13 Sorting and Searching

Overview

This chapter discusses several standard algorithms for sorting, i.e., putting a number of values in order. It also discusses the binary search algorithm for finding a particular value quickly in an array of sorted values. The algorithms described here can be useful in various situations. They should also help you become more comfortable with logic involving arrays. These methods would go in a utilities class of methods for Comparable objects, such as the CompOp class of Listing 7.2. For this chapter you need a solid understanding of arrays (Chapter Seven).

- Sections 13.1-13.2 discuss two basic elementary algorithms for sorting, the SelectionSort and the InsertionSort.
- Section 13.3 presents the binary search algorithm and big-oh analysis, which provides a way of comparing the speed of two algorithms.
- Sections 13.4-13.5 introduce two recursive algorithms for sorting, the QuickSort and the MergeSort, which execute much faster than the elementary algorithms when you have more than a few hundred values to sort.
- Sections 13.6 goes further with big-oh analysis.
- Section 13.7 presents several additional sorting algorithms -- the bucket sort, the radix sort, and the shell sort.

13.1 The SelectionSort Algorithm For Comparable Objects

When you have hundreds of Comparable values stored in an array, you will often find it useful to keep them in sorted order from lowest to highest, which is ascending order. To be precise, **ascending order** means that there is no case in which one element is larger than the one after it -- if y is listed after x, then `x.CompareTo(y) <= 0`. Sometimes you prefer to store them from highest to lowest, which is **descending order** (no element is smaller than the one after it). The usual sorting problem is to write an independent method that puts an array into ascending order.

Finding the smallest in a partially-filled array

As a warmup to a sorting algorithm, look at a simpler problem for an array of Comparable objects: How would you find the smallest value in a given range of values in an array?

Say the parameters of the method are the array `item`, the first index `start`, and the ending index end, where the values to search are from `item[start]` up through `item[end]`; assume `start <= end`. You look at the first value: `smallestSoFar = item[start]`. Then you go through the rest of the values in the array one at a time. Whenever the current value is smaller than `smallestSoFar`, you store it in `smallestSoFar`. So an independent class method to do this would be as follows:

```
public static Comparable findMinimum (Comparable[ ] item,
                                      int start, int end)
{  Comparable smallestSoFar = item[start];
   for (int k = start + 1;  k <= end;  k++)
   {  if (item[k].compareTo (smallestSoFar) < 0)
         smallestSoFar = item[k];
   }
   return smallestSoFar;
}  //======================
```

Say you apply this logic to the array in Figure 13.1 with `start` being 2 and `end` being 5. The figure indicates the values by decimal numbers to make this example clearer. You begin by noting the smallest so far is 5.0, which is the value at index 2. Next you go through each index value from index 3 on up, stopping after processing index value 5:

- You compare `item[3]` to 5.0 but make no change, because `item[3]` is not smaller than 5.0.
- You compare `item[4]` to 5.0 and assign that value 3.0 to be the smallest so far, because `item[4]` is smaller than 5.0.
- You compare `item[5]` to 3.0 but make no change, because `item[5]` is not smaller than 3.0. The end result is that you return the value 3.0 from the method.

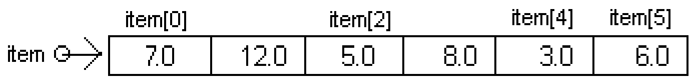

Figure 13.1 An array with six decimal values

Naturally this logic only works if the array has non-null Comparable values in components 2 through 5. Now try a mild modification of this logic: How would you find the smallest value and swap it with the value at index `start`? It is not enough to find the smallest; you have to find the index where the smallest is stored. The coding is very similar to that of `findMinimum`:

```
int indexSmallest = start;
for (int k = start + 1;  k <= end;  k++)
{  if (item[k].compareTo (item[indexSmallest]) < 0)
      indexSmallest = k;
}
```

Then you can swap the value at that index with the value at index `start`. For instance, for the array in Figure 13.1, you would find that the index of the smallest is 4, because that is where the 3.0 is. So you swap the value at index 4 with the value at index `start`:

```
Comparable saved = item[start];
item[start] = item[indexSmallest];
item[indexSmallest] = saved;
```

The SelectionSort Algorithm

With this warmup, you can look at a standard method of putting all array values in ascending order. This algorithm is the **SelectionSort Algorithm**. The plan is to select the smallest of all the values and swap it into component 0. Then select the smallest of all the values from index 1 on up and swap it into component 1. At this point you have the two smallest values of all, in ascending order in the first two components of the array.

You next select the smallest of all the values from index 2 on up and swap it into component 2. Now you have the three smallest values of all, in ascending order in the first three components of the array. Next you select the smallest of all the values from index 3 on up and swap it into component 3. This continues until you come to the end of the values in the array. Then all of the values are in ascending order.

Figure 13.2 illustrates the sequence of steps on the array with six values, at indexes 0 through 5, from Figure 13.1. The process only needs five steps, because once the first five are at the right index, the last one must be (do you see why?). On Step 5a, the smallest of the two values is already at the right spot, so swapping it has no effect.

The indexes of the six components are:	0	1	2	3	4	5
And the values stored in the array are initially:	7	12	5	8	3	6
Step 1a: Find the location of the smallest at index 0...5:	?	?	?	?		?
Step 1b: Swap it with the value at index 0:	**3**	12	5	8	**7**	6
Step 2a: Find the location of the smallest at index 1...5:	"	?		?	?	?
Step 2b: Swap it with the value at index 1:	"	**5**	**12**	8	7	6
Step 3a: Find the location of the smallest at index 2...5:	"	"	?	?	?	
Step 3b: Swap it with the value at index 2:	"	"	**6**	8	7	**12**
Step 4a: Find the location of the smallest at index 3...5:	"	"	"	?		?
Step 4b: Swap it with the value at index 3:	"	"	"	**7**	**8**	12
Step 5a: Find the location of the smallest at index 4...5:	"	"	"	"		?
Step 5b: Swap it with the value at index 4:	"	"	"	"	**8**	12

Figure 13.2 Sequence of operations for the SelectionSort

This algorithm can be expressed in just a few statements, using the coding previously developed. We will sort a partially-filled array of Comparable values, i.e., we have a `size` int value and we are only concerned with the array values indexed 0 up to but not including `size`. We call `swapMinToFront` for `start` having the values 0, 1, 2, 3, etc., stopping when `start` is `size-1`. The logic is in Listing 13.1.

Listing 13.1 The selectionSort method for a partially-filled array, in CompOp

```
/** Precondition:  size <= item.length; item[0]...item[size-1]
 *  are non-null values all Comparable to each other.
 *  Postcondition: The array contains the values it initially
 *  had but with item[0]...item[size-1] in ascending order. */


public static void selectionSort (Comparable[] item, int size)
{  for (int k = 0;  k < size - 1;  k++)
      swapMinToFront (item, k, size - 1);
}  //=======================


private static void swapMinToFront (Comparable[] item,
                                    int start, int end)
{  int indexSmallest = start;
   for (int k = start + 1;  k <= end;  k++)
   {  if (item[k].compareTo (item[indexSmallest]) < 0)
         indexSmallest = k;
   }
   Comparable saved = item[start];
   item[start] = item[indexSmallest];
   item[indexSmallest] = saved;
}  //=======================
```

This method might be called from the WorkerList class of Section 7.6 (whose instance variables are a partially-filled array `itsItem` of Worker objects and an int value `itsSize` saying how many values in `itsItem` are useable) using this statement:

```
CompOp.selectionSort (this.itsItem, this.itsSize);
```

An example of an independent class method that calls on the `selectionSort` method to sort 3000 randomly-chosen decimal numbers, then prints the sorting time in seconds and returns the median value, is the following. It uses the Double methods described in Section 11.4, though all you need to know about them for this chapter is the following:

- `new Double(x)` creates an object with an instance variable storing the double x,
- `someDouble.doubleValue()` returns the double value of the Double object, and
- the Double class implements the Comparable interface, having `compareTo`.

```
public static double median()        // independent
{  final int numToSort = 3000;
   Double[] item = new Double [numToSort];
   for (int k = 0;  k < numToSort;  k++)
      item[k] = new Double (Math.random());
   long t = System.currentTimeMillis();
   selectionSort (item, numToSort);
   System.out.println ((System.currentTimeMillis() - t)/1000);
   return item[numToSort / 2].doubleValue();
}  //======================
```

Exercise 13.1 Revise Listing 13.1 to perform a SelectionSort on a null-terminated array: You have only the `item` array parameter, no `size`. You are to sort all values up to the first one that is null. Precondition: At least one component contains null.
Exercise 13.2 Revise Listing 13.1 to put the values in descending order.
Exercise 13.3* Rewrite Listing 13.1 to find the largest value each time and swap it to the rear of the array.
Exercise 13.4* Rewrite Listing 13.1 to omit swapping when the smallest is already at the front. Does this speed up or slow down the execution?

13.2 The InsertionSort Algorithm For Comparable Objects

The InsertionSort is another standard sorting algorithm. As a warmup, start with this problem: If you know that the first `m` values in an array are already in ascending order, but the next one (at `item[m]`) is probably out of order, how would you get them all in order? The accompanying design block is a plan for the solution.

DESIGN to re-order values

```
1. Set the value from item[m] aside, thereby leaving an empty spot in the array.
2. Compare the value before the empty spot with the value you set aside.
   If the value before the empty spot is larger, then...
        2a. Move that value into the empty spot, so where it came from
            is now empty.
        2b. Repeat from Step 2.
3. Put the value you set aside in the empty spot.
```

This logic is defective: What if all of the values are larger than the one you set aside? You have to stop when the empty spot is at the very front of the array, i.e., `item[0]`. Making this adjustment, the following method solves the problem.

```
private static void insertInOrder (Comparable[ ] item, int m)
{  Comparable save = item[m];
   for (;  m > 0 && item[m - 1].compareTo (save) > 0;  m--)
      item[m] = item[m - 1];
   item[m] = save;
}  //======================
```

The InsertionSort Algorithm

Now that you have had a warmup, you can look at the second standard algorithm to put the array values in ascending order. This algorithm is the **InsertionSort Algorithm**. The plan is as follows (see the illustration in Figure 13.3):

1. Put the first two values in order.
2. Insert the third value in its proper place in the sequence formed by the first two.
3. Insert the fourth value in its proper place in the sequence formed by the first three.
4. Insert the fifth value in its proper place in the sequence formed by the first four.
5. Keep this up until you have them all in order.

Suppose the values stored in the array are initially:	12	7	8	5	13	6	9
Step 1: Put the first two values in order:	7	12	"	"	"	"	"
Step 2: Insert the third in order among the first two:	7	**8**	12	"	"	"	"
Step 3: Insert the fourth in order among the first three:	**5**	7	8	12	"	"	"
Step 4: Insert the fifth in order among the first four:	5	7	8	12	**13**	"	"
Step 5: Insert the sixth in order among the first five:	5	**6**	7	8	12	13	"
Step 6: Insert the seventh in order among the first six:	5	6	7	8	**9**	12	13

Figure 13.3 Sequence of operations for the InsertionSort

You will have noticed that the logic needed for each of Steps 2, 3, 4, etc. is in the `insertInOrder` method just developed. In fact, it can be used for Step 1 as well (calling the method with `m == 1`). The complete InsertionSort algorithm is coded in Listing 13.2. The precondition and postcondition are the same as for Listing 13.1.

