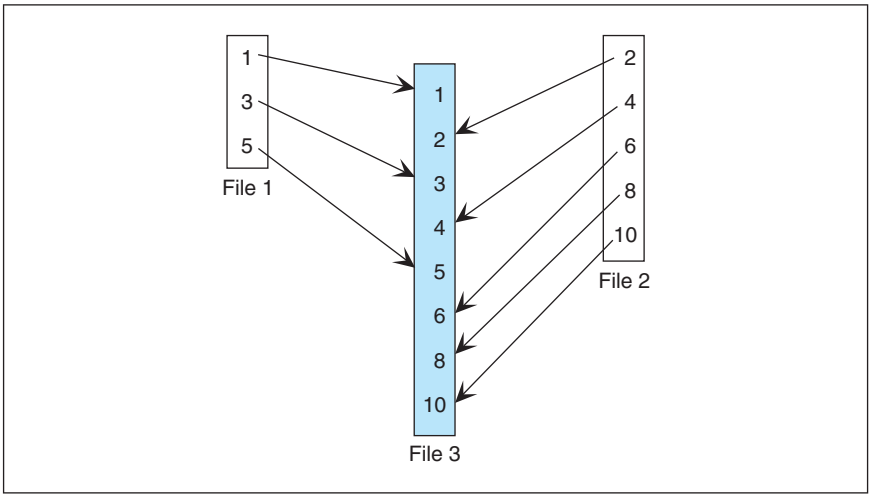

Figure 11-17 A simple merge

```
algorithm mergeFiles
Merge two ordered files into one file.
    Pre    input files are ordered
    Post   input files sequentially combined in output file
 1  open files
 2  read (file1 into record1)
 3  read (file2 into record2)
 4  loop (not end file1 OR not end file2)
```

Algorithm 11-9 Merge files

```
    1   if (record1.key <= record2.key)
        1   write  (file3 from record1)
        2   read   (file1 into record1)
        3   if (end of file1)
            1   record1.key = +∞
        4   end if
    2   else
        1   write  (file3 from record2)
        2   read   (file2 into record2)
        3   if (end of file2)
            1   record2.key = +∞
        4   end if
    3   end if
  5 end loop
  6 close files
end mergeFiles
```

Algorithm 11-9 Merge files *(continued)*

Algorithm 11-9 Analysis

Although merge files is a relatively simple algorithm, one point is worth discussing. When one file reaches the end, there may be more data in the second file. We therefore need to keep processing until both files are at the end. We could write separate blocks of code to handle the end-of-file processing for each file, but there is a simpler method. When one file hits its end, we simply set its key value to an artificially high value. In the algorithm this value is identified as +∞ (see Statements 4.1.3.1 and 4.2.3.1). Then, when we compare the two keys (Statement 4.1), the file at the end will be forced high and the other file will be processed. The high value is often called a **sentinel.** The only limitation to this logic is that the sentinel value cannot be a valid data value.

Merging Unordered Files

In merge sorting, however, we usually have a different situation than that shown above: The input files are not completely sorted. The data will run in sequence, and then there will be a sequence break followed by another series of data in sequence. This situation is demonstrated in Figure 11-18.

The series of consecutively ordered data in a file is known as a **merge run.** In Figure 11-18, all three files have three merge runs. To make them easy to see, we have spaced and colored the merge runs in each file. A **stepdown** occurs when the sequential ordering of the records in a merge file is broken. The end of each merge run is identified by a stepdown. For example, in Figure 11-18, there is a stepdown in file 2 when key 16 in the first merge run is followed by key 9 in the

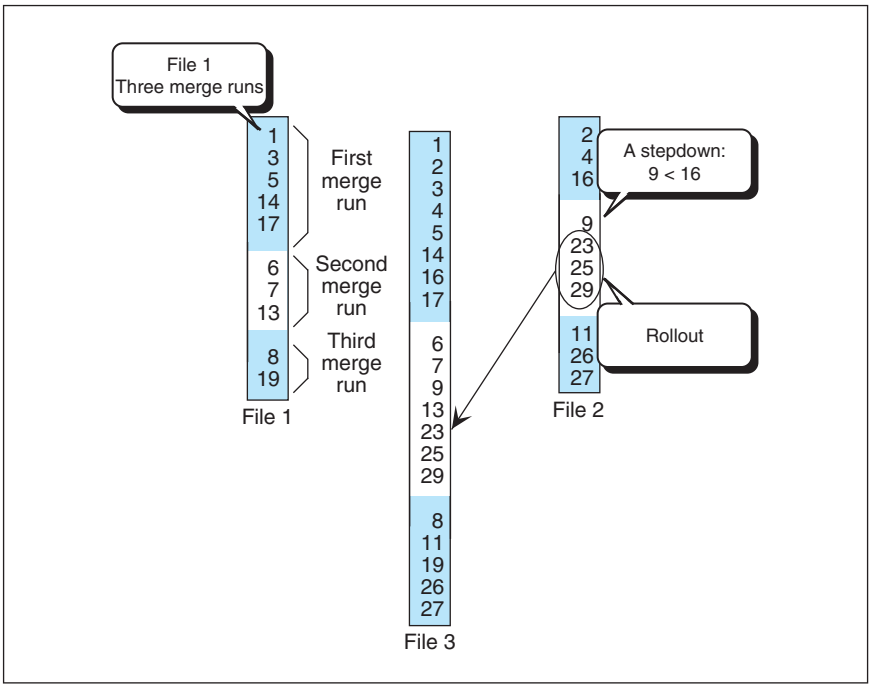

Figure 11-18 Merging files example

second merge run. An end of file is also considered a stepdown. Finally, in the merging of a file, the process of copying a consecutive series of records to the merge output file after a stepdown in the alternate merge input is known as a **rollout**. In Figure 11-18, after record 13 has been copied to the output file and a stepdown occurs, the remaining records (23, 25, and 29) in the second merge run in file 2 are rolled out to the merge output. Now let's look closely at the merge process in Figure 11-18. When merging files that are not completely ordered, we merge the corresponding merge runs from each file into a merge run in the output file. Thus, we see that merge run 1 in file 1 merges with merge run 1 in file 2 to produce merge run 1 in file 3. Similarly, merge runs 2 and 3 in the input files merge to produce merge runs 2 and 3 in the output.

