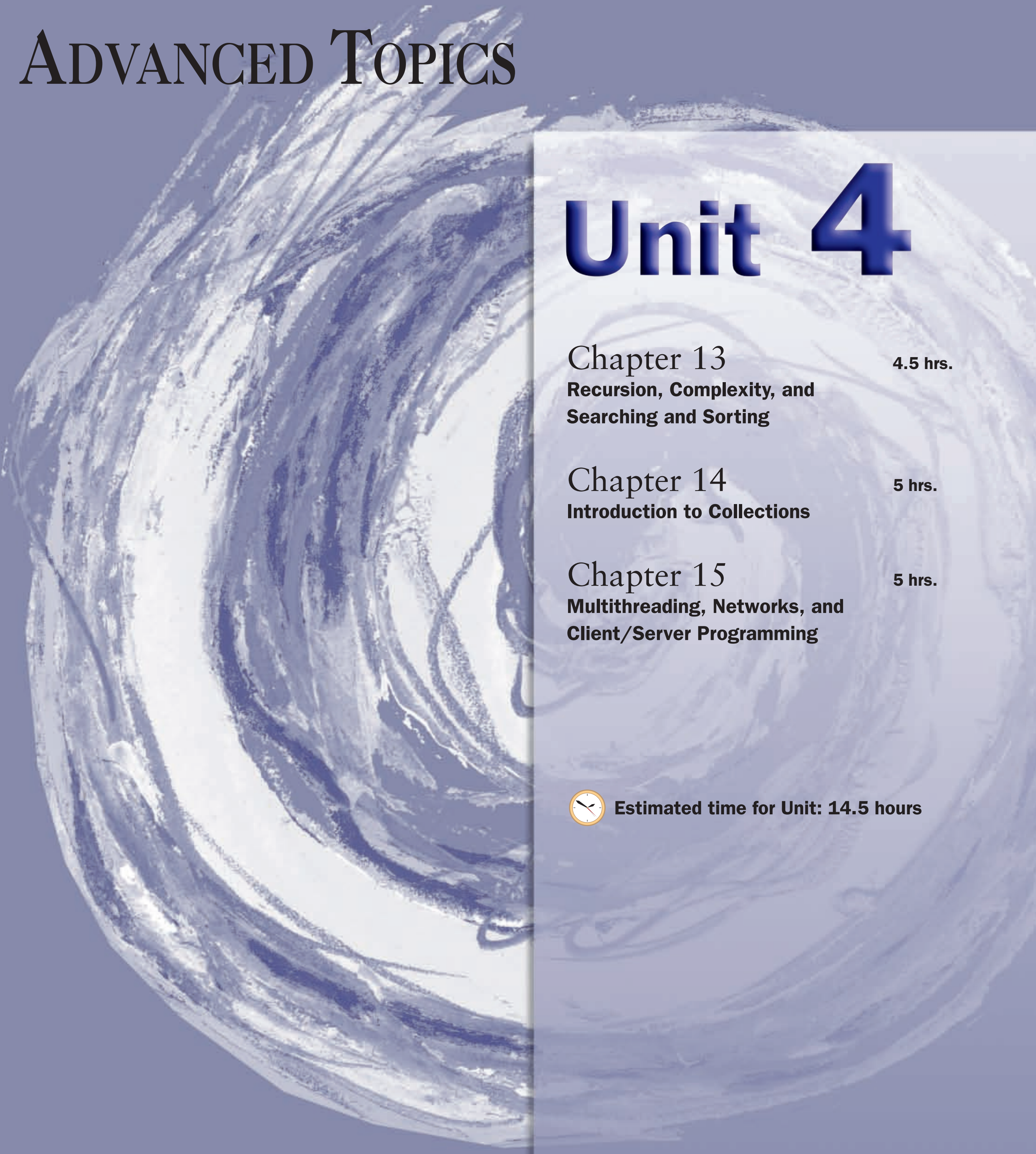


ADVANCED TOPICS

Unit 4

Estimated time for Unit: 14.5 hours

CHAPTER 13

RECURSION, COMPLEXITY, AND SEARCHING AND SORTING

OBJECTIVES

Upon completion of this chapter, you should be able to:

- Design and implement a recursive method to solve a problem.
- Understand the similarities and differences between recursive and iterative solutions of a problem.
- Check and test a recursive method for correctness.
- Understand how a computer executes a recursive method.
- Perform a simple complexity analysis of an algorithm using big-O notation.
- Recognize some typical orders of complexity.
- Understand the behavior of a complex sort algorithm such as the quicksort.

Estimated Time: 4.5 hours

VOCABULARY

Activation record
Big-O notation
Binary search algorithm
Call stack
Complexity analysis
Infinite recursion
Iterative process
Merge sort
Quicksort
Recursive method
Recursive step
Stack
Stack overflow error
Stopping state
Tail-recursive

In this chapter, we continue our discussion of sorting and searching and introduce two new topics: recursion and complexity analysis. These topics are intertwined because searching and sorting can involve recursion and complexity analysis. A recursive algorithm is one that refers to itself by name in a manner that appears to be circular. Everyday algorithms, such as a recipe to bake a cake or instructions to change car oil, are not expressed recursively, but recursive algorithms are common in computer science. Complexity analysis is concerned with determining an algorithm's efficiency—that is, how its runtime and memory usage vary as a function of the quantity of data processed. Consider the searching and sorting algorithms presented in Chapter 12. When we test these algorithms on arrays of 10 to 20 elements, they are blindingly fast, but how fast can we expect them to be when the arrays are 100 times larger? Because painting two houses generally takes twice as long as painting one, we might guess that the same linear relationship between size and speed applies equally to sorting algorithms. We would be quite wrong, however, and in this chapter we learn techniques for analyzing algorithm efficiency more accurately. In more advanced computer science courses, you will have the chance to study this chapter's topics in greater depth.

13.1 Recursion

When asked to add the integers from 1 to *n*, we usually think of the process iteratively. We start with 1, add 2, then 3, and so forth until we reach *n*, or expressed differently

```
sum(n) = 1 + 2 + 3 + ... + n, where n >= 1
```

Java's looping constructs make implementing the process easy. There is, however, a completely different way to look at the problem, which at first seems very strange:

```
sum(1) = 1
sum(n) = n + sum(n - 1) if n > 1
```

At first glance, expressing `sum(n)` in terms of `sum(n - 1)` seems to yield a circular definition, but closer examination shows that it does not. Consider, for example, what happens when the definition is applied to the problem of calculating `sum(4)`:

```
sum(4) = 4 + sum(3)
       = 4 + 3 + sum(2)
       = 4 + 3 + 2 + sum(1)
       = 4 + 3 + 2 + 1
```

The fact that `sum(1)` is defined to be 1 without making reference to further invocations of `sum` saves the process from going on forever and the definition from being circular. Functions that are defined in terms of themselves in this way are called *recursive*. The following, for example, are two ways to express the definition of factorial, the first iterative and the second recursive:

1. factorial(n) = 1 * 2 * 3 * ... * n, where n >= 1
2. factorial(1) = 1

   factorial(n) = n * factorial(n - 1) if n > 1

In this case, no doubt, the iterative definition is more familiar and thus easier to understand than the recursive one; however, such is not always the case. Consider the definition of Fibonacci numbers first encountered in Chapter 7. The first and second numbers in the Fibonacci sequence are 1. Thereafter, each number is the sum of its two immediate predecessors, as follows:

```
1  1  2  3  5  8  13  21  34  55  89  144  233 ...
```

or

```
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2) if n > 2
```

This is a recursive definition, and it is hard to imagine how one could express the definition nonrecursively. Turn back to Chapter 7 to see how difficult it is to write an iterative version of this method.

From these examples, we can see that recursion involves two factors. First, some function `f(n)` is expressed in terms of `f(n - 1)` and perhaps `f(n - 2)` and so on. Second, to prevent the definition from being circular, `f(1)` and perhaps `f(2)` and so on are defined explicitly.

Implementing Recursion

Given a recursive definition of some process, it is usually easy to write a ***recursive method*** that implements it. A method is said to be recursive if it calls itself. Let us start with a method that computes factorials:

```
int factorial (int n){
//Precondition n >= 1
   if (n == 1)
      return 1;
   else
      return n * factorial (n - 1);
}
```

For comparison, the following is an iterative version of the method. As you can see, it is slightly longer and no easier to understand.

```
int factorial (int n){
   int product = 1;
   for (int i = 2; i <= n; i++)
      product = product * i;
   return product;
}
```

As a second example of recursion, following is a method that calculates Fibonacci numbers:

```
int fibonacci (int n){
   if (n <= 2)
      return 1;
   else
      return fibonacci (n - 1) + fibonacci (n - 2);
}
```

Turn back to Chapter 6 to see how we wrote an iterative version of this method.

Tracing Recursive Calls

We can better understand recursion if we trace the sequence of recursive calls and returns that occur in a typical situation. Suppose we want to compute the factorial of 4. We call `factorial(4)`, which in turn calls `factorial(3)`, which in turn calls `factorial(2)`, which in turn calls `factorial(1)`, which returns 1 to `factorial(2)`, which returns 2 to `factorial(3)`, which returns 6 to `factorial(4)`, which returns 24, as shown in the following run-time trace:

```
factorial(4)
            calls factorial(3)
                              calls factorial(2)
                                                calls factorial(1)
                                                which returns 1
                              which returns 2 * 1       which is 2
            which returns 3 * 2     which is 6
which returns 4 * 6       which is 24
```

At first, it seems strange to have all these invocations of the `factorial` method, each in a state of suspended execution waiting for the completion of the ones further down the line. When the last invocation completes its work, it returns to its predecessor, which completes its work, and so forth up the line, until eventually the original invocation reactivates and finishes the job. Fortunately, we do not have to repeat this dizzying mental exercise every time we use recursion.

Guidelines for Writing Recursive Methods

Just as we must guard against writing infinite loops, so, too, we must avoid recursions that never come to an end. First, a recursive method must have a well-defined termination or ***stopping state***. For the factorial method, this was expressed in the lines

```
// Preconditon: n >= 1
if (n == 1)
   return 1;
```

Second, the ***recursive step***, in which the method calls itself, must eventually lead to the stopping state. For the factorial method, the recursive step was expressed in the lines

```
else
   return n * factorial(n - 1);
```

Because each invocation of the factorial method is passed a smaller value, eventually the stopping state must be reached. Had we accidentally written

```
else
   return n * factorial(n + 1);
```

the method would describe an ***infinite recursion***. Eventually, the user would notice and terminate the program, or else the Java interpreter would run out of memory, at which point the program would terminate with a ***stack overflow error***.

Following is a subtler example of a malformed recursive method:

```
int badMethod (int n){
   if (n == 1)
      return 1;
   else
      return n * badMethod(n - 2);
}
```

This method works fine if `n` is odd, but when `n` is even, the method passes through the stopping state and keeps on going. For instance,

```
badMethod(4)
  calls badMethod(2)
    calls badMethod(0)
      calls badMethod(-2)
        calls badMethod(-4)
          calls badMethod(-6)
            ...
```

Run-Time Support for Recursive Methods

Computers provide the following support at run time for method calls:

- A large storage area known as a ***call stack*** is created at program startup.
- When a method is called, an ***activation record*** is added to the top of the call stack.
- The activation record contains, among other things, space for the parameters passed to the method, the method's local variables, and the value returned by the method.
- When a method returns, its activation record is removed from the top of the stack.

