

Unit 7: Recursion - It's not a bad word!

Learning Objectives

- 1: The student can use computing tools and techniques to create artifacts.
- 4: The student can use programming as a creative tool.
- 20: The programs can use abstraction (procedures) to manage complexity in a program.

Readings/Lectures

- Reading 7.01: All Algorithms are Not Equal (/bjc-course/curriculum/07-recursion/readings/01-what-is-recursion)
- Reading 7.02: Algorithm Analysis (</bjc-course/curriculum/07-recursion/readings/02-recursion-in-a-nutshell>)

External Resources

- Berkeley Lecture on Recursion (<https://coursessharing.org/courses/6/lectures/17>)
- Koch's Snowflake, Explained (<http://math.rice.edu/~lanius/frac/koch.html>)
- Snowflake BYOB File (</bjc-course/curriculum/07-recursion/code/snowflake.ypr>)

Labs/Exercises

- Lab 7.01: Make a Recursive Tree (</bjc-course/curriculum/07-recursion/labs/01-make-a-recursive-tree>)
 - Lab 7.02: Recursive Bunnies (</bjc-course/curriculum/07-recursion/labs/02-recursive-bunnies>)
-

[Curriculum \(/bjc-course/curriculum\)](/bjc-course/curriculum/) / [Unit 7 \(/bjc-course/curriculum/07-recursion\)](/bjc-course/curriculum/07-recursion/) /

[Reading 1 \(/bjc-course/curriculum/07-recursion/readings/01-what-is-recursion\)](/bjc-course/curriculum/07-recursion/readings/01-what-is-recursion/) /

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...

$\text{Factorial}(n) = n * \text{Factorial}(n - 1)$

The $\text{Factorial}(n - 1)$ 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.

$\text{Factorial}(5)$

$5 * \text{Factorial}(4)$

$5 * 4 * \text{Factorial}(3)$

$5 * 4 * 3 * \text{Factorial}(2)$

$5 * 4 * 3 * 2 * \text{Factorial}(1)$

which rolls out like this...