Listing 13.2 The insertionSort method for a partially-filled array, in CompOp

```
/** Precondition:  size <= item.length; item[0]...item[size-1]
 *  are non-null values all Comparable to each other.
 *  Postcondition: The array contains the values it initially
 *  had but with item[0]...item[size-1] in ascending order. */


public static void insertionSort (Comparable[] item, int size)
{  for (int k = 1;  k < size;  k++)
      insertInOrder (item, k);
}  //=======================


private static void insertInOrder (Comparable[] item, int m)
{  Comparable save = item[m];
   for (;  m > 0 && item[m - 1].compareTo (save) > 0;  m--)
      item[m] = item[m - 1];
   item[m] = save;
}  //=======================
```

Some people want to put at the beginning of the `insertionSort` method a test to save time when the size is 0, beginning the coding with the phrase `if (size > 0)`. However, that would violate the **Principle of Rara Avis**: It is less efficient to speed up execution in cases that rarely occur (i.e., cases that are "rare birds", thus the "rara avis") if that slows down execution in the vast majority of cases.

Contrasting the two sorting algorithms

At each call of `insertInOrder`, the list of values is actually two lists: The "bad list", which is all the unsorted values indexed `k` and higher, versus the "good list" of sorted values at indexes `0..k-1`. Each call of `insertInOrder` increases by 1 the number of items in the good list and consequently decreases by 1 the number of items in the bad list. When the InsertionSort finishes, the entire list is the good list, i.e., it is sorted.

For the SelectionSort, the bad list is also the unsorted values indexed `k` and higher, and the good list of sorted values is at indexes `0..k-1`. Each call of `swapMinToFront` increases by 1 the number of items in the good list and consequently decreases by 1 the number of items in the bad list. When the SelectionSort finishes, the entire list is the good list, i.e., it is sorted.

Both algorithms gradually transform the bad into the good, one value at a time. The way they transform differs, but the overall effect is the same: What was originally all bad (unsorted) is at the end all good (sorted).

The difference between them can be described this way: The SelectionSort logic slowly selects a value (the smallest) from the bad list so it can quickly put it on the end of the good list (by a simple swap). The InsertionSort logic quickly selects a value (the first one available) from the bad list so it must slowly put it where it goes within the good list (by a careful insertion process).

Loop invariants verify that the algorithms work right

A **loop invariant** is a condition that is true every time the continuation condition of a looping statement is evaluated. For these two sorting algorithms, a loop invariant tells what it means to be a "good" list:

Loop invariant for the main loop of `insertionSort` (any time at which the loop condition `k < size` is evaluated): The values in item[0]...item[k-1] are in ascending order and are the same values that were originally in item[0]...item[k-1].

How do we know this condition is in fact true every time `k < size` is evaluated? It depends on two facts that you can easily check out:

1. The first time `k < size` is evaluated, `k` is 1, so item[0]...item[k-1] contain only one value, so they are by definition in ascending order.
2. If at any point when `k < size` is evaluated, item[0]...item[k-1] are in ascending order and are the values that were originally there, then one iteration of the loop shifts the last few values up by one component and inserts `item[k]` into the place that was vacated by the shift, immediately after the rightmost value that is less than or equal to `item[k]`. So item[0]..item[k] are now in ascending order and are the values that were originally in components 0 through `k`. Then, just before testing `k < size` again, k is incremented, which means that the invariant condition is again true: item[0]...item[k-1] are the original first `k` values in ascending order.

Once you see that both of those assertions are true, you should be able to see that together they imply that the loop invariant is true when the loop terminates: item[0]...item[k-1] are in ascending order and are the original first `k` values. But at that point, `k` is equal to `size`, which means that item[0]...item[size-1] are in ascending order and are the original `size` values. Conclusion: The `insertionSort` coding sorts the values in the partially-filled array.

The same pattern of reasoning can be applied to verify that the `selectionSort` coding in the earlier Listing 13.1 actually sorts the values given it:

Loop invariant for the main loop of `selectionSort` (any time at which the loop condition `k < size-1` is evaluated): The values in item[0]...item[k-1] are in ascending order and are the `k` smallest values that were originally in item[0]...item[size-1].

How do we know that condition is in fact true every time `k < size-1` is evaluated? It depends on two facts that you can easily check out:

1. The first time `k < size-1` is evaluated, `k` is zero, so item[0]...item[k-1] contain no values at all, so they are the 0 smallest values in ascending order (vacuously).
2. If at any point when `k < size-1` is evaluated, item[0]...item[k-1] are the `k` smallest of the original values in ascending order, then one iteration of the loop finds the (k+1)th smallest of the original values and puts it after the k smallest values. So item[0]...item[k] are now in ascending order and are the `k+1` smallest values that were originally in the array. Then, just before testing `k < size-1` again, k is incremented, which means that the invariant condition is again true: item[0]...item[k-1] are the `k` smallest of the original values in ascending order.

Once you see that both of those assertions are true, you should be able to see that together they imply that the loop invariant is true when the loop terminates: item[0]...item[k-1] are the `k` smallest of the original values in ascending order. But at that point, `k` is equal to `size-1`, which means that item[0]...item[size-2] are in ascending order and are all but the largest of the original values. Conclusion: The `selectionSort` algorithm sorts the values in the partially-filled array.

As a general principle, when you have a complex looping algorithm and you want to verify that it produces a given result under any and all conditions, proceed as follows: Find a condition that is trivially true the first time the loop condition is evaluated, and is "incrementally true" for the loop (i.e., if true at the time of one evaluation, it is true at the time of the next evaluation), so you can **inductively deduce** that it will be true when the loop terminates. If you have chosen the condition well, its truth when the loop terminates will be an obvious proof that the loop produces the required result.

A game example You play a two-person game in which you are to lay out more than 40 counters and your opponent has the first move. Each move is to take 1, 2, or 3 counters. The person who takes the last counter wins. So you are executing the loop `while (counters left) {opponent moves; you move;}` whose long-term objective is to take the last counter. You can assure this by establishing a short-term objective on each iteration that your move leaves a multiple of 4 counters. This is the loop invariant. So initially you choose to lay out 44 or 48 or 52... counters. For each of your moves, you leave 4 fewer counters than you left on the turn before. Then you will win.

Exercise 13.5 Rewrite `insertInOrder` in Listing 13.2 to check whether the value to be set aside is in fact smaller than the one in front of it; if not, do not set it aside. Does this speed up or slow down the algorithm?
Exercise 13.6 Revise Listing 13.2 to perform an InsertionSort on a null-terminated array: You have only the `item` parameter, no `size`. You are to sort all values up to the first one that is null. Precondition: At least one component contains null.
Exercise 13.7 Revise Listing 13.2 to perform an InsertionSort on an array of doubles.
Exercise 13.8 Revise Listing 13.2 to allow for null values scattered throughout the array. A null value is to be considered larger than any non-null value.
Exercise 13.9* Rewrite Listing 13.2 to perform the insertion from the other end of the array. That is, put the top 2 in order, then the top 3, then the top 4, etc.
Exercise 13.10* The loop condition in `insertInOrder` makes two tests each time through. Rewrite the method to execute faster by making only one test on each iteration, as follows: If `item[0]` is not larger than `save`, have a faster loop to insert `save` where it goes, otherwise have a separate loop to insert `save` at the front of the array.

13.3 Big-oh And Binary Search

Why have two or more different sorting methods? Because one may be better for one purpose and another may be better for a different purpose. We will now look at the execution time of the two elementary algorithms you have seen so far.

Counting the comparisons made for elementary sorting

When you study the logic in Listing 13.2, you should be able to see that a call of `insertInOrder` with `k` as the second parameter could make anywhere from 1 to `k` comparisons. So if `N` denotes the total number of items to be sorted, the InsertionSort could make as many as `1 + 2 + 3 + ... + (N-1)` comparisons of data altogether before it gets them all in order. That adds up to `(N-1)*N/2`, using basic counting methods, i.e., almost half of N-squared. In effect, each of the `N` items could be compared with each of the other `(N-1)` items before the sorting is done.

If you look back at the logic in the earlier Listing 13.1, you will see that the first time you call `swapMinToFront`, when k is zero, you will make `N-1` comparisons. The second time you will make `N-2` comparisons. This continues until k is the index of the next-to-last value, which requires only one comparison. So the total number of comparisons is `1 + 2 + 3 +...+(N-1)`, i.e., `(N-1)*N/2`.

The `(N-1)*N/2` comparisons that the InsertionSort could make is the **worst case**; the average should be somewhere around one-quarter of N-squared. The SelectionSort always takes almost one-half of N-squared comparisons, but it usually has much less movement of data than the InsertionSort. Which is faster depends on the time for a comparison versus the time for a movement of data, but it is usually the InsertionSort.

Both the InsertionSort and the SelectionSort are called **elementary sorting algorithms**. "Elementary" means that the number of comparisons made in the process of sorting N items is on average a constant multiple of N-squared. The multiple for InsertionSort is ¼ and the multiple for SelectionSort is ½. There are other elementary sorting algorithms; the first one invented for computer programs was probably the one called the BubbleSort. But it executes more slowly than the two described here. A SelectionSort variation, based on finding both the maximum and the minimum on each pass through the data, is described in the appendix of major programming projects; its multiple is 3/8.

The technical way of saying this is that **the big-oh behavior of elementary sorts is N-squared**, where N is the number of items to be sorted. It means that, if it takes X amount of time to sort a certain number of items, then it takes roughly 4X amount of time to sort twice as many items, and it takes roughly 100X amount of time to sort ten times as many items, at least if X is fairly large. In general, <u>the amount of time to do the job is roughly proportional to the square of the number of items to be processed, for large values of N</u>.

To see this, suppose you have a processor that takes 1 second to sort 1000 items. That 1 second is what you need for roughly M times 1 million comparison operations, where M is the multiplier for the elementary sort (M == ¼ for the InsertionSort, M == ½ for the SelectionSort) and 1 million is the square of 1000. To sort 2000 items requires M times 4 million comparisons, which is four times as long as for sorting 1000 items. To sort 10,000 items requires M times 100 million comparisons, which is 100 times as long as for sorting 1000 items.

In particular, it would take a thousand-squared seconds to sort one million items, which is over eleven days. This is not practical. The next two sections discuss far faster algorithms for sorting, on the order of big-oh of N times log(N). But first we look at an algorithm for searching an array and its execution time.

Binary Search

A key algorithm for searching an array for a target value when the array is already sorted in ascending order is called Binary Search. The logic of a **Binary Search Algorithm** to find whether the target value is present is: The target value, if it is in the array, must be in the range from the `lowerBound` of 0 to the `upperBound` of `size-1`, inclusive. First look at the value at the middle component `item[midPoint]`. If that is less than the target value, the target can only be above that component, because the values are sorted in ascending order; so the revised `lowerBound` is `midPoint+1`. On the other hand, if `item[midPoint]` is not less than the target value, the target must be in the range from `lowerBound` to `midPoint`, inclusive; so the revised `upperBound` is `midPoint`.

Whichever of the two cases apply, you have cut the number of possibilities about in half. This process can be repeated until `lowerBound` is equal to `upperBound`. At that point, the target value must be in the only component in that range, if it is in the array at all.

