**Lab 5**

[**http://eliza.csc.ncsu.edu/snap/**](http://eliza.csc.ncsu.edu/snap/)

# **What is Recursion?**

This activity addresses the concept of recursion. It provides an illustration of how recursion is used in computing.

## Introduction

What is recursion? A recursive process is one in which objects are defined in terms of other objects of the same type (Wolfram MathWorld). So what does that mean??? It is a function (procedure) that calls itself! When performing recursion, we try to reduce a large problem into a series of similar, smaller problems that can be solved in a similar ways. In fact, it is often best if we can reduce the large problem into a collection of trivially small problems.

The best way to understand is to look at an example. Factorials are an excellent example.

5! (factorial) is 5 \* 4 \* 3 \* 2 \* 1 = 120, right?

So basically…

5!

is the same as

5 \* 4!

Which is the same as

5 \* 4 \* 3!,

Which is the same as

5 \* 4 \* 3 \* 2!

Which is the same as

5 \* 4 \* 3 \* 2 \* 1!

1! is just 1.

Let’s look at it in another way: n! is n \* (n – 1)! This is what we just demonstrated. If I create a procedure called Factorial that will get the answer of the factorial of the number I send over to the procedure, then I could create a recursive function like this…

Factorial(n) = n \* Factorial(n -1)

The Factorial(n – 1) would keep executing until it reached a stopping point, in this case 1 as that is the last number I will need to multiple. So I am using my procedure to “call” itself, sending over a simpler version (in this case a smaller number) until I get to the smallest number (in this case the number 1). This smallest number is what is called the Base Case – when the function should stop calling itself.

Factorial (5)

5 \* Factorial(4)

5 \* 4 \* Factorial(3)

5 \* 4 \* 3 \* Factorial(2)

5 \* 4 \* 3 \* 2 \* Factorial(1)

which rolls out like this

5 \* 4 \* 3 \* 2 \* 1

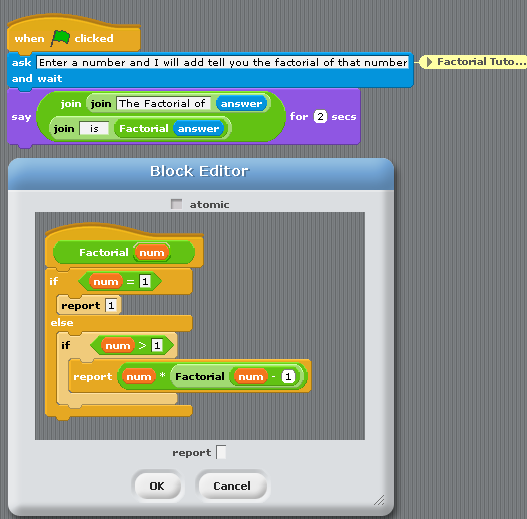
5 \* 4 \* 3 \* 2

5 \* 4 \* 6

5 \* 24

120

## Factorial Recursion in Snap



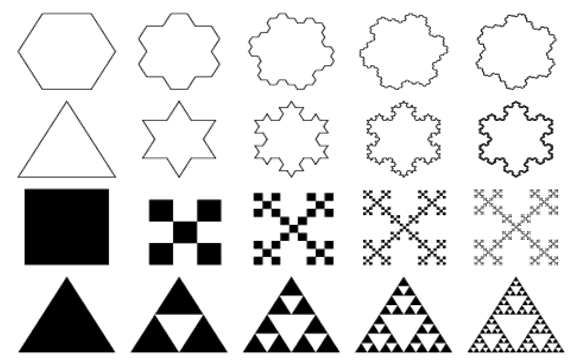
## Visual Recursion

Let’s look at visual recursion where we have a recursive call that is going to draw something, in our case fractals.

## What are Fractals?

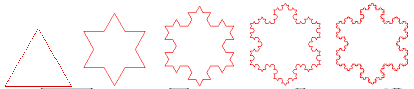
They’re everywhere, those bright, weird, beautiful shapes called fractals. But what are they?

Fractals are geometric figures, just like rectangles, circles and squares, but fractals have special properties that those figures do not have. Using your computer, find an image of a fractal. You may see some of the following images.



## The Koch Snowflake

The Koch Snowflake is a famous fractal. How is it made? It starts with an equilateral triangle. Divide one side of the triangle into three equal parts and remove the middle section. Replace it with two lines the same length as the section you removed. Do this to all three sides of the triangle. Keep repeating steps 1, 2 and 3 infinitely and you get the fractal, the Koch Snowflake.

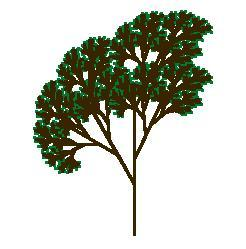


These fractals look really complicated to draw manually. But by using recursion we can achieve these impressive images very easily. Recursion when understood can make programming very easy in appropriate scenarios.

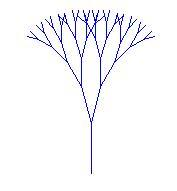
# **Make a Recursive Tree**

When you make blocks,**be sure that the ATOMIC box is not checked**! (This will allow you to stop your program if you write a program that will go on forever!)