$5 * 4 * 3 * 2 * 1$

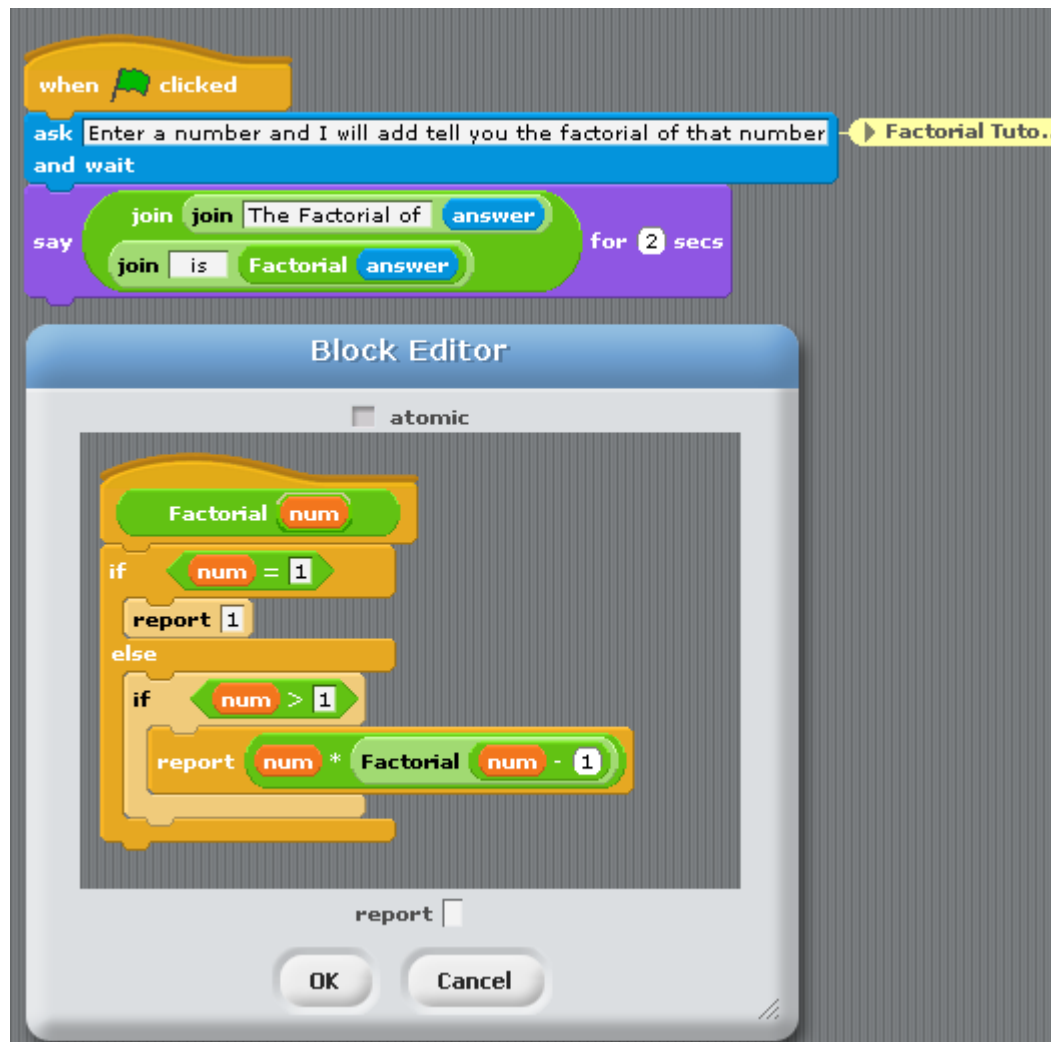
$5 * 4 * 3 * 2$

$5 * 4 * 6$

$5 * 24$

120

Factorial Recursion in BYOB



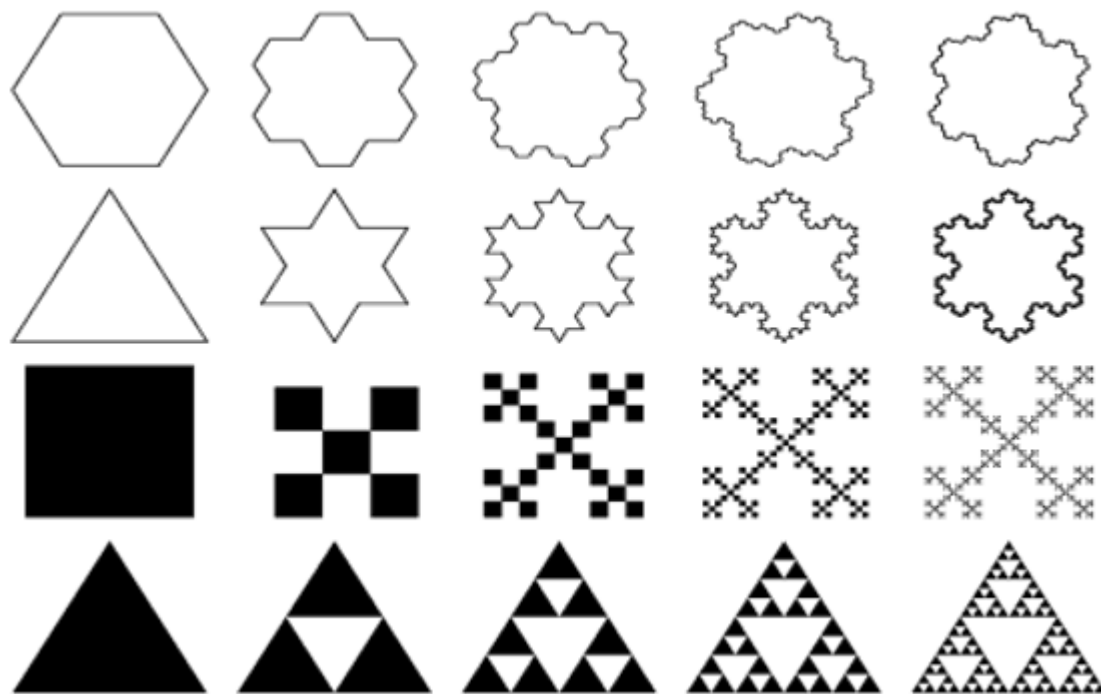
Visual Recursion

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?