Figure 13.4 illustrates a search for the value 40 in an ordered array of 16 values. Initially `lowerBound` is 0 and `upperBound` is 15 (listed in the first two columns), so `midPoint` is 7 (in the third column). 40 is greater than `item[7]` (boldfaced in the figure) so we set `lowerBound` to 8. Now `midPoint` is 11 and 40 is not greater than `item[11]`, so we set `upperBound` to 11. This continues until we isolate the value in `item[10]`.

lB	uB	m	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	15	7	20	22	24	26	28	30	32	**34**	36	38	40	42	44	46	48	50
8	15	11									36	38	40	**42**	44	46	48	50
8	11	9									36	**38**	40	42				
10	11	10											**40**	42				
10	10												40					

Figure 13.4 Finding the value 40 in an array of 16 values

This logic expressed in Java is in Listing 13.3. The coding includes an extra test to make sure `size` is positive before beginning the subdivision process.

Listing 13.3 The binarySearch method for an ordered partially-filled array

```
/** Precondition:  item[0]...item[size-1] are all non-null,
 *  in ascending order, and Comparable with target.
 *  Returns: whether target is one of the array values. */


public static boolean binarySearch (Comparable[] item,
                                   int size, Comparable target)
{  if (size <= 0)                                         //1
      return false;                                       //2
   int lowerBound = 0;                                    //3
   int upperBound = size - 1;                             //4
   while (lowerBound < upperBound)                        //5
   {  int midPoint = (lowerBound + upperBound) / 2;       //6
      if (item[midPoint].compareTo (target) < 0)          //7
         lowerBound = midPoint + 1;                       //8
      else                                                //9
         upperBound = midPoint;                           //10
   }                                                      //11
   return item[lowerBound].equals (target);               //12
}  //=======================
```

Counting the comparisons made for BinarySearch

If the number of items to be searched is 1000, then you only need eleven comparisons to find out whether the target value is in the array. This is because, when you start from 1000 and repeatedly divide by 2, you reduce it to just one possibility in only 10 iterations; the eleventh comparison is to find out whether that one possibility is the target value.

The number of times you divide a number by 2 to get it down to 1 is its **logarithm base 2** (more precisely, it is the logarithm after rounding up if needed). Call this rounded-up number **`log2(N)`**. `log2(1000)` is 10, so binary search of a thousand items only requires 11 comparisons. Similarly, `log2(1,000,000)` is 20, so binary search of a million items only requires 21 comparisons. And `log2(1,000,000,000)` is 30, so binary search of a billion items only requires 31 comparisons. Note that in Java, log2(N) is the same as Math.ceil (Math.log(N) / Math.log(2)).

The big-oh

In general, if N is the number of items to be searched, then the Binary Search Algorithm requires `log2(N)+1` comparisons to find out whether a particular target value is there. The technical way of saying this is that **the big-oh behavior of binary search is log(N)**, where N is the number of items to be searched. It means that the amount of time to do the job is roughly proportional to the logarithm of the number of items to be processed, for large values of N. A useful loop invariant for the binarySearch method is the following:

Loop invariant for the one loop in binarySearch (any time at which the loop condition `lowerBound < upperBound` is evaluated): The target value is in item[lowerBound]...item[upperBound] or else is not in the array at all.

By contrast, the **sequential search** algorithm that you have seen many times before looks at potentially every element in the list. The **big-oh behavior of sequential search is N**: the amount of time to do the job is roughly proportional to the number of items to be processed, for large values of N.

Exercise 13.11 The `binarySearch` logic divides the possibilities up into two parts, but they are not always the same size. When is one part larger by one item? Which part of the array is larger in that case, the front or the rear half?
Exercise 13.12 Explain what Exceptions could be thrown, and when, if the initial statement testing `size` were omitted from the coding of `binarySearch`.
Exercise 13.13 A method makes a copy of an array parameter by creating a new array of the same size and assigning each value in the given array to the new array. What is the big-oh execution time for this method?
Exercise 13.14 (harder) In the `binarySearch` method, what would be the consequence of rounding off `midPoint` in the other direction from what is done in Listing 13.3 (i.e., what Exceptions could be thrown or other problems occur)?
Exercise 13.15* Rewrite the `binarySearch` method to check to see whether `item[midPoint]` actually equals the target value each time and, if so, to stop the process early. Then explain why this slows execution on average.
Exercise 13.16* Revise the `binarySearch` method to return an int value: Return the index where the target was found, except return a negative int `-n-1` if the target is not in the array (n is the index where target should be inserted to keep all values in order).
Exercise 13.17** Write a `binarySearch` method for a null-terminated array with execution time big-oh of log(L) where L is the length of the array.
Exercise 13.18** Write out logical reasoning to verify that the loop invariant for `binarySearch` is trivially true at the first test and incrementally true for each iteration.

Part B Enrichment And Reinforcement

13.4 The Recursive QuickSort Algorithm For Comparable Objects

People have made attempts to improve on the speed of the elementary sorting algorithms of SelectionSort and InsertionSort. One idea is the **QuickSort** logic, which the accompanying design block describes, assuming you create a QuickSorter object whose job it is to sort the elements of an array.

DESIGN for the sorting logic of a QuickSorter object
If you have at least two elements to sort in ascending order, then...

1. Choose one of the elements of the array at random; call it the `pivot`.
2. Move everything smaller than the `pivot` towards the front of the array.
3. Move everything larger than the `pivot` towards the rear of the array.
4. Put the `pivot` in the component between those two groups of elements.
5. Sort the values before the `pivot`.
6. Sort the values after the `pivot`.

Why this is almost always faster

Let us compare this logic with that of the InsertionSort. Say you find `insertionSort` takes 800 milliseconds to sort 2000 items using a particular processor. Since it makes 1999 passes through the data, calling `insertInOrder` each time, the calls of `insertInOrder` take 0.4 milliseconds each on average. The average call of `insertInOrder` inserts a value into about 1000 items and makes about 500 comparisons to do so. Therefore, since steps 1 through 4 of the QuickSort logic make 1999 comparisons, that must take under 2 milliseconds.

Suppose that steps 5 and 6 called `insertionSort` instead of using another QuickSort logic, once for the half that are smaller than the pivot and once for the half that are larger. Then each call would take about 200 milliseconds, one quarter of the 800 milliseconds for sorting 2000, since the execution time of `insertionSort` is roughly proportional to the square of the number of items sorted. So altogether the QuickSort logic would take 2+200+200 = 402 milliseconds instead of 800 milliseconds. And that is assuming it switches to the `insertionSort` logic for steps 5 and 6. If it continued with the QuickSort logic, the improvement in speed would be far greater.

You probably noticed a big IF in that logic: If the pivot turns out to divide the values roughly in half, it almost doubles the speed. But the pivot is chosen at random, so it may be very close to one end or the other of the range of values. That would make the execution time much the same as the InsertionSort.

This is the drawback to the QuickSort: If the pivot is one of the five or so smallest values, or one of the five or so largest values, the vast majority of the time the pivot is chosen, then the QuickSort can take a little longer than the InsertionSort. But that is extremely rare for randomly-ordered data. The QuickSort will almost always drastically speed up the sorting process relative to the InsertionSort. An exercise has you implement a "tune-up" in which you select the **middle of three** values taken from the bottom, the top, and the middle of the array. This virtually guarantees a drastic speed-up.

Recursion

A single method may be called many times at different points during the execution of the program. Each of these calls is a different **activation** of the method. The runtime system records information about each activation in separate "method objects" that it creates and discards during execution of the program. You must keep this point firmly in mind when thinking about **recursion**: A method is allowed to contain a call of itself. The QuickSort logic described earlier requires the use of recursion.

As an example of recursion, the `insertionSort` method could have been written recursively as follows (though we normally do not do this where a simple for-statement accomplishes the same thing):

```
public static void insertionSort (Comparable[] item, int size)
{  if (size > 1)
   {  insertionSort (item, size - 1);                //recur
      insertInOrder (item, size - 1);
   }
}  //=======================
```

For that matter, the `selectionSort` method could have been written recursively if we had chosen to have a `swapMaxToRear` method that swaps the largest value to the rear of the array instead of a `swapMinToFront` method that swaps the smallest value to the front of the array:

```
public static void selectionSort (Comparable[] item, int size)
{  if (size > 1)
   {  swapMaxToRear (item, 0, size - 1);
      selectionSort (item, size - 1);                //recur
   }
}  //=======================
```

Say you want to print out all the 6-digit binary numbers (all 64 of them). Then a call of `printBinary ("", 6)` does the job if you have method shown in Listing 13.4. Study it for a while to see that it prints out every binary number that has `numDigits` digits, each one prefixed by the given String `prefix`:

Listing 13.4 The independent printBinary method

```
/** Print every binary number of numDigits digits. */

public static void printBinary (String prefix, int numDigits)
{  if (numDigits <= 1)
      System.out.println (prefix + "0\n" + prefix + "1");
   else
   {  printBinary (prefix + "0", numDigits - 1);      //recur
      printBinary (prefix + "1", numDigits - 1);      //recur
   }
}  //=======================
```

If you try to do that with a while-statement, it will not be nearly as compact and clear. In the three methods just given, each expression that makes a recursive call has this hallmark property:

(a) it is guarded by a test that a certain variable, called the **recursion-control variable**, is greater than a certain **cutoff** amount, and
(b) it passes to the new activation a value of the recursion-control variable that is at least 1 smaller than the existing activation has.

In the three examples just given, the expression that makes the recursive call is marked `//recur` out at the side:

- For `insertionSort`, the recursion-control variable is `size` and its cutoff is 1.
- For `selectionSort`, the recursion-control variable is `size` and its cutoff is 1.
- For `printBinary`, the recursion-control variable is `numDigits` and its cutoff is 1.

In some applications of recursion, the coding itself does not declare a variable specifically as the recursion-control variable. But any coding that uses recursion properly will involve a quantity that could be assigned to a recursion-control variable if need be.

Of course, it is possible to write recursive coding that falls into an "infinite loop" because it does not have a recursion-control variable. But then again, it is also possible to write a while-statement that falls into an "infinite loop" because it does not have an "iteration-control variable" that eventually reaches a certain cutoff and terminates the loop.

It is the recursion-control variable and the cutoff value that guarantee that recursion does not go on forever. Think of each activation of the method as being a certain height above the floor. The recursion-control variable measures the height above the floor. Each time you step from one activation to another, the height drops. When you spiral down to the floor (the cutoff value), the recursion stops. In short, a correctly coded recursion process does not go in circles, it goes in spirals.

Object design

A reasonable approach is to create a QuickSorter object, giving it the array that contains the values to be sorted. Then you can ask it to sort one part or another part of the array. So the primary message sent to a QuickSorter object should be as follows, where `start` and `end` are the first and last indexes of the part of the array to be sorted:

```
   someQuickSorter.sort (start, end);
```

The object responds to the `sort` message by first making sure that it has at least two values to sort, i.e., that `start < end`. If so, it rearranges the values around the pivot, then creates another QuickSorter object, its helper, to sort each half. Since the rearranging is rather complex, you could put it off to some other method, which will "split" the array in two and report the index where the pivot was put. That leads to the following basic coding for the `sort` method:

```
   int mid = this.split (start, end);
   helper.sort (start, mid - 1);
   helper.sort (mid + 1, end);
```

The `split` method is to split the sortable portion of the array into two parts, those below index `mid` that are smaller than the pivot, and those above index `mid` that are larger than the pivot. The QuickSorter object's helper then sorts each part separately. This overall logic is illustrated in Figure 13.5.

1. The initial array	7	12	5	10	3	6	8
2. Remove the pivot 7	**X**	12	5	10	3	6	8
3. Split, smaller below, larger above	6	3	5	**X**	10	12	8
4. Put the pivot back in the middle	6	3	5	**7**	10	12	8
5. Sort the ones smaller than 7	3	5	6	**7**	10	12	8
6. Sort the ones larger than 7	3	5	6	**7**	8	10	12

Figure 13.5 The overall logic of the QuickSort

Development of the split method

Splitting the portion of the array in parts smaller than and larger than a pivot value is tricky. Call the lowest and highest indexes for the range `lo` and `hi`. They are initialized to the `start` and `end` values, but `lo` and `hi` will change during the logic whereas `start` and `end` will stay their original values. So it will be less confusing to have different names.

Take the pivot value from `itsItem[lo]` (an exercise improves this to a random choice; then the algorithm is called **randomized QuickSort**). That leaves the component empty. What should go there at the lower part of the array? An element that is smaller than the pivot. And where should you take it from? From where it should not be, namely, the higher part of the array. So look at `itsItem[hi]` to see if that is smaller than the pivot. Say it is not. Then `itsItem[hi]` is in the right place, so decrement `hi` and look at `itsItem[hi]`, which is the one below the original `hi` one. Say that one is smaller than the pivot. So you move the contents of `itsItem[hi]` down to the empty spot `itsItem[lo]`, which leaves `itsItem[hi]` empty.

What should go into `itsItem[hi]`, in the higher part of the array? An element that is larger than the pivot. And where should you take it from? From where it should not be, namely, the lower part of the array. You start looking at `itsItem[lo+1]`. So you increment `lo` to keep track of the position in the lower part of the array. Keep incrementing `lo` until you get to a component that has a larger value than the pivot. Move that value up into the empty spot at `itsItem[hi]`. Then go back to looking in the higher part of the array for a value to move into the now-empty spot `itsItem[lo]`.

You actually do not need multiple QuickSorter objects; just one can do the entire job itself. Listing 13.5 (see next page) makes this adjustment, which speeds up execution. Study the complete coding in Listing 13.5 until you understand everything about it.

It keeps track of whether you are looking in the higher part of the array (versus the lower part) with a boolean variable `lookHigh` (line 7). When `lookHigh` is true, you are decrementing `hi` in the higher part of the array until you find a smaller value than the pivot, which you then move into the empty `itsItem[lo]` (line 13). When `lookHigh` is false, you are incrementing `lo` in the lower part of the array until you find a larger value than the pivot, which you then move into the empty `itsItem[hi]` (line 21).

This coding uses a language feature not used in any other chapter: The phrase `itsItem[lo++]=` in line 13 assigns a value to `itsItem[lo]` and <u>thereafter</u> increments `lo`. In general, the value of the expressions `x++` and `x--` is the value that `x` had before it incremented or decremented. Thus `itsItem[hi--]` in line 21 assigns a value to `itsItem[hi]` and <u>thereafter</u> decrements `hi`. If you use this shorthand feature, follow this safety rule: Never use it in a statement that mentions the incremented/decremented variable (`lo` or `hi` or whatever) more than once. The reason is that its value is very tricky to figure out in such a case.

The loop invariant

When `lo` and `hi` coincide, you have found the empty spot where the pivot should go. You can verify that the loop in the `split` method does what it should do by checking that the following statement is a loop invariant:

Loop invariant for the while loop in `split` (any time the loop condition `lo < hi` is evaluated): No value from `start` through `lo-1` is larger than the pivot and no value from hi+1 through end is smaller than the `pivot`. Also, `itsItem[lo]` is "empty" if `lookHigh` is true, but `itsItem[hi]` is "empty" if `lookHigh` is false.

Listing 13.5 The QuickSort Algorithm for a partially-filled array

```
// a method in the CompOp class; same conditions as Listing 13.1
   public static void quickSort (Comparable[] item, int size)
   {  new QuickSorter (item).sort (0, size - 1);
   }  //=======================


public class QuickSorter
{
   private Comparable[] itsItem; // the array to be sorted

   public QuickSorter (Comparable[] item)
   {  itsItem = item;
   }  //=======================

   /** Precondition: start <= end; itsItem[start]...itsItem[end]
    *  are the Comparable values to be sorted. */

   public void sort (int start, int end)
   {  if (start < end)                                          //1
      {  int mid = split (start, end);                          //2
         sort (start, mid - 1);                                 //3
         sort (mid + 1, end);                                   //4
      }                                                         //5
   }  //=======================

   private int split (int lo, int hi)
   {  Comparable pivot = itsItem[lo];                           //6
      boolean lookHigh = true;                                  //7
      while (lo < hi)                                           //8
      {  if (lookHigh)                                          //9
         {  if (itsItem[hi].compareTo (pivot) >= 0)             //10
               hi--;                                            //11
            else                                                //12
            {  itsItem[lo++] = itsItem[hi];                     //13
               lookHigh = false;                                //14
            }                                                   //15
         }                                                      //16
         else                                                   //17
         {  if (itsItem[lo].compareTo (pivot) <= 0)             //18
               lo++;                                            //19
            else                                                //20
            {  itsItem[hi--] = itsItem[lo];                     //21
               lookHigh = true;                                 //22
            }                                                   //23
         }                                                      //24
      }                                                         //25
      itsItem[lo] = pivot;                                      //26
      return lo;                                                //27
   }  //=======================
}
```

By "empty" we mean that you may overwrite that value without losing any value that was in the original portion of the array to be sorted. In Figure 13.6, the X marks the component that is "empty". The positions of `lo` and `hi` after the action described at the left of Figure 13.6 are boldfaced so you can track their movements. On each iteration of the loop, one or the other moves one step closer to the middle.

Initially lo=20 and hi = 26:	20	21	22	23	24	25	26
The array values are: (boldfaced at lo and at hi)	**7**	12	5	10	3	6	**8**
First: Take the pivot from index lo, so pivot is 7:	**X**	12	5	10	3	6	**8**
Iteration 1: lookHigh is true but itsItem[hi]>pivot so hi--:	**X**	12	5	10	3	**6**	8
Iteration 2: lookHigh is true, itsItem[hi]<pivot so move:	6	**12**	5	10	3	**X**	8
Iteration 3: lookHigh is false, itsItem[lo]>pivot so move:	6	**X**	5	10	**3**	12	8
Iteration 4: lookHigh is true, itsItem[hi]<pivot so move:	6	3	**5**	10	**X**	12	8
Iteration 5: lookHigh is false, itsItem[lo]<pivot so lo++:	6	3	5	**10**	**X**	12	8
Iteration 6: lookHigh is false, itsItem[lo]>pivot so move:	6	3	5	**X**	10	12	8
Now lo == hi, so X marks the spot where pivot goes:	6	3	5	**7**	10	12	8

Figure 13.6 Sequence of operations for the split method

The `split` method in the QuickSort logic can be used to solve a problem that arises in practice from time to time: Given a large number of values in an array, find the Kth smallest, e.g., the 575th smallest or the 2300th smallest. Once you split the array to find the correct position of the pivot, you calculate whether that position is above or below the Kth position. Then you re-split the one part of the array that contains the Kth position. This **QuickSelect** problem is left as a major programming project.

Language elements
For any int variable k, k++ has the value that k had before it was incremented. Similarly, k-- evaluates as the value k had before it was decremented. So if k is 4, then k++ is 4 and k-- is 4.
On the other hand, ++k yields the value k had after incrementing and --k yields the value that k had after decrementing. Examples: If k is 4, then ++k is 5 and --k is 3.
This compactness comes with an increased risk of getting the logic wrong.
It is most dangerous if you mention k anywhere else in the same statement.

Exercise 13.19 Trace the action of the `split` method on the sequence {7, 2, 9, 1, 5, 8, 4} in the way shown in Figure 13.6.
Exercise 13.20 Revise the QuickSorter class to put values in descending order rather than in ascending order.
Exercise 13.21 What is the sequence of all pivot values used by all calls of `quickSort` if the initial call is for the sequence {4, 1, 8, 3, 9, 5, 6, 2, 7}. Hint: There are five of them.
Exercise 13.22 What difference would it make if you replaced ">= 0" by "> 0" and replaced "<= 0" by "< 0" in the `split` method?
Exercise 13.23 (harder) (a) Rewrite the `split` method to take the pivot from index `hi` instead of index `lo`. (b) Same problem, but take it from a random index `lo` to `hi`.
Exercise 13.24 (harder) Explain why the loop invariant is trivially true at the first test of the loop condition.
Exercise 13.25* Explain why the loop invariant is incrementally true for each iteration.
Exercise 13.26* What difference would it make if the `>=` and `<=` operators were replaced by `>` and `<` respectively in the `split` method?
Exercise 13.27* Rewrite the `quickSort` and `sort` methods so that the `sort` method is never called when there are less than two items to sort. Is this more efficient?
Exercise 13.28* Rewrite the `split` method to find a pivot value that is much more likely to be close to the middle, as follows: Compare the three values at indexes `lo`, `hi`, and `(lo+hi)/2`. Whichever of them is between the other two is to be the pivot.
Exercise 13.29* Write an InsertionSorter class of objects analogous to QuickSorter. This would allow people to use the InsertionSort logic for any subsequence of an array, not just starting from zero. Extra credit: Revise QuickSorter to call an InsertionSorter object to do the sorting when it is given 10 items to sort or less.
Exercise 13.30** Rewrite the `split` method so that the body of the while-statement contains two additional while-statements, one whose only subordinate statement is `hi++` and the other whose only subordinate statement is `lo--`. This lets you eliminate the `lookHigh` variable entirely. Does this re-coding make it execute faster or slower?

13.5 The Recursive MergeSort Algorithm For Comparable Objects

The irritating thing about the QuickSort algorithm is that it is unreliable. The vast majority of the time, it executes faster than the InsertionSort for a large number of values. But in rare cases it is not significantly faster. The key to the problem is that the pivot value does not always divide the group of values to be sorted in two equal parts. The MergeSort algorithm presented in this section avoids this problem yet executes about as fast.

Putting a mostly sorted group in order

For a warmup, consider this problem: Somewhere in the `item` array is a sequence of values. The first half of them are in ascending order and the second half of them are also in ascending order. Your job is to rearrange this sequence so that they are all in ascending order.

This could be very messy, swapping values all around, except for one thing: You get to have another array to put the values in as you sort them. That makes it so much easier. Look at a picture to see what is going on: The upper array in Figure 13.7 has two subsequences of four values, each sequence in ascending order. The lower array shows what the result of the process should be.

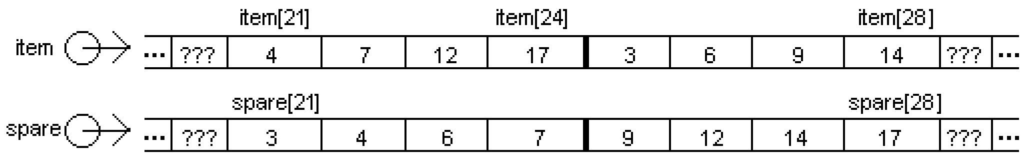

Figure 13.7 Problem: Move the 8 values into spare in ascending order

There are really only two decent ways to go about solving this problem -- working from smallest to largest or vice versa. The accompanying design block describes the former. The key point is that the smallest of all of the values must be either the first (smallest) value in the lower half or the first (smallest) value in the higher half. Compare these two values. Whichever is smaller, put that one in the first available spot in the spare array and move on to the one after it in its subsequence.

STRUCTURED DESIGN for merging two ascending sequences

```
1.  Set lo to be the first index of the lower group of ordered values.
2.  Set hi to be the first index of the higher group of ordered values.
3.  Compare the values at lo and hi in the item array to see which is smaller.
4.  If the one at hi is smaller, then..
            4a. Move the one at hi to the next available spot in the spare array
                and increment hi.
5.  Otherwise...
            5a. Move the one at lo to the next available spot in the spare array
                and increment lo.
6.  Repeat from Step 3 until you have put all the values into the spare array.
```

Suppose you have `end` store the highest index value that `hi` can have, and have `mid` store the highest index value that `lo` can have (because it is in the middle of the portion to be sorted). In Figure 13.7, for instance, `mid` is 24 and `end` is 28. Then the following coding is a translation of the design. The conditional test is complex because you cannot compare `item[lo]` with `item[hi]` unless you make sure that `lo` has not yet gone past `mid` and that `hi` has not yet gone past `end`. Study the condition so you can see why it works.

```
for (int spot = lo;  spot <= end;  spot++)
{  if (lo > mid || (hi <= end
                     && item[lo].compareTo (item[hi]) > 0))
      spare[spot] = item[hi++];
   else
      spare[spot] = item[lo++];
}
```

For Figure 13.7, `spot` and `lo` are initially 21 and `hi` is initially 25. Compare 4 and 3. Since 3 is smaller, the 3 from `item[25]` moves into `spare[21]` and `hi` is increased to 26. Next compare 4 and 6. Since the 4 is smaller, the 4 from `item[21]` moves into `spare[22]` and `lo` is increased to 22. The rest of this example is left as an exercise.

Loop invariant for the for-loop in the merging process (any time when the loop condition `spot <= end` is evaluated): If `start` is the original value of `lo`, then all values in the `spare` array from index `start` through index `spot-1` are the `spot-start` smallest of the values that are in `item[start]` through `item[end]`, in ascending order.

The MergeSort logic

The idea behind the MergeSort is that you divide the range of values to be sorted into two exactly equal parts (or with a difference of 1 if you have an odd number of values). First you sort the first half, next you sort the second half, and finally you merge the two halves together as one long sequence in ascending order.

Why is this so much faster than the InsertionSort logic? Say the InsertionSort takes 800 milliseconds to sort 2000 items and you decide to do a MergeSort but sort the two halves with the InsertionSort logic. Then, as explained earlier, the InsertionSort will take about 200 milliseconds to sort the first 1000 items, and about 200 milliseconds to sort the second 1000 items, and the MergeSort will take about 2 milliseconds to merge them into another array in order. Total execution time is 402 milliseconds, barely more than half as long as using the InsertionSort for all the sorting. And if the MergeSort logic is used to sort each of the two halves, that speeds it up far far more.

This logic does not have the disadvantage that the QuickSort has of sometimes degenerating into an elementary sorting algorithm. But it has a different disadvantage: It requires a second array to get the job done.

The object design

The obvious object design for the MergeSort algorithm is about the same as for the QuickSort except you have to make allowance for the second (spare) array. You need the following method in CompOp, to be consistent with the other sorting algorithms:

```
public static void mergeSort (Comparable[] item, int size)
{  Comparable[] spare = new Comparable [item.length];
   new MergeSorter (item, spare).sort (0, size - 1);
}  //======================
```

How does the MergeSorter object sort the values? It first checks that it has at least two values to sort, since otherwise it does not need to take any action. If it has at least two values, it creates another MergeSorter object as a helper to sort each half of the array separately. Then it executes the merge logic discussed previously to merge the two sorted halves together. The logic apparently goes something like this:

```
   int mid = (start + end) / 2;
   helper.sort (start, mid);
   helper.sort (mid + 1, end);
   this.merge (start, mid, mid + 1, end);
```

But now you have a problem. Since the executor is to leave the values in `itsItem` array, it should merge them from `itsSpare` array into `itsItem` array. That means that its helper should leave each of its sorted halves in `itsSpare` array. So some MergeSorter objects leave the values they sort in `itsItem` and others leave them in `itsSpare`. You need some way of telling a MergeSorter object which one it is to do.

The solution, given in Listing 13.6 (see next page), is to have two sort methods, one for sorting into `itsItem` array (lines 1-6) and one for sorting into `itsSpare` array (lines 7-14). A MergeSorter object that is to sort into `itsItem` array tells its helper to sort into `itsSpare` array so it can merge the two halves back into `itsItem`. That helper will tell its own helper to sort into `itsItem` array so it can merge the two halves into `itsSpare`. This is **indirect recursion** -- The `sort` method does not call itself directly, it calls the `sortToSpare` method which calls the `sort` method.

When a MergeSorter object gets back the two sorted halves, it calls the `merge` method (lines 15-21). That method's first parameter is the array the values are coming from and its second parameter is the array the values are going into. Some MergeSorter objects pass `itsItem` to the first parameter and others pass `itsSpare` to the first parameter.

You actually do not need multiple MergeSorter objects; just one will do. Instead of having a helper, a MergeSorter object does the whole job itself. The coding in Listing 13.6 makes this adjustment, which speeds up execution. You should study the complete coding in Listing 13.6 until you understand everything about it.

Faster sorting with the MergeSort

Every iteration of the loop in the `merge` method makes four tests, until one of the two halves runs out of values. It would execute faster if it could make just two tests. A nice way to do this is to sort the first half in increasing order and the second half in decreasing order. Then to merge the two halves of this "rise-then-fall" sequence, `lo` increments from `start` towards the middle and `hi` decrements from `end` towards the middle, similar to what happens in the QuickSort logic. When they meet, the loop stops. The following is the resulting `merge` method, making only two tests on each iteration:

```
   private void merge (Comparable[] from, Comparable[] into,
                                  int lo, int hi)
   {  int spot = lo;
      while (lo < hi)
         into[spot++] = from[lo].compareTo (from[hi]) > 0
                     ? from[hi--]  :  from[lo++];
      into[spot] = from[lo];
   }  //======================
```

Of course, now you have to also write a `mergeDown` method for merging the same kind of "rise-then-fall" sequence in descending order. And you need two additional methods `sortDown` and `sortToSpareDown` that call on the `mergeDown` method, unless you add a boolean parameter to each of `sort` and `sortToSpare` to select the right merging method. But still, it will execute moderately faster than the simple MergeSort algorithm. This is left as a major programming project.

Listing 13.6 The MergeSort Algorithm for a partially-filled array

```
public class MergeSorter
{
   private Comparable[] itsItem;  // the array to be sorted
   private Comparable[] itsSpare; // spare to facilitate sorting


   public MergeSorter (Comparable[] item, Comparable[] spare)
   {  itsItem = item;
      itsSpare = spare;
   }  //======================


   public void sort (int start, int end)
   {  if (start < end)                                           //1
      {  int mid = (start + end) / 2;                            //2
         sortToSpare (start, mid);                               //3
         sortToSpare (mid + 1, end);                             //4
         merge(itsSpare, itsItem, start, mid, mid + 1, end);     //5
      }                                                          //6
   }  //======================


   private void sortToSpare (int start, int end)
   {  if (start >= end)                                          //7
         itsSpare[start] = itsItem[start];                       //8
      else                                                       //9
      {  int mid = (start + end) / 2;                            //10
         sort (start, mid);                                      //11
         sort (mid + 1, end);                                    //12
         merge(itsItem, itsSpare, start, mid, mid + 1, end);     //13
      }                                                          //14
   }  //======================


   private void merge (Comparable[] from, Comparable[] into,
                       int lo, int mid, int hi, int end)
   {  for (int spot = lo;  spot <= end;  spot++)                 //15
      {  if (lo > mid || (hi <= end                              //16
                     && from[lo].compareTo (from[hi]) > 0))      //17
            into[spot] = from[hi++];                             //18
         else                                                    //19
            into[spot] = from[lo++];                             //20
      }                                                          //21
   }  //======================
}
```

Another speed-up is for the `merge` method to see whether the highest value in the lower group is less than the lowest value in the upper group and, if so, skip the merging. This executes much faster for nearly-sorted lists. It is also left as a major programming project.

Non-recursive merge sort

The sequence of actions of the merge sort on a set of 16 data values is as follows. From this description you can probably see how it can be coded non-recursively:

1. Copy values 0 and 1 into the spare array, swapping if needed. Do the same for values 2 and 3, then for values 4 and 5, then for values 6 and 7, then values 8 and 9, then values 10 and 11, then values 12 and 13, then values 14 and 15.
2. Merge values 0 and 1 with values 2 and 3 back into `itsItem`. Do the same for merging values 4 and 5 with values 6 and 7, then for values 8...11, then for values 12...15.
3. Merge values 0...3 with values 4...7 into the spare array; then merge values 8...11 with values 12...15 into the spare array.
4. Merge values 0...7 with values 8...15 from the spare array into `itsItem`.

These 16 values ended up back in the `itsItem` array as they should. If you had anywhere from 9 to 16 values to sort, you would sort by 2s into the spare array, then merge groups of 4 into `itsItem` (the last group may be smaller), then merge groups of 8 into the spare array (the last group may be smaller), then all of them back into `itsItem`. If you had anywhere from 33 to 64 values to sort, two extra passes would leave the data values in the `itsItem` array also.

If you had anywhere from 17 to 32 values to sort, this process would leave you in the wrong array. You can avoid the extra copying pass by making the first pass consist of swapping each pair of out-of-order values with each other in the `itsItem` array. Then the second pass merges groups of 4 into the spare array, etc. In general, you do this swapping-in-place when the number N of values to sort has log2(N) an odd number. That is, `(int) Math.ceil (Math.log(N) / Math.log(2)) % 2` is 1.

Exercise 13.31 Write out a complete trace of the merge action for Figure 13.7.
Exercise 13.32 (harder) Revise Listing 13.6 to omit the `sortToSpare` method. Instead, pass an extra boolean parameter to `sort` that tells whether the data should end up in `itsItem` or in `itsSpare`. Is this an improvement? Why or why not?
Exercise 13.33 (harder) The MergeSort logic works faster if you handle the sorting of just one or two values separately and straightforwardly. Insert coding into Listing 13.6 to do this; begin with `if (start + 1 >= end)`, since that condition tells you when there are one or two values.
Exercise 13.34 (harder) Suppose `mergeSort` is called for an array of 8 data values. What are the values of the indexes `start` and `end` on each of the 15 calls of `sort` or `sortToSpare`? List them in the order they occur.
Exercise 13.35* Rewrite Listing 13.6 to omit the `sortToSpare` method. Instead, merge from `itsItem` into `itsSpare` and then copy all the sorted values back into `itsItem`.
Exercise 13.36* Essay: How much does the modification described in the previous exercise slow the execution of the algorithm? Work it out as a formula.
Exercise 13.37* Rewrite the `merge` method in Listing 13.6 to have the loop execute only as long as `lo <= mid && hi <= end`. Then clean up whatever is left after that loop. This reduces the number of tests per iteration from four to three.
Exercise 13.38* Rewrite Listing 13.6 to have two kinds of MergeSorter objects, one merging into `itsItem` and the other merging into `itsSpare`.
Exercise 13.39*** Rewrite the MergeSort logic to work without using recursion, as indicated by the discussion at the end of this section.
Exercise 13.40*** Rewrite the QuickSort logic to work without using recursion.

13.6 More On Big-Oh

Question: How many comparisons does the MergeSort Algorithm make when there are `N` values to sort? Answer: First note that the body of the for-loop of `merge` executes exactly once for each value to be sorted. The MergeSorter object created by the `mergeSort` method gets `N` values to sort, so the if-condition in its `merge` method executes `N` times. The last time, the `compareTo` method is not evaluated, and perhaps some other times as well, so at most `N-1` comparisons are made.

The initial call of `sort` by the `mergeSort` method includes two "second-level" calls of `sort` , one on each half of the data, so those calls get `N/2` values to sort the first time and the rest of the `N` values the second time, so the if-condition in the `merge` method called by those two calls of `sort` executes `N/2 + (N-N/2)` times, a total of `N` times. The last time for each call, the `compareTo` method is not evaluated, and perhaps some other times as well, so at most `N-2` comparisons are made. Similarly, the four "third-level" calls of `sort` with either `N/4` or `N/4+1` elements to sort evaluate the if-condition N times but make a total of at most `N-4` comparisons.

If say there are 32 values to sort altogether, we have calculated a total of 31+30+28 comparisons so far. The 8 "fourth-level" calls of `sort` make at most 3 comparisons each and the 16 "fifth-level" calls of `sort` make at most 1 comparison each. The total is at most `31+30+28+24+16 = 32*5-31` comparisons. In general, at most `N*log2(N)-N+1` comparisons are made to get `N` values sorted in ascending order, where `log2(N)` is the number of times you divide `N` by 2 to reduce it to 1 or less.

The technical way of saying this is that **the big-oh behavior of MergeSort is N*log(N)**, where N is the number of items to be sorted. It means that <u>the amount of time to do the job is roughly proportional to the number of items processed multiplied by the logarithm of the number of items processed, for large values of N</u>.

If we define compMS(N) to be the maximum number of comparisons the MergeSort requires to sort N values, then compMS(N) can be calculated as N-1 plus the number required to sort N/2 values (one half) plus the number required to sort N-N/2 values (the other half). Since no comparisons are required to sort just 1 value, the compMS function is defined by the following **recurrence equations**:

compMS(N) = (N-1) + compMS(N/2) + compMS(N - N/2)   for N > 1
compMS(1) = 0

A concrete example

If a particular processor requires 0.01 seconds to sort 100 items using an InsertionSort, it will take about $10{,}000^2$ times as long to sort a million items, because 1 million is 10,000 times as much and the InsertionSort algorithm is big-oh of N^2. That is 1 million seconds, which is over 11 days.

Suppose it takes that same processor 0.02 seconds to sort 100 items using MergeSort (perhaps the MergeSort is slower for so few items). Then it will still take only 600 seconds to sort a million items using the MergeSort, i.e., 10 minutes. This is calculated from 0.02 seconds = `M * 100 * log2(100)`, where the multiplier M depends on the processor. So the time for a million items is `M * 1,000,000 * log2(1,000,000)` which is only 30,000 times as long as 0.02 seconds, to sort 10,000 times as many items. Figure 13.8 shows comparative values for N being various powers of 10.

N	N * log2(N)	N-squared
10	40	100
100	700	10,000
1,000	10,000	1 million
10,000	140,000	100 million
100,000	1.7 million	10,000 million
1 million	20 million	1 million million

Figure 13.8 The relation of N to N*log2(N) and N-squared

Clearly the MergeSort executes faster than any elementary sort for very large amounts of data. Does that mean it is more efficient? Efficiency is not just a matter of execution time. **Efficiency depends on three factors -- space, time, and programmer effort.** The MergeSort takes twice as much space (the `itsSpare` array) and a lot more programmer effort to develop. So it is better to use an elementary sort unless the amount of data is so large as to make the extra space and effort worth the savings in time.

Execution time for QuickSort

The QuickSort algorithm in the best case makes slightly fewer comparisons than the MergeSort makes in the worst case, i.e., when the QuickSort's pivot turns out to be right in the middle each time. Specifically, one execution of the QuickSort algorithm for N values takes `N-1` comparisons to get the pivot in the right spot, then it sorts `N/2` values in one half and `N - N/2 - 1` values in the other half (assuming a perfect split).

Since the split operation of the QuickSort on average only moves about half of its values around in the array, but the merge operation of the MergeSort moves all of the values around in the arrays, the best-case performance of QuickSort is somewhat better than the worst-case performance of MergeSort.

Statistical analysis has shown that the average-case performance of QuickSort is 1.386 times the best-case performance time (that number is twice the natural log of 2, in case you are interested in where it came from). That is 38.6% slower than the best-case time, so QuickSort has on average a slightly greater execution time than MergeSort. However, the worst-case performance of QuickSort is big-oh of N^2 and somewhat worse than InsertionSort.

Fastest sorting algorithms

We next show that any sorting algorithm that uses no information about the values being sorted except what the `compareTo` method supplies must make at least `N*Log(N)-1.5N` comparisons to be sure of sorting `N` different values. Here we use Log to denote logarithms base 2 without rounding up (e.g., Log(3) is about 1.58). Since we have already shown that the MergeSort logic never requires more than `N*log2(N)-N+1` comparisons, it follows that you can hardly do much better than the MergeSort.

To see why this assertion is true, consider that the `N` different values can be in any of `N!` (N-factorial) different orderings. Each use of the `compareTo` method eliminates from consideration at most half of the number of possible orderings. So there will always be some cases where you have to make at least `Log(N!)` calls of `compareTo` before you know which ordering you have.

Once we prove the general (Stirling) formula that $N! > N^N/2^{1.5N}$, it will follow that

$$Log(N!) > Log(N^N/2^{1.5N}) = N*Log(N/2^{1.5}) = N*Log(N)-1.5N.$$

This general formula is obviously true when N is 1 or 2 (since then the right side of the formula is less than 1 but the left side is at least 1). Once you have verified the general formula holds for some particular value of N, you can inductively verify it for N+1 as follows: The left side of the general formula for N+1 obviously becomes N+1 times as large as it was for N, but its right side becomes less than N+1 times as large (according to the logic in the next paragraph), so the general formula still holds for N+1. The Induction Principle tells us that the general formula will always be true.

How do we know that replacing N by N+1 in the expression $N^N/2^{1.5N}$ makes the expression less than N+1 times as large? Because the denominator increases by about 2.82 (twice the square root of 2) and the numerator increases by about 2.72 times N+1. Specifically, the N^N numerator increases by two factors: $((N+1)/N)^N$ and `N+1`. The first factor is less than Math.E, since Math.E is defined to be the upper limit of all such values. And Math.E is about 2.72, which is less than 2.82. Q.E.D.

Stability

The MergeSort has another advantage in that it is a stable sort. A sort is **stable** if two values that are equal end up in the same relative position that they had in the unsorted sequence. For example, if `A.compareTo(B)` is zero, and A was initially at index 22 and B was initially at index 37, then A should be at a lower index than B in the sorted array. This is always true for the MergeSort and the InsertionSort, but it is not always true for the QuickSort and the SelectionSort. So the QuickSort and SelectionSort algorithms described in this chapter are not stable sorts. Of the three sorts described in the next section, the ShellSort is not stable but the other two are.

An extended example

The next example illustrates how you can use the `insertionSort` method and others presented in this chapter to sort an array of values using a criterion other than what the `compareTo` method returns. Say you have a class of objects named Worker, and one of the instance methods returns an int value, e.g., `someWorker.getBirthYear()` is the year in which that Worker was born. You can then sort an array of Workers in order of birth year, even if the Worker class does not implement the Comparable interface.

If you have a partially-filled array `itsItem` that contains `itsSize` Worker objects, then the following statement would sort them in ascending order of birth year and also obtain the number of comparisons of pairs of Workers that were made to get them sorted:

```
   int n = WorkerByBirth.numComps (itsItem, itsSize);
```

Listing 13.6 (see next page) defines a WorkerByBirth object to have one instance variable, a Worker, and one instance method such that `x.compareTo(y) < 0` tells whether Worker `x` has a smaller birth year than Worker `y` has. This is in the upper part of the listing. Note that you would only have to replace the one return statement in the `compareTo` method to have a class that puts people in ascending order of the number of children they have or puts houses in descending order of their selling price or puts any kind of object in the order dictated by any int-valued instance method.

The `numComps` class method takes any partially-filled array of Workers and creates a second array containing one WorkerByBirth object corresponding to each Worker object. Then it calls the `insertionSort` method developed earlier in this chapter to sort these Comparable objects in increasing order of birth year. Finally, it extracts the now-reordered Worker objects back into the original array.

Listing 13.7 Sorting a partially-filled array of Workers by birth year

```
class WorkerByBirth implements Comparable
{
   private static int theNumComps = 0;
   private Object itsWorker;


   public WorkerByBirth (Object given)
   {  itsWorker = given;
   }  //======================


   public int compareTo (Object ob)
   {  theNumComps++;
      return ((Worker) itsWorker).getBirthYear()
                    - ((Worker) ob).getBirthYear();
   }  //======================


   public static int numComps (Object[] givenArray, int size)
   {  Comparable[] tempArray = new Comparable [size];
      for (int k = 0;  k < size;  k++)
         tempArray[k] = new WorkerByBirth (given[k]);
      theNumComps = 0;
      CompOp.insertionSort (tempArray, size);
      for (int k = 0;  k < size;  k++)
         given[k] = tempArray[k].itsWorker;
      return theNumComps;
   }  //======================
}
```

Before calling the `insertionSort` method, `numComps` sets a class variable `theNumComps` to zero. Each call of the `compareTo` method in the WorkerByBirth class increments `theNumComps`. So the value of `theNumComps` returned is the number of comparisons of two Workers made during execution of the `insertionSort` method.

Note that you would only have to replace the name `insertionSort` by `quickSort` to use the QuickSort algorithm for the sorting rather than the InsertionSort algorithm, and similarly for other sorting algorithms. So you can use this coding to count the number of comparisons made for any sorting algorithm with just one name-change.

Exercise 13.41 For the processor described in this section (where sorting 100 values takes 0.01 second for InsertionSort and 0.02 seconds for MergeSort), how long would it take to sort ten million values with each method?
Exercise 13.42 Give an example to show that the SelectionSort is not stable.
Exercise 13.43* Give an example to show that the QuickSort is not stable.
Exercise 13.44* Explain why the InsertionSort is stable.
Exercise 13.45* Explain why the MergeSort is stable.
Exercise 13.46* Revise Listing 13.7 so that it will sort an array of String values in descending order of their second characters. Strings without a second character are considered to come before Strings that have second characters.
Exercise 13.47* If compQS(N) is the minimum number of comparisons that the QuickSort requires to sort N items, it is defined by the following recurrence equations:
compsQS(N)= (N-1) + compsQS(N/2) + compsQS(N-N/2-1) for N > 1; compsQS(1)=0.
Use this information to calculate compsQS(N) for 3, 7, 15, 31, and 63 (all are 2^k-1).

13.7 Additional Sorting Algorithms: Bucket, Radix, and Shell

There are many other sorting algorithms that computer scientists have invented over the years. One frequently-used algorithm is the heap sort, which is discussed at length in Chapter Eighteen. We briefly describe some other sorting algorithms in this section. You need to review the definition of a queue in Section 7.11 first, or read the first section of Chapter Fourteen, because this section uses queues to solve sorting problems.

The bucket sort

Sometimes you have a collection of objects that you want sorted based on age or the number of children or some other non-negative integer attribute of the objects. For generality, say that the objects have an instance method `getIntValue()` that returns a non-negative int value. Then you could keep an array of queues indexed by this int value. That is, `itsItem[k]` is a queue of all elements for which `getIntValue()` returns `k`. Putting each element into the data structure is a big-oh of 1 operation using the following statement:

```
   itsItem[element.getIntValue()].enqueue (element);
```

After you put all the elements in this data structure, remove them one at a time to get them in order of priority. This is called a **bucket sort**, since each queue can be thought of as a bucket containing data values. Its execution time is big-oh of N if the range of values for `getIntValue` does not change as N changes and is not too large. This sorting algorithm is a stable algorithm because we use queues rather than just any old kind of bucket.

A specific application of this algorithm sorts a large number of data values, each of which has an instance variable consisting of a 3-letter word formed with lowercase letters. If you want the data values sorted in increasing order of these words, which might be stored in an instance variable named `itsWord`, you could use the following `getIntValue` method to obtain a number in the range from 0 up to 26*26*26 (i.e., 17,576):

```
   public int getIntValue()
   {  return (itsWord.charAt (0) - 'a') * 26 * 26
                + (itsWord.charAt (1) - 'a') * 26
                + (itsWord.charAt (2) - 'a');
   }
```

The radix sort

If the range of values produced by `getIntValue` is too large (e.g., sorting by social security number would need an array with a billion components), use a **radix sort** instead:

1. Put each element into one of ten queues based on the last digit of `getIntValue`.
2. Combine the ten queues into one grand list, with the queue for digit '0' first, the queue for digit '1' second, etc. The queue for digit '9' will be at the end.
3. Repeat steps 1 and 2 for the next-to-last digit, then the third-to-last digit, etc., working right-to-left.

When this algorithm finishes, the elements will be sorted in order of `getIntValue`. To see this, assume that we are using 9-digit numbers (as for social security numbers). Because of the last pass through the data, all the elements that begin with '0' will be first on the list, followed by those that begin with '1', followed by those that begin with '2', etc. All the elements that begin with '0' were on the grand list at the end of the next-to-last pass in order of their second digit. That order did not change during the last pass. So within the group of elements that begin with '0', all the ones whose second digit is '0' come first, then come those whose second digit is '1', etc.

Total execution time is proportional to nine times the number of elements that you are sorting, since you make a total of nine passes through the data (one per digit). This assumes that combining the ten queues into one queue is a big-oh of 1 operation, which it is if you use NodeQueues with the append method described in Chapter Fourteen. This radix sort is a stable sorting method.

Back in the 1960's, there was a massive and expensive machine that sorted punch cards according to this radix sort algorithm. Say the social security number was stored in columns 40 through 48 of punch cards. The operator would stack the cards in an input hopper and push the button to indicate column 48. The machine would read column 48 of each punch card and drop the card into one of ten stacks depending on the digit in that column. The operator would collect the ten stacks, carefully putting stack '0' on top and stack '9' on the bottom. Then the operator would put the combined stacks in the input hopper and push the button to indicate column 47, etc.

An in-place merge sort

The big problem with the merge sort is that it requires a second array for temporary storage. The obvious way to avoid using a second array is as follows, assuming we are sorting the values in the index range `start` to `end` inclusive:

1. Divide the range in half by computing `mid = (start + end) / 2`;
2. Sort `item[start]...item[mid]` in place.
2. Sort `item[mid+1]...item[end]` in place.
3. Combine the two halves by inserting `item[mid+1]` where it goes in the first half, then inserting `item[mid+2]` where it goes in the first half, then `item[mid+3]`, etc.

The problem with this algorithm is that the merging part generally has to shift approximately N/2 values back down the array to make room for each new value inserted (N denotes the total number of items to sort). Since these large movements of data also take place when sorting the two halves, the overall execution time is generally longer than for the insertion sort. So this obvious algorithm is a bust.

However, if you only had to shift a very few data items for each insertion, this would actually work pretty fast. So let us divide the values to be sorted with items indexed 0, 2, 4, 6, 8, etc. in one half, and items indexed 1, 3, 5, 7, etc. in the other half. Sort the first half (the even-indexed) and sort the second half (the odd-indexed) separately. Then merge the two by doing an insertion sort on the whole. Since almost all of the data values should be very close to where they should be, we typically only move each data value 1 or 2 or 3 positions in this final sorting pass. So it takes time proportional to N. Since we make Log2(N) passes through the data, this algorithm seems likely to take not much more than N*Log2(N) execution time. But maybe more -- it depends.

Listing 13.8 (see next page) shows how this algorithm can be implemented non-recursively. Say we have 6,000 items to sort. We first calculate `jump` as 4096, the largest power of two that is less than the given size. Then `sortGroups (item, size, 4096)` compares each item with the one 4096 below it, swapping them if needed to get them in increasing order.

On the next call of `sortGroups`, `jump` is 2048, so the values at indexes 0, 2048, 4096, and 6144 are put in order using an insertion sort logic. Since the values at indexes 0 and 4096 are already in order, and so are the values at indexes 2048 and 6144, this will take less time than it otherwise would. Similarly, the values at indexes 1, 2049, 4097, and 6145 are put in order using an insertion sort logic, etc.

Listing 13.8 An in-place merge sort algorithm for a partially-filled array

```
/** Precondition:  size <= item.length; item[0]...item[size-1]
 *  are non-null values all Comparable to each other.
 *  Postcondition: The array contains the values it initially
 *  had but with item[0]...item[size-1] in ascending order. */


public static void mergeSortBy2s (Comparable[] item, int size)
{  int jump = 1;
   while (jump * 2 < size)
      jump *= 2;
   for ( ; jump >= 1;  jump /= 2)
      sortGroups (item, size, jump);
}  //=======================


private static void sortGroups(Comparable[] item, int size,
                                                  int jump)
{  for (int k = jump;  k < size;  k++)
      insertInOrder (item, k, jump);  // left as an exercise
}  //=======================
```

This process continues until on the last pass `jump` is 1, which does in effect an ordinary insertion sort. However, since the odd-indexed data values will already be in order, and so will the even-indexed data values, this last pass should take very little time. The coding for the required `insertInOrder` method is left as an exercise.

The shell sort

For the sorting method in Listing 13.8, it can sometimes happen that most of the odd-numbered values are rather small and most of the even-numbered values are rather large. So the last pass will still do a great deal of moving of data. This problem can be almost entirely eliminated if we sort of "mix up" the subsequences that we are sorting.

This can be done by using a `jump` value of one less than a power of 2 (4095 in the example) on the first pass. Then the next pass will divide it by 2, which gives a `jump` value of 2047. The next time it will be 1023, then 511, then 255, etc.

This sorting algorithm is called the **shell sort** (named after the person who invented it, Donald Shell). Any sequence of jump values can be used, as long as each is at least twice as small as the one before. The normal sequence of jump values used is 1, 4, 13, 40, 121, etc., i.e., the sum of the first few powers of 3 (121 is 1 + 3 + 9 + 27 + 81). Note that the jump values described in for the in-place merge sort are sums of the first few powers of 2 (1 + 2 + 4 + 8 + 16 + 32 + 64 + 128...). Note also that neither the shell sort nor the in-place merge sort is a stable sorting algorithm.

Exercise 13.48 Revise Listing 13.8 to do a shell sort with jumps of 1, 4, 13, 40, 121, etc.
Exercise 13.49* Write the `insertInOrder` method required for Listing 13.8.
Exercise 13.50* Essay: Explain clearly and in detail why the bucket sort and radix sort are both stable sorts.
Exercise 13.51** Write `public static void bucketSort (Coded[] item, int size)` to sort the `size` elements of the `item` array using the bucket sort logic. All values are elements of a class named Coded that defines `getIntValue()` and also defines `public final int MAX_VALUE` giving the largest value of `getIntValue()`.

*13.8 About The Arrays Utilities Class (*Sun Library)*

The **Arrays** class in the `java.util` package has 55 methods that can be useful at times. They are named `equals`, `sort`, `fill`, `binarySearch`, and `asList`.

- `Arrays.equals(someArray, anotherArray)` returns a boolean that tells whether the two arrays have the same values in the same order. There is one such method for each of the 8 primitive types (e.g., two arrays of ints) and one for two arrays of Objects (using that kind of Object's `equals` method).
- `Arrays.sort(someArray)` sorts in ascending numeric order an array of byte, char, double, float, int, short, or long values (thus 7 such methods). It uses a modification of the QuickSort algorithm that maintains N*log(N) performance on many sequences of data that would cause the simple QuickSort to degenerate into N^2 performance.
- `Arrays.sort(someArray, startInt, toInt)`: the same as the above except it only sorts the values at the indexes from `startInt` up to but not including `toInt`. As usual, it requires `0 <= startInt <= toInt <= someArray.length`.
- `Arrays.sort(someArray)` sorts in ascending order an array of Comparable objects with `compareTo`. It uses the MergeSort algorithm, thus it is a stable sort.
- `Arrays.sort(someArray, startInt, toInt)`: the same as the above for an array of Comparable objects except it only sorts the values at the indexes from `startInt` up to but not including `toInt`.
- `Arrays.fill(someArray, someValue)` assigns `someValue` to each component of the array. There is one such method for each of the 8 primitive types (e.g., an array of longs with a long `someValue`) and one for Objects.
- `Arrays.fill(someArray, startInt, toInt, someValue)`: the same as the above except it only assigns components from `startInt` up to but not including `toInt`.
- `Arrays.binarySearch(someArray, someValue)` returns the int index of `someValue` in the array using binary search. The array should be in ascending order. There is one such method for each of the 7 numeric types (e.g., an array of doubles with a double `someValue`) and one for Objects (using `compareTo`). If `someValue` is not found, then the value returned is negative: `-(n+1)` where n is the index where you would insert `someValue` to keep all the values in order.
- `Arrays.asList(someArrayOfObjects)` returns a List object "backed by" the array. This List cannot be modified in size, and each change to the List is reflected immediately in the array itself (`set` is supported but `add` and `remove` are not). The List interface is described in Section 15.10.
- Three additional Arrays methods are analogous to the three Object methods above that use `compareTo`, but they have an additional Comparator parameter.

The **Comparator** interface in the `java.util` package specifies two methods that can be used for Objects instead of the "natural" `compareTo` and `equals` methods:

- `someComparator.compare(someObject, anotherObject)` returns an int value with the same meaning as `compareTo`, but the comparison is usually based on some other criterion than what `compareTo` uses.
- `someComparator.equals(someObject, anotherObject)` returns a boolean telling whether the Comparator's `compare` method returns zero, or both objects are null.

