Chapter 9

Recursion

In this chapter we introduce you to the very powerful programming technique of *recursion*. So far the only ways we have seen to perform repetitive tasks have involved loops. While *iterative* constructs are very important for programming, recursion is in many ways a more powerful technique for solving certain types of complex problems.

A key idea in designing a program to solve a complex problem is to break the complex problem into simpler problems, solve the simpler problems, and then assemble the final answer from these simpler pieces. An important reason for writing methods is to provide conceptually simple operations (even if their implementations might be quite complex) to be used in solving more complex problems. Examples of this that we have seen so far in this book include ...

The key idea involved in recursive algorithms is to fashion a solution to a complex problem by solving one or more simpler versions of the *same* problem, and then using those to complete the solution to the more complex version of the problem.

Here is a simple example. Suppose we are given a large stack of student papers and are told to arrange them alphabetically by the student's name. One strategy might be to first separate the papers into two piles, where the first pile has student names from the first half of the alphabet and the second has student names from the second half. If we (or better yet, an assistant) sort each of these smaller piles separately, then we can arrange all of the papers alphabetically by taking the stack with the first half of the alphabet and placing it on the stack with the second half of the alphabet.

This seems like a perfectly sensible way of arranging the papers, but it leaves open one critical problem. After separating the papers into two piles, how do we sort each of those? While we could come up with some alternative procedure to sort these smaller stacks, the most straightforward way of proceeding is to sort them using the same algorithm. That is, separate those into two piles, where the first pile has the names from the first half of those names in the pile (e.g., the first quarter of the alphabet when we are dealing with the first of our piles) and the second has the rest (e.g., the second quarter of the alphabet).

In other words, when we get someone to sort the smaller piles, we get them to use the same algorithm or set of instructions as we used with the bigger piles. In particular, they divide their piles in half alphabetically according to the range of names in their piles, sort each of those (presumably by each getting two more assistants to help each of them) and then put the piles on top of each other in the right order. A method in Java written in this style that results in other calls to the same set of instructions (i.e, the same method) is said to be recursive, and the call to the same method is said to be a recursive call.

We are still left with one problem. At some point it will no longer be possible to divide the

stacks in half because there will be at most one element left in the stack. Luckily that is an easy case to handle. Just leave the stack (of either 0 or 1 papers) there and declare it to be sorted – after all, all the paper(s) in the (trivial) stack are in alphabetical order. As we will see later, it is very important that every recursive method have a way of stopping – in particular, that it eventually gets to a case where there is no recursive call.

We can write our algorithm quite succinctly:

```
To arrange papers in order from pile:

if there is more than 1 paper to be arranged then
   place the papers into two piles,
      one, pile1, with all papers from the lowest names in the range of
              names in the stack
      the other, pile2, with all the papers from the highest names

   arrange papers in order from pile1
   arrange papers in order from pile2
   finish by placing the first stack on top of the second stack
else (there is at most 1 paper left)
   don't do anything -- the papers are already in the proper order.
```

The thing that makes this algorithm recursive is that in the middle of the algorithm it indicates that the same set of instructions should be followed on other inputs; in this case, the two smaller piles of papers. It is worth noting that we made no commitment as to who should perform these operations on the smaller piles of papers. It is convenient to think that there are assistants available who would be handed the smaller piles of papers and a copy of the above instructions and who could perform the task. Of course if such assistants are not available, then whoever started executing the task could also do the similar tasks on the smaller piles as long as they remembered where they were in the original instructions when they finished the smaller task.

To test your understanding of this algorithm, get a stack of 8 notecards and write the letters of the alphabet from "A" to "H" on them. Shuffle the cards and now use the algorithm above to arrange them in order. Have assistants do the arranging of cards in smaller stacks. If assistants are not available you'll have to arrange the smaller stacks in order by following the instructions yourself.

9.1 Graphic problems using recursion

We find it convenient to illustrate the ideas of recursion by starting with some graphical programs. In this section we will illustrate recursion with three programs that draw and manipulate nested rectangles, a fractal image, and even broccoli.

9.1.1 Nested rectangles

Our first recursive picture will be quite simple. We will design a class that will draw nested rectangles of the sort shown in Figure 9.1. Eventually we would like to make these have as many capabilities as the `FramedRect` or `FilledRect` objects from the objectdraw library. For example we would them to have `move` and `moveTo` methods. For now, however, we'll just be content in being able to draw them.

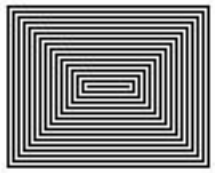


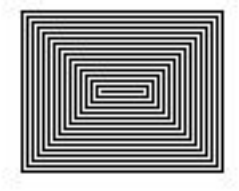
Figure 9.1: Nested rectangles

Because you are used to thinking in terms of loops, you can probably figure out a nice iterative way of drawing these using a `while` loop. Here is an example of a constructor for the class `NestedRect` that uses a `while` loop.

```
// Draw nested rectangles at the given x and y coordinates and
// with the given height and width (both of which must be
// non-negative)
public NestedRect(double x, double y, double width,
                  double height, DrawingCanvas canvas) {
    new FramedRect( x, y, width, height, canvas );
    while ( width >= 4 && height >= 4) {
        width = width - 4;
        height = height - 4;
        x = x + 2;
        y = y + 2;
        new FramedRect( x, y, width, height, canvas );
    }
}
```

The idea of the constructor is quite simple. First we draw a framed rectangle with the dimensions passed in to the constructor. Then, as long as the width and height of the rectangle you have drawn are both greater than or equal to four, adjust the width, height, x, and y coordinates in order to draw a new smaller rectangle centered inside the one just drawn. We presume the width and height sent in as parameters are always non-negative, so we always draw the outer rectangle – otherwise we might not see anything in the picture.

To see how to do this recursively, we need to change our point of view of how we see nested rectangles. Instead of drawing them as a series of `FramedRect`s, we instead draw an outer `FramedRect` an then, if there is enough space, draw a smaller nested rectangle object inside. See the picture below in which the outer framed rectangle is drawn lighter than the collection of nested rectangles inside.



That is, when we look at pictures as being recursive, we try to find a smaller picture of what we are looking at as part of the larger picture. Normally we look for the simplest description that includes the recursive description. In this case, that is a description with a single outer framed rectangle and a smaller NestedRectangle centered inside. Below is a recursive constructor for `NestedRect` that corresponds to this recursive description:

```
// Draw a nested rectangle at the given x and y coordinates and
// with the given height and width (both of which must be positive)
public NestedRect(double x, double y, double width,
                  double height, DrawingCanvas canvas) {
    new FramedRect( x, y, width, height, canvas );
    if ( width >= 4 && height >= 4) {
        new NestedRect(x+2, y+2, width-4, height-4, canvas);
    }
}
```

In this version of the constructor, we first draw the framed rectangle (just as before), but then if the width and height are both at least four, we go ahead and draw a new nested rectangle in a position 2 to the right and down from the first one and with width and height both reduced by 4 pixels.

The new constructor is a bit shorter than the iterative version, but that is not really very important. What is important is that we have a simple description of a nested rectangle as an outer rectangle with slightly smaller nested rectangle centered inside. This is actually easier to grasp than a description that forces the programmer to think about executing a while loop some possibly large number of times.

In the iterative version of the constructor for a nested rectangle, the `while` loop terminated when either the width or height was reduced to a value below 4. We need a similar method to stop a recursive algorithm. In this case, when the width or height passed in as parameters get to be less than 4, only the outer rectangle is drawn. If we initially start with positive values of width and height, then each call of the constructor for `NestedRect` inside the if statement will use smaller and smaller values for width and height – to be exact, each is 4 units smaller than the previous. Eventually either the length or width will get a value smaller than 4, the condition in the `if` statement will be false, and no more nested rectangles will be drawn.

Let us see what happens when we actually call the constructor. Suppose we evaluate

```
new NestedRect(10,10,7,9,canvas);
```

This will result in the execution of

```
new FramedRect(10,10,7,9,canvas);
```

which draws a 7 by 9 rectangle with upper left corner at (10,10). Next the `if` statement is evaluated. Because `width` is 7 and `height` is 9, the condition is true. Thus we execute:

```
new NestedRect(12,12,3,5,canvas);
```

where the new parameters are calculated according to the expressions given in the constructor call. We start executing at the beginning of the constructor with the new parameters, and hence evaluate

```
new NestedRect(12,12,3,5,canvas);
```

which draws a 3 by 5 rectangle with upper left corner at (12,12). But now when we evaluate the condition for the `if` statement, it will be false because the new width, 3, is not `>= 4`. As a result the constructor terminates.

In summary, the original call of the constructor ends up drawing two rectangles before concluding. This is exactly what the iterative constructor would do with the same input, but the style of writing the recursive version is quite different.

Let's go on now and think about the next phase of writing a `NestedRect` class. Suppose we want to write a method that moves the nested rectangle a given amount on the canvas. In order to do this we will of course need to associate names with the parts of the nested rectangle.

Because we need to be able to move every `FramedRect` in a `NestedRect` object, it may seem like we need a new name for each `FramedRect`. This would be a problem because we do not know in advance how many rectangles will be drawn. However, if we think recursively then we will see that only two names are necessary. To move a `NestedRect` object, we simply move the outer rectangle and then move the inner `NestedRect`. See Figure 9.2 for the definition of this class with `moveTo` and `removeFromCanvas` methods.

While the original recursive constructor used no instance variables, the new version of the constructor uses two instance variables. One, `outerRect`, is a `FramedRect` variable that names the outer rectangle. The other, `rest`, is a `NestedRect` variable that names the smaller nested rectangle contained inside `outerRect`. We say that objects of type `NestedRect` are *recursive* data structures because each of these objects has an instance variable, in this case named `rest`, of the same type as the object itself.

If the constructor is provided with either a width or height value that is less than 4 then `rest` will not be assigned a value. Thus its value will be `null`, as all object instance variables are given a value of `null` by default. We will use this information to provide a way of stopping the recursion in methods.

Thus in the `moveTo` method, we will always move `outerRect`, but we only move `rest` if the value of `rest` is different from `null`. That is, we send a `move` message to `rest` only if it actually contains a smaller `NestedRect` to move. Of course, it is also a good thing that we don't try to send a message to `rest` when it is `null`, because we would get one of those dreaded `null` pointer error messages. However, it is also important because we need the method to eventually terminate.

Here are the details of what happens inside `moveTo`. First we move the `outerRect` to the desired position. If there are any nested rectangles inside (i.e., if `rest != null`), then we move the `rest` to a position 2 units to the right and below the position we moved the `outerRect`. This ensures that the `rest` stays centered inside `outerRect`.

Other methods for `NestedRect` have similar structure. We have shown `removeFromCanvas` as an example. It operates by removing `outerRect` from the canvas, and then, if `rest` is non-`null`, removing `rest` from the canvas.

```
public class NestedRect {
    FramedRect outerRect;    // outermost rectangle in picture
    NestedRect rest;         // inside nested rect

    // Draw nested rectangles at the given x and y coordinates and
    // with the given height and width (both of which must be
    // non-negative)
    public NestedRect(double x, double y, double width,
                      double height, DrawingCanvas canvas) {
        outerRect = new FramedRect( x, y, width, height, canvas );
        if ( width >= 4 && height >= 4) {
            rest = new NestedRect(x+2, y+2, width-4,
                                  height-4, canvas);
        }
    }

    // Move nested rect to (x,y)
    public void moveTo(double x, double y) {
        outerRect.moveTo(x,y);
        if (rest != null) {
            rest.moveTo(x+2, y+2);
        }
    }

    // Remove the nested rect from the canvas
    public void removeFromCanvas() {
        outerRect.removeFromCanvas();
        if (rest != null) {
            rest.removeFromCanvas();
        }
    }
}
```

Figure 9.2: Recursive version of `NestedRect` class.

It should be clear that the methods of `NestedRect` have a structure which is based on the recursive structure of objects from the class. That is, they do what is necessary to `outerRect` and then, if `rest != null`, they call the method recursively on `rest` (though possibly with slightly different parameters).

Of course, not all methods have that shape. For example, if we were to add a `contains` method for `NestedRect`, we would simply check to see if the `Location` parameter were contained in `outerRect`, as any point contained in the nested rectangles is contained in `outerRect`. However, most methods will follow the recursive structure of the objects.

Exercise 9.1.1 *1. Add methods `setColor` and `move` to the recursive version of `NestedRect`. If we were to also add a method `getColor`, why wouldn't it have to be recursive as well?*

2. Write a class extending `WindowController` that draws a `NestedRect` object when the mouse button is pressed, drags it around the screen when the mouse is dragged, and removes it from the screen when the mouse button is released.

An iterative version of `NestedRect` [1]

By way of contrast, let's take a quick look at an iterative version of nested rectangles. One difficulty that we run into immediately is that we need to know how many rectangles will be drawn so that we can give names to each of them. Unfortunately, because the number of rectangles drawn depends on the values of the height and width, we cannot know when we write the class how many framed rectangles will be needed.

When the height and width of the `NestedRect` are large, there will be many rectangles drawn. Because of this and the fact that we don't know in advance how many rectangles will be needed, it makes more sense to consider storing these rectangles in an array, rather than trying to come up with individual names for each. While we could do some calculations to determine how many rectangles will be needed when we create the array in the constructor, we will be lazy and simply assume there will never be more than 50 `FramedRectangle`s needed in drawing a `NestedRectangle`.

An iterative version of the `NestedRect` class is provided in Figure 9.3. Both the `moveTo` and `removeFromCanvas` methods require for loops to go through all of the elements of the `rects` array and perform the appropriate operations. The code for `moveTo` is complicated by the fact that successive framed rects are moved farther and farther.

While the iterative version of `NestedRect` is not that much more complex than the recursive version, it does require more support from complex data structures like arrays.

9.1.2 Writing and understanding recursion

Now that we have run through this complete example, let's step back and see if we can understand how to write recursive constructors and methods in general. Recursive algorithms always have *base cases* and *recursive cases*. Base cases are those where there is no recursive call, while recursive cases are those where there is a recursive call of the method or constructor being defined.

We write the constructor or method as follows:

1. Write the "base case". This is the case where there is no recursive call. Convince yourself that this works correctly.

2. Write the "recursive case".

[1] This section should be skipped if the reader has not yet covered the chapter on arrays.

```
public class NestedRect {
    FramedRect[] rects;     // Array of nested rectangles
    int numRects;           // Number of rectangles stored in rects

    // Draw nested rectangles at the given x and y coordinates and
    // with the given height and width (both of which must be
    // non-negative)
    public NestedRect(double x, double y, double width,
                      double height, DrawingCanvas canvas) {
        rects = new FramedRect[50];
        rects[0] = new FramedRect( x, y, width, height, canvas );
        numRects = 1;
        while ( width >= 4 && height >= 4) {
            x = x + 2;
            y = y + 2;
            width = width - 4;
            height = height -4;
            numRects++
            rects[i] = new FramedRect(x, y, width, height, canvas);
        }
    }

    // Move nested rect to (x,y)
    public void moveTo(double x, double y) {
        for (int count = 0; count < numRects; count++) {
            rects[count].moveTo(x+2*count,y+2*count);
        }
    }

    // Remove the nested rect from the canvas
    public void removeFromCanvas() {
        for (int count = 0; count < numRects; count++) {
            rects[count].removeFromCanvas();
        }
    }
}
```

Figure 9.3: Iterative version of `NestedRect` class using array.

- Make sure all recursive calls go to simpler cases than the one you are writing. Make sure that the simpler cases will eventually get to a base case.
- Make sure that the general case will work properly if all of the recursive calls work properly.

Let's examine this advice in light of the `NestedRect` example. We first look at the constructor. The base case here is when either the width or height is less than 4. If so, the constructor creates the `FramedRect` for `outer` and leaves `rest` with the value `null` because the condition for the `if` statement is false. This is the correct behavior for the constructor because if the width or height is less than 4 there is only room for one rectangle in the picture.

The recursive case occurs when both the `width` and `height` are 4 or larger. In this case, the recursive call passes along values for width and height that are 4 pixels smaller than the original call. Thus this is a call to a simpler case. Because successive recursive calls have smaller and smaller values for the `width` and `height`, we will eventually get a call where the `width` or `height` is less than 4, and hence gives a base case.

Next we need to convince ourselves that the general case works properly as long as all of the (simpler) recursive calls work. In the general case of the constructor, we draw an outer `FramedRect`, and then create a new `NestedRect` object 2 units to the right and below, and with width and height 4 units less than the original. If that recursive call of the constructor really does work properly – i.e., it does draw nested rectangles at the position and with that width and height, then the `outerRect` will be drawn in just the right place to make a slightly bigger collection of nested rectangles. Thus, if we assume the simpler recursive calls of the constructor do what they are supposed to, then the general case will work properly.

Let's go through a similar argument for the `moveTo` method. This time the base case is when `rest` is `null`. In that case, there is only one rectangle and we move it to the correct position. This is clearly correct.

In the recursive case, after having moved `outerRect` to the correct position, we send the `moveTo` method to `rest`, which is a simpler set of `NestedRect`. Thus the recursive call goes to a simpler object than the one we are trying to move. Because successive collections of nested rectangles are smaller and simpler, eventually we will get to one where there is only the outer rectangle and `rest` is `null`. Thus we will eventually get to the base case.

Finally, suppose that the message send of `moveTo` to `rest` correctly moves those nested rectangles to `(x+2,y+2)`. Then moving the outer rectangle to `(x,y)` results in correctly nested rectangles, with the outer one at `(x,y)`, as desired.

Similar arguments can be used to show that `removeFromCanvas` correctly removes all of the rectangles from the canvas.

Why does verifying these three items suffice to ensure that our recursive algorithm will work correctly? A proof of this could be given using mathematical induction. Since not all readers of this text may have encountered mathematical induction, we will instead give a more intuitive argument for why this suffices.

Rather than giving an argument in general, let's give a specific argument in the case of the constructor for `NestedRect`. A similar argument works for all other cases of recursive constructors and methods.

We will start from the smallest collections of `NestedRect` and work our way up. If the smallest side is less than 4 pixels, then the constructor will construct exactly one `FramedRect` of the desired size in the location specified. The recursive call of the constructor is not executed because the condition in the `if` statement is false. Clearly drawing the one `FramedRect` is what should happen,

so everything works fine when the smallest side is less than 4 pixels. Notice that this corresponds to checking the base case of the recursion.

Now suppose the smallest side is less than 8 pixels, but greater than or equal to 4 pixels. Then when the constructor is called, a `FramedRect` is drawn with the specified dimensions and at the specified location. The condition on the `if` statement is now true, so the recursive call of the constructor is executed. Because the original smallest side was less than 8 pixels, and the recursive call involves parameters for length and width that are reduced by 4 each, the smallest side for the recursive call is less than 4 pixels. It is also drawn 2 pixels to the right and below the original call.

We know the call of the constructor is correct when the smallest side is less than 4 pixels, so we know that the recursive call does the right thing – that is, it draws a single `FramedRect` in the desired position. Since the original call resulted in drawing a `FramedRect` which is 4 pixels larger on each side, we know that the combination of drawing the `FramedRect` and the recursive call result in drawing two nested rectangles, as desired.

Now suppose the smallest side is less than 12 pixels, but greater than or equal to 8 pixels. Examining the constructor we see that it draws a `FramedRect` with the specified dimensions and at the specified location and then calls the constructor recursively with length and width each reduced by 4. That is, the recursive call has smallest side less than 8 pixels, but greater than or equal to 4 pixels. However, we already know that a call of the constructor with smallest side of this size draws nested rectangles correctly. Thus it is easy to convince ourselves by looking at where each of these is drawn that a call with smallest side less than 12 pixels, but greater than or equal to 8 pixels, works correctly and draws 3 nested framed rectangles.

We could continue in this way as far as we like, but you should be able to recognize the pattern now. Each time we draw a new framed rectangle and rely on the fact that we already know that the recursive call of the constructor does what it is supposed to. This is exactly what we our instructions told us needed to be done in the two clauses for writing recursive cases of a recursive constructor or method.

Thus in examing the first case (for smallest side less than 4 pixels), we applied part 1 of the rules for writing recursion involving the base case. In each of the other cases we applied part 2 of the rules involving the recursive case. We did this by checking that the recursive call was indeed simpler and that the new case worked under the assumption that the recursive call of the constructor worked.

Very similar arguments can be used to show that the `moveTo` and `removeFromCanvas` methods for `NestedRect` are correct. The idea of our rules for checking base cases and recursive cases is that they tell us exactly what must be verified in order to have confidence that our recursive constructors and methods will be correct.

9.1.3 Broccoli

Recursion can be used to create other interesting and fun examples. A picture of broccoli is shown in Figure 9.4. If you examine the picture carefully, you will notice the recursive structure of the plant. At the base is a stem that is about one-fourth of the height of the plant. At the upper end the stem branches off three ways – one to the left, one straight up, and one to the right. If we look at what grows out of those three branches, we see that those structures themselves look exactly like smaller broccoli plants. We will take advantage of this structure by drawing broccoli as a stem with three smaller broccoli plants attached to the ends of the stem.

We draw broccoli by following the approach in the previous section. We will keep track of the size of the broccoli by keeping track of the size of the stem. This will help us decide whether we

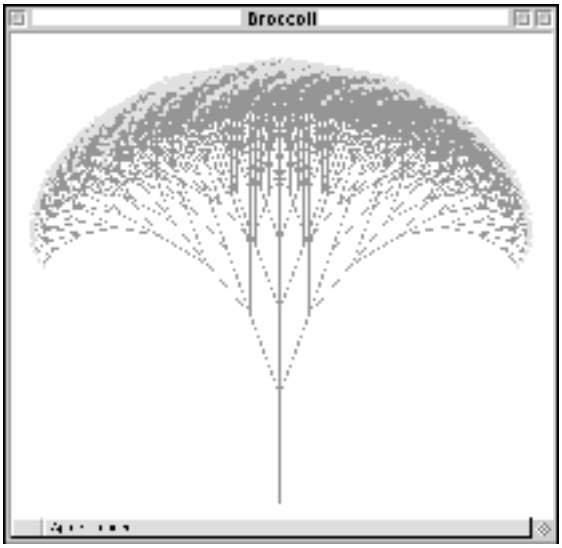


Figure 9.4: A broccoli plant

are in the base case or in the recursive case

If the stem is more than 25 pixels long then we will attach three smaller broccoli plants to the end of the stem. Each of these will have a stem which is 80% of the size of the one just drawn. Otherwise, if the stem is at most 25 pixels long then we will attach three flowers to the end of the stem. Thus the base case occurs when the stem length is at most 25 pixels and the recursive case when it is larger than 25 pixels. Notice that if we start with a stem length of greater than 25 pixels, then after reseting the stem length to 80% of its value over and over, we will eventually get a stem length which is 25 pixels or smaller. Thus our recursive constructor will terminate. We'll verify the correctness after we have written the code.

Before we start writing the code, there are two issues we need to resolve. The first is that while the first stem will be drawn vertically, successive stems will be drawn at other angles (in particular, the left-most and right-most branches will be drawn at angles of $\pi/9$ radians (or 20^o) from that of the base stem. The `Line` class in the objectdraw library has a constructor that takes end points of a line, but it would be more convenient to have a constructor that took the starting point, angle, and length of a line.

The ObjectDraw library contains a class `AngLine` that has the same methods as `Line`, but has a constructor with parameters that are more convenient for drawing lines at an angle. We show the constructor declaration below:

```
public AngLine(Location start, double length, double radianAngle,
                                    DrawingCanvas canvas)
```

The second issue we must contend with shows up when we start thinking about how to write the `Broccoli` class. In order to be able to move the broccoli around on the screen, we would like names for the different parts of the broccoli. We need an instance variable, `stem`, of type `AngLine` to represent the stem. We also need three instance variables to represent the three broccoli parts that grow off of the end of the stem. When the length of the stem is greater than 25 pixels, the three broccoli parts will themselves be instances of `Broccoli`, but if the length does not exceed

```
/**
 * BroccoliPart is an interface for parts of broccoli.
 */
public interface BroccoliPart
{
  // move the broccoli part by x in the horizontal direction and
  // y in the vertical direction
  public void move( double x, double y);
}
```

Figure 9.5: BrocolliPart interface

25 pixels then the three parts will be instances of a class `Flower` that we will design. Thus these instance variables need to be able to hold either instances of the `Broccoli` or `Flower` class.

We have seen this kind of problem before, and have seen that interfaces are exactly suited to solve this problem. Thus we will create a new interface, `BroccoliPart`, that will specify exactly the methods needed by these values, and we will ensure that both the `Broccoli` and `Flower` classes implement the interface.

In fact, all we will require of our broccoli parts is that we can move them around the screen. Thus, all we really need is that they provide a `move` method. The rather trivial definition of interface `BroccoliPart` is given in Figure 9.5.

With all of this out of the way, we are now ready to define the classes `Broccoli` and `Flower`. Because the `Flower` class is the simplest (it is not recursive), we define it first. See Figure 9.6.

The constructor for flower takes the starting location `startCoords`, length of stem, `size`, angle that the stem makes with the x-axis, `direction`, and the `canvas` that it will be drawn on. An instance of `Flower` is composed of a `AngLine`, `stem`, which is colored dark green, connected to a `FilledOval`, `bud`, which is colored yellow. Notice how handy the method `getEnd` is in finding the end of a line which was created as an `AngLine`. The `move` method for `Flower` is trivial, as we simply move the `stem` and `bud` the appropriate distance.

Figure 9.7 contains the code for the `Broccoli` class. The constructor takes the same parameters as `Flower`. it draws the `stem` and then, depending on `size`, adds either three `Broccoli`s or three `Flower`s to the end of the stem. The instance variables, `left`, `center`, and `right`, all have type `BroccoliPart`, so they can hold values from either the `Broccoli` or `Flower` classes. The value of `center` will face the same direction as the stem, while the the values of `left` and `right` are tipped at angles of $\pi/9$ or 20^o to the left or right of the stem.

The base case of the constructor is when size does not exceed 25 pixels. In that case, the constructor creates the `stem` and three `Flower`s that attach to the end. This is a reasonable picture for the base case.

The recursive case is where `size` is greater than 25. In this case, the constructor creates the `stem` and then three `Broccoli`s that attach to the end. The recursive calls to the `Broccoli` constructor have a size that is 80% of the original size. Thus they construct simpler `Broccoli`s (the complexity of `Broccoli` is measured by the `size` of its stem), as required by our rules for writing recursive constructors and methods. If each time the new size is 80% of the previous size then eventually it will become 25 or smaller, so we will eventually get to the base case.

Finally, we must verify that the constructor does the right thing if we assume that all simpler calls of the constructor do the right thing. In this case, it is easy that the constructor does the

```
public class Flower implements BroccoliPart {
    // color of Broccoli - dark green
    private static final Color broccoliColor = new Color(0,135,0);

    private AngLine stem;                       // stem of broccoli

    private FilledOval bud;                 // Flower of broccoli plant

    /**
     * Draw flower at end of a stem
     * @param startCoords - starting point of stem
     * @param size - length of the stem
     * @param direction - angle of stem with x-axis
     * @param canvas - where picture is drawn
     */
    public Flower(Location startCoords, double size,
                  double direction, DrawingCanvas canvas)
    {
        // Draw stem and color green
        stem = new AngLine(startCoords,size,direction,canvas);
        stem.setColor(broccoliColor);

        Location destCoords = stem.getEnd();    // end of stem

        bud = new FilledOval(destCoords,3,3,canvas);
        bud.setColor(Color.yellow);
    }

    /**
     * @param x,y amount to move flower
     */
    public void move( double x, double y)
    {
        stem.move(x,y);                          // move stem
        bud.move(x,y);
    }
}
```

Figure 9.6: `Flower` class implementing `BroccoliPart`

```
/**
 * Class to recursively draw broccoli
 */
public class Broccoli implements BroccoliPart {
    private BroccoliPart left, center, right;   // branches of broccoli
    private AngLine stem;                          // stem of broccoli

    // Draw broccoli by recursively drawing branches (and flowers)
    public Broccoli(Location startCoords, double size,
                    double direction, DrawingCanvas canvas)
    {
        // Draw stem and color green
        stem = new AngLine(startCoords,size,direction,canvas);
        stem.setColor(Color.green);
        Location destCoords = stem.getEnd();    // end of stem

        if ( size > 25 )                // Big enough to keep growing
        {
            left = new Broccoli(destCoords, .8*size,
                                direction + Math.PI/9.0, canvas);
            center = new Broccoli(destCoords, .8*size, direction, canvas);
            right = new Broccoli(destCoords, .8*size,
                                 direction - Math.PI/9.0, canvas);
        } else {                        // Draw flowers
            left = new Flower(destCoords, .8*size,
                              direction + Math.PI/9.0, canvas);
            center = new Flower(destCoords, .8*size, direction, canvas);
            right = new Flower(destCoords, .8*size,
                               direction - Math.PI/9.0, canvas);
        }
    }

    // @param x,y amount to move broccoli
    public void move( double x, double y)
    {
        stem.move(x,y);                 // move stem

        left.move(x,y);                 // move other parts
        center.move(x,y);
        right.move(x,y);
    }
}
```

Figure 9.7: `Broccoli` class

right thing as it is easy to see from the picture that a non-trivial `Broccoli` consists of a stem with three `Broccoli` objects attached to the end of the stem.

Because we have verified all the conditions for writing correct recursive constructors and methods, we can be confident that the constructor does what it is supposed to.

Now let us take a look at the `move` method for `Broccoli`. To move `Broccoli` we just move the `stem` and the `left`, `center`, and `right` `BroccoliParts`. This seems quite reasonable, but it is worth noticing that the structure of this method is quite different from the recursive constructors and methods that we have discussed to this point. All of the other recursive methods and constructors included an `if` statement to separate the base case from the recursive case.

Exercise 9.1.2 *Before reading further, see if you can figure out how the recursive and base cases occur for this method.*

The key to understanding the mystery of where the base case is handled is understanding that which `move` method body is executed when the `move` message is sent to `left`, `center`, and `right` depends on the class of the values held in those variables. Thus, if the values held in the three variables are from class `Flower` then the `move` method of `Flower`, which is `not` recursive, will be executed. On the other hand, if the values are from class `Broccoli`, then the `move` method of `Broccoli` will be executed, which does involve recursive calls of `move`.

Thus the base case corresponds to when the values of the instance variables are from class `Flower`, while the recursive case occurs when the values are `Broccoli`. We can check that `move` works correctly by following the usual rules. First there is a base case (when the variables represent `Flowers`), and it works correctly by moving the stem and bud the appropriate amount.

The recursive case (when the instance variables represent `Broccoli`) involves a call to a simpler case – the broccoli instance variables have shorter stems (because of the way the constructor works) and hence they will eventually not exceed 25 pixels, and thus we will eventually get to a case where the instance variables are indeed `Flowers`. Finally if we assume the recursive sends of `move` to the instance variables work correctly, then it is easy to see that the complete method works correctly: the stem is moved the appropriate amount and each of the three broccoli parts are moved the same amount.

Exercise 9.1.3 *Add new methods `moveTo` and `removeFromCanvas` to the `Broccoli` and `Flower` classes.*

Exercise 9.1.4 *To make the `Broccoli` drawing more interesting, we can make animate its growth by making it into an `ActiveObject`. The constructor should draw the stem and then call `start()`. The `run()` method should pause for 500 milliseconds and then create the three instance variables that correspond to the three branches.*

Exercise 9.1.5 *Parsley is another plant that can be drawn recursively. Write a program to draw parsley as shown in Figure 9.8.*

9.1.4 Snowflake fractals

Another interesting recursively designed figure is the snowflake curve shown in Figure 9.9. The snowflake is drawn by drawing a triangle. But rather than drawing the triangle out of straight lines, we will draw it out of objects of type `PushLine`.

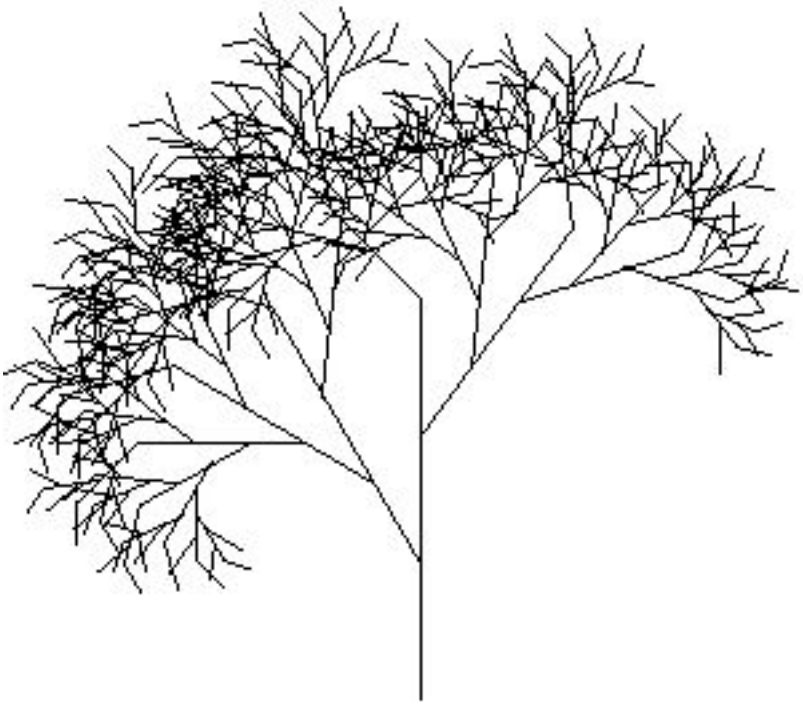


Figure 9.8: Parsley

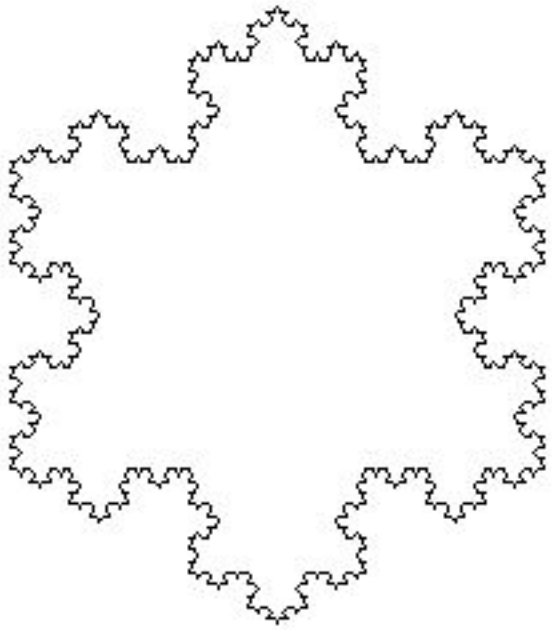


Figure 9.9: Snowflake fractal.

9.2 Non-graphical recursion

An object of type `PushLine` is drawn by drawing a straight line and then erasing the middle third of the line. The middle third is replaced by two lines of the same length as the original middle third that are connected to the endpoints of the original middle third and are themselves connected to form the shape `_/\_`. (See also Figure **??**.) You might find it convenient to think of this new figure as being drawn by grabbing the center of that middle segment of the line and pushing it out until it is twice as long as it was when it started. If the middle segment were still where it was, then the new lines and the original would form an equilateral triangle.

Of course, we don't want to stop there. (This is recursion after all!) Instead take all four of the resulting line segments, break them into thirds and push them out again. Continue until the length of the segments falls below some predetermined limit. When that happens, we simply draw straight lines.

The class `PushLine` will have instance variables to hold the different parts of the figure. Most of the time those instance variables will themselves be holding `PushLine`s. However, at the base case they will simply be objects of class `AngLine`.

As in the `Broccoli` class example, we would like to have an instance variable that can hold either of the two possibilities for the sides: `AngLine` or `PushLine`. To make this possible, we will make sure both implement the same interface, `Drawable1DInterface`, which is provided in Figure 9.10.

The interface `Drawable1DInterface` is intended to describe a one-dimensional objects with a start and end location. It is a bit simpler than normally would be expected – for example, there is no way to set the start and end points. However, it is sufficient for our purposes here. Both AngLine and `PushLine` classes implement the interface `LineInterface`.

The code for class `PushLine` is given in Figures 9.11 and 9.12. The constructor of `PushLine` has a structure similar to that of the `Broccoli` constructor in the last section. If the `length` is smaller than `limit`, then four straight line segments will be constructed from class `AngLine`. However, if the `length` is at least as large as `limit`, then four `PushLine`s will be constructed.

The `move` and `setColor` methods are defined recursively. That is, the same message is passed along to the four segments held as instance variables. In recursive cases, these messages will be sent to other `PushLine`s, while in base cases, the messages will be sent to `AngLine`s.

The methods `getStart`, `getEnd`, and `moveTo` are not recursive, though `moveTo` calculates how far the `PushLine` should move and then invokes its own `move` method.

The argument for the correctness of the constructor and methods is essentially the same as for Broccoli. We can complete the program for drawing snowflake curves by writing an extension of `WindowController` that draws three `PushLine`s in the `begin` method. See Figure 9.13 for details.

9.3 Non-graphic problems and recursion

In this section we illustrate recursive solutions to two problems that do not involve drawing recursive pictures. The first involves a fast algorithm for raising a number to a non-negative integer power, while the second involves the solution to an interesting puzzle.

9.3.1 Fast exponentiation

Fast algorithms for raising large numbers to large integer powers are important to the RSA algorithm for public key cryptography. We won't discuss cryptography here, but we will investigate a

```
public interface Drawable1DInterface {

    /**
     * Gets the line's start point.
     *
     * @return the start point
     */
    public Location getStart ();

    /**
     * Gets the line's end point.
     *
     * @return the end point
     */
    public Location getEnd ();

    /**
     * move line by (dx,dy)
     * @param dx,dy amount to move line
     */
    public void move(double dx, double dy);

    /**
     * move line to (x,y)
     * @param x,y - new location for start of line
     */
    public void moveTo(double x, double y);

    /**
     * set color of object to newColor
     * @param newColor - new color of object
     */
     public void setColor(Color newColor);

}
```

Figure 9.10: `Drawable1DInterface` implemented by `AngLine` and `PushLine`.

```
public class PushLine implements LineInterface
{
    private Location start, end;              // start and end points of line

    // four lines constructing "PushLine"
    LineInterface firstLine, secondLine, thirdLine, fourthLine;

    /**
     * Constructor for line which eventually pushes out into fractal
     * @param start - starting coordinates for line
     * @param length - length of original line
     * @param radians - angle of line from the horizontal
     * @param limit - if length < limit then don't push out into fractal
     * @param canvas - where line will be drawn
     */
    public PushLine(Location start, double length, double radians, double limit,
                                            DrawingCanvas canvas) {
        this.start = start;

        // Lines to be pushed
        Location secondPt, thirdPt, fourthPt;           // end points of lines

        if (length < limit) { // Base case -- just make lines in _/\_ shape
            firstLine = new AngLine(start,length/3,radians,canvas);
            secondPt = firstLine.getEnd();
            secondLine = new AngLine(secondPt,length/3,radians+Math.PI/3,canvas);
            thirdPt = secondLine.getEnd();
            thirdLine = new AngLine(thirdPt,length/3,radians-Math.PI/3,canvas);
            fourthPt = thirdLine.getEnd();
            fourthLine = new AngLine(fourthPt,length/3,radians,canvas);
        }
        else  {                 // Recursive case -- make _/\_ shape with PushLines
            firstLine = new PushLine(start,length/3,radians,limit,canvas);
            secondPt = firstLine.getEnd();
            secondLine = new PushLine(secondPt,length/3,radians+Math.PI/3,limit,canvas);
            thirdPt = secondLine.getEnd();
            thirdLine = new PushLine(thirdPt,length/3,radians-Math.PI/3,limit,canvas);
            fourthPt = thirdLine.getEnd();
            fourthLine = new PushLine(fourthPt,length/3,radians,limit,canvas);
        }

        end = fourthLine.getEnd();
    }
```

Figure 9.11: `PushLine`, Part 1.

```
    /**
     * @returns start point of line
     */
    public Location getStart() {
            return start;
    }

    /**
     * @returns end point of line
     */
    public Location getEnd() {
            return end;
    }

    /**
     * move line by (dx,dy)
     * @param dx,dy amount to move line
     */
    public void move(double dx, double dy) {
            firstLine.move(dx,dy);
            secondLine.move(dx,dy);
            thirdLine.move(dx,dy);
            fourthLine.move(dx,dy);
    }

    /**
     * move line to (x,y)
     * @param x,y - new location for start of line
     */
    public void moveTo(double x, double y) {
            move( x - firstLine.getStart().getX(),
                  y - firstLine.getStart().getY() );
    }

    /**
     * set color of object to newColor
     * @param newColor - new color of object
     */
     public void setColor(Color newColor) {
        firstLine.setColor(newColor);
        secondLine.setColor(newColor);
        thirdLine.setColor(newColor);
        fourthLine.setColor(newColor);
     }

}
```

Figure 9.12: `PushLine` class, Part 2.

```
public class FractalApplet extends WindowController
{

   public static final double PI = 3.14159; // The number Pi
   public static final int limit = 10; // Size at which stop pushing line into fractal
   public static final double size = 200.0; // Size of lines in original triangle
public static final double startX = 100.0; // Starting coordinates of lower left corner
public static final double startY = 150.0; // of triangle

    PushLine line1, line2, line3; // 3 lines bounding star

    Location lastLocation; // last location of mouse
  /**
   * Initializes the applet by creating the triangle with three "PushLine"s.
   */
  public void begin(){
    line1 = new PushLine(startX,startY,size,0,limit,canvas);
    Location dest = line1.getEnd();
    line2 = new PushLine(dest,size,4*PI/3,limit,canvas);
    line3 = new PushLine(line2.getEnd(),size,2*PI/3,limit,canvas);
  }

// Get ready to move star
public void onMousePress( Location point)
{
lastLocation = point;
}

// Drag the star around
public void onMouseDrag( Location point)
{
line1.move( point.getX()-lastLocation.getX(),
            point.getY()-lastLocation.getY());
line2.move( point.getX()-lastLocation.getX(),
            point.getY()-lastLocation.getY());
line3.move( point.getX()-lastLocation.getX(),
            point.getY()-lastLocation.getY());
lastLocation = point;
}
}
```

Figure 9.13: Class to draw a snowflake curve.

recursive algorithm that is substantially faster than the usual way of raising numbers to powers.

We begin by describing a simple recursive method to raise integers to non-negative powers that is no more (or less) efficient than the obvious iterative algorithm.

As usual, the basic idea is to describe how you would complete the solution to the problem if someone were to provide you with a partial solution. In this case, if we want to raise a number to the nth power, we presume there is someone else who can raise it to the n-1st power.

```
/**
 *  @param exponent >= 0
 *  @returns base raised to exponent power
 **/
public int simpleRecPower(int base, int exponent)
{
   if (exponent == 0)
      return 1;
   else
      return base * simpleRecPower(base,exponent-1);
}
```

Let's check this out with the rules stated in Section 9.1.2.

1. The base case is where `exponent` == 0. It returns 1, which is the correct answer for raising `base` to the 0th power.

2. The recursive case is in the `else` clause.

   - The recursive call involves `base` to the `exponent - 1`st power, which is smaller than `exponent`. Because `exponent` is assumed to be greater than 0 (the original assumption was that `exponent` is greater than equal to 0, but because we are in the else clause we know `exponent` is not 0), the recursive calls will eventually get down to the base case of 0.
   - If we assume that evaluating `simpleRecPower(base,exponent-1)` results in $\texttt{base}^{\texttt{exponent-1}}$ then the else clause returns

$$\begin{aligned} \texttt{base*simpleRecPower(base,exponent-1)} &= \texttt{base} * \texttt{base}^{\texttt{exponent-1}} \\ &= \texttt{base}^{\texttt{exponent}}. \end{aligned}$$

Thus we can be confident that the above algorithm calculates $\texttt{base}^{\texttt{exponent}}$. It is also easy to see that the evaluation of `simpleRecPower(base,n)` results in exactly n multiplications, because there is exactly one multiplication associated with each recursive call.

Using a simple modification of the above recursive program, we can get a much more efficient algorithm for calculating very large powers of integers. In particular, if we use the above program (or the equivalent simple iterative program) it will take 1024 multiplications to calculate $\texttt{b}^{1024}$, for any integer `b`, while the program we are about to present cuts this count down to only 11 multiplications!

The algorithm above takes advantage of the following simple rules of exponents:

- $base^0 = 1$
- $base^{exp+1} = base * base^{exp}$

The new algorithm that we present below takes advantage of one more rule of exponents:

- $base^{2*exp} = (base^2)^{exp}$

The key is that by using the last of these rules as often as possible, we can cut down the amount of work considerably, by reducing the size of the exponents in recursive calls faster. See the code below:

```
/**
    @param exponent >= 0
    @returns base raised to exponent power
**/
public int fastRecPower(int base, int exponent) {
    if (exponent == 0)
        return 1
    else if (exponent%2 == 1)       // exponent is odd
        return base * fastRecPower(base, exponent-1)
    else                            // exponent is even
        return fastRecPower(base * base, exponent / 2)
}
```

The `fastRecPower` method performs exactly the same computation as the previous one when the exponent is 0 or an odd integer. However, it works very differently when the exponent is even. In that case, it squares the base and divides the exponent in half.

Before analyzing exactly why this method works, let's first look at an example of the use of this algorithm and count the number of multiplications.

```
fastRecPower(3,16) = fastRecPower(9,8)                // mult
                   = fastRecPower(81,4)               // mult
                   = fastRecPower(6561,2)             // mult
                   = fastRecPower(43046721,1)         // mult
                   = 43046721 * fastRecPower(43046721,0)
                   = 43046721 * 1                     // mult
                   = 43046721
```

Thus it only took 5 multiplications using `fastRecPower`, whereas it would have taken 16 multiplications the other way. While divisions are usually at least as expensive as multiplications on computers, divisions by two can be done very efficiently because numbers are represented in binary. Thus we will not bother to count division by 2 or using the "%" operation with 2 as something worth worrying about in terms of the time complexity of the algorithm.

In general it takes somewhere between $log_2(\texttt{exponent}) + 1$ (for exponents that are powers of 2) and $2 * log_2(\texttt{exponent})$ multiplications to compute a power this way. While this doesn't make a difference for small values of `exponent`, it does make quite a difference when `exponent` is large. For example, as noted above, computing `fastRecPower(b,1024)` would only take 11 multiplications, while computing it the other way would take 1024 multiplications.

Let's once again use the standard rules for understanding recursion to see why this algorithm is correct.

1. The base case is again where `exponent == 0`. It returns 1, which is the correct answer for raising `base` to the 0th power.

2. The recursive case is in the `else` clause, but this time divides into two cases, depending on whether `exponent` is odd or even.

   - When `exponent` is odd, the recursive call is with `exponent - 1`, while, when `exponent` is even, the recursive call is with `exponent / 2`. In either case, each of these calls is for a smaller integer power because `exponent` is greater than zero. Thus no matter which case is selected at each call, eventually the power will decrease to the base case, 0.
   - We have already given the correctness argument for the code in the odd case in our earlier analysis of the `simpleRecPower` method. Let's convince ourself that it works for the even case.

     Let `exponent` be an even positive integer. Then the method returns the value of `fastRecPower(base * base, exponent / 2)`. That is simpler than the original call because `exponent/2` is less than `exponent`. Thus we can assume that it returns the correct answer:

     $$\begin{aligned}(\texttt{base} * \texttt{base})^{\texttt{exponent/2}} &= (\texttt{base}^2)^{\texttt{exponent/2}} \\ &= \texttt{base}^{\texttt{2 * exponent/2}} \\ &= \texttt{base}^{\texttt{exponent}}\end{aligned}$$

     Note that 2 * `exponent/2` = `exponent` only because `exponent` is even. If `exponent` is odd, the truncation in integer division would cause the product to evaluate to `exponent - 1`.

Thus we see that the `fastRecPower` algorithm is correct.

While this algorithm can be rewritten in an iterative style, the recursive algorithm makes it clearer where the rules of exponents are coming into play in the algorithm.

Exercise 9.3.1 *a. It is not hard to see how to rewrite the `simpleRecPower` algorithm with a loop. Please write it out.*

b. Writing the iterative equivalent of the `fastRecPower` algorithm is a bit trickier. Please write it out.

9.3.2 Towers of Hanoi

One of the more interesting uses of recursion comes from an old story:

> The Towers of Hanoi puzzle is reputed have arisen from a story about a group of Buddhist monks in the Tower of Brahma. In the monastery were 3 diamond-tipped needles which were supported vertically. On the first of the diamond-tipped needles were 64 golden disks, arranged in order of size so that the largest was on the bottom. The monks were reputedly given the task of moving the 64 golden disks from the first to the third golden-tipped needle. Making the problem a bit more complicated were the restrictions that only one disk can be moved at a time from one needle to another (no disk may just be set aside) and that it is never allowable to put a large disk on a smaller one.

One can buy a children's puzzle based on this story, though these puzzles are typically made of plastic or wood, and only come with 8 or fewer disks (for reasons that will become apparent later). See a picture in Figure 9.14.

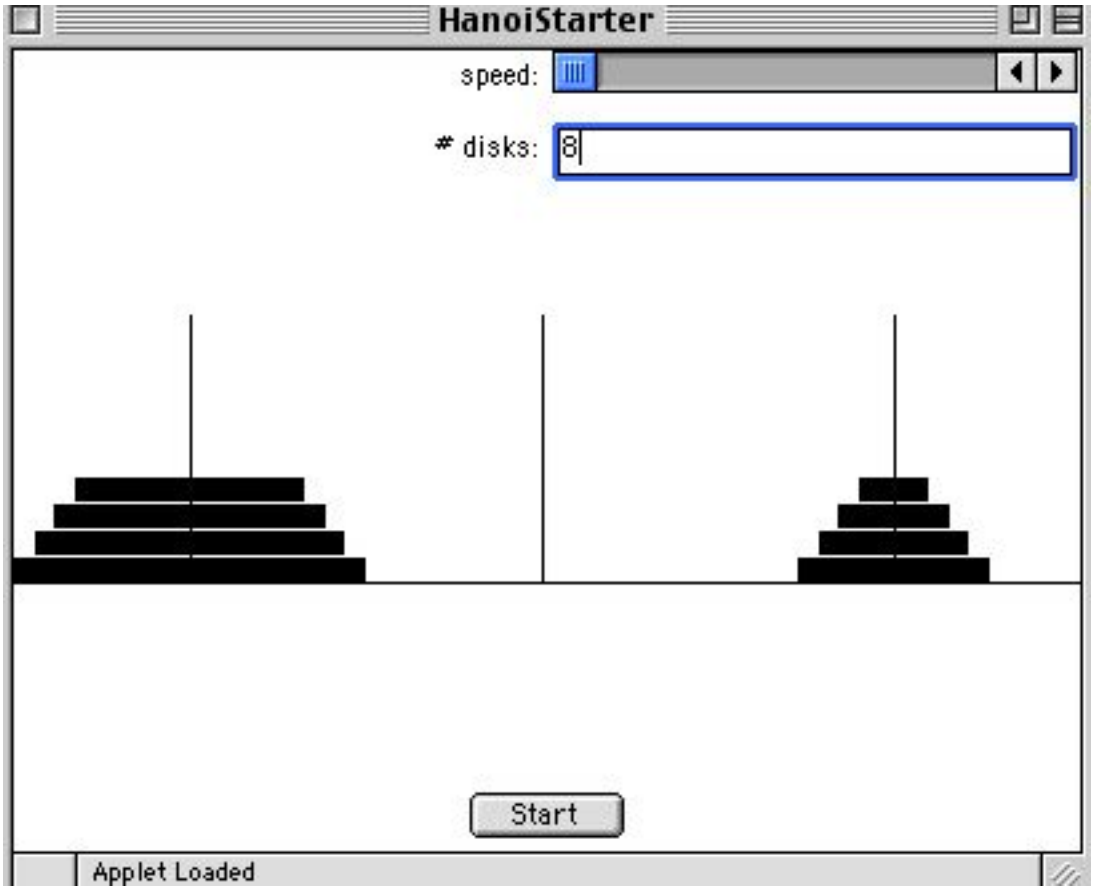


Figure 9.14: Towers of Hanoi puzzle with 8 disks.

The key to solving the puzzle is to consider how you could move the biggest disk from the bottom item of the first needle to the bottom item of the third needle. Then think recursively!

A little thought should convince you that to move the biggest disk from the first to last needle, you must first move all of the smaller disks to the middle needle. If any of the smaller disks are still left on the first needle then they would be on top of the biggest disk, and hence the biggest disk could not be moved. Similarly, if any of the smaller disks are on the third needle, then we could not possibly move the biggest disk onto that needle, as that would be an illegal move.

Hence all but the largest disk must be moved to the middle needle. Then the biggest disk must be moved to the third needle. Finally the remaining n-1 smaller disks must be moved from the middle needle to the last needle.

We can write down this procedure more carefully as follows:
To move n disks from the first needle to the last needle using a helper needle:

1. If there is only one disk to move, just move it from the first to last needle, and you are done.
2. Otherwise move the top n-1 disks from the first needle to the helper needle (following the rules, of course).
3. Then move the bottom disk from the first to the last needle.
4. Then move the top n-1 disks from the helping to the last needle (following the rules, of course).

Here is a method to do this, written in Java. It assumes that the method `moveDisk`, which moves a single disk from one needle to another has been written elsewhere.

```
public void recHanoi (int numDisks, int first, int last, int helper)
{
    if (numDisks == 1)
    {
        moveDisk(numDisks, first, last);  \emph{// move the only disk}
    }
    else
    {
        recHanoi(numDisks - 1, first, helper, last);
        moveDisk(numDisks, first, last);  \emph{// move the bottom disk}
        recHanoi(numDisks - 1, helper, last, first);
    }
}
```

We can understand this method a little better by looking at a specific example:

```
                                  {towers(1,A,C,B)     {moveDisk(1, A, C)
               {towers(2,A,B,C)   {moveDisk(2, A, B)
               {                  {towers(1,C,B,A)     {moveDisk(1, C, B)
towers(3,A,C,B) {moveDisk(3, A, C)
               {                  {towers(1,B,A,C)     {moveDisk(1, B, A)
               {towers(2,B,C,A)   {moveDisk(2, B, C)
                                  {towers(1,A,C,B)     {moveDisk(1, A, C)
```

That is, a call to `towers(3,A,C,B)` results in calls to `towers(2,A,B,C)`, `moveDisk(3,A,C)`, and `towers(2,B,C,A)`. Each of these recursive calls of method `towers` gets expanded, and so on.

How do we know this method will work properly? Once more, we go back to our standard check list.

1. The base case of a single disk just moves it where it is supposed to go, and hence is correct.

   - If you start with positive number of disks then the recursive calls to `tower` will be with one fewer disk. Hence recursive calls will eventually get down to 1 disk, the base case.
   - If we assume that the method works for n-1 disks, then it will work for n disks. (*Can you give a convincing argument!*)

Exercise 9.3.2 *We have not shown the code for method `moveDisk`. It could either just print out a message (using `System.out.println`) describing which disk is moved from where to where, or it could result in altering a picture of needles and disks. Write out the code for the text-only version of `moveDisk`. Then write out the code for an animated graphic version of towers of Hanoi where there is a delay between each move of a disk. That is, let an `ActiveObject` control the animation of the algorithm. Interestingly, the code for the graphics is several times as long as the actual code for determining which disk should be moved next.*

Exercise 9.3.3 *Determine how many calls of `moveDisk` are required to run `towers(n,A,B,C)` to completion. Hint: first make a table of the number of moves for n ranging from 1 to 10. From the table, guess a formula involving n for the number of calls of `moveDisk`. Use an inductive argument similar to that given for the correctness of recursive programs to give a convincing argument that your formula is correct. If a robotic arm could move 1 disk per second, how long would it take to move all 64 disks from the first needle to the last needle using this algorithm. Do you now understand why the commercial version of the game only includes 8 disks?*

Exercise 9.3.4 *The recursive algorithm given in the method is the most efficient solution in terms of the number of disks moved. Try to find a convincing argument for this. Hint: Think about what has to be the configuration in order to move the biggest disk.*

We will present more examples of recursion later in Chapter xxx when we examine algorithms for sorting and searching. Meanwhile, you will find that recursive algorithms are relatively simple to write as long as you keep the rules in Section 9.1.2 in mind.