After finishing this chapter, you should be able to:

- Provide brief definitions of the following terms: base case, base condition, conditional recursion, exponential recursion, infinite recursion, iteration, iterative process, linear recursion, overhead, recursion, and recursive method.
- Describe the usage of a recursive algorithm.
- Describe how recursion differs from iteration, including the cost of recursion, and roughly how to tell when one might be more appropriate to use than the other.
- Describe the difference between infinite recursion and conditional recursion.
- Create a recursive method in Alice.
- Convert an iterative Alice method into a linear recursive method.
- Convert a linear recursive Alice method into an iterative method.
WHAT IS RECURSION?

This chapter is about recursion. Generally, something is said to be recursive if each of the parts that make up the thing have a structure—in other words, a design or pattern—that repeats the structure of the whole thing itself. Consider the fern in Figure 7-1. Each leaf of the fern has a pattern that is virtually identical to the overall fern itself. In fact, if you look closely at the leaf, you can see that the parts that make up the leaf have patterns that match the leaf itself. If you were to go deeper and deeper into the structure of the fern, you would see that the same pattern is repeated again and again, each time on a smaller scale, each time in more detail. The fern has a recursive structure—parts of the fern have structures that mirror the overall structure of the fern itself. There are many such examples of recursion in the natural world.

Recursion is not a simple concept, so often it takes a while for someone who has never seen it before to really understand what it’s all about. This chapter is not intended to make you an expert on recursion but only to expose you to the topic. By the time you are done, you should have a basic understanding of what a recursive algorithm is. Before going on, let’s take a look at another example. Can you see how the following very short story is recursive?
Once upon a time a young princess was sitting with her father, the king. She said to her father, “My Royal Father, can you read me a story?”

“Yes my Royal Daughter,” said the king, as he opened the Royal Storybook. He began to read, “Once upon a time a young princess was sitting with her father, the king. She said to her father, ‘My Royal Father, can you read me a story?’”

“Yes my Royal Daughter,” said the king, as he opened the Royal Storybook. He began to read, “Once upon a time a young princess was sitting with her father, the king. She said to her father, ‘My Royal Father, can you read me a story?’”

**RECURSIVE ALGORITHMS**

In computer programming, an algorithm that calls itself is said to be recursive. Throughout this book you have seen how one method can call another method. Consider Figure 7-2a. The primitive method `bunny.move` is one of several methods called from within `world.my first method`. Recursion occurs when a method calls itself in the same way that `world.my first method` calls the `bunny.move` method. Figure 7-2b shows a simple example of a recursive method, `bunny.hop to carrot`, which calls itself. A part of the algorithm is identical to the overall algorithm because the method being called is the overall algorithm. In a sense, the method is nested within itself, just as the recursive structure of the fern is nested within itself in the example above.

**Let’s take a look at another example of a recursive algorithm. Almost everyone knows about the famous ceiling of the Sistine Chapel in Rome, painted by Michelangelo, but do you know about its floor? The marble floor of the Sistine Chapel has patterns in the tiles that look like Figure 7-3. This pattern is an example of a mathematical design called a Sierpinski gasket.**
Here is the algorithm for drawing a Sierpinski gasket:

```
Begin with an equilateral triangle
Sierpinski (triangle)
  Start
  Find the mid point of each side of the triangle
  Draw lines connecting the midpoints, which will form four
  smaller triangles that can be called triangles A, B, C,
  and D, with D in the center and the others around it.
  Color in (or cut out) the center triangle // triangle D
  Do Sierpinski (triangle A)
  Do Sierpinski (triangle B)
  Do Sierpinski (triangle C)
  Stop
```

The Sierpinski Gasket algorithm splits the triangle into four smaller triangles, and then calls itself for three of the four smaller triangles. Figure 7-4 shows the result of the algorithm through 5 levels of recursion.