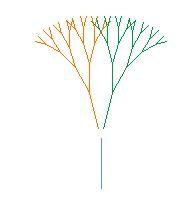
Our goal is to draw a tree like this:



but we’ll start with a simpler version that shows the technique clearly, although less prettily:



The key to understanding this technique is to see that the tree is a fractal, that is, a picture that contains smaller versions of itself:

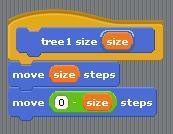


We’re going to create a TREE block in Snap. It’ll start with a MOVE block to draw the trunk of the tree, then a TURN block turning left, then a TREE block to draw the left smaller tree, and so on.

“Wait!” you’re probably thinking. “How can we use a TREE block inside a TREE block? It doesn’t exist yet!” That’s the big idea for this assignment.

We’re going to work up to the complicated tree starting with very simple steps.

Step 1. Make a TREE1 block (so named because it draws just one level of the tree) using this script:



This looks ridiculously simple, but trust us, it’ll get interesting soon. When run, the script draws one tree branch, and then *moves the sprite back to its original position*. That’s going to be really important when we start using scripts within scripts; we always want to be able to assume that our tree blocks leave the sprite in the same position, and facing the same direction, after it as before it.

Step 2. Point the sprite facing upward, and put the pen down. Then try TREE1 SIZE 50. You should get a result something like this:



Step 3. Make a TREE2 block that draws two levels:

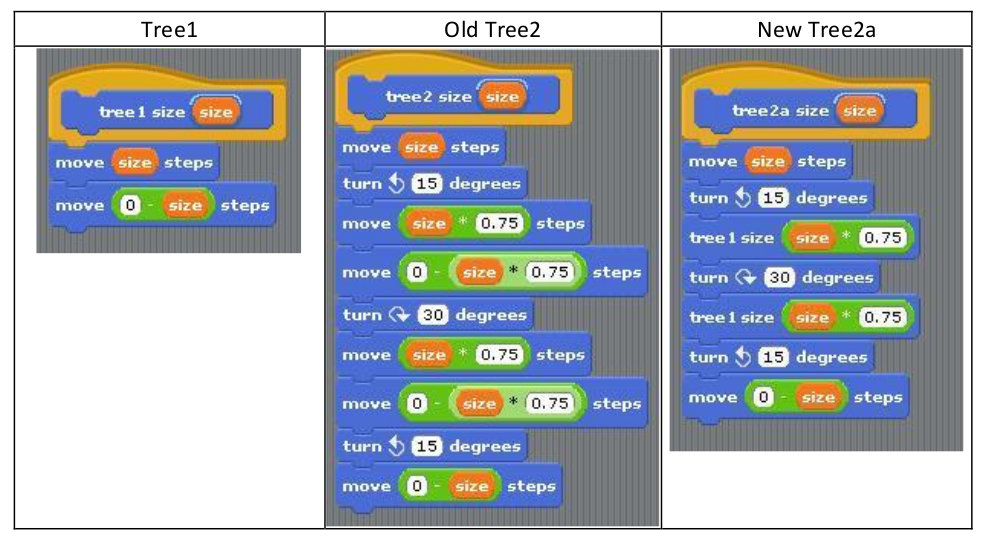


When run, it should give this result:



* At this rate we’ll never be able to make the beautiful version of the tree. In the next step, we’ll discuss how to simplify this process.
* Before going on, make sure that you can mentally trace through the code provided. Paying close attention to the forwards and turns is going to be important to see the big picture and understand the recursion.

Step 4. That TREE2 block is pretty long and repetitive. But we can simplify it if we notice that in two places it has a move forward/move back pair of blocks, and that this is what TREE1 does! So we can use TREE1 to shorten TREE2. Compare the code below to convince yourself that the new Tree2a will work the same as the original Tree2.



Note that the tree blocks inside this TREE2A script are TREE1 blocks, not TREE2 blocks! So there’s no mysterious TREE-using-TREE situation here; it’s not unusual for one block to be used in another block’s script.

Step 5. Make a TREE3 block that uses the TREE2 block, on the same pattern, and see if you get the result that you expect.

Step 6. If you can stand it, make a TREE4 block that uses the TREE3 block and try it out.

**This would be a good time to save your project.**

Step 7. Okay, here’s the big idea: We can write a TREE block that uses itself in its own script *provided that it knows how many levels it’s expected to draw*! So, in addition to the size input, it’ll have a LEVELS input:

In the earlier steps, TREE3 used TREE2; TREE2 used TREE1. Here, TREE will use TREE, but reducing the number of levels by 1.

Step 8. Once you can draw a tree of five or six levels using your TREE block, see if you can make one like the first picture in this handout. It’s different from what we’ve done so far because the smaller trees are drawn part way up the trunk, instead of at the top of the trunk, and because the pen color is green for the lowest-level branches (the TREE1-like ones) and brown for the others. You don’t have to get it exactly like the picture; just try to make a more realistic-looking tree.