When a stepdown is detected in an input merge file, the merge run in the alternate file must be rolled out to synchronize the two files. Thus, in merge run 2, when the stepdown between record 13 and record 8 is detected, we must roll out the remaining three records in file 2 so that we can begin merging the third merge runs.

Finally, it is important to note that the merge output is not a completely ordered file. In this particular case, two more merge runs are required. To see the complete process, we turn to a higher-level example.

The Sorting Process

Assume that a file of 2300 records needs to be sorted. In an external sort, we begin by sorting as many records as possible and creating two or more merge files. Assuming that the record size and the memory available for our sort program allow a maximum sort array size of 500 records, we begin by reading and sorting the first 500 records and writing them to a merge output file. As we sort the first 500 records, we keep the remaining 1800 records in secondary storage. After writing out the first merge run, we read the second 500 records, sort them, and write them to an alternate merge output file. We continue the sort process, writing 500 sorted records (records 1001 to 1500) to the first merge output file and another 500 sorted records (records 1501 to 2000) to the second merge output file. Finally, we sort the last 300 records and write them to the first output merge file. At this point we have created the situation we see in Figure 11-19. This first processing of the data into merge runs is known as the **sort phase**.

After completing the sort phase of an external sort, we proceed with the merge phase. Each complete reading and merging of the input merge files to one or more output merge files is considered a separate **merge phase.** Depending on how many merge runs are created, there will be zero or more merge phases. If all of the data fit into memory at one time, or if the file was sorted to begin with, then there will be only one merge run and the data on the first merge output file are completely sorted. This situation is the exception, however; several merge phases are generally required.

Computer scientists have developed many different merge concepts over the years. We present three that are representative: natural merge, balanced merge, and polyphase merge.

Natural Merge

A **natural merge** sorts a constant number of input merge files to one merge output file. Between each merge phase, a **distribution phase** is

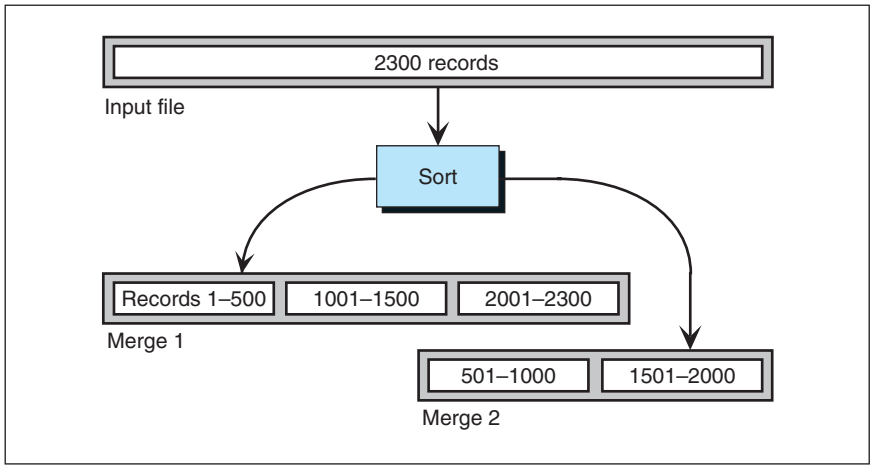

Figure 11-19 Sort phase in an external sort

required to redistribute the merge runs to the input files for remerging. Figure 11-20 is a diagram of our 2300 records as they would be sorted using a natural two-way merge—that is, a natural merge with two input merge files and one output merge file.

Note

> **In the natural merge, each phase merges a constant number of input files into one output file.**

In the natural merge, all merge runs are written to one file. Therefore, unless the file is completely ordered, the merge runs must be distributed to two merge files between each merge phase. This processing is very inefficient, especially because reading and writing records are among the slowest of all data processing. The question, therefore, is how can we make the merge process more efficient. The answer is found in the balanced merge.

Balanced Merge

A **balanced merge** uses a constant number of input merge files and the same number of output merge files. Any number of merge files can be used, although more than four is uncommon. Because multiple merge files are created in each merge phase, no distribution phase is required. Figure 11-21 sorts our 2300 records using a balanced two-way merge.

Note

> **The balanced merge eliminates the distribution phase by using the same number of input and output merge files.**

Four merge files are required in the balanced two-way merge. The first merge phase merges the first merge run on file 1 with the first merge run on file 2 and writes it to file 3. It then merges the second merge run on file 1 with the second merge run on file 2 and writes it to file 4. At this point, all of the merge runs on file 2 have been processed, so we roll out the remaining merge run on file 1 to merge file 3. This rollout of 300 records is wasted effort. We would like to be able to eliminate this step to make the merge processing as efficient as possible. We can if we use the polyphase merge.

Polyphase Merge

In the **polyphase merge**, a constant number of input merge files are merged to one output merge file and input merge files are immediately reused when their input has been completely merged. Polyphase merge is the most complex of the merge sorts we have discussed.