In the figure, you can see that the process of dividing the triangle into smaller triangles and then coloring in (or cutting out) one of them is repeated over and over again, each time at a smaller level, each time in more detail. The result is a complex structure created by a small, efficient algorithm.
The Sierpinski gasket algorithm demonstrates the power of recursion for drawing complex structures, but the real power of recursion in computer programming is the way in which computer scientists use recursive algorithms to analyze and solve complex problems. An algorithm is really just a set of instructions describing how to complete a process. Recursion can describe a very sophisticated process with a simple and efficient set of instructions. Many of the most important algorithms in computing, such as those used to search large databases quickly, to perform matrix algebra, and to play games such as chess, depend on the use of recursion. Unfortunately, students in introductory computer programming courses don’t yet have the skills to write such programs, but you can spend some time to learn about some of the basic ideas associated with recursion so that you will be able to use recursion appropriately when the time comes to do so.

It’s also worth learning about recursion because of its importance in the world around us. Blood vessels in the human body have a structure that can be best described by recursion, in which they continue to divide into smaller and smaller vessels until they are tiny capillaries only a few blood cells wide. The pattern of seeds on the head of a sunflower, the geometry of a snail’s shell, and even DNA gene sequencing all appear to be recursive structures. The newly emerging fields of Fractal Geometry and Chaos Theory are largely based on recursion. Seemingly random events, such as the electronic interference in a telephone circuit, can be described using recursive functions. There is also some evidence that the human brain is recursive. A basic understanding of recursion will not only help you to become a better computer programmer, but it may also help you to better understand the world around you.

RECURSION COMPARED TO ITERATION

An iterative process is one that uses a loop to repeat a set of instructions, so iteration is just another name for looping. In many cases, either iteration or recursion can be used to do the same thing. The bunny.hop to carrot method shown earlier in this chapter is a good example of such a case. Figure 7-5 shows the recursive bunny.hop to carrot method on the left, with an iterative version of the same method using a while loop on the right. Both methods do the same thing, but one uses recursion, and the other uses iteration.
This brings us to one of the most important questions about recursion—if both iteration and recursion can be used to do the same thing, then when should iteration be used, and when should recursion be used? To be able to answer that question, you need to know a little more about some of the characteristics of recursive methods, including the cost of recursion and the difference between linear and exponential recursion.

**THE COST OF RECURSION**

**Overhead** is the extra CPU time and extra space in a computer's memory needed to manage a method as it runs. The computer must set up a section in its memory every time a new method is called—even if the new method is a copy of an existing method. We don't see this happening, yet it does take up time and space in the computer's memory. In the *bunny.hop to carrot* example, the computer must set up a new section in its memory each time the recursive *bunny.hop to carrot* method is called. If the bunny hops five times, then that means five new sections will be needed in the computer's memory to manage the five copies of the method. If it hops a thousand times, then a thousand new sections in memory will be required. The iterative *bunny.hop to carrot* method does not have such overhead, so even though it has more steps than the recursive method, it is more efficient when implemented on a computer.

People who deal with theoretical algorithms, such as mathematicians, almost always think recursion is a good idea. For more practical computer programmers, the overhead associated with recursion must be considered. Often it is better to use iteration than recursion for methods that can be written using a simple loop. But before giving up on recursion completely, let's go back to the Sierpinski gasket algorithm to see the difference between linear recursion and exponential recursion.
LINEAR RECURSION AND EXPONENTIAL RECURSION

Linear recursion occurs when a method calls itself only once each time through the method. Exponential recursion occurs when a method calls itself more than once in each pass through the method. The graphs in Figure 7-6 show the two compared to each other. The graph on the left shows the number of copies of the method running for each level of recursion in an algorithm that calls itself once. The graph on the right shows the same thing for an algorithm that calls itself twice. You can see that the linear recursion graph looks like a straight line. Only one additional copy of the algorithm is created at each level of recursion. Yet look how much more quickly the number of copies of the exponential algorithm grows. At each level of recursion, the number of copies of the program doubles. If it called itself three times, then the number of copies would triple at each level, and so on. A method that uses exponential recursion can very quickly generate many copies of itself to work simultaneously on smaller examples of the overall problem.