## Vary Your Tree

Change your tree in the following ways. You should probably try these one at a time so that you can always tell which change you made caused a new bug.

* change the turn angle
* change the scale factor
* change the number of recursive calls

Note: If your character goes off the screen your image might get messed up because you’ll tell your character to change their y-position but the character literally won’t be able to.

## Base Cases

Now that you have some experience with some recursion, we’re going to point out some of the anatomy of a recursive procedure.

We always need a way to figure out that we’re done and shouldn’t call the recursive call again. Lines 2-4 are the “Base Case” where we handle things that are SO simple we don’t need to call the recursive case again.

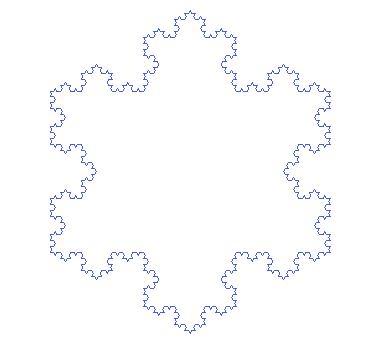
If this isn’t the base case we think to ourselves “Woah - That seems complicated. I’ll just do a small part of the problem and delegate to someone else”. Here’s what we do:

* draw the trunk of the tree (line 6),
* position the character for the left sub-tree (line 7),
* delegate to another copy of the tree block to draw the left sub-tree (line 8),
* re-position the character for the right sub-tree (line 9),
* delegate to another copy of the tree block to draw the right sub-tree (line 10),
* re-position the character to retrace the trunk (line 11), and
* re-trace the trunk and leave the character exactly where it started out.

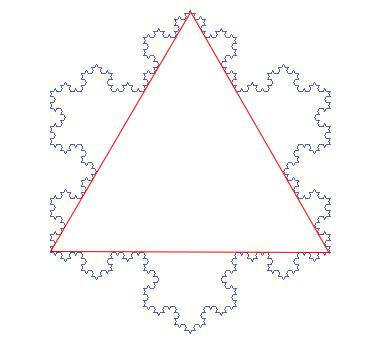
This wasn’t the “best” one we discussed in a previous step, but is a good example of a base case. (Note here we addressed the problem of getting infinite recursion when we have a level less than 0.

# **Koch Snowflake**

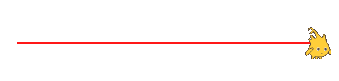
Using recursion we can make beautiful things! In this step you will learn to make the beautiful Koch snowflake (below).



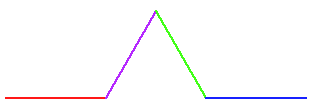
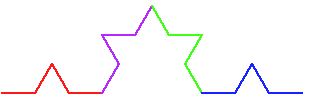
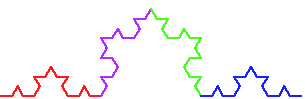
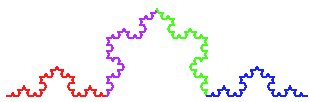
The entire snowflake consists of three copies of a fractal, arranged in a triangular shape-

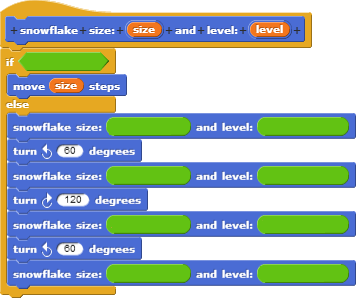


This fractal does not return to the same place between recursive calls. For example, the base case has the sprite start on the left and go to the right (below).



Then at every successive level, it repeats the previous level 4 times in a pattern shown below. Look at the images below to see how the previous level is repeated in each higher level.

Something that is hard to tell visually is that each recursive call is one-third the size of the caller. (This makes the entire fractal the same distance from left to right in every level.) Fill in the blanks in the following recursive procedure to draw the Koch snowflake. Note that at the end of each recursive call we do not end where we started as we did with the trees. 

When you get one side of the snowflake working, combine three of these to make the entire snowflake.

# **Lab- Add Binary Search to the Guessing Game**

Using the starting code provided in Lab 3, upload your expanded guessing game.

Your Noseguy sprite should utilize binary search in his Guess function to guess the number in as few guesses as possible.

The best route for this would be to set a maximum and minimum value, and adjust them as Noseguy is told “higher” or “lower” by Snap. Think how you would play as a human. You should have Noseguy able to guess the number in 4 guesses or less.

Binary search is an efficient way of searching in a list, and is recursive. The algorithm splits the list and checks the middle value, if it is not found it compares the values.

If the search value is higher than the middle value, the algorithm moves to the upper half of the list; if it’s lower, it moves to the lower half. This half becomes a new list that is passed into the algorithm again- this allows the search to be used recursively.

The algorithm then splits this new list in half, and checks the middle for a match. If not found, it checks to see if it must go higher or lower again. This continues until it has no more values or finds a match.

For example, in guessing between 0-20, if the number was 2, the algorithm would guess 10. 2 is lower, so it would guess between 0-10. It would then guess 5, but since 2 is smaller, it would split the list to 0-4 and and guess again with the middle value of 2.