Note

> **In the polyphase merge, a constant number of input files are merged to one output file. As the data in each input file are completely merged, it immediately becomes the output file and what was the output file becomes an input file.**

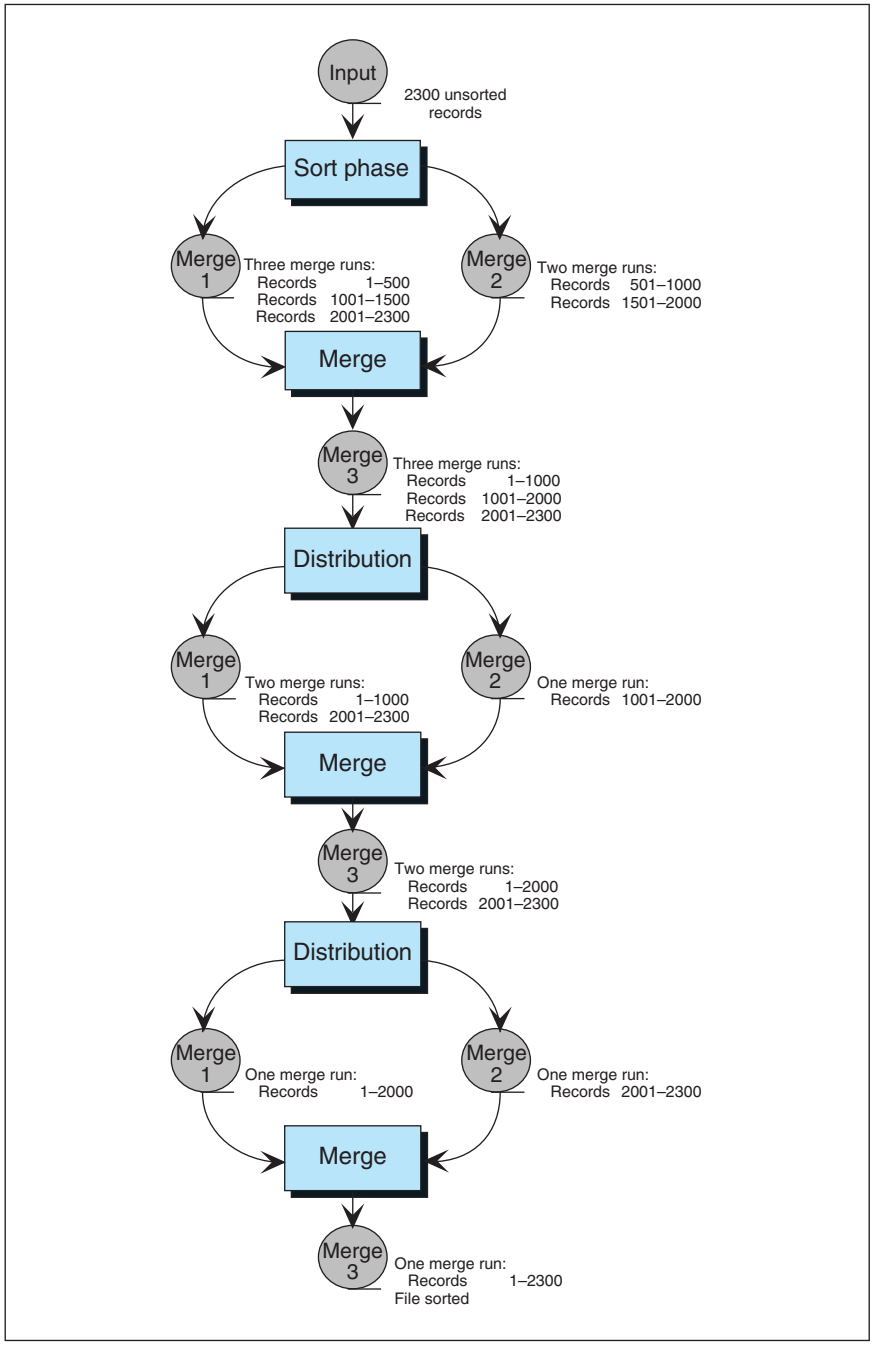

Figure 11-20 A natural two-way merge sort

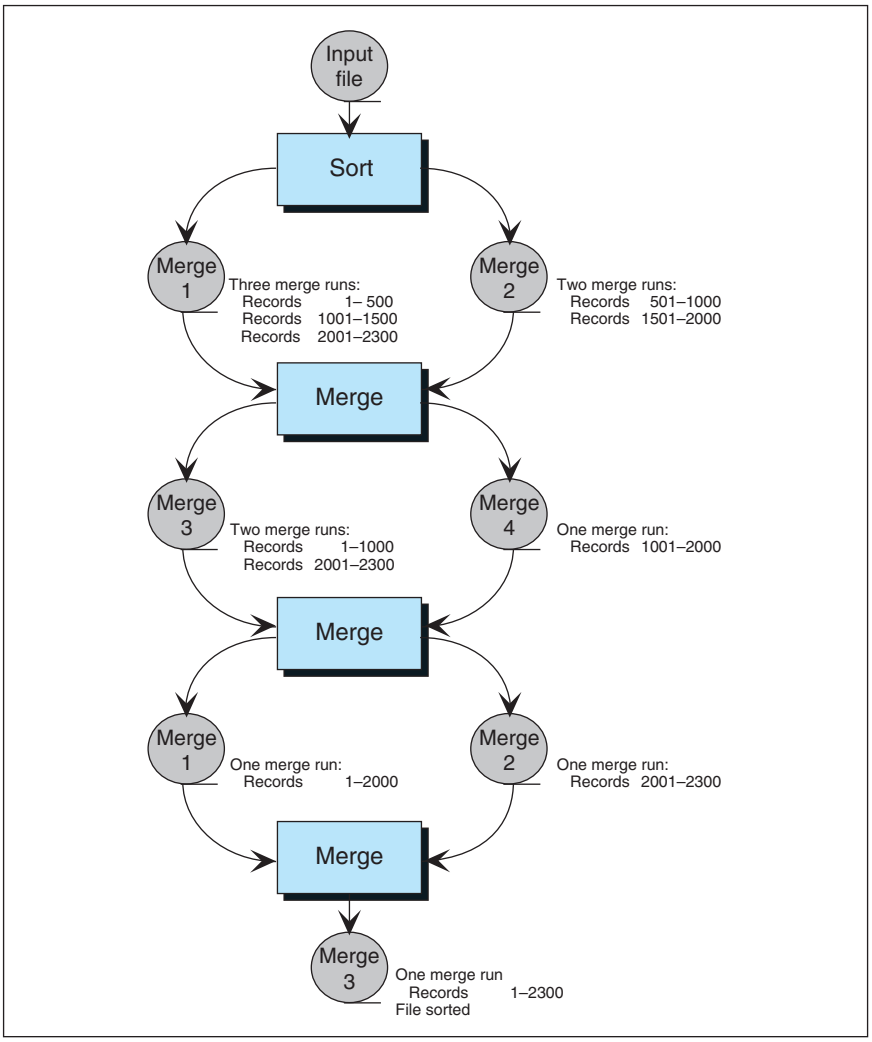

Figure 11-21 A balanced two-way merge sort

To demonstrate the process, let's study Figure 11-22 carefully. The processing begins as it does for the natural two-way merge. The first merge run on file 1 is merged with the first merge run on file 2. Then the second merge run on file 1 is merged with the second merge run on file 2. At this point, merge 2 is empty and the first merge phase is complete. We therefore close merge 2 and open it as output and close merge 3 and open it as input. The third merge run on file 1 is then merged with the first merge run on file 3, with the merged data being written to merge 2. Because merge 1 is empty, merge phase 2 is complete. We therefore close merge 1 and open it as output while closing

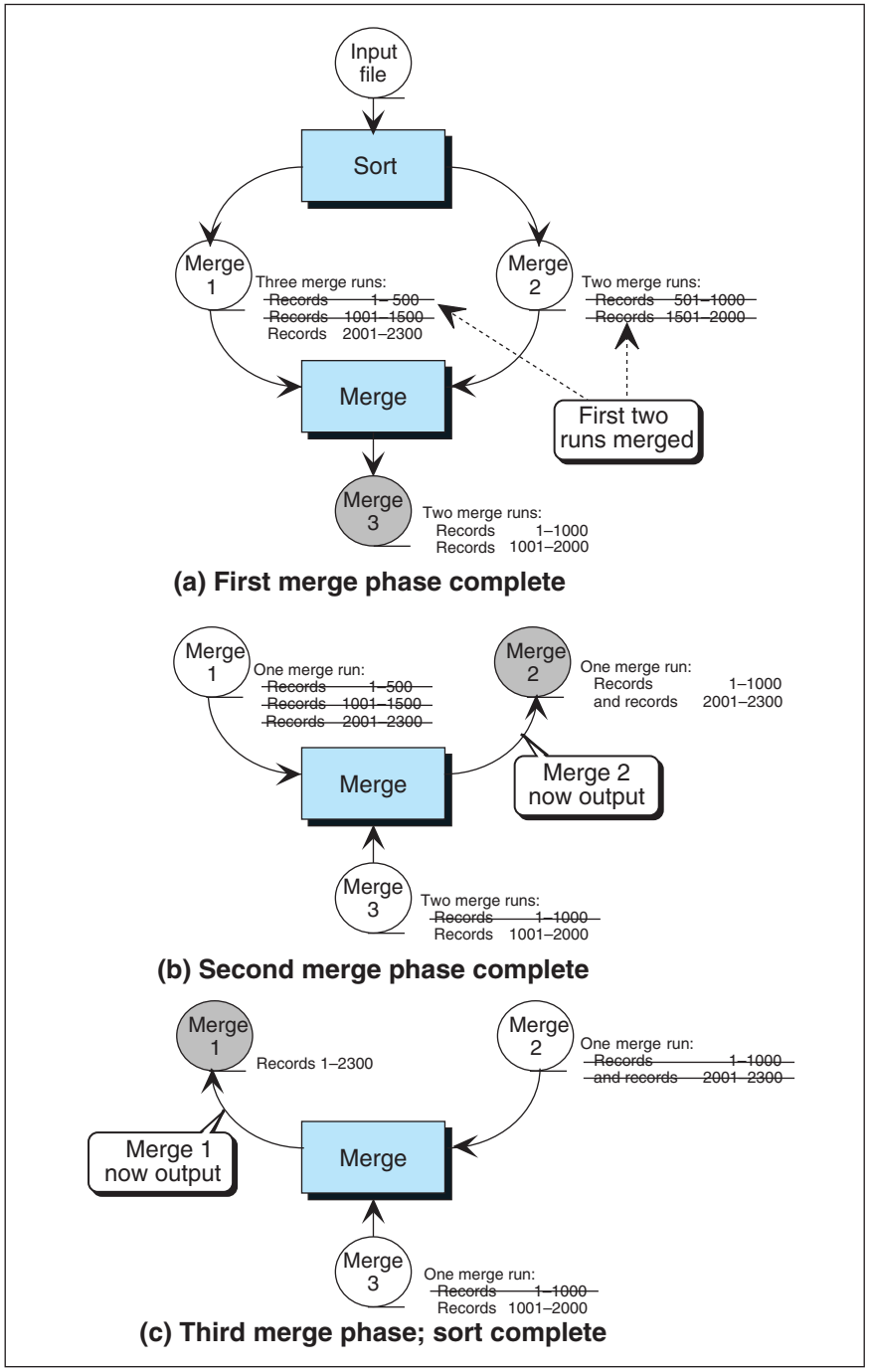

Figure 11-22 The polyphase merge sort

merge 2 and opening it as input. Because there is only one merge run on each of the input files, the sort is complete when these two merge runs have been merged to merge 1.

Sort Phase Revisited

With the polyphase sort, we have improved the merge phases as much as possible. We now return to the sort phase and try to improve it. Let's assume that we used the fastest possible sort in the sort phase. Even if we were able to double its internal sort speed, little would be gained. With today's modern computers operating in picosecond speeds and file processing operating in microsecond speeds, little is gained by improving the sort speed. In fact, the opposite is actually true. If we slow up the sort speed a little using a slower sort, we can actually improve overall speed. Let's see how this is done.

One class of sorts, tree sorts, allows us to start writing data to a merge file before the sort is complete. By using tree sorts, then, we can write longer merge runs, which reduces the number of merge runs and therefore speeds up the sorting process. Because we have studied one tree sort, the heap sort, let's see how it could be used to write longer merge runs.

Figure 11-23 shows how we can use the heap sort to write long merge runs. Given a list of 12 elements to be sorted and using a heap of 3 nodes, we sort the data into 2 merge runs. We begin by filling the sort array and then turning it into a **minimum** heap. After creating the heap, we write the smallest element, located at the root, to the first merge file and read the fourth element (97) into the root node. We again reheap and this time write 21 to the merge file. After reading 87 and rebuilding the heap, we write 78 to the merge file.

After reading 62, we have the heap shown in Figure 11-23(g). At this point, the data we just read (62) is smaller than the last element we wrote to the merge file (78). Thus it cannot be placed in the current merge run in merge 1. By definition, it belongs to the next merge run. Therefore, we exchange it with the last element in the heap (87) and subtract 1 from the heap size, making it two elements. We indicate that the third element, containing 62, is not currently in the heap by shading it. After we reheap, we write 87 to merge 1. We then read 94, reheap, and write it to merge 1. When we read 85, we discover that it is less than the largest key in merge 1 (94) so it must also be eliminated from the heap. After exchanging it with the last element in the heap, we have the situation shown in Figure 11-23(l).

After writing 97 to merge 1, we read 76. Because 76 is less than the last element written to the merge file, it also belongs to the next merge run. At this point, all elements in the array belong to the second merge run. The first merge run is therefore complete, as shown in Figure 11-23(m).

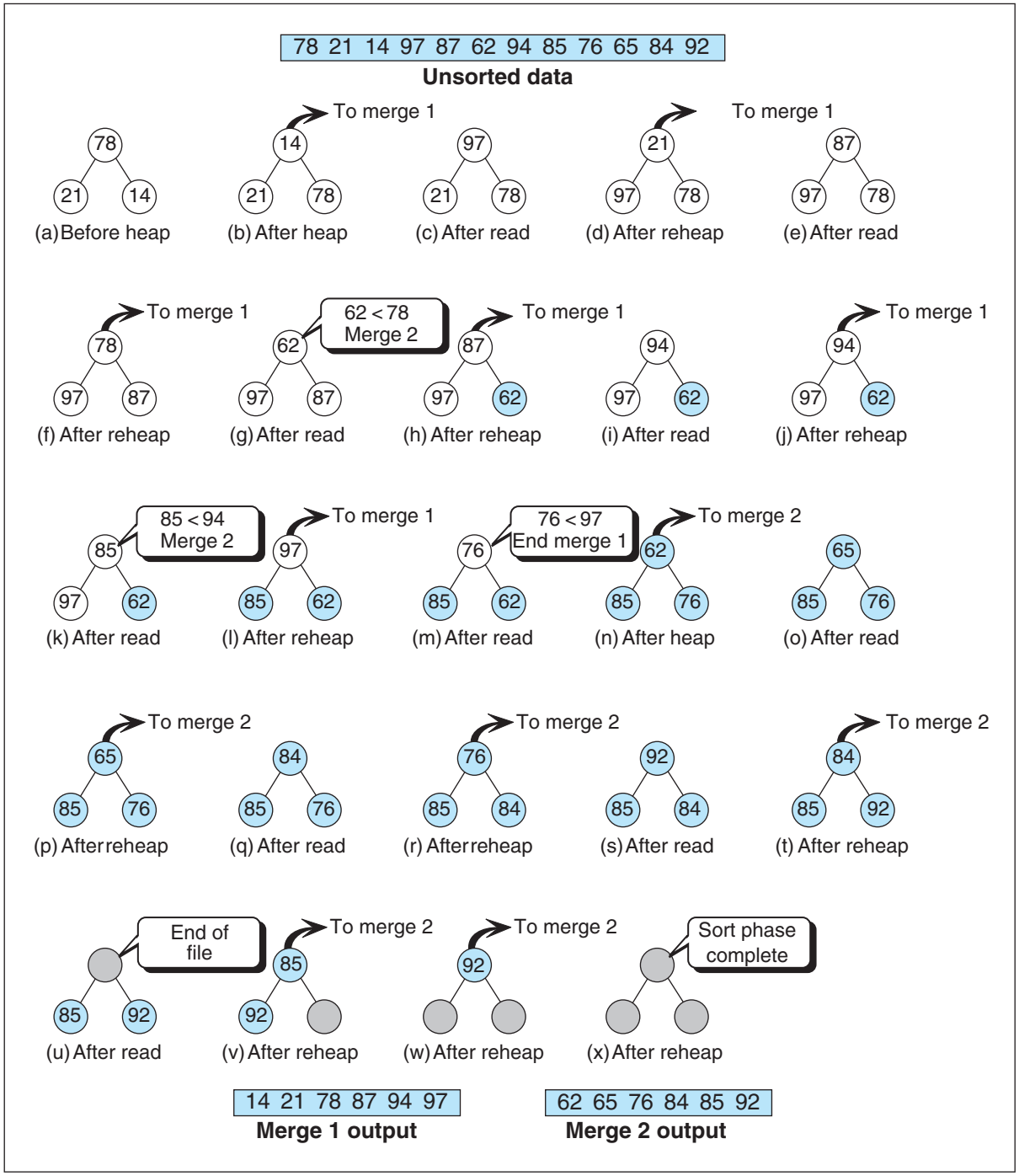

Figure 11-23 External sort phase using heap sort

After rebuilding the heap, we write the first element to merge file 2. We continue reading, reheaping, and writing until we write 84 to the merge file. At this point, we have read all of the data to be sorted and the input file is at the end of the file, as shown in Figure 11-23(u). We therefore move the last element to the heap, subtract 1 from the heap size, and reheap. After writing 85 to the merge run, we move the last element in the heap to the root, reheap, and write it to the merge file. Because the heap is now empty, the sort is complete.

If you examine the merge runs created in this example, you will note that they are both twice as long as the array size. As a rule of thumb, our heuristic studies have indicated that the average merge run size will be twice the size of the array being used for the sort. These longer merge runs will eliminate one merge pass, which is a significant time saver.

11-7 SUMMARY

- One of the most common applications in computer science is sorting.
- Sorts are generally classified as either internal or external.
  - In an internal sort, all of the data are held in primary storage during the sorting process.
  - An external sort uses primary storage for the data currently being sorted and secondary storage for any data that will not fit in primary memory.
- Data may be sorted in either ascending or descending order.
- Sort stability is an attribute of a sort indicating that data with equal keys maintain their relative input order in the output.
- Sort efficiency is a measure of the relative efficiency of a sort.
- Each traversal of the data during the sorting process is referred to as a pass.
- Internal sorting can be divided into three broad categories: insertion, selection, and exchange.
- Two methods of insertion sorting were discussed in this chapter: straight insertion sort and shell sort.
  - In the straight insertion sort, the list at any moment is divided into two sublists: sorted and unsorted. In each pass, the first element of the unsorted sublist is transferred to the sorted sublist by inserting it at the appropriate place.
  - The shell sort algorithm is an improved version of the straight insertion sort, in which the process uses different increments to sort the list. In each increment, the list is divided into segments, which are sorted independent of each other.
- Two methods of selection sorting were discussed in this chapter: straight selection sort and heap sort.
  - In the straight selection sort, the list at any moment is divided into two sublists: sorted and unsorted. In each pass, the process selects the smallest element from the unsorted sublist and exchanges it with the element at the beginning of the unsorted sublist.
  - The heap sort is an improved version of the straight selection sort. In the heap sort, the largest element in the heap is exchanged with the last element in the unsorted sublist. However, selecting the largest

element is much easier in this sort method because the largest element is the root of the heap.

- Two methods of exchange sorting were discussed in this chapter: bubble sort and quick sort.
  - In the bubble sort, the list at any moment is divided into two sublists: sorted and unsorted. In each pass, the smallest element is bubbled up from the unsorted sublist and moved to the sorted sublist.
  - The quick sort is the new version of the exchange sort in which the list is continuously divided into smaller sublists and exchanging takes place between elements that are out of order. Each pass of the quick sort selects a pivot and divides the list into three groups: a partition of elements whose key is less than the pivot's key, the pivot element that is placed in its ultimate correct position, and a partition of elements greater than or equal to the pivot's key. The sorting then continues by quick sorting the left partition followed by quick sorting the right partition.
- The efficiency of straight insertion, straight selection, and bubble sort is $O(n^2)$.
- The efficiency of shell sort is $O(n^{1.25})$, and the efficiency of heap and quick sorts is $O(n\log_2 n)$.
- External sorting allows a portion of the data to be stored in secondary storage during the sorting process.
- External sorting consists of two phases: the sort phase and the merge phase.
- The merge phase uses one of three methods: natural merge, balanced merge, and polyphase merge.
  - In natural merge, each phase merges a constant number of input files into one output file. The natural merge requires a distribution process between each merge phase.
  - In the balanced merge, we eliminate the distribute processes by using the same number of input and output files.
  - In polyphase merge, a number of input files are merged into one output file. However, the input file, which is exhausted first, is immediately used as an output file. The output file in the previous phase becomes one of the input files in the next phase.
- To improve the efficiency of sort phase in external sorting, we can use a tree sort, such as the minimum heap sort. This sort allows us to write to a merge file before the sorting process is complete, which results in longer merge runs.

11-8 PRACTICE SETS

Exercises

1. An array contains the elements shown below. The first two elements have been sorted using a straight insertion sort. What would be the value of the elements in the array after three more passes of the straight insertion sort algorithm?

```
3 13 7 26 44 23 98 57
```

2. An array contains the elements shown below. Show the contents of the array after it has gone through a one-increment pass of the shell sort. The increment factor is $k = 3$.

```
23 3 7 13 89 7 66 2 6 44 18 90 98 57
```

3. An array contains the elements shown below. The first two elements have been sorted using a straight selection sort. What would be the value of the elements in the array after three more passes of the selection sort algorithm?

```
7 8 26 44 13 23 98 57
```

4. An array contains the elements shown below. What would be the value of the elements in the array after three passes of the heap sort algorithm?

```
44 78 22 7 98 56 34 2 38 35 45
```

5. An array contains the elements shown below. The first two elements have been sorted using a bubble sort. What would be the value of the elements in the array after three more passes of the bubble sort algorithm? Use the version of bubble sort that starts from the end and bubbles up the smallest element.

```
7 8 26 44 13 23 57 98
```

6. An array contains the elements shown below. Using a quick sort show the contents of the array after the first pivot has been placed in its correct location. Identify the three sublists that exist at that point.

```
44 78 22 7 98 56 34 2 38 35 45
```

7. After two passes of a sorthing algorithm, the following array:

```
47 3 21 32 56 92
```

has been rearranged as shown below.

```
3 21 47 32 56 92
```

Which sorting algorithm is being used (straight selection, bubble, straight insertion)? Defend your answer.

8. After two passes of a sorting algorithm, the following array:

```
80 72 66 44 21 33
```

has been rearranged as shown below.

```
21 33 80 72 66 44
```

Which sorting algorithm is being used (straight selection, bubble, straight insertion)? Defend your answer.

9. After two passes of a sorting algorithm, the following array:

```
47 3 66 32 56 92
```

has been rearranged as shown below.

```
3 47 66 32 56 92
```

Which sorting algorithm is being used (straight selection, bubble, straight insertion)? Defend your answer.

10. Show the result after each merge phase when merging the following two files:

```
6 12 19 23 34 · 8 11 17 20 25 · 9 10 15 25 35
13 21 27 28 29 · 7 30 36 37 39
```

11. Starting with the following file, show the contents of all of the files created using external sorting and the natural merge method:

```
37 9 23 56 4 5 12 45 78 22 33 44 14 17 57 11 35 46 59
```

12. Rework Exercise 11 using the balanced merge method.
13. Rework Exercise 11 using the polyphase merge method.

Problems

14. Modify the insertion sort C++ code on page 515 to count the number of data moves needed to order an array of 1000 random numbers. A data move is movement of an element of data from one position in the array to another, to a hold area, or from a hold area back to the array. Display the array before and after the sort. At the end of the program, display the total moves needed to sort the array.
15. Repeat Problem 14 using the shell sort on page 516.
16. Repeat Problem 14 using the selection sort on page 524.
17. Repeat Problem 14 using the heap sort on page 525.
18. Repeat Problem 14 using the bubble sort on page 538.
19. Repeat Problem 14 using the quick sort on page 539.
20. Change the bubble sort algorithm on page 538 as follows: Use two-directional bubbling in each pass. In the first bubbling, the smallest element is bubbled up; in the second bubbling, the largest element is bubbled down. This sort is known as the **shaker sort.**
21. Write an algorithm that applies the incremental idea of the shell sort to selection sort. The algorithm first applies the straight section sort to items $n/2$ elements apart (first, middle, and last). It then applies it to $n/3$ elements apart, then to elements $n/4$ apart, and so on.
22. Write a recursive version of the selection sort algorithm on page 524.
23. Rewrite the insertion sort algorithm on page 515 using a singly linked list instead of an array.

Projects

24. Merge sorting is an example of a divide-and-conquer paradigm. In our discussions, we used merge only as an external sorting method. It can also be used for internal sorting. Let's see how it works. If we have a list of only two elements, we can simply divide it into two halves and then merge them. In other words, the merge sort is totally dependent on two processes, distribution and merge. This elementary process is shown in Figure 11-24.

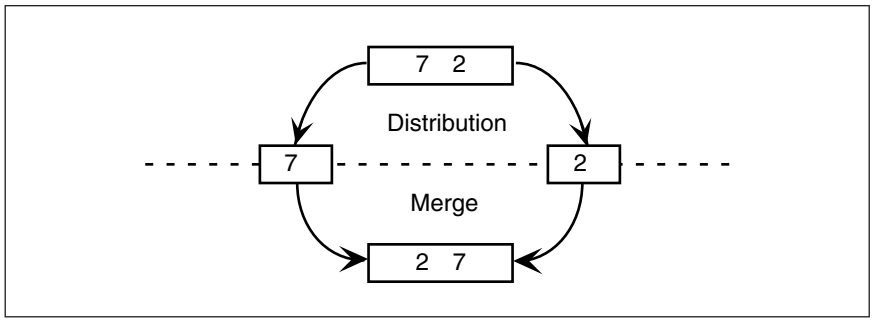

Figure 11-24 Split and merge for Project 24

Given a list longer than two elements, we can sort by repeating the distribution and merge processes. Because we don't know how many elements are in the input list when we begin, we can distribute the list originally by writing the first element to one output list, the second element to a second output list, and then continue writing alternatively to the first list and then the second list until the input list has been divided into two output lists. The output lists can then be sorted using a balanced two-way merge. This process is shown in Figure 11-25. It could be called the *sortless sort* because it sorts without ever using a sort phase.

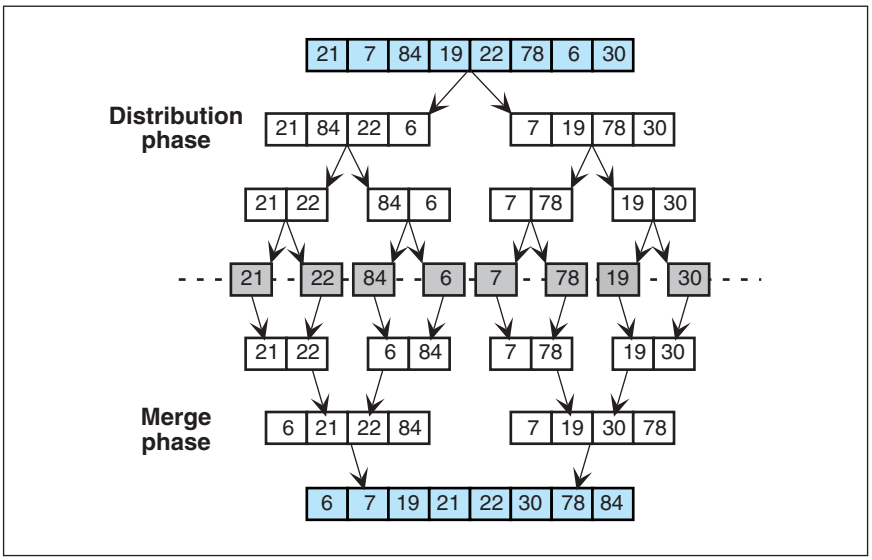

Figure 11-25 The sortless sort

Write a C++ program to sort an array of 500 random numbers using this approach. Print the data before and after the sort.