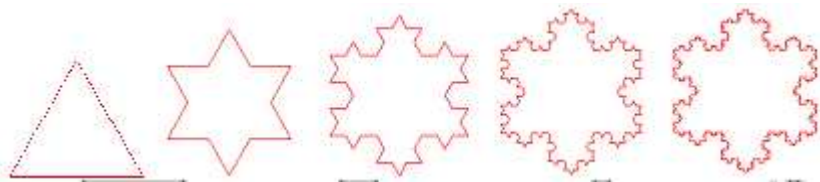
They're everywhere, those bright, weird, beautiful shapes called fractals. But what are they?

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.



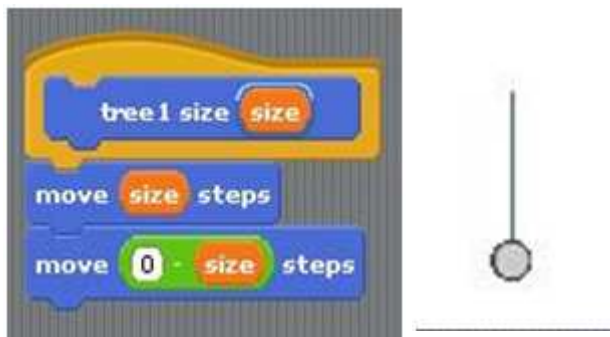
Let's look at another example of recursion.

Recursion to Draw a Tree

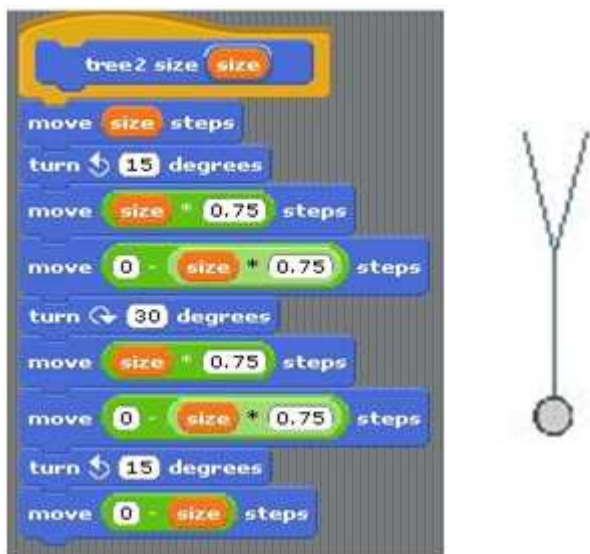
This is a simple tree where we want to draw the base (the line for the trunk), then the branches. Note that I am drawing smaller and smaller versions of the same lines. This is perfect for recursion!



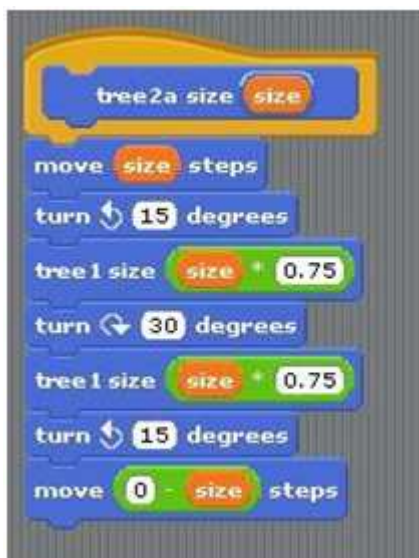
Let's start with drawing the base. This is our "trunk" of the tree.