To understand how a recursive method uses the call stack, we ignore, for the sake of simplicity, all parts of the activation record except for the parameters and the return value. The method `factorial` has one of each:

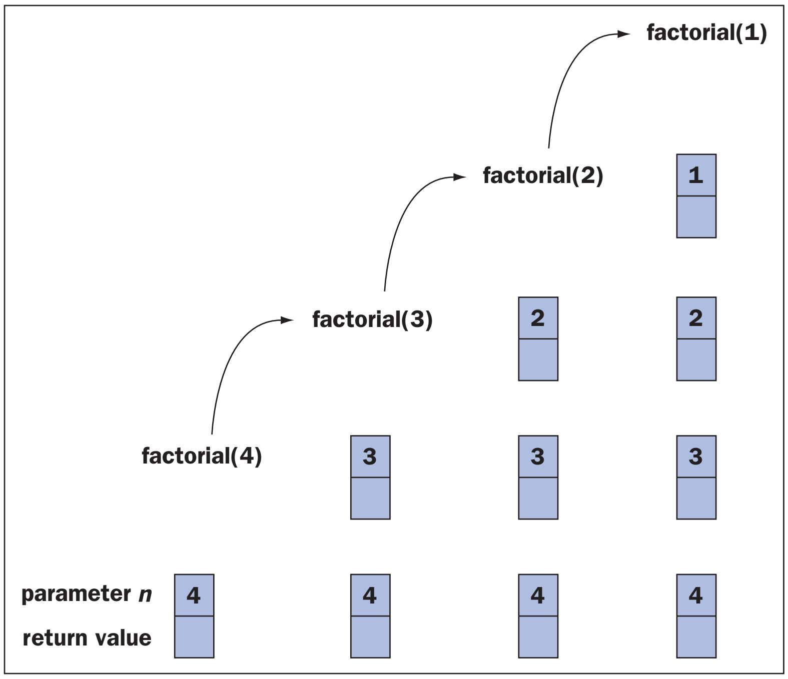
```
int factorial (int n){
   if (n <= 1)
      return 1;
   else
      return n * factorial (n - 1);
}
```

Thus, an activation record for this method requires cells for the following items:

- The value of the parameter `n`
- The return value of `factorial`

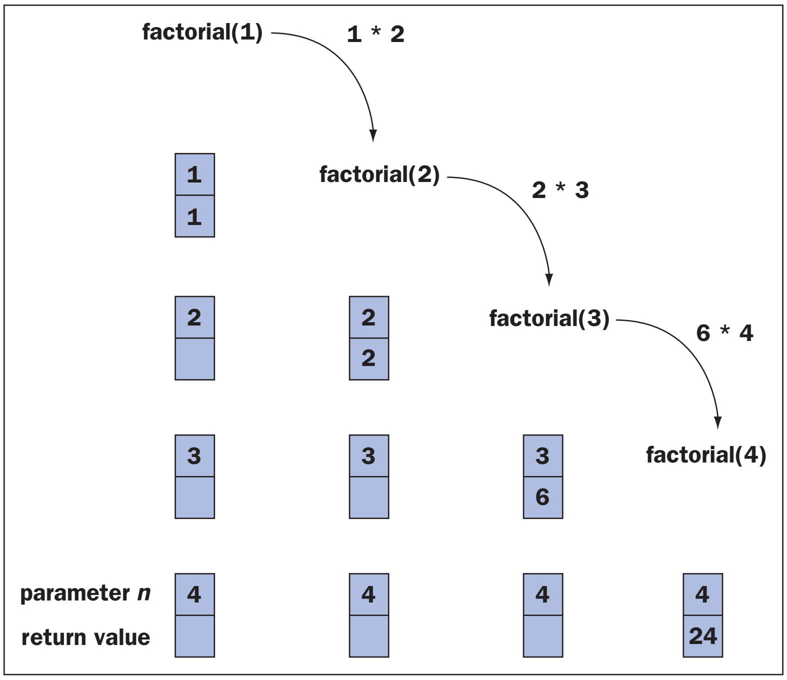
Suppose we call `factorial(4)`. A trace of the state of the call stack during calls to `factorial` down to `factorial(1)` is shown in Figure 13-1.

FIGURE 13-1
Activation records on the call stack during recursive calls to factorial



When the recursion unwinds, the return value from each call is multiplied by the parameter `n` in the record below it, and the top record is removed, as shown in the trace in Figure 13-2.

FIGURE 13-2
Activation records on the call stack during returns from recursive calls to factorial



When to Use Recursion

Recursion can always be used in place of iteration, and vice versa. Ignoring the fact that arbitrarily substituting one for the other is pointless and sometimes difficult, the question of which is better to use remains. Recursion involves a method repeatedly calling itself. Executing a method call and the corresponding `return` statement usually takes longer than incrementing and testing a loop control variable. In addition, a method call ties up some memory that is not freed until the method completes its task. Naïve programmers often state these facts as an argument against ever using recursion. However, there are many situations in which recursion provides the clearest, shortest, and most elegant solution to a programming task—as we soon see. As a beginning programmer, you should not be overly concerned about squeezing the last drop of efficiency out of a computer. Instead, you need to master useful programming techniques, and recursion ranks among the best.

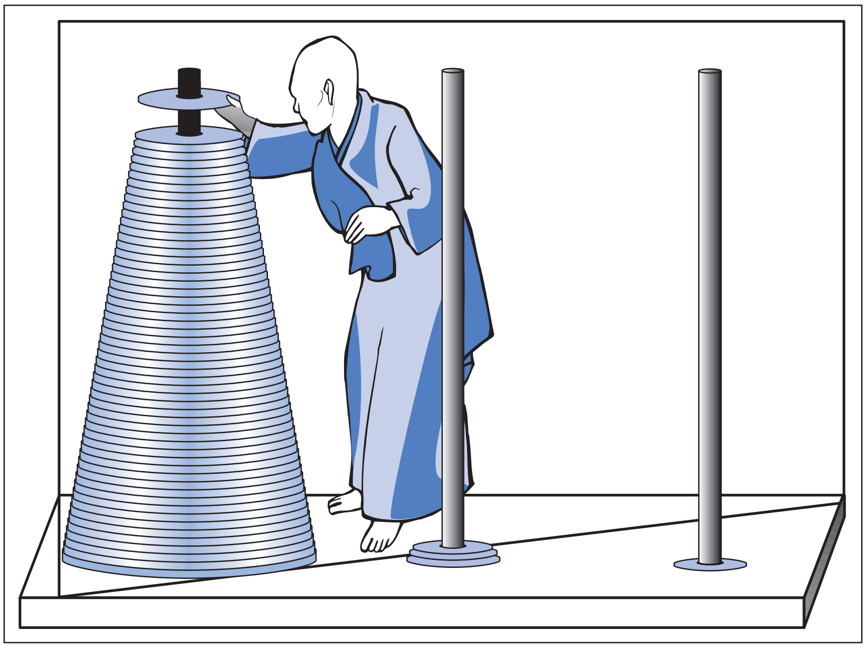
We close this section by presenting two well-known and aesthetically pleasing applications of recursion: the Towers of Hanoi and the Eight Queens problem.

Towers of Hanoi

Many centuries ago in the city of Hanoi, the monks in a certain monastery were continually engaged in what now seems a peculiar enterprise. Sixty-four rings of increasing size had been placed on a vertical wooden peg (Figure 13-3). Beside it were two other pegs, and the monks were attempting to move all the rings from the first to the third peg—subject to two constraints:

- Only one ring could be moved at a time.
- A ring could be moved to any peg, provided it was not placed on top of a smaller ring.

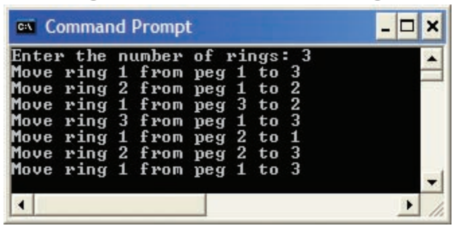
FIGURE 13-3
The Towers of Hanoi



The monks believed that the world would end and humankind would be freed from suffering when the task was finally completed. The fact that the world is still here today and you are enduring the frustrations of writing computer programs seems to indicate the monks were interrupted in their work. They were, but even if they had stuck with it, they would not finish anytime soon. A little experimentation should convince you that for n rings, $2^n - 1$ separate moves are required. At the rate of one move per second, $2^{64} - 1$ moves take about 600 billion years.

It might be more practical to harness the incredible processing power of modern computers to move virtual rings between virtual pegs. We are willing to start you on your way by presenting a recursive algorithm for printing the required moves. In the spirit of moderation, we suggest that you begin by running the program for small values of n. Figure 13-4 shows the result of running the program with three rings. In the output, the rings are numbered from smallest (1) to largest (3). You might try running the program with different numbers of rings to satisfy yourself that the printed output is correct. The number of lines of output corresponds to the formula given earlier.

FIGURE 13-4
Running the TowersOfHanoi program with three rings



```
Enter the number of rings: 3
Move ring 1 from peg 1 to 3
Move ring 2 from peg 1 to 2
Move ring 1 from peg 3 to 2
Move ring 3 from peg 1 to 3
Move ring 1 from peg 2 to 1
Move ring 2 from peg 2 to 3
Move ring 1 from peg 1 to 3
```

The program uses a recursive method called `move`. The first time this method is called, it is asked to move all *n* rings from peg 1 to peg 3. The method then proceeds by calling itself to move the top *n* – 1 rings to peg 2, prints a message to move the largest ring from peg 1 to peg 3, and finally calls itself again to move the *n* – 1 rings from peg 2 to peg 3. Following is the code:

```
/* Example 13.1: TowersOfHanoi.java
Print the moves required to move the rings in the Towers of Hanoi problem.
1) Enter the number of rings as input.
2) WARNING: Do not run this program with 64 rings.
*/

import java.util.Scanner;

public class TowersOfHanoi {

   public static void main (String [] args) {
   // Obtain the number of rings from the user.
   // Call the recursive move method to move the rings from peg 1 to peg 3
   // with peg 2 available for intermediate usage.
   //  Preconditions  -- number of rings != 64
   //  Postconditions -- the moves are printed in the terminal window

      Scanner reader = new Scanner(System.in);
      System.out.print("Enter the number of rings: ");
      int numberOfRings = reader.nextInt();
      move (numberOfRings, 1, 3, 2);
   }

   private static void move (int n, int i, int j, int k){
   // Print the moves for n rings going from peg i to peg j
   //  Preconditions  -- none
   //  Postconditions -- the moves have been printed
      if (n > 0){                   //Stopping state is n == 0

         // Move the n-1 smaller rings from peg i to peg k
         move (n - 1, i, k, j);

         // Move the largest ring from peg i to peg j
         System.out.println("Move ring " + n + " from peg " + i + " to " +
                            j);

         // Move the n-1 smaller rings from peg k to peg j
         move (n - 1, k, j, i);

         // n rings have now been moved from peg i to peg j
      }
```

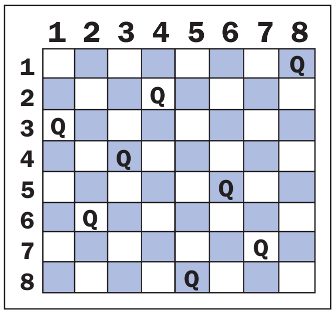
```
        }
    }
```

Although you should try this program with different numbers, you should stick with smaller numbers (i.e., <10). Large values of `n`, even with modern processing speeds, can still take a long time to run and could tie up your computer for some time. Do not attempt to run this program with 64 rings.

Eight Queens Problem

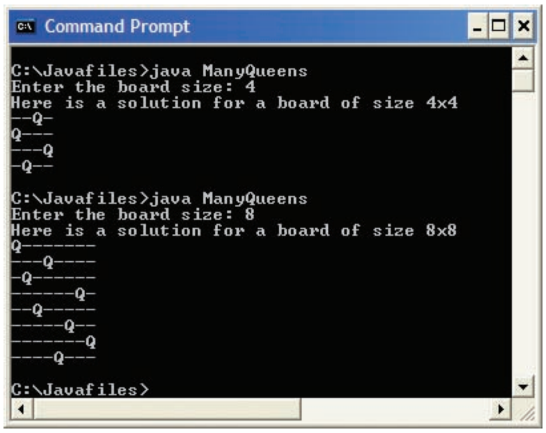
The Eight Queens problem consists of placing eight queens on a chessboard in such a manner that the queens do not threaten each other. A queen can attack any other piece in the same row, column, or diagonal, so there can be at most one queen in each row, column, and diagonal of the board. It is not obvious that there is a solution, but Figure 13-5 shows one.

