CHAPTER 13

RECURSION

CHAPTER GOALS

To learn to "think recursively"

To be able to use recursive helper methods

To understand the relationship between recursion and iteration

To understand when the use of recursion affects the efficiency of an algorithm

To analyze problems that are much easier to solve by recursion than by iteration

To process data with recursive structures using mutual recursion

CHAPTER CONTENTS

The method of recursion is a powerful technique for breaking up complex computational problems into simpler, often smaller, ones. The term "recursion" refers to the fact that the same computation recurs, or occurs repeatedly, as the problem is solved. Recursion is often the most natural way of thinking about a problem, and there are some computations that are very difficult to perform without recursion. This chapter shows you both simple and complex examples of recursion and teaches you how to "think recursively".

13.1 Triangle Numbers Revisited

Chapter 5 contains a simple introduction to writing recursive methods—methods that call themselves with simpler inputs. In that chapter, you saw how to print triangle patterns such as this one:

```
[]
[][]
[][][]
[][][][]
```

The key observation is that you can print a triangle pattern of a given side length, provided you know how to print the smaller triangle pattern that is shown in blue.

Using the same method as the one described in this section, you can compute the volume of a Mayan pyramid.

In this section, we will modify the example slightly and use recursion to compute the area of a triangle shape of side length n, assuming that each `[]` square has area 1. This value is sometimes called the nth *triangle number*. For example, as you can tell from looking at the above triangle, the third triangle number is 6 and the fourth triangle number is 10.

We will develop an object-oriented solution that gives another perspective on recursive problem solving. Instead of calling a method with simpler values, we will construct a simpler object.

Here is the outline of the class that we will develop:

```
public class Triangle
{
   private int width;

   public Triangle(int aWidth)
   {
      width = aWidth;
   }

   public int getArea()
   {
      . . .
   }
}
```

If the width of the triangle is 1, then the triangle consists of a single square, and its area is 1. Let's take care of this case first:

```
public int getArea()
{
   if (width == 1) { return 1; }
   . . .
}
```

To deal with the general case, consider this picture:

```
[]
[][]
[][][]
[][][][]
```

Suppose we knew the area of the smaller, colored triangle. Then we could easily compute the area of the larger triangle as

```
smallerArea + width
```

How can we get the smaller area? Let's make a smaller triangle and ask it!

```
Triangle smallerTriangle = new Triangle(width - 1);
int smallerArea = smallerTriangle.getArea();
```

Now we can complete the `getArea` method:

```
public int getArea()
{
   if (width == 1) { return 1; }
   else
   {
      Triangle smallerTriangle = new Triangle(width - 1);
      int smallerArea = smallerTriangle.getArea();
      return smallerArea + width;
   }
}
```

A recursive computation solves a problem by using the solution to the same problem with simpler inputs.

Here is an illustration of what happens when we compute the area of a triangle of width 4.

- The `getArea` method makes a smaller triangle of width 3.
- It calls `getArea` on that triangle.
  - That method makes a smaller triangle of width 2.
  - It calls `getArea` on that triangle.
    - That method makes a smaller triangle of width 1.
    - It calls `getArea` on that triangle.
      - That method returns 1.
    - The method returns `smallerArea + width` = 1 + 2 = 3.
  - The method returns `smallerArea + width` = 3 + 3 = 6.
- The method returns `smallerArea + width` = 6 + 4 = 10.

This solution has one remarkable aspect. To solve the area problem for a triangle of a given width, we use the fact that we can solve the same problem for a lesser width. This is called a *recursive* solution.

The call pattern of a **recursive method** looks complicated, and the key to the successful design of a recursive method is *not to think about it.* Instead, look at the

getArea method one more time and notice how utterly reasonable it is. If the width is 1, then, of course, the area is 1. The next part is just as reasonable. Compute the area of the smaller triangle *and don't think about why that works*. Then the area of the larger triangle is clearly the sum of the smaller area and the width.

There are two key requirements to make sure that the recursion is successful:

- Every recursive call must simplify the computation in some way.
- There must be special cases to handle the simplest computations directly.

For a recursion to terminate, there must be special cases for the simplest values.

The getArea method calls itself again with smaller and smaller width values. Eventually the width must reach 1, and there is a special case for computing the area of a triangle with width 1. Thus, the getArea method always succeeds.

Actually, you have to be careful. What happens when you call the area of a triangle with width –1? It computes the area of a triangle with width –2, which computes the area of a triangle with width –3, and so on. To avoid this, the getArea method should return 0 if the width is ≤ 0.

Recursion is not really necessary to compute the triangle numbers. The area of a triangle equals the sum

```
1 + 2 + 3 + . . . + width
```

Of course, we can program a simple loop:

```
double area = 0;
for (int i = 1; i <= width; i++)
{
   area = area + i;
}
```

Many simple recursions can be computed as loops. However, loop equivalents for more complex recursions—such as the one in our next example—can be complex.

Actually, in this case, you don't even need a loop to compute the answer. The sum of the first n integers can be computed as

$$1 + 2 + \cdots + n = n \times (n + 1)/2$$

Thus, the area equals

```
width * (width + 1) / 2
```

Therefore, neither recursion nor a loop is required to solve this problem. The recursive solution is intended as a "warm-up" to introduce you to the concept of recursion.

section_1/Triangle.java

```
1  /**
2     A triangular shape composed of stacked unit squares like this:
3     []
4     [][]
5     [][][]
6     . . .
7  */
8  public class Triangle
9  {
10    private int width;
11
12    /**
13       Constructs a triangular shape.
14       @param aWidth the width (and height) of the triangle
15    */
```

```
16    public Triangle(int aWidth)
17    {
18       width = aWidth;
19    }
20
21    /**
22       Computes the area of the triangle.
23       @return the area
24    */
25    public int getArea()
26    {
27       if (width <= 0) { return 0; }
28       else if (width == 1) { return 1; }
29       else
30       {
31          Triangle smallerTriangle = new Triangle(width - 1);
32          int smallerArea = smallerTriangle.getArea();
33          return smallerArea + width;
34       }
35    }
36 }
```

section_1/TriangleTester.java

```
1  public class TriangleTester
2  {
3     public static void main(String[] args)
4     {
5        Triangle t = new Triangle(10);
6        int area = t.getArea();
7        System.out.println("Area: " + area);
8        System.out.println("Expected: 55");
9     }
10 }
```

Program Run

```
Area: 55
Expected: 55
```

1. Why is the statement `else if (width == 1) { return 1; }` in the final version of the `getArea` method unnecessary?
2. How would you modify the program to recursively compute the area of a square?
3. In some cultures, numbers containing the digit 8 are lucky numbers. What is wrong with the following method that tries to test whether a number is lucky?

```
public static boolean isLucky(int number)
{
   int lastDigit = number % 10;
   if (lastDigit == 8) { return true; }
   else
   {
      return isLucky(number / 10); // Test the number without the last digit
   }
}
```

4. In order to compute a power of two, you can take the next-lower power and double it. For example, if you want to compute 2^{11} and you know that $2^{10} = 1024$, then $2^{11} = 2 \times 1024 = 2048$. Write a recursive method `public static int pow2(int n)` that is based on this observation.
5. Consider the following recursive method:

```
public static int mystery(int n)
{
   if (n <= 0) { return 0; }
   else
   {
      int smaller = n - 1;
      return mystery(smaller) + n * n;
   }
}
```

What is `mystery(4)`?

Practice It Now you can try these exercises at the end of the chapter: P13.1, P13.2, P13.10.

Common Error 13.1

Infinite Recursion

A common programming error is an infinite recursion: a method calling itself over and over with no end in sight. The computer needs some amount of memory for bookkeeping for each call. After some number of calls, all memory that is available for this purpose is exhausted. Your program shuts down and reports a "stack overflow".

Infinite recursion happens either because the arguments don't get simpler or because a special terminating case is missing. For example, suppose the `getArea` method was allowed to compute the area of a triangle with width `0`. If it weren't for the special test, the method would construct triangles with width –1, –2, –3, and so on.

13.2 Problem Solving: Thinking Recursively

How To 5.2 in Chapter 5 tells you how to solve a problem recursively by pretending that "someone else" will solve the problem for simpler inputs and by focusing on how to turn the simpler solutions into a solution for the whole problem.

In this section, we walk through these steps with a more complex problem: testing whether a sentence is a *palindrome*—a string that is equal to itself when you reverse all characters. Typical examples are

- A man, a plan, a canal—Panama!
- Go hang a salami, I'm a lasagna hog

and, of course, the oldest palindrome of all:

- Madam, I'm Adam

When testing for a palindrome, we match upper- and lowercase letters, and ignore all spaces and punctuation marks.

Thinking recursively is easy if you can recognize a subtask that is similar to the original task.

We want to implement the following `isPalindrome` method:

```
/**
   Tests whether a text is a palindrome.
   @param text a string that is being checked
   @return true if text is a palindrome, false otherwise
*/
public static boolean isPalindrome(String Text)
{
   . . .
}
```

Step 1 Consider various ways to simplify inputs.

In your mind, focus on a particular input or set of inputs for the problem that you want to solve. Think how you can simplify the inputs in such a way that the same problem can be applied to the simpler input.

When you consider simpler inputs, you may want to remove just a little bit from the original input—maybe remove one or two characters from a string, or remove a small portion of a geometric shape. But sometimes it is more useful to cut the input in half and then see what it means to solve the problem for both halves.

In the palindrome test problem, the input is the string that we need to test. How can you simplify the input? Here are several possibilities:

- Remove the first character.
- Remove the last character.
- Remove both the first and last characters.
- Remove a character from the middle.
- Cut the string into two halves.

These simpler inputs are all potential inputs for the palindrome test.

Step 2 Combine solutions with simpler inputs into a solution of the original problem.

In your mind, consider the solutions for the simpler inputs that you discovered in Step 1. Don't worry *how* those solutions are obtained. Simply have faith that the solutions are readily available. Just say to yourself: These are simpler inputs, so someone else will solve the problem for me.

Now think how you can turn the solution for the simpler inputs into a solution for the input that you are currently thinking about. Maybe you need to add a small quantity, perhaps related to the quantity that you lopped off to arrive at the simpler input. Maybe you cut the original input in half and have solutions for each half. Then you may need to add both solutions to arrive at a solution for the whole.

Consider the methods for simplifying the inputs for the palindrome test. Cutting the string in half doesn't seem like a good idea. If you cut

```
"Madam, I'm Adam"
```

in half, you get two strings:

```
"Madam, I"
```

and

```
"'m Adam"
```

The first string isn't a palindrome. Cutting the input in half and testing whether the halves are palindromes seems a dead end.

The most promising simplification is to remove the first *and* last characters. Removing the `M` at the front and the `m` at the back yields

```
"adam, I'm Ada"
```

Suppose you can verify that the shorter string is a palindrome. Then *of course* the original string is a palindrome—we put the same letter in the front and the back. That's extremely promising. A word is a palindrome if

- The first and last letters match (ignoring letter case).

and

- The word obtained by removing the first and last letters is a palindrome.

Again, don't worry how the test works for the shorter string. It just works.

There is one other case to consider. What if the first or last letter of the word is not a letter? For example, the string

```
"A man, a plan, a canal, Panama!"
```

ends in a `!` character, which does not match the `A` in the front. But we should ignore non-letters when testing for palindromes. Thus, when the last character is not a letter but the first character is a letter, it doesn't make sense to remove both the first and the last characters. That's not a problem. Remove only the last character. If the shorter string is a palindrome, then it stays a palindrome when you attach a nonletter.

The same argument applies if the first character is not a letter. Now we have a complete set of cases.

- If the first and last characters are both letters, then check whether they match. If so, remove both and test the shorter string.
- Otherwise, if the last character isn't a letter, remove it and test the shorter string.
- Otherwise, the first character isn't a letter. Remove it and test the shorter string.

In all three cases, you can use the solution to the simpler problem to arrive at a solution to your problem.

Step 3 Find solutions to the simplest inputs.

A recursive computation keeps simplifying its inputs. Eventually it arrives at very simple inputs. To make sure that the recursion comes to a stop, you must deal with the simplest inputs separately. Come up with special solutions for them, which is usually very easy.

However, sometimes you get into philosophical questions dealing with *degenerate* inputs: empty strings, shapes with no area, and so on. Then you may want to investigate a slightly larger input that gets reduced to such a trivial input and see what value you should attach to the degenerate inputs so that the simpler value, when used according to the rules you discovered in Step 2, yields the correct answer.

Let's look at the simplest strings for the palindrome test:

- Strings with two characters
- Strings with a single character
- The empty string

We don't have to come up with a special solution for strings with two characters. Step 2 still applies to those strings—either or both of the characters are removed. But we

do need to worry about strings of length 0 and 1. In those cases, Step 2 can't apply. There aren't two characters to remove.

The empty string is a palindrome—it's the same string when you read it backwards. If you find that too artificial, consider a string "mm". According to the rule discovered in Step 2, this string is a palindrome if the first and last characters of that string match and the remainder—that is, the empty string—is also a palindrome. Therefore, it makes sense to consider the empty string a palindrome.

A string with a single letter, such as "I", is a palindrome. How about the case in which the character is not a letter, such as "!"? Removing the ! yields the empty string, which is a palindrome. Thus, we conclude that all strings of length 0 or 1 are palindromes.

Step 4 Implement the solution by combining the simple cases and the reduction step.

Now you are ready to implement the solution. Make separate cases for the simple inputs that you considered in Step 3. If the input isn't one of the simplest cases, then implement the logic you discovered in Step 2.

ONLINE EXAMPLE

The Palindromes class.

Here is the isPalindrome method:

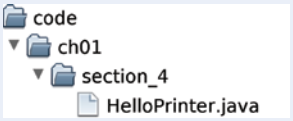
```
public static boolean isPalindrome(String text)
{
   int length = text.length();

   // Separate case for shortest strings.
   if (length <= 1) { return true; }
   else
   {
      // Get first and last characters, converted to lowercase.
      char first = Character.toLowerCase(text.charAt(0));
      char last = Character.toLowerCase(text.charAt(length - 1));

      if (Character.isLetter(first) && Character.isLetter(last))
      {
         // Both are letters.
         if (first == last)
         {
            // Remove both first and last character.
            String shorter = text.substring(1, length - 1);
            return isPalindrome(shorter);
         }
         else
         {
            return false;
         }
      }
      else if (!Character.isLetter(last))
      {
         // Remove last character.
         String shorter = text.substring(0, length - 1);
         return isPalindrome(shorter);
      }
      else
      {
         // Remove first character.
         String shorter = text.substring(1);
         return isPalindrome(shorter);
      }
   }
}
```

6. Consider the task of removing all punctuation marks from a string. How can we break the string into smaller strings that can be processed recursively?
7. In a recursive method that removes all punctuation marks from a string, we decide to remove the last character, then recursively process the remainder. How do you combine the results?
8. How do you find solutions for the simplest inputs when removing punctuation marks from a string?
9. Provide pseudocode for a recursive method that removes punctuation marks from a string, using the answers to Self Checks 6–8.

Practice It Now you can try these exercises at the end of the chapter: R13.3, P13.3, P13.6.

WORKED EXAMPLE 13.1 **Finding Files**

In this Worked Example, we find all files with a given extension in a directory tree.



13.3 Recursive Helper Methods

Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

Sometimes it is easier to find a recursive solution if you change the original problem slightly. Then the original problem can be solved by calling a recursive helper method.

Here is a typical example: Consider the palindrome test of Section 13.2. It is a bit inefficient to construct new string objects in every step. Now consider the following change in the problem. Rather than testing whether the entire sentence is a palindrome, let's check whether a substring is a palindrome:

Sometimes, a task can be solved by handing it off to a recursive helper method.

```
/**
   Tests whether a substring is a palindrome.
   @param text a string that is being checked
   @param start the index of the first character of the substring
   @param end the index of the last character of the substring
   @return true if the substring is a palindrome
*/
public static boolean isPalindrome(String text, int start, int end)
```

This method turns out to be even easier to implement than the original test. In the recursive calls, simply adjust the `start` and `end` parameter variables to skip over matching letter pairs and characters that are not letters. There is no need to construct new `String` objects to represent the shorter strings.

```
public static boolean isPalindrome(String text, int start, int end)
{
```

Available online in WileyPLUS and at `www.wiley.com/college/horstmann`.

```
   // Separate case for substrings of length 0 and 1.
   if (start >= end) { return true; }
   else
   {
      // Get first and last characters, converted to lowercase.
      char first = Character.toLowerCase(text.charAt(start));
      char last = Character.toLowerCase(text.charAt(end));

      if (Character.isLetter(first) && Character.isLetter(last))
      {
         if (first == last)
         {
            // Test substring that doesn't contain the matching letters.
            return isPalindrome(text, start + 1, end - 1);
         }
         else
         {
            return false;
         }
      }
      else if (!Character.isLetter(last))
      {
         // Test substring that doesn't contain the last character.
         return isPalindrome(text, start, end - 1);
      }
      else
      {
         // Test substring that doesn't contain the first character.
         return isPalindrome(text, start + 1, end);
      }
   }
}
```

You should still supply a method to solve the whole problem—the user of your method shouldn't have to know about the trick with the substring positions. Simply call the helper method with positions that test the entire string:

```
public static boolean isPalindrome(String text)
{
   return isPalindrome(text, 0, text.length() - 1);
}
```

ONLINE EXAMPLE

The `Palindromes` class with a helper method.

Note that this call is *not* a recursive method call. The `isPalindrome(String)` method calls the helper method `isPalindrome(String, int, int)`. In this example, we use overloading to declare two methods with the same name. The `isPalindrome` method with just a `String` parameter variable is the method that we expect the public to use. The second method, with one `String` and two `int` parameter variables, is the recursive helper method. If you prefer, you can avoid overloaded methods by choosing a different name for the helper method, such as `substringIsPalindrome`.

Use the technique of recursive helper methods whenever it is easier to solve a recursive problem that is equivalent to the original problem—but more amenable to a recursive solution.

10. Do we have to give the same name to both `isPalindrome` methods?

11. When does the recursive `isPalindrome` method stop calling itself?

12. To compute the sum of the values in an array, add the first value to the sum of the remaining values, computing recursively. Of course, it would be inefficient to set

up an actual array of the remaining values. Which recursive helper method can solve the problem?

13. How can you write a recursive method `public static void sum(int[] a)` without needing a helper function?

Practice It Now you can try these exercises at the end of the chapter: P13.4, P13.7, 13.11.

13.4 The Efficiency of Recursion

As you have seen in this chapter, recursion can be a powerful tool to implement complex algorithms. On the other hand, recursion can lead to algorithms that perform poorly. In this section, we will analyze the question of when recursion is beneficial and when it is inefficient.

In most cases, iterative and recursive approaches have comparable efficiency.

Consider the Fibonacci sequence: a sequence of numbers defined by the equation

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

That is, each value of the sequence is the sum of the two preceding values. The first ten terms of the sequence are

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55$$

It is easy to extend this sequence indefinitely. Just keep appending the sum of the last two values of the sequence. For example, the next entry is 34 + 55 = 89.

We would like to write a method that computes f_n for any value of n. Here we translate the definition directly into a recursive method:

section_4/RecursiveFib.java

```
1  import java.util.Scanner;
2
3  /**
4     This program computes Fibonacci numbers using a recursive method.
5  */
6  public class RecursiveFib
7  {
8     public static void main(String[] args)
9     {
10       Scanner in = new Scanner(System.in);
11       System.out.print("Enter n: ");
12       int n = in.nextInt();
13
14       for (int i = 1; i <= n; i++)
15       {
16          long f = fib(i);
17          System.out.println("fib(" + i + ") = " + f);
18       }
```

```
19      }
20
21      /**
22         Computes a Fibonacci number.
23         @param n an integer
24         @return the nth Fibonacci number
25      */
26      public static long fib(int n)
27      {
28         if (n <= 2) { return 1; }
29         else { return fib(n - 1) + fib(n - 2); }
30      }
31   }
```

Program Run

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

That is certainly simple, and the method will work correctly. But watch the output closely as you run the test program. The first few calls to the `fib` method are fast. For larger values, though, the program pauses an amazingly long time between outputs.

That makes no sense. Armed with pencil, paper, and a pocket calculator you could calculate these numbers pretty quickly, so it shouldn't take the computer anywhere near that long.

To find out the problem, let us insert **trace messages** into the method:

section_4/RecursiveFibTracer.java

```
1   import java.util.Scanner;
2
3   /**
4      This program prints trace messages that show how often the
5      recursive method for computing Fibonacci numbers calls itself.
6   */
7   public class RecursiveFibTracer
8   {
9      public static void main(String[] args)
10     {
11        Scanner in = new Scanner(System.in);
12        System.out.print("Enter n: ");
13        int n = in.nextInt();
14
15        long f = fib(n);
16
17        System.out.println("fib(" + n + ") = " + f);
18     }
19
20     /**
```

```
21       Computes a Fibonacci number.
22       @param n an integer
23       @return the nth Fibonacci number
24    */
25    public static long fib(int n)
26    {
27       System.out.println("Entering fib: n = " + n);
28       long f;
29       if (n <= 2) { f = 1; }
30       else { f = fib(n - 1) + fib(n - 2); }
31       System.out.println("Exiting fib: n = " + n
32             + " return value = " + f);
33       return f;
34    }
35 }
```

Program Run

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Exiting fib: n = 5 return value = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8
```

Figure 1 shows the pattern of recursive calls for computing `fib(6)`. Now it is becoming apparent why the method takes so long. It is computing the same values over and over. For example, the computation of `fib(6)` calls `fib(4)` twice and `fib(3)` three times. That is very different from the computation we would do with pencil and paper. There we would just write down the values as they were computed and add up the last two to get the next one until we reached the desired entry; no sequence value would ever be computed twice.

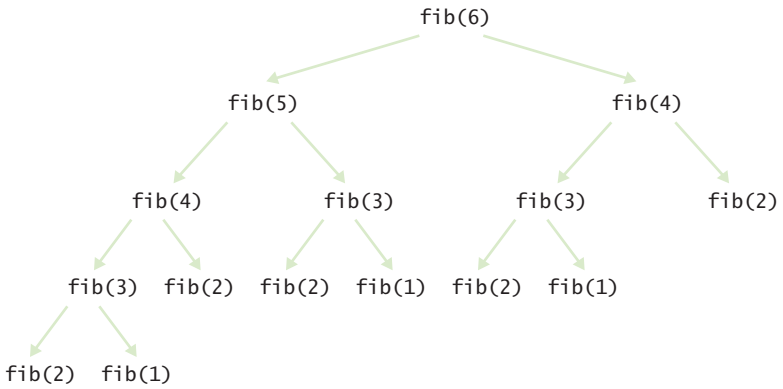


Figure 1 Call Pattern of the Recursive `fib` Method

If we imitate the pencil-and-paper process, then we get the following program:

section_4/LoopFib.java

```
1  import java.util.Scanner;
2
3  /**
4     This program computes Fibonacci numbers using an iterative method.
5  */
6  public class LoopFib
7  {
8     public static void main(String[] args)
9     {
10       Scanner in = new Scanner(System.in);
11       System.out.print("Enter n: ");
12       int n = in.nextInt();
13
14       for (int i = 1; i <= n; i++)
15       {
16          long f = fib(i);
17          System.out.println("fib(" + i + ") = " + f);
18       }
19    }
20
21    /**
22       Computes a Fibonacci number.
23       @param n an integer
24       @return the nth Fibonacci number
25    */
26    public static long fib(int n)
27    {
28       if (n <= 2) { return 1; }
29       else
30       {
31          long olderValue = 1;
32          long oldValue = 1;
33          long newValue = 1;
34          for (int i = 3; i <= n; i++)
35          {
36             newValue = oldValue + olderValue;
37             olderValue = oldValue;
38             oldValue = newValue;
```

```
39            }
40            return newValue;
41         }
42      }
43   }
```

Program Run

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

This method runs *much* faster than the recursive version.

In this example of the `fib` method, the recursive solution was easy to program because it exactly followed the mathematical definition, but it ran far more slowly than the iterative solution, because it computed many intermediate results multiple times.

Occasionally, a recursive solution runs much slower than its iterative counterpart. However, in most cases, the recursive solution is only slightly slower.

Can you always speed up a recursive solution by changing it into a loop? Frequently, the iterative and recursive solution have essentially the same performance. For example, here is an iterative solution for the palindrome test:

```
public static boolean isPalindrome(String text)
{
   int start = 0;
   int end = text.length() - 1;
   while (start < end)
   {
      char first = Character.toLowerCase(text.charAt(start));
      char last = Character.toLowerCase(text.charAt(end));

      if (Character.isLetter(first) && Character.isLetter(last))
      {
         // Both are letters.
         if (first == last)
         {
            start++;
            end--;
         }
         else
         {
            return false;
         }
      }
      if (!Character.isLetter(last)) { end--; }
      if (!Character.isLetter(first)) { start++; }
   }
   return true;
}
```

ONLINE EXAMPLE

The `LoopPalindromes` class.

This solution keeps two index variables: `start` and `end`. The first index starts at the beginning of the string and is advanced whenever a letter has been matched or a

nonletter has been ignored. The second index starts at the end of the string and moves toward the beginning. When the two index variables meet, the iteration stops.

Both the iteration and the recursion run at about the same speed. If a palindrome has n characters, the iteration executes the loop between $n/2$ and n times, depending on how many of the characters are letters, because one or both index variables are moved in each step. Similarly, the recursive solution calls itself between $n/2$ and n times, because one or two characters are removed in each step.

In such a situation, the iterative solution tends to be a bit faster, because each recursive method call takes a certain amount of processor time. In principle, it is possible for a smart compiler to avoid recursive method calls if they follow simple patterns, but most Java compilers don't do that. From that point of view, an iterative solution is preferable.

In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

However, many problems have recursive solutions that are easier to understand and implement correctly than their iterative counterparts. Sometimes there is no obvious iterative solution at all—see the example in the next section. There is a certain elegance and economy of thought to recursive solutions that makes them more appealing. As the computer scientist (and creator of the GhostScript interpreter for the PostScript graphics description language) L. Peter Deutsch put it: "To iterate is human, to recurse divine."

14. Is it faster to compute the triangle numbers recursively, as shown in Section 13.1, or is it faster to use a loop that computes `1 + 2 + 3 + . . . + width`?

15. You can compute the factorial function either with a loop, using the definition that $n! = 1 \times 2 \times \ldots \times n$, or recursively, using the definition that $0! = 1$ and $n! = (n - 1)! \times n$. Is the recursive approach inefficient in this case?

16. To compute the sum of the values in an array, you can split the array in the middle, recursively compute the sums of the halves, and add the results. Compare the performance of this algorithm with that of a loop that adds the values.

Practice It Now you can try these exercises at the end of the chapter: R13.7, R13.9. P13.5, P13.25.

13.5 Permutations

The permutations of a string can be obtained more naturally through recursion than with a loop.

In this section, we will study a more complex example of recursion that would be difficult to program with a simple loop. (As Exercise P13.11 shows, it is possible to avoid the recursion, but the resulting solution is quite complex, and no faster).

We will design a method that lists all permutations of a string. A permutation is simply a rearrangement of the letters in the string. For example, the string `"eat"` has six permutations (including the original string itself):

```
"eat"
"eta"
"aet"
"ate"
"tea"
"tae"
```

Using recursion, you can find all arrangements of a set of objects.

Now we need a way to generate the permutations recursively. Consider the string "eat". Let's simplify the problem. First, we'll generate all permutations that start with the letter 'e', then those that start with 'a', and finally those that start with 't'. How do we generate the permutations that start with 'e'? We need to know the permutations of the substring "at". But that's the same problem—to generate all permutations—with a simpler input, namely the shorter string "at". Thus, we can use recursion. Generate the permutations of the substring "at". They are

```
"at"
"ta"
```

For each permutation of that substring, prepend the letter 'e' to get the permutations of "eat" that start with 'e', namely

```
"eat"
"eta"
```

Now let's turn our attention to the permutations of "eat" that start with 'a'. We need to produce the permutations of the remaining letters, "et". They are:

```
"et"
"te"
```

We add the letter 'a' to the front of the strings and obtain

```
"aet"
"ate"
```

We generate the permutations that start with 't' in the same way.

That's the idea. The implementation is fairly straightforward. In the `permutations` method, we loop through all positions in the word to be permuted. For each of them, we compute the shorter word that is obtained by removing the `i`th letter:

```
String shorter = word.substring(0, i) + word.substring(i + 1);
```

We compute the permutations of the shorter word:

```
ArrayList<String> shorterPermutations = permutations(shorter);
```

Finally, we add the removed letter to the front of all permutations of the shorter word.

```
for (String s : shorterPermutations)
{
   result.add(word.charAt(i) + s);
}
```

As always, we have to provide a special case for the simplest strings. The simplest possible string is the empty string, which has a single permutation—itself.

Here is the complete `Permutations` class:

section_5/Permutations.java

```
1  import java.util.ArrayList;
2
3  /**
4     This class computes permutations of a string.
5  */
6  public class Permutations
7  {
8     public static void main(String[] args)
9     {
10       for (String s : permutations("eat"))
11       {
```

```
12          System.out.println(s);
13       }
14    }
15
16    /**
17       Gets all permutations of a given word.
18       @param word the string to permute
19       @return a list of all permutations
20    */
21    public static ArrayList<String> permutations(String word)
22    {
23       ArrayList<String> result = new ArrayList<String>();
24
25       // The empty string has a single permutation: itself
26       if (word.length() == 0)
27       {
28          result.add(word);
29          return result;
30       }
31       else
32       {
33          // Loop through all character positions
34          for (int i = 0; i < word.length(); i++)
35          {
36             // Form a shorter word by removing the ith character
37             String shorter = word.substring(0, i) + word.substring(i + 1);
38
39             // Generate all permutations of the simpler word
40             ArrayList<String> shorterPermutations = permutations(shorter)
41
42             // Add the removed character to the front of
43             // each permutation of the simpler word
44             for (String s : shorterPermutations)
45             {
46                result.add(word.charAt(i) + s);
47             }
48          }
49          // Return all permutations
50          return result;
51       }
52    }
53 }
```

Program Run

```
eat
eta
aet
ate
tea
tae
```

Compare the `Permutations` and `Triangle` classes. Both of them work on the same principle. When they work on a more complex input, they first solve the problem for a simpler input. Then they combine the result for the simpler input with additional work to deliver the results for the more complex input. There really is no particular complexity behind that process as long as you think about the solution on that level only.

However, behind the scenes, the simpler input creates even simpler input, which creates yet another simplification, and so on, until one input is so simple that the result can be obtained without further help. It is interesting to think about this process, but it can also be confusing. What's important is that you can focus on the one level that matters—putting a solution together from the slightly simpler problem, ignoring the fact that the simpler problem also uses recursion to get its results.

Random Fact 13.1 The Limits of Computation

Have you ever wondered how your instructor or grader makes sure your programming homework is correct? In all likelihood, they look at your solution and perhaps run it with some test inputs. But usually they have a correct solution available. That suggests that there might be an easier way. Perhaps they could feed your program and their correct program into a "program comparator", a computer program that analyzes both programs and determines whether they both compute the same results. Of course, your solution and the program that is known to be correct need not be identical—what matters is that they produce the same output when given the same input.

How could such a program comparator work? Well, the Java compiler knows how to read a program and make sense of the classes, methods, and statements. So it seems plausible that someone could, with some effort, write a program that reads two Java programs, analyzes what they do, and determines whether they solve the same task. Of course, such a program would be very attractive to instructors, because it could automate the grading process. Thus, even though no such program exists today, it might be tempting to try to develop one and sell it to universities around the world.

However, before you start raising venture capital for such an effort, you should know that theoretical computer scientists have proven that it is impossible to develop such a program, *no matter how hard you try*.

There are quite a few of these unsolvable problems. The first one, called the *halting problem,* was discovered by the British researcher Alan Turing in 1936. Because his research occurred before the first actual computer was constructed, Turing had to devise a theoretical device, the *Turing machine,* to explain how computers could work. The Turing machine consists of a long magnetic tape, a read/write head, and a program that has numbered instructions of the form: "If the current symbol under the head is *x*, then replace it with *y*, move the head one unit left or right, and continue with instruction *n*" (see figure below). Interestingly enough, with only these instructions, you can program just as much as with Java, even though it is incredibly tedious to do so. Theoretical computer scientists like Turing machines because they can be described using nothing more than the laws of mathematics.

Expressed in terms of Java, the halting problem states: "It is impossible to write a program with two inputs, namely the source code of an arbitrary Java program *P* and a string *I*, that decides whether the program *P*, when executed with the input *I*, will halt—that is, the program will not get into an infinite loop with the given input". Of course, for some kinds of programs and inputs, it is possible to decide whether the program halts with the given input. The halting problem asserts that it is impossible to come up with a single decision-making algorithm that works with all programs and inputs. Note that you can't simply run the program *P* on the input *I* to settle this question. If the program runs for 1,000 days, you don't know that the program is in an infinite loop. Maybe you just have to wait another day for it to stop.

Such a "halt checker", if it could be written, might also be useful for grading homework. An instructor could use it to screen student submissions to see if they get into an infinite loop with a particular input, and then stop checking them. However, as Turing demonstrated, such a program cannot be written. His argument is ingenious and quite simple.

Suppose a "halt checker" program existed. Let's call it *H*. From *H,* we will develop another program, the "killer" program *K*. *K* does the following computation. Its input is a string containing the source code for a program *R*. It then applies the halt checker on the input program *R* and the input string *R*. That is, it checks whether the program *R* halts if its input is its own source code. It sounds bizarre to feed a program to itself, but it isn't impossible.

Alan Turing

17. What are all permutations of the four-letter word `beat`?
18. Our recursion for the permutation generator stops at the empty string. What simple modification would make the recursion stop at strings of length 0 or 1?
19. Why isn't it easy to develop an iterative solution for the permutation generator?

Practice It Now you can try these exercises at the end of the chapter: P13.11, P13.12, P13.13.

For example, the Java compiler is written in Java, and you can use it to compile itself. Or, as a simpler example, a word counting program can count the words in its own source code.

When *K* gets the answer from *H* that *R* halts when applied to itself, it is programmed to enter an infinite loop. Otherwise *K* exits. In Java, the program might look like this:

```
public class Killer
{
   public static void main(
      String[] args)
   {
      String r = read program input;
      HaltChecker checker =
         new HaltChecker();
      if (checker.check(r, r))
      {
         while (true)
         { // Infinite loop
         }
      }
      else
      {
         return;
      }
   }
}
```

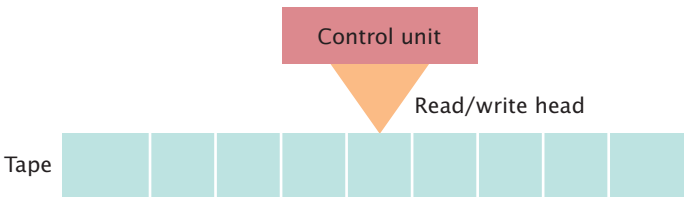
Now ask yourself: What does the halt checker answer when asked whether *K* halts when given *K* as the input? Maybe it finds out that *K* gets into an infinite loop with such an input. But wait, that can't be right. That would mean that `checker.check(r, r)` returns `false` when `r` is the program code of *K*. As you can plainly see, in that case, the `killer` method returns, so K didn't get into an infinite loop. That shows that *K* must halt when analyzing itself, so `checker.check(r, r)` should return `true`. But then the `killer` method doesn't terminate—it goes into an infinite loop. That shows that it is logically impossible to implement a program that can check whether *every* program halts on a particular input.

It is sobering to know that there are *limits* to computing. There are problems that no computer program, no matter how ingenious, can answer.

Theoretical computer scientists are working on other research involving the nature of computation. One important question that remains unsettled to this day deals with problems that in practice are very time-consuming to solve. It may be that these problems are intrinsically hard, in which case it would be pointless to try to look for better algorithms. Such theoretical research can have important practical applications. For example, right now, nobody knows whether the most common encryption schemes used today could be broken by discovering a new algorithm. Knowing that no fast algorithms exist for breaking a particular code could make us feel more comfortable about the security of encryption.

Program

Instruction number	If tape symbol is	Replace with	Then move head	Then go to instruction
1	0	2	right	2
	1	1	left	4
2	0	0	right	2
	1	1	right	2
	2	0	left	3
3	0	0	left	3
	1	1	left	3
	2	2	right	1
4	1	1	right	5
	2	0	left	4



The Turing Machine

13.6 Mutual Recursion

In a mutual recursion, a set of cooperating methods calls each other repeatedly.

In the preceding examples, a method called itself to solve a simpler problem. Sometimes, a set of cooperating methods calls each other in a recursive fashion. In this section, we will explore such a **mutual recursion**. This technique is significantly more advanced than the simple recursion that we discussed in the preceding sections.

We will develop a program that can compute the values of arithmetic expressions such as

```
3+4*5
(3+4)*5
1-(2-(3-(4-5)))
```

Computing such an expression is complicated by the fact that `*` and `/` bind more strongly than `+` and `-`, and that parentheses can be used to group subexpressions.

Figure 2 shows a set of **syntax diagrams** that describes the syntax of these expressions. To see how the syntax diagrams work, consider the expression `3+4*5`:

- Enter the *expression* syntax diagram. The arrow points directly to *term*, giving you no alternative.
- Enter the *term* syntax diagram. The arrow points to *factor*, again giving you no choice.
- Enter the *factor* diagram. You have two choices: to follow the top branch or the bottom branch. Because the first input token is the number `3` and not a `(`, follow the bottom branch.
- Accept the input token because it matches the number. The unprocessed input is now `+4*5`.
- Follow the arrow out of *number* to the end of *factor*. As in a method call, you now back up, returning to the end of the *factor* element of the *term* diagram.

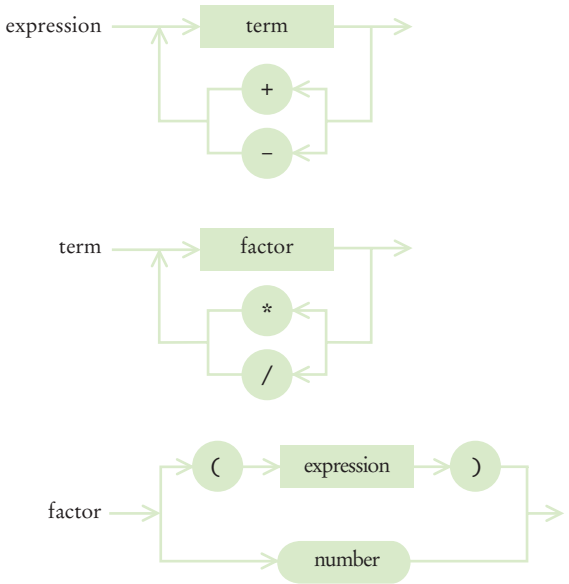


Figure 2 Syntax Diagrams for Evaluating an Expression

- Now you have another choice—to loop back in the *term* diagram, or to exit. The next input token is a `+`, and it matches neither the `*` or the `/` that would be required to loop back. So you exit, returning to *expression*.
- Again, you have a choice, to loop back or to exit. Now the `+` matches one of the choices in the loop. Accept the `+` in the input and move back to the *term* element. The remaining input is `4*5`.

In this fashion, an expression is broken down into a sequence of terms, separated by `+` or `-`, each term is broken down into a sequence of factors, each separated by `*` or `/`, and each factor is either a parenthesized expression or a number. You can draw this breakdown as a tree. Figure 3 shows how the expressions `3+4*5` and `(3+4)*5` are derived from the syntax diagram.

Why do the syntax diagrams help us compute the value of the tree? If you look at the syntax trees, you will see that they accurately represent which operations should be carried out first. In the first tree, 4 and 5 should be multiplied, and then the result should be added to 3. In the second tree, 3 and 4 should be added, and the result should be multiplied by 5.

At the end of this section, you will find the implementation of the `Evaluator` class, which evaluates these expressions. The `Evaluator` makes use of an `ExpressionTokenizer` class, which breaks up an input string into tokens—numbers, operators, and parentheses. (For simplicity, we only accept positive integers as numbers, and we don't allow spaces in the input.)

When you call `nextToken`, the next input token is returned as a string. We also supply another method, `peekToken`, which allows you to see the next token without consuming it. To see why the `peekToken` method is necessary, consider the syntax diagram of the term type. If the next token is a `"*"` or `"/"`, you want to continue adding and subtracting terms. But if the next token is another character, such as a `"+"` or `"-"`, you want to stop without actually consuming it, so that the token can be considered later.

To compute the value of an expression, we implement three methods: `getExpressionValue`, `getTermValue`, and `getFactorValue`. The `getExpressionValue` method first calls `getTermValue` to get the value of the first term of the expression. Then it

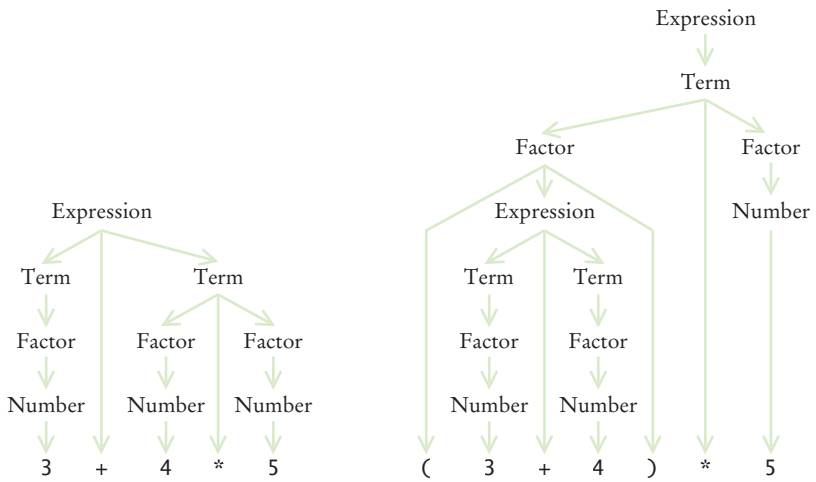


Figure 3 Syntax Trees for Two Expressions

checks whether the next input token is one of + or -. If so, it calls `getTermValue` again and adds or subtracts it.

```
public int getExpressionValue()
{
   int value = getTermValue();
   boolean done = false;
   while (!done)
   {
      String next = tokenizer.peekToken();
      if ("+".equals(next) || "-".equals(next))
      {
         tokenizer.nextToken(); // Discard "+" or "-"
         int value2 = getTermValue();
         if ("+".equals(next)) { value = value + value2; }
         else { value = value - value2; }
      }
      else
      {
         done = true;
      }
   }
   return value;
}
```

The `getTermValue` method calls `getFactorValue` in the same way, multiplying or dividing the factor values.

Finally, the `getFactorValue` method checks whether the next input is a number, or whether it begins with a `(` token. In the first case, the value is simply the value of the number. However, in the second case, the `getFactorValue` method makes a recursive call to `getExpressionValue`. Thus, the three methods are mutually recursive.

```
public int getFactorValue()
{
   int value;
   String next = tokenizer.peekToken();
   if ("(".equals(next))
   {
      tokenizer.nextToken(); // Discard "("
      value = getExpressionValue();
      tokenizer.nextToken(); // Discard ")"
   }
   else
   {
      value = Integer.parseInt(tokenizer.nextToken());
   }
   return value;
}
```

To see the mutual recursion clearly, trace through the expression `(3+4)*5`:

- `getExpressionValue` calls `getTermValue`
  - `getTermValue` calls `getFactorValue`
    - `getFactorValue` consumes the `(` input
    - `getFactorValue` calls `getExpressionValue`
      - `getExpressionValue` returns eventually with the value of 7, having consumed `3 + 4`. This is the recursive call.
    - `getFactorValue` consumes the `)` input

- `getFactorValue` returns 7
- `getTermValue` consumes the inputs * and 5 and returns 35

- `getExpressionValue` returns 35

As always with a recursive solution, you need to ensure that the recursion terminates. In this situation, that is easy to see when you consider the situation in which `getExpressionValue` calls itself. The second call works on a shorter subexpression than the original expression. At each recursive call, at least some of the tokens of the input string are consumed, so eventually the recursion must come to an end.

section_6/Evaluator.java

```
1  /**
2     A class that can compute the value of an arithmetic expression.
3  */
4  public class Evaluator
5  {
6     private ExpressionTokenizer tokenizer;
7
8     /**
9        Constructs an evaluator.
10       @param anExpression a string containing the expression
11       to be evaluated
12    */
13    public Evaluator(String anExpression)
14    {
15       tokenizer = new ExpressionTokenizer(anExpression);
16    }
17
18    /**
19       Evaluates the expression.
20       @return the value of the expression
21    */
22    public int getExpressionValue()
23    {
24       int value = getTermValue();
25       boolean done = false;
26       while (!done)
27       {
28          String next = tokenizer.peekToken();
29          if ("+".equals(next) || "-".equals(next))
30          {
31             tokenizer.nextToken(); // Discard "+" or "-"
32             int value2 = getTermValue();
33             if ("+".equals(next)) { value = value + value2; }
34             else { value = value - value2; }
35          }
36          else
37          {
38             done = true;
39          }
40       }
41       return value;
42    }
43
44    /**
45       Evaluates the next term found in the expression.
46       @return the value of the term
47    */
```

```
48    public int getTermValue()
49    {
50       int value = getFactorValue();
51       boolean done = false;
52       while (!done)
53       {
54          String next = tokenizer.peekToken();
55          if ("*".equals(next) || "/".equals(next))
56          {
57             tokenizer.nextToken();
58             int value2 = getFactorValue();
59             if ("*".equals(next)) { value = value * value2; }
60             else { value = value / value2; }
61          }
62          else
63          {
64             done = true;
65          }
66       }
67       return value;
68    }
69
70    /**
71       Evaluates the next factor found in the expression.
72       @return the value of the factor
73    */
74    public int getFactorValue()
75    {
76       int value;
77       String next = tokenizer.peekToken();
78       if ("(".equals(next))
79       {
80          tokenizer.nextToken(); // Discard "("
81          value = getExpressionValue();
82          tokenizer.nextToken(); // Discard ")"
83       }
84       else
85       {
86          value = Integer.parseInt(tokenizer.nextToken());
87       }
88       return value;
89    }
90 }
```

section_6/ExpressionTokenizer.java

```
1  /**
2     This class breaks up a string describing an expression
3     into tokens: numbers, parentheses, and operators.
4  */
5  public class ExpressionTokenizer
6  {
7     private String input;
8     private int start; // The start of the current token
9     private int end; // The position after the end of the current token
10
11    /**
12       Constructs a tokenizer.
13       @param anInput the string to tokenize
14    */
```

```
15    public ExpressionTokenizer(String anInput)
16    {
17       input = anInput;
18       start = 0;
19       end = 0;
20       nextToken(); //  Find the first token
21    }
22
23    /**
24       Peeks at the next token without consuming it.
25       @return the next token or null if there are no more tokens
26    */
27    public String peekToken()
28    {
29       if (start >= input.length()) { return null; }
30       else { return input.substring(start, end); }
31    }
32
33    /**
34       Gets the next token and moves the tokenizer to the following token.
35       @return the next token or null if there are no more tokens
36    */
37    public String nextToken()
38    {
39       String r = peekToken();
40       start = end;
41       if (start >= input.length()) { return r; }
42       if (Character.isDigit(input.charAt(start)))
43       {
44          end = start + 1;
45          while (end < input.length()
46                && Character.isDigit(input.charAt(end)))
47          {
48             end++;
49          }
50       }
51       else
52       {
53          end = start + 1;
54       }
55       return r;
56    }
57 }
```

section_6/ExpressionCalculator.java

```
1  import java.util.Scanner;
2
3  /**
4     This program calculates the value of an expression
5     consisting of numbers, arithmetic operators, and parentheses.
6  */
7  public class ExpressionCalculator
8  {
9     public static void main(String[] args)
10    {
11       Scanner in = new Scanner(System.in);
12       System.out.print("Enter an expression: ");
13       String input = in.nextLine();
14       Evaluator e = new Evaluator(input);
```

```
15         int value = e.getExpressionValue();
16         System.out.println(input + "=" + value);
17      }
18   }
```

Program Run

```
Enter an expression: 3+4*5
3+4*5=23
```

20. What is the difference between a term and a factor? Why do we need both concepts?
21. Why does the expression evaluator use mutual recursion?
22. What happens if you try to evaluate the illegal expression `3+4*)5`? Specifically, which method throws an exception?

Practice It Now you can try these exercises at the end of the chapter: R13.11, P13.16.

13.7 Backtracking

Backtracking examines partial solutions, abandoning unsuitable ones and returning to consider other candidates.

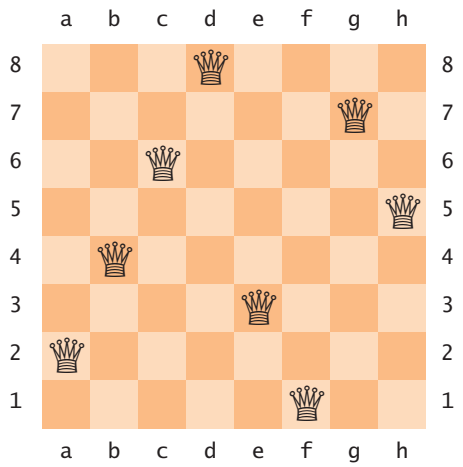
Backtracking is a problem solving technique that builds up partial solutions that get increasingly closer to the goal. If a partial solution cannot be completed, one abandons it and returns to examining the other candidates.

Backtracking can be used to solve crossword puzzles, escape from mazes, or find solutions to systems that are constrained by rules. In order to employ backtracking for a particular problem, we need two characteristic properties:

1. A procedure to examine a partial solution and determine whether to
   - Accept it as an actual solution.
   - Abandon it (either because it violates some rules or because it is clear that it can never lead to a valid solution).
   - Continue extending it.
2. A procedure to extend a partial solution, generating one or more solutions that come closer to the goal.

In a backtracking algorithm, one explores all paths towards a solution. When one path is a dead end, one needs to backtrack and try another choice.

Figure 4
A Solution to the Eight Queens Problem



Backtracking can then be expressed with the following recursive algorithm:

```
Solve(partialSolution)
   Examine(partialSolution).
   If accepted
      Add partialSolution to the list of solutions.
   Else if not abandoned
      For each p in extend(partialSolution)
         Solve(p).
```

Of course, the processes of examining and extending a partial solution depend on the nature of the problem.

As an example, we will develop a program that finds all solutions to the eight queens problem: the task of positioning eight queens on a chess board so that none of them attacks another according to the rules of chess. In other words, there are no two queens on the same row, column, or diagonal. Figure 4 shows a solution.

In this problem, it is easy to examine a partial solution. If two queens attack another, reject it. Otherwise, if it has eight queens, accept it. Otherwise, continue.

It is also easy to extend a partial solution. Simply add another queen on an empty square.

However, in the interest of efficiency, we will be a bit more systematic about the extension process. We will place the first queen in row 1, the next queen in row 2, and so on.

We provide a class `PartialSolution` that collects the queens in a partial solution, and that has methods to examine and extend the solution:

```
public class PartialSolution
{
   private Queen[] queens;

   public int examine() { . . . }
   public PartialSolution[] extend() { . . . }
}
```

The `examine` method simply checks whether two queens attack each other:

```
public int examine()
{
```

```
      for (int i = 0; i < queens.length; i++)
      {
         for (int j = i + 1; j < queens.length; j++)
         {
            if (queens[i].attacks(queens[j])) { return ABANDON; }
         }
      }
      if (queens.length == NQUEENS) { return ACCEPT; }
      else { return CONTINUE; }
   }
```

The `extend` method takes a given solution and makes eight copies of it. Each copy gets a new queen in a different column.

```
   public PartialSolution[] extend()
   {
      // Generate a new solution for each column
      PartialSolution[] result = new PartialSolution[NQUEENS];
      for (int i = 0; i < result.length; i++)
      {
         int size = queens.length;

         // The new solution has one more row than this one
         result[i] = new PartialSolution(size + 1);

         // Copy this solution into the new one
         for (int j = 0; j < size; j++)
         {
            result[i].queens[j] = queens[j];
         }

         // Append the new queen into the ith column
         result[i].queens[size] = new Queen(size, i);
      }
      return result;
   }
```

You will find the `Queen` class at the end of the section. The only challenge is to determine when two queens attack each other diagonally. Here is an easy way of checking that. Compute the slope and check whether it is ±1. This condition can be simplified as follows:

$$\begin{aligned}(\text{row}_2 - \text{row}_1)/(\text{column}_2 - \text{column}_1) &= \pm 1\\ \text{row}_2 - \text{row}_1 &= \pm(\text{column}_2 - \text{column}_1)\\ |\text{row}_2 - \text{row}_1| &= |\text{column}_2 - \text{column}_1|\end{aligned}$$

Have a close look at the `solve` method in the `EightQueens` class on page W617. The method is a straightforward translation of the pseudocode for backtracking. Note how there is nothing specific about the eight queens problem in this method—it works for any partial solution with an `examine` and `extend` method (see Exercise P13.19).

Figure 5 shows the `solve` method in action for a four queens problem. Starting from a blank board, there are four partial solutions with a queen in row 1 (1). When the queen is in column 1, there are four partial solutions with a queen in row 2 (2). Two of them are immediately abandoned immediately. The other two lead to partial solutions with three queens (3) and (4), all but one of which are abandoned. One partial solution is extended to four queens, but all of those are abandoned as well (5).

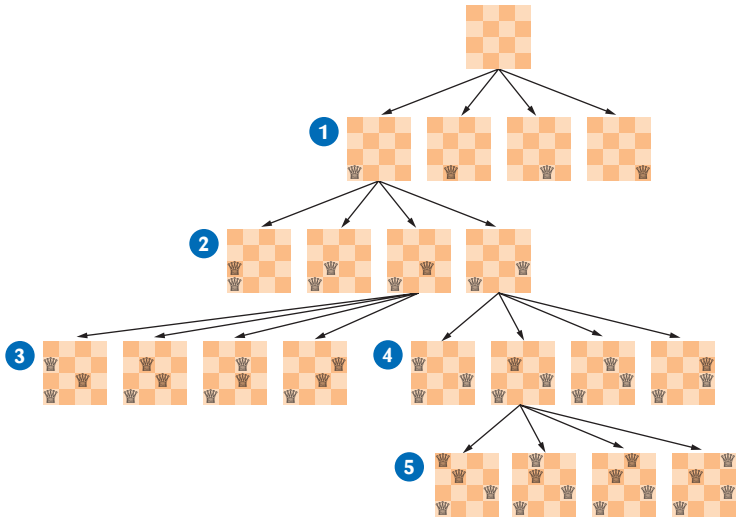


Figure 5 Backtracking in the Four Queens Problem

Then the algorithm backtracks, giving up on a queen in position a1, instead extending the solution with the queen in position b1 (not shown).

When you run the program, it lists 92 solutions, including the one in Figure 4. Exercise P13.21 asks you to remove those that are rotations or reflections of another.

section_7/PartialSolution.java

```
1  /**
2     A partial solution to the eight queens puzzle.
3  */
4  public class PartialSolution
5  {
6     private Queen[] queens;
7     private static final int NQUEENS = 8;
8
9     public static final int ACCEPT = 1;
10    public static final int ABANDON = 2;
11    public static final int CONTINUE = 3;
12
13    /**
14       Constructs a partial solution of a given size.
15       @param size the size
16    */
17    public PartialSolution(int size)
18    {
19       queens = new Queen[size];
20    }
21
22    /**
23       Examines a partial solution.
24       @return one of ACCEPT, ABANDON, CONTINUE
25    */
26    public int examine()
27    {
```

```
28          for (int i = 0; i < queens.length; i++)
29          {
30             for (int j = i + 1; j < queens.length; j++)
31             {
32                if (queens[i].attacks(queens[j])) { return ABANDON; }
33             }
34          }
35          if (queens.length == NQUEENS) { return ACCEPT; }
36          else { return CONTINUE; }
37       }
38
39       /**
40          Yields all extensions of this partial solution.
41          @return an array of partial solutions that extend this solution.
42       */
43       public PartialSolution[] extend()
44       {
45          // Generate a new solution for each column
46          PartialSolution[] result = new PartialSolution[NQUEENS];
47          for (int i = 0; i < result.length; i++)
48          {
49             int size = queens.length;
50
51             // The new solution has one more row than this one
52             result[i] = new PartialSolution(size + 1);
53
54             // Copy this solution into the new one
55             for (int j = 0; j < size; j++)
56             {
57                result[i].queens[j] = queens[j];
58             }
59
60             // Append the new queen into the ith column
61             result[i].queens[size] = new Queen(size, i);
62          }
63          return result;
64       }
65
66       public String toString() { return Arrays.toString(queens); }
67    }
```

section_7/Queen.java

```
1  /**
2     A queen in the eight queens problem.
3  */
4  public class Queen
5  {
6     private int row;
7     private int column;
8
9     /**
10       Constructs a queen at a given position.
11       @param r the row
12       @param c the column
13    */
14    public Queen(int r, int c)
15    {
```

```
16          row = r;
17          column = c;
18       }
19
20       /**
21          Checks whether this queen attacks another.
22          @param other the other queen
23          @return true if this and the other queen are in the same
24          row, column, or diagonal
25       */
26       public boolean attacks(Queen other)
27       {
28          return row == other.row
29             || column == other.column
30             || Math.abs(row - other.row) == Math.abs(column - other.column);
31       }
32
33       public String toString()
34       {
35          return "" + "abcdefgh".charAt(column) + (row + 1) ;
36       }
37    }
```

section_7/EightQueens.java

```
1  import java.util.Arrays;
2
3  /**
4     This class solves the eight queens problem using backtracking.
5  */
6  public class EightQueens
7  {
8     public static void main(String[] args)
9     {
10       solve(new PartialSolution(0));
11    }
12
13    /**
14       Prints all solutions to the problem that can be extended from
15       a given partial solution.
16       @param sol the partial solution
17    */
18    public static void solve(PartialSolution sol)
19    {
20       int exam = sol.examine();
21       if (exam == PartialSolution.ACCEPT)
22       {
23          System.out.println(sol);
24       }
25       else if (exam != PartialSolution.ABANDON)
26       {
27          for (PartialSolution p : sol.extend())
28          {
29             solve(p);
30          }
31       }
32    }
33 }
```

Program Run

```
[a1, e2, h3, f4, c5, g6, b7, d8]
[a1, f2, h3, c4, g5, d6, b7, e8]
[a1, g2, d3, f4, h5, b6, e7, c8]
 . . .
[f1, a2, e3, b4, h5, c6, g7, d8]
 . . .
[h1, c2, a3, f4, b5, e6, g7, d8]
[h1, d2, a3, c4, f5, b6, g7, e8]
```

(92 solutions)

SELF CHECK

23. Why does `j` begin at `i + 1` in the `examine` method?

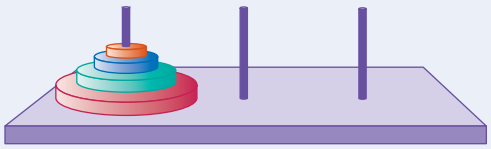
24. Continue tracing the four queens problem as shown in Figure 5. How many solutions are there with the first queen in position `a2`?

25. How many solutions are there altogether for the four queens problem?

Practice It Now you can try these exercises at the end of the chapter: P13.19, P13.23, P13.24.

WORKED EXAMPLE 13.2 Towers of Hanoi

No discussion of recursion would be complete without the "Towers of Hanoi". In this Worked Example, we solve the classic puzzle with an elegant recursive solution.



CHAPTER SUMMARY

Understand the control flow in a recursive computation.

- A recursive computation solves a problem by using the solution to the same problem with simpler inputs.
- For a recursion to terminate, there must be special cases for the simplest values.

Design a recursive solution to a problem.

Identify recursive helper methods for solving a problem.

- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

Available online in WileyPLUS and at `www.wiley.com/college/horstmann`.

Contrast the efficiency of recursive and non-recursive algorithms.

- Occasionally, a recursive solution runs much slower than its iterative counterpart. However, in most cases, the recursive solution is only slightly slower.
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

Review a complex recursion example that cannot be solved with a simple loop.

- The permutations of a string can be obtained more naturally through recursion than with a loop.

Recognize the phenomenon of mutual recursion in an expression evaluator.

- In a mutual recursion, a set of cooperating methods calls each other repeatedly.

Use backtracking to solve problems that require trying out multiple paths.

- Backtracking examines partial solutions, abandoning unsuitable ones and returning to consider other candidates.

REVIEW EXERCISES

■ **R13.1** Define the terms

a. Recursion

b. Iteration

c. Infinite recursion

d. Recursive helper method

■■ **R13.2** Outline, but do not implement, a recursive solution for finding the smallest value in an array.

■■ **R13.3** Outline, but do not implement, a recursive solution for sorting an array of numbers. *Hint:* First find the smallest value in the array.

■■ **R13.4** Outline, but do not implement, a recursive solution for generating all subsets of the set $\{1, 2, \ldots, n\}$.

■■■ **R13.5** Exercise P13.15 shows an iterative way of generating all permutations of the sequence $(0, 1, \ldots, n - 1)$. Explain why the algorithm produces the correct result.

■ **R13.6** Write a recursive definition of x^n, where $n \geq 0$, similar to the recursive definition of the Fibonacci numbers. *Hint:* How do you compute x^n from x^{n-1}? How does the recursion terminate?

■■ **R13.7** Improve upon Exercise R13.6 by computing x^n as $(x^{n/2})^2$ if n is even. Why is this approach significantly faster? *Hint:* Compute x^{1023} and x^{1024} both ways.

■ **R13.8** Write a recursive definition of $n! = 1 \times 2 \times \ldots \times n$, similar to the recursive definition of the Fibonacci numbers.

■■ **R13.9** Find out how often the recursive version of `fib` calls itself. Keep a static variable `fibCount` and increment it once in every call to `fib`. What is the relationship between `fib(n)` and `fibCount`?

■■■ **R13.10** Let moves(n) be the number of moves required to solve the Towers of Hanoi problem (see Worked Example 13.2). Find a formula that expresses moves(n) in terms of moves($n - 1$). Then show that moves(n) = $2^n - 1$.

■■ **R13.11** Trace the expression evaluator program from Section 13.6 with inputs `3 - 4 + 5`, `3 - (4 + 5)`, `(3 - 4) * 5`, and `3 * 4 + 5 * 6`.

PROGRAMMING EXERCISES

■ **P13.1** Given a class `Rectangle` with instance variables `width` and `height`, provide a recursive `getArea` method. Construct a rectangle whose width is one less than the original and call its `getArea` method.

■■ **P13.2** Given a class `Square` with instance variable `width`, provide a recursive `getArea` method. Construct a square whose width is one less than the original and call its `getArea` method.

■ **P13.3** Write a recursive method `String reverse(String text)` that reverses a string. For example, `reverse("Hello!")` returns the string `"!olleH"`. Implement a recursive solution by removing the first character, reversing the remaining text, and combining the two.

■■ **P13.4** Redo Exercise P13.3 with a recursive helper method that reverses a substring of the message text.

■ **P13.5** Implement the `reverse` method of Exercise P13.3 as an iteration.

■■ **P13.6** Use recursion to implement a method

```
public static boolean find(String text, String str)
```

that tests whether a given text contains a string. For example, `find("Mississippi", "sip")` returns true.

Hint: If the text starts with the string you want to match, then you are done. If not, consider the text that you obtain by removing the first character.

■■ **P13.7** Use recursion to implement a method

```
public static int indexOf(String text, String str)
```

that returns the starting position of the first substring of the text that matches `str`. Return –1 if `str` is not a substring of the text.

For example, `s.indexOf("Mississippi", "sip")` returns 6.

Hint: This is a bit trickier than Exercise P13.6, because you must keep track of how far the match is from the beginning of the text. Make that value a parameter variable of a helper method.

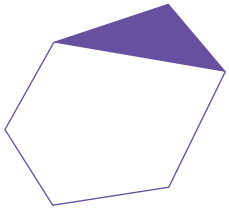
■ **P13.8** Using recursion, find the largest element in an array.

Hint: Find the largest element in the subset containing all but the last element. Then compare that maximum to the value of the last element.

■ **P13.9** Using recursion, compute the sum of all values in an array.

■■ **P13.10** Using recursion, compute the area of a polygon. Cut off a triangle and use the fact that a triangle with corners (x_1, y_1), (x_2, y_2), (x_3, y_3) has area

$$\frac{\left| x_1 y_2 + x_2 y_3 + x_3 y_1 - y_1 x_2 - y_2 x_3 - y_3 x_1 \right|}{2}$$



■■ **P13.11** The following method was known to the ancient Greeks for computing square roots. Given a value $x > 0$ and a guess g for the square root, a better guess is $(x + g/x) / 2$. Write a recursive helper method `public static squareRootGuess(double x, double g)`. If g^2 is approximately equal to x, return g, otherwise, return `squareRootGuess` with the better guess. Then write a method `public static squareRoot(double x)` that uses the helper method.

■■■ **P13.12** Implement a `SubstringGenerator` that generates all substrings of a string. For example, the substrings of the string `"rum"` are the seven strings

```
"r", "ru", "rum", "u", "um", "m", ""
```

Hint: First enumerate all substrings that start with the first character. There are n of them if the string has length n. Then enumerate the substrings of the string that you obtain by removing the first character.

■■■ **P13.13** Implement a `SubsetGenerator` that generates all subsets of the characters of a string. For example, the subsets of the characters of the string `"rum"` are the eight strings

```
"rum", "ru", "rm", "r", "um", "u", "m", ""
```

Note that the subsets don't have to be substrings—for example, `"rm"` isn't a substring of `"rum"`.

■■■ **P13.14** In this exercise, you will change the `permutations` method of Section 13.4 (which computed all permutations at once) to a `PermutationIterator` (which computes them one at a time).

```
public class PermutationIterator
{
   public PermutationIterator(String s) { . . . }
   public String nextPermutation() { . . . }
   public boolean hasMorePermutations() { . . . }
}
```

Here is how you would print out all permutations of the string `"eat"`:

```
PermutationIterator iter = new PermutationIterator("eat");
while (iter.hasMorePermutations())
{
   System.out.println(iter.nextPermutation());
}
```

Now we need a way to iterate through the permutations recursively. Consider the string `"eat"`. As before, we'll generate all permutations that start with the letter `'e'`, then those that start with `'a'`, and finally those that start with `'t'`. How do we generate the permutations that start with `'e'`? Make another `PermutationIterator` object (called `tailIterator`) that iterates through the permutations of the substring `"at"`. In the `nextPermutation` method, simply ask `tailIterator` what *its* next permutation is, and then add the `'e'` at the front. However, there is one special case. When the tail

generator runs out of permutations, all permutations that start with the current letter have been enumerated. Then

- Increment the current position.
- Compute the tail string that contains all letters except for the current one.
- Make a new permutation iterator for the tail string.

You are done when the current position has reached the end of the string.

■■■ **P13.15** The following class generates all permutations of the numbers 0, 1, 2, . . ., $n - 1$, without using recursion.

```
public class NumberPermutationIterator
{
   private int[] a;

   public NumberPermutationIterator(int n)
   {
      a = new int[n];
      done = false;
      for (int i = 0; i < n; i++) { a[i] = i; }
   }

   public int[] nextPermutation()
   {
      if (a.length <= 1) { return a; }

      for (int i = a.length - 1; i > 0; i--)
      {
         if (a[i - 1] < a[i])
         {
            int j = a.length - 1;
            while (a[i - 1] > a[j]) { j--; }
            swap(i - 1, j);
            reverse(i, a.length - 1);
            return a;
         }
      }
      return a;
   }

   public boolean hasMorePermutations()
   {
      if (a.length <= 1) { return false; }
      for (int i = a.length - 1; i > 0; i--)
      {
         if (a[i - 1] < a[i]) { return true; }
      }
      return false;
   }

   public void swap(int i, int j)
   {
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }

   public void reverse(int i, int j)
   {
      while (i < j) { swap(i, j); i++; j--; }
```

```
   }
}
```

The algorithm uses the fact that the set to be permuted consists of distinct numbers. Thus, you cannot use the same algorithm to compute the permutations of the characters in a string. You can, however, use this class to get all permutations of the character positions and then compute a string whose `i`th character is `word.charAt(a[i])`. Use this approach to reimplement the `PermutationIterator` of Exercise P13.14 without recursion.

■■ **P13.16** Extend the expression evaluator in Section 13.6 so that it can handle the `%` operator as well as a "raise to a power" operator `^`. For example, `2 ^ 3` should evaluate to 8. As in mathematics, raising to a power should bind more strongly than multiplication: `5 * 2 ^ 3` is 40.

■■■ **P13.17** Implement an iterator that produces the moves for the Towers of Hanoi puzzle described in Worked Example 13.2. Provide methods `hasMoreMoves` and `nextMove`. The `nextMove` method should yield a string describing the next move. For example, the following code prints all moves needed to move five disks from peg 1 to peg 3:

```
DiskMover mover = new DiskMover(5, 1, 3);
while (mover.hasMoreMoves())
{
   System.out.println(mover.nextMove());
}
```

Hint: A disk mover that moves a single disk from one peg to another simply has a `nextMove` method that returns a string

```
Move disk from peg source to target
```

A disk mover with more than one disk to move must work harder. It needs another `DiskMover` to help it move the first $d - 1$ disks. The `nextMove` asks that disk mover for its next move until it is done. Then the `nextMove` method issues a command to move the dth disk. Finally, it constructs another disk mover that generates the remaining moves.

It helps to keep track of the state of the disk mover:

- `BEFORE_LARGEST`: A helper mover moves the smaller pile to the other peg.
- `LARGEST`: Move the largest disk from the source to the destination.
- `AFTER_LARGEST`: The helper mover moves the smaller pile from the other peg to the target.
- `DONE`: All moves are done.

■■■ **P13.18** *Escaping a Maze*. You are currently located inside a maze. The walls of the maze are indicated by asterisks (`*`).

```
* *******
*     * *
* ***** *
* * *   *
* * *** *
*   *   *
*** * * *
*     * *
******* *
```

Use the following recursive approach to check whether you can escape from the maze: If you are at an exit, return `true`. Recursively check whether you can escape

from one of the empty neighboring locations without visiting the current location. This method merely tests whether there is a path out of the maze. Extra credit if you can print out a path that leads to an exit.

■■ **P13.19** The backtracking algorithm will work for any problem whose partial solutions can be examined and extended. Provide a `PartialSolution` interface type with methods `examine` and `extend`, a `solve` method that works with this interface type, and a class `EightQueensPartialSolution` that implements the interface.

■■ **P13.20** Using the `PartialSolution` interface and `solve` method from Exercise P13.19, provide a class `MazePartialSolution` for solving the maze escape problem of Exercise P13.18.

■■■ **P13.21** Refine the program for solving the eight queens problem so that rotations and reflections of previously displayed solutions are not shown. Your program should display twelve unique solutions.

■■■ **P13.22** Refine the program for solving the eight queens problem so that the solutions are written to an HTML file, using tables with black and white background for the board and the Unicode character ♕ `'\u2655'` for the white queen.

■■ **P13.23** Generalize the program for solving the eight queens problem to the *n* queens problem. Your program should prompt for the value of *n* and display the solutions.

■■ **P13.24** Using backtracking, write a program that solves summation puzzles in which each letter should be replaced by a digit, such as

```
send + more = money
```

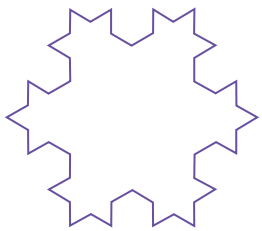
Other examples are `base + ball = games` and `kyoto + osaka = tokyo`.

■■ **P13.25** The recursive computation of Fibonacci numbers can be speeded up significantly by keeping track of the values that have already been computed. Provide an implementation of the `fib` method that uses this strategy. Whenever you return a new value, also store it in an auxiliary array. However, before embarking on a computation, consult the array to find whether the result has already been computed. Compare the running time of your improved implementation with that of the original recursive implementation and the loop implementation.

■■■ **Graphics P13.26** *The Koch Snowflake.* A snowflake-like shape is recursively defined as follows. Start with an equilateral triangle:

Next, increase the size by a factor of three and replace each straight line with four line segments:

Repeat the process:



Write a program that draws the iterations of the snowflake shape. Supply a button that, when clicked, produces the next iteration.

ANSWERS TO SELF-CHECK QUESTIONS

1. Suppose we omit the statement. When computing the area of a triangle with width 1, we compute the area of the triangle with width 0 as 0, and then add 1, to arrive at the correct area.
2. You would compute the smaller area recursively, then return

   ```
   smallerArea + width + width - 1.
   ```

   ```
   [][][][]
   [][][][]
   [][][][]
   [][][][]
   ```

   Of course, it would be simpler to compute the area simply as `width * width`. The results are identical because

   $$1 + 0 + 2 + 1 + 3 + 2 + \cdots + n + n - 1 = \frac{n(n+1)}{2} + \frac{(n-1)n}{2} = n^2$$

3. There is no provision for stopping the recursion. When a number < 10 isn't 8, then the method should return `false` and stop.
4. ```
   public static int pow2(int n)
   {
      if (n <= 0) { return 1; } // 2^0 is 1
      else { return 2 * pow2(n - 1); }
   }
   ```
5. `mystery(4)` calls `mystery(3)`
   `mystery(3)` calls `mystery(2)`
   `mystery(2)` calls `mystery(1)`
   `mystery(1)` calls `mystery(0)`
   `mystery(0)` returns `0`.
   `mystery(1)` returns `0 + 1 * 1 = 1`
   `mystery(2)` returns `1 + 2 * 2 = 5`
   `mystery(3)` returns `5 + 3 * 3 = 14`
   `mystery(4)` returns `14 + 4 * 4 = 30`
6. In this problem, *any* decomposition will work fine. We can remove the first or last character and then remove punctuation marks from the remainder. Or we can break the string in two substrings, and remove punctuation marks from each.
7. If the last character is a punctuation mark, then you simply return the shorter string with punctuation marks removed. Otherwise, you reattach the last character to that result and return it.
8. The simplest input is the empty string. It contains no punctuation marks, so you simply return it.
9. ```
   If str is empty, return str.
   last = last letter in str
   simplerResult = removePunctuation(
         str with last letter removed)
   If (last is a punctuation mark)
      Return simplerResult.
   Else
      Return simplerResult + last.
   ```
10. No—the second one could be given a different name such as `substringIsPalindrome`.
11. When `start >= end`, that is, when the investigated string is either empty or has length 1.
12. A `sumHelper(int[] a, int start, int size)`. The method calls `sumHelper(a, start + 1, size)`.
13. Call `sum(a, size - 1)` and add the *last* element, `a[size - 1]`.
14. The loop is slightly faster. It is even faster to simply compute `width * (width + 1) / 2`.
15. No, the recursive solution is about as efficient as the iterative approach. Both require $n - 1$ multiplications to compute $n!$.
16. The recursive algorithm performs about as well as the loop. Unlike the recursive Fibonacci algorithm, this algorithm doesn't call itself again on the same input. For example, the sum of the array 1 4 9 16 25 36 49 64 is computed as the sum of 1 4 9 16 and 25 36 49 64, then as the sums of 1 4, 9 16, 25 36, and 49 64, which can be computed directly.
17. They are `b` followed by the six permutations of `eat`, `e` followed by the six permutations of

`bat`, `a` followed by the six permutations of `bet`, and `t` followed by the six permutations of `bea`.

18. Simply change `if (word.length() == 0)` to `if (word.length() <= 1)`, because a word with a single letter is also its sole permutation.

19. An iterative solution would have a loop whose body computes the next permutation from the previous ones. But there is no obvious mechanism for getting the next permutation. For example, if you already found permutations `eat`, `eta`, and `aet`, it is not clear how you use that information to get the next permutation. Actually, there is an ingenious mechanism for doing just that, but it is far from obvious—see Exercise P13.15.

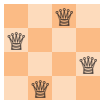
20. Factors are combined by multiplicative operators (`*` and `/`); terms are combined by additive operators (`+`, `-`). We need both so that multiplication can bind more strongly than addition.

21. To handle parenthesized expressions, such as `2+3*(4+5)`. The subexpression `4+5` is handled by a recursive call to `getExpressionValue`.

22. The `Integer.parseInt` call in `getFactorValue` throws an exception when it is given the string `")"`.

23. We want to check whether any `queen[i]` attacks any `queen[j]`, but attacking is symmetric. That is, we can choose to compare only those for which `i < j` (or, alternatively, those for which `i > j`). We don't want to call the `attacks` method when `i` equals `j`; it would return true.

24. One solution:



25. Two solutions: The one from Self Check 24, and its mirror image.