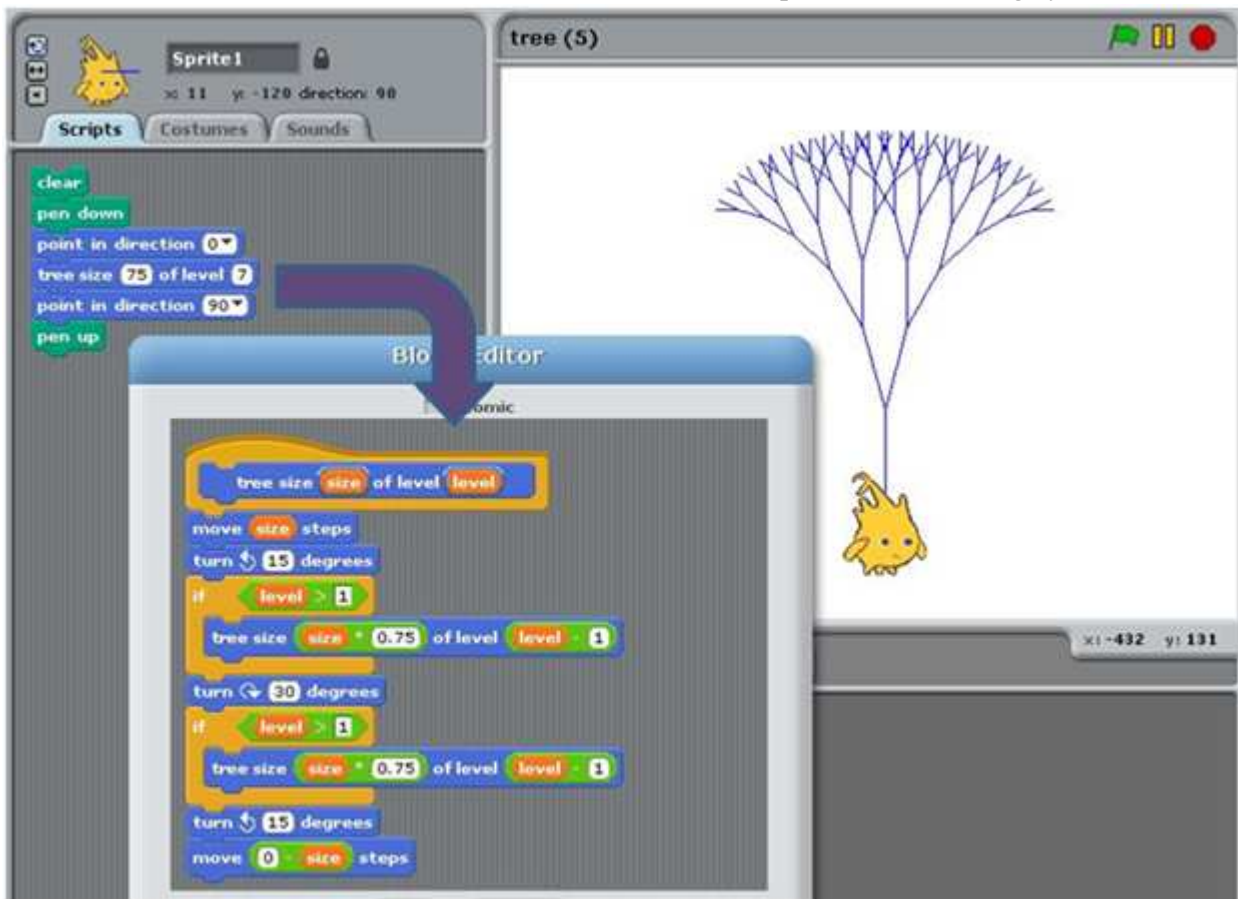
In BYOB, we will tell byob to move “size” steps, then move back. Next, we will need to draw the first branches. Move the steps (back up to the top); then turn and move (75%) and then move (75%) back. Turn back to center + amount to other side; then move (75%) and move (75%) back. This way we get the smaller branches.