13.9 Review Of Chapter Thirteen

In general:

- For any int variable k, if you use the expression k++ or k-- in a statement, it has the value that k had <u>before</u> it was incremented or decremented. When the variable is after the ++ or -- operator, it has the final value <u>after</u> incrementing or decrementing. For instance, if k is 4, then `item[k++]=2` assigns 2 to `item[4]` and changes k to 5, but `item[++k]=2` assigns 2 to `item[5]` and changes k to 5. If a variable appears twice in the same statement, and one appearance has the ++ or -- operator on it, the effect is too hard to keep straight, so do not do that.
- The source code for a method X can contain a call of X or a call of a method that (eventually) calls X. This is **recursion**. At run time, each execution of a method call generates an **activation** of that method. During execution of that activation, additional method calls generate complete different activations. Recursion means that two different activations can be executing the same logic sequence.
- A recursive method cannot loop forever if it has a **recursion-control variable**, either explicitly declared or implicit in the logic. A recursion-control variable for method X is a variable with a **cutoff** amount which satisfies (a) each call of X from within X requires that the recursion-control variable for the current activation is greater than the cutoff, (b) each call of X from within X passes a value of the recursion-control variable to the new activation that is at least 1 smaller than the existing activation has.
- We say that a function T(N) is **big-oh** of another function f(N) when you can find positive constants C and K such that T(N) <= C * f(N) whenever N >= K. The most functions that occur for f(N) are log(N), N, N*log(N), and N^2.

About some classic algorithms:

- The **SelectionSort** puts a list of two or more values in order by selecting the smallest, putting it first, then applying the SelectionSort process to the rest of the list.
- The **InsertionSort** puts a list of two or more values in order by applying the InsertionSort process to the sublist consisting of all but the last value, then inserting that last value in order.
- The **BinarySearch** algorithm finds a target value in an ordered list of two or more values by comparing the target value with the middle value of the list to decide which half could contain the target value, then applying the BinarySearch process to find the target value in that half of the list.
- The **QuickSort** puts a list of two or more two or more values in order by choosing one element, dividing the list in the two sublists of those larger and those smaller than the chosen element, then applying the QuickSort process to each of the two sublists.
- The **MergeSort** puts a list of two or more values in order by dividing it in half, applying the MergeSort process to each half, and then merging the two sorted halves together in order.
- Average execution time for the InsertionSort and SelectionSort is roughly proportional to N^2 where `N` is the number of values to sort. Sorting algorithms with this property are called **elementary sorts**.
- Average execution time for the BinarySearch algorithm is roughly proportional to `log2(N)`. In this context, **`log2(N)`** is the lowest power you put on 2 to get a number not smaller than `N`. In particular, `log2(N)` is 6 for any number in the range from 33 to 64.
- Average execution time for the MergeSort and QuickSort is roughly proportional to `N*log2(N)`. This describes the **big-oh behavior** of these algorithms. Figure 13.16 compares the sorting algorithms for efficiency. **Efficiency** depends on three factors: space, time, and programmer effort.

- A sort is **stable** if, of two values that are equal, the one that came earlier in the unsorted list ends up earlier in the sorted list. For the main algorithms described in this chapter, the InsertionSort and MergeSort are stable sorts; the SelectionSort and QuickSort are not.

	worst-case time	average-case time	extra space	programmer effort
SelectionSort	N-squared	N-squared	1	low
InsertionSort	N-squared	N-squared fast	1	low
QuickSort	N-squared	N * log(N) fast	1	medium
MergeSort	N * log(N)	N * log(N) fast	N	medium
HeapSort	N * log(N)	N * log(N)	1	high

Figure 13.16 Efficiency of five sorting algorithms (HeapSort is in Chapter 18)

Answers to Selected Exercises

13.1 Replace the loop condition k < size - 1 by item[k + 1] != null and replace the loop condition k <= end by item[k] != null.

13.2 Replace the phrase "< 0" by the phrase "> 0", the word "Min" by the word "Max", and the word "Smallest" by the word "Largest".

13.5
```
private static void insertInOrder (Comparable[ ] item, int m)
{    if (item[m].compareTo (item[m - 1]) < 0)
     {    Comparable save = item[m];
          item[m] = item[m - 1];
          for (m--;  ...  // all the rest as shown in Listing 13.2 after the for's first semicolon
     }
}
```
This executes faster only because item[m - 1] is moved up before the loop begins, so that the loop has one less iteration than in the original logic. Otherwise, if you made the comparison of item[m] with item[m - 1] twice, it would usually execute more slowly (the probability is quite low that item[m] is larger than the one before, once m becomes fairly large).

13.6 Replace the loop condition k < size by item[k] != null.

13.7 Replace "Comparable" by "double" in three places. Replace the second for-loop condition by: m > 0 && item[m - 1] > save

13.8 Insert the following phrase before the call of insertInOrder: if (item[k] != null)
Change the condition in the for-loop to be the following:
m > 0 && (item[m - 1] == null || item[m - 1].compareTo (save) > 0)

13.11 If the range lowerBound...upperBound has an odd number of values, say 13, then upperBound is lowerBound+12, so midPoint is lowerBound + 6, which divides the range into the 6 values above midPoint and the 7 values up through midPoint. In general, the front half is 1 larger when there are an odd number of values to search (upperBound - lowerBound is an even number).

13.12 If size is 0, then the body of the loop will not execute, and item[lowerBound] will be item[0]. Then the return statement might throw a NullPointerException or an ArrayIndexOutOfBoundsException.

13.13 The method is big-oh of N, where N is the number of values in the array.

13.14 If there are an even number of values to search, at least 4, then rounding the other way would make the lower part 2 larger than the upper part; the imbalance would slow the convergence somewhat. If however upperBound is lowerBound + 1, thus 2 values to search, then rounding up would make midPoint equal to upperBound. If target is larger than all values in the array, item[midPoint] is then smaller than target, so lowerBound becomes size, which might throw a NullPointerException or an ArrayIndexOutOfBoundsException. If upperBound is lowerBound+1 and item[upperBound] is not smaller than target, you would have an infinite loop.

13.19 pivot is 7, leaving **X**, 2, 9, 1, 5, 8, **4**. 4<7 moves the 4 to the X, leaving 4, **2**, 9, 1, 5, 8, **X**.
2<7 stays where it is. 9>7 moves the 9 to the X, leaving 4, 2, **X**, 1, 5, **8**, 9. 8>7 stays where it is.
5<7 moves the 5 to the X, leaving 4, 2, 5, **1**, **X**, 8, 9. 1<7 stays where it is. 7 replaces the X.

13.20 Replace >= by <= and <= by >= in the two if-conditions that mention compareTo.

13.21 The first pivot is 4, and the sort leaves {2, 1, 3, 4, 9, 5, 6, 8, 7}. The next pivot is 2, which leaves {1, 2, 3, **4**, 9, 5, 6, 8, 7}. The next pivot is 9, which leaves {1, 2, 3, **4**, 7, 5, 6, 8, **9**}. That just leaves {7, 5, 6, 8} to sort. The next pivot is 7, which leaves {1, 2, 3, **4**, 6, 5, **7**, 8, **9**}. The final pivot is 6.

13.22 Any value equal to the pivot would be moved from one side to the other of where the pivot goes. This would slow down the execution (due to the extra moves) but not change the outcome except that two objects x, y for which x.compareTo(y)==0 might be in a different order in the outcome.

13.23 (a) Replace the first two statements by Comparable pivot = itsItem[hi]; boolean lookHigh = false;
(b) Replace the first statement by int spot = lo + int (Math.random() * (hi + 1 - lo));
Comparable pivot = itsItem[spot]; itsItem[spot] = itsItem[lo];

13.24 At the first evaluation of the while-condition lo < hi, lo equals start (since start was passed in as the initial value of the parameter lo), so there are NO values from start through lo-1, so all of them are (vacuously) less than the pivot. Similarly, hi equals end, so there are no values from hi+1 through end, so all of those values are (vacuously) greater than the pivot. Finally, lookHigh is true and itsItem[lo] is where the pivot came from, so itsItem[lo] is "empty".

13.31 Initially we have (lo)4, 7, 12, 17, (hi)3, 6, 9, 14. Since 4 is larger than 3, move 3 to spare, so we now have (lo)4, 7, 12, 17, X, (hi)6, 9, 14. Since 4 is smaller than 6, move 4 to spare, so we now have X, (lo)7, 12, 17, X, (hi)6, 9, 14. Since 7 is larger than 6, move 6 to spare, so we now have X, (lo)7, 12, 17, X, X, (hi)9, 14. Since 7 is smaller than 9, move 7 to spare, so we now have X, X, (lo)12, 17, X, X, (hi)9, 14. Since 12 is larger than 9, move 9 to spare, so we now have X, X, (lo)12, 17, X, X, X, (hi)14. Since 12 is smaller than 14, move 12 to spare, so we now have X, X, X, (lo)17, X, X, X, (hi)14. Since 17 is larger than 14, move 14 to spare, then 17.

13.32 The sort method is revised as follows, to allow us to discard the sortToSpare method:

```
public void sort (int start, int end, boolean toSpare)
{       if (start >= end)
        {       if (toSpare)
                        itsSpare[start] = itsItem[start];
        }
        else
        {       int mid = (start + end) / 2;
                sort (start, mid, ! toSpare);
                sort (mid + 1, end, ! toSpare);
                if (toSpare)
                        merge (itsItem, itsSpare, start, mid, mid + 1, end);
                else
                        merge (itsSpare, itsItem, start, mid, mid + 1, end);
        }
}
```

This is worse. The extra tests make this execute slower, and it seems harder to understand.

13.33 Replace if(start < end) in the body of the sort method by the following:

```
if (start + 1 >= end)
{       if (start < end && itsItem[start].compareTo (itsItem[end]) > 0)
        {       Comparable save = itsItem[start];
                itsItem[start] = itsItem[end];
                itsItem[end] = save;
        }
}else
```

Replace the first two lines at the beginning of the sortToSpare method by the following:

```
if (start + 1 >= end)
{       if (start >= end)
                itsSpare[start] = itsItem[start];
        else if (itsItem[start].compareTo (itsItem[end]) > 0)
        {       itsSpare[end] = itsItem[start];
                itsSpare[start] = itsItem[end];
        }
        else
        {       itsSpare[start] = itsItem[start];
                itsSpare[end] = itsItem[end];
        }
}else
```

13.34 The calls are (a) sort(0,7), which calls (b) sortToSpare(0,3), which calls sort(0,1), which calls sortToSpare(0,0) and sortToSpare(1,1). Backing up, (b) calls sort(2,3) which calls sortToSpare(2,2) and sortToSpare(3,3). Backing up even more, (a) calls (c) sortToSpare(4,7), which calls sort(4,5), which calls sortToSpare(4,4) and sortToSpare(5,5). Backing up, (c) calls sort(6,7), which calls sortToSpare(6,6) and sortToSpare(7,7). Those are all of the 15 calls.
calls sort(2,2) and sort(3,3). Then (b) calls (d) sort(4,7), which similarly gives

13.41 For the InsertionSort, multiply the 11 days by the square of 10, thus 1100 days (about 3 years). For the MergeSort, the time is M * 10million * log2(10million). The second factor is 10 times what it was for a million items, and the third factor is 24/20 times what it was for a million items, so it takes 10*1.2 = 12 times 10 minutes, thus 2 hours.

13.42 For the sequence 5, 5, 3, 8, the first pass swaps the 3 with the first 5, so those two 5's are not in their original relative position. Other passes do not change anything.

13.48 Replace the middle 3 lines of the mergeSortBy2s method by the following lines:

```
while (jump * 3 + 1 < size)
        jump = jump * 3 + 1;
for ( ; jump >= 1;  jump /= 3)
```