The recursive Sierpinski gasket algorithm is an example of exponential recursion because it calls itself three times. The number of triangles triples with each level of recursion. It is very unlikely that a more efficient algorithm to do the same thing could easily be written using iteration. In this case, recursion probably is the best solution. Generally, a method that uses exponential recursion—that is, a method which calls itself more than once—is often very difficult to replace with a more efficient iterative solution. Exponentially recursive algorithms are often far more efficient than iterative solutions to do the same thing because they can repeatedly break a problem into several smaller parts, then attack those smaller parts in a way that an iterative algorithm cannot. Even considering the overhead for recursion, they are usually far more efficient than iterative solutions.

So, we now have a rough answer to our question about when to use recursion. For simple tasks that can be described with linear recursion, iteration seems to be better because of the overhead of recursion, but for complex problems that can be broken down into smaller parts, exponential recursion usually works much better, even allowing for the overhead of the recursion.
INFINITE RECURSION AND CONDITIONAL RECURSION

There’s one more important concept about recursion that computer programming students should understand—the difference between infinite recursion and conditional recursion. **Infinite recursion** means that an algorithm calls itself repeatedly without stopping. Infinite recursion occurs when an algorithm does not contain any instructions about when to stop the recursion. On a computer, infinite recursion continues until all available memory is used, or until it triggers some timeout mechanism in the operating system. The version of the Sierpinski gasket algorithm presented earlier is an example of infinite recursion. If you examine the algorithm carefully, you will see that the *Stop* instruction is never executed.

Recursion that is not infinite is called conditional recursion since the algorithm must contain some condition that will stop the recursion. The condition that stops the recursion is called the **base condition** or the **base case**. The algorithm usually contains an *If/Else* instruction that causes the algorithm to keep calling itself until the base condition is met. In the recursive *bunny.hop to carrot* method, the algorithm continues to call itself until the bunny is within 1 meter of the carrot. The base condition is that the distance between the bunny and the carrot is 1 meter or less. The *If/Else* command is set up to continue the recursion while the bunny is more than 1 meter away from the carrot.

A properly structured recursive algorithm should always contain a Boolean expression in a selection sequence to test for the base condition in the same way that the *If/Else* instruction in the *bunny.hop to carrot* method does. In the case of *bunny.hop to carrot*, the algorithm simply stops when the base condition occurs. In other algorithms, something can be added to the *Else* part of the instruction to tell the computer what to do when the base condition has occurred.

In conditional exponentially recursive algorithms, the recursive step breaks the problem into smaller parts, and then calls the algorithm itself for each of those parts, continuing to do so until the base case is reached. Sometimes that is enough to solve the original problem; sometimes another step is needed to combine all of the small solutions into one big solution. It is such conditional exponentially recursive algorithms that lead to sophisticated and highly efficient solutions to large programming problems. Unfortunately, they are often used with complex data structures, such as trees, graphs, and matrices, which are not easily demonstrated in introductory Alice programming, although the underlying software that runs Alice worlds does use them, especially the operating system.

In the tutorials in this chapter you will work with simple linear recursive methods to get a feeling for recursion and to compare recursive methods to methods that do the same thing with simple loops.
TUTORIAL 7A—CREATING A RECURSIVE METHOD

In this tutorial you will create a recursive method to make an airplane taxi from the end of a runway to a spot near an airplane hangar. Let’s start with the specifications for the world. You must set up a scene with an aircraft that has just landed at an airport. The aircraft must taxi from the end of the runway to a spot near an airplane hangar. The task is to create a method that will enable an object to move from wherever it is when the method is called to a specific spot. In this case the aircraft should taxi to within 1 meter of the part of the airport named garage02.

Immediately there are a few things that will come to mind when a good programmer is presented with this situation:

1. This seems to be the kind of programming project that is part of a larger project. You will be building a short method for a very specific task. A good programmer in this situation will think of reusable code. The method could be constructed with a target parameter instead of hard coding garage02 into the method.

2. The specifications do not call for you to build an airport, so either one already exists, or someone else will create one as a separate part of the project.

3. What’s an aircraft? The vehicles folder in the local Alice object gallery has several airplanes, including a biplane, a jet, a navy jet, and a seaplane. The gallery also has a helicopter and a blimp, which might be considered aircraft. In a situation like this, the programmer would need to go back to whomever provided the specifications to get more information.

4. Finally, there are no internal specifications for the method, only functional specifications. This means that the specifications tell the programmer what the new method should do, but not how the method should do it. They don’t say anything about what techniques can be used inside the method. The specifications don’t say, for example, “use a loop to...”. This gives the programmer more flexibility in designing the internal workings of the method, but a good programmer will ask about this to be sure that there really is some flexibility here. Sometimes specifications don’t tell you everything.

So, after reading the specifications a good programmer will put together a set of questions for the person who provided them. Communication like this among the members of a team working on a software development project is important. In this case, the programmer
would ask which airport should be used, which aircraft should be used, and whether any particular techniques should be used or avoided inside the method. The answers to these questions are that the airport in the buildings folder in the local Alice object gallery should be used; the biplane should be used as the aircraft; and there are no restrictions on the techniques to be used in the method, except that recursion should be used.

Now that things are a little clearer, you can begin. In a real situation a programmer would probably create an iterative method or simply use the existing primitive move to method for this project, but your purpose in this tutorial is to experiment with recursion.

**SETTING THE SCENE**

First you need to build the scene for the new world.

1. Start Alice and open a new world with a **grass** background.
2. Click the **ADD OBJECTS** button to enter Scene Editor mode, and find the Airport object class in the Buildings folder within the Local Gallery.
3. Click the **Airport** tile, but pause for moment before actually adding it to the world. Notice that the Airport class information window, shown in Figure 7-7, says that the size of the object is 753 kilobytes, and that it has 44 parts. This is a large and complex object. Also notice that there are no methods listed in the window, so the object only includes the standard primitive methods. Click the **Add instance to world** button to add an airport to the world.

**FIGURE 7-7:** The Airport object class information window

![Airport Object Class Information Window](image)

4. Note that a tile for the airport appears in the Object tree, and the view of the airport in the World window will probably look something like Figure 7-8. The view is as if you were standing on the ground at the airport.
5. You need to position the camera to get a better view of the airport. Using the blue camera control arrows, move, tilt, and pan the camera so that the view of the airport looks more like Figure 7-9, but without the red plane. The trick is to move the camera and not the airport. This will probably take a few minutes to do unless you are very familiar with the Scene Editor. Remember the magic of the Undo button, which can be used if you are unhappy with one of your moves. Both the right end of the runway and the hangars on the left should be visible in the window when you are done.

6. Next, add a biplane to the world, and position it near the right end of the runway, similar to the way the red plane is shown in Figure 7-9. Your biplane won’t be red; the one in the picture is red so that you can see it better in the printed image. The biplane should look as if it just landed and is about ⅔ of the way down the runway. You will need to move and turn the biplane.

7. Now that the scene is ready, click the **DONE** button to exit Scene Editor mode, and save the world with the name **airport setup**. This will protect you from needing to re-create the scene in case something goes wrong.
CODING THE RECURSION

Now that the scene is ready, you can create a recursive taxi method to make the biplane move as desired. The pseudo-code for the *bunny.hop to carrot* method earlier in this chapter can serve as a template for a new taxi method. Here is the pseudo-code modified for the taxi method:

```java
biplane.taxi (target)
Start
If ( [biplane.distance to target] > 1 meter)
{
    biplane.point at target
    biplane.move forward 1 meter
    biplane.taxi (target)
}
Stop
```

It seems fairly straightforward, so let’s create the method. You can start by creating the method and putting the *If/Else* command in place.

1. Select the *biplane* tile in the Object tree and the *methods* tab in the Details area.
2. Click the create new method button, type taxi as the method name, and then click the OK button. The method biplane.taxi should appear in the Editor area.

3. The method will need a parameter named target with the data type object. Click the create new parameter button, type the name target, make sure that object is selected as the data type, and then click the OK button.

4. Drag an If/Else tile from the bottom of the Editor area and drop it in the new taxi method in place of Do Nothing. Select true as the condition. Your method should now look like Figure 7-10.

5. The Boolean condition needs to be a comparison of two values using the greater than operator. Select world in the Object tree and the functions tab in the Details area. One of the functions listed under math is $a > b$. Drag a copy of this tile and drop it into the IF/Else instruction tile in place of true. Select 1 as the value for $a$ and 1 as the value for $b$ from the short menus that appear.

6. You need to replace the first 1 with the biplane’s distance to function. Select biplane in the Object tree and drag and drop a copy of the distance to function in place of the first number 1 in the condition in the IF/Else tile.

7. When the menu appears asking for an object, select expressions, and then target. The instruction should now match the pseudo-code above.
Now you need to add the instructions that belong inside the *If/Else* structure. The pseudo-code shows that there are three of them: `biplane.point at (target)`, `biplane.move forward 1 meter`, and `biplane.taxi (target)`. You can add them in the order in which they will occur.

1. Select the **methods** tab in the Details area and drag and drop a copy of the `biplane turn to face` instruction tile into the *If/Else* tile in the Editor area in place of the words *Do Nothing* between *If* and *Else*. When the menu appears asking you for the target, select *expressions*, and then *target*. Also, click *more* and change the style to *abruptly*.

2. Next drag a copy of the `biplane move` tile from the methods tab in the Details area and drop it into the new method just below the `biplane turn to face` instruction. Select *forward* as the direction and *1 meter* as the amount; then click *more* and change the style to *abruptly*. Note that when you play the finished world, this value of 1 meter will affect the speed of the plane as it taxis. You can come back to this instruction and change the amount to make the plane move more quickly or more slowly.

3. Now drag a copy of the `taxi target` tile from the methods tab in the Details area and drop it into the new method just below the `biplane move forward` instruction. When the menu appears, select *expressions* and then *target*. When you do this a Recursion Warning window similar to the one in Figure 7-11 will appear, warning you that you are about to create a recursive method.

4. Click the button labeled **Yes I understand what I am doing**, and the instruction tile will pop into place in the method.

5. Finally, the method will work more smoothly if the `biplane turn to face` and `biplane move` instructions happen at the same time. Drag a **Do together** tile from the bottom of the Editor area and drop it into the method between the `biplane move forward` instruction and the `biplane.taxi` instruction; then drag the `biplane turn to face` and `biplane move` instruction tiles into the **Do together** tile. The method is complete and should now be similar to Figure 7-12.
To test the method, you need to call it from world.my first method, so world.my first method needs to be modified to call biplane.taxi. Also, let’s add an instruction to make the biplane turn to face the camera after it moves. There are two reasons for this: first, it will help to see if everything is normal after the taxi method is finished, and second it looks a little more interesting. This can only be done outside of the taxi method; otherwise it would interfere with taxi meeting its specifications. In a sense, world.my first method is being used as a testing shell for biplane.taxi.

1. First, select world.my first method in the Editor area.
2. Drag a copy of the taxi target tile from the methods tab in the Details area and drop it into world.my first method in place of Do Nothing. When the menu appears, select airport, then runwayandParking, then garage02. Garage02 is the name for the part of the airport object to which the biplane should move. Actually, it is a sub-part of the runwayandParking object, which is a sub-part of the airport object.
3. Drag and drop a copy of the biplane.turn to face tile in the methods area on the Details tab into world.my first method below the biplane.taxi instruction. When the menu appears, select camera as the target.
4. Now save the world with the name recursive taxi.

You can now test the world, and see if it works as expected. When you do so you may notice that the biplane passes through part of the airport terminal. Exercise 3 at the end of this chapter deals with this. Try the world a few times to see if it works. If it does not work properly, then it’s time to track down the error and fix it.
TUTORIAL 7B—CONVERTING AN EXISTING ITERATIVE METHOD TO RECURSION

In addition to creating recursive methods from scratch, it’s useful to be able to rewrite an iterative method as a recursive method, and vice-versa. In this tutorial you will modify the sail to method in the sail to object world created as part of Tutorial 5C, changing the sail to method from an iterative method into a recursive method. In the next tutorial, you will change it back to an iterative method.

1. Start Alice and open the sail to object world from Tutorial 5C, or open a copy of the world from the CD that comes with this book.
2. Before doing anything else, save the world with the name recursive sail to so that your changes will not mess up the original world.
3. Select the sailboat in the Object tree and the methods tab in the Details area, if necessary.
4. Click the edit button next to the sail to method tile; the sailboat.sail to method should appear in the Editor area, as shown in Figure 7-13.

FIGURE 7-13: The iterative sail to method in the Editor area
The pseudo-code for the existing iterative method and the pseudo-code for a new recursive method that does the same thing are shown below:

```plaintext
Sailboat.sail to (target)
While( not [sailboat is within 5 meters of target] )
{
  Do together
  {
    sailboat.turn to face target
    sailboat.move forward 2 meters
  }
}
Stop

Sailboat.sail to (target)
Start
If ( not [sailboat is within 5 meters of target] )
{
  Do together
  {
    sailboat.turn to face target
    sailboat.move forward 2 meters
  }
  sailboat.sail to (target)
}
Stop
```

Comparing the two versions of the `sail to` method, you can see that an `IF/Else` instruction is needed instead of a `While` loop, and a recursive call needs to be added inside the method. Let's continue with our task with the next set of steps.

1. Drag an `IF/Else` instruction tile from the bottom of the Editor area and drop it in the method below the `While` instruction tile. Select `true` as the condition from the menu that appears.

2. Drag the Boolean condition tile that starts with the word `not` from the `While` tile and drop it into the `IF/Else` tile in place of `true`. If you have done this correctly, then your method should look like Figure 7-14. If it is not correct, then use the `Undo` button to back up and try again.
3. Drag the `Do together` tile with the `sailboat turn to face target` and `sailboat move forward 1 meter` instructions from the `While` tile, and drop it into the `If/Else` tile in place of `Do Nothing` between `If` and `Else`.

4. Now you are ready to add the recursive call to the method. Drag a copy of the `sail to target` method tile from the methods tab in the Details area, and drop it in the `IF/Else` instruction below the `Do together` tile but above `Else`. Select `expressions`, and then `target` from the menu that appears. When you do this a Recursion Warning window similar to the one in Figure 7-11 back in Tutorial 7A will appear. The window is warning you that you are about to create a recursive method and asks you if this is what you intended.

5. Click the button labeled `Yes, I understand what I am doing`, and the instruction tile will appear in the method.

6. The last thing you need to do is to get rid of the `While` tile, which should now be empty and is no longer needed. Either right-click the `While` tile and `select` delete, or drag the tile to the trash can.

7. The changes to the method are now complete, and it is ready for testing. The method should look like Figure 7-15. Save the world before continuing.
Play the world and see how your new method works. If all is well, it should appear to function no differently than the old sail to object world did. The changes are all internal and will not really be seen by the user unless a problem occurs.
CHAPTER SUMMARY

This chapter consisted of a discussion of recursive algorithms, including a comparison of recursion and iteration, a look at linear and exponential recursion, and a look at infinite and conditional recursion, followed by three tutorials. The discussion of recursive algorithms included the following:

- Generally, something is said to be recursive if each of the parts that make up the thing have a structure—in other words, a design or pattern—that repeats the structure of the whole thing itself.

- In computer programming, an algorithm that calls itself is said to be recursive.

- An iterative process is one that uses a loop to repeat a set of instructions, so iteration is just another name for looping. In many cases, either iteration or recursion can be used to do the same thing.

- Overhead is the extra CPU time and extra space in a computer’s memory needed to manage a method as it runs. The computer must set up a section in its memory every time a new method is called—even if the new method is a copy of an existing method.

- Often it is better to use iteration than recursion for methods that can be written using a simple loop because of the overhead associated with recursion.

- Linear recursion occurs when a method calls itself only once each time through the method.

- Exponential recursion occurs when a method calls itself more than once in each pass through the method. Exponentially recursive algorithms are often far more efficient than iterative solutions to do the same thing because they can repeatedly break a problem into several smaller parts, then attack those smaller parts in a way that an iterative algorithm cannot.

- For simple tasks that can be described with linear recursion, iteration seems to be better because of the overhead of recursion, but for complex problems that can be broken down into smaller parts, exponential recursion usually works much better, even allowing for the overhead of the recursion.

- Infinite recursion means that an algorithm calls itself repeatedly without stopping. Recursion that is not infinite is called conditional recursion since the algorithm must contain some condition that will stop the recursion.

- The condition that stops the recursion is called the base condition or the base case. A properly structured recursive algorithm should contain a Boolean expression in a selection sequence to test for the base condition.

In Tutorial 7A, you created a new world with a recursive method.

In Tutorial 7B, you converted an existing iterative method into a recursive method.
REVIEW QUESTIONS

1. Define the following terms:
   - base case
   - exponential recursion
   - linear recursion
   - base condition
   - infinite recursion
   - overhead
   - base step
   - iteration
   - recursion
   - conditional recursion
   - iterative process
   - recursive method

2. What is the difference between recursion and iteration?

3. Why will a program with infinite recursion eventually crash on a computer system?

4. In a case where both iterative and recursive solutions to a programming problem exist, how can a programmer tell whether to use iteration or recursion?

5. The recursive biplane.taxi method in Tutorial 7B works only with the biplane. How can this be modified so that it will work with any object?

6. The following joke was all the rage in 4th grade classrooms last year. Explain how this is related to recursion.
   “There were three men in a boat, Joe, Pete and Repeat. Joe and Pete fell out. Who was left?”

7. Write both iterative and recursive methods in pseudo-code that will multiply any two positive integers using only addition.

8. It was recently discovered that the famous graphic artist M. C. Escher used the “Droste effect” in at least one of his drawings. See if you can find out on the Internet what the Droste effect is, how it is related to recursion, and why it’s called the Droste effect.

9. Assume that a computer student wants to impress his father, who is a computer security expert for the CIA, by writing a program like the one below. The parameter for the program is the network address of a computer system.

   Search for dad's account at (computer system)
   Start
   If dad has an account on this computer
      Print "Hello dad, I found your computer."
   Else
      Find two other computers connected to this one
      Search for dad's account at (first computer)
      Search for dad's account at (second computer)
   Stop
The preceding code is enough to cause some issues between father and son; however, assume that the student plays with it some more and then makes a mistake, leaving out the If/Else instruction so that the algorithm looks like this:

```
Search for Dad's account at (computer system )
Start
Print "Hello Dad, I found your computer."
Find two other computers connected to this one
Search for Dad's account at (first computer)
Search for Dad's account at (second computer)
Stop
```

a. What kind of recursion is within this program?
b. Assuming that the student knows enough systems programming to make the instructions work, what will the program actually do?
c. If it takes 10 seconds for the algorithm to run, including the time it needs to connect to other computers, how many copies of the program will be running 1 minute after the program first starts? After 2 minutes? After 1 hour? After 6 hours?
d. What would something like this do to the Internet?
e. What sentence did US District Court Judge Howard G. Munson give Cornell University graduate student Robert Tappan Morris when he was convicted of doing something similar to this in November, 1988?

10. An important technique in mathematics is proof by induction. We start a proof by induction by proving that if something is true for one number, then it must be true for the next number also. In other words, if what we are trying to prove is true for $n$, then it must be true for $n+1$. This is called the inductive step in the proof. Next we show that it is true for the number 1. This is called the base step in the proof. Putting the base step and the inductive step together, we can then see that the item in question must be true for all positive integers because if it is true is true for 1, then it must be true for 1+1, then for 2+1, then 3+1, and so on. How is a proof by induction similar to conditional linear recursion?

EXERCISES

1. The Alice people gallery has hebuilder and shebuilder classes of objects that include a walk method. Create a simple Alice world with a few objects of your choice, including a character that you create with hebuilder or shebuilder, and create both iterative and recursive methods to make the character walk to a target object.

2. There occurs in the story Alice’s Adventures in Wonderland a section in which Alice grows smaller, and then bigger. Create an Alice world with aliceLiddel standing between a tall flower and a short flower and with an infinitely recursive method to keep repeating the following process:
   a. She touches the tall flower and then shrinks.
   b. She touches the short flower and then grows.
      Hint: Use the resize primitive method.

4. The biplane in the taxi world created in Tutorial 7A moves right though part of the airport to get to its spot near the hangar. See if you can correct this by adding a cone to the world from the shapes gallery, positioning the cone someplace on the tarmac, then have the plane move twice, first to the cone, and then from the cone to its final spot. Once it works you can then make the cone invisible by changing its opacity property.

5. In Tutorial 7B you converted an existing iterative method into a recursive method. Starting with the recursive sail to world from the end of that tutorial, or the copy of it on the disk that comes with this book, convert the recursive taxi method in the world back to an iterative method.

6. A recursive algorithm can be written to draw a spiral, starting in the middle and expanding outward with each revolution. A similar algorithm can be written to draw a “spirally increasing pseudo-square” like that shown in Figure 7-16. Such an algorithm could be used for a search pattern to search for a lost object, or for a robotic lawn mower that could be left in the middle of a large field and then programmed to start mowing the field from the center outwards. Create a “recursive Zamboni” Alice world with a snow background and a Zamboni with a recursive method to make the Zamboni follow the “spirally increasing pseudo-square” pattern described above. A Zamboni is the machine that is used to clean an ice skating rink, such as at a hockey game between periods. The Alice vehicle gallery has a Zamboni.

7. The Fibonacci sequence is one of the most commonly found patterns in all of nature. The sequence starts with 0 and 1, then each term in the sequence is generated by adding the previous two terms. The first few terms in the sequence are 0, 1, 1, 2, 3, 5, 8, 13, 21, and 34. Numbers that appear in the sequence are called Fibonacci numbers.
   a. Write a recursive algorithm in pseudo-code to generate the Fibonacci sequence.
   b. Write an iterative algorithm in pseudo-code to generate the Fibonacci sequence.
   c. Which algorithm would probably work better on a computer, and why?
   d. Write an algorithm in pseudo-code to test a number to see if it is a Fibonacci number.
   e. Create an Alice world with Count penguin and the Fibonacci penguin. Count penguin will simply start counting slowly when the world starts by saying each number. The Fibonacci penguin will jump up and down and flap its wings while saying “Fibonacci, Fibonacci, (n) is Fibonacci!” when Count penguin says a number that is part of the Fibonacci sequence. The Fibonacci penguin should say the actual number in place of $(n)$. 
8. **Images generated by certain recursive algorithms are sometimes called fractal images.**
   a. Draw the figure generated by the algorithm below:

   ```plaintext
   Start with a square
   Fractal (square)
   Start
   Divide the square into four smaller squares
   Fill in the upper-left square and the lower-right square
   If the squares are still big enough to draw in, then
     Fractal (upper-right square)
     Fractal (lower-left square)
   Stop
   ```

   b. Using pseudo-code, write your own recursive algorithm to generate a fractal image,
   c. Search the Internet for fractal geometry to see some very interesting fractal images.

9. **Here is an example of a very short recursive story (which was already discussed in the main part of the chapter):**

   ```plaintext
   Once upon a time a young princess was sitting with her father, the king. She said to her father, “My Royal Father, can you read me a story?”
   “Yes my Royal Daughter” said the king, as he opened the Royal Storybook. He began to read, “Once upon a time a young princess was sitting with her father, the king. She said to her father, ‘My Royal Father, can you read me a story?’ …”
   ```

   Create an Alice world that contains a very short recursive story. Start with either an outline or storyboard, and consider how camera moves can add to the very short story. The Alice Local Gallery’s Objects folder contains Book, Monitor, Picture Frame and TV classes. One of these may prove helpful in your story.

10. **One well known puzzle that has a recursive solution is the “Towers of Hanoi” puzzle, which can be seen on the Internet at [http://www.mazeworks.com/hanoi/](http://www.mazeworks.com/hanoi/). See if you can write a simple recursive algorithm in pseudo-code as a solution to the problem. Your solution should work no matter how many discs there are. You should be warned however, that even though the algorithm is only a few instructions long, it could take a while to figure it out.**