We can now use our code from drawing the first tree to help us draw the next level.



We can use variables to make the drawing relative to previous settings. This is one example of the finished code.

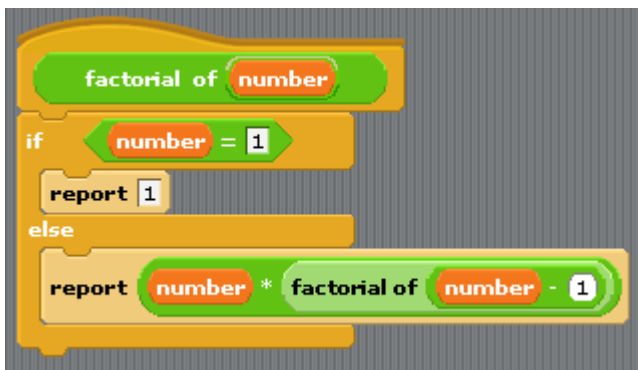


Material from UC Berkeley

Recursion in a Nutshell

We have spent the last couple of labs learning about recursion. You have probably realized that the technique for writing recursive blocks is relatively straightforward, although the body of the block can be a bit tricky to figure out until you have gotten enough practice, or have worked through several examples of what the block should do.

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. For example, finding the factorial (<http://en.wikipedia.org/wiki/Factorial>) of a number is an operation that is often written recursively because each solution builds off of the solution before it ($5! = 4! * 5$; $12! = 11! * 12$). It can be written recursively as a set of trivially small problems (each step only multiplies two numbers) that build up to a significant solution.



The process for writing recursive functions can be summarized in a few steps:

Step 1: Determine the base case

The base case is the simplest possible solution to the overall problem. In the example of the factorial, it is simple to find the answer when the number provided is 1: the answer will always be 1. Every recursive problem has a base case – sometimes it will be straightforward and other times, it may require a little more thought. What would be the base case for the following recursive problems? * Searching through a list for the first occurrence of a particular number and replacing it with zero.


- Counting the number of elements in a list.
- Computing X^Y , where X and Y can both be specified as arguments to the block.

Step 2: Choose a more complicated example

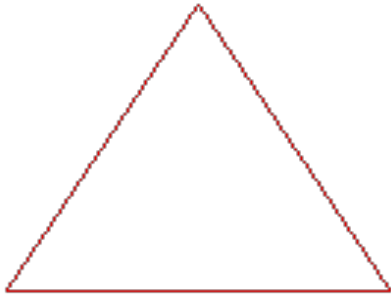
Continuing our work on the factorial problem, we will consider **factorial of 10**. We could choose any number here: 3, 5, 100, 24481, as long as it is not the base case. Hint: sometimes choosing values right next to the base case (2, in this case) makes it harder to detect the relationship in step 3, but any value outside of the base case will do.

Step 3: Consider an example that is slightly simpler (closer to the base case)

factorial of 9 might work here. Assuming we can get the answer to **factorial of 9**, how can we use its solution

to help us figure out ? Figuring this out will allow us to determine the “relationship” between solutions, which is a pivotal point in writing a recursive function. In this case, $10! = 9! * 10$, which shows a clear relationship that we can represent in BYOB.

Cynthia Lanius



The Koch Snowflake

Table of Contents

Introduction

Why study fractals?

What's so hot about
fractals, anyway?

Making fractals

Sierpinski Triangle

Using Java

Math questions

Sierpinski Meets

Pascal

Jurassic Park Fractal

Using JAVA

It grows complex

Real first iteration

Encoding the fractal

World's Largest

Koch Snowflake ✓

Using Java

Infinite perimeter

Finite area

Anti-Snowflake

Using Java

Fractal Properties

Self-similarity

Fractional dimension

Formation by iteration

For Teachers

Teachