Chapter 8

Recursion

What is in This Chapter ?

This chapter explains the notion of ***recursion*** as it appears in computer science. Recursion usually involves thinking in a different manner than you may be used to. It is a very powerful technique, however, that can simplify algorithms and make your life easier ... once you understand it. We discuss briefly the idea of using recursion for simple **math-based problems** and for simple problems which are often found in computer science. We then discuss a couple of **graphics problems** that are solved using recursion and then conclude with a couple of **search-related** recursive examples. A more thorough discussion of recursive techniques is covered in further courses.

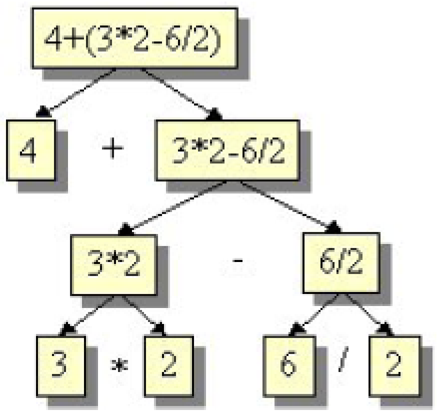
8.1 Thinking Recursively

Recall in the first chapter, we discussed the notion of "divide-and-conquer" in regards to breaking a problem down into smaller, more manageable, modular pieces called functions or procedures.

Sometimes when we break a problem down into smaller pieces of the same type of problem, we are simply reducing the amount of information that we must process, since the problem is solved in the same manner ... it is just a smaller version of the same problem.

For example, there are many real-world examples where we often break problems into smaller ones of the same type in order to solve the bigger problem:

1. Jigsaw puzzles are solved in "steps": border, interesting portion, grass, sky, etc..
2. Math problems are broken down into smaller/simpler problems
3. Even climbing stairs eventually breaks down to climbing one step at a time.



Such problems can be solved *recursively*.

> ***Recursion** is a divide-and-conquer technique in which the problem being solved is expressed (or defined) in terms of smaller problems of the same type.*

The word ***Recursion*** actually comes from a Latin word meaning "a running back". This makes sense because recursion is the process of actually "going off" and breaking down a problem into small pieces and then bringing the solutions to those smaller pieces back together to form the complete solution. So ...

- recursion **breaks down** a complex problem into **smaller sub-problems**
- the sub-problems are **smaller** instances of the **same type** of problem.

So, when using recursion, we need to consider how to:

1. break the problem down into smaller sub-problems
2. deal with each smaller sub-problem
3. merge the results of the smaller sub-problems to answer the original problem

In fact, it is actually easier than this sounds. Most of the time, we simply take the original problem and **break/bite off** a small piece that we can work with. We simply keep biting off the small pieces of the problem, solve them, and then merge the results.

Consider the simple example of computing the factorial of an integer. You may recall that the function works as follows:

```
0! = 1
1! = 1
2! = 2x1
3! = 3x2x1
4! = 4x3x2x1
5! = 5x4x3x2x1
etc..
```

So, the function is defined non-recursively as follows:

```
1                                            if N = 0
N! = N x (N-1) x (N-2) x ... x 3 x 2 x 1     if N ≥ 1
```

If you were asked to write a factorial function, you could do so with a FOR loop as follows:

```java
public static int factorial(int n) {
   int    answer = 1;

   for (int i=2; i<=n; i++)
      answer = answer * i;

   return answer;
}
```

If you examine the code carefully, you will see that it provides the answer as defined earlier. This code is simple and straight-forward to write.

The factorial problem, however, can also be defined ***recursively*** as follows:

```
1                     if N = 0
N! = N x (N-1)!       if N ≥ 1
```

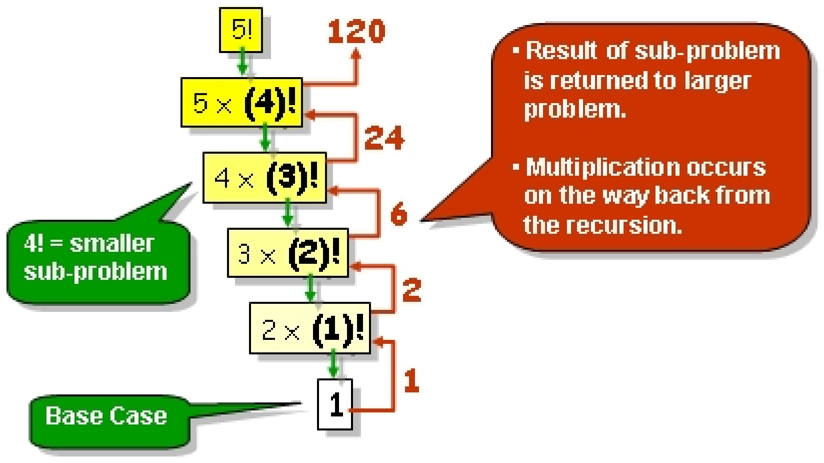
Notice how **N!** is expressed as a function of the smaller sub-problem of **(N-1)!** which is of the same type of problem (i.e., it is also a factorial problem, but for a smaller number).

So, if we had the solution for **(N-1)!** … let's say it is **S** … then we could use this to compute the solution to **N!** as follows: **N!** = **N** x **S**

However, how do we get the solution for **(N-1)!** ?
We use the same formula, setting **N** to **N-1** as follows:

```
(N-1)! = (N-1) x ((N-1)-1)!    =   (N-1) x (N-2)!
```

Similarly, we can recursively break down **(N-2)!** in the same way. Eventually, as we keep reducing **N** by **1** each time, we end up with **N=0 or 1** and for that simple problem we know the answer is **1**. So breaking it all down we see the solution of **5!** as follows:


5!
5 × (4)!
4 × (3)!
3 × (2)!
2 × (1)!
1
120
24
6
2
1
4! = smaller sub-problem
Base Case
• Result of sub-problem is returned to larger problem.
• Multiplication occurs on the way back from the recursion.

If you were asked to write such a **recursive** factorial function, you could do so as follows:

```
public static int factorialRecursive(int n) {
   int    answer;

   if (n == 0)
      answer = 1;                             // if n=0, f = 1
   else
      answer = n * factorialRecursive(n-1);   // if n>=1, f = n*f(n-1)

   return answer;
}
```

First, you will notice that there is no FOR loop anymore. Second, you'll notice that the function is calling itself. So, it is a very different way of thinking … isn't it ? Looking at the comments, however, you will notice that the code follows almost exactly from the recursive formula.

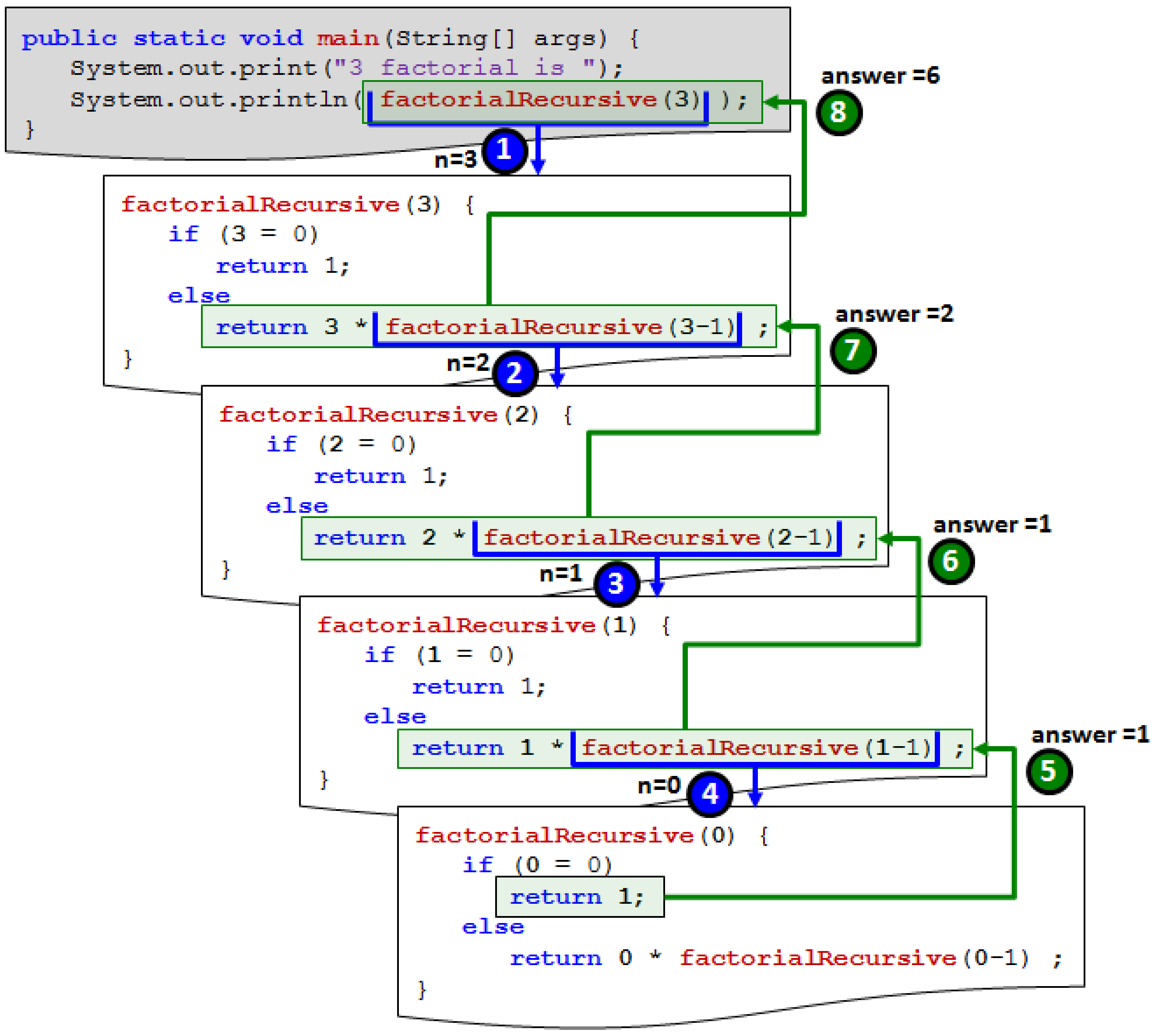
The **factorialRecursive** function repeatedly calls itself over and over again with smaller values of **n** each time until finally **n** has the smallest value of **0**, in which case the code stops calling itself recursively.

The function can actually be simplified by removing the answer variable as follows:

```
public static int factorialRecursive(int n) {

   if (n == 0)
      return 1;                          // if n=0, f = 1

   return n * factorialRecursive(n-1);   // if n>=1, f = n*f(n-1)
}
```

A recursive function is easily identifiable by the fact that a function or procedure calls itself. The "stopping situation" is called the ***base case*** of the problem. In general, a recursive function (or procedure) may have multiple base cases (i.e., multiple stopping conditions).

To understand the inner-workings of the function as it calls itself, consider the following diagram which shows how the control passes from one function call to the next when computing the factorial of **3**. There are **8** main "steps" of the recursion as shown numbered. You will notice that each time the **factorialRecursive** function is called, the value of **n** decreases (i.e., steps 1 through 4). Finally, when **n** matches the base case (i.e., **0**), the value of **1** is returned to the previous function call (i.e., step 5). This returned value is then used in the expression **1*1** and the result of **1** is returned to the previous function call (i.e., step 6). This value is then used in the expression **2*1** and the result of **2** is returned to the previous function call (i.e., step 7). This is then used in the expression **3*2** and the result of **6** is returned to the main method (i.e., step 8). The result of 6 is then displayed.



```
public static void main(String[] args) {
    System.out.print("3 factorial is ");
    System.out.println( factorialRecursive(3) );
}
```

```
factorialRecursive(3) {
    if (3 = 0)
        return 1;
    else
        return 3 * factorialRecursive(3-1) ;
}
```

```
factorialRecursive(2) {
    if (2 = 0)
        return 1;
    else
        return 2 * factorialRecursive(2-1) ;
}
```

```
factorialRecursive(1) {
    if (1 = 0)
        return 1;
    else
        return 1 * factorialRecursive(1-1) ;
}
```

```
factorialRecursive(0) {
    if (0 = 0)
        return 1;
    else
        return 0 * factorialRecursive(0-1) ;
}
```

If you were to compare the non-recursive solution with the recursive solution ... which do you find to be simpler ?

```
public static int factorial(int n) {
    int    answer = 1;

    for (int i=2; i<=n; i++)
        answer = anwser * i;

    return answer;
}
```
	```
	public static int factorial(int n) {

	    if (n == 0)
	        return 1;

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

You may feel that the non-recursive version is conceptually simpler, even though the code is longer. In this particular example, you are probably right. So the question that arises is ... "Why should we use recursion ?".

Here are 4 reasons:

1. Some problems are **naturally** recursive and are easier to express recursively than non-recursively. For example, some math problems are defined recursively.

2. Sometimes, using recursion results in a simpler, **more elegant solution**.

3. Recursive solutions may actually be **easier to understand**.

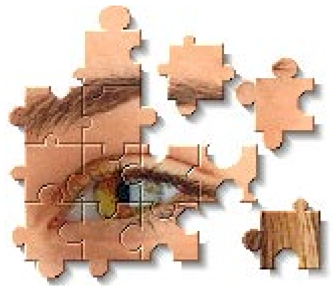
4. In some cases, a recursive solution may be the **only way to approach** a seemingly **overwhelming problem**.

As we do various examples, the advantages of using recursion should become clear to you.

How do we write our own recursive functions and procedures ? It is important to remember the following two very important facts about the sub-problems:

- must be an instance of the **same kind** of problem
- must be **smaller** than the original problem

The trick is to understand how to "bite off" small pieces of the problem and knowing when to stop doing so. When we write the code, we will usually start with the base cases since they are the ones that we know how to handle. For example, if we think of our "real world" examples mentioned earlier, here are the base cases:



1. For the **jigsaw puzzle**, we divide up the pieces until we have just a few (maybe 5 to 10) pieces that form an interesting part of our picture. We stop dividing at that time and simply solve (by putting together) that simple *base case* problem of this small sub-picture. So the base case is the problem in which there are only a few pieces which all part of some identifiable portion of the puzzle.

2. For the **math problems**, we simply keep breaking down the problem until either there is a simple expression remaining (e.g., 2 + 3) or until we have a single number with no operations (e.g., 7). These are simple base cases in which there is either one operation to do or none.

3. For the **stair climbing problem**, each stair that we climb brings us closer to solving the problem of climbing all the stairs. Each time, we take a step up, the problem is smaller, and of the same type. The base case is our simplest case when there is only one stair to climb. We simply climb that stair and there is no recursion to do at that point.

So then … to write recursive methods, we first need to understand "where" the recursion will fit in. That is, we need to understand how to reduce the problem and then merge the results. It is important to fully understand the problem and the kind of parameters that you will need as the function/procedure gets called repeatedly. Make sure you understand how the problem gets smaller (i.e., usually the parameters change in some way). Then, you can implement the simple base cases and add in the recursion. Let us look at a few examples.

8.2 Math Examples

Recursive math examples are easy to program because we are often given the recursive definition directly and then it easily transfers into a program.

Example:

For example, consider computing the monthly payments that a person would have to make in order to pay back a mortgage on his/her home. Consider the following notation:

- **amount** = mortgage amount borrowed (e.g., $130,000)
- **rate** = annual interest rate as a percentage (e.g., 3.5%)
- **months** = length of time for the mortgage to be paid off (e.g., 300 months for a 25 year mortgage)
- **payment(amount, rate, months)** = monthly payment to pay off mortgage (e.g., $647.57)

We would like to calculate the value for the **payment** function. Here is the function and its parameters as defined:

```
public static double payment(float amount, float rate, int months) {

}
```

If we borrow money for **0** months, then the whole point of borrowing is silly. Assume then that **months** > 0. What if **months** = 1 ... that is ... we want to pay off the entire mortgage one month after we borrowed the money ? Then we must pay one month interest and so the value of the **payment** function should return:

amount + **amount** * (**rate**/12) … or … **amount** * (1 + (**rate**/12))

which is one month of interest on the original loan amount. We divide by 12 since the rate is per year and we want only 1 month of interest. For a 2-month term, the payment would be:

[amount * (1 + (**rate**/12))**]** * (1 + (**rate**/12))

and for a 3-month term:

{[amount * (1 + (**rate**/12))**]** * (1 + (**rate**/12)))**}** * (1 + (**rate**/12))

So you probably notice that the interest to be paid each month is compounded monthly...so we pay interest on interest owed and this grows larger each month. Don't worry if you don't understand the math here … because this is not a "math course".

There are a few ways to compute mortgage payments, but we will use the following formula so that we can practice our recursion. The formula for computing the payment is actually recursive and is defined as follows:

$$p(a, r, m) = \frac{a}{\left\{ \left[\frac{a}{p(a, r, m-1)} + \frac{1}{(1 + r/12)^{m}} \right] \right\}}$$

Here, **a** = amount, **r** = rate, **m** = months. Looking at the formula, do you see that the problem is recursive ? The result of the payment function **p(a,r,m)** depends on the result of the payment function **p(a,r,m-1)** for one less month. It is easy to see that the problem for one less month is the same problem (i.e., same function) and that the problem is smaller.

Now let us determine the base case (i.e., the simplest case). Well, the **amount** and **rate** do not change throughout the computation. Only the **month** value changes ... by getting smaller. So the base case must be the stopping condition for the **months** being decreased. This base case is the simplest case which follows from the definition that we discussed earlier:

```
public static double payment(float amount, float rate, int months) {

    if (months == 1)
        return amount * (1 + rate/12);

}
```

That wasn't so bad. Now what about the recursive part ? It follows from the formula:

```
public static double payment(float amount, float rate, int months) {

    if (months == 1)
        return amount * (1 + rate/12);

    return amount / ((amount / payment(amount, rate, months-1)) +
                          (1/Math.pow(1 + rate/12, months)));
}
```

As you can see, the code is easy to write … provided that we have a recursive definition to begin with.

Example:

Imagine now that we have a teacher who wants to choose 10 students from a class of 25, to participate in a special project. Assuming that each student is unique in his/her own way ... how many different groups of students can be chosen ?

This is a classic problem using binomial coefficients that is often encountered in combinatorics for determining the amount of variety in a solution, giving insight as to the complexity of a problem. In general, if we have **n** items and we want to choose **k** items, we are looking for the solution of how many groups of **k**-elements can be chosen from the **n** elements.

The expression is often written as follows and pronounced "**n** choose **k**": $\binom{n}{k}$

Assuming that **n** is fixed, we can vary **k** to obtain a different answer.
The simplest solutions for this problem are when **k** = 0 or **k** = **n**. If **k** = 0, we are asking how many groups of zero can be chosen from **n** items. The answer is **1** ... there is only one way to

choose no items. Similarly, if **k** = **n** then we want to choose all items ... and there is only one way to do that as well. Also, if **k** > **n**, then there is no way to do this and the answer is zero (e.g., cannot choose 10 students from a group of 6).

Otherwise, we can express the problem as a recursive solution by examining what happens when we select a single item from the **n** items. Imagine therefore that we select one particular student that we definitely want to participate in the project (e.g., the teacher's "pet").

We need to then express the solution to the problem, taking into account that we are going to use that one student. In the remainder of the classroom, we therefore have **n-1** students left and we still need to select **k-1** students. Here is what is left to do →

$$\binom{n-1}{k-1}$$

The other situation is that we decide that we definitely DO NOT want that particular student to participate in the project (e.g., he's been a "bad boy" lately). In this case, if we eliminate that student, we still need to select **k** students, from the **n-1** remaining. Here is what is left to do →

$$\binom{n-1}{k}$$

Now, I'm sure you will agree that if we examine one particular student ... then we will either select that student for the project or not. These two cases represent all possible solutions to the problem. Therefore, we can express the entire solution as:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Yay! This is a recursive definition. The problem is of the same type and gets smaller each time (since **n** gets smaller and **k** does also, in one case). This problem is interesting because there are two recursive calls to the same problem. That is, the problem branches off twice recursively.

So then, to write the code, we start with the function signature:

```
public static int studentCombinations(int n, int k) {
}
```

What about the base case(s) ? Well, we discussed the situations where **k** = **n**, **k** = 0, **k** > **n**, so these are the simplest cases:

```
public static int studentCombinations(int n, int k) {
   if (k > n)
      return 0;
   if (k == 0)
      return 1;
   if (k == n)
      return 1;
}
```

As before, the recursive cases are easy, since we have the definition.

The remainder of the code will be interesting … as the function will call itself twice:

```java
public static int studentCombinations(int n, int k) {
   if (k > n)
      return 0;
   if (k == 0)
      return 1;
   if (k == n)
      return 1;

   return studentCombinations(n-1, k-1) + studentCombinations(n-1, k);
}
```

Not too bad ... was it ? Do you understand how the recursion will eventually stop ?

8.3 Graphical Examples

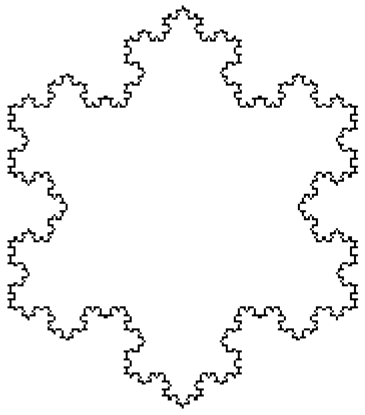
Recursion also appears in many graphical applications. For example, fractals are often used in computer graphics and in computer games, trees are often created and displayed using recursion.

A ***fractal*** *is a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole. (wikipedia)*

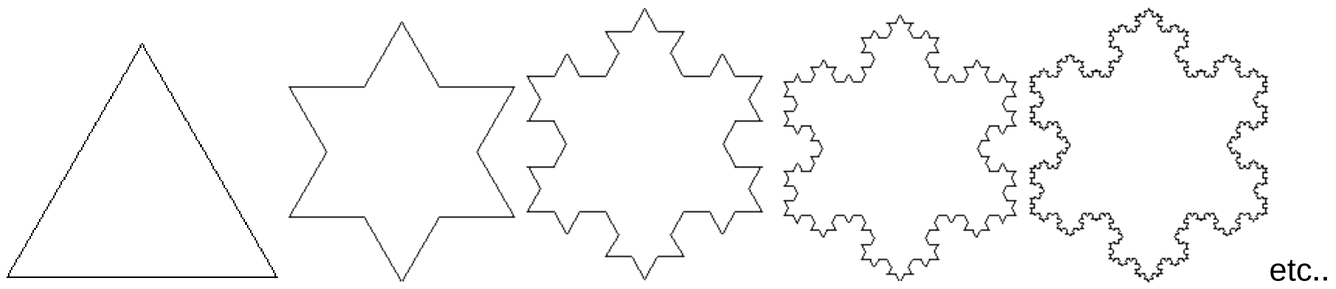
So, a fractal is often drawn by repeatedly re-drawing a specific pattern in a way that the pattern becomes smaller and smaller, but is of the same type.

Example:

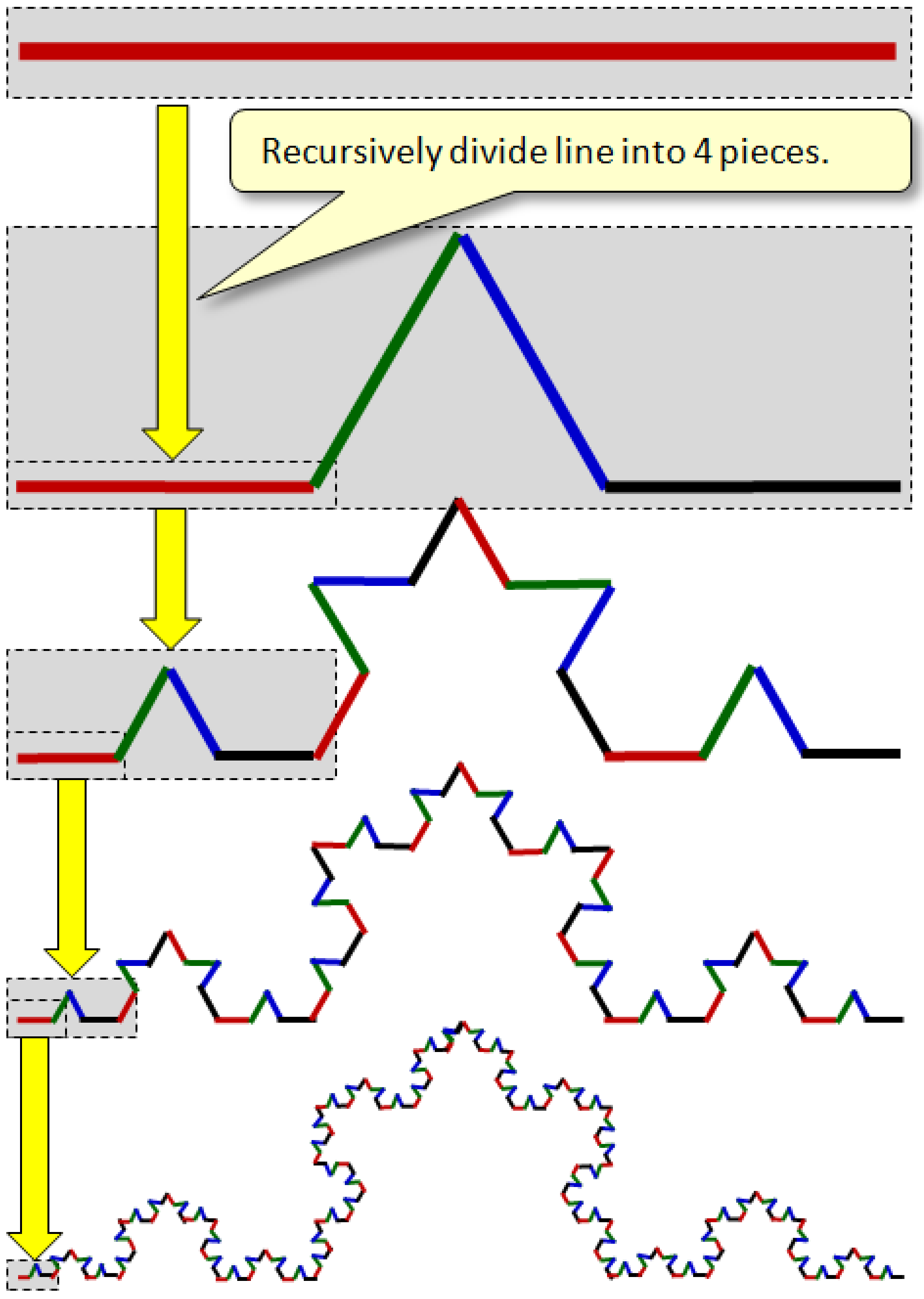
Consider how we can draw a snowflake. See if you can detect the recursion in the following snowflake (called the **Koch Snowflake**):



It may not be easy to see. However ... if we were to examine the process of creating this snowflake recursively, notice how it can be done by starting with an equilateral triangle:



The recursion happens on each side of the triangle. Consider a single horizontal edge. We can recursively alter the edge into 4 segments as follows:



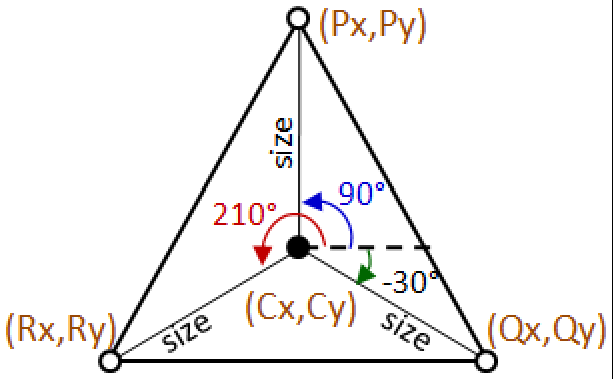
To accomplish this, we need to understand how a single edge is broken down into 4 pieces. Then we need to understand how this gets done recursively. Notice above how each recursive edge gets smaller. In fact, during each step of the recursion, the edge gets smaller by a factor of 3. Eventually, the edge will be very small in length (e.g., 2) and the recursion can stop, since breaking it down any further will not likely make a difference graphically.

The additional value of **180°** in the **cosine** function is used because of the reverse-**y**-coordinate system that is used in computer graphics. In mathematics, the origin **(0,0)** of the coordinate system is in the **bottom** left corner. In computer graphics, however, the origin **(0,0)** is at the **top** left corner. By adding **180°** to our angles we are compensating for the different origin, essentially flipping the **y**-coordinate.

To begin, we will draw the simple triangular snowflake … which is the first stage before the recursion can begin. So, we need to draw three edges to represent the starting triangle. Assume that we want the snowflake centered at some location **(Cx, Cy)** and that the snowflake's size is defined by the length of the three triangle edges called **size**.

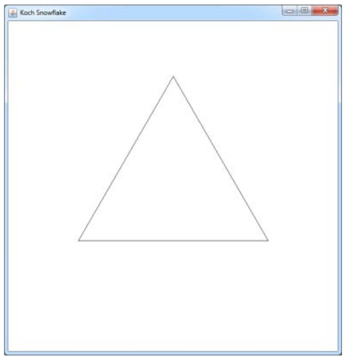
We can draw the triangle with simple straight line edges as follows:

```
Function: drawSnowflake
        Cx, Cy:   location of center of snowflake
        size:     size of snowflake (i.e., length of
                  inner triangle edge)

