Sorting

9

The efficiency of data handling can often be substantially increased if the data are sorted according to some criteria of order. For example, it would be practically impossible to find a name in the telephone directory if the names were not alphabetically ordered. The same can be said about dictionaries, book indexes, payrolls, bank accounts, student lists, and other alphabetically organized materials. The convenience of using sorted data is unquestionable and must be addressed in computer science as well. Although a computer can grapple with an unordered telephone book more easily and quickly than a human can, it is extremely inefficient to have the computer process such an unordered data set. It is often necessary to sort data before processing.

The first step is to choose the criteria that will be used to order data. This choice will vary from application to application and must be defined by the user. Very often, the sorting criteria are natural, as in the case of numbers. A set of numbers can be sorted in ascending or descending order. The set of five positive integers (5, 8, 1, 2, 20) can be sorted in ascending order resulting in the set (1, 2, 5, 8, 20) or in descending order resulting in the set (20, 8, 5, 2, 1). Names in the phone book are ordered alphabetically by last name, which is the natural order. For alphabetic and nonalphabetic characters, the American Standard Code for Information Interchange (ASCII) code is commonly used, although other choices such as Extended Binary Coded Decimal Interchange Code (EBCDIC) are possible. Once a criterion is selected, the second step is how to put a set of data in order using that criterion.

The final ordering of data can be obtained in a variety of ways, and only some of them can be considered meaningful and efficient. To decide which method is best, certain criteria of efficiency have to be established and a method for quantitatively comparing different algorithms must be chosen.

To make the comparison machine-independent, certain critical properties of sorting algorithms should be defined when comparing alternative methods. Two such properties are the number of comparisons and the number of data movements. The choice of these two properties should not be surprising. To sort a set of data, the data have to be compared and moved as necessary; the efficiency of these two operations depends on the size of the data set.

Because determining the precise number of comparisons is not always necessary or possible, an approximate value can be computed. For this reason, the number of comparisons and movements is approximated with big-O notation by giving the order of magnitude of these numbers. But the order of magnitude can vary depending on the initial ordering of data. How much time, for example, does the machine spend on data ordering if the data are already ordered? Does it recognize this initial ordering immediately or is it completely unaware of that fact? Hence, the efficiency measure also indicates the "intelligence" of the algorithm. For this reason, the number of comparisons and movements is computed (if possible) for the following three cases: best case (often, data already in order), worst case (it can be data in reverse order), and average case (data in random order). Some sorting methods perform the same operations regardless of the initial ordering of data. It is easy to measure the performance of such algorithms, but the performance itself is usually not very good. Many other methods are more flexible, and their performance measures for all three cases differ.

The number of comparisons and the number of movements do not have to coincide. An algorithm can be very efficient on the former and perform poorly on the latter, or vice versa. Therefore, practical reasons must aid in the choice of which algorithm to use. For example, if only simple keys are compared, such as integers or characters, then the comparisons are relatively fast and inexpensive. If strings or arrays of numbers are compared, then the cost of comparisons goes up substantially, and the weight of the comparison measure becomes more important. If, on the other hand, the data items moved are large, such as structures, then the movement measure may stand out as the determining factor in efficiency considerations. All theoretically established measures have to be used with discretion, and theoretical considerations should be balanced with practical applications. After all, the practical applications serve as a rubber stamp for theory decisions.

Sorting algorithms are of different levels of complexity. A simple method can be only 20 percent less efficient than a more elaborate one. If sorting is used in the program once in a while and only for small sets of data, then using a sophisticated and slightly more efficient algorithm may not be desirable; the same operation can be performed using a simpler method and simpler code. But if thousands of items are to be sorted, then a gain of 20 percent must not be neglected. Simple algorithms often perform better with a small amount of data than their more complex counterparts whose effectiveness may become obvious only when data samples become very large.

9.1 Elementary Sorting Algorithms

9.1.1 Insertion Sort

An *insertion sort* starts by considering the two first elements of the array `data`, which are `data[0]` and `data[1]`. If they are out of order, an interchange takes place. Then, the third element, `data[2]`, is considered. If `data[2]` is less than `data[0]` and `data[1]`, these two elements are shifted by one position; `data[0]` is placed at position 1, `data[1]` at position 2, and `data[2]` at position 0. If `data[2]` is less

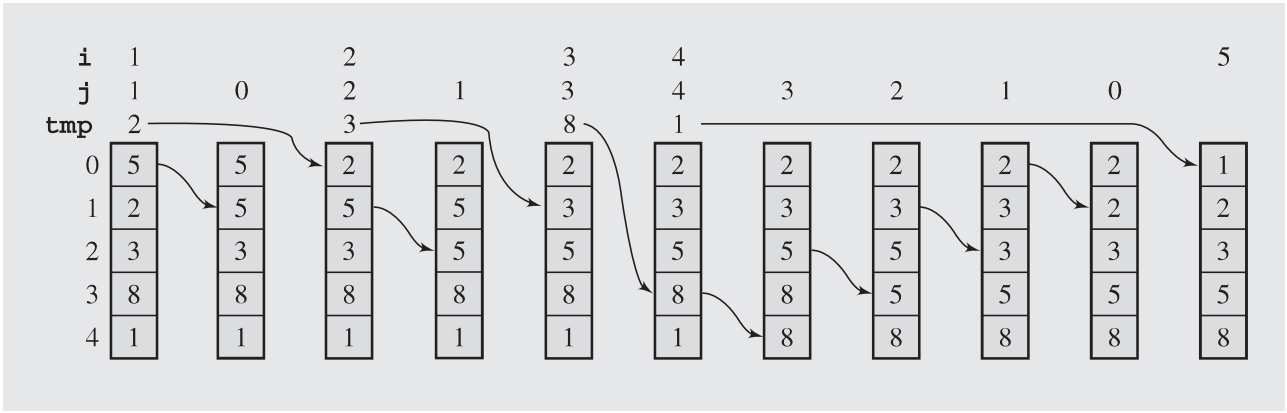
than `data[1]` and not less than `data[0]`, then only `data[1]` is moved to position 2 and its place is taken by `data[2]`. If, finally, `data[2]` is not less than both its predecessors, it stays in its current position. Each element `data[i]` is inserted into its proper location j such that $0 \leq j \leq i$, and all elements greater than `data[i]` are moved by one position.

An outline of the insertion sort algorithm is as follows:

```
insertionsort(data[],n)
   for i = 1 to n-1
      move all elements data[j] greater than data[i] by one position;
      place data[i] in its proper position;
```

Note that sorting is restricted only to a fraction of the array in each iteration, and only in the last pass is the whole array considered. Figure 9.1 shows what changes are made to the array [5 2 3 8 1] when `insertionsort()` executes.

FIGURE 9.1 The array [5 2 3 8 1] sorted by insertion sort.



Because an array having only one element is already ordered, the algorithm starts sorting from the second position, position 1. Then for each element `tmp = data[i]`, all elements greater than `tmp` are copied to the next position, and `tmp` is put in its proper place.

An implementation of insertion sort is:

```
template<class T>
void insertionsort(T data[], int n) {
    for (int i = 1,j; i < n; i++) {
        T tmp = data[i];
        for (j = i; j > 0 && tmp < data[j-1]; j--)
            data[j] = data[j-1];
        data[j] = tmp;
    }
}
```

An advantage of using insertion sort is that it sorts the array only when it is really necessary. If the array is already in order, no substantial moves are performed; only the variable `tmp` is initialized, and the value stored in it is moved back to the same position. The algorithm recognizes that part of the array is already sorted and stops execution accordingly. But it recognizes only this, and the fact that elements may already be in their proper positions is overlooked. Therefore, they can be moved from these positions and then later moved back. This happens to numbers 2 and 3 in the example in Figure 9.1. Another disadvantage is that if an item is being inserted, all elements greater than the one being inserted have to be moved. Insertion is not localized and may require moving a significant number of elements. Considering that an element can be moved from its final position only to be placed there again later, the number of redundant moves can slow down execution substantially.

To find the number of movements and comparisons performed by `insertionsort()`, observe first that the outer `for` loop always performs $n - 1$ iterations. However, the number of elements greater than `data[i]` to be moved by one position is not always the same.

The best case is when the data are already in order. Only one comparison is made for each position *i,* so there are $n - 1$ comparisons, which is $O(n)$, and $2(n - 1)$ moves, all of them redundant.

The worst case is when the data are in reverse order. In this case, for each *i,* the item `data[i]` is less than every item `data[0]`, . . . , `data[i-1]`, and each of them is moved by one position. For each iteration *i* of the outer `for` loop, there are *i* comparisons, and the total number of comparisons for all iterations of this loop is

$$\sum_{i=1}^{n-1} i = 1 + 2 + \cdots + (n - 1) = \frac{n(n - 1)}{2} = O(n^2)$$

The number of times the assignment in the inner `for` loop is executed can be computed using the same formula. The number of times `tmp` is loaded and unloaded in the outer `for` loop is added to that, resulting in the total number of moves:

$$\frac{n(n - 1)}{2} + 2(n - 1) = \frac{n^2 + 3n - 4}{2} = O(n^2)$$

Only extreme cases have been taken into consideration. What happens if the data are in random order? Is the sorting time closer to the time of the best case, $O(n)$, or to the worst case, $O(n^2)$? Or is it somewhere in between? The answer is not immediately evident, and requires certain introductory computations. The outer `for` loop always executes $n - 1$ times, but it is also necessary to determine the number of iterations for the inner loop.

For every iteration *i* of the outer `for` loop, the number of comparisons depends on how far away the item `data[i]` is from its proper position in the currently sorted subarray `data[0 . . . i-1]`. If it is already in this position, only one test is performed that compares `data[i]` and `data[i-1]`. If it is one position away from its proper place, two comparisons are performed: `data[i]` is compared with `data[i-1]` and then with `data[i-2]`. Generally, if it is *j* positions away from its proper location, `data[i]` is compared with $j + 1$ other elements. This means that, in iteration *i* of the outer `for` loop, there are either 1, 2, . . . , or *i* comparisons.

Under the assumption of equal probability of occupying array cells, the average number of comparisons of `data[i]` with other elements during the iteration i of the outer `for` loop can be computed by adding all the possible numbers of times such tests are performed and dividing the sum by the number of such possibilities. The result is

$$\frac{1 + 2 + \cdots + i}{i} = \frac{\frac{1}{2}i(i+1)}{i} = \frac{i+1}{2}$$

To obtain the average number of all comparisons, the computed figure has to be added for all is (for all iterations of the outer `for` loop) from 1 to $n - 1$. The result is

$$\sum_{i=1}^{n-1} \frac{i+1}{2} = \frac{1}{2}\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} \frac{1}{2} = \frac{\frac{1}{2}n(n-1)}{2} + \frac{1}{2}(n-1) = \frac{n^2 + n - 2}{4}$$

which is $O(n^2)$ and approximately one-half of the number of comparisons in the worst case.

By similar reasoning, we can establish that, in iteration i of the outer `for` loop, `data[i]` can be moved either 0, 1, . . . , or i times; that is

$$\frac{0 + 1 + \cdots + i}{i} = \frac{\frac{1}{2}i(i+1)}{i+1} = \frac{i}{2}$$

times plus two unconditional movements (to `tmp` and from `tmp`). Hence, in all the iterations of the outer `for` loop we have, on the average,

$$\sum_{i=1}^{n-1}\left(\frac{i}{2} + 2\right) = \frac{1}{2}\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 2 = \frac{\frac{1}{2}n(n-1)}{2} + 2(n-1) = \frac{n^2 + 7n - 8}{4}$$

movements, which is also $O(n^2)$.

This answers the question: is the number of movements and comparisons for a randomly ordered array closer to the best or to the worst case? Unfortunately, it is closer to the latter, which means that, on the average, when the size of an array is doubled, the sorting effort quadruples.

9.1.2 Selection Sort

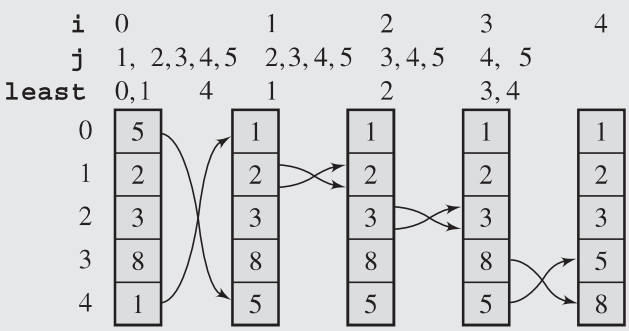
Selection sort is an attempt to localize the exchanges of array elements by finding a misplaced element first and putting it in its final place. The element with the lowest value is selected and exchanged with the element in the first position. Then, the smallest value among the remaining elements `data[1],...,data[n-1]` is found and put in the second position. This selection and placement by finding, in each pass `i`, the lowest value among the elements `data[i],...,data[n-1]` and swapping it with `data[i]` are continued until all elements are in their proper positions. The following pseudocode reflects the simplicity of the algorithm:

```
selectionsort(data[],n)
   for i = 0 to n-2
      select the smallest element among data[i],...,data[n-1];
      swap it with data[i];
```

It is rather obvious that `n-2` should be the last value for `i`, because if all elements but the last have been already considered and placed in their proper positions, then the *n*th element (occupying position `n-1`) has to be the largest. An example is shown in Figure 9.2. Here is a C++ implementation of selection sort:

```
template<class T>
void selectionsort(T data[], int n) {
    for (int i = 0,j,least; i < n-1; i++) {
        for (j = i+1, least = i; j < n; j++)
            if (data[j] < data[least])
                least = j;
        swap(data[least],data[i]);
    }
}
```

FIGURE 9.2 The array [5 2 3 8 1] sorted by selection sort.



where the function `swap()` exchanges elements `data[least]` and `data[i]` (see the end of Section 1.2). Note that `least` is not the smallest element but its position.

The analysis of the performance of the function `selectionsort()` is simplified by the presence of two `for` loops with lower and upper bounds. The outer loop executes `n` − 1 times, and for each `i` between 0 and `n` − 2, the inner loop iterates `j` = (`n` − 1) − `i` times. Because comparisons of keys are done in the inner loop, there are

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1) + \cdots + 1 = \frac{n(n-1)}{2} = O(n^2)$$

comparisons. This number stays the same for all cases. There can be some savings only in the number of swaps. Note that if the assignment in the `if` statement is executed, only the index `j` is moved, not the item located currently at position `j`. Array elements are swapped unconditionally in the outer loop as many times as this loop executes, which is `n-1`. Thus, in all cases, items are moved the same number of times, 3(`n`−1).

The best thing about this sort is the required number of assignments, which can hardly be beaten by any other algorithm. However, it might seem somewhat

unsatisfactory that the total number of exchanges, 3(n−1), is the same for all cases. Obviously, no exchange is needed if an item is in its final position. The algorithm disregards that and swaps such an item with itself, making three redundant moves. The problem can be alleviated by making `swap()` a conditional operation. The condition preceding the `swap()` should indicate that no item less than `data[least]` has been found among elements `data[i+1],...,data[n−1]`. The last line of `selectionsort()` might be replaced by the lines:

```
if (data[i] != data[least])
   swap (data[least], data[i]);
```

This increases the number of array element comparisons by n−1, but this increase can be avoided by noting that there is no need to compare items. We proceed as we did in the case of the `if` statement of `selectionsort()` by comparing the indexes and not the items. The last line of `selectionsort()` can be replaced by:

```
if (i != least)
   swap (data[least], data[i]);
```

Is such an improvement worth the price of introducing a new condition in the procedure and adding $n-1$ index comparisons as a consequence? It depends on what types of elements are being sorted. If the elements are numbers or characters, then interposing a new condition to avoid execution of redundant swaps gains little in efficiency. But if the elements in `data` are large compound entities such as arrays or structures, then one swap (which requires three assignments) may take the same amount of time as, say, 100 index comparisons, and using a conditional `swap()` is recommended.

9.1.3 Bubble Sort

A bubble sort can be best understood if the array to be sorted is envisaged as a vertical column whose smallest elements are at the top and whose largest elements are at the bottom. The array is scanned from the bottom up, and two adjacent elements are interchanged if they are found to be out of order with respect to each other. First, items `data[n-1]` and `data[n-2]` are compared and swapped if they are out of order. Next, `data[n-2]` and `data[n-3]` are compared, and their order is changed if necessary, and so on up to `data[1]` and `data[0]`. In this way, the smallest element is bubbled up to the top of the array.

However, this is only the first pass through the array. The array is scanned again comparing consecutive items and interchanging them when needed, but this time, the last comparison is done for `data[2]` and `data[1]` because the smallest element is already in its proper position, namely, position 0. The second pass bubbles the second smallest element of the array up to the second position, position 1. The procedure continues until the last pass when only one comparison, `data[n-1]` with `data[n-2]`, and possibly one interchange are performed.

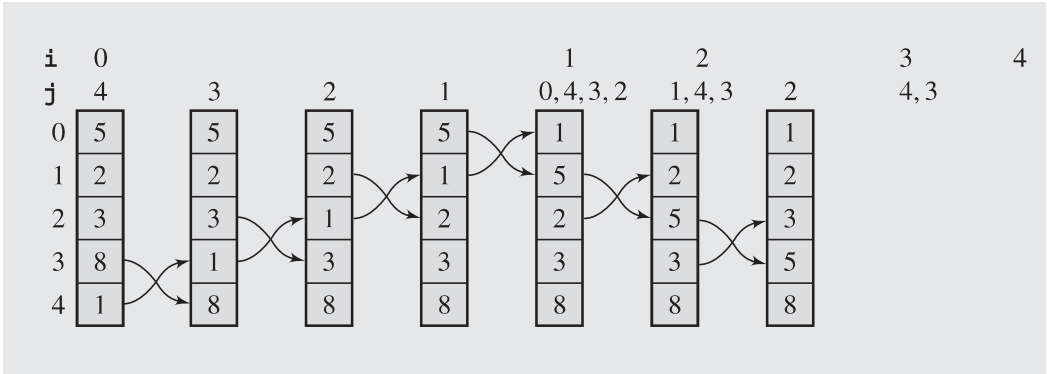
A pseudocode of the algorithm is as follows:

```
bubblesort(data[],n)
   for i = 0 to n−2
        for j = n-1 down to i+1
             swap elements in positions  j  and j-1  if they are out of order;
```

Figure 9.3 illustrates the changes performed in the integer array [5 2 3 8 1] during the execution of `bubblesort()`. Here is an implementation of bubble sort:

```
template<class T>
void bubblesort(T data[], int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = n-1; j > i; --j)
            if (data[j] < data[j-1])
                swap(data[j],data[j-1]);
}
```

FIGURE 9.3 The array [5 2 3 8 1] sorted by bubble sort.



The number of comparisons is the same in each case (best, average, and worst) and equals the total number of iterations of the inner `for` loop

$$\sum_{i=0}^{n-2}(n-1-i) = \frac{n(n-1)}{2} = O(n^2)$$

comparisons. This formula also computes the number of swaps in the worst case when the array is in reverse order. In this case, $3\frac{n(n-1)}{2}$ moves have to be made.

The best case, when all elements are already ordered, requires no swaps. To find the number of moves in the average case, note that if an i-cell array is in random order, then the number of swaps can be any number between zero and $i-1$; that is, there can be either no swap at all (all items are in ascending order), one swap, two swaps, . . . , or $i-1$ swaps. The array processed by the inner `for` loop is `data[i]`, . . . , `data[n-1]`, and the number of swaps in this subarray—if its elements are randomly ordered—is either zero, one, two, . . . , or $n-1-i$. After averaging the sum of all these possible numbers of swaps by the number of these possibilities, the average number of swaps is obtained, which is

$$\frac{0+1+2+\cdots+(n-1-i)}{n-i} = \frac{n-i-1}{2}$$

If all these averages for all the subarrays processed by `bubblesort()` are added (that is, if such figures are summed over all iterations i of the outer `for` loop), the result is

$$\sum_{i=0}^{n-2} \frac{n-i-1}{2} = \frac{1}{2}\sum_{i=0}^{n-2}(n-1) - \frac{1}{2}\sum_{i=0}^{n-2} i$$

$$= \frac{(n-1)^2}{2} - \frac{(n-1)(n-2)}{4} = \frac{n(n-1)}{4}$$

swaps, which is equal to $\frac{3}{4}n(n-1)$ moves.

The main disadvantage of bubble sort is that it painstakingly bubbles items step by step up toward the top of the array. It looks at two adjacent array elements at a time and swaps them if they are not in order. If an element has to be moved from the bottom to the top, it is exchanged with every element in the array. It does not skip them as selection sort did. In addition, the algorithm concentrates only on the item that is being bubbled up. Therefore, all elements that distort the order are moved, even those that are already in their final positions (see numbers 2 and 3 in Figure 9.3, the situation analogous to that in insertion sort).

What is bubble sort's performance in comparison to insertion and selection sort? In the average case, bubble sort makes approximately twice as many comparisons and the same number of moves as insertion sort, as many comparisons as selection sort, and n times more moves than selection sort.

It could be said that insertion sort is twice as fast as bubble sort. In fact it is, but this fact does not immediately follow from the performance estimates. The point is that when determining a formula for the number of comparisons, only comparisons of data items have been included. The actual implementation for each algorithm involves more than just that. In `bubblesort()`, for example, there are two loops, both of which compare indexes: `i` and `n-1` in the first loop, `j` and `i` in the second. All in all, there are $\frac{n(n-1)}{2}$ such comparisons, and this number should not be treated too lightly. It becomes negligible if the data items are large structures. But if `data` consists of integers, then comparing the data takes a similar amount of time as comparing indexes. A more thorough treatment of the problem of efficiency should focus on more than just data comparison and exchange. It should also include the overhead necessary for implementation of the algorithm.

An apparent improvement of bubble sort is obtained by adding a flag to discontinue processing after a pass in which no swap was performed:

```
template<class T>
void bubblesort2(T data[], const int n) {
     bool again = true;
     for (int i = 0; i < n-1 && again; i++)
          for (int j = n-1, again = false; j > i; --j)
               if (data[j] < data[j-1]) {
                    swap(data[j],data[j-1]);
                    again = true;
               }
}
```

The improvement, however, is insignificant because in the worst case the improved bubble sort behaves just as the original one. The worst case for the number

of comparisons is when the largest element is at the very beginning of data before sorting starts because this element can be moved only by one position in each pass. There are $(n - 1)!$ such worst cases in an array in which all elements are different. The cases, when the second largest element is at the beginning or the largest element is in the second position, and there are also $(n - 1)!$ such cases, are just as bad (only one fewer pass would be needed than in the worst case). The cases when the third largest element is in the first position are not far behind, etc. Therefore, very seldom the flag `again` fulfills its duty and very often – because an additional variable has to be maintained by `bubblesort2()` – the improved version is even slower than `bubblesort()`. Therefore, by itself, `bubblesort2()` is not an interesting modification of bubble sort. But comb sort, which builds on `bubblesort2()`, certainly is.

9.1.4 Comb Sort

A significant improvement of bubble sort is obtained by preprocessing the data by comparing elements *step* positions away from one another, which is an idea behind the comb sort. In each pass, *step* becomes smaller until it is equal to 1 (Dobosiewicz 1980; Box and Lacey 1991). The idea is that large elements are moved toward the end of the array before proper sorting begins. Here is an implementation:

```
template<class T>
void combsort(T data[], const int n) {
    int step = n, j, k;
    while ((step = int(step/1.3)) > 1)        // phase 1
        for (j = n-1; j >= step; j--) {
            k = j-step;
            if (data[j] < data[k])
                swap(data[j],data[k]);
        }

    bool again = true;
    for (int i = 0; i < n-1 && again; i++)    // phase 2
        for (j = n-1, again = false; j > i; --j)
            if (data[j] < data[j-1]) {
                swap(data[j],data[j-1]);
                again = true;
             }

}
```

Here is an example of the execution of comb sort:

phase 1:

pass	step	data[]																		
		41	11	18	7	16	25	4	23	32	31	22	9	1	22	3	7	31	6	10
1	14	3	7	18	6	10	25	4	23	32	31	22	9	1	22	41	11	31	7	16
2	10	3	7	1	6	10	11	4	7	16	31	22	9	18	22	41	25	31	23	32
3	7	3	7	1	6	9	11	4	7	16	31	22	10	18	22	41	25	31	23	32

```
 4      5      3  4  1  6  9 11  7  7 16 31 22 10 18 22 41 25 31 23 32
 5      3      3  4  1  6  7 10  7  9 11 18 22 16 31 22 23 25 31 41 32
 6      2      1  4  3  6  7  9  7 10 11 16 22 18 23 22 31 25 31 41 32
phase 2:
 7             1  3  4  6  7  7  9 10 11 16 18 22 22 23 25 31 31 32 41
 8             1  3  4  6  7  7  9 10 11 16 18 22 22 23 25 31 31 32 41
```

The data are processed from right to left. In phase 1, in pass 1, comparisons are made between elements $\left\lfloor \frac{19}{1.3} \right\rfloor = 14$ positions away from one another: 16 and 10 (leading to a swap), 7 and 6 (another swap), 18 and 31, 11 and 7 (swap), and 41 and 3 (swap). In pass 2, the distance between elements being compared is $\left\lfloor \frac{14}{1.3} \right\rfloor = 10$: 32 and 16 (swap), 23 and 7 (swap), . . . , and 3 and 22. Phase 2, which is `bubblesort2()`, requires only a modest two passes.

One problem is with determining the distance between positions of elements to be compared. Numerous experimental runs indicate (Box and Lacey 1991) that, generally, the factor $s = 1.3$ should be used to determine distances $\left\lfloor \frac{n}{s} \right\rfloor, \left\lfloor \frac{\lfloor \frac{n}{s} \rfloor}{s} \right\rfloor, \ldots$ which can be approximated with the decrementing sequence $\frac{n}{s}, \frac{n}{s^2}, \ldots, \frac{n}{s^p}$. Because the last step in phase 1 is equal to 2, that is, $\frac{n}{s^p} = 2$, we find that $P = \frac{\lg \frac{n}{2}}{\lg s}$. Using these approximations, the number of steps in phase 1 is determined to be always equal to

$$\sum_{i=1}^{P} \left(n - \frac{n}{s^i} \right) = P \cdot n - \frac{2 - n}{1 - s} = O(n \lg n)$$

The worst case is $O(n^2)$ due to the second phase (Drozdek 2005).

Experimental runs indicate that the improvement is indeed dramatic and the impressive performance of comb sort is comparable to the performance of quicksort (Figure 9.19).

9.2 Decision Trees

The three sorting methods analyzed in previous sections were not very efficient. This leads to several questions: Can any better level of efficiency for a sorting algorithm be expected? Can algorithms, at least theoretically, be more efficient by executing faster? If so, when can we be satisfied with an algorithm and be sure that the sorting speed is unlikely to be increased? We need a quantitative measurement to estimate a *lower bound* of sorting speed.

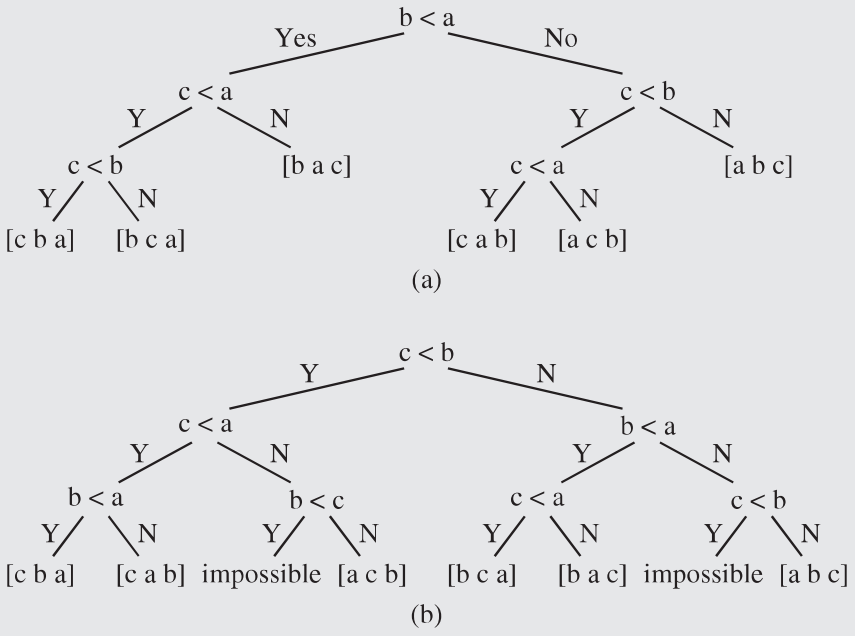
This section focuses on the comparisons of two elements and not the element interchange. The questions are: On the average, how many comparisons have to be made to sort n elements? Or what is the best estimate of the number of item comparisons if an array is assumed to be ordered randomly?

Every sorting algorithm can be expressed in terms of a binary tree in which the arcs carry the labels Y(es) or N(o). Nonterminal nodes of the tree contain conditions

or queries for labels, and the leaves have all possible orderings of the array to which the algorithm is applied. This type of tree is called a *decision tree*. Because the initial ordering cannot be predicted, all possibilities have to be listed in the tree in order for the sorting procedure to grapple with any array and any possible initial order of data. This initial order determines which path is taken by the algorithm and what sequence of comparisons is actually chosen. Note that different trees have to be drawn for arrays of different length.

Figure 9.4 illustrates decision trees for insertion sort and bubble sort for an array [a b c]. The tree for insertion sort has six leaves, and the tree for bubble sort has eight leaves. How many leaves does a tree for an n-element array have? Such an array can be ordered in $n!$ different ways, as many ways as the possible permutations of the

FIGURE 9.4 Decision trees for (a) insertion sort and (b) bubble sort as applied to the array [a b c].



array elements, and all of these orderings have to be stored in the leaves of the decision tree. Thus, the tree for insertion sort has six leaves because $n = 3$, and $3! = 6$.

But as the example of the decision tree for bubble sort indicates, the number of leaves does not have to equal $n!$. In fact, it is never less than $n!$, which means that it can be greater than $n!$. This is a consequence of the fact that a decision tree can have leaves corresponding to failures, not only to possible orderings. The failure nodes are reached by an inconsistent sequence of operations. Also, the total number of leaves can be greater than $n!$ because some orderings (permutations) can occur in more than one leaf, because the comparisons may be repeated.

One of the interesting properties of decision trees is the average number of arcs traversed from the root to reach a leaf. Because one arc represents one comparison, the average number of arcs reflects the average number of key comparisons when executing a sorting algorithm.

As already established in Chapter 6, an i-level complete decision tree has 2^{i-1} leaves, $2^{i-1}-1$ nonterminal nodes (for $i \geq 1$), and 2^i-1 total nodes. Because all noncomplete trees with the same number of i levels have fewer nodes than that, $k + m \leq 2^i-1$, where m is the number of leaves and k is the number of nonleaves. Also, $k \leq 2^{i-1}-1$ and $m \leq 2^{i-1}$ (Section 6.1 and Figure 6.5). The latter inequality is used as an approximation for m. Hence, in an i-level decision tree, there are at most 2^{i-1} leaves.

Now, a question arises: What is a relationship between the number of leaves of a decision tree and the number of all possible orderings of an n-element array? There are $n!$ possible orderings, and each one of them is represented by a leaf in a decision tree. But the tree also has some extra nodes due to repetitions and failures. Therefore, $n! \leq m \leq 2^{i-1}$, or $2^{i-1} \geq n!$. This inequality answers the following question: How many comparisons are performed when using a certain sorting algorithm for an n-element array in the worst case? Or rather, what is the lowest or the best figure expected in the worst case? Note that this analysis pertains to the worst case. We assume that i is a level of a tree regardless of whether or not it is complete; i always refers to the longest path leading from the root of the tree to the lowest tree level, which is also the largest number of comparisons needed to reach an ordered configuration of array stored in the root. First, the inequality $2^{i-1} \geq n!$ is transformed into $i-1 \geq \lg(n!)$, which means that the path length in a decision tree with at least $n!$ leaves must be at least $\lg(n!)$, or rather, it must be $\lceil \lg(n!) \rceil$, where $\lceil x \rceil$ is an integer not less than x. See the example in Figure 9.5.

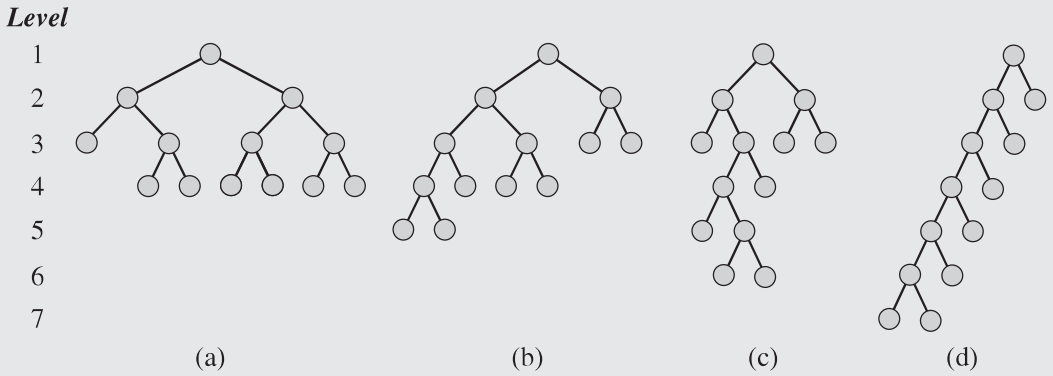
It can be proven that, for a randomly chosen leaf of an m-leaf decision tree, the length of the path from the root to the leaf is not less than $\lg m$ and that, both in the average case and the worst case, the required number of comparisons, $\lg(n!)$, is $O(n \lg n)$ (see Section A.2 in Appendix A). That is, $O(n \lg n)$ is also the best that can be expected in average cases.

It is interesting to compare this approximation to some of the numbers computed for sorting methods, especially for the average and worst cases. For example, insertion sort requires only $n-1$ comparisons in the best case, but in the average and the worst cases, this sort turns into an n^2 algorithm because the functions relating the number of comparisons to the number of elements are, for these cases, the big-Os of n^2. This is much greater than $n \lg n$, especially for large numbers. Consequently, insertion sort is not an ideal algorithm. The quest for better methods can be continued with at least the expectation that the number of comparisons should be approximated by $n \lg n$ rather than by n^2.

The difference between these two functions is best seen in Figure 9.6 if the performance of the algorithms analyzed so far is compared with the expected performance $n \lg n$ in the average case. The numbers in the table in Figure 9.6 show that if 100 items are sorted, the desired algorithm is four times faster than insertion sort and eight times faster than selection sort and bubble sort. For 1,000 items, it is 25 and 50 times faster, respectively. For 10,000, the difference in performance differs by factors

FIGURE 9.5 Examples of decision trees for an array of three elements.

These are some possible decision trees for an array of three elements. These trees must have at least 3! = 6 leaves. For the sake of the example, it is assumed that each tree has one extra leaf (a repetition or a failure). In the worst and average cases, the number of comparisons is $i - 1 \geq \lceil \lg(n!) \rceil$. In this example, $n = 3$, so $i - 1 \geq \lceil \lg 3! \rceil = \lceil \lg 6 \rceil \approx \lceil 2.59 \rceil = 3$. And, in fact, only for the best balanced tree (a), the nonrounded length of the average path is less than three.



These are the sums of the lengths of the paths from the root to all leaves in trees (a) – (d) and the average path lengths:

(a) $2 + 3 + 3 + 3 + 3 + 3 + 3 = 20$; average $= \frac{20}{7} \approx 2.86$

(b) $4 + 4 + 3 + 3 + 3 + 2 + 2 = 21$; average $= \frac{21}{7} = 3$

(c) $2 + 4 + 5 + 5 + 3 + 2 + 2 = 23$; average $= \frac{23}{7} \approx 3.29$

(d) $6 + 6 + 5 + 4 + 3 + 2 + 1 = 27$; average $= \frac{27}{7} \approx 3.86$

FIGURE 9.6 Number of comparisons performed by the simple sorting method and by an algorithm whose efficiency is estimated by the function $n \lg n$.

sort type	n	100	1,000	10,000
insertion	$\frac{n^2 + n - 2}{4}$	2,524.5	250,249.5	25,002,499.5
selection, bubble	$\frac{n(n-1)}{2}$	4,950	499,500	49,995,000
expected	$n \lg n$	664	9,966	132,877

of 188 and 376, respectively. This can only serve to encourage the search for an algorithm embodying the performance of the function $n \lg n$.

9.3 Efficient Sorting Algorithms

9.3.1 Shell Sort

The $O(n^2)$ limit for a sorting method is much too large and must be broken to improve efficiency and decrease run time. How can this be done? The problem is that the time required for ordering an array by the three sorting algorithms usually grows faster than the size of the array. In fact, it is customarily a quadratic function of that size. It may turn out to be more efficient to sort parts of the original array first and then, if they are at least partially ordered, to sort the entire array. If the subarrays are already sorted, we are that much closer to the best case of an ordered array than initially. A general outline of such a procedure is as follows:

```
divide data into h subarrays;
for i = 1 to h
    sort subarray data_i;
sort array data;
```

If h is too small, then the subarrays $\texttt{data}_i$ of array `data` could be too large, and sorting algorithms might prove inefficient as well. On the other hand, if h is too large, then too many small subarrays are created, and although they are sorted, it does not substantially change the overall order of `data`. Lastly, if only one such partition of `data` is done, the gain on the execution time may be rather modest. To solve that problem, several different subdivisions are used, and for every subdivision, the same procedure is applied separately, as in:

```
determine numbers h_t ... h_1 of ways of dividing array data into subarrays;
for (h=h_t; t > 1; t--, h=h_t)
    divide data into h subarrays;
    for i = 1 to h
        sort subarray data_i;
sort array data;
```

This idea is the basis of the *diminishing increment sort,* also known as *Shell sort* and named after Donald L. Shell who designed this technique. Note that this pseudocode does not identify a specific sorting method for ordering the subarrays; it can be any simple method. Usually, however, Shell sort uses insertion sort.

The heart of Shell sort is an ingenious division of the array `data` into several subarrays. The trick is that elements spaced farther apart are compared first, then the elements closer to each other are compared, and so on, until adjacent elements are compared on the last pass. The original array is logically subdivided into subarrays by picking every h_tth element as part of one subarray. Therefore, there are h_t subarrays, and for every $h = 1, \ldots, h_t$,

$$\texttt{data}_{h_t h}[i] = \texttt{data}[h_t \cdot i + (h-1)]$$

For example, if $h_t = 3$, the array `data` is subdivided into three subarrays $\texttt{data}_1$, $\texttt{data}_2$, and $\texttt{data}_3$ so that

$\texttt{data}_{31}\texttt{[0]} = \texttt{data[0]}, \texttt{data}_{31}\texttt{[1]} = \texttt{data[3]}, \ldots, \texttt{data}_{31}\texttt{[i]} = \texttt{data[3*i]}, \ldots$
$\texttt{data}_{32}\texttt{[0]} = \texttt{data[1]}, \texttt{data}_{32}\texttt{[1]} = \texttt{data[4]}, \ldots, \texttt{data}_{32}\texttt{[i]} = \texttt{data[3*i+1]}, \ldots$
$\texttt{data}_{33}\texttt{[0]} = \texttt{data[2]}, \texttt{data}_{33}\texttt{[1]} = \texttt{data[5]}, \ldots, \texttt{data}_{33}\texttt{[i]} = \texttt{data[3*i+2]}, \ldots$

and these subarrays are sorted separately. After that, new subarrays are created with an $h_{t-1} < h_t$, and insertion sort is applied to them. The process is repeated until no subdivisions can be made. If $h_t = 5$, the process of extracting subarrays and sorting them is called a 5-sort.

Figure 9.7 shows the elements of the array `data` that are five positions apart and are logically inserted into a separate array—"logically" because physically they still occupy the same positions in `data`. For each value of increment h_t, there are h_t subarrays, and each of them is sorted separately. As the value of the increment decreases, the number of subarrays decreases accordingly, and their sizes grow. Much of `data`'s disorder has been removed in the earlier iterations, so on the last pass, the array is much closer to its final form than before all the intermediate *h*-sorts.

FIGURE 9.7 The array [10 8 6 20 4 3 22 1 0 15 16] sorted by Shell sort.

`data` before 5-sort	10	8	6	20	4	3	22	1	0	15	16
Five subarrays before sorting	10	—	—	—	—	3	—	—	—	—	16
		8	—	—	—	—	22				
			6	—	—	—	—	1			
				20	—	—	—	—	0		
					4	—	—	—	—	15	
Five subarrays after sorting	3	—	—	—	—	10	—	—	—	—	16
		8	—	—	—	—	22				
			1	—	—	—	—	6			
				0	—	—	—	—	20		
					4	—	—	—	—	15	
`data` after 5-sort and before 3-sort	3	8	1	0	4	10	22	6	20	15	16
Three subarrays before sorting	3	—	—	0	—	—	22	—	—	15	
		8	—	—	4	—	—	6	—	—	16
			1	—	—	10	—	—	20		
Three subarrays after sorting	0	—	—	3	—	—	15	—	—	22	
		4	—	—	6	—	—	8	—	—	16
			1	—	—	10	—	—	20		
`data` after 3-sort and before 1-sort	0	4	1	3	6	10	15	8	20	22	16
`data` after 1-sort	0	1	3	4	6	8	10	15	16	20	22

One problem still has to be addressed, namely, choosing the optimal value of the increment. In the example in Figure 9.7, the value of 5 is chosen to begin with, then 3, and 1 is used for the final sort. But why these values? Unfortunately, no convincing answer can be given. In fact, any decreasing sequence of increments can be used as long as the last one, h_1, is equal to 1. Donald Knuth has shown that even if there are only two increments, $(\frac{16n}{\pi})^{\frac{1}{3}}$ and 1, Shell sort is more efficient than insertion sort because it takes $O(n^{\frac{5}{3}})$ time instead of $O(n^2)$. But the efficiency of Shell sort can be improved by using a larger number of increments. It is imprudent, however, to use sequences of increments such as 1, 2, 4, 8, . . . or 1, 3, 6, 9, . . . because the mixing effect of data is lost.

For example, when using 4-sort and 2-sort, a subarray, $\texttt{data}_{2,i}$, for $i = 1, 2$, consists of elements of two arrays, $\texttt{data}_{4,i}$ and $\texttt{data}_{4,j}$, where $j = i + 2$, and only those. It is much better if elements of $\texttt{data}_{4,i}$ do not meet together again in the same array because a faster reduction in the number of exchange inversions is achieved if they are sent to different arrays when performing the 2-sort. Using only powers of 2 for the increments, as in Shell's original algorithm, the items in the even and odd positions of the array do not interact until the last pass, when the increment equals 1. This is where the mixing effect (or lack thereof) comes into play. But there is no formal proof indicating which sequence of increments is optimal. Extensive empirical studies along with some theoretical considerations suggest that it is a good idea to choose increments satisfying the conditions

$$h_1 = 1$$

$$h_{i+1} = 3h_i + 1$$

and stop with h_t for which $h_{t+2} \geq n$. For $n = 10{,}000$, this gives the sequence

$$1, 4, 13, 40, 121, 364, 1093, 3280$$

Experimental data have been approximated by the exponential function, the estimate, $1.21n^{\frac{5}{4}}$, and the logarithmic function $.39n \ln^2 n - 2.33n \ln n = O(n \ln^2 n)$. The first form fits the results of the tests better. $1.21n^{1.25} = O(n^{1.25})$ is much better than $O(n^2)$ for insertion sort, but it is still much greater than the expected $O(n \lg n)$ performance.

Figure 9.8 contains a function to sort the array `data` using Shell sort. Note that before sorting starts, increments are computed and stored in the array `increments`.

The core of Shell sort is to divide an array into subarrays by taking elements h positions apart. Three features of this algorithm vary from one implementation to another:

1. The sequence of increments
2. A simple sorting algorithm applied in all passes except the last
3. A simple sorting algorithm applied only in the last pass, for 1-sort

In our implementation, as in Shell's, insertion sort is applied in all h-sorts, but other sorting algorithms can be used. For example, Incerpi and Sedgewick use two

FIGURE 9.8 Implementation of Shell sort.

```
template<class T>
void Shellsort(T data[], int n) {
    register int i, j, hCnt, h;
    int increments[20], k;
//  create an appropriate number of increments h
    for (h = 1, i = 0; h < n; i++) {
        increments[i] = h;
        h = 3*h + 1;
    }
 // loop on the number of different increments h
    for (i--; i >= 0; i--) {
        h = increments[i];
     // loop on the number of subarrays h-sorted in ith pass
        for (hCnt = h; hCnt < 2*h; hCnt++) {
         // insertion sort for subarray containing every hth element of
            for (j = hCnt; j < n; ) {   // array data
                T tmp = data[j];
                k = j;
                while (k-h >= 0 && tmp < data[k-h]) {
                    data[k] = data[k-h];
                    k -= h;
                }
                data[k] = tmp;
                j += h;
            }
        }
    }
}
```

iterations of cocktail shaker sort and a version of bubble sort in each h-sort and finish with insertion sort, obtaining what they call a *shakersort*. All these versions perform better than simple sorting methods, although there are some differences in performance among versions. Analytical results concerning the complexity of these sorts are not available. All results regarding complexity are of an empirical nature.

9.3.2 Heap Sort

Selection sort makes $O(n^2)$ comparisons and is very inefficient, especially for large n. But it performs relatively few moves. If the comparison part of the algorithm is improved, the end results can be promising.

Heap sort was invented by John Williams and uses the approach inherent to selection sort. Selection sort finds among the n elements the one that precedes all other $n - 1$ elements, then the least element among those $n - 1$ items, and so forth, until the array is sorted. To have the array sorted in ascending order, heap sort puts the largest element at the end of the array, then the second largest in front of it, and so on. Heap sort starts from the end of the array by finding the largest elements, whereas selection sort starts from the beginning using the smallest elements. The final order in both cases is indeed the same.

Heap sort uses a heap as described in Section 6.9. A heap is a binary tree with the following two properties:

1. The value of each node is not less than the values stored in each of its children.
2. The tree is perfectly balanced and the leaves in the last level are all in the leftmost positions.

A tree has the heap property if it satisfies condition 1. Both conditions are useful for sorting, although this is not immediately apparent for the second condition. The goal is to use only the array being sorted without using additional storage for the array elements; by condition 2, all elements are located in consecutive positions in the array starting from position 0, with no unused position inside the array. In other words, condition 2 reflects the packing of an array with no gaps.

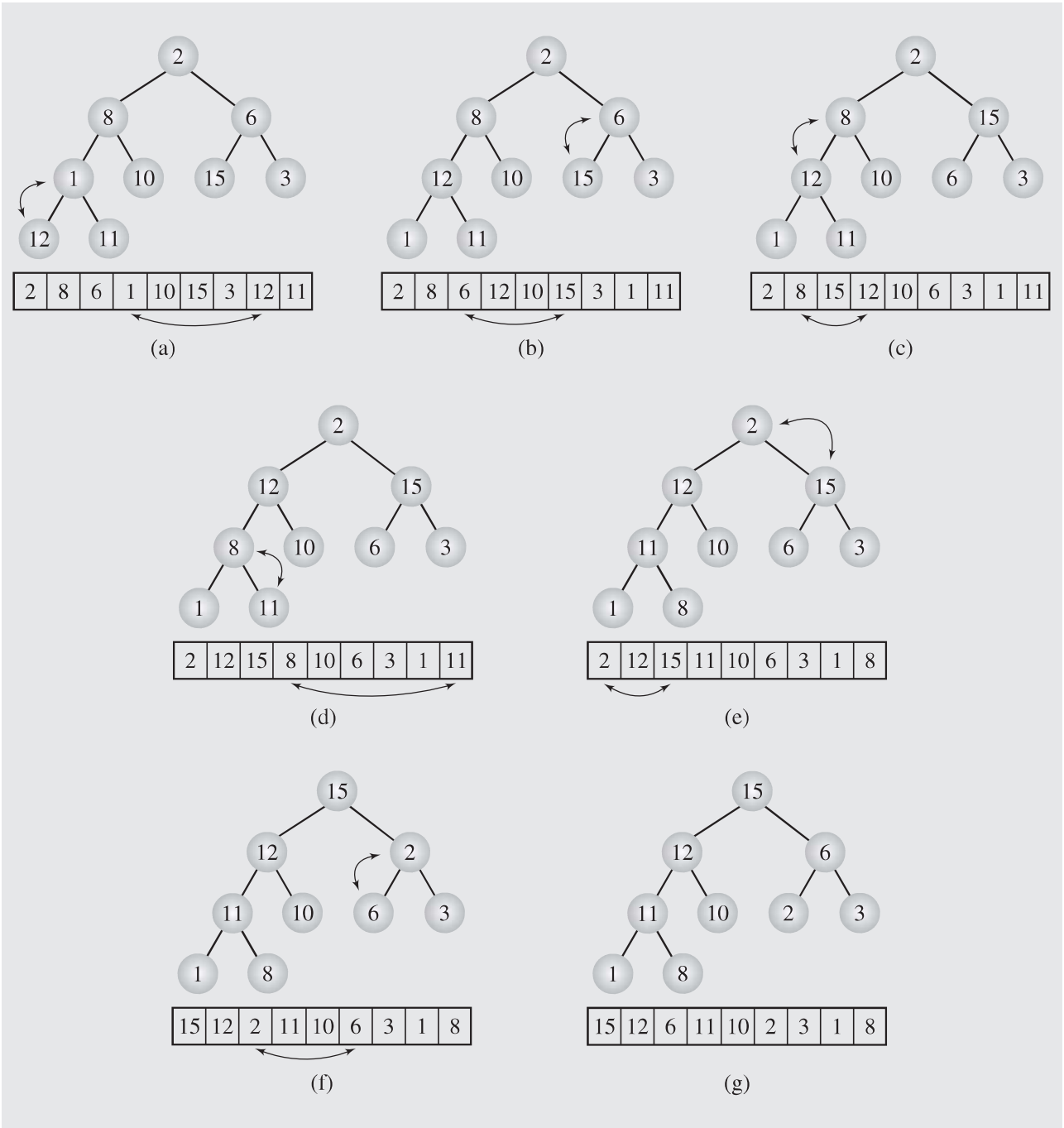
Elements in a heap are not perfectly ordered. It is known only that the largest element is in the root node and that, for each other node, all its descendants are not greater than the element in this node. Heap sort thus starts from the heap, puts the largest element at the end of the array, and restores the heap that now has one less element. From the new heap, the largest element is removed and put in its final position, and then the heap property is restored for the remaining elements. Thus, in each round, one element of the array ends up in its final position, and the heap becomes smaller by this one element. The process ends with exhausting all elements from the heap and is summarized in the following pseudocode:

```
heapsort(data[],n)
    transform data into a heap;
        for i = down to 2
        swap the root with the element in position i;
        restore the heap property for the tree data[0],...,data[i-1];
```

In the first phase of heap sort, an array is transformed into a heap. In this process, we use a bottom-up method devised by Floyd and described in Section 6.9.2. All steps leading to the transformation of the array [2 8 6 1 10 15 3 12 11] into a heap are illustrated in Figure 9.9 (see also Figure 6.58).

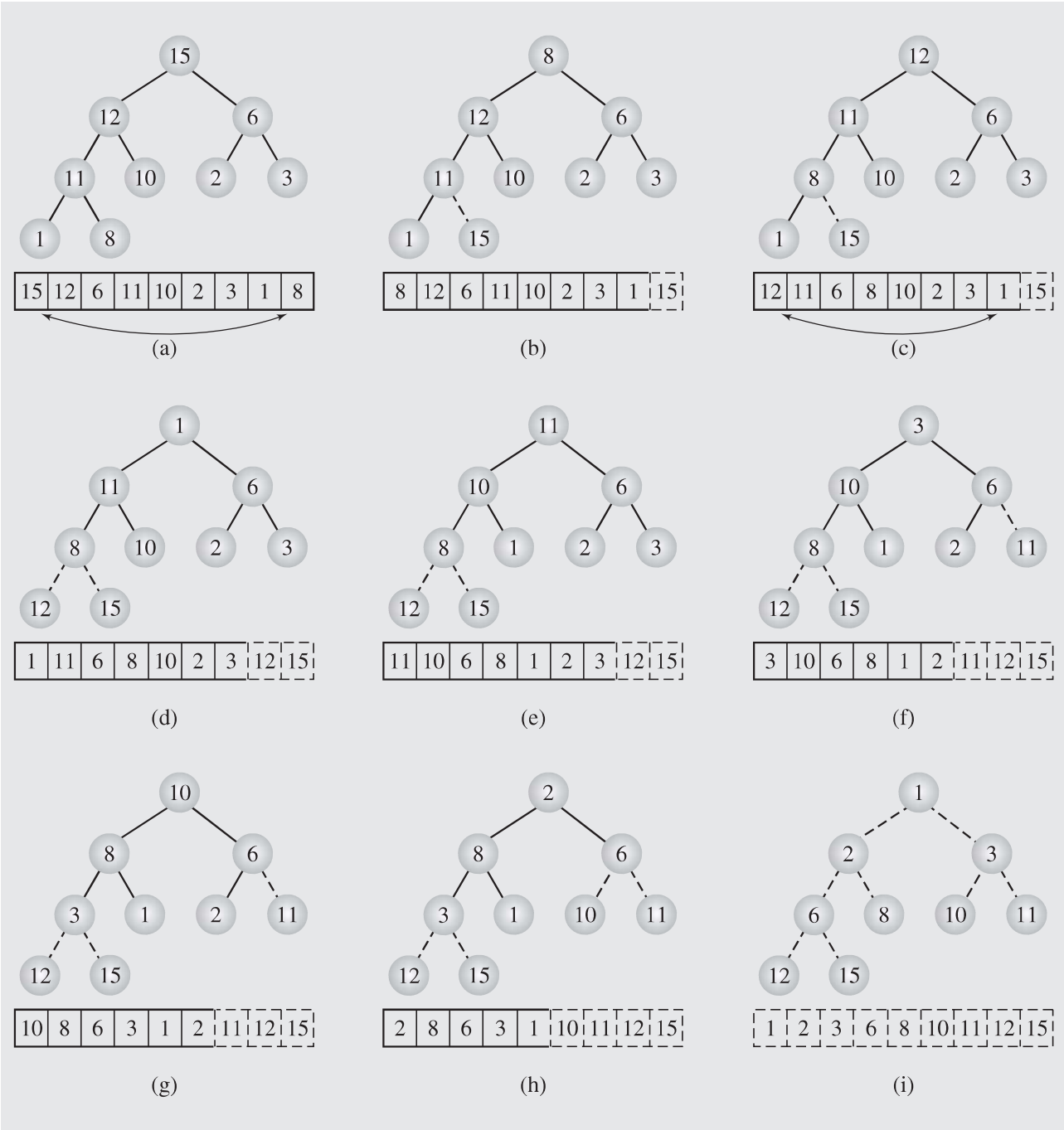
The second phase begins after the heap has been built (Figures 9.9g and 9.10a). At that point, the largest element, number 15, is moved to the end of the array. Its place is taken by 8, thus violating the heap property. The property has to be restored, but this time it is done for the tree without the largest element, 15. Because it is already in its proper position, it does not need to be considered anymore and is removed (pruned) from the tree (indicated by the dashed lines in Figure 9.10). Now, the largest

FIGURE 9.9 Transforming the array [2 8 6 1 10 15 3 12 11] into a heap.



element among `data[0]`, . . . , `data[n-2]` is looked for. To that end, the function `moveDown()` from Section 6.9 (Figure 6.56) is called to construct a heap out of all the elements of `data` except the last, `data[n-1]`, which results in the heap in Figure 9.10c.

FIGURE 9.10 Execution of heap sort on the array [15 12 6 11 10 2 3 1 8], which is the heap constructed in Figure 9.9.



Number 12 is sifted up and then swapped with 1, giving the tree in Figure 9.10d. The function `moveDown()` is called again to select 11 (Figure 9.10e), and the element is swapped with the last element of the current subarray, which is 3 (Figure 9.10f). Now 10 is selected (Figure 9.10g) and exchanged with 2 (Figure 9.10h). The reader can easily

construct trees and heaps for the next passes through the loop of `heapsort()`. After the last pass, the array is in ascending order and the tree is ordered accordingly. An implementation of `heapsort()` is as follows:

```
template<class T>
void heapsort(T data[], int n) {
    for (int i = n/2 - 1; i >= 0; --i)    // create a heap;
        moveDown (data,i,n-1);
    for (int i = n-1; i >= 1; --i) {
        swap(data[0],data[i]); // move the largest item to data[i];
        moveDown(data,0,i-1);  // restore the heap property;
    }
}
```

Heap sort might be considered inefficient because the movement of data seems to be very extensive. First, all effort is applied to moving the largest element to the leftmost side of the array in order to move it to the furthest right. But therein lies its efficiency. In the first phase, to create the heap, `heapsort()` uses `moveDown()`, which performs $O(n)$ steps (see Section 6.9.2).

In the second phase, `heapsort()` exchanges $n-1$ times the root with the element in position i and also restores the heap $n-1$ times, which in the worst case causes `moveDown()` to iterate $\lg i$ times to bring the root down to the level of the leaves. Thus, the total number of moves in all executions of `moveDown()` in the second phase of `heapsort()` is $\sum_{i=1}^{n-1} \lg i$, which is $O(n \lg n)$. In the worst case, `heapsort()` requires $O(n)$ steps in the first phase, and in the second phase, $n-1$ swaps and $O(n \lg n)$ operations to restore the heap property, which gives $O(n) + O(n \lg n) + (n-1) = O(n \lg n)$ exchanges for the whole process in the worst case.

For the best case, when the array contains identical elements, `moveDown()` is called $\frac{n}{2}$ times in the first phase, but no moves are performed. In the second phase, `heapsort()` makes one swap to move the root element to the end of the array, resulting in only $n-1$ moves. Also, in the best case, n comparisons are made in the first phase and $2(n-1)$ in the second. Hence, the total number of comparisons in the best case is $O(n)$. However, if the array has distinct elements, then in the best case the number of comparisons equals $n \lg n - O(n)$ (Ding and Weiss 1992).

9.3.3 Quicksort

Shell sort approached the problem of sorting by dividing the original array into subarrays, sorting them separately, and then dividing them again to sort the new subarrays until the whole array is sorted. The goal was to reduce the original problem to subproblems that can be solved more easily and quickly. The same reasoning was a guiding principle for C. A. R. Hoare, who invented an algorithm appropriately called *quicksort.*

The original array is divided into two subarrays, the first of which contains elements less than or equal to a chosen key called the *bound* or *pivot.* The second subarray includes elements equal to or greater than the bound. The two subarrays can be sorted separately, but before this is done, the partition process is repeated for both subarrays. As a result, two new bounds are chosen, one for each subarray. The four subarrays are created because each subarray obtained in the first phase is now divided into two segments. This process of partitioning is carried down until there are

only one-cell arrays that do not need to be sorted at all. By dividing the task of sorting a large array into two simpler tasks and then dividing those tasks into even simpler tasks, it turns out that in the process of getting prepared to sort, the data have already been sorted. Because the sorting has been somewhat dissipated in the preparation process, this process is the core of quicksort.

Quicksort is recursive in nature because it is applied to both subarrays of an array at each level of partitioning. This technique is summarized in the following pseudocode:

```
quicksort(array[])
   if length(array) > 1
        choose bound; // partition array into subarray1 and subarray2
        while there are elements left in array
              include element either in subarray1 = {el: el ≤ bound}
              or in subarray2 = {el: el ≥ bound};
        quicksort(subarray1);
        quicksort(subarray2);
```

To partition an array, two operations have to be performed: a bound has to be found and the array has to be scanned to place the elements in the proper subarrays. However, choosing a good bound is not a trivial task. The problem is that the subarrays should be approximately the same length. If an array contains the numbers 1 through 100 (in any order) and 2 is chosen as a bound, then an imbalance results: The first subarray contains only 1 number after partitioning, whereas the second has 99 numbers.

A number of different strategies for selecting a bound have been developed. One of the simplest consists of choosing the first element of an array. That approach can suffice for some applications; however, because many arrays to be sorted already have many elements in their proper positions, a more cautious approach is to choose the element located in the middle of the array. This approach is incorporated in the implementation in Figure 9.11.

Another task is scanning the array and dividing the elements between its two subarrays. The pseudocode is vague about how this can be accomplished. In particular, it does not decide where to place an element equal to the bound. It only says that elements are placed in the first subarray if they are less than or the same as the bound and in the second if they are greater than or the same as the bound. The reason is that the difference between the lengths of the two subarrays should be minimal. Therefore, elements equal to the bound should be so divided between the two subarrays as to make this difference in size minimal. The details of handling this depend on a particular implementation, and one such implementation is given in Figure 9.11. In this implementation, `quicksort(data[],n)` preprocesses the array to be sorted by locating the largest element in the array and exchanging it with the last element of the array. Having the largest element at the end of the array prevents the index `lower` from running off the end of the array. This could happen in the first inner `while` loop if the bound were the largest element in the array. The index `lower` would be constantly incremented, eventually causing an abnormal program termination. Without this preprocessing, the first inner `while` loop would have to be

```
while (lower < last && data[lower] < bound)
```

The first test, however, would be necessary only in extreme cases, but it would be executed in each iteration of this `while` loop.

FIGURE 9.11 Implementation of quicksort.

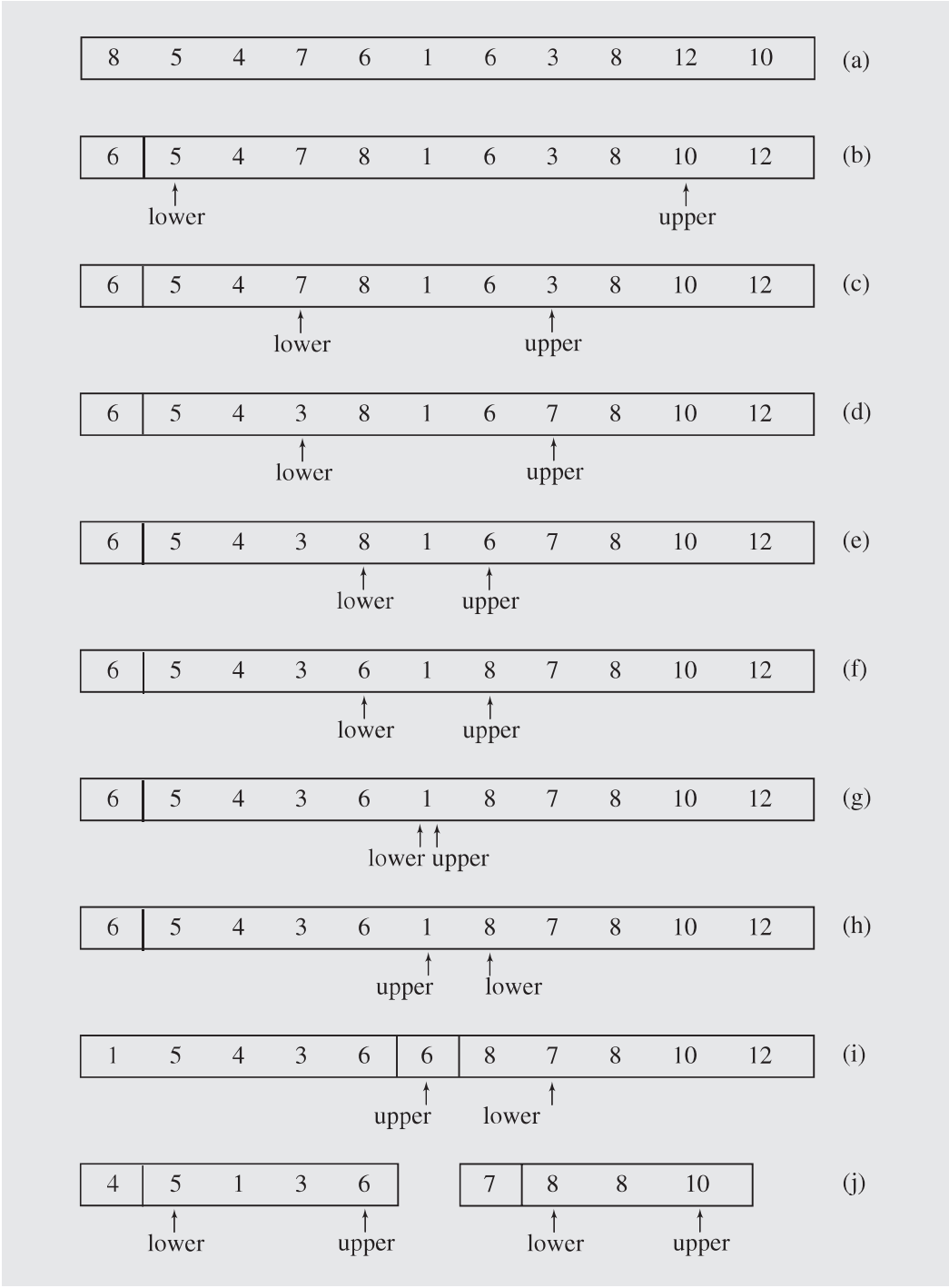
```
template<class T>
void quicksort(T data[], int first, int last) {
    int lower = first+1, upper = last;
    swap(data[first],data[(first+last)/2]);
    T bound = data[first];
    while (lower <= upper) {
        while (bound > data[lower])
            lower++;
        while (bound < data[upper])
            upper--;
        if (lower < upper)
            swap(data[lower++],data[upper--]);
        else lower++;
    }
    swap(data[upper],data[first]);
    if (first < upper-1)
        quicksort (data,first,upper-1);
    if (upper+1 < last)
        quicksort (data,upper+1,last);
}

template<class T>
void quicksort(T data[], int n) {
    int i, max;
    if (n < 2)
        return;
    for (i = 1, max = 0; i < n; i++)    // find the largest
        if (data[max] < data[i])        // element and put it
            max = i;                    // at the end of data[];
    swap(data[n-1],data[max]); // largest el is now in its
    quicksort(data,0,n-2);     // final position;
}
```

In this implementation, the main property of the bound is used, namely, that it is a boundary item. Hence, as befits the boundary item, it is placed on the borderline between the two subarrays obtained as a result of one call to `quicksort()`. In this way, the bound is located in its final position and can be excluded from further processing. To ensure that the bound is not moved around, it is stashed in the first position, and after partitioning is done, it is moved to its proper position, which is the rightmost position of the first subarray.

Figure 9.12 contains an example of partitioning the array [8 5 4 7 6 1 6 3 8 12 10]. In the first partitioning, the largest element in the array is located and exchanged with

FIGURE 9.12 Partitioning the array [8 5 4 7 6 1 6 3 8 12 10] with `quicksort()`.



the last element, resulting in the array [8 5 4 7 6 1 6 3 8 10 12]. Because the last element is already in its final position, it does not have to be processed anymore. Therefore, in the first partitioning, `lower` = 1, `upper` = 9, and the first element of the array, 8, is exchanged with the bound, 6 in position 4, so that the array is [6 5 4 7 8 1 6 3 8 10 12]

(Figure 9.12b). In the first iteration of the outer `while` loop, the inner `while` loop moves `lower` to position 3 with 7, which is greater than the bound. The second inner `while` loop moves `upper` to position 7 with 3, which is less than the bound (Figure 9.12c). Next the elements in these two cells are exchanged, giving the array [6 5 4 3 8 1 6 7 8 10 12] (Figure 9.12d). Then `lower` is incremented to 4 and `upper` is decremented to 6 (Figure 9.12e). This concludes the first iteration of the outer `while` loop.

In its second iteration, neither of the two inner `while` loops modifies any of the two indexes because `lower` indicates a position occupied by 8, which is greater than the bound, and `upper` indicates a position occupied by 6, which is equal to the bound. The two numbers are exchanged (Figure 9.12f), and then both indexes are updated to 5 (Figure 9.12g).

In the third iteration of the outer `while` loop, `lower` is moved to the next position containing 8, which is greater than the bound, and `upper` stays in the same position because 1 in this position is less than the bound (Figure 9.12h). But at that point, `lower` and `upper` cross each other, so no swapping takes place, and after a redundant increment of `lower` to 7, the outer `while` loop is exited. At that point, `upper` is the index of the rightmost element of the first subarray (with the element not exceeding the bound), so the element in this position is exchanged with the bound (Figure 9.12i). In this way, the bound is placed in its final position and can be excluded from subsequent processing. Therefore, the two subarrays that are processed next are the left subarray, with elements to the left of the bound, and the right subarray, with elements to its right (Figure 9.12j). Then partitioning is performed for these two subarrays separately, and then for subarrays of these subarrays, until subarrays have less than two elements. The entire process is summarized in Figure 9.13, in which all the changes in all current arrays are indicated.

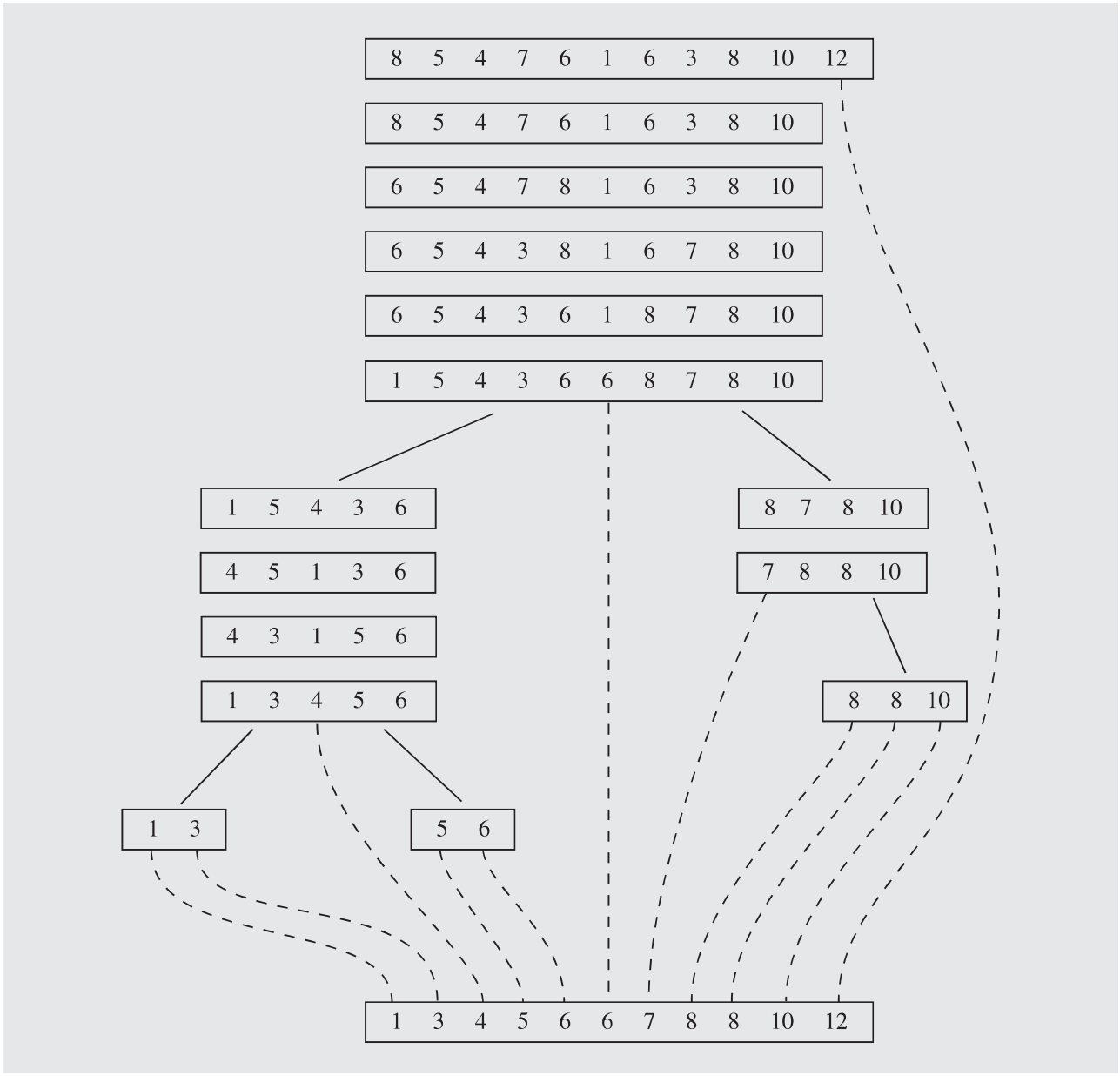
The worst case occurs if in each invocation of `quicksort()`, the smallest (or largest) element of the array is chosen for the bound. This is the case if we try to sort the array [5 3 1 2 4 6 8]. The first bound is 1, and the array is broken into an empty array and the array [3 5 2 4 6] (the largest number, 8, does not participate in partitioning). The new bound is 2, and again only one nonempty array, [5 3 4 6], is obtained as the result of partitioning. The next bound and array returned by partition are 3 and [5 4 6], then 4 and [5 6], and finally 5 and [6]. The algorithm thus operates on arrays of size $n - 1, n - 2, \ldots, 2$. The partitions require $n - 2 + n - 3 + \cdots + 1$ comparisons, and for each partition, only the bound is placed in the proper position. This results in a run time equal to $O(n^2)$, which is hardly a desirable result, especially for large arrays or files.

The best case is when the bound divides an array into two subarrays of approximately length $\frac{n}{2}$. If the bounds for both subarrays are well chosen, the partitions produce four new subarrays, each of them with approximately $\frac{n}{4}$ cells. If, again, the bounds for all four subarrays divide them evenly, the partitions give eight subarrays, each with approximately $\frac{n}{8}$ elements. Therefore, the number of comparisons performed for all partitions is approximately equal to

$$n + 2\frac{n}{2} + 4\frac{n}{4} + 8\frac{n}{8} + \cdots + n\frac{n}{n} = n(lg\, n + 1)$$

which is $O(n \lg n)$. This is due to the fact that parameters in the terms of this sum (and also the denominators) form a geometric sequence so that $n = 2^k$ for $k = \lg n$ (assuming that n is a power of 2).

FIGURE 9.13 Sorting the array [8 5 4 7 6 1 6 3 8 12 10] with `quicksort()`.



Now we can answer the question asked before: Is the average case, when the array is ordered randomly, closer to the best case, $n \lg n$, or to the worst, $O(n^2)$? Some calculations show that the average case requires only $O(n \lg n)$ comparisons (see Appendix A.3), which is the desired result. The validity of this figure can be strengthened by referring to the tree obtained after disregarding the bottom rectangle in Figure 9.13. This tree indicates how important it is to keep the tree balanced, for the smaller the number of levels, the quicker the sorting process. In the extreme case, the tree can be turned into a linked list in which every nonleaf node has only one child. That rather rare phenomenon is possible and prevents us from calling quicksort the

ideal sort. But quicksort seems to be closest to such an ideal because, as analytic studies indicate, it outperforms other efficient sorting methods by at least a factor of 2.

How can the worst case be avoided? The partition procedure should produce arrays of approximately the same size, which can be achieved if a good bound is chosen. This is the crux of the matter: How can the best bound be found? Only two methods will be mentioned. The first method randomly generates a number between `first` and `last`. This number is used as an index of the bound, which is then interchanged with the first element of the array. In this method, the partition process proceeds as before. Good random number generators may slow down the execution time as they themselves often use sophisticated and time-consuming techniques. Thus, this method is not highly recommended.

The second method chooses a median of three elements: the first, middle, and last. For the array [1 5 4 7 8 6 6 3 8 12 10], 6 is chosen from the set [1 6 10], and for the first generated subarray, the bound 4 is chosen from the set [1 4 6]. Obviously, there is the possibility that all three elements are always the smallest (or the largest) in the array, but it does not seem very likely.

Is quicksort the best sorting algorithm? It certainly is—usually. It is not bulletproof, however, and some problems have already been addressed in this section. First, everything hinges on which element of the file or array is chosen for the bound. Ideally, it should be the median element of the array. An algorithm to choose a bound should be flexible enough to handle all possible orderings of the data to be sorted. Because some cases always slip by these algorithms, from time to time quicksort can be expected to be anything but quick.

Second, it is inappropriate to use quicksort for small arrays. For arrays with fewer than 30 items, insertion sort is more efficient than quicksort (Cook and Kim 1980). In this case the initial pseudocode can be changed to

```
quicksort2 (array[])
   if length(array) > 30
      partition array into subarray1 and subarray2;
      quicksort2(subarray1);
      quicksort2(subarray2);
   else insertionsort(array);
```

and the implementations changed accordingly. However, the table in Figure 9.19 later in this chapter indicates that the improvement is not significant.

9.3.4 Mergesort

The problem with quicksort is that its complexity in the worst case is $O(n^2)$ because it is difficult to control the partitioning process. Different methods of choosing a bound attempt to make the behavior of this process fairly regular; however, there is no guarantee that partitioning results in arrays of approximately the same size. Another strategy is to make partitioning as simple as possible and concentrate on merging the two sorted arrays. This strategy is characteristic of *mergesort*. It was one of the first sorting algorithms used on a computer and was developed by John von Neumann.

The key process in mergesort is merging sorted halves of an array into one sorted array. However, these halves have to be sorted first, which is accomplished by

merging the already sorted halves of these halves. This process of dividing arrays into two halves stops when the array has fewer than two elements. The algorithm is recursive in nature and can be summarized in the following pseudocode:

```
mergesort(data[])
   if data have at least two elements
      mergesort(left half of data);
      mergesort(right half of data);
      merge(both halves into a sorted list);
```

Merging two subarrays into one is a relatively simple task, as indicated in this pseudocode:

```
merge(array1[], array2[], array3[])
   i1, i2, i3 are properly initialized;
   while both array2 and array3 contain elements
      if array2[i2] < array3[i3]
           array1[i1++] = array2[i2++];
      else array1[i1++] = array3[i3++];
   load into array1 the remaining elements of either array2 or array3;
```

For example, if `array2` = [1 4 6 8 10] and `array3` = [2 3 5 22], then the resulting `array1` = [1 2 3 4 5 6 8 10 22].

The pseudocode for `merge()` suggests that `array1`, `array2`, and `array3` are physically separate entities. However, for the proper execution of `mergesort()`, `array1` is a concatenation of `array2` and `array3` so that `array1` before the execution of `merge()` is [1 4 6 8 10 2 3 5 22]. In this situation, `merge()` leads to erroneous results, because after the second iteration of the `while` loop, `array2` is [1 2 6 8 10] and `array1` is [1 2 6 8 10 2 3 5 22]. Therefore, a temporary array has to be used during the merging process. At the end of the merging process, the contents of this temporary array are transferred to `array1`. Because `array2` and `array3` are subarrays of `array1`, they do not need to be passed as parameters to `merge()`. Instead, indexes for the beginning and the end of `array1` are passed, because `array1` can be a part of another array. The new pseudocode is

```
merge (array1[], first, last)
   mid = (first + last) / 2;
   i1 = 0;
   i2 = first;
   i3 = mid + 1;
   while both left and right subarrays of array1 contain elements
      if array1[i2] < array1[i3]
           temp[i1++] = array1[i2++];
      else temp[i1++] = array1[i3++];
   load into temp the remaining elements of array1;
   load to array1 the content of temp;
```

The entire `array1` is copied to `temp` and then `temp` is copied back to `array1`, so the number of movements in each execution of `merge()` is always the same and

is equal to 2 · (`last−first` + 1). The number of comparisons depends on the ordering in `array1`. If `array1` is in order or if the elements in the right half precede the elements in the left half, the number of comparisons is (`first` + `last`)/2. The worst case is when the last element of one half precedes only the last element of the other half, as in [1 6 10 12] and [5 9 11 13]. In this case, the number of comparisons is `last−first`. For an n-element array, the number of comparisons is $n-1$.

The pseudocode for `mergesort()` is now

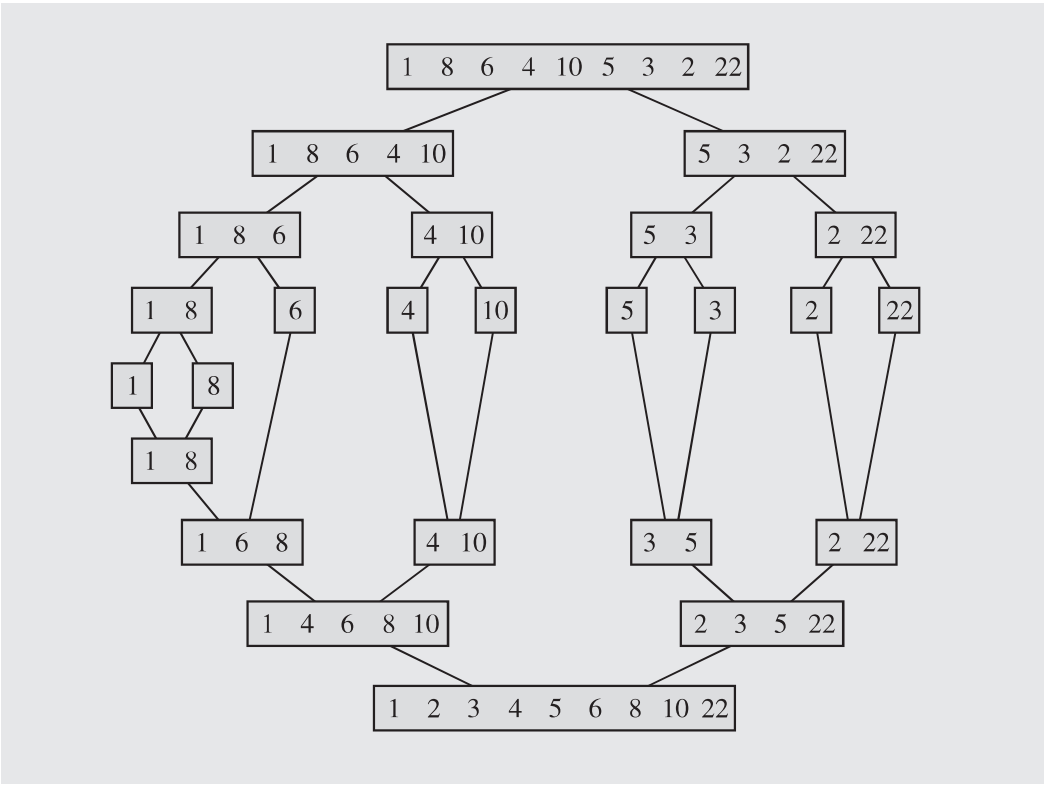
```
mergesort (data[], first, last)
   if first < last
      mid = (first + last) / 2;
      mergesort(data, first, mid);
      mergesort(data, mid+1, last);
      merge(data, first, last);
```

Figure 9.14 illustrates an example using this sorting algorithm. This pseudocode can be used to analyze the computing time for mergesort. For an n-element array, the number of movements is computed by the following recurrence relation:

$$M(1) = 0$$

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

FIGURE 9.14 The array [1 8 6 4 10 5 3 2 22] sorted by mergesort.



$M(n)$ can be computed in the following way:

$$\begin{aligned} M(n) &= 2\left(2M\left(\frac{n}{4}\right) + 2\left(\frac{n}{4}\right)\right) + 2n = 4M\left(\frac{n}{4}\right) + 4n \\ &= 4\left(2M\left(\frac{n}{8}\right) + 2\left(\frac{n}{4}\right)\right) + 4n = 8M\left(\frac{n}{8}\right) + 6n \\ &\vdots \\ &= 2^i M\left(\frac{n}{2^i}\right) + 2in \end{aligned}$$

Choosing $i = \lg n$ so that $n = 2^i$ allows us to infer

$$M(n) = 2^i M\left(\frac{n}{2^i}\right) + 2in = nM(1) + 2n \lg n = 2n \lg n = O(n \lg n)$$

The number of comparisons in the worst case is given by a similar relation:

$$\begin{aligned} C(1) &= 0 \\ C(n) &= 2C\left(\frac{n}{2}\right) + n - 1 \end{aligned}$$

which also results in $C(n)$ being $O(n \lg n)$.

Mergesort can be made more efficient by replacing recursion with iteration (see the exercises at the end of this chapter) or by applying insertion sort to small portions of an array, a technique that was suggested for quicksort. However, mergesort has one serious drawback: the need for additional storage for merging arrays, which for large amounts of data could be an insurmountable obstacle. One solution to this drawback uses a linked list; analysis of this method is left as an exercise.

9.3.5 Radix Sort

Radix sort is a popular way of sorting used in everyday life. To sort library cards, we may create as many piles of cards as letters in the alphabet, each pile containing authors whose names start with the same letter. Then, each pile is sorted separately using the same method; namely, piles are created according to the second letter of the authors' names. This process continues until the number of times the piles are divided into smaller piles equals the number of letters of the longest name. This method is actually used when sorting mail in the post office, and it was used to sort 80-column cards of coding information in the early days of computers.

When sorting library cards, we proceed from left to right. This method can also be used for sorting mail because all ZIP codes have the same length. However, it may be inconvenient for sorting lists of integers because they may have an unequal number of digits. If applied, this method would sort the list [23 123 234 567 3] into the list [123 23 234 3 567]. To get around this problem, zeros can be added in front of each number to make them of equal length so that the list [023 123 234 567 003] is sorted into the list [003 023 123 234 567]. Another technique looks at each number as a string of bits so that all integers are of equal length. This approach will be discussed shortly. Still another way to sort integers is by proceeding right to left, and this method is discussed now.

When sorting integers, 10 piles numbered 0 through 9 are created, and initially, integers are put in a given pile according to their rightmost digit so that 93 is put in pile 3. Then, piles are combined and the process is repeated, this time with the second rightmost digit; in this case, 93 ends up on pile 9. The process ends after the leftmost digit of the longest number is processed. The algorithm can be summarized in the following pseudocode:

```
radixsort()
    for d = 1 to the position of the leftmost digit of longest number
        distribute all numbers among piles 0 through 9 according to the dth digit;
        put all integers on one list;
```

The key to obtaining a proper outcome is the way the 10 piles are implemented and then combined. For example, if these piles are implemented as stacks, then the integers 93, 63, 64, 94 are put on piles 3 and 4 (other piles being empty):

```
pile 3: 63 93
pile 4: 94 64
```

These piles are then combined into the list 63, 93, 94, 64. When sorting them according to the second rightmost digit, the piles are as follows:

```
pile 6: 64 63
pile 9: 94 93
```

and the resulting list is 64, 63, 94, 93. The processing is finished, but the result is an improperly sorted list.

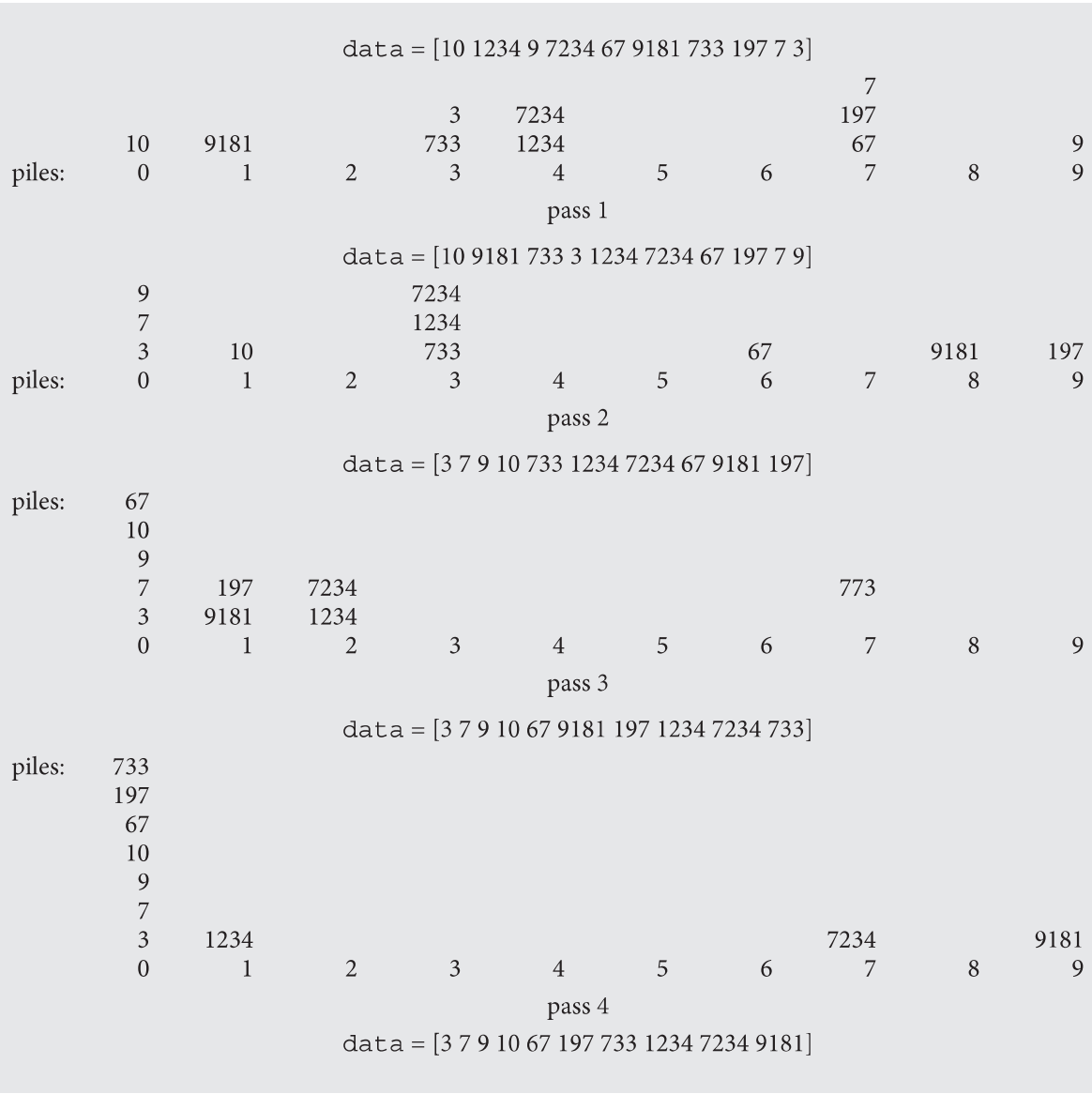
However, if piles are organized as queues, the relative order of elements on the list is retained. When integers are sorted according to the digit in position d, then within each pile, integers are sorted with regard to the part of the integer extending from digit 1 to $d-1$. For example, if after the third pass, pile 5 contains the integers 12534, 554, 3590, then this pile is ordered with respect to the two rightmost digits of each number. Figure 9.15 illustrates another example of radix sort.

Here is an implementation of radix sort:

```
void radixsort(long data[], int n) {
    register int d, j, k, factor;
    const int radix = 10;
    const int digits = 10; // the maximum number of digits for a long
    Queue<long> queues[radix];   // integer;
    for (d = 0, factor = 1; d < digits; factor *= radix, d++) {
        for (j = 0; j < n; j++)
            queues[(data[j] / factor) % radix].enqueue(data[j]);
        for (j = k = 0; j < radix; j++)
            while (!queues[j].empty())
                data[k++] = queues[j].dequeue();
    }
}
```

This algorithm does not rely on data comparison as did the previous sorting methods. For each integer from `data`, two operations are performed: division by a

FIGURE 9.15 Sorting the list 10, 1234, 9, 7234, 67, 9181, 733, 197, 7, 3 with radix sort.



`factor` to disregard digits following digit d being processed in the current pass and division modulo `radix` (equal to 10) to disregard all digits preceding d for a total of $2n$`digits` = $O(n)$ operations. The operation `div` can be used, which combines both `/` and `%`. In each pass, all integers are moved to piles and then back to `data` for a total of $2n$`digits` = $O(n)$ moves. The algorithm requires additional space for piles, which if implemented as linked lists, is equal to kn bytes depending on the size k of the pointers. Our implementation uses only `for` loops with counters; therefore, it requires the same amount of passes for each case: best, average, and worst. The body of the only `while` loop is always executed n times to dequeue integers from all queues.

The foregoing discussion treated integers as combinations of digits. But as already mentioned, they can be regarded as combinations of bits. This time, division

and division modulo are not appropriate, because for each pass, b bits for each number have to be extracted, where $1 \le b \le 31$ for 31-bit nonnegative integers. In this case, 2^b queues are required.

An implementation is as follows:

```
void bitRadixsort(long data[], int n, int b) {
    int pow2b = 1;
    pow2b <<= b;
    int i, j, k, pos = 0, mask = pow2b-1;
    int last = (bits % b == 0) ? (bits/b) : (bits/b + 1);
    Queue<long> *queues = new Queue<long>[pow2b];
    for (i = 0; i < last; i++) {
        for (j = 0; j < n; j++)
            queues[(data[j] & mask) >> pos].enqueue(data[j]);
        mask <<= b;
        pos = pos+b;
        for (j = k = 0; j < pow2b; j++)
            while (!queues[j].empty())
                data[k++] = queues[j].dequeue();
    }
}
```

where `bits` = 31. In the choice of b, two things have to be considered: the number of queues is proportional to b, which increases space requirements; on the other hand, the number of passes through the data decreases with the increase of b.

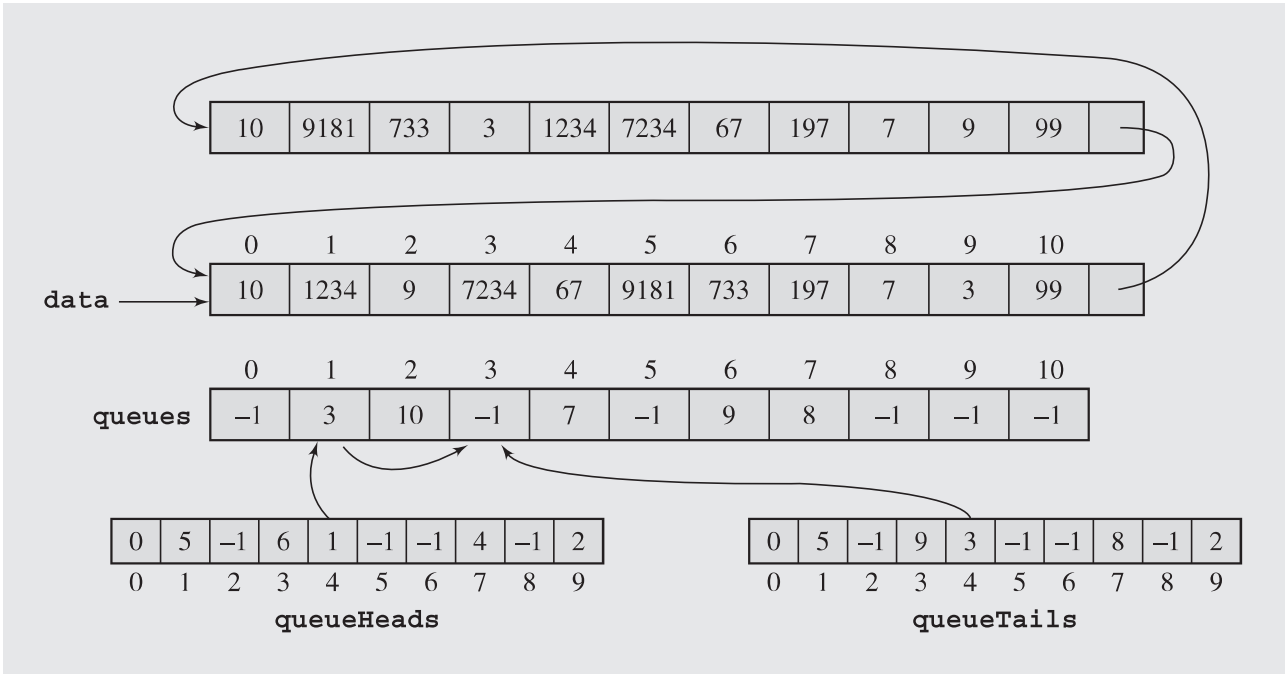
Division is replaced here by bitwise-and operation `&`. The variable `mask` has b bits set to 1 and the rest of them are set to 0. After each iteration this cluster of b bits is shifted to the left. For each number `data[j]`, the number is output in `queues[(data[j] & mask) >> pos]` where `pos` (= $b \cdot i$) indicates by how many positions to shift the result of bitwise-anding, so that the result is a number between 0 and 2^b-1, which is a range of possible values represented by b bits. In this way, $\lceil 31/b \rceil$ passes are needed for integers during which 2^b queues are used. Experiments indicate that b = 8 and 256 queues or b = 11 and 2048 queues are optimal choices. The run-times shown in Figure 9.19 are for b = 8. For b < 8, `bitRadixsort()` is slower than `radixsort()`. For b = 7, they are about the same and then `bitRadixsort()` becomes better. However, for a large b, space requirements become too exacting. For b = 16, a total of 65,536 queues are needed, and for b = 31, the requirement grows to 2,147,483,648, i.e., over two billion queues. Incidentally, for b = 31, radix sort turns into a more complicated version of the counting sort to be discussed in the next section.

Quicker operations cannot outweigh a larger number of moves: `bitRadixsort()` is much slower than `radixsort()` because the queues are implemented as linked lists, and for each item included in a particular queue, a new node has to be created and attached to the queue. For each item copied back to the original array, the node has to be detached from the queue and disposed of using `delete`. Although theoretically obtained performance $O(n)$ is truly impressive, it does not include operations on queues, and it hinges upon the efficiency of the queue implementation.

A better implementation is an array of size *n* for each queue, which requires creating these queues only once. The efficiency of the algorithm depends only on the number of exchanges (copying to and from queues). However, if radix *r* is a large number and a large amount of data has to be sorted, then this solution requires *r* queues of size *n,* and the number $(r + 1) \cdot n$ (original array included) may be unrealistically large.

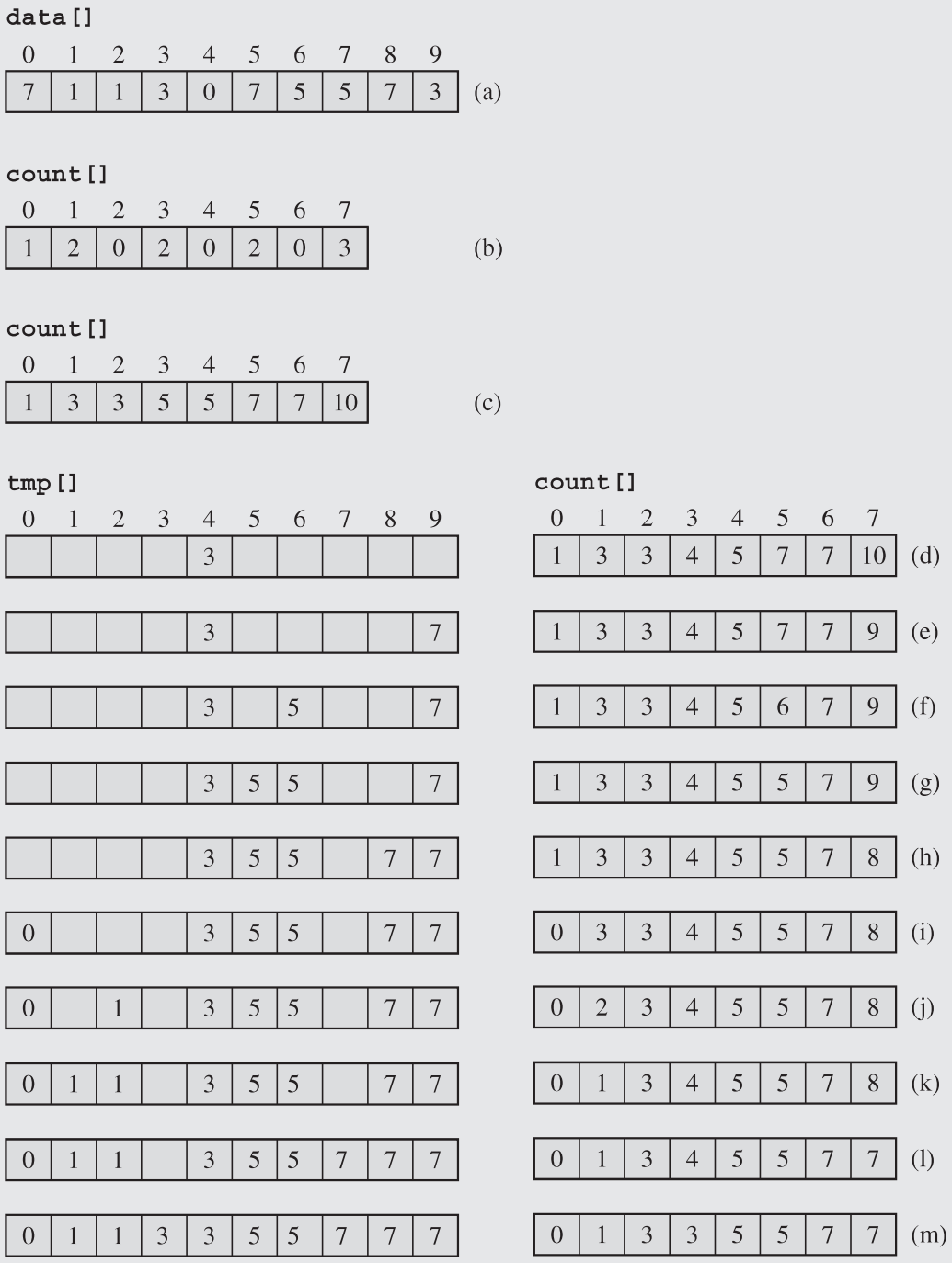
A better solution uses one integer array `queues` of size *n* representing linked lists of indexes of numbers belonging to particular queues. Cell *i* of the array `queueHeads` contains an index of the first number in `data` that belongs to this queue, whose *d*th digit is *i*. `queueTails[]` contains a position in `data` of the last number whose *d*th digit is *i*. Figure 9.16 illustrates the situation after the first pass, for $d = 1$. `queueHeads[4]` is 1, which means that the number in position 1 in `data`, 1234, is the first number found in `data` with 4 as the last digit. Cell `queues[1]` contains 3, which is an index of the next number in `data` with 4 as the last digit, 7234. Finally, `queues[3]` is –1 to indicate the end of the numbers meeting this condition.

FIGURE 9.16 An implementation of radix sort.



The next stage orders data according to information gathered in `queues`. It copies all the data from the original array to some temporary storage and then back to this array. To avoid the second copy, two arrays can be used, constituting a two-element circular linked list. After copying, the pointer to the list is moved to the next node, and the array in this node is treated as storage of numbers to be sorted. The improvement is significant because the new implementation runs several times faster than the implementation that uses queues (see Figure 9.19 later in this chapter).

FIGURE 9.17 An example of application of counting sort.



9.3.6 Counting Sort

The counting sort first counts the number of times each number occurs in the array `data[]` using an array of counters `count[]`, which is indexed with numbers from `data[]`. Then, counters indicating the number of integers $\leq i$ are added and stored in `count[i]`. In this way, `count[i]-1` indicates the home position of i in `data[]`. Making provision for the possibility of multiple occurrence of any number in `data[]`, the counting sort can be given by the following code:

```
void countingsort(long data[], const long n) {
      long i;
      long largest = data[0];
      long *tmp = new long[n];
      for (i = 1; i < n; i++)          // find the largest number
           if (largest < data[i])      // in data and create the array
                largest = data[i];     // of counters accordingly;
      unsigned long *count = new unsigned long[largest+1];
      for (i = 0; i <= largest; i++)
           count[i] = 0;
      for (i = 0; i < n; i++)          // count numbers in data[];
           count[data[i]]++;
      for (i = 1; i <= largest; i++) // count numbers ≤ i;
           count[i] = count[i-1] + count[i];
      for (i = n-1; i >= 0; i--) {  // put numbers in order in tmp[];
           tmp[count[data[i]]-1] = data[i];
           count[data[i]]--;
      }
      for (i = 0; i < n; i++)          // transfer numbers from tmp[]
           data[i] = tmp[i];           // to the original array;
}
```

An example of application of the algorithm is given in Figure 9.17. First, the largest number in `data[]` is determined, which is number 7, and then the array `count[]` is created. Next, the numbers in `data[]` are counted and counters are stored in `count[]`: number 0 occurs `count[0]` = 1 time in `data[]`, number 1 occurs `count[1]` = 2 times, number 2 is not used in `data[]` at all, etc. (Figure 9.17b). Next, `count[1]` is updated to contain cumulative counts: `count[1]` = 3 means that there are three 0s and 1s in `data[]`, `count[2]` = 3 means that there are three 0s, 1s, and 2s in `data[]`, and generally, `count[i]` = k means that `data[]` contains k numbers less than or equal to i (Figure 9.17c). Then, each number is placed in its proper position in a temporary array `tmp[]`, starting from the end of `data[]`: first, number `data[9]` = 3 is placed in `tmp[]`, `tmp[count[3]-1]` = `data[9]`, and `count[3]` is decremented (Figure 9.17d). Afterward, the penultimate number `data[8]` = 7 is placed in the temporary array, `tmp[count[7]-1]` = `data[8]` and `count[7]` is decremented (Figure 9.17e). The process continues until all numbers are placed in `tmp[]` (Figure 9.17f-m), after which the content of `tmp[]` is transferred to `data[]`.

Counting sort is linear in max (n, largest number in `data[]`). This means that even for small arrays it can be very expensive if at least one number in `data[]` is very large. For example, for the array [1, 2, 1, 10000], `count[]` would have 10001 cells and all of them would have to be processed. If it can be guaranteed that numbers in `data[]` are small, then counting sort is very efficient even for very large arrays. In order to harness counting sort, the sort can be used as part of radix sort as suggested in this pseudocode (cf. `bitRadixsort()`):

```
bitRadixsort3(data[],b)
    for i = 0 to last-1
        sort data[] with counting sort using a mask of b bits;
```

In this way, `count[]` would have at most 2^b+1 cells. This result is a significant improvement of radix sort's performance, which becomes comparable to that of quicksort. The performance becomes particularly good if parameters of this version of radix sort are fine-tuned to the size of cache memory (Rahman, and Raman 2001). Note that embedding counting sort in radix sort is possible because counting sort is stable, that is, it does not change the order of equal numbers and so partial order obtained in one pass is not disturbed by a next pass.

9.4 Sorting in the Standard Template Library

The STL provides many sorting functions, particularly in the library `<algorithm>`. The functions are implementing some of the sorting algorithms discussed in this chapter: quicksort, heap sort, and mergesort. A program in Figure 9.18 demonstrates these functions. For descriptions of the functions, see also Appendix B.

The first set of functions are partial sorting functions. The first version picks k = `middle - first` smallest elements from a container and puts them in order in the range [`first, middle`). For example,

```
partial_sort(v1.begin(),v1.begin()+3,v1.end());
```

takes the three smallest integers from the entire vector `v1` and puts them in the first three cells of the vector. The remaining integers are put in cells four through seven. In this way, `v1` = [1,4,3,6,7,2,5] is transformed into `v1` = [1,2,3,6,7,4,5]. This version orders elements in ascending order. An ordering relation can be changed if it is provided as the fourth parameter to the function call. For example,

```
partial_sort(v2.begin()+1,v2.begin()+4,v2.end(),greater<int>());
```

picks the largest three integers from the vector `v2` and puts them in positions two to four in descending order. The order of the remaining integers outside this range is not specified. This call transforms `v2` = [1,4,3,6,7,2,5] into `v2` = [1,7,6,5,3,2,4].

The third version of partial sorting takes k = `last1 - first1` or `last2 - first2`, whichever is smaller, first elements from the range [`first1,last1`), and writes them over elements in the range [`first2,last2`). The call

```
i3 = partial_sort_copy(v2.begin(),v2.begin()+4,v3.begin(),v3.end());
```

takes the first four integers in vector `v2` = [1,7,6,5], and puts them in ascending order in the first four cells of `v3` so that `v3` = [9,9,9,9,9,9] is transformed into

FIGURE 9.18 Demonstration of sorting functions.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional> // greater<>

using namespace std;

class Person {
public:
    Person(char *n = "", int a = 0) {
       name = strdup(n);
       age = a;
    }
    bool operator==(const Person& p) const {
       return strcmp(name,p.name) == 0;
    }
    bool operator<(const Person& p) const {
       return strcmp(name,p.name) < 0;
    }
private:
    char *name;
    int age;
    friend ostream& operator<< (ostream& out, const Person& p) {
        out << "(" << p.name << "," << p.age << ")";
        return out;
    }
};

bool f1(int n) {
    return n < 5;
}

template<class T>
void printVector(char *s, const vector<T>& v) {
    cout << s << " = (";
    if (v.size() == 0) {
        cout << ")\n";
        return;
    }
    for (vector<T>::const_iterator i = v.begin(); i! = v.end()-1; i++)
        cout << *i << ',';
```

Continues

FIGURE 9.18 *(continued)*

```
    cout << *i << ")\n";
}

iut main() {
    int a[] = {1,4,3,6,7,2,5};
    vector<int> v1(a,a+7), v2(a,a+7), v3(6,9), v4(6,9);
    vector<int>::iterator i1, i2, i3, i4;
    partial_sort(v1.begin(),v1.begin()+3,v1.end());
    printVector("v1",v1);                       // v1 = (1,2,3,6,7,4,5)
    partial_sort(v2.begin()+1,v2.begin()+4,v2.end(),greater<int>());
    printVector("v2",v2);                       // v2 = (1,7,6,5,3,2,4)
    i3 = partial_sort_copy(v2.begin(),v2.begin()+4,v3.begin(),v3.end());
    printVector("v3",v3);                       // v3 = (1,5,6,7,9,9)
    cout << *(i3-1) << ' ' << *i3 << endl;      // 7 9
    i4 = partial_sort_copy(v1.begin(),v1.begin()+4,v4.begin(),v4.end(),
                           greater<int>());
    printVector("v4",v4);                       // v4 = (6,3,2,1,9,9)
    cout << *(i4-1) << ' ' << *i4 << endl;      // 1 9
    i1 = partition(v1.begin(),v1.end(),f1);     // v1 = (1,2,3,4,7,6,5)
    printVector("v1",v1);
    cout << *(i1-1) << ' ' << *i1 << endl;      // 4 7
    i2 = partition(v2.begin(),v2.end(),bind2nd(less<int>(),5));
    printVector("v2",v2);                       // v2 = (1,4,2,3,5,6,7)
    cout << *(i2-1) << ' ' << *i2 << endl;      // 3 5
    sort(v1.begin(),v1.end());                  // v1 = (1,2,3,4,5,6,7)
    sort(v1.begin(),v1.end(),greater<int>()); // v1 = (7,6,5,4,3,2,1)

    vector<Person> pv1, pv2;
    for (int i = 0; i < 20; i++) {
        pv1.push_back(Person("Josie",60 - i));
        pv2.push_back(Person("Josie",60 - i));
    }
    sort(pv1.begin(),pv1.end());        // pv1 = ((Josie,41)...(Josie,60))
    stable_sort(pv2.begin(),pv2.end());// pv2 = ((Josie,60)...(Josie,41))

    vector<int> heap1, heap2, heap3(a,a+7), heap4(a,a+7);
    for (i = 1; i <= 7; i++) {
        heap1.push_back(i);
        push_heap(heap1.begin(),heap1.end());
        printVector("heap1",heap1);
    }
```

FIGURE 9.18 *(continued)*

```
    // heap1 = (1)
    // heap1 = (2,1)
    // heap1 = (3,1,2)
    // heap1 = (4,3,2,1)
    // heap1 = (5,4,2,1,3)
    // heap1 = (6,4,5,1,3,2)
    // heap1 = (7,4,6,1,3,2,5)
    sort_heap(heap1.begin(),heap1.end()); // heap1 = (1,2,3,4,5,6,7)
    for (i = 1; i <= 7; i++) {
        heap2.push_back(i);
        push_heap(heap2.begin(),heap2.end(),greater<int>());
        printVector("heap2",heap2);
    }
    // heap2 = (1)
    // heap2 = (1,2)
    // heap2 = (1,2,3)
    // heap2 = (1,2,3,4)
    // heap2 = (1,2,3,4,5)
    // heap2 = (1,2,3,4,5,6)
    // heap2 = (1,2,3,4,5,6,7)
    sort_heap(heap2.begin(),heap2.end(),greater<int>());
    printVector("heap2",heap2);              // heap2 = (7,6,5,4,3,2,1)
    make_heap(heap3.begin(),heap3.end());    // heap3 = (7,6,5,1,4,2,3)
    sort_heap(heap3.begin(),heap3.end());    // heap3 = (1,2,3,4,5,6,7)
    make_heap(heap4.begin(),heap4.end(),greater<int>());
    printVector("heap4",heap4);              // heap4 = (1,4,2,6,7,3,5)
    sort_heap(heap4.begin(),heap4.end(),greater<int>());
    printVector("heap4",heap4);              // heap4 = (7,6,5,4,3,2,1)
    return 0;
}
```

`v3` = [1,5,6,7,9,9]. As an extra, an iterator is returned that refers to the first position after the last copied number. The fourth version of the partial sorting function is similar to the third, but it allows for providing a relation with which elements are sorted. The program in Figure 9.18 also demonstrates the partition function, which orders two ranges with respect to one another by putting in the first range numbers for which a one-argument Boolean function is true; numbers for which the function is false are in the second range. The call

```
i1 = partition(v1.begin(),v1.end(),f1);
```

uses an explicitly defined function `f1()` and transforms `v1` = [1,2,3,6,7,4,5] into `v1` = [1,2,3,4,7,6,5] by putting all numbers less than 5 in front of numbers that are not less than 5. The call

```
i2 = partition(v2.begin(),v2.end(),bind2nd(less<int>(),5));
```

accomplished the same for `v2` with the built-in functional `bind2nd`, which binds the second argument of the operator < to 5, effectively generating a function that works just like `f1()`.

Probably the most useful is the function `sort()` that implements quicksort. The call

```
sort(v1.begin(),v1.end());
```

transforms `v1` = [1,2,3,6,7,4,5] into fully ordered `v1` = [1,2,3,4,5,6,7]. The second version of `sort()` allows for specifying any ordering relation.

The STL also provides stable sorting functions. A sorting algorithm is said to be *stable* if equal keys remain in the same relative order in output as they are initially (that is, if `data[i]` equals `data[j]` for $i < j$ and the ith element ends up in the kth position and the jth element in the mth position, then $k < m$). To see the difference between sorting and stable sorting, consider a vector of objects of type `Person`. The definition of `operator<` orders the `Person` objects by name without taking age into account. Therefore, two objects with the same name but with different ages are considered equal, just as in the definition of `operator==`, although this operator is not used in sorting; only the less than operator is. After creating two equal vectors, `pv1` and `pv2`, both equal to [("Josie,"60) . . . ("Josie,"41)], we see that `sort()` transforms this vector into [("Josie,"41) . . . ("Josie,"60)], but `stable_sort()` leaves it intact by retaining the relative order of equal objects. The feat is possible by using merge-sort in the stable sorting routine. Note that, for small numbers of objects, `sort()` is also stable because, for a small number of elements, insertion sort is used, not quicksort (see `quicksort2()` at the end of Section 9.3.3).

9.5 Concluding Remarks

Figure 9.19 compares the run times for different sorting algorithms and different numbers of integers being sorted. They were all run on a laptop. At each stage, the number of long integers has been doubled to see the factors by which the run times raise. These factors are included in each column except for the first three columns and are shown along with the run times. The factors are rounded to one decimal place, whereas run times (in seconds) are rounded to three decimal places. For example, heap sort required 0.040 seconds to sort an array of 25,000 integers in ascending order and 0.087 seconds to sort 50,000 integers, also in ascending order. Doubling the number of data is associated with the increase of run time by a factor of 0.087/0.040 = 2.2, and the number 2.2 follows 0.087 in the fourth column.

The table in Figure 9.19 indicates that the run time for elementary sorting methods, which are squared algorithms, grows approximately by a factor of 4 after the amount of data is doubled, whereas the same factor for nonelementary methods, whose complexity is $O(n \lg n)$, is approximately 2. This is also true for the four implementations of radix sort, whose complexity equals $2n$`digits` or $2n$`bits`. The table also shows that counting sort is the fastest algorithm among all sorting methods; however, it is limited in that it can only sort nonnegative integers, and so do all four versions of radix sort. Among the remaining algorithms, Shell sort, merge sort, and quicksort perform the best. Most of the time, they run at least twice as fast as any other algorithm.