25. Write a program that sorts an array of random numbers using the shell sort and the quick sort. Both sorts should use the same data. Each sort should be executed twice. For the first sort, fill the array with random numbers between 1 and 999. For the second sort, fill the array with a nearly ordered list. Construct your nearly ordered list by reversing elements 19 and 20 in the sorted random number list. For each sort, count the number of comparisons and moves necessary to order this list.

Run the program three times, once with an array of 100 items, once with an array of 500 items, and once with an array of 1000 items. For the first execution only (100 elements), print the unsorted data followed by the sort data in 10 × 10 matrixes (10 rows of 10 numbers each). For all runs, print the number of comparisons and the number of moves required to order the data.

To make sure your statistics are as accurate as possible, you must analyze each loop limit condition test and each selection statement in your sort algorithms. The following notes should help with this analysis:

a. All loops require a count increment in their body.

b. Pretest loops (*while* and *for*) also require a count increment either before (recommended) or after the loop to count the last test.

c. Remember that C++ uses the shortcut rule for evaluating Boolean and/or expressions. The best way to count them is with a comma expression, as shown below. Use similar code for the selection statements.

```
while ((count++, a) && (count++, b))
```

Analyze the heuristics you generated and write a short report (less than one page) concerning what you discovered about these two sorts. Include the data in Table 11-4, one for the random data and one for the nearly ordered data. Calculate the ratio to one decimal place.

	Shell	Quick	Ratio (Shell/Quick)
Compares			
100			
500			
1000			
Moves			
100			
500			
1000			

Table 11-4 Sorting statistics format for Project 25

26. Repeat Project 25 adding heap sort and using your computer's internal clock. For each sort algorithm, start the clock as the last statement before calling the sort and read the clock as the first statement after the sort. Write a short report comparing the run times with the suggested algorithmics in the text. If your results do not agree with the algorithmics, increase the array size in 1000-element increments to get a better picture.

27. Radix sorting—also known as digit, pocket, and bucket sorting—is a very efficient sort for large lists whose keys are relatively short. If fact, if we consider only its big-O notation, which is $O(n)$, it is one of the best. Radix sorts were used extensively in the punched-card era to sort cards on electronic accounting machines (EAMs).

In a radix sort, each pass through the list orders the data one digit at a time. The first pass orders the data on the units (least significant) digit. The second pass orders the data on the tens digit. The third pass orders the data on the hundreds digit, and so forth until the list is completely sorted by sorting the data on the most significant digit.

In EAM sorts, the punched cards were sorted into pockets. In today's systems, we would sort them into a linked list, with a separate linked list for each digit. Using a pocket or bucket approach, we sort eight four-digit numbers in Figure 11-26.[9]

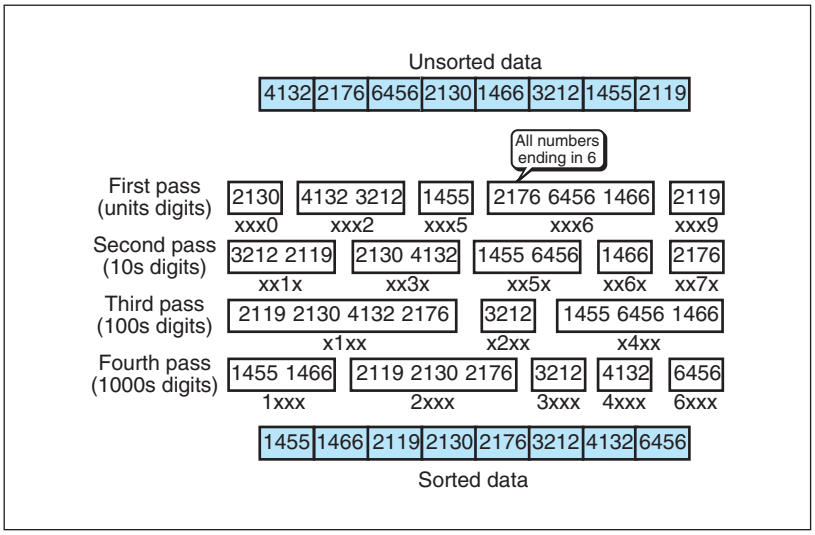

Figure 11-26 Radix sorting example for Project 27

If you analyze this sort, you will see that there are *k* sort passes, where *k* is the number of digits in the key. For each sort pass, we have *n* operations, where *n* is the number of elements to be sorted. This gives us an efficiency of *kn*, which in big-O notation is $O(n)$.

Write a program that uses the radix sort to sort 1000 random digits. Print the data before and after the sort. Each sort bucket should be a linked list. At the end of the sort, the data should be in the original array.

28. Modify Project 26 to include the radix sort. Build a table of sort times and write a short paper explaining the different sort timings and their relationship to their sort efficiency.

9. Note: For efficiency, radix sorts are not recommended for small lists or for keys containing a large number of digits.