FIGURE 13-5
A solution of the Eight Queens problem



We now present a program that attempts to solve this problem and others of a similar nature. We call it the `ManyQueens` program, and it attempts to place *n* queens safely on an *n* × *n* board. The program either prints a solution or a message saying that there is none (Figure 13-6).

FIGURE 13-6
Output of the ManyQueens program for boards of size 4 and 8



```
C:\Javafiles>java ManyQueens
Enter the board size: 4
Here is a solution for a board of size 4x4
--Q-
Q---
---Q
-Q--

C:\Javafiles>java ManyQueens
Enter the board size: 8
Here is a solution for a board of size 8x8
Q-------
----Q---
-Q------
------Q-
--Q-----
-----Q--
-------Q
---Q----

C:\Javafiles>
```

At the heart of the program is a recursive method called `canPlaceQueen`. Initially, the board is empty, and the first time the method is called, it places a queen at the top of column 1. It then calls itself to place a queen in the first safe square of column 2 and then again to place a queen in the first safe square of column 3 and so forth, until finally it calls itself to place a queen in the first safe square of the last column. If at some step (say, for column 5) the method fails, then it returns and processing resumes in the previous column by looking for the next safe square. If there is one, then the process moves onward to column 5 again or else back to column 3. And so it goes. Either a solution is found or all possibilities are exhausted. The program includes a second method called `attacked`. It determines if a queen placed in row *r*, column *c* is threatened by any queens already present in columns 1 to *c* – 1. Following is the code:

```
/* Example 13.2: ManyQueens.java
Determine the solution to the Many Queens problem for a chessboard
of any size.
1) There is a single input indicating the size of the board.
2) If there is a solution display it, else indicate that there is none.
*/

import java.util.Scanner;

public class ManyQueens {

   public static void main (String [] args) {
   // Process the user's input. Call a recursive function
   // to determine if there is a solution. Print the results.
   //  Preconditions  -- the input is an integer greater
   //                    than or equal to 1
   //  Postconditions -- display a solution or a message stating that there
   //                    is none

      int boardSize;        // The size of the board, for instance, 8 would
                            // indicate an 8x8 board
      boolean[][] board;    // A two-dimensional array representing the board
                            // An entry of false indicates that a square is
                            // unoccupied

      // Initialize the variables
      Scanner reader = new Scanner(System.in);
      System.out.print("Enter the board size: ");
      boardSize = reader.nextInt();
      board = new boolean[boardSize][boardSize];
      for (int row = 0; row < boardSize; row++)
         for (int col = 0; col < boardSize; col++)
            board[row][col] = false;

      // Determine if there is a solution
      if (! canPlaceQueen (0, board))

         // There is no solution
         System.out.println ("Impossible on a board of size " +
                             boardSize + "x" + boardSize);

      else{

         // There is a solution, so print it
         System.out.println ("Here is a solution for a board of size " +
                             boardSize + "x" + boardSize);
         for (int row = 0; row < boardSize; row++){
            for (int col = 0; col < boardSize; col++){
               if (board[row][col])
                  System.out.print ("Q ");
               else
                  System.out.print ("- ");
            }
            System.out.println();
         }

      }
   }

   private static boolean canPlaceQueen (int col, boolean[][] board){
   // Mark as true the first unattacked location in column col that
   // permits a solution across the remaining columns.
   //  Preconditions  -- 0 <= col < board.length
   //  Postconditions -- if an entry in col gets marked true
   //                    return true else return false

      for (int row = 0; row < board.length; row++){ // Iterate down the column
         if (! attacked (row, col, board)){         // if square is not under attack
            if (col == board.length -1){            // if this is the last column
               board[row][col] = true;              // end recursion, set square true
               return true;                         // recursive ascent true
            }else{                                  // else
               board[row][col] = true;              // trial solution, set square true
                                                    // if recursive descent succeeds
               if (canPlaceQueen (col + 1, board))
                  return true;                      // recursive ascent true
               else                                 // else
                  board[row][col] = false;          // trial solution didn't work
            }                                       // end if
         }                                          // end if
      }
      return false;                                 // recursive ascent false
   }

   private static boolean attacked (int row, int col, boolean[][] board){
  // Determine if the square at location (row, col) is under attack.
  // from any queen in columns 0 to col - 1
  //  Preconditions  -- 0 <= row, col < board.length
  //  Postconditions -- returns true if square under attack else false

  // Look for horizontal attack
      int i, j, k;
      for (j = 0; j < col; j++){
         if (board[row][j])
            return true;
      }

      // Look for attack from a descending diagonal
      i = row - 1;
      j = col - 1;
      while (i >= 0 && j >= 0)
         if (board[i][j])
            return true;
         else{
            i--;
            j--;
         }
      }
```

```
         // Look for attack from an ascending diagonal
         i = row + 1;
         j = col - 1;
         while (i < board.length && j >= 0)
            if (board[i][j])
               return true;
            else{
               i++;
               j--;
            }
         }

         return false;
      }
   }
```

EXERCISE 13.1

1. What keeps a recursive definition from being circular?
2. What are the two parts of any recursive method?
3. Why is recursion more expensive than iteration?
4. What are the benefits of using recursion?
5. Consider the following definition of the method raise, which raises a given number to a given exponent:

   ```
   int raise(int base, int expo){
      if (expo == 0)
         return 1;
      else
         return base * raise(base, expo - 1);
   }
   ```

   Draw a trace of the complete execution of `raise(2, 5)`.
6. Consider the following method:

   ```
   int whatAMethod(int n){
      if (n == 0)
         return 1;
      else
         return whatAMethod(n);
   }
   ```

   What happens during the execution of `whatAMethod(3)`?

Programming Skills

RECURSION NEED NOT BE EXPENSIVE

We have seen that the use of recursion has two costs: Extra time and extra memory are required to manage recursive function calls. These costs have led some to argue that recursion should never be used in programs. However, as Guy Steele has shown (in "Debunking the 'Expensive Procedure Call' Myth," *Proceedings of the National Conference of the ACM*, 1977), some systems can run recursive algorithms as if they were iterative ones, with no additional overhead. The key condition is to write a special kind of recursive algorithm called a ***tail-recursive*** algorithm. An algorithm is tail-recursive if no work is done in the algorithm after a recursive call. For example, according to this criterion, the factorial method that we presented earlier is not tail-recursive because a multiplication is performed after each recursive call. We can convert this version of the factorial method to a tail-recursive version by performing the multiplication before each recursive call. To do this, we need an additional parameter that passes the accumulated value of the factorial down on each recursive call. In the last call of the method, this value is returned as the result:

```
int tailRecursiveFactorial (int n, int result){
   if (n == 1)
      return result;
   else
      return tailRecursiveFactorial (n - 1, n * result);
}
```

Note that the multiplication is performed before the recursive call of the method—that is, when the parameters are evaluated. On the initial call to the method, the value of `result` should be 1:

```
int factorial (int n){
   return tailRecursiveFactorial (n, 1);
}
```

Steele showed that a smart compiler could translate tail-recursive code in a high-level language to a loop in machine language. The machine code treats the method's parameters as variables associated with a loop and generates an ***iterative process*** rather than a recursive one. Thus, there is no linear growth of method calls, and extra stack memory is not required to run tail-recursive methods on these systems.

The catch is that a programmer must be able to convert a recursive method to a tail-recursive method and find a compiler that generates iterative machine code from tail-recursive methods. Unfortunately, some methods are difficult or impossible to convert to tail-recursive versions, and the needed optimizations are not part of most standard compilers. If you find that your Java compiler supports this optimization, you should try converting some methods to tail-recursive versions and see if they run faster than the original versions.

13.2 Complexity Analysis

We need to ask an important question about every method we write: What is the effect on the method of increasing the quantity of data processed? Does doubling the amount of data double the method's execution time, triple it, quadruple it, or have no effect? This type of examination is called ***complexity analysis***. Let us consider some examples.

Sum Methods

First, consider the `sum` method presented in Chapter 10. This method processes an array whose size can be varied. To determine the method's execution time, beside each statement we place a symbol (`t1`, `t2`, etc.) that indicates the time needed to execute the statement. Because we have no way of knowing what these times really are, we can do no better.

```
int sum (int[] a){
   int result = 0;                       // Assignment: time = t1
   for (int i = 0; i < a.length; i++){ // Overhead for going once around the
                                         // loop: time = t2
      result += a[i];                    // Assignment: time = t3
   }
   return result;                        // Return: time = t4
}
```

Adding these times together and remembering that the method goes around the loop *n* times, where *n* represents the array's size, yields

```
executionTime
     = t1 + n * (t2 + t3) + t4
     = k1 + n * k2                where k1 and k2 are method-dependent constants
     ≈ n * k2                     for large values of n, where ≈ means approximately equal to
```

Thus, the execution time is linearly dependent on the array's length, and as the array's length increases, the contribution of `k1` becomes negligible. Consequently, we can say with reasonable accuracy that doubling the length of the array doubles the execution time of the method. Computer scientists express this linear relationship between the array's length and execution time using ***big-O notation***:

```
executionTime = O(n).
```

Or phrased slightly differently, the execution time is order *n*. Observe that from the perspective of big-O notation, we make no distinction between a method whose execution time is

```
1000000 + 1000000*n
```

and one whose execution time is

```
n / 1000000
```

although from a practical perspective the difference is enormous.

Complexity analysis can also be applied to recursive methods. Following is a recursive version of the sum method. It, too, is O(*n*).

```
int sum (int[] a, int i){
   if (i >= a.length)                  // Comparison: t1
      return 0;                        // Return: t2
   else
      return a[i] + sum (a, i + 1);   // Call and return: t3
}
```

The method is called initially with `i = 0`. A single activation of the method takes time

```
t1 + t2      if  i >= a.length
```

and

```
t1 + t3      if  i < a.length.
```

The first case occurs once and the second case occurs the `a.length` times that the method calls itself recursively. Thus, if `n` equals `a.length`, then

```
 executionTime
     = t1 + t2 + n * (t1 + t3)
     = k1 + n * k2                  where k1 and k2 are method-dependent constants
     = O(n)
```

Other O(*n*) Methods

Several of the array processing methods presented in Chapters 10 and 12 are O(*n*). Following is a linear search method from Chapter 12:

```
int search (int[] a, int searchValue){
   for (int i = 0; i < a.length; i++)        // Loop overhead: t1
      if (a[i] == searchValue)               // Comparison: t2
         return i;                           // Return point 1: t3
   return location;                          // Return point 2: t4
}
```

The analysis of the linear search method is slightly more complex than that of the sum method. Each time through the loop, a comparison is made. If and when a match is found, the method returns from the loop with the search value's index. If we assume that the search is usually made for values present in the array, then on average, we can expect that half the elements in the array be examined before a match is found. Putting all of this together yields

```
executionTime
     = (n / 2) * (t1 + t2) + t3
     = n * k1 + k2                  where k1 and k2 are method-dependent constants.
     = O(n)
```

Now let us look at a method that processes a two-dimensional array:

```
int[] sumRows (int[][] a){
   int[] rowSum = new int[a.length];                   // Instantiation: t1
   for (int row = 0; row < a.length; row++){           // Loop overhead: t2
      for (int col = 0; col < a[row].length; col++){   // Loop overhead: t3
         rowSum[row] += a[row][col];                   // Assignment: t4
      }
   }
   return rowSum;                                      // Return: t5
}
```

Let `n` represent the total number of elements in the array and `r` the number of rows. For the sake of simplicity, we assume that each row has the same number of elements, say, `c`. The execution time can be written as

```
executionTime
     = t1 + r * (t2 + c * (t3 + t4)) + t5
     = (k1 + (n/c) * k2) + (n/c) * t2 + n * (t3 + t4) + t5   where r = n/c
     = k1 + n * (k2/c + t2 / c + t3 + t4) + t5
     = k3 + n * k4                      where k1, k2, k3, and k4 are constants
     = O(n)
```

Notice that we have replaced `t1 by (k1 + (n/c) * k2)`. This is based on the assumption that the JVM can allocate a block of memory for the array `rowSum` in constant time `(k1)` followed by the time needed to initialize all entries to zero `((n/c) * k2)`.