1.      Px ← Cx + size * cos(90°)
2.      Py ← Cy + size * sin(90°+180°)
3.      Qx ← Cx + size * cos(-30°)
4.      Qy ← Cy + size * sin(-30°+180°)
5.      Rx ← Cx + size * cos(210°)
6.      Ry ← Cy + size * sin(210°+180°)
7.      draw line from (Px, Py) to (Qx, Qy)
8.      draw line from (Qx, Qy) to (Rx, Ry)
9.      draw line from (Rx, Ry) to (Px, Py)
```



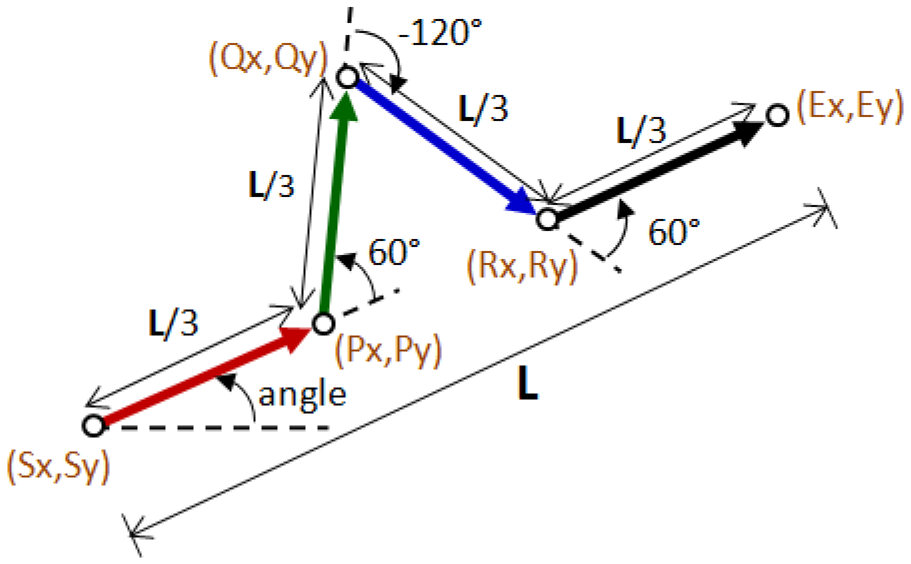
To do this in JAVA, we need to make a kind of **JPanel** for drawing, similar to our **TerrainPanel** used in our Fire Spread Simulation. We need to do the actual drawing from within the **paintComponent()** method of this panel's class. Here is the code. The only part that is important for you to understand is the **drawSnowflake()** procedure. Notice that the Graphics object, called **aPen**, is passed in as a parameter. This allows us to call the **drawLine()** library method which draws a line on the screen, given the start and endpoint coordinates.

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

public class KochPanel extends JPanel {
```



```
    public KochPanel() {
        setPreferredSize(new Dimension(600, 600));
        setBackground(Color.WHITE);
    }

    // Display the snowflake
    public void paintComponent(Graphics aPen) {
        super.paintComponent(aPen);
        drawSnowflake(300,300,200,aPen);
    }

    // This code does all the actual drawing
    public void drawSnowflake(int cx, int cy, int size, Graphics aPen) {
       int px = cx + (int)(size * Math.cos(Math.toRadians(90)));
       int py = cy + (int)(size * Math.sin(Math.toRadians(90 + 180)));
       int qx = cx + (int)(size * Math.cos(Math.toRadians(-30)));
       int qy = cy + (int)(size * Math.sin(Math.toRadians(-30+180)));
       int rx = cx + (int)(size * Math.cos(Math.toRadians(210)));
       int ry = cy + (int)(size * Math.sin(Math.toRadians(210+180)));

       aPen.drawLine(px,py,qx,qy);
       aPen.drawLine(qx,qy,rx,ry);
       aPen.drawLine(rx,ry,px,py);
    }

   // Create the panel in a window and make it visible
   public static void main(String args[]) {
       JFrame frame = new JFrame("Koch Snowflake");
        frame.add(new KochPanel());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack(); // Makes size according to panel's preference
        frame.setVisible(true);
   }
}
```

Let’s work on the recursive part now. Consider an edge going from a starting location **(Sx, Sy)** and going in direction **angle°** towards location **(Ex, Ey)**. Here is how we break the edge down into 4 pieces:



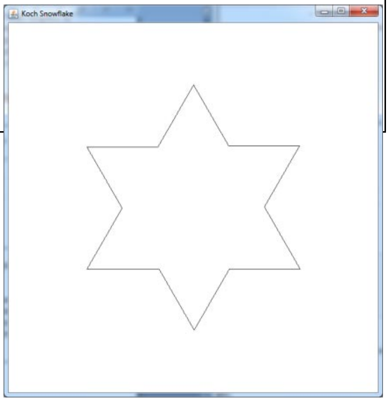
So the code for doing this, is as follows:

```
// This code draws a single Koch edge
public void drawKochEdge(int sx, int sy, int ex, int ey, int a, Graphics p) {
      int length = (int)(Point.distance(sx, sy, ex, ey) / 3.0);

      int px = sx + (int)(length * Math.cos(Math.toRadians(a)));
      int py = sy + (int)(length * Math.sin(Math.toRadians(a + 180)));
      int qx = px + (int)(length * Math.cos(Math.toRadians(a + 60)));
      int qy = py + (int)(length * Math.sin(Math.toRadians(a + 60 + 180)));
      int rx = qx + (int)(length * Math.cos(Math.toRadians(a - 60)));
      int ry = qy + (int)(length * Math.sin(Math.toRadians(a - 60 + 180)));

      p.drawLine(sx,sy,px,py);
      p.drawLine(px,py,qx,qy);
      p.drawLine(qx,qy,rx,ry);
      p.drawLine(rx,ry,ex,ey);
}
```

Notice that the code takes an incoming angle (i.e., **a**) and then draws the edge with respect to that angle. It does this by first computing points **p**, **q** and **r** and then connecting them together. The above code will work for any start/end locations. The **angle** is passed in as a parameter, but alternatively it can be computed from the start and end points of the edge. It is important that the angle passed is actually the correct angle from the start to the end location.

So, we will now adjust our **drawSnowflake()** method to use the **drawKochEdge()** procedure instead of simply drawing a line as follows:

```
    // This code does all the actual drawing
    public void drawSnowflake(int cx, int cy, int size, Graphics aPen) {
       int px = cx + (int)(size * Math.cos(Math.toRadians(90)));
       int py = cy + (int)(size * Math.sin(Math.toRadians(90 + 180)));
       int qx = cx + (int)(size * Math.cos(Math.toRadians(-30)));
       int qy = cy + (int)(size * Math.sin(Math.toRadians(-30+180)));
       int rx = cx + (int)(size * Math.cos(Math.toRadians(210)));
       int ry = cy + (int)(size * Math.sin(Math.toRadians(210+180)));

       drawKochEdge(px, py, qx, qy, -60, aPen);
       drawKochEdge(qx, qy, rx, ry, 180, aPen);
       drawKochEdge(rx, ry, px ,py, 60, aPen);
    }
```



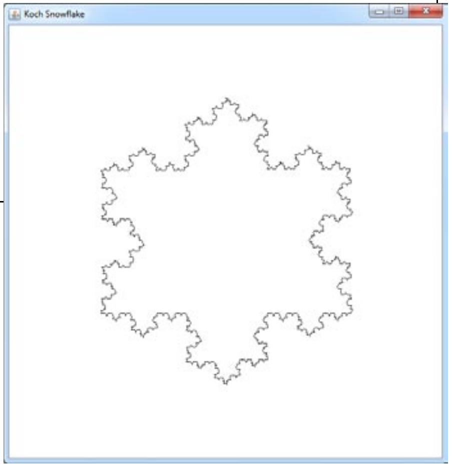
Notice now that we also pass in the angle of the edges, which are **-60°**, **180°** and **60°**. The graphics pen is also passed in.

The code above will draw the shape with one level of recursion. But how do we complete the code so that it does many levels of recursion ?

We need to come up with a base case for stopping the recursion. This can be done in the **drawKochEdge** procedure by noticing when the length of the edge is very small (e.g., 2). In this case, we draw a regular line as opposed to breaking it up into 4 and recursively drawing.

Notice the code:

```
// This code recursively draws a Koch edge
public void drawKochEdge(int sx, int sy, int ex, int ey, int a, Graphics p) {
      int length = (int)(Point.distance(sx, sy, ex, ey) / 3.0);

      if (length < 2)
            p.drawLine(sx,sy,ex,ey);
      else {
            int px = sx + (int)(length * Math.cos(Math.toRadians(a)));
            int py = sy + (int)(length * Math.sin(Math.toRadians(a + 180)));
            int qx = px + (int)(length * Math.cos(Math.toRadians(a + 60)));
            int qy = py + (int)(length * Math.sin(Math.toRadians(a + 60 + 180)));
            int rx = qx + (int)(length * Math.cos(Math.toRadians(a - 60)));
            int ry = qy + (int)(length * Math.sin(Math.toRadians(a - 60 + 180)));

            drawKochEdge(sx, sy, px, py, a, p);
            drawKochEdge(px, py, qx, qy, a+60, p);
            drawKochEdge(qx, qy, rx, ry, a-60, p);
            drawKochEdge(rx, ry, ex, ey, a, p);
      }
}
```

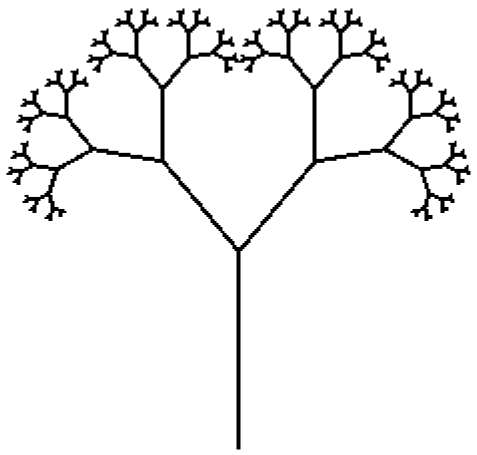


Notice now, that for the recursive case, we call **drawKochEdge** recursively instead of calling the **drawLine**(). This allows the edges of snowflake to be drawn recursively. Notice as well that the angle offsets for the 2nd and 3rd edge (i.e., 60° and -60°) are required for the recursion to know which angle we are "coming in at".

This code will produce the desired snowflake pattern shown here.

Example:

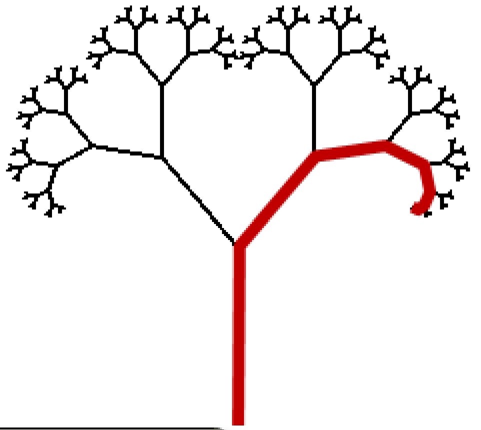
Consider drawing a tree. We can draw the stem of a tree as a branch and then repeatedly draw smaller and smaller branches of the tree until we reach the end of the tree. Here, for example, is a fractal tree →



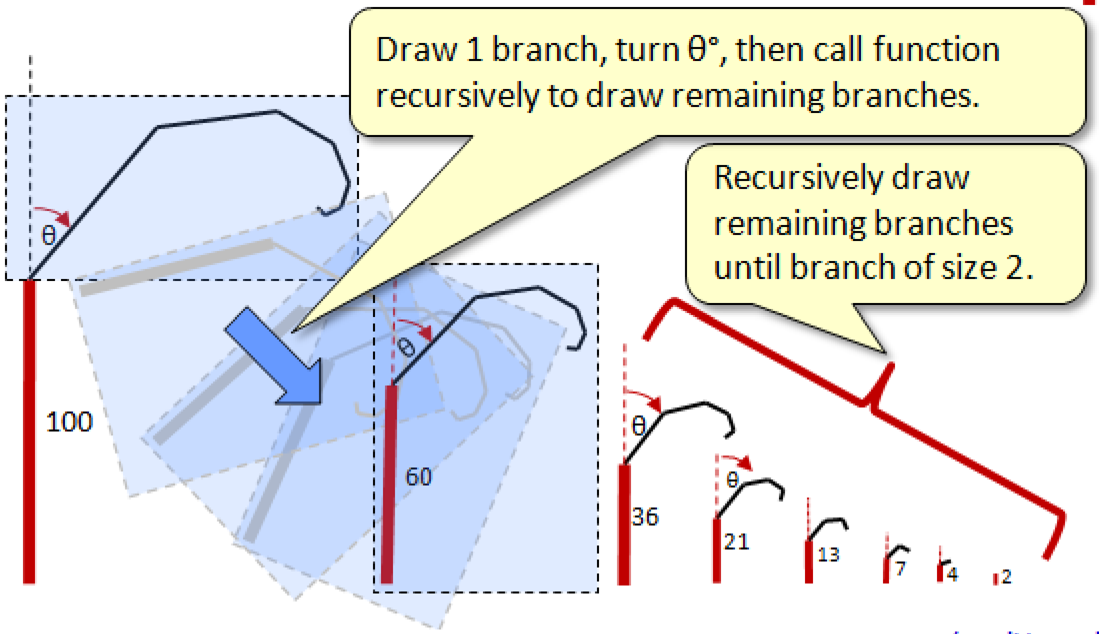
How can we draw this ? Well, imagine drawing just one portion of this tree starting at the bottom facing upwards. We can travel for some number of pixels (e.g., 100) and then bend to the right, perhaps **40°** as shown here. Then, we can draw for a smaller distance, perhaps **60%** of the previous branch. We can do this repeatedly, turning **40°** and drawing a new smaller branch. We can stop when the branch is small enough (perhaps 2 or 3 pixels in length).

Consider doing this for just one portion of branches in the tree (e.g., the right sequence of branches shown here in bold/red).

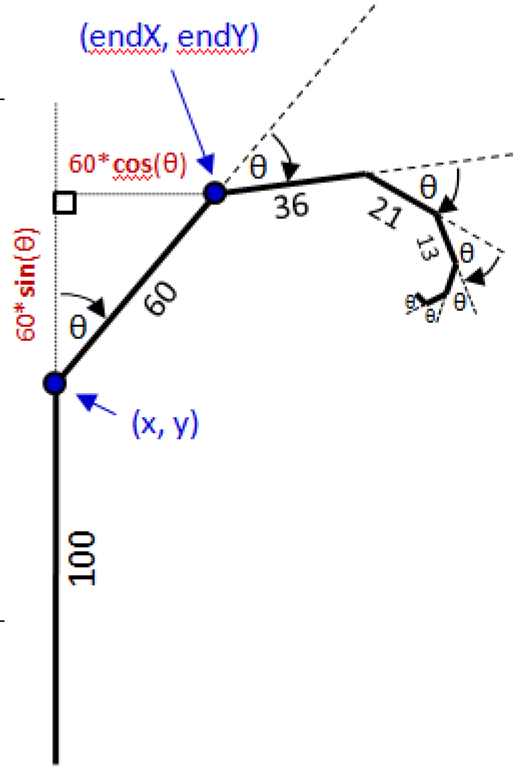
Assume that the **bendAngle** is set at **40°** and that the **shrinkFactor** is **0.6** (i.e., **60%**). Consider the code to draw the rightmost set of branches of the tree. Assume that the initial **(x,y)** location is centered at the bottom of the window and that the drawing direction is north (i.e., upwards at **90°**).



Each time we can draw a single branch, then turn **bendAngle** and then recursively draw the remaining branches starting with branch length reduced by the **shrinkFactor**.


Draw 1 branch, turn θ°, then call function recursively to draw remaining branches.
Recursively draw remaining branches until branch of size 2.
θ
100
60
36
21
13
7
4
2

Here is the code:



```
Procedure: drawSingleTreePath
        length:      length of tree branch
        x, y:        location to start drawing at
        dir:         direction to draw towards

1.      if (length < 2)
2.          return
3.      endX ← x + length * cos(dir)
4.      endY ← y + length * sin(dir+180°)
5.      draw line from (x, y) to (endX, endY)
6.      drawSingleTreePath(length * shrinkFactor,
                                endX, endY, dir + bendAngle)
```

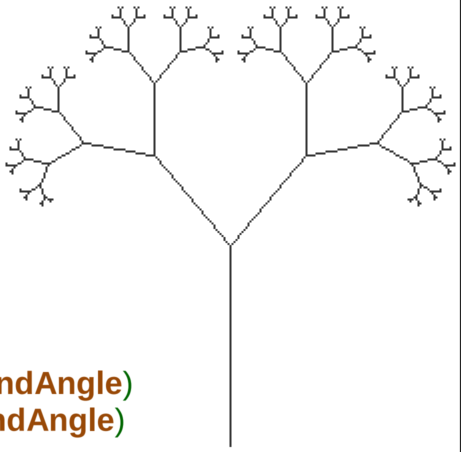
Assume that we begin the tree by drawing a 100 pixel length vertical line from the bottom center of the screen (**windowWidth**/2, **windowHeight**) facing upwards (i.e. North at 90°) to (**windowWidth** /2, **windowHeight**) We would complete the drawing simply by calling the function with initial values corresponding to facing North and a 100***shrinkFactor** = 60 pixel length as follows:

drawSingleTreePath (60, **windowWidth** /2, **windowHeight**, 90°)

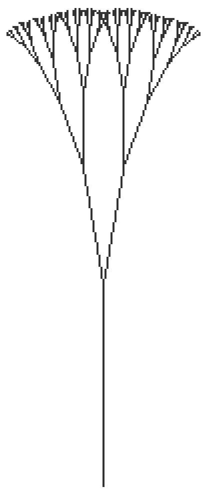
The recursion stops when the tree branch size reaches 2, which is somewhat arbitrary. To draw the remainder of the tree, we simply add code to draw left as well:

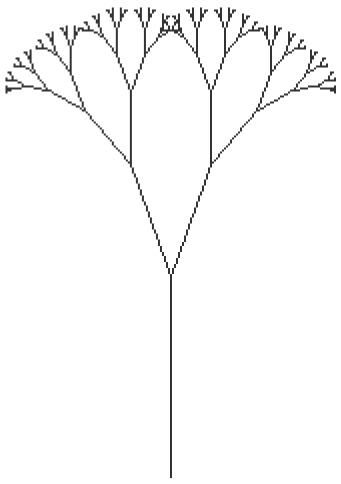
Procedure: drawTree

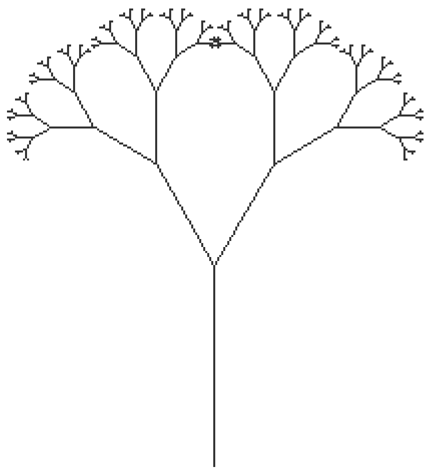
```
        length:      length of tree branch
        x, y:        location to start drawing at
        dir:         direction to draw towards

1.      if (length < 2)
2.          return
3.      endX ← x + length * cos(dir)
4.      endY ← y + length * sin(dir + 180°)
5.      draw line from (x, y) to (endX, endY)
6.      drawTree(length * shrinkFactor, endX, endY, dir + bendAngle)
7.      drawTree(length * shrinkFactor, endX, endY, dir - bendAngle)
```

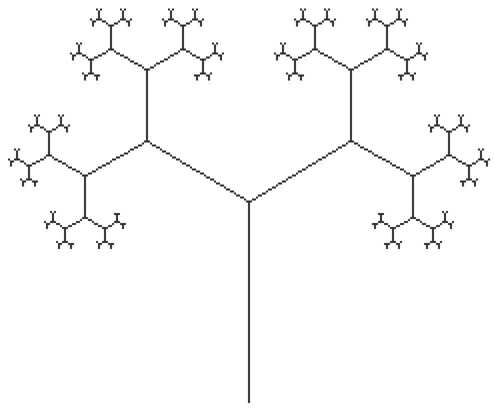


Notice how the tree takes shape because each time we draw a single branch, we branch out **40°** left and **40°** right. Notice how the trees differ as we change the **bendAngle** to other angles other than **40°**:

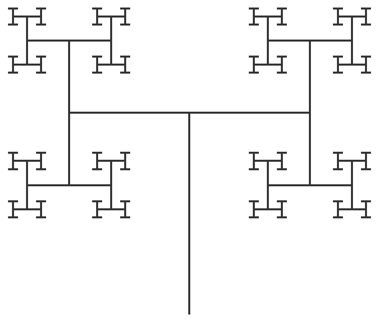

10°


20°

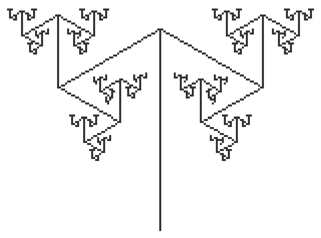

30°



60°

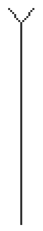


90°

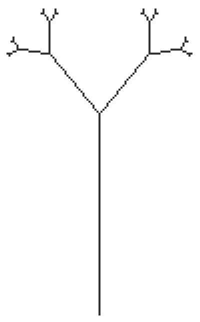


120°

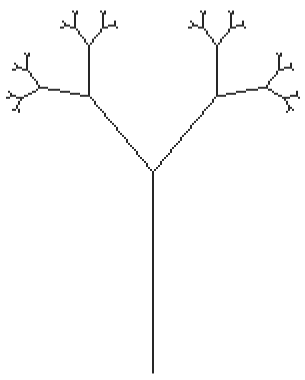
Also, as we adjust the **shrinkFactor** to other values, notice how the tree changes (here the **bendAngle** is fixed at 40°:



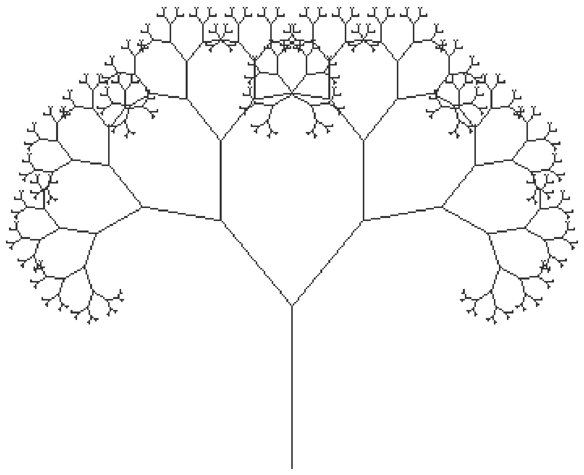
0.10



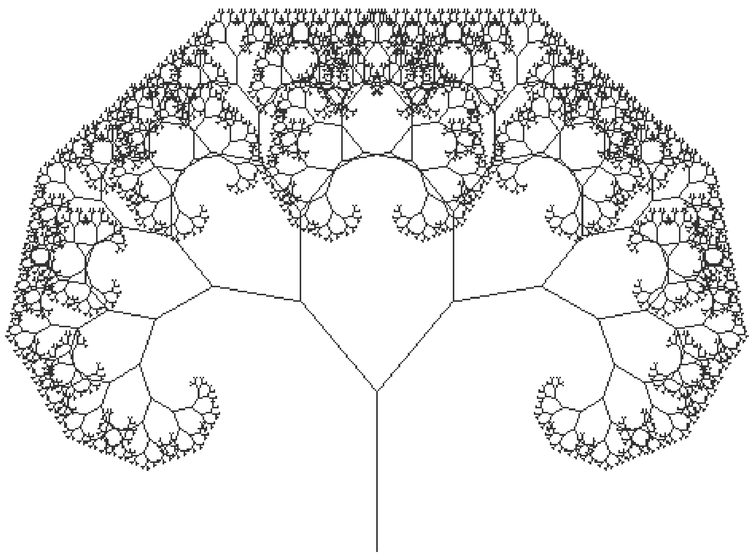
0.4



0.5



0.7



0.75

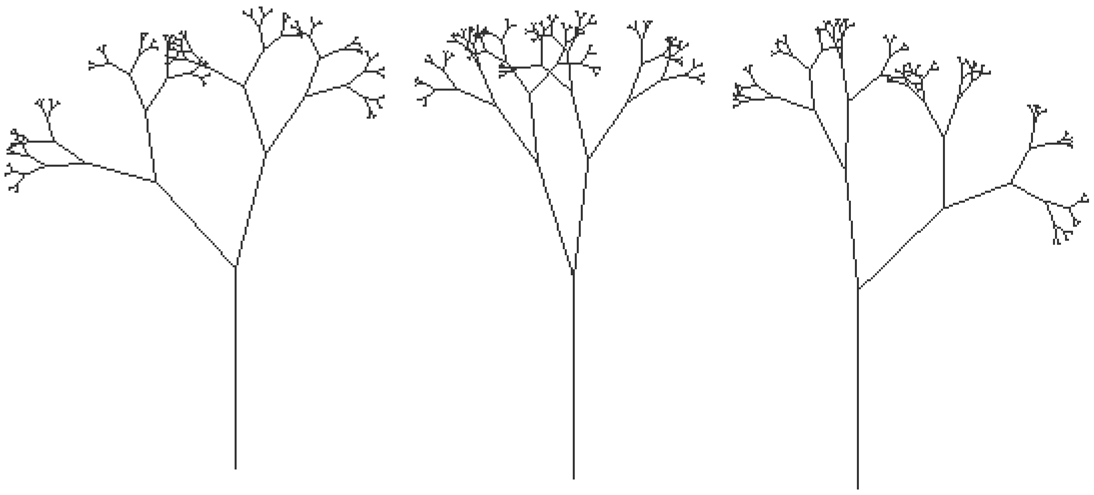
Example:

Now, consider drawing a more realistic tree. A real tree does not have branches that grow at exact angles. Let us begin by adjusting the **bendAngle** to be random instead of fixed at say **40°**. That way, each branch of the tree can bend randomly. Of course, we want to have some limits as to how random they should bend.

We can change the **bendAngle** so that it bends at least some amount each time (e.g., **π/32** = **5.6°**) plus some random amount (e.g., **0** to **π/4** ... or **0°** to **45°**):

bendAngle ← **5.6°** + (random value from **0°** to **45°**)

The result is that our trees will look less symmetrical and will bend more naturally:

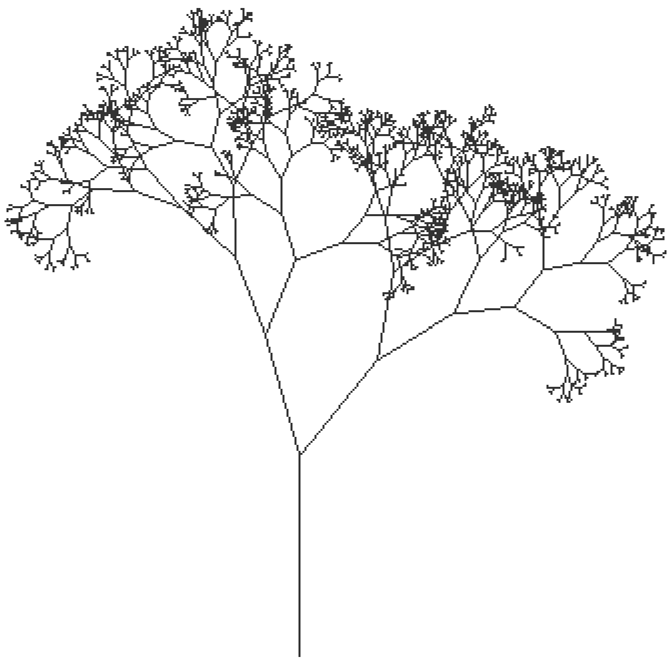
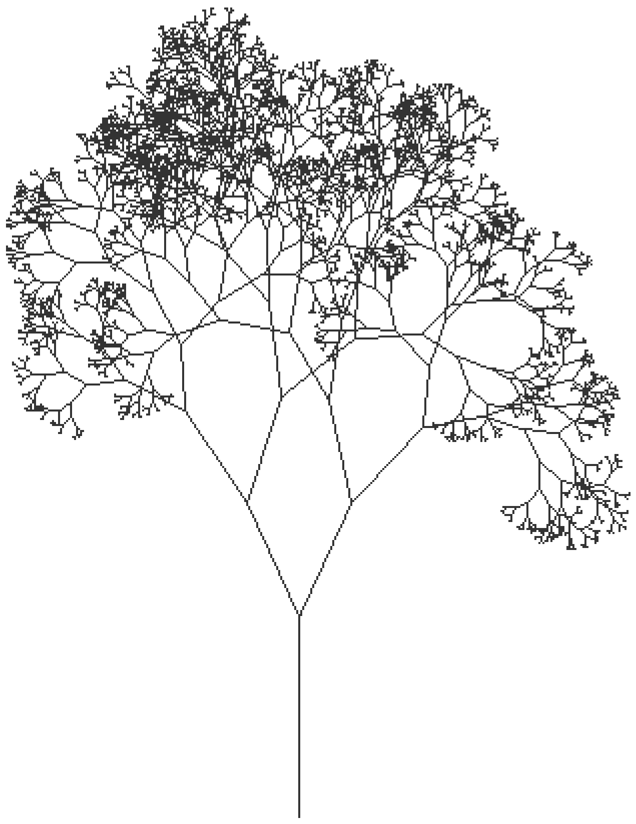


Another factor that we can adjust is the length of each branch. Currently, they shrink by a constant factor of **shrinkFactor** which is **0.6** in the examples above. However, we can shrink them by a random amount (within reason, of course). So, we can add or subtract a random amount from the **shrinkFactor**. It is easiest to simply multiply the **shrinkFactor** by some value, perhaps in the range from **1.0** to **1.5**. That will allow some branches to be larger than **60%** of their "parent" branch.

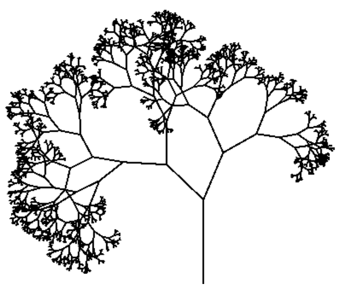
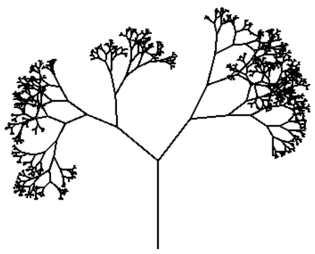
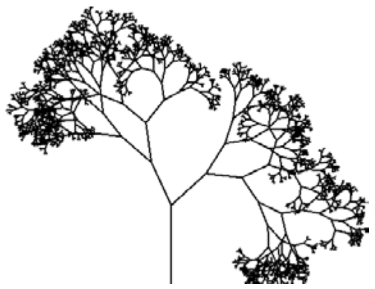
Here is the code as we now have it using a variable **bendAngle** and a new variable called **stretch** to allow randomly longer branches:

```
Procedure: drawNaturalTree
        length:     length of tree branch
        x, y        location to start drawing at
        dir:        direction to draw towards

1.      if (length < 2)
2.          return

3.      bendAngle ← 5.6° + (random value from 0° to 45°)
4.      endX ← x + length * cos(dir + bendAngle)
5.      endY ← y + length * sin(dir + bendAngle+180°)
6.      draw line from (x, y) to (endX, endY)
7.      stretch ← 1 + (random value from 0 to 0.5)
8.      drawNaturalTree (length * shrinkFactor * stretch, endX, endY, dir - bendAngle)
9.      drawNaturalTree(length * shrinkFactor * stretch, endX, endY, dir + bendAngle)
```

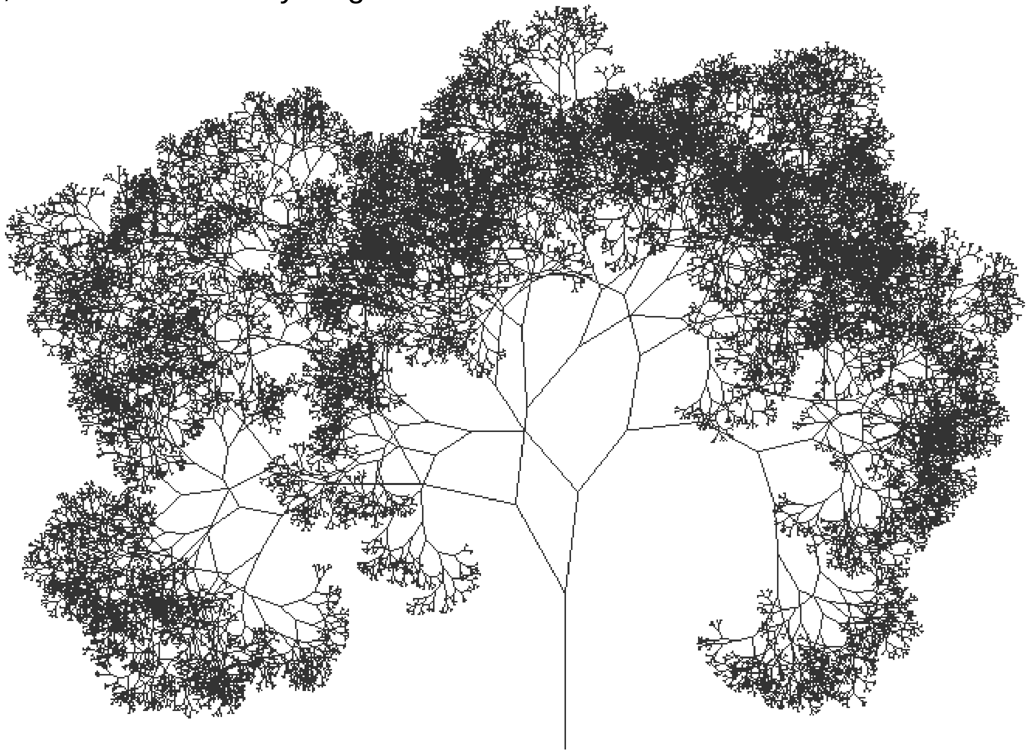
Here are two examples of results due to this change:

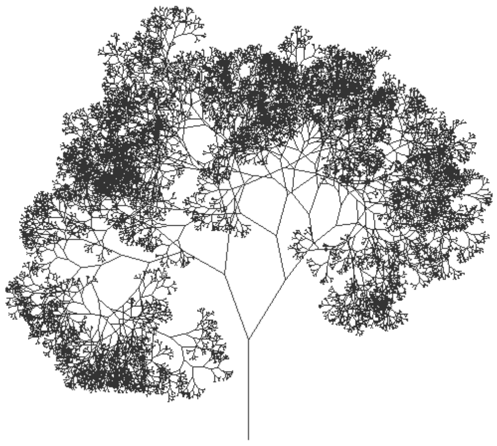
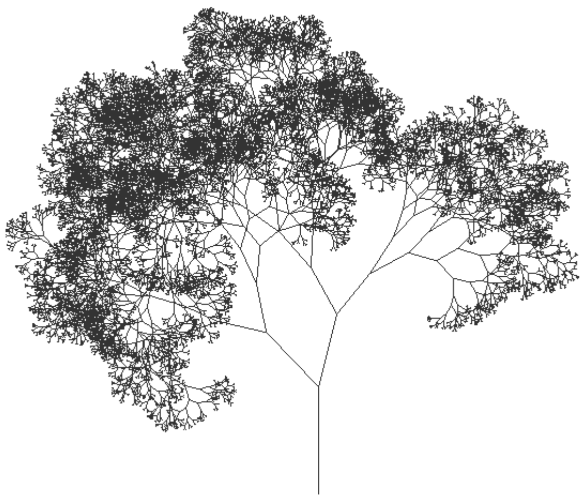



Notice how tree is more "bushy". That is because the recursion goes further before it stops (i.e., it draws more branches). This is because the base case of the recursion stops only when the branch reaches a specific small size. Due to the randomness of the parameters, we may end up with some lopsided trees:

Of course, we can make everything even *bushier* if we increase the **shrinkFactor** to **65%**:



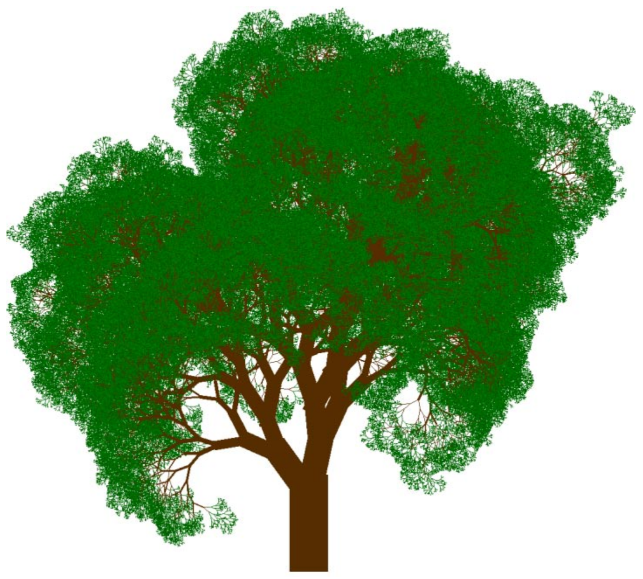



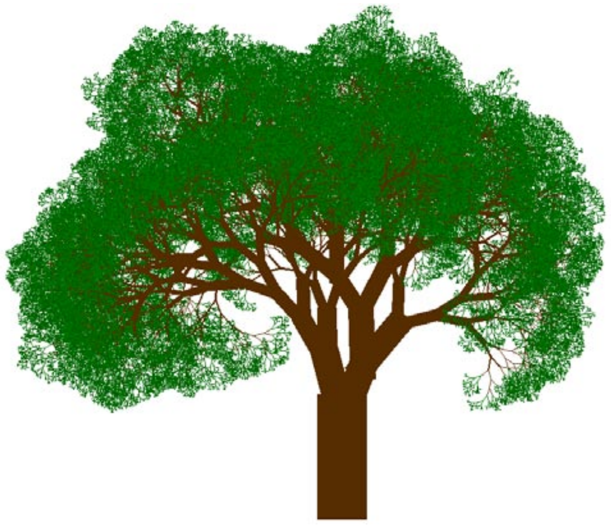
All that remains now is to make the branches thicker and color them brown. For very small branches, we color them green:

```
Procedure: drawColoredTree
        length:     length of tree branch
        x:          x location to start drawing at
        y:          y location to start drawing at
        dir:        direction to draw towards

1.      if (length < 2)
2.          return
3.      set line drawing width to (length * length / 150)
4.      if (length > 10) then
5.          set line drawing color to brown
6.      otherwise
7.          set line drawing color to a random amount of green (e.g., 80 to 130)
8.      bendAngle ← 5.6° + (random value from 0° to 45°)
9.      endX ← x + length * cos(dir + b*bendAngle)
10.     endY ← y + length * sin(dir + b*bendAngle+180°)
11.     draw line from (x, y) to (endX, endY)
12.     stretch ← 1 + (random value from 0 to 0.5)
13.     drawColoredTree (length * shrinkFactor * stretch, endX, endY, dir - bendAngle)
14.     drawColoredTree (length * shrinkFactor * stretch, endX, endY, dir + bendAngle)
```

Notice the **FOR** loop now which repeats exactly **2** times. This reduces our code since the code for drawing the left branch is the same as that for drawing the right branch, except that the angle offset is negative (represented by the value of **b**).

Here is the result (with **shrinkFactor** at **67%** and BEND_ANGLE at random between 10° and 40°:





Here is the JAVA code:

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

public class TreePanel extends JPanel {
    public static int    BEND_ANGLE;
    public static float  SHRINK_FACTOR = 0.67f;
    public static final int WINDOW_WIDTH = 1200;
    public static final int WINDOW_HEIGHT = 1200;

    public TreePanel() {
        setPreferredSize(new Dimension(WINDOW_WIDTH, WINDOW_HEIGHT));
        setBackground(Color.WHITE);
    }

    // Display the Tree
    public void paintComponent(Graphics aPen) {
        super.paintComponent(aPen);
        drawTree(100, WINDOW_WIDTH/2, WINDOW_HEIGHT, 90, aPen);
    }

    // This code does all the actual recursive drawing of a branch
    public void drawTree(int length, int x, int y, int dir, Graphics aPen) {
        if (length < 2)
            return;
        ((Graphics2D)aPen).setStroke(new BasicStroke(length*length / 150));
        if (length > 10)
            aPen.setColor(new Color(85, 45, 0));
        else
            aPen.setColor(new Color (0, 80 + (int)(Math.random()*50), 0));

        BEND_ANGLE = (int)(10 + Math.random()*30);
        double stretch = 1 + Math.random()/2;
        int endX = x + (int)(length * Math.cos(Math.toRadians(dir)));
        int endY = y + (int)(length * Math.sin(Math.toRadians(dir + 180)));
        aPen.drawLine(x, y, endX, endY);

        drawTree((int)(length * SHRINK_FACTOR * stretch), endX, endY,
                        dir - BEND_ANGLE, aPen);
        drawTree((int)(length * SHRINK_FACTOR * stretch), endX, endY,
                        dir + BEND_ANGLE, aPen);
    }


   // Create the panel in a window and make it visible
   public static void main(String args[]) {
      JFrame frame = new JFrame("Tree Drawing");
      frame.add(new TreePanel());
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack(); // Makes size according to panel's preference
      frame.setVisible(true);
   }
}
```

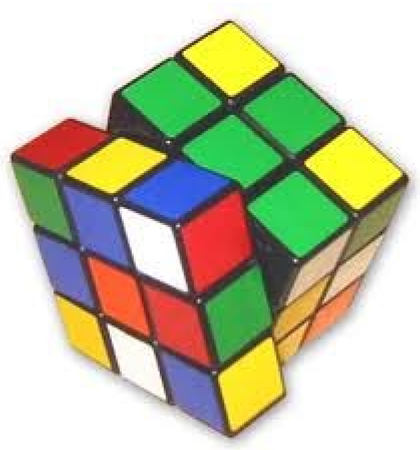
8.4 Search Examples

Sometimes, we use recursion to search for an answer to a problem. Searching for an answer recursively can often lead to a simpler, more elegant solution than writing search code that uses many loops and IF statements.

Example:

Consider writing code to solve a Rubik's cube. The task may seem daunting at first ... but there are many systematic ways of solving the cube in steps or phases. Any non-recursive code that you would write could be quite complex, tedious and/or cumbersome. However, a recursive solution is quite simple.

Assume that the cube can be solved by making at most **100** quarter turns of one of the cube's sides (a reasonable assumption). Here is the idea behind the recursive solution:



> Try turning **1** of the **6** sides **1/4** of a turn, then try to solve the cube in **99** more turns, recursively. If the recursion comes back with a solution from the **99** additional moves, then the **1/4** turn that we made was a good one, hence part of the solution. If however, the recursion came back after **99** moves with an unsolved cube, then the **1/4** turn that we made was a bad choice ... undo it and then try turning one of the other **5** sides.

Here is the pseudocode for a function to solve the cube recursively in **n** steps, returning **true** if it has been solved, or **false** otherwise:

```
Function: SolveRubiksCube
        cube:     the cube data structure
        n:        maximum number of steps remaining to solve the cube

        if (cube is solved) then
              return true
        if (n is 0) then
              return false
        for each side of the cube s from 1 to 6 do {
              turn side s 1/4 turn clockwise
              recursiveAnswer ← SolveRubiksCube(cube, n-1)
              if (recursiveAnswer is true) then
                    return true
              turn side s 1/4 turn counter-clockwise          // i.e., undo the turn
        }
        return false
```

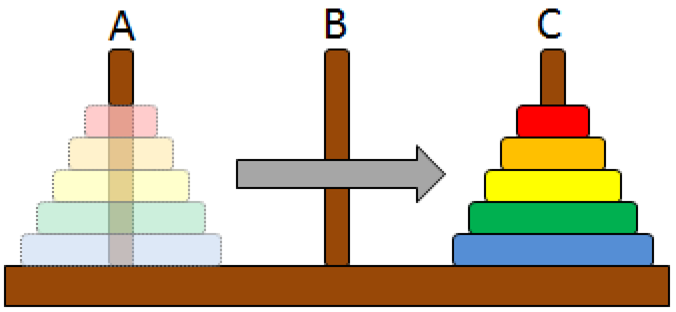
That is the solution! It is quite simple. Do you see the power of recursion now ? Of course, this particular solution can be quite slow, as it blindly tries all possible turns without any particular strategy. Nevertheless, the solution is simple to understand. It would be difficult to produce such a simple function to solve this problem without using recursion.

Example:

Another problem that is naturally solved recursively is the **Towers of Hanoi** puzzle. In this puzzle game, there are three pegs and a set of **n** disks of various sizes that can be moved from one peg to another. To begin, all **n** pegs are stacked on the first peg in order of increasing sizes such that the largest disk is at the bottom and the smallest is at the top. The objective of the game is to move the entire tower of disks to a different peg by moving one disk at a time and doing so with a minimum number of steps.

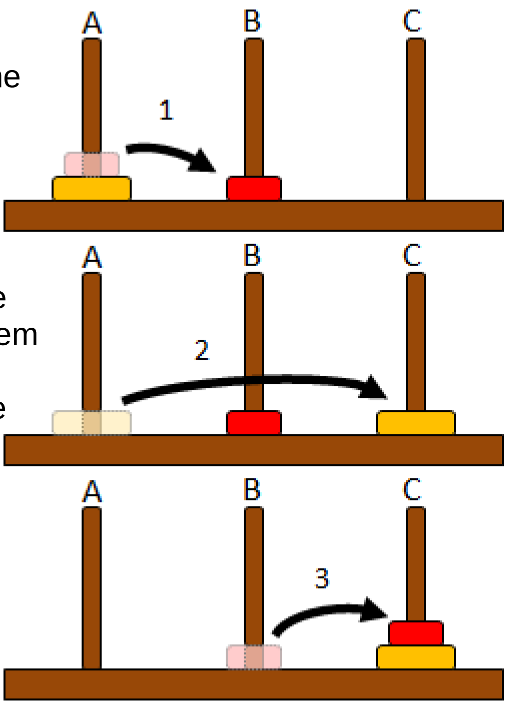
There are just two rules:

1. disks must be moved from one peg to another, not placed aside on the table
2. a disk can never be placed onto a smaller-sized disk.



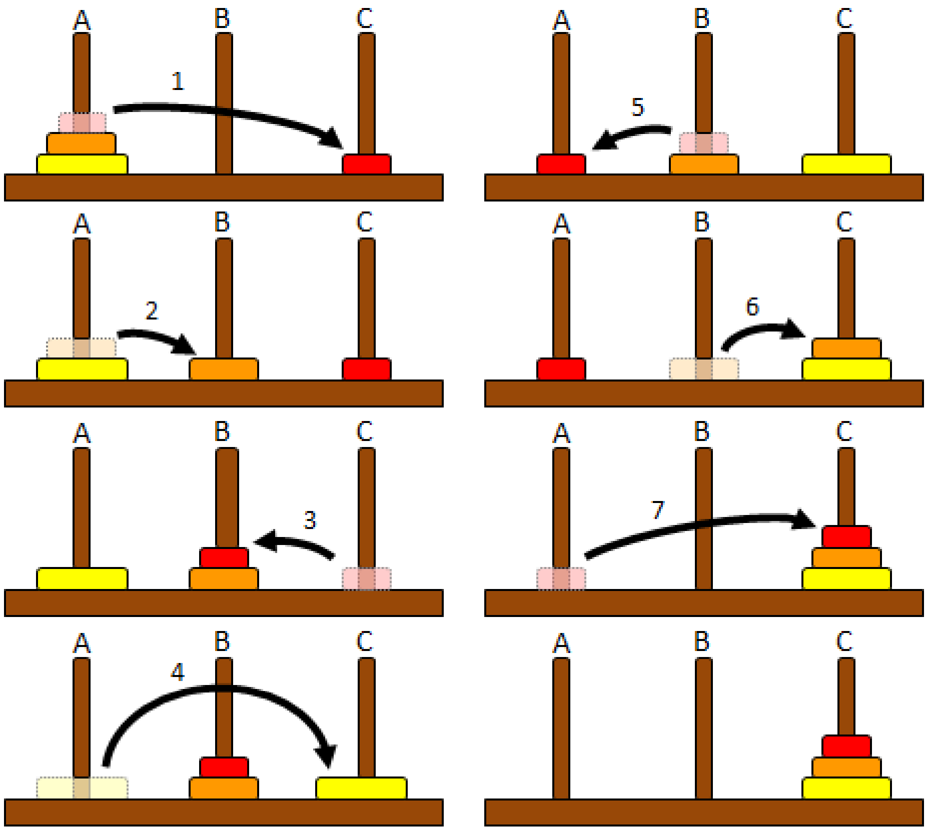
To approach this problem recursively, we must understand how to break the problem down into a smaller problem of the same type. Since all we are doing is moving disks, then a "smaller" sub-problem would logically just be moving less disks. For example, if **1** disk is moved, then only **n-1** disks remain to be moved ... which a smaller problem.

So what then would be the base case ? Well ... what is the simplest number of disks to move ? Zero. A simple problem is also the case where **n = 1**. In that case, we simply pick up the disk and move it. For values of **n** greater than **1**, we now have to worry about moving some disks out of the way. Here is the solution when **n = 2** →

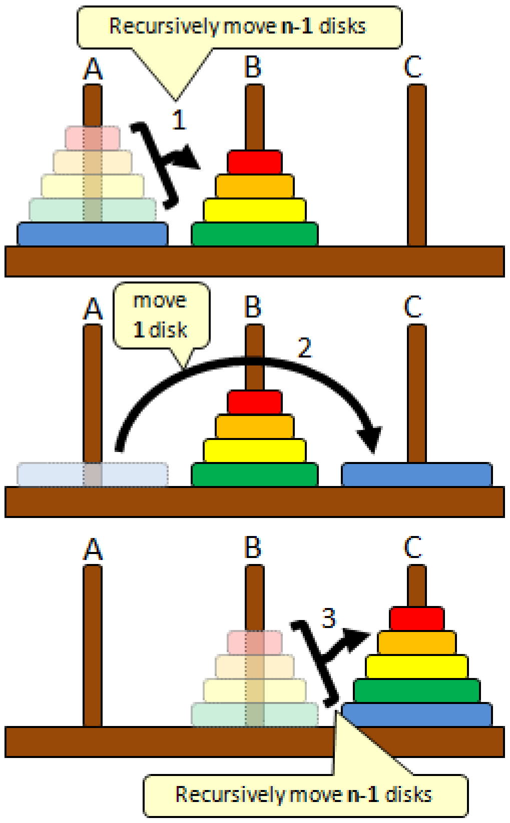


Notice that after the small red disk is moved to peg B, then we have the smaller/simpler problem of moving the one larger disk from peg **A** to peg **C**. Finally, we have the simpler/smaller problem of moving the red disk from peg **B** to peg **C** and we are done.

It may be hard to see the recursion, at this point, so let us look at the solution for **n = 3**:



You may notice that steps 1 to 3 are basically the same as the problem where **n = 2**, except that we are moving the disks from peg **A** to peg **B** instead of to peg **C**. Then step 4 simply moves the 1 disk from peg **A** to peg **C**. Finally, we are left with the recursive sub-problem again of moving 2 disks from peg **B** to peg **C**.



So, the recursive solution of moving **n** disks is based on this idea:

1. Move the **n-1** smaller disks out of the way to the spare peg **B**

2. Move the large disk to peg **C**

3. Move the **n-1** smaller disks to peg **C**

As you can see above, there are two recursive sub-problems corresponding to moving **n-1** disks out of the way first, and then solving the **n-1** problem once the largest bottom disk is in place.

So, we can now write the code by defining a procedure that indicates the **peg to move from**, the **peg to move to**, the **extra peg** and the **number of disks to move**.

```
Procedure: TowersOfHanoi
        pegA:   the peg to move from
        pegC:   the peg to move to
        pegB:   the extra peg
        n:      number of disks to move

1.      if (n is 1) then
2.          Move 1 disk from pegA to pegC
        otherwise {
3.          TowersOfHanoi(pegA, pegB, pegC, n-1)
4.          Move 1 disk from pegA to pegC
5.          TowersOfHanoi(pegB, pegC, pegA, n-1)
        }
```

As you can see, the problem is expressed in a simple manner with only about 5 lines. Notice that there was no need to check if **n = 0** since that would represent a problem that is already solved.