Chapter 14 Sorting and Searching

CHAPTER GOALS

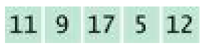
- To study several sorting and searching algorithms
- To appreciate that algorithms for the same task can differ widely in performance
- To understand the big-Oh notation
- To learn how to estimate and compare the performance of algorithms
- To learn how to measure the running time of a program

One of the most common tasks in data processing is sorting. For example, a collection of employees may need to be printed out in alphabetical order or sorted by salary. We will study several sorting methods in this chapter and compare their performance. This is by no means an exhaustive treatment of the subject of sorting. You will likely revisit this topic at a later time in your computer science studies. A good overview of the many sorting methods available can be found in [1].

Once a sequence of objects is sorted, one can locate individual objects rapidly. We will study the *binary search* algorithm, which carries out this fast lookup.

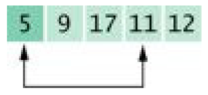
14.1 Selection Sort

In this section, we show you the first of several sorting algorithms. A *sorting algorithm* rearranges the elements of a collection so that they are stored in sorted order. To keep the examples simple, we will discuss how to sort an array of integers before going on to sorting strings or more complex data. Consider the following array `a`:



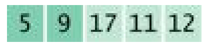
The selection sort algorithm sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.

An obvious first step is to find the smallest element. In this case the smallest element is 5, stored in `a [3]`. We should move the 5 to the beginning of the array. Of course, there is already an element stored in `a[0]`, namely 11. Therefore we cannot simply move `a [3]` into `a[0]` without moving the 11 somewhere else. We don't yet know where the 11 should end up, but we know for certain that it should not be in `a [0]`. We simply get it out of the way by *swapping* it with `a [3]`.



Now the first element is in the correct place. In the foregoing figure, the darker color indicates the portion of the array that is already sorted.

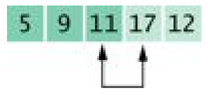
Next we take the minimum of the remaining entries `a[1] . . . a[4]`. That minimum value, 9, is already in the correct place. We don't need to do anything in this case and can simply extend the sorted area by one to the right:



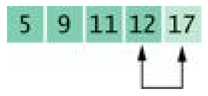
628

629

Repeat the process. The minimum value of the unsorted region is 11, which needs to be swapped with the first value of the unsorted region, 17:



Now the unsorted region is only two elements long, but we keep to the same successful strategy. The minimum value is 12, and we swap it with the first value, 17.



That leaves us with an unprocessed region of length 1, but of course a region of length 1 is always sorted. We are done.

Let us program this algorithm. For this program, as well as the other programs in this chapter, we will use a utility method to generate an array with random entries. We place it into a class `ArrayUtil` so that we don't have to repeat the code in every example. To show the array, we call the static `toString` method of the `Arrays` class in the Java library and print the resulting string.

This algorithm will sort any array of integers. If speed were not an issue, or if there simply were no better sorting method available, we could stop the discussion of sorting right here. As the next section shows, however, this algorithm, while entirely correct, shows disappointing performance when run on a large data set.

Advanced Topic 14.1 discusses insertion sort, another simple (and similarly inefficient) sorting algorithm.

ch14/selsort/SelectionSorter.java

```java
/**
   This class sorts an array, using the selection sort
   algorithm.
*/
public class SelectionSorter
{
   /**
      Constructs a selection sorter.
      @param anArray the array to sort
   */
   public SelectionSorter(int[] anArray)
   {
      a = anArray;
   }

   /**
      Sorts the array managed by this selection sorter.
   */
   public void sort()
   {
      for (int i = 0; i < a.length - 1; i++)
      {
         int minPos = minimumPosition(i);
```

```
            swap(minPos, i);
        }
    }

    /**
    Finds the smallest element in a tail range of the array.
        @param from the first position in a to compare
        @return the position of the smallest element in the
        range a[from] . . . a[a.length - 1]
    */
    private int minimumPosition (int from)
    {
        int minPos = from;
        for (int i = from + 1; i < a.length; i++)
           if (a[i] < a[minPos]) minPos = i;
        return minPos;
    }

    /**
    Swaps two entries of the array.
        @param i the first position to swap
        @param j the second position to swap
    */
    private void swap(int i, int j)
     {
        int temp = a[i] ;
        a[i] = a[j];
        a[j] = temp;
     }

    private int[] a;
}
```

ch14/selsort/SelectionSortDemo.java

```
import java. util.Arrays;

/**
This program demonstrates the selection sort algorithm by
 sorting an array that is filled with random numbers.
*/
```

```
public class SelectionSortDemo
 {
     public static void main(String[] args)
     {
         int[] a = ArrayUtil.randomIntArray(20, 100);
         System.out.println(Arrays.toString(a));

         SelectionSorter sorter = new SelectionSorter(a);
         sorter.sort();

         System.out.println(Arrays.toString(a));
     }
 }
```

ch14/selsort/ArrayUtil.java

```
import java.util.Random;

/**
   This class contains utility methods for array manipulation.
*/
public class ArrayUtil
{
   /**
      Creates an array filled with random values.
         @param length the length of the array
         @param n the number of possible random values
         @return an array filled with length numbers between
      0 and n - 1
    */
    public static int[] randomIntArray(int length, int n)
    {
         int[] a = new int[length];
         for (int i = 0; i < a.length; i++)
            a[i] = generator.nextInt(n);

         return a;
    }
```

```

           private static Random generator = new
Random();
}
```

Typical Output

```
[65, 46, 14, 52, 38, 2, 96, 39, 14, 33, 13, 4, 24,
99, 89, 77, 73, 87, 36, 81]
[2, 4, 13, 14, 14, 24, 33, 36, 38, 39, 46, 52, 65,
73, 77, 81, 87, 89, 96, 99]
```

SELF CHECK

1. Why do we need the `temp` variable in the `swap` method? What would happen if you simply assigned `a[i]` to `a[j]` and `a[j]` to `a[i]`?

2. What steps does the selection sort algorithm go through to sort the sequence 6 5 4 3 2 1?

14.2 Profiling the Selection Sort Algorithm

To measure the performance of a program, you could simply run it and measure how long it takes by using a stopwatch. However, most of our programs run very quickly, and it is not easy to time them accurately in this way. Furthermore, when a program takes a noticeable time to run, a certain amount of that time may simply be used for loading the program from disk into memory (for which we should not penalize it) or for screen output (whose speed depends on the computer model, even for computers with identical CPUs). We will instead create a `StopWatch` class. 631

632

This class works like a real stopwatch. You can start it, stop it, and read out the elapsed time. The class uses the `System.currentTimeMillis` method, which returns the milliseconds that have elapsed since midnight at the start of January 1, 1970. Of course, you don't care about the absolute number of seconds since this historical moment, but the *difference* of two such counts gives us the number of milliseconds of a time interval. Here is the code for the `StopWatch` class:

ch14/selsort/StopWatch.java

```
/**
   A stopwatch accumulates time when it is running. You can
   repeatedly start and stop the stopwatch. You can use a
   stopwatch to measure the running time of a program.
*/
public class StopWatch
{
   /**
      Constructs a stopwatch that is in the stopped state
      and has no time accumulated.
   */
   public StopWatch()
   {
      reset();
   }

   /**
      Starts the stopwatch. Time starts accumulating now.
   */
   public void start()
   {
      if (isRunning) return;
      isRunning = true;
      startTime = System.currentTimeMillis();
   }

   /**
      Stops the stopwatch. Time stops accumulating and is
      is added to the elapsed time.
   */
   public void stop()
   {
      if (!isRunning) return;
      isRunning = false;
      long endTime = System.currentTimeMillis();
      elapsedTime = elapsedTime + endTime - startTime;
   }

   /**
      Returns the total elapsed time.
      @return the total elapsed time
   */
   public long getElapsedTime()
   {
      if (isRunning)
```

632

633

```
      {
         long endTime = System.currentTimeMillis();
         return elapsedTime + endTime - startTime;
      }
      else
         return elapsedTime;
   }

   /**
      Stops the watch and resets the elapsed time to 0.
   */
   public void reset()
   {
      elapsedTime = 0;
      isRunning = false;
   }

   private long elapsedTime;
   private long startTime;
   private boolean isRunning;
}
```

Here is how we will use the stopwatch to measure the performance of the sorting algorithm:

ch14/selsort/SelectionSortTimer.java

```
import java.util.Scanner;

/**
   This program measures how long it takes to sort an
   array of a user-specified size with the selection
```

```
    sort algorithm.
*/
public class SelectionSortTimer
{
    public static void main(String[] args)
    {
            Scanner in = new Scanner(System.in);
            System.out.print("E ter array size: ");
            int n = in.nextInt();

    // Construct random array

            int[] a = ArrayUtil.randomIntArray(n, 100);
            SelectionSorter sorter = new SelectionSorter(a) ;

    // Use stopwatch to time selection sort

            StopWatch timer = new StopWatch();

            timer.start();
            sorter.sort();
            timer.stop();

        System.out.println("Elapsed time: "
                + timer.getElapsedTime() + "milliseconds");
    }
}
```

633

634

Output

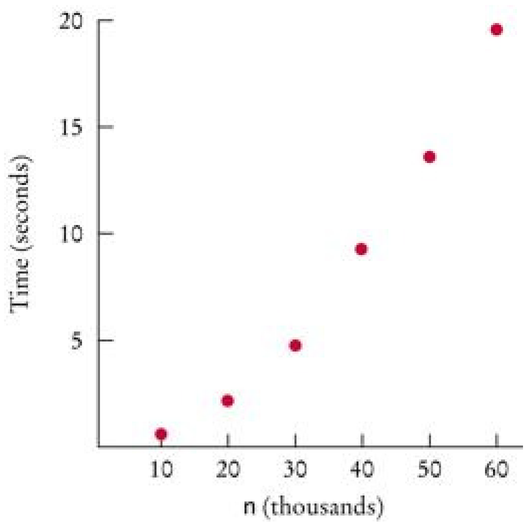
```
Enter array size: 100000
Elapsed time: 27880 milliseconds
```

By starting to measure the time just before sorting, and stopping the stopwatch just after, you don't count the time it takes to initialize the array or the time during which the program waits for the user to type in n.

Here are the results of some sample runs:

n	Milliseconds
10,000	786
20,000	2,148
30,000	4,796
40,000	9,192
50,000	13,321
60,000	19,299

Figure 1



Time Taken by Selection Sort

634

635

These measurements were obtained with a Pentium processor with a clock speed of 2 GHz, running Java 6 on the Linux operating system. On another computer the actual numbers will look different, but the relationship between the numbers will be the same. Figure 1 shows a plot of the measurements. As you can see, doubling the size of the data set more than doubles the time needed to sort it.

SELF CHECK

3. Approximately how many seconds would it take to sort a data set of 80,000 values?

4. Look at the graph in Figure 1. What mathematical shape does it resemble?

14.3 Analyzing the Performance of the Selection Sort Algorithm

Let us count the number of operations that the program must carry out to sort an array with the selection sort algorithm. We don't actually know how many machine operations are generated for each Java instruction or which of those instructions are more time-consuming than others, but we can make a simplification. We will simply count how often an array element is *visited*. Each visit requires about the same amount of work by other operations, such as incrementing subscripts and comparing values.

Let n be the size of the array. First, we must find the smallest of n numbers. To achieve that, we must visit n array elements. Then we swap the elements, which takes two visits. (You may argue that there is a certain probability that we don't need to swap the values. That is true, and one can refine the computation to reflect that observation. As we will soon see, doing so would not affect the overall conclusion.) In the next step, we need to visit only $n - 1$ elements to find the minimum. In the following step, $n - 2$ elements are visited to find the minimum. The last step visits two elements to find the minimum. Each step requires two visits to swap the elements. Therefore, the total number of visits is

$$\begin{aligned} n + 2 + (n - 1) + 2 + \ldots + 2 + 2 &= n + (n - 1) + \ldots + 2 + (n - 1) \cdot 2 \\ &= 2 + \ldots + (n - 1) + n + (n - 1) \cdot 2 \\ &= \frac{n(n + 1)}{2} - 1 + (n - 1) \cdot 2 \end{aligned}$$

because

$$1 + 2 + \ldots + (n - 1) + n = \frac{n(n + 1)}{2}$$

After multiplying out and collecting terms of n, we find that the number of visits is

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

635

636

We obtain a quadratic equation in *n*. That explains why the graph of Figure 1 looks approximately like a parabola.

Now let us simplify the analysis further. When you plug in a large value for *n* (for example, 1,000 or 2,000), then $\frac{1}{2}n^2$ is 500,000 or 2,000,000. The lower term, $\frac{5}{2}n - 3$, doesn't contribute much at all; it is only 2,497 or 4,997, a drop in the bucket compared to the hundreds of thousands or even millions of comparisons specified by the $\frac{1}{2}n^2$ term. We will just ignore these lower-level terms. Next, we will ignore the constant factor $\frac{1}{2}$. We are not interested in the actual count of visits for a single *n*. We want to compare the ratios of counts for different values of *n*. For example, we can say that sorting an array of 2,000 numbers requires four times as many visits as sorting an array of 1,000 numbers:

$$\frac{\left(\frac{1}{2} \cdot 2000^2\right)}{\left(\frac{1}{2} \cdot 1000^2\right)} = 4$$

The factor $\frac{1}{2}$ cancels out in comparisons of this kind. We will simply say, “The number of visits is of order n^2”. That way, we can easily see that the number of comparisons increases fourfold when the size of the array doubles: $(2n)^2 = 4n^2$.

To indicate that the number of visits is of order n^2, computer scientists often use *big-Oh notation*: The number of visits is $O(n^2)$. This is a convenient shorthand.

In general, the expression $f(n) = O(g(n))$ means that f grows no faster than g, or, more formally, that for all n larger than some thresh-old, the ratio $f(n)/g(n) \leq C$ for some constant value C. The function g is usually chosen to be very simple, such as n^2 in our example.

Computer scientists use the big-Oh notation $f(n) = O(g(n))$ to express that the function f grows no faster than the function g.

To turn an exact expression such as

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

into big-Oh notation, simply locate the fastest-growing term, n^2, and ignore its constant coefficient, no matter how large or small it may be.

We observed before that the actual number of machine operations, and the actual number of microseconds that the computer spends on them, is approximately proportional to the number of element visits. Maybe there are about 10 machine operations (increments, comparisons, memory loads, and stores) for every element visit. The number of machine operations is then approximately $10 \times \frac{1}{2}n^2$. Again, we aren't interested in the coefficient, so we can say that the number of machine operations, and hence the time spent on the sorting, is of the order of n^2 or $O(n^2)$.

The sad fact remains that doubling the size of the array causes a fourfold increase in the time required for sorting it with selection sort. When the size of the array increases by a factor of 100, the sorting time increases by a factor of 10,000. To sort an array of a million entries, (for example, to create a telephone directory) takes 10,000 times as long as sorting 10,000 entries. If 10,000 entries can be sorted in about 1/2 of a second (as in our example), then sorting one million entries requires well over an hour. We will see in the next section how one can dramatically improve the performance of the sorting process by choosing a more sophisticated algorithm.

Selection sort is an $O(n^2)$ algorithm. Doubling the data set means a fourfold increase in processing time.

SELF CHECK

5. If you increase the size of a data set tenfold, how much longer does it take to sort it with the selection sort algorithm?

6. How large does n need to be so that $\frac{1}{2}n^2$ is bigger than $\frac{5}{2}n - 3$?

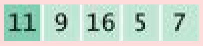
ADVANCED TOPIC 14.1: Insertion Sort

Insertion sort is another simple sorting algorithm. In this algorithm, we assume that the initial sequence

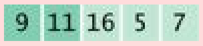
```
    a[0] a[1] . . . a[k]
```

of an array is already sorted. (When the algorithm starts, we set k to 0.) We enlarge the initial sequence by inserting the next array element, a[k + 1], at the proper location. When we reach the end of the array, the sorting process is complete.

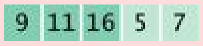
For example, suppose we start with the array



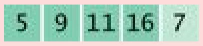
Of course, the initial sequence of length 1 is already sorted. We now add a[1], which has the value 9. The element needs to be inserted before the element 11. The result is



Next, we add a [2], which has the value 16. As it happens, the element does not have to be moved.



We repeat the process, inserting a[3] or 5 at the very beginning of the initial sequence.



Finally, a[4] or 7 is inserted in its correct position, and the sorting is completed.

The following class implements the insertion sort algorithm:

```
public class InsertionSorter
{
   /**
   Constructs an insertion sorter.
      @param anArray the array to sort
   */
   public InsertionSorter(int[] anArray)
   {
      a = anArray;
   }
```

```
      /**
         Sorts the array managed by this insertion
   sorter.
      */
      public void sort()
      {
         for (int i = 1; i < a.length; i ++)
         {
            int next = a[i];
            // Find the insertion location
            // Move all larger elements up
            int j = i;
            while (j > 0 && a[j - 1] > next)
            {
               a[j] = a[j - 1];
               j--;
            }
            // Insert the element
            a[j] = next;
         }
      }

      private int[] a;
   }
```

How efficient is this algorithm? Let *n* denote the size of the array. We carry out $n - 1$ iterations. In the *k*th iteration, we have a sequence of *k* elements that is already sorted, and we need to insert a new element into the sequence. For each insertion, we need to visit the elements of the initial sequence until we have found the location in which the new element can be inserted. Then we need to move up the remaining elements of the sequence. Thus, $k + 1$ array elements are visited. Therefore, the total number of visits is

$$2 + 3 + \ldots + n = \frac{n(n+1)}{2} - 1$$

We conclude that insertion sort is an $O(n^2)$ algorithm, on the same order of efficiency as selection sort.

Insertion sort is an $O(n^2)$ algorithm.

Insertion sort has one desirable property: Its performance is $O(n)$ if the array is already sorted—see Exercise R14.13. This is a useful property in practical applications, in which data sets are often partially sorted.

ADVANCED TOPIC 14.2: Oh, Omega, and Theta

We have used the big-Oh notation somewhat casually in this chapter, to describe the growth behavior of a function. Strictly speaking, $f(n) = O(g(n))$ means that f grows *no faster* than g. But it is permissible for f to grow much slower. Thus, it is technically correct to state that $f(n) = n^2 + 5n - 3$ is $O(n^3)$ or even $O(n^{10})$.

Computer scientists have invented additional notation to describe the growth behavior of functions more accurately. The expression

$$f(n) = \Omega(g(n))$$

means that f grows at least as fast as g, or, formally, that for all n larger than some threshold, the ratio $f(n)/g(n) \geq C$ for some constant value C. (The Ω symbol is the capital Greek letter omega.) For example, $f(n) = n^2 + 5n - 3$ is $\Omega(n^2)$ or even $\Omega(n)$.

The expression

$$f(n) = \Theta(g(n))$$

means that f and g grow at the same rate—that is, both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ hold. (The Θ symbol is the capital Greek letter theta.)

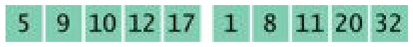
The Θ notation gives the most precise description of growth behavior. For example, $f(n) = n^2 + 5n - 3$ is $\Theta(n^2)$ but not $\Theta(n)$ or $\Theta(n^3)$.

The Ω and Θ notation is very important for the precise analysis of algorithms. However, in casual conversation it is common to stick with big-Oh, while still giving as good an estimate as one can.

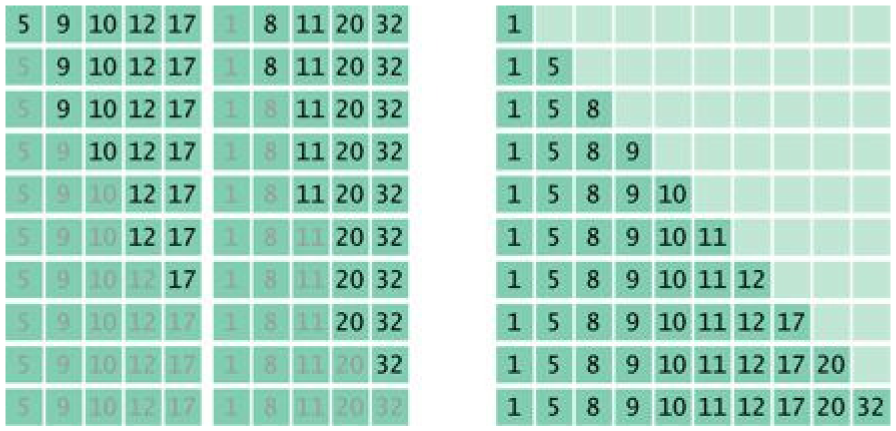
14.4 Merge Sort

In this section, you will learn about the merge sort algorithm, a much more efficient algorithm than selection sort. The basic idea behind merge sort is very simple.

Suppose we have an array of 10 integers. Let us engage in a bit of wishful thinking and hope that the first half of the array is already perfectly sorted, and the second half is too, like this:



Now it is simple to *merge* the two sorted arrays into one sorted array, by taking a new element from either the first or the second subarray, and choosing the smaller of the elements each time:



In fact, you probably performed this merging before when you and a friend had to sort a pile of papers. You and the friend split the pile in half, each of you sorted your half, and then you merged the results together.

The merge sort algorithm sorts an array by cutting the array in half, recursively sorting each half, and then merging the sorted halves.

639

640

That is all well and good, but it doesn't seem to solve the problem for the computer. It still must sort the first and second halves of the array, because it can't very well ask a few buddies to pitch in. As it turns out, though, if the computer keeps dividing the array into smaller and smaller subarrays, sorting each half and merging them back together, it carries out dramatically fewer steps than the selection sort requires.

Let us write a `MergeSorter` class that implements this idea. When the `MergeSorter` sorts an array, it makes two arrays, each half the size of the original, and sorts them recursively. Then it merges the two sorted arrays together:

```
public void sort()
{
   if (a.length <= 1) return;
   int[] first = new int[a.length / 2];
   int[] second = new int [a.length - first.length];
   System.arraycopy(a, 0, first, 0, first.length);
   System.arraycopy(a,
      first.length, second, 0, second.length);
   MergeSorter firstSorter = new MergeSorter(first);
   MergeSorter secondSorter = new
MergeSorter(second);
   firstSorter.sort();
   secondSorter.sort();
   merge(first, second);
}
```

The `merge` method is tedious but quite straightforward. You will find it in the code that follows.

ch14/mergesort/MergeSorter.java

```
/**
This class sorts an array, using the merge sort algorithm.
*/
public class MergeSorter
{
   /**
  Constructs a merge sorter.
         @param anArray the array to sort
   */
   public MergeSorter(int[] anArray)
   {
       a = anArray;
   }

   /**
  Sorts the array managed by this merge sorter.
```

```
   */
   public void sort()
   {
      if (a.length <= 1) return;
      int[] first = new int[a.length / 2];
      int[] second = new int[a.length - first.length];
      System.arraycopy(a, 0, first, 0, first.length);
      System.arraycopy(a, first.length, second, 0, second.length);
      MergeSorter firstSorter = new MergeSorter(first);
      MergeSorter secondSorter = new MergeSorter(second);
      firstSorter.sort();
      secondSorter.sort();
      merge(first, second);
   }

   /**
      Merges two sorted arrays into the array managed by this
      merge sorter.
      @param first the first sorted array
      @param second the second sorted array
   */
   private void merge(int[] first, int[] second)
   {
      // Merge both halves into the temporary array

      int iFirst = 0;
         // Next element to consider in the first array
      int iSecond = 0;
         // Next element to consider in the second array
      int j = 0;
         // Next open position in a

      // As long as neither iFirst nor iSecond past the end, move
      // the smaller element into a
      while (iFirst < first.length && iSecond < second.length)
      {
         if (first[iFirst] < second[iSecond])
        {
           a[j] = first[iFirst];
           iFirst++;
        }
        else
        {
           a[j] = second[iSecond];
           iSecond++;
        }
        j++;
      }

  // Note that only one of the two calls to arraycopy below
  // copies entries

  // Copy any remaining entries of the first array
      System.arraycopy(first, iFirst, a, j,
first.length - iFirst);

  // Copy any remaining entries of the second half
      System.arraycopy(second, iSecond, a, j,
second.length - iSecond);
      }

   private int[] a;
}
```

ch14/mergesort/MergeSortDemo.java

```
import java.util.Arrays;

/**
 This program demonstrates the merge sort algorithm by
 sorting an array that is filled with random numbers.
*/
public class MergeSortDemo
{
   public static void main(String[] args)
   {
        int[] a = ArrayUtil.randomIntArray(20,
100);
        System.out.println(Arrays.toString(a));
        MergeSorter sorter = new MergeSorter(a);
        sorter.sort();
        System.out.println(Arrays.toString(a));
    }
}
```

Typical Output

```
[8, 81, 48, 53, 46, 70, 98, 42, 27, 76, 33, 24, 2,
76, 62, 89, 90, 5, 13, 21]
[2, 5, 8, 13, 21, 24, 27, 33, 42, 46, 48, 53, 62,
70, 76, 76, 81, 89, 90, 98]
```

SELF CHECK

7. Why does only one of the two arraycopy calls at the end of the `merge` method do any work?

8. Manually run the merge sort algorithm on the array 8 7 6 5 4 3 2 1.

14.5 Analyzing the Merge Sort Algorithm

The merge sort algorithm looks a lot more complicated than the selection sort algorithm, and it appears that it may well take much longer to carry out these repeated subdivisions. However, the timing results for merge sort look much better than those for selection sort (see table on next page).

Figure 2 shows a graph comparing both sets of performance data. That is a tremendous improvement. To understand why, let us estimate the number of array element visits that are required to sort an array with the merge sort algorithm. First, let us tackle the merge process that happens after the first and second halves have been sorted.

Each step in the merge process adds one more element to `a`. That element may come from `first` or `second`, and in most cases the elements from the two halves must be compared to see which one to take. Let us count that as 3 visits (one for a and one each for `first` and `second`) per element, or $3n$ visits total, where n denotes the length of `a`. Moreover, at the beginning, we had to copy from a to `first` and `second`, yielding another $2n$ visits, for a total of $5n$.

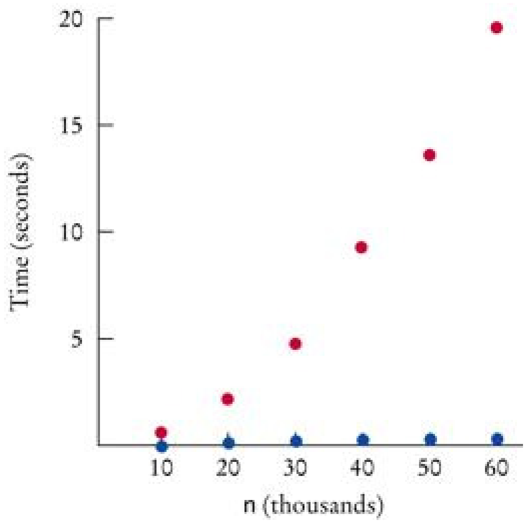
n	Merge Sort (milliseconds)	Selection Sort (milliseconds)
10,000	40	786
20,000	73	2,148
30,000	134	4,796
40,000	170	9,192
50,000	192	13,321
60,000	205	19,299

If we let $T(n)$ denote the number of visits required to sort a range of n elements through the merge sort process, then we obtain

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 5n$$

because sorting each half takes $T(n/2)$ visits. Actually, if n is not even, then we have one subarray of size $(n - 1)/2$ and one of size $(n + 1)/2$. Although it turns out that this detail does not affect the outcome of the computation, we will nevertheless assume for now that n is a power of 2, say $n = 2^m$. That way, all subarrays can be evenly divided into two parts.

Figure 2



Merge Sort Timing (blue) versus Selection Sort (red)

Unfortunately, the formula

$$T(n) = 2T\left(\frac{n}{2}\right) + 5n$$

does not clearly tell us the relationship between *n* and *T*(*n*). To understand the relationship, let us evaluate *T*(*n*/2), using the same formula:

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 5\frac{n}{2}$$

Therefore

$$T(n) = 2 \times 2T\left(\frac{n}{4}\right) + 5n + 5n$$

Let us do that again:

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 5\frac{n}{4}$$

hence

$$T(n) = 2 \times 2 \times 2T\left(\frac{n}{8}\right) + 5n + 5n + 5n$$

This generalizes from 2, 4, 8, to arbitrary powers of 2:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + 5nk$$

Recall that we assume that $n = 2^m$; hence, for $k = m$,

$$T(n) = 2^m T\left(\frac{n}{2^m}\right) + 5nm$$

$$= nT(1) + 5nm$$

$$= n + 5n \log_2(n)$$

Because $n = 2^m$, we have $m = \log_2(n)$.

To establish the growth order, we drop the lower-order term n and are left with $5n \log_2(n)$. We drop the constant factor 5. It is also customary to drop the base of the logarithm, because all logarithms are related by a constant factor. For example,

$$\log_2(x) = \log_{10}(x) / \log_{10}(2) \approx \log_{10}(x) \times 3.32193$$

Hence we say that merge sort is an $O(n \log(n))$ algorithm.

Merge sort is an $O(n \log(n))$ algorithm. The $n \log(n)$ function grows much more slowly than n^2.

644

645

Is the $O(n \log(n))$ merge sort algorithm better than the $O(n^2)$ selection sort algorithm? You bet it is. Recall that it took $100^2 = 10{,}000$ times as long to sort a million records as it took to sort 10,000 records with the $O(n^2)$ algorithm. With the $O(n \log(n))$ algorithm, the ratio is

$$\frac{1{,}000{,}000 \ \log\ (1{,}000{,}000)}{10{,}000 \ \log\ (10{,}000)} = 100\left(\frac{6}{4}\right) = 150$$

Suppose for the moment that merge sort takes the same time as selection sort to sort an array of 10,000 integers, that is, 3/4 of a second on the test machine. (Actually, it is much faster than that.) Then it would take about 0.75×150 seconds, or under 2 minutes, to sort a million integers. Contrast that with selection sort, which would take over 2 hours for the same task. As you can see, even if it takes you several hours to learn about a better algorithm, that can be time well spent.

In this chapter we have barely begun to scratch the surface of this interesting topic. There are many sorting algorithms, some with even better performance than the merge sort algorithm, and the analysis of these algorithms can be quite challenging. If you are a computer science major, you may revisit these important issues in a later computer science class.

The Arrays class implements a sorting method that you should use for your Java programs.

However, when you write Java programs, you don't have to implement your own sorting algorithm. The `Arrays` class contains static `sort` methods to sort arrays of integers and floating-point numbers. For example, you can sort an array of integers simply as

```
int[] a = . . .;
Arrays.sort(a);
```

That `sort` method uses the quicksort algorithm—see Advanced Topic 14.3 for more information about that algorithm.

SELF CHECK

9. Given the timing data for the merge sort algorithm in the table at the beginning of this section, how long would it take to sort an array of 100,000 values?

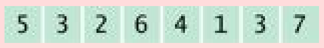
10. Suppose you have an array `double [] values` in a Java program. How would you sort it?

ADVANCED TOPIC 14.3: The Quicksort Algorithm

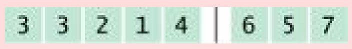
Quicksort is a commonly used algorithm that has the advantage over merge sort that no temporary arrays are required to sort and merge the partial results.

The quicksort algorithm, like merge sort, is based on the strategy of divide and conquer. To sort a `range a[from] . . . a[to]` of the array `a`, first 645
rearrange the elements in the range so that no element in the range `a[from] . . . a[p]` is larger than any element in the range `a[p + 1] . . . a[to]`. This step is called *partitioning* the range. 646

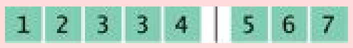
For example, suppose we start with a range



Here is a partitioning of the range. Note that the partitions aren't yet sorted.



You'll see later how to obtain such a partition. In the next step, sort each partition, by recursively applying the same algorithm on the two partitions. That sorts the entire range, because the largest element in the first partition is at most as large as the smallest element in the second partition.

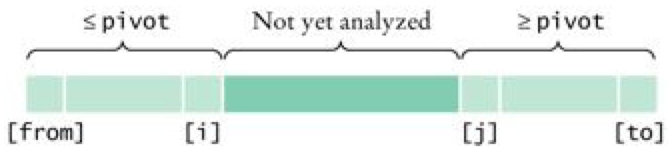


Quicksort is implemented recursively as follows:

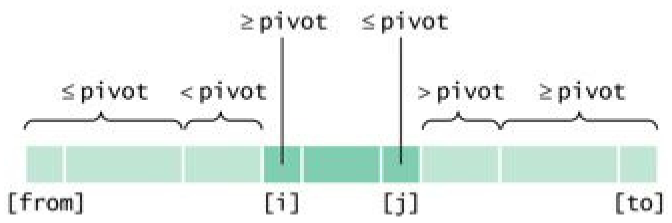
```
public void sort(int from, int to)
{
   if (from >= to) return;
   int p = partition(from, to);
   sort(from, p);
   sort(p + 1, to);
}
```

Let us return to the problem of partitioning a range. Pick an element from the range and call it the *pivot*. There are several variations of the quicksort algorithm. In the simplest one, we'll pick the first element of the range, `a[from]`, as the pivot.

Now form two regions `a[from] . . . a[i]`, consisting of values at most as large as the pivot and `a[j] . . . a[to]`, consisting of values at least as large as the pivot. The region `a[i + 1] . . . a[j - 1]` consists of values that haven't been analyzed yet. (See Partitioning a Range.) At the beginning, both the left and right areas are empty; that is, `i = from - 1` and `j = to + 1`.



Partitioning a Range



Extending the Partitions

Then keep incrementing `i` while `a[i] < pivot` and keep decrementing `j` while `a[j] > pivot`. Extending the Partitions shows `i` and `j` when that process stops.

Now swap the values in positions `i` and `j`, increasing both areas once more. Keep going while `i < j`. Here is the code for the `partition` method:

```
private int partition (int from, int to)
 {
    int pivot = a[from];
    int i = from - 1;
    int j = to + 1;
    while (i < j)
    {
       i++; while (a[i] < pivot) i++;
       j--; while (a[j] > pivot) j--;
       if (i < j) swap(i, j);
    }
   return j;
 }
```

On average, the quicksort algorithm is an $O(n \log(n))$ algorithm. Because it is simpler, it runs faster than merge sort in most cases. There is just one unfortunate aspect to the quicksort algorithm. Its *worst-case* runtime behavior is $O(n^2)$. Moreover, if the pivot element is chosen as the first element of the region, that worst-case behavior occurs when the input set is already sorted—a common situation in practice. By selecting the pivot element more cleverly, we can make it extremely unlikely for the worst-case behavior to occur. Such "tuned" quicksort algorithms are commonly used, because their performance is generally excellent. For example, as was mentioned, the `sort` method in the `Arrays` class uses a quicksort algorithm.

Another improvement that is commonly made in practice is to switch to insertion sort when the array is short, because the total number of operations of insertion sort is lower for short arrays. The Java library makes that switch if the array length is less than 7.

RANDOM FACT 14.1: The First Programmer

Before pocket calculators and personal computers existed, navigators and engineers used mechanical adding machines, slide rules, and tables of logarithms and trigonometric functions to speed up computations. Unfortunately, the tables—for which values had to be computed by hand—were notoriously inaccurate. The mathematician Charles Babbage (1791—1871) had the insight that if a machine could be constructed that produced printed tables automatically, both calculation and typesetting errors could be avoided. Babbage set out to develop a machine for this purpose, which he called a *Difference Engine* because it used successive differences to compute polynomials. For example, consider the function $f(x) = x^3$. Write down the values for $f(1)$, $f(2)$, $f(3)$, and so on. Then take the *differences* between successive values:

```
1
        7
8
        19
27
        37
64
        61
125
        91
216
```

Repeat the process, taking the difference of successive values in the second column, and then repeat once again:

```
1
          7
8                 12
          19                  6
27                18
```

```
                 37                   6
    64                  24
                 61                   6
    125                  30
                 91
    216
```

Now the differences are all the same. You can retrieve the function values by a pattern of additions—you need to know the values at the fringe of the pattern and the constant difference. This method was very attractive, because mechanical addition machines had been known for some time. They consisted of cog wheels, with 10 cogs per wheel, to represent digits, and mechanisms to handle the carry from one digit to the next. Mechanical multiplication machines, on the other hand, were fragile and unreliable. Babbage built a successful prototype of the Difference Engine (see the Babbage's Difference Engine figure) and, with his own money and government grants, proceeded to build the table-printing machine. However, because of funding problems and the difficulty of building the machine to the required precision, it was never completed.

Babbage's Difference Engine

While working on the Difference Engine, Babbage conceived of a much grander vision that he called the *Analytical Engine*. The Difference Engine was designed to carry out a limited set of computations—it was no smarter than a pocket calculator is today. But Babbage realized that such a machine could be made *programmable* by storing programs as well as data. The internal storage of the Analytical Engine was to consist of 1,000 registers of 50 decimal digits each. Programs and constants were to be stored on punched cards—a technique that was, at that time, commonly used on looms for weaving patterned fabrics.

Ada Augusta, Countess of Lovelace (1815—1852), the only child of Lord Byron, was a friend and sponsor of Charles Babbage. Ada Lovelace was one of the first people to realize the potential of such a machine, not just for computing mathematical tables but for processing data that were not numbers. She is considered by many the world's first programmer. The Ada programming language, a language developed for use in U.S. Department of Defense projects (see Random Fact 9.2), was named in her honor.

14.6 Searching

Suppose you need to find the telephone number of your friend. You look up his name in the telephone book, and naturally you can find it quickly, because the telephone book is sorted alphabetically. Quite possibly, you may never have thought how important it is that the telephone book is sorted. To see that, think of the following problem: Suppose you have a telephone number and you must know to what party it belongs. You could of course call that number, but suppose nobody picks up on the other end. You could look through the telephone book, a number at a time, until you find the number. That would obviously be a tremendous amount of work, and you would have to be desperate to attempt that.

This thought experiment shows the difference between a search through an unsorted data set and a search through a sorted data set. The following two sections will analyze the difference formally.

If you want to find a number in a sequence of values that occur in arbitrary order, there is nothing you can do to speed up the search. You must simply look through all elements until you have found a match or until you reach the end. This is called a *linear* or *sequential search*.

A linear search examines all values in an array until it finds a match or reaches the end.

How long does a linear search take? If we assume that the element `v` is present in the array `a`, then the average search visits *n*/2 elements, where *n* is the length of the array. If it is not present, then all *n* elements must be inspected to verify the absence. Either way, a linear search is an $O(n)$ algorithm.

A linear search locates a value in an array in $O(n)$ steps.

Here is a class that performs linear searches through an array `a` of integers. When searching for the value `v`, the `search` method returns the first index of the match, or `-1` if `v` does not occur in `a`.

ch14/linsearch/LinearSearcher.java

```
/**
A class for executing linear searches through an array.
*/
public class LinearSearcher
{
        /**
    Constructs the LinearSearcher.
            @param anArray an array of integers
        */
       public LinearSearcher(int[] anArray)
       {
             a = anArray;
       }

        /**
    Finds a value in an array, using the linear search
    algorithm.
            @param v the value to search
            @return the index at which the value occurs, or -1
    if it does not occur in the array
        */
    public int search(int v)
    {
        for (int i = 0; i < a.length; i++)
        {
            if (a[i] == v)
                return i;
        }
        return -1;
    }

    private int[] a;
}
```

ch14/linsearch/LinearSearchDemo.java

```
import java.util.Arrays;
import java.util.Scanner;

/**
   This program demonstrates the linear search algorithm.
*/
public class LinearSearchDemo
{
   public static void main(String[] args)
   {
      int[] a = ArrayUtil.randomIntArray(20, 100);
      System.out.println(Arrays.toString(a));
      LinearSearcher searcher = new LinearSearcher(a);

      Scanner in = new Scanner(System.in);

      boolean done = false;
      while (!done)
      {
         System.out.print("Enter number to search for, -1 to quit: ");
         int n = in.nextInt();
         if (n == -1)
            done = true;
         else
```

650

651

```
            {
                  int pos = searcher.search(n);
                 System.out.
println("Foundinposition" + pos);
            }
         }
      }
   }
```

Typical Output

```
  [46, 99, 45, 57, 64, 95, 81, 69, 11, 97, 6, 85,
61, 88, 29, 65, 83, 88, 45, 88]
  Enter number to search for, -1 to quit: 11
  Found in position 8
```

SELF CHECK

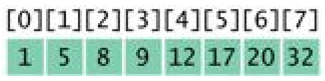
11. Suppose you need to look through 1,000,000 records to find a telephone number. How many records do you expect to search before finding the number?

12. Why can't you use a "for each" loop for (`int element : a`) in the `search` method?

14.7 Binary Search

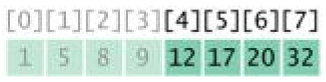
Now let us search for an item in a data sequence that has been previously sorted. Of course, we could still do a linear search, but it turns out we can do much better than that.

Consider the following sorted array `a`. The data set is:

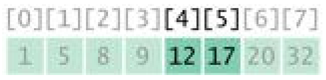


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

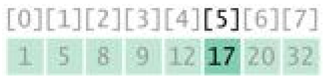
We would like to see whether the value 15 is in the data set. Let's narrow our search by finding whether the value is in the first or second half of the array. The last point in the first half of the data set, `a [3]`, is `9`, which is smaller than the value we are 651 looking for. Hence, we should look in the second half of the array for a match, that is, in the sequence:



Now the last value of the first half of this sequence is 17; hence, the value must be located in the sequence:



The last value of the first half of this very short sequence is 12, which is smaller than the value that we are searching, so we must look in the second half:



It is trivial to see that we don't have a match, because 15 ≠ 17. If we wanted to insert 15 into the sequence, we would need to insert it just before `a[5]`.

A binary search locates a value in a sorted array by determining whether the value occurs in the first or second half, then repeating the search in one of the halves.

This search process is called a *binary search*, because we cut the size of the search in half in each step. That cutting in half works only because we know that the sequence of values is sorted.

The following class implements binary searches in a sorted array of integers. The `search` method returns the position of the match if the search succeeds, or `-1` if `v` is not found in `a`.

ch14/binsearch/BinarySearcher.java

```
/**
   A class for executing binary searches through an array.
*/
public class BinarySearcher
{
    /**
    Constructs a BinarySearcher.
         @param anArray a sorted array of integers
    */
   public BinarySearcher(int[] anArray)
   {
         a = anArray;
   }

   /**
   Finds a value in a sorted array, using the binary
   search algorithm.
         @param v the value to search
         @return the index at which the value occurs, or -1
   if it does not occur in the array
   */
   public int search(int v)
   {
          int low = 0;
          int high = a.length - 1;
          while (low <= high)
          {
                 int mid = (low + high) / 2;
                 int diff = a [mid] - v;

                 if (diff == 0) // a[mid] == v
                     return mid;
                 else if (diff < 0) // a[mid] < v
                        low = mid + 1;
                 else
                        high = mid - 1;
      }
      return -1;
   }

   private int[] a;
}
```

Let us determine the number of visits of array elements required to carry out a search. We can use the same technique as in the analysis of merge sort. Because we look at the middle element, which counts as one comparison, and then search either the left or the right subarray, we have

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

Using the same equation,

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1$$

By plugging this result into the original equation, we get

$$T(n) = T\left(\frac{n}{4}\right) + 2$$

That generalizes to

$$T(n) = T\left(\frac{n}{2^k}\right) + k$$

As in the analysis of merge sort, we make the simplifying assumption that n is a power of 2, $n = 2^m$, where $m = \log_2(n)$. Then we obtain

$$T(n) = 1 + \log_2(n)$$

Therefore, binary search is an $O(\log(n))$ algorithm.

653

654

That result makes intuitive sense. Suppose that n is 100. Then after each search, the size of the search range is cut in half, to 50, 25, 12, 6, 3, and 1. After seven comparisons we are done. This agrees with our formula, because $\log_2(100) \approx 6.64386$, and indeed the next larger power of 2 is $2^7 = 128$.

A binary Search locates a value in an array in $O(\log(n))$ steps.

Because a binary search is so much faster than a linear search, is it worthwhile to sort an array first and then use a binary search? It depends. If you search the array only once, then it is more efficient to pay for an $O(n)$ linear search than for an $O(n \log(n))$ sort and an $O(\log(n))$ binary search. But if you will be making many searches in the same array, then sorting it is definitely worthwhile.

The Arrays class contains a static binarySearch method that implements the binary search algorithm, but with a useful enhancement. If a value is not found in the array, then the returned value is not − 1, but − k − 1, where k is the position before which the element should be inserted. For example,

```
int[] a = { 1, 4, 9 };
int v = 7;
int pos = Arrays.binarySearch(a, v);
   // Returns -3; v should be inserted before
position 2
```

SELF CHECK

13. Suppose you need to look through a sorted array with 1,000,000 elements to find a value. Using the binary search algorithm, how many records do you expect to search before finding the value?

14. Why is it useful that the Arrays.binarySearch method indicates the position where a missing element should be inserted?

15. Why does Arrays.binarySearch return − k − 1 and not − k to indicate that a value is not present and should be inserted before position k?

14.8 Sorting Real Data

In this chapter we have studied how to search and sort arrays of integers. Of course, in application programs, there is rarely a need to search through a collection of integers. However, it is easy to modify these techniques to search through real data.

The sort method of the Arrays class sorts objects of classes that implement the Comparable interface.

The Arrays class supplies a static sort method for sorting arrays of objects. However, the Arrays class cannot know how to compare arbitrary objects. Suppose, for example, that you have an array of Coin objects. It is not obvious how the coins should be sorted. You could sort them by their names, or by their values. The Arrays. sort method cannot make that decision for you. Instead, it requires that the objects belong to classes that implement the `Comparable` interface. That interface has a single method:

```
public interface Comparable
{
  int compareTo(Object otherObject);
}
```

The call

```
a.compareTo(b)
```

must return a negative number if `a` should come before `b`, 0 if `a` and `b` are the same, and a positive number otherwise.

Several classes in the standard Java library, such as the `String` and `Date` classes, implement the `Comparable` interface.

You can implement the `Comparable` interface for your own classes as well. For example, to sort a collection of coins, the `Coi n` class would need to implement this interface and define a `compareTo` method:

```
public class Coin implements Comparable
{
   . . .
   public int compareTo(Object otherObject)
   {
        Coin other = (Coin) otherObject;
        if (value < other.value) return -1;
        if (value == other.value) return 0;
        return 1;
   }
   . . .
}
```

When you implement the `compareTo` method of the `Comparable` interface, you must make sure that the method defines a *total ordering relationship*, with the following three properties:

- *Antisymmetric*: If `a.compareTo(b)` ≤ 0, then `b.compareTo(a)` ≥ 0
- *Reflexive*: `a.compareTo(a)` = 0

- *Transitive*: If `a.compareTo(b)` ≤ 0 and `b.compareTo(c)` ≤ 0, then `a.compareTo(c)` ≤ 0

Once your `Coin` class implements the `Comparable` interface, you can simply pass an array of coins to the `Arrays. sort` method:

```
Coin[] coins = new Coin[n];
// Add coins
. . .
Arrays.sort(coins);
```

If the coins are stored in an `ArrayList`, use the `Collections.sort` method instead; it uses the merge sort algorithm:

The `Collections` class contains a sort method that can sort array lists.

```
ArrayList<Coin> coins = new ArrayList<Coin>();
// Add coins
. . .
Collections. sort (coins);
```

As a practical matter, you should use the sorting and searching methods in the `Arrays` and `Collections` classes and not those that you write yourself. The library algorithms have been fully debugged and optimized. Thus, the primary purpose of this chapter was not to teach you how to implement practical sorting and 655
searching algorithms. Instead, you have learned something more important, namely 656
that different algorithms can vary widely in performance, and that it is worthwhile to learn more about the design and analysis of algorithms.

SELF CHECK

16. Why can't the `Arrays.sort` method sort an array of `Rectangle` objects?

17. What steps would you need to take to sort an array of `BankAccount` objects by increasing balance?

Common Error 14.1: The `compareTo` Method Can Return Any Integer, Not Just – 1, 0, and 1

The `call a.compareTo(b)` is allowed to return *any* negative integer to denote that `a` should come before `b`, not necessarily the value – 1. That is, the test

```
if (a.compareTo(b) == -1) // ERROR!
```

is generally wrong. Instead, you should test

```
if (a.compareTo(b) < 0) // OK
```

Why would a `compareTo` method ever want to return a number other than – 1, 0, or 1 ? Sometimes, it is convenient to just return the difference of two integers. For example, the `compareTo` method of the `Stri ng class` compares characters in matching positions:

```
char c1 = charAt(i);
char c2 = other.charAt(i);
```

If the characters are different, then the method simply returns their difference:

```
if (c1 ! = c2) return c1 - c2;
```

This difference is a negative number if `c1` is less than `c2`, but it is not necessarily the number – 1.

Advanced Topic 14.4: The Parameterized `Comparable` Interface

As of Java version 5.0, the `Comparabl e` interface is a parameterized type, similar to the `Array-List` type:

```
public interface Comparable<T>
{
        int compareTo(T other)
}
```

The type parameter specifies the type of the objects that this class is willing to accept for comparison. Usually, this type is the same as the class type itself. For example, the `Coin` class would implement `Comparable<Coin>`, like this:

```
public class Coin implements Comparable<Coin>
{
   . . .
   public int compareTo(Coin other)
   {
        if (value < other.value) return -1;
        if (value == other.value) return 0;
        return 1;
   }
   . . .
}
```

The type parameter has a significant advantage: You need not use a cast to convert an `Object` parameter into the desired type.

ADVANCED TOPIC 14.5: The Comparator Interface

Sometimes, you want so sort an array or array list of objects, but the objects don't belong to a class that implements the `Comparabl e` interface. Or, perhaps, you want to sort the array in a different order. For example, you may want to sort coins by name rather than by value.

You wouldn't want to change the implementation of a class just in order to call `Arrays.sort`. Fortunately, there is an alternative. One version of the `Arrays.sort` method does not require that the objects belong to classes that implement the `Comparable` interface. Instead, you can supply arbitrary objects. However, you must also provide a *comparator* object whose job is to compare objects. The comparator object must belong to a class that implements the `Comparator` interface. That interface has a single method, `compare`, which compares two objects.

As of Java version 5.0, the `Comparator` interface is a parameterized type. The type parameter specifies the type of the `compare` parameters. For example, `Comparator<Coin>` looks like this:

```
public interface Comparator<Coin>
```

```
{
   int compare (Coin a, Coin b);
}
```

The call

```
comp.compare(a, b)
```

must return a negative number if `a` should come before `b`, 0 if `a` and `b` are the same, and a positive number otherwise. (Here, `comp` is an object of a class that implements `Comparator<Coin>`.)

For example, here is a `Comparator` class for coins:

```
public class CoinComparator implements
Comparator<Coin>
{
   public int compare(Coin a, Coin b)
   {
        if (a.getValue() < b.getValue()) return -1;
        if (a.getValue() == b.getValue()) return 0;
        return 1;
   }
}
```

To sort an array of coins by value, call

```
Arrays.sort(coins, new CoinComparator());
```

CHAPTER SUMMARY

1. The selection sort algorithm sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.
2. Computer scientists use the big-Oh notation $f(n) = O(g(n))$ to express that the function f grows no faster than the function g.
3. Selection sort is an $O(n^2)$ lgorithm. Doubling the data set means a fourfold increase in processing time.
4. Insertion sort is an $O(n^2)$ algorithm.

5. The merge sort algorithm sorts an array by cutting the array in half, recursively sorting each half, and then merging the sorted halves.

6. Merge sort is an $O(n \log(n))$ algorithm. The $n \log(n)$ function grows much more slowly than n^2.

7. The `Arrays` class implements a sorting method that you should use for your Java programs.

8. A linear search examines all values in an array until it finds a match or reaches the end.

9. A linear search locates a value in an array in $O(n)$ steps.

10. A binary search locates a value in a sorted array by determining whether the value occurs in the first or second half, then repeating the search in one of the halves.

11. A binary search locates a value in an array in $O(\log(n))$ steps.

12. The `sort` method of the `Arrays` class sorts objects of classes that implement the `Comparable` interface.

13. The `Collections` class contains a `sort` method that can sort array lists.

FURTHER READING

1. Michael T. Goodrich and Roberto Tamassia, Data Structures and Algorithms in Java, 3rd edition,John Wiley & Sons, 2003.

CLASSES, OBJECTS, AND METHODS INTRODUCED IN THIS CHAPTER

```
java.lang.Comparable<T>
    compareTo
java.lang.System
    currentTimeMillis
java.util.Arrays
    binarySearch
    sort
    toString
java.util.Collections
    binarySearch
    sort
java.util.Comparator<T>
    compare
```

REVIEW EXERCISES

★★ **Exercise R14.1.** *Checking against off-by-one errors*. When writing the selection sort algorithm of Section 14.1, a programmer must make the usual choices of `<` against `< =` , `a.length` against `a.length - 1`, and `from` against `from + 1`. This is a fertile ground for off-by-one errors. Conduct code walkthroughs of the algorithm with arrays of length 0, 1,2, and 3 and check carefully that all index values are correct.

★ **Exercise R14.2.** What is the difference between searching and sorting?

★★ **Exercise R14.3.** For the following expressions, what is the order of the growth of each?

a. $n^2 + 2n + 1$

b. $n^{10} + 9n^9 + 20n^8 + 145n^7$

c. $(n + 1)^4$

d. $(n^2 + n)^2$

e. $n + 0.001n^3$

f. $n^3 - 1000n^2 + 10^9$

g. $n + \log(n)$

h. $n^2 + n \log(n)$

i. $2^n + n^2$

j. $\frac{n^3 + 2n}{n^2 + 0.75}$

★ **Exercise R14.4.** We determined that the actual number of visits in the selection sort algorithm is

$$T(n) = \frac{1}{2}n^2 + \frac{5}{2}n - 3$$

We characterized this method as having $O(n^2)$ growth. Compute the actual ratios

$$T(2{,}000) \;/\; T(1{,}000)$$

$$T(4{,}000) \;/\; T(1{,}000)$$

$$T(10{,}000) \;/\; T(1{,}000)$$

and compare them with

$$f(2{,}000) \;/ f(1{,}000)$$

$$f(4{,}000) \;/ f(1{,}000)$$

$$f(10{,}000) \;/ f(1{,}000)$$

where $f(n) = n^2$.

★ **Exercise R14.5.** Suppose algorithm A takes 5 seconds to handle a data set of 1,000 records. If the algorithm A is an $O(n)$ algorithm, how long will it take to handle a data set of 2,000 records? Of 10,000 records?

★★ **Exercise R14.6.** Suppose an algorithm takes 5 seconds to handle a data set of 1,000 records. Fill in the following table, which shows the approximate growth of the execution times depending on the complexity of the algorithm.

	$O(n)$	$O(n^2)$	$O(n^3)$	$O(n \log(n))$	$O(2^n)$
1,000	5	5	5	5	5
2,000					
3,000		45			
10,000					

For example, because $3{,}000^2/1{,}000^2 = 9$, the algorithm would take 9 times as long, or 45 seconds, to handle a data set of 3,000 records.

★★ **Exercise R14.7.** Sort the following growth rates from slowest to fastest growth.

$O(n)$	$O(n \log(n))$
$O(n^3)$	$O(2^n)$
$O(n^n)$	$O(\sqrt{n})$
$O(\log(n))$	$O(n\sqrt{n})$
$O(n^2 \log(n))$	$O(n^{\log(n)})$

★ **Exercise R14.8.** What is the growth rate of the standard algorithm to find the minimum value of an array? Of finding both the minimum and the maximum?

660

661

★ **Exercise R14.9.** What is the growth rate of the following method?

```
public static int count(int[] a, int c)
{
      int count = 0;
      for (int i = 0; i < a.length; i ++)
```

```
         {
            if (a[i] == c) count++;
         }
         return count;
   }
```

★★ **Exercise R14.10.** Your task is to remove all duplicates from an array. For example, if the array has the values

4 7 11 4 9 5 11 7 3 5

then the array should be changed to

4 7 11 9 5 3

Here is a simple algorithm. Look at `a[i]`. Count how many times it occurs in `a`. If the count is larger than 1, remove it. What is the growth rate of the time required for this algorithm?

★★ **Exercise R14.11.** Consider the following algorithm to remove all duplicates from an array. Sort the array. For each element in the array, look at its next neighbor to decide whether it is present more than once. If so, remove it. Is this a faster algorithm than the one in Exercise R14.10?

★★★ **Exercise R14.12.** Develop an $O(n \log(n))$ algorithm for removing duplicates from an array if the resulting array must have the same ordering as the original array.

★★★ **Exercise R14.13.** Why does insertion sort perform significantly better than selection sort if an array is already sorted?

★★★ **Exercise R14.14.** Consider the following speedup of the insertion sort algorithm of Advanced Topic 14.1. For each element, use the enhanced binary search algorithm that yields the insertion position for missing elements. Does this speedup have a significant impact on the efficiency of the algorithm?

Additional review exercises are available in WileyPLUS.

PROGRAMMING EXERCISES

★ **Exercise P14.1.** Modify the selection sort algorithm to sort an array of integers in descending order.

★ **Exercise P14.2.** Modify the selection sort algorithm to sort an array of coins by their value.

★★ **Exercise P14.3.** Write a program that generates the table of sample runs of the selection sort times automatically. The program should ask for the smallest and largest value of `n` and the number of measurements and then make all sample runs.

★ **Exercise P14.4.** Modify the merge sort algorithm to sort an array of strings in lexicographic order.

★★★ **Exercise P14.5.** Write a telephone lookup program. Read a data set of 1,000 names and telephone numbers from a file that contains the numbers in random order. Handle lookups by name and also reverse lookups by phone number. Use a binary search for both lookups.

★★ **Exercise P14.6.** Implement a program that measures the performance of the insertion sort algorithm described in Advanced Topic 14.1.

★★★ **Exercise P14.7.** Write a program that sorts an `ArrayList<Coin>` in decreasing order so that the most valuable coin is at the beginning of the array. Use a `Comparator`.

★★ **Exercise P14.8.** Consider the binary search algorithm in Section 14.7. If no match is found, the `search` method returns -1. Modify the method so that if a is not found, the method returns $-k-1$, where k is the position before which the element should be inserted. (This is the same behavior as `Arrays.binarySearch`.)

★★ **Exercise P14.9.** Implement the `sort` method of the merge sort algorithm without recursion, where the length of the array is a power of 2. First merge adjacent regions of size 1, then adjacent regions of size 2, then adjacent regions of size 4, and so on.

★★★ **Exercise P14.10.** Implement the `sort` method of the merge sort algorithm without recursion, where the length of the array is an arbitrary number. Keep merging adjacent regions whose size is a power of 2, and pay special attention to the last area whose size is less.

★★★ **Exercise P14.11.** Use insertion sort and the binary search from Exercise P14.8 to sort an array as described in Exercise R14.14. Implement this algorithm and measure its performance.

★ **Exercise P14.12.** Supply a class `Person` that implements the `Comparable` interface. Compare persons by their names. Ask the user to input 10 names and generate 10 `Person` objects. Using the `compareTo` method, determine the first and last person among them and print them.

★★ **Exercise P14.13.** Sort an array list of strings by increasing *length. Hint*: Supply a `Comparator`.

★★★ **Exercise P14.14.** Sort an array list of strings by increasing length, and so that strings of the same length are sorted lexicographically. *Hint*: Supply a `Comparator`.

Additional programming exercises are available in WileyPLUS.

PROGRAMMING PROJECTS

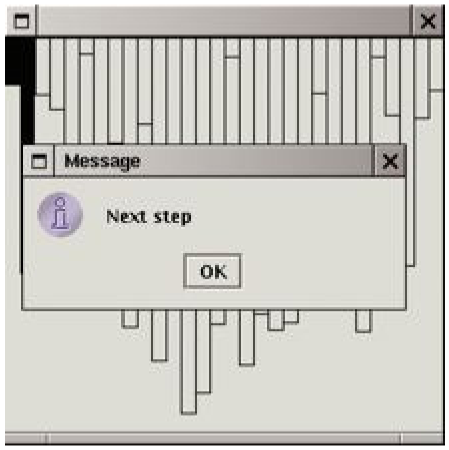
★★★ **Project 14.1.** Write a program that keeps an appointment book. Make a class `Appoi ntment` that stores a description of the appointment, the appointment day, the starting time, and the ending time. Your program should keep the appointments in a sorted array list. Users can add appointments and print out all appointments for a given day. When a new appointment is added, use binary search to find where it should be inserted in the array list. Do not add it if it conflicts with another appointment.

★★★G **Project 14.2.** Implement a *graphical animation* of sorting and searching algorithms. Fill an array with a set of random numbers between 1 and 100. Draw each array element as a bar, as in Figure 3.

Whenever the algorithm changes the array, wait for the user to click a button, then call the `repaint` method.

Animate selection sort, merge sort, and binary search. In the binary search animation, highlight the currently inspected element and the current values of `from` and `to`.

Figure 3



Graphical Animation

ANSWERS TO SELF-CHECK QUESTIONS

1. Dropping the `temp` variable would not work. Then `a[i]` and `a[j]` would end up being the same value.
2. 1|54326,12|4356,123456
3. Four times as long as 40,000 values, or about 50 seconds.
4. A parabola.
5. It takes about 100 times longer.
6. If n is 4, then $1/2n^2$ is 8 and $5/2n - 3$ is 7.

7. When the preceding `while` loop ends, the loop condition must be false, that is, `iFirst >= first.length` or `iSecond >= second.length` (De Morgan's Law). Then `first.length - iFirst <= 0` or `iSecond.length - iSecond <= 0`.

8. First sort 8 7 6 5. Recursively, first sort 8 7. Recursively, first sort 8. It's sorted. Sort 7. It's sorted. Merge them: 7 8. Do the same with 6 5 to get 5 6. Merge them to 5 6 7 8. Do the same with 4 3 2 1: Sort 4 3 by sorting 4 and 3 and merging them to 3 4. Sort 2 1 by sorting 2 and 1 and merging them to 1 2. Merge 3 4 and 1 2 to 1 2 3 4. Finally, merge 5 6 7 8 and 1 2 3 4 to 1 2 3 4 5 6 7 8.

9. Approximately 100,000 · log(100,000) / 50,000 · log(50,000) = 2 · 5 / 4.7 = 2.13 times the time required for 50,000 values. That's 2.13 · 97 milliseconds or approximately 207 milliseconds.

10. By calling `Arrays.sort(values).`

11. On average, you'd make 500,000 comparisons.

12. The `search` method returns the index at which the match occurs, not the data stored at that location.

13. You would search about 20. (The binary log of 1,024 is 10.)

14. Then you know where to insert it so that the array stays sorted, and you can keep using binary search.

15. Otherwise, you would not know whether a value is present when the method returns 0.

16. The `Rectangle` class does not implement the `Comparable` interface.

17. The `BankAccount` class needs to implement the `Comparable` interface. Its `compareTo` method must compare the bank balances.