An O(n^2) Method

Not all array processing methods are O(n), as an examination of the `bubbleSort` method reveals. Let us first analyze a "dumber" version of this method than the one presented in Chapter 12. This one does not track whether or not an exchange was made in the nested loop, so there is no early exit.

```
void bubbleSort(int[] a){
   int k = 0;                                      // Assignment: t1

   // Make n - 1 passes through array

   while (k < a.length() - 1){                     // Loop overhead: t2
      k++;
      for (int j = 0; j < a.length() - k; j++)     // Loop overhead: t3
         if (a[j] > a[j + 1])                      // Comparison: t4
            swap(a, j, j + 1);                     // Assignments: t5
  }
}
```

The outer loop of the sort method executes $n - 1$ times, where n is the length of the array. Each time the inner loop is activated, it iterates a different number of times. On the first activation it iterates $n - 1$ times, on the second $n - 2$, and so on, until on the last activation it iterates once. Thus, the average number of iterations is $n / 2$. On some iterations, elements `a[i]` and `a[j]` are interchanged in time `t4`, and on other iterations they are not. So on the average iteration, let's say time `t6` is spent doing an interchange. The execution time of the method can now be expressed as

```
executionTime
     = t1 + (n - 1) * (t2 + (n / 2) * (t3 + t4 + t6))
     = t1 + n * t2 - t2 + (n * n / 2) * (t3 + t4 + t6) -
                          (n / 2) * (t3 + t4 + t6)
     = k1 + n * k2 + n * n * k3
     ≈ n * n * k3
```

= O(n^2) for large values of n

As discussed in Chapter 12, we can alter this method to track whether or not an exchange was made within the nested loop. If no exchange was made, then the array must be sorted and we can exit the method early. However, because we usually make an exchange on each pass on the average, this trick does not improve the bubble sort's complexity on the average. Nevertheless, it can improve the method's behavior to linear in the best case (a case in which the array is already sorted).

Common Big-O Values

We have already seen several methods that are O(n) and one that is O(n^2). These are just two of the most frequently encountered big-O values. Table 13-1 lists some other common big-O values together with their names.

TABLE 13-1
Names of some common big-O values

BIG-O VALUE	NAME
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	n log n
O(n^2)	Quadratic
O(n^3)	Cubic
O(2^n)	**Exponential**

As an example of O(1), consider a method that returns the sum of the first and last numbers in an array. This method's execution time is independent of the array's length. In other words, it takes constant time. Later in the chapter, we will see examples of methods that are logarithmic and n log n.

The values in Table 13-1 are listed from "best" to "worst." For example, given two methods that perform the same task, but in different ways, we tend to prefer the one that is O(*n*) over the one that is O(n^2). This statement requires some elaboration. For instance, suppose that the exact run time of two methods is

```
10,000 + 400n        // method 1
```

and

```
10,000 + n²          // method 2
```

For small values of *n*, method 2 is faster than method 1; however, and this is the important point, for all values of *n* larger than a certain threshold, method 1 is faster. The threshold in this example is 400. So if you know ahead of time that *n* will always be less than 400, you are advised to use method 2, but if *n* will have a large range of values, method 1 is superior.

By the way, from the perspective of complexity analysis, we do not need to distinguish between base 2 and base 10 logarithms because they differ only by a constant factor:

$$\log_2 n = \log_{10} n * \log_2 10$$

To get a feeling for how the common big-O values vary with *n*, consider Table 13-2. We use base 10 logarithms. This table vividly demonstrates that a method might be useful for small values of *n* but totally worthless for large values. Clearly, methods that take exponential time have limited value, even if it were possible to run them on the world's most powerful computer for billions of years. Unfortunately, there are many important problems for which even the best algorithms take exponential time. You can achieve lasting fame by being the first person to replace one of these exponential time algorithms with one that takes less than exponential time.

TABLE 13-2
How big-O values vary depending on *n*

n	1	LOG *n*	*n*	*n* LOG *n*	n^2	n^3	2^n
10	1	1	10	10	100	1,000	1,024
100	1	2	100	200	10,000	1,000,000	≈ 1.3 e30
1,000	**1**	**3**	**1,000**	**3,000**	**1,000,000**	**1,000,000,000**	**≈ 1.1 e301**

An O(r^n) Method

We have seen two algorithms for computing Fibonacci numbers, one iterative and the other recursive. The iterative algorithm presented in Chapter 7 is O(*n*). However, the much simpler recursive algorithm presented earlier in this chapter is O(r^n), where $r \approx 1.62$. While O(r^n) is better than O(2^n), it is still exponential. It is beyond the book's scope to prove that the recursive algorithm is O(r^n). Nonetheless, it is easy to demonstrate that the number of recursive calls increases rapidly with *n*. For instance, Figure 13-7 shows the calls involved when we use the recursive method to compute the sixth Fibonacci number. To keep the diagram reasonably compact, we write `(6)` instead of `fibonacci(6)`.

FIGURE 13-7
Calls needed to compute the sixth Fibonacci number recursively

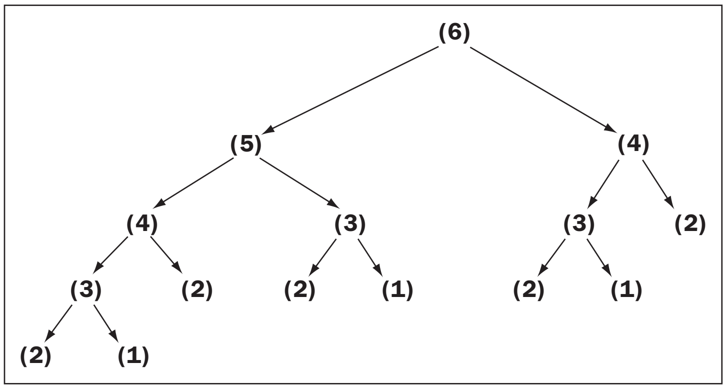


Table 13-3 shows the number of calls as a function of *n*.

TABLE 13-3
Calls needed to compute the *n*th Fibonacci number recursively

n	CALLS NEEDED TO COMPUTE *n*TH FIBONACCI NUMBER
2	1
4	5
8	41
16	1,973
32	**4,356,617**

The values in Table 13-3 were obtained by running the following program:

```
// Example 13.3: Test fibonacci method

public class Tester{

   private static int count = 0;

   public static void main (String[] args){
      for (int i = 1; i <= 5; i++){
         count = 0;
         int n = (int)Math.pow(2, i);
         int fibn = fibonacci(n);
         System.out.println ("" + n + ":" + count);
      }
   }

   private static int fibonacci (int n){
      count++;
      if (n <= 2)
         return 1;
      else
```

```
            return fibonacci(n - 1) + fibonacci(n - 2);
        }
    }
```

Programs such as this are frequently useful for gaining an empirical sense of an algorithm's efficiency.

Best-Case, Worst-Case, and Average-Case Behavior

Many algorithms do not have a single measure of complexity that applies to all cases. Sometimes an algorithm's behavior improves or gets worse when it encounters a particular arrangement of data. For example, the bubble sort algorithm developed in Chapter 12 can terminate as soon as the array becomes sorted. If the input array is already sorted, the bubble sort requires just $N - 1$ comparisons. In many other cases, however, bubble sort requires $O(N_2)$ comparisons. Clearly, a more detailed analysis may be needed to make programmers aware of these special cases.

A thorough analysis of an algorithm's complexity divides its behavior into three types of cases:

1. *Best case*. Under what circumstances does an algorithm do the least amount of work? What is the algorithm's complexity in this best case?
2. *Worst case*. Under what circumstances does an algorithm do the most amount of work? What is the algorithm's complexity in this worst case?
3. *Average case*. Under what circumstances does an algorithm do a typical amount of work? What is the algorithm's complexity in this typical case?

Let's consider three examples of this kind of analysis for summation, linear search, and bubble sort.

Because the summation algorithm must visit each number in the array, no matter how the numbers are ordered, the algorithm is always linear. Therefore, its best-case, worst-case, and average-case behaviors are $O(n)$.

Linear search is a bit different. We consider only the case in which the target element is in the array. The algorithm stops and returns a result as soon as it finds the target element. Clearly, in the best case, this element is at the first position. In the worst case, the target is in the last position. Therefore, the algorithm's best-case behavior is $O(1)$ and its worst-case behavior is $O(n)$. To compute the average-case behavior, we add up all of the comparisons that must be made to locate a target in each position and divide by n. This is $(1 + 2 + \ldots + n) / n$, or $n / 2$. Therefore, by approximation, the average-case behavior of linear search is also $O(n)$.

As we saw in Chapter 12, the "smarter" version of bubble sort can terminate as soon as the array becomes sorted. In the best case, this happens when the input array is already sorted. Therefore, bubble sort's best-case behavior is $O(n)$. However, this case is very rare (1 out of $n!$). In the worst case, even this version of bubble sort will have to bubble each element down to its proper position in the array. The algorithm's worst-case behavior is clearly $O(n^2)$. Bubble sort's average-case behavior is also $O(n^2)$, although a rigorous demonstration of this fact is a bit more involved than it is for linear search.

As we will see, there are algorithms whose best-case and average-case behaviors are similar, but whose behavior degrades considerably in the worst case. Whether you are choosing an algorithm or developing a new one, it is important to be aware of these distinctions.

EXERCISE 13.2

1. Using big-O notation, state the time complexity of the following recursive methods:
   a. `factorial`
   b. `raise` (see Exercise 13.1, Question 5)

2. Recursive methods use stack space. Using big-O notation, state the space complexity of the following recursive methods:
   a. `factorial`
   b. `fibonacci`

3. State the time complexity of the following sort method:

```
void sort(int[] a){
   for (int j = 0; j < a.length - 1; j++){
      int minIndex = j;
      for (int k = j + 1; k < a.length; k++)
         if (a[k] < a[minIndex])
            minIndex = k;
      if (minIndex != j){
         int temp = a[j];
         a[j] = a[minIndex];
         a[minIndex] = temp;
      }
   }
}
```

13.3 Binary Search

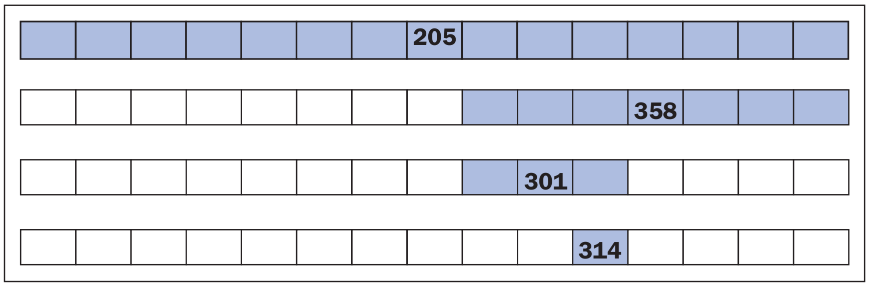
Searching is such a common activity that it is important to do it quickly. As mentioned earlier, a linear search starts at the beginning of an array and looks at consecutive elements until either the search value is located or the array's end is encountered. Imagine using this technique to find a number by hand in a list of 10 million entries. It would take an intolerably long time, especially if the elements are strings (recall that all characters in two strings must sometimes be compared to determine equality).

Alternatively, as mentioned in Chapter 12, if we know in advance that the list is in ascending order, we can quickly zero in on the search value or determine that it is absent using the ***binary search algorithm***. We shall show that this algorithm is O(log *n*).

We start by looking at the middle of the list. We might be lucky and find the search value immediately. If not, we know whether to continue the search in the first or the second half of the list. Now we reapply the technique repeatedly. At each step, we reduce the search region by a factor of 2. Soon we either must find the search value or narrow the search down to a single element. A list of 1 million entries involves at most 20 steps.

Figure 13-8 is an illustration of the binary search algorithm. We are looking for the number 320. At each step, we highlight the sublist that might still contain 320. Also at each step, all the numbers are invisible except the one in the middle of the sublist, which is the one that we are comparing to 320.

FIGURE 13-8
Binary search algorithm



After only four steps, we have determined that 320 is not in the list. Had the search value been 205, 358, 301, or 314, we would have located it in four or fewer steps. The binary search algorithm is guaranteed to search a list of 15 sorted elements in a maximum of four steps. Incidentally, the list with all the numbers visible looks like Figure 13-9.

FIGURE 13-9
The list for the binary search algorithm with all numbers visible

15	36	87	95	100	110	194	205	297	301	314	358	451	467	486

Table 13-4 shows the relationship between a list's length and the maximum number of steps needed to search the list. To obtain the numbers in the second column, add 1 to the larger numbers in the first column and take the logarithm base 2. Hence, a method that implements a binary search is O(log *n*).

TABLE 13-4
Maximum number of steps needed to binary search lists of various sizes

LENGTH OF LIST	MAXIMUM NUMBER OF STEPS NEEDED
1	1
2 to 3	2
4 to 7	3
8 to 15	4
16 to 31	5
32 to 63	6
64 to 127	7
128 to 255	8
256 to 511	9
512 to 1023	10

TABLE 13-4 Continued
Maximum number of steps needed to binary search lists of various sizes

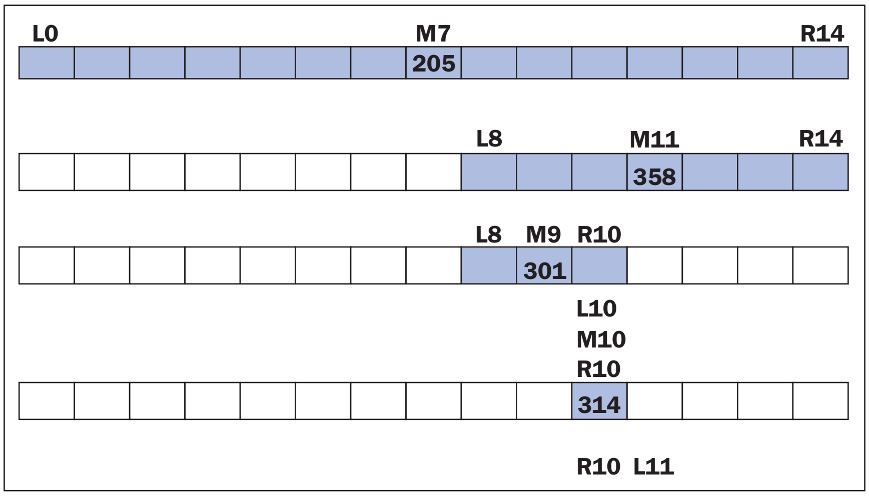
LENGTH OF LIST	MAXIMUM NUMBER OF STEPS NEEDED
1024 to 2047	11
2^n to $2^{n+1} - 1$	**n + 1**

We now present two versions of the binary search algorithm, one iterative and one recursive, and both O(log *n*). We will forgo a formal analysis of the complexity. First, the iterative version, as introduced in Chapter 12:

```
// Iterative binary search of an ascending array
int search (int[] a, int target){
   int left = 0;                             // Establish the initial
   int right = a.length - 1;                 // endpoints of the array
   while (left <= right){                    // Loop until the endpoints cross
      int midpoint = (left + right) / 2;     // Compute the current midpoint
      if (a[midpoint] == target)             // Target found; return its index
         return midpoint;
      else if (a[midpoint] < target)         // Target to right of midpoint
         left = midpoint + 1;
      else                                   // Target to left of midpoint
         right = midpoint - 1;
   }
   return -1;                                // Target not found
}
```

Figure 13-10 illustrates an iterative search for 320 in the list of 15 elements. L, M, and R are abbreviations for `left`, `midpoint`, and `right`. At each step, the figure shows how these variables change. Because 320 is absent from the list, eventually `(left > right)` and the method returns –1.

FIGURE 13-10
Steps in an iterative binary search for the number 320



Now for the recursive version of the algorithm:

```
// Recursive binary search of an ascending array
int search (int[] a, int target, int left, int right){
   if (left > right)
      return -1;
   else{
      int midpoint = (left + right) / 2;
      if (a[midpoint] == target)
         return midpoint;
      else if (a[midpoint] < target)
         return search (a, target, midpoint + 1, right);
      else
         return search (a, target, left, midpoint - 1);
   }
}
```

At heart, the two versions are similar, and they use the variables `left`, `midpoint`, and `right` in the same way. Of course, they differ in that one uses a loop and the other uses recursion. We conclude the discussion by showing how the two methods are called:

```
int[] a = {15,36,87,95,100,110,194,205,297,301,314,358,451,467,486};
int x = 320;
int location;

location = search (a, x);                    // Iterative version
location = search (a, x, 0, a.length - 1);   // Recursive version
```

EXERCISE 13.3

1. The efficiency of the method `raise`, in Question 5 of Exercise 13.1, can be improved by the following changes: If the exponent is even, then raise the base to the exponent divided by 2 and return the square of this number. Otherwise, the exponent is odd, so `raise` the number as before. Rewrite the method `raise` using this strategy.

2. Draw a trace of the complete execution of `raise(2, 6)` as defined in Question 1 in this exercise.

3. What is the time complexity of `raise` in Question 1?

13.4 Quicksort

The sort algorithms presented in Chapter 12 are O(n^2). There are a number of variations on the algorithms, some of which are marginally faster, but they, too, are O(n^2). In contrast, there are also several better algorithms that are O($n \log n$). ***Quicksort*** is one of the simplest of these. The general idea behind quicksort is this: Break an array into two parts and then move elements around so that all the larger values are in one end and all the smaller values are in the other. Each of the two parts is then subdivided in the same manner, and so on, until the subparts contain only a single value, at which point the array is sorted. To illustrate the process, suppose an unsorted array, called `a`, looks like Figure 13-11.

FIGURE 13-11
Unsorted array

5	12	3	11	2	7	20	10	8	4	9

Phase 1

1. If the length of the array is less than 2, then it is done.
2. Locate the value in the middle of the array and call it the *pivot*. The pivot is 7 in this example (Figure 13-12).

FIGURE 13-12
Step 2 of quicksort

5	12	3	11	2	$\underline{7}$	20	10	8	4	9

3. Tag the elements at the left and right ends of the array as `i` and `j`, respectively (Figure 13-13).

FIGURE 13-13
Step 3 of quicksort

5	12	3	11	2	$\underline{7}$	20	10	8	4	9
i										j

4. While `a[i] <` pivot value, increment `i`.
   While `a[j] >` pivot value, decrement `j` (Figure 13-14).

FIGURE 13-14
Step 4 of quicksort

5	12	3	11	2	$\underline{7}$	20	10	8	4	9
	i								j	

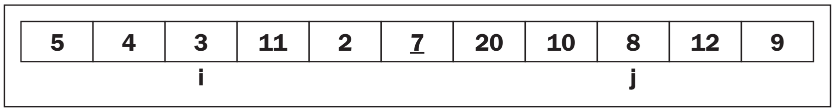
5. If `i > j` then
     end the phase
   else
     interchange `a[i]` and `a[j]` (Figure 13-15).

FIGURE 13-15
Step 5 of quicksort

5	4	3	11	2	$\underline{7}$	20	10	8	12	9
	i								j	

6. Increment `i` and decrement `j`.
   If `i > j` then end the phase (Figure 13-16).

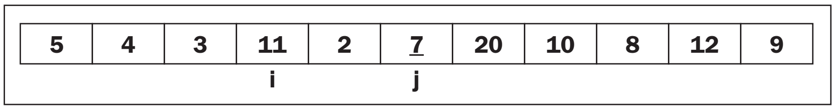
FIGURE 13-16
Step 6 of quicksort



7. Repeat Step 4, that is,

```
While a[i] < pivot value, increment i
While a[j] > pivot value,  decrement j (Figure 13-17).
```

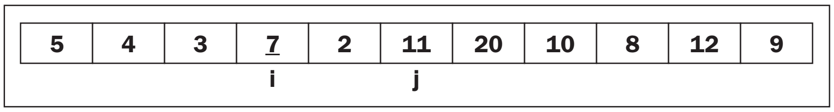
FIGURE 13-17
Step 7 of quicksort



8. Repeat Step 5, that is,

```
If i > j then
   end the phase
else
   interchange a[i] and a[j] (Figure 13-18).
```

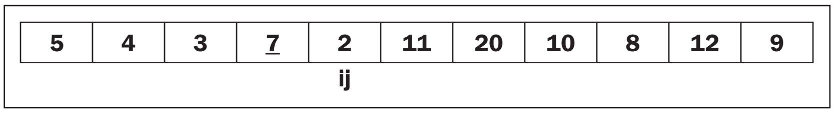
FIGURE 13-18
Step 8 of quicksort



9. Repeat Step 6, that is,

```
Increment i and decrement j.
If i > j then end the phase (Figure 13-19).
```

FIGURE 13-19
Step 9 of quicksort



10. Repeat Step 4, that is,

```
While a[i] <  pivot value, increment i
While a[j] > pivot value, decrement j (Figure 13-20).
```

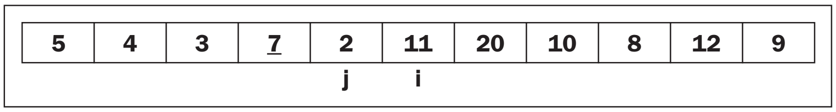


FIGURE 13-20
Step 10 of quicksort

11. Repeat Step 5, that is,

```
If i > j then
   end the phase
else
   interchange a[i] and a[j].
```

12. This ends the phase. Split the array into the two subarrays `a[0..j]` and `a[i..10]`. For clarity, the left subarray is shaded (Figure 13-21). Notice that all the elements in the left subarray are less than or equal to the pivot, and those in the right are greater than or equal to the pivot.

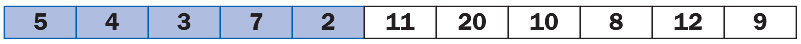


FIGURE 13-21
Step 12 of quicksort

Phase 2 and Onward

Reapply the process to the left and right subarrays, and then divide each subarray in two, and so on, until the subarrays have lengths of at most 1.

Complexity Analysis

We now present an informal analysis of the quicksort's complexity. During phase 1, `i` and `j` moved toward each other. At each move, either an array element is compared to the pivot or an interchange takes place. As soon as `i` and `j` pass each other, the process stops. Thus, the amount of work during phase 1 is proportional to n, the array's length.

The amount of work in phase 2 is proportional to the left subarray's length plus the right subarray's length, which together yield n. And when these subarrays are divided, there are four pieces whose combined length is n, so the combined work is proportional to n yet again. At successive phases, the array is divided into more pieces, but the total work remains proportional to n.

To complete the analysis, we need to determine how many times the arrays are subdivided. We make the optimistic assumption that each time the dividing line turns out to be as close to the center as possible. In practice, this is not usually the case. We already know from our discussion of the binary search algorithm that when we divide an array in half repeatedly, we arrive at a single element in about $\log_2 n$ steps. Thus, the algorithm is $O(n \log n)$ in the best case. In the worst case, the algorithm is $O(n^2)$.

Implementation

The quicksort algorithm can be coded using either an iterative or a recursive approach. The iterative approach also requires a data structure called a ***stack***. Because we have described it recursively, we might as well implement it that way too.

```
void quickSort (int[] a, int left, int right){

   if (left >= right) return;

   int i = left;
   int j = right;
   int pivotValue = a[(left + right) / 2];
   while (i < j){
      while (a[i] < pivotValue) i++;
      while (pivotValue < a[j]) j--;
      if (i <= j){
         int temp = a[i];
         a[i] = a[j];
         a[j] = temp;
         i++;
         j--;
      }
   }
   quickSort (a, left, j);
   quickSort (a, i, right);
}
```

EXERCISE 13.4

1. Describe the strategy of quicksort and explain why it can reduce the time complexity of sorting from $O(n^2)$ to $O(n \log n)$.

2. Why is quicksort not $O(n \log n)$ in all cases? Describe the worst-case situation for quicksort.

3. Describe three strategies for selecting a pivot value in quicksort.

4. Jack has a bright idea: When the length of a subarray in quicksort is less than a certain number, say, 50 elements, run an insertion sort to process that subarray. Explain why this is a bright idea.

13.5 Merge Sort

Another algorithm, called ***merge sort***, employs a recursive, divide-and-conquer strategy to break the $O(n^2)$ barrier. Here is an outline of the algorithm:

- Compute the middle position of an array and recursively sort its left and right subarrays (divide and conquer).
- Merge the two sorted subarrays back into a single sorted array.

- Stop the process when subarrays can no longer be subdivided.

This top-level design strategy can be implemented as three Java methods:

- `mergeSort`—the public method called by clients
- `mergeSortHelper`—a private helper method that hides the extra parameter required by recursive calls
- `merge`—a private method that implements the merging process

The merging process uses an extra array, which we call `copyBuffer`. To avoid the overhead of allocating and deallocating the `copyBuffer` each time `merge` is called, the buffer is allocated once in `mergeSort` and subsequently passed to `mergeSortHelper` and `merge`. Each time `mergeSortHelper` is called, it needs to know the bounds of the subarray with which it is working. These bounds are provided by two parameters, low and high. Here is the code for `mergeSort`:

```
void mergeSort(int[] a){
   // a            array being sorted
   // copyBuffer   temp space needed during merge

   int[] copyBuffer = new int[a.length];
   mergeSortHelper(a, copyBuffer, 0, a.length - 1);
}
```

After verifying that it has been passed a subarray of at least two items, `mergeSortHelper` computes the midpoint of the subarray, recursively sorts the portions below and above the midpoint, and calls `merge` to merge the results. Here is the code for `mergeSortHelper`:

```
void mergeSortHelper(int[] a, int[] copyBuffer,
                      int low, int high){
   // a            array being sorted
   // copyBuffer   temp space needed during merge
   // low, high    bounds of subarray
   // middle       midpoint of subarray

   if (low < high){
      int middle = (low + high) / 2;
      mergeSortHelper(a, copyBuffer, low, middle);
      mergeSortHelper(a, copyBuffer, middle + 1, high);
      merge(a, copyBuffer, low, middle, high);
   }
}
```

Figure 13-22 shows the subarrays generated during recursive calls to `mergeSortHelper`, starting from an array of eight items. Note that in this example the subarrays are evenly subdivided at each stage and there are 2^{k-1} subarrays to be merged at stage k. Had the length of the initial array not been a power of two, then an exactly even subdivision would not have been achieved at each stage and the last stage would not have contained a full complement of subarrays.

FIGURE 13-22
Subarrays generated during calls of `mergeSort`

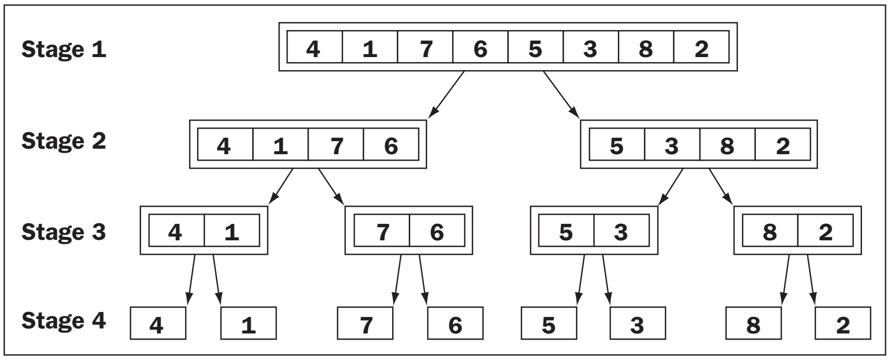
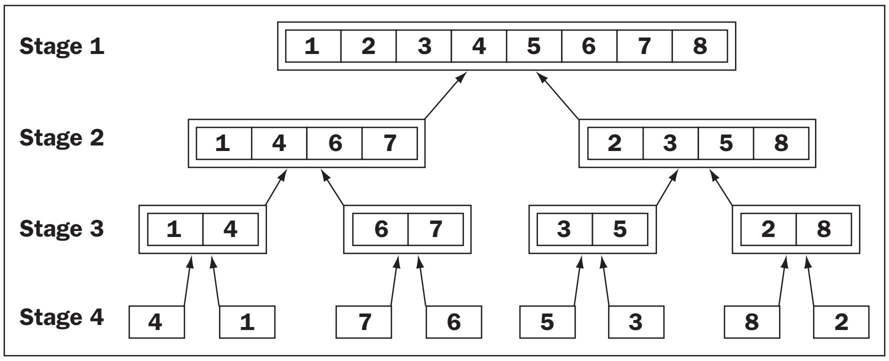


Figure 13-23 traces the process of merging the subarrays generated in the previous figure.

FIGURE 13-23
Merging the subarrays generated during a merge sort



Finally, here is the code for the `merge` method:

```
void merge(int[] a, int[] copyBuffer,
               int low, int middle, int high){
   // a            array that is being sorted
   // copyBuffer   temp space needed during the merge process
   // low          beginning of first sorted subarray
   // middle       end of first sorted subarray
   // middle + 1   beginning of second sorted subarray
   // high         end of second sorted subarray

   // Initialize i1 and i2 to the first items in each subarray
   int i1 = low, i2 = middle + 1;

13    // Interleave items from the subarrays into the copyBuffer in such a
14    // way that order is maintained.
15    for (int i = low; i <= high; i++){
16       if (i1 > middle)
17          copyBuffer[i] = a[i2++];     // First subarray exhausted
18       else if (i2 > high)
19          copyBuffer[i] = a[i1++];     // Second subarray exhausted
20       else if (a[i1] < a[i2])
21          copyBuffer[i] = a[i1++];     // Item in first subarray is less
22       else
23          copyBuffer[i] = a[i2++];     // Item in second subarray is less
24    }
25
26    for (int i = low; i <= high; i++) // Copy sorted items back into
27       a[i] = copyBuffer[i];           // proper position in a
28 }
```

The `merge` method combines two sorted subarrays into a larger sorted subarray. The first subarray lies between `low` and `middle` and the second between `middle + 1` and `high`. The process consists of three steps:

1. Set up index pointers to the first items in each subarray (line 11). These are at positions `low` and `middle + 1`.
2. Starting with the first item in each subarray, repeatedly compare items. Copy the smaller item from its subarray to the copy buffer and advance to the next item in the subarray. Repeat until all items have been copied from both subarrays. If the end of one subarray is reached before the other's, finish by copying the remaining items from the other subarray (lines 15–24).
3. Copy the portion of `copyBuffer` between `low` and `high` back to the corresponding positions in the array a (lines 26–27).

Complexity Analysis for Merge Sort

The run time of the `merge` method is dominated by the two `for` statements, each of which loop (`high - low + 1`) times. Consequently, the method's run time is O (`high - low`), and all the merges at a single stage take O(n) time. Because merge sort splits subarrays as evenly as possible at each stage, the number of stages is O(log n), and the maximum run time for merge sort is O(n log n) in all cases.

Merge sort has two space requirements that depend on the array's size. First, O(log n)space is required on the call stack to support recursive calls. Second, O(n) space is used by the copy buffer.

Improving Merge Sort

Merge sort can be improved in three ways. First, the `merge` method can be modified so that the first `for` statement makes a single comparison on each iteration. Second, there exists a complex process that allows one to merge two subarrays without using a copy buffer and without changing the order of the method. Third, subarrays below a certain size can be sorted using an alternative approach.

Case Study: Comparing Sort Algorithms

For the benefit of those who are unconvinced by mathematical analysis, we now develop a program that compares the speed of our two sort algorithms.

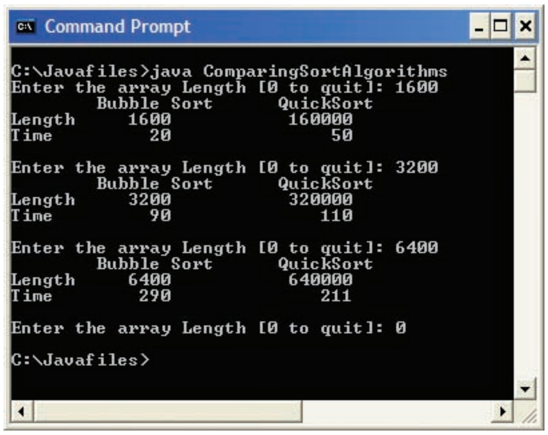
Request

Write a program that allows the user to compare sort algorithms.

Analysis

The program compares bubble sort and quicksort. Because quicksort runs much more quickly than bubble sort, we do not run both sorts on the same array. Instead, we run quicksort on an array that is 100 times longer than the array used with bubble sort. Also, because we have already compared these algorithms by counting their operations, we record run times and compare these instead. The proposed interface is shown in Figure 13-24.

FIGURE 13-24
How run time varies with array length for bubble sort on the left and quicksort on the right



```
C:\Javafiles>java ComparingSortAlgorithms
Enter the array Length [0 to quit]: 1600
           Bubble Sort       QuickSort
Length      1600            160000
Time          20                50

Enter the array Length [0 to quit]: 3200
           Bubble Sort       QuickSort
Length      3200            320000
Time          90               110

Enter the array Length [0 to quit]: 6400
           Bubble Sort       QuickSort
Length      6400            640000
Time         290               211

Enter the array Length [0 to quit]: 0

C:\Javafiles>
```

The user enters the size of the array of integers. The program then performs these actions:

1. Loads two arrays with randomly generated integers ranging from 0 to 100,000. One array is of the size specified by the user, and the other array is 100 times that size.
2. Runs the bubble sort algorithm on the smaller array and the quicksort algorithm on the larger array, and records the running times of each sort in milliseconds.
3. Displays the array sizes and run times for each sort in labeled columns.

Design

The bubble sort and quicksort algorithms have already been presented in this text. There are two other primary tasks to consider:

1. Load an array with randomly generated numbers.
2. Obtain the run time of a sort.

We accomplish the first task by using an instance of the Random class to generate the integer assigned to each cell in an array.

We accomplish the second task by using Java's Date class, as defined in the package java.util. When the program creates a new instance of Date, this object contains the current date down to the nearest millisecond, on the computer's clock. Thus, one can record the date at the beginning and end of any process by creating two Date objects, as follows:

```
Date d1 = new Date();   // Record the date at the start of a process.
<run any process>
Date d2 = new Date();   // Record the date at the end of a process.
```

To obtain the elapsed time between these two dates, one can use the Date instance method getTime(). This method returns the number of milliseconds from January 1, 1970, 00:00:00 GMT until the given date. Thus, the elapsed time in milliseconds for the example process can be computed as follows:

```
long elapsedTime = d2.getTime() - d1.getTime();
```

Implementation

To obtain the neatly formatted output shown in Figure 13-24, it is convenient to use the printf method as explained in Chapter 8. Following is the code:

```
// 13.1 Case Study: Compare two sort algorithms

import java.util.*;

public class ComparingSortAlgorithms {

   public static void main(String[] args){
      Random gen = new Random();
      Scanner reader = new Scanner(System.in);
      while(true){
         System.out.print("Enter the array Length [0 to quit]: ");
         int arrayLength = reader.nextInt();
         if (arrayLength <= 0)
            break;
         // Instantiate two arrays,
         // one of this length and the other a 100 times longer.
         int[] a1 = new int[arrayLength];
         int[] a2 = new int[arrayLength * 100];

         // Initialize the first array
         for (int i = 0; i < a1.length; i++)
            a1[i] = gen.nextInt(100001);
         // Random numbers between 0 and 100,000

         // Initialize the second array
         for (int i = 0; i < a2.length; i++)
            a2[i] = gen.nextInt(100001);

         // Time bubble sort
         Date d1 = new Date();
         bubbleSort (a1);
         Date d2 = new Date();
         long elapsedTime1 = d2.getTime() - d1.getTime();

         // Time quicksort
         d1 = new Date();
         quickSort (a2, 0, a2.length - 1);
         d2 = new Date();
         long elapsedTime2 = (d2.getTime() - d1.getTime());

         // Display results in pretty format
         System.out.printf("       %12s %14s%n", "Bubble Sort",
                          "QuickSort");
         System.out.printf("Length %8d %16d%n",   arrayLength,  arrayLength
                           * 100);
         System.out.printf("Time   %8d %16d%n%n", elapsedTime1,
  elapsedTime2);
      }
   }

   private static void bubbleSort (int[] a){
      for (int i = 0; i < a.length - 1; i++){
         for (int j = i + 1; j < a.length; j++){
            if (a[i] > a[j]){
               int temp = a[i];
               a[i] = a[j];
               a[j] = temp;
            }
         }
      }
   }

   private static void quickSort (int[] a, int left, int right){
      if (left >= right) return;
      int i = left;
      int j = right;
      int pivotValue = a[(left + right)/2];  // Pivot is at midpoint
      while (i < j){
         while (a[i] < pivotValue) i++;
         while (pivotValue < a[j]) j--;
         if (i <= j){
            int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            i++;
            j--;
         }
      }
      quickSort (a, left, j);
      quickSort (a, i, right);
   }
}
```

13.6 Graphics and GUIs: Drawing Recursive Patterns

Recursive patterns have played an important role in modern mathematics, art, and the study of nature. In this section, we show how to generate and visualize some recursive patterns that are present in fractals and abstract art. But first we introduce another GUI control, the slider, which is used in our examples.

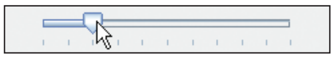
Extra Challenge

This Graphics and GUIs section gives you the opportunity to explore concepts and programming techniques required to develop modern graphics applications and graphical user interfaces. This material is not required in order to proceed with the other chapters of the book.

Sliders

A slider is a GUI control that allows the user to select a value within a range of values. The slider tool appears as a knob that can be dragged from one value to another along a bar. The bar of values can be oriented vertically or horizontally. The slider is usually labeled with ticks that indicate the positions of discrete values (see Figure 13-25). Alternatively, the user can move the knob in either direction along the bar by pressing the keyboard's left or right cursor key.

FIGURE 13-25
A slider control



The following code segment creates a slider with a horizontal alignment, a range of values from 0 to 500, and an initially selected value of 250. It then sets the major tick spacing to 50 and makes the ticks visible.

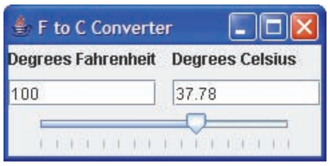
```
JSlider slider = new JSlider(SwingConstants.HORIZONTAL, 0, 500, 250);
slider.setMajorTickSpacing(50);
slider.setPaintTicks(true);
```

A slider also responds to the messages `getValue` and `setValue`, which examine and reset its current value, respectively.

When a user moves a slider's knob, the slider emits an event of type `ChangeEvent`. This event can be detected by a listener object that implements the method `stateChanged`. This method is specified in the `ChangeListener` interface. These resources are included in the package `javax.swing.event`. (Until now, all other events and listener resources have come from the package `java.awt.event`.) As usual, the programmer defines a listener class, instantiates it, and attaches this instance to the slider with the method `addChangeListener`.

Armed with this knowledge of sliders, let's modify the temperature conversion program of Section 7.6. In that version of the program, the user enters a Fahrenheit or Celsius value in the appropriate text field and clicks the appropriate button to convert from one type of measure to the other. In the new version, the user manipulates a slider along the Fahrenheit scale and simply observes the appropriate outputs in the two fields. The new user interface is shown in Figure 13-26.

FIGURE 13-26
User interface for the temperature conversion program



Following is the code for the revised view class, with explanatory comments. Note that Fahrenheit values are now integers and Celsius values are displayed with two digits of precision.

```
// Example 13.4: Revised temperature conversion program
// that uses a slider to change degrees Fahrenheit and degrees Celsius

import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;

public class GUIWindow extends JFrame{

   // >>>>>>> The model <<<<<<<<

   // Declare and intantiate the thermometer
   private Thermometer thermo = new Thermometer();

   // >>>>>>> The view <<<<<<<<

   // Declare and instantiate the window objects.
   private JLabel fahrLabel         = new JLabel("Degrees Fahrenheit");
   private JLabel celsiusLabel      = new JLabel("Degrees Celsius");
   private JTextField fahrField     = new JTextField("32.0");
   private JTextField celsiusField = new JTextField("0.0");
   // Create a slider with horizontal orientation, minimum value of -400,
   // maximum value of 400, and initially selected value of 32
   private JSlider slider = new JSlider(SwingConstants.HORIZONTAL, -400,
                                        400, 32);

   // Constructor
   public GUIWindow(){
      // Set up panels to organize widgets and
      // add them to the window
      JPanel dataPanel = new JPanel(new GridLayout(2, 2, 12, 6));
      dataPanel.add(fahrLabel);
      dataPanel.add(celsiusLabel);
      dataPanel.add(fahrField);
      dataPanel.add(celsiusField);
      // Single input control now is a slider
      slider.setMajorTickSpacing(50);
      slider.setPaintTicks(true);
      JPanel sliderPanel = new JPanel();
      sliderPanel.add(slider);
      Container container = getContentPane();
```

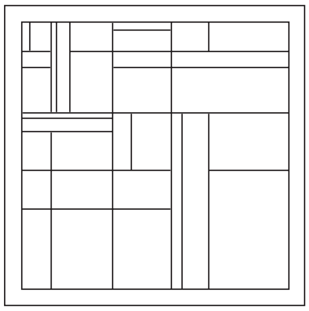
```
         container.add(dataPanel, BorderLayout.CENTER);
         container.add(sliderPanel, BorderLayout.SOUTH);
         // Attach a listener to the slider
         slider.addChangeListener(new SliderListener());
      }

      // >>>>>>> The controller <<<<<<<<

      // Single listener responds to slider movement
      private class SliderListener implements ChangeListener{
         public void stateChanged(ChangeEvent e){
            int fahr = slider.getValue();              // Obtain slider's value
            fahrField.setText("" + fahr);              // Output Fahrenheit value
            thermo.setFahrenheit(fahr);                // Reset thermometer
            double celsius = thermo.getCelsius();      // Obtain Celsius value
            String str = String.format("%.2f", celsius); // Format to 2 places
            celsiusField.setText(str);                 // Output Celsius value
         }
      }
   }
```

Recursive Patterns in Abstract Art

Abstract art is a style of painting and sculpture that originated and flourished in Europe and the United States in the mid-twentieth century. One of its primary features was the use of lines, shapes, and colors to form patterns that made no attempt to represent objects in the physical world. These patterns perhaps expressed an idea, a feeling, or an emotion, but that was left to the viewer's interpretation.

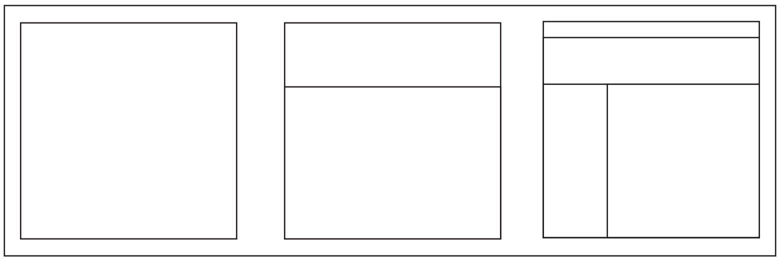
The artist Piet Mondrian (1872–1944) developed a style of abstract painting that exhibited simple recursive patterns. For example, an "idealized" pattern from one of his paintings might look like that shown in Figure 13-27.

FIGURE 13-27



To generate such a pattern with a computer, an algorithm would begin by drawing a rectangle and then repeatedly draw two unequal subdivisions, as shown in Figure 13-28.

FIGURE 13-28
Generating Mondrian-like patterns



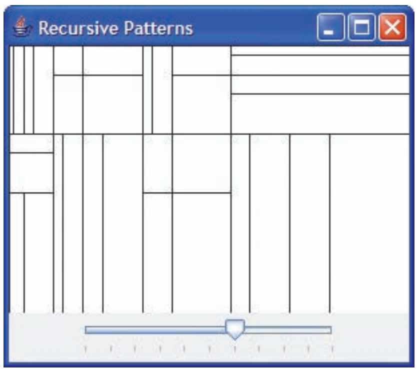
As you can see, the algorithm continues this process of subdivision for a number of levels, until an "aesthetically right moment" is reached. In this version, the algorithm appears to divide the current rectangle into portions representing one-third and two-thirds of its area, and it appears to alternate the subdivisions randomly between the horizontal and vertical axes.

Let's develop a simpler version of this algorithm that divides a rectangle along its vertical axis only. The algorithm starts with a given level and the coordinates of a rectangle's upper-left and lower-right corner points. If the level is greater than 0, the rectangle is drawn and the algorithm is run recursively twice to draw the subdivisions. The first recursive call receives the coordinates of one-third of the rectangle to the left and the second call receives the coordinates of the two-thirds of the rectangle to the right. Both calls receive a level that's one less than the current one. Eventually the level becomes 0 and the algorithm just returns. Here is a pseudocode for this algorithm:

```
mondrian(Graphics g, int x1, int y1, int x2, int y2, int level)
   if (level > 0)
      g.drawRect(x1, y1, x2 - x1, y2 - y1)
      mondrian(g, x1, y1, (x2 - x1) / 3 + x1, y2, level - 1)
      mondrian(g, (x2 - x1) / 3 + x1, y1, x2, y2, level - 1)
```

We now develop a graphics program that allows the user to display Mondrian-like patterns of different levels of detail. The user interface consists of a window with a drawing area and a slider (See Figure 13-29).

FIGURE 13-29
User interface for the Mondrian painting program



The slider allows the user to select levels from 0 to 10. Level 1 produces a simple rectangle at the bounds of the panel, level 2 subdivides the single rectangle, level 3 divides these results, and so on. Here is the code for the main graphics window class:

```
// Example 13.5: Main window for drawing recursive patterns

import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;

public class GUIWindow extends JFrame{

   // Create a slider with horizontal orientation, minimum value of 0,
   // maximum value of 10, and initially selected value of 0
   private JSlider slider = new JSlider(SwingConstants. HORIZONTAL,
                                        0, 10, 0);
   private ColorPanel panel = new ColorPanel(Color.white);
   // Track the current value of the slider for state changes
   private int level = 0;

   public GUIWindow(){
      // Add ticks to the slider and show them
      slider.setMajorTickSpacing(1);
      slider.setPaintTicks(true);
      JPanel sliderPanel = new JPanel();
      sliderPanel.add(slider);
      Container container = getContentPane();
      container.add(panel, BorderLayout.CENTER);
      container.add(sliderPanel, BorderLayout.SOUTH);
      // Attach a listener to the slider
      slider.addChangeListener(new SliderListener());
   }

   // >>>>>>> The controller <<<<<<<<

   private class SliderListener implements ChangeListener{
      public void stateChanged(ChangeEvent e){
         int value = slider.getValue();        // Obtain slider's value
         if (value != level){                  // Check for change in value
            level = value;                     // Reset level if changed
            panel.setLevel(level);             // Draw a new picture
         }
      }
   }
}
```

Note that the `GUIWindow` class maintains an extra instance variable named `level`. This variable tracks the current value of the slider. The reason we need this extra variable is that a slider can emit change events even if its value doesn't change. We don't want to set the panel's level and draw a new picture each time a change event occurs but only when the change event results in a change of the slider's value. The method `stateChanged` handles this restriction by examining and updating `level` if the slider's value has changed since the last event.

The `ColorPanel` class implements our recursive algorithm to draw the rectangles and also uses a random number to choose whether to subdivide them along the horizontal or vertical axis.

Further refinements, including the addition of color, are left as exercises. Here is the code for `ColorPanel`:

```java
// Example 13.5: Panel to draw Mondrian-like paintings

import javax.swing.*;
import java.awt.*;
import java.util.Random;

public class ColorPanel extends JPanel{

   private int level;
   private Random gen;

   public ColorPanel(Color backColor){
      setBackground(backColor);
      setPreferredSize(new Dimension(300, 200));
      level = 0;
      gen = new Random();
   }

   public void setLevel(int newLevel){
      level = newLevel;
      repaint();
   }

   public void paintComponent (Graphics g){
      super.paintComponent(g);
      mondrian(g, 0, 0, getWidth(), getHeight(), level);
   }

   private void mondrian(Graphics g, int x1, int y1, int x2, int y2,
                         int level){
      if (level > 0){
         g.drawRect(x1, y1, x2 - x1, y2 - y1);
         int vertical = gen.nextInt(2) // Decide whether to split vertically
         if (vertical == 0){           // or horizontally.
            mondrian(g, x1, y1, (x2 - x1) / 3 + x1, y2, level - 1);
            mondrian(g, (x2 - x1) / 3 + x1, y1, x2, y2, level - 1);
         }
         else{
            mondrian(g, x1, y1, x2, (y2 - y1) / 3 + y1, level - 1);
            mondrian(g, x1, (y2 - y1) / 3 + y1, x2, y2, level - 1);

         }
      }
   }
}
```

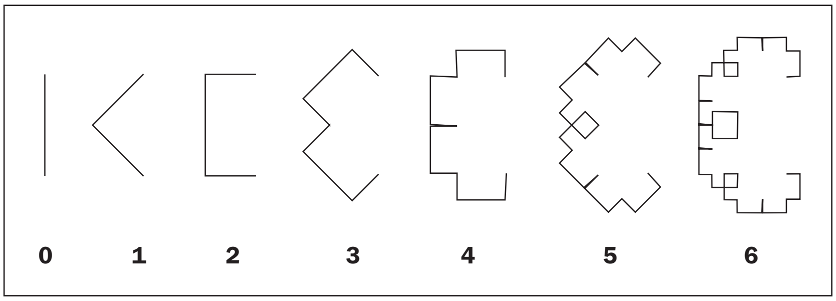
Recursive Patterns in Fractals

Fractals are highly repetitive or recursive patterns. A *fractal object* appears geometric, yet it cannot be described with ordinary Euclidean geometry. Strangely, a fractal curve is not one-dimensional, and a fractal surface is not two-dimensional. Instead, every fractal shape has its own fractal dimension.

An ordinary curve has a precise finite length between any two points. By contrast, a fractal curve has an indefinite length between any two points. The apparent length depends on the level of detail considered. As we zoom in on a segment of a fractal curve, we can see more and more details, and its length appears greater and greater. Consider a coastline. Seen from a distance, it has many wiggles but a discernible length. Now put a piece of the coastline under magnification. It has many similar wiggles, and the discernible length increases. Self-similarity under magnification is the defining characteristic of fractals and is seen in the shapes of mountains, the branching patterns of tree limbs, and many other natural objects.

One example of a fractal curve is a *c-curve*. Figure 13-30 shows c-curves of the first seven degrees. The level-0 c-curve is a simple line segment. The level-1 c-curve replaces the level-0 c-curve with two smaller level-0 c-curves meeting at right angles. The level-2 c-curve does the same thing for each of the two line segments in the level-1 c-curve. This pattern of subdivision can continue indefinitely.

FIGURE 13-30
The first seven degrees of the c-curve



Let's develop an algorithm to draw an *n*-level c-curve. The algorithm receives a graphics object, the current level, and the endpoints of a line segment as parameters. At level 0, the algorithm draws a simple line segment. A level *n* c-curve consists of two level *n* – 1 c-curves constructed as follows:

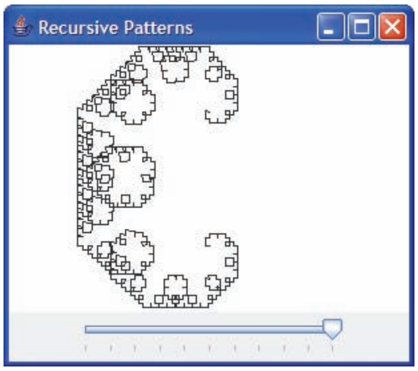
- `Let xm be (x1 + x2 + y1 – y2) / 2.`
- `Let ym be (x2 + y1 + y2 – x1) / 2.`
- The first level *n* – 1 c-curve uses the line segment (*x*1, *y*1), (*xm*, *ym*), and level *n* – 1, so we recur with these parameters.
- The second level *n* – 1 c-curve uses the line segment (*xm*, *ym*), (*x*2, *y*2), and level *n* – 1, so we recur with these parameters.

In effect, as we showed in Figure 13-30, we replace each line segment by two shorter ones that meet at right angles. Here is the pseudocode for the algorithm:

```
cCurve (Graphics g, int x1, int y1, int x2, int y2, int level)
   if (level == 0)
         g.drawLine (x1, y1, x2, y2)
   else
      set xm to (x1 + x2 + y1 - y2) / 2
      set ym to (x2 + y1 + y2 - x1) / 2
      cCurve (g, x1, y1, xm, ym, level - 1)
      cCurve (g, xm, ym, x2, y2, level - 1)
```

The user interface of a program to draw c-curves, as shown in Figure 13-31, is the same one used in the abstract art drawing program. The user adjusts the level by dragging or clicking the slider. The initial window displays a c-curve of level 0. The endpoints of this line segment are (150, 50) and (150, 150). This line segment is a good starting point for higher-degree curves, all of which fit nicely within the initial window boundaries.

FIGURE 13-31
User interface for the c-curve program



The following code shows the implementation of the class `ColorPanel` for drawing c-curves:

```
// Example 13.6: Panel to draw c-curves

import javax.swing.*;
import java.awt.*;
import java.util.Random;

public class ColorPanel extends JPanel{

   private int level;

   public ColorPanel(Color backColor){
      setBackground(backColor);
      setPreferredSize(new Dimension(300, 200));
      level = 0;
   }

   public void setLevel(int newLevel){
      level = newLevel;
      repaint();
   }

   public void paintComponent (Graphics g){
      super.paintComponent(g);
      cCurve(g, 150, 50, 150, 150, level);
   }

   private void cCurve (Graphics g, int x1, int y1, int x2, int y2,
                        int level){
      if (level == 0)
         g.drawLine (x1, y1, x2, y2);
```

```
            else{
               int xm = (x1 + x2 + y1 - y2) / 2;
               int ym = (x2 + y1 + y2 - x1) / 2;
               cCurve (g, x1, y1, xm, ym, level - 1);
               cCurve (g, xm, ym, x2, y2, level - 1);
            }
         }
      }
```

EXERCISE 13.6

1. List how many calls of the `cCurve` method are produced when the initial level is 2, 4, and 8, and give an estimate of the complexity of the algorithm using big-O notation.
2. The `mondrian` method always places the smaller rectangle at the top or the left of a subdivision. Explain how you would modify the method so that it will place the smaller rectangle randomly.

Design, Testing, and Debugging Hints

- When designing a recursive method, be sure that
  1. The method has a well-defined stopping state.
  2. The method has a recursive step that changes the size of the data, so that the stopping state will eventually be reached.
- Recursive methods can be easier to write correctly than the equivalent iterative methods.
- More efficient code is usually more complex than less efficient code. Thus, it may be harder to write more efficient code correctly than less efficient code. Before trying to make your code more efficient, you should demonstrate through analysis that the proposed improvement is really significant (for example, you will get $O(n \log n)$ behavior rather than $O(n^2)$ behavior).

SUMMARY

In this chapter, you learned:

- A recursive method is a method that calls itself to solve a problem.
- Recursive solutions have one or more base cases or termination conditions that return a simple value or `void`. They also have one or more recursive steps that receive a smaller instance of the problem as a parameter.
- Some recursive methods also combine the results of earlier calls to produce a complete solution.
- The run-time behavior of an algorithm can be expressed in terms of big-O notation. This notation shows approximately how the work of the algorithm grows as a function of its problem size.

- There are different orders of complexity, such as constant, linear, quadratic, and exponential.
- Through complexity analysis and clever design, the order of complexity of an algorithm can be reduced to produce a much more efficient algorithm.
- The quicksort is a sort algorithm that uses recursion and can perform much more efficiently than selection sort, bubble sort, or insertion sort.

VOCABULARY *Review*

Define the following terms:

Activation record	Infinite recursion	Recursive step
Big-O notation	Iterative process	Stack
Binary search algorithm	Merge sort	Stack overflow error
Call stack	Quicksort	Stopping state
Complexity analysis	Recursive method	Tail-recursive

REVIEW *Questions*

FILL IN THE BLANK

Complete the following sentences by writing the correct word or words in the blanks provided.

1. The __________ of a recursive algorithm is the part in which a problem is solved directly, without further recursion.
2. The __________ of a recursive algorithm is the part in which the problem is reduced in size.
3. The memory of a computer is formatted into a large __________ to support recursive method calls.
4. The memory for each recursive method call is organized in a group of cells called a(n) __________.
5. The type of error in a recursive algorithm that causes it to run forever is called a(n) __________.
6. When a recursive method does not stop, a(n) __________ error occurs at run time.
7. The linear, quadratic, and logarithmic orders of complexity are expressed as __________, __________, and __________ using big-O notation.
8. The bubble sort algorithm has a run-time complexity of __________ in the best case and __________ in the worst case.
9. The quicksort algorithm has a run-time complexity of __________ in the best case and __________ in the worst case.

PROJECTS

Some of the projects ask you to implement a method and test it in a `Tester` program. Be sure that the method is defined as a `static` method; otherwise, you will get a syntax error.

PROJECT 13-1

Use a `Tester` program to implement and test a recursive method to compute the greatest common divisor (gcd) of two integers. The recursive definition of `gcd` is

```
gcd(a, b) = b, when a = 0
gcd(a, b) = gcd(b, a % b), when a > 0
```

PROJECT 13-2

Write a recursive method that returns a string with the characters in reverse order and test the method with a `Tester` program. The string and the index position should be parameters. If the position is less than the string's length, recurse with the rest of the string after this position and return the result of appending the character at this position to the result. Otherwise, return the empty string.

PROJECT 13-3

Design, implement, and test a recursive method that expects a positive integer parameter and returns a string representing that integer with commas in the appropriate places. The method might be called as follows:

```
String formattedInt = insertCommas(1000000);  // Returns "1,000,000"
```

(*Hint*: Recurse by repeated division and build the string by concatenating after each recursive call.)

PROJECT 13-4

The phrase "*n* choose *k*" is used to refer to the number of ways in which we can choose *k* objects from a set of *n* objects, where *n* >= *k* >= 0. For example, 52 choose 13 would express the number of possible hands that could be dealt in the game of bridge. Write a program that takes the values of *n* and *k* as inputs and displays as output the value *n* choose *k*. Your program should define a recursive method, `nChooseK(n, k)`, that calculates and returns the result. (*Hint*: We can partition the selections of *k* objects from *n* objects as the groups of *k* objects that come from $n - 1$ objects and the groups of *k* objects that include the *n*th object in addition to the groups of $k - 1$ objects chosen from among $n - 1$ objects.) If you test the program with $n = 52$ and $k = 13$, you should be prepared to wait quite a while for the solution!

PROJECT 13-5

Modify the case study of this chapter so that it counts comparison and exchange operations in both sort algorithms and displays these statistics as well. Run the program with two array sizes and make a prediction on the number of comparisons, exchanges, and run times for a third size.

PROJECT 13-6

Write a tester program to help assess the efficiency of the Towers of Hanoi program. This program should be similar to the one developed for the Fibonacci method.

PROJECT 13-7

Write a tester program to help assess the efficiency of the Many Queens program. This program should be similar to the one developed for the Fibonacci method.

PROJECT 13-8

Modify the program that draws Mondrian-like paintings so that it fills each rectangle with a randomly generated color.

CRITICAL *Thinking*

Jill is trying to decide whether to use a recursive algorithm or an iterative algorithm to solve a problem. Explain to her the costs and benefits of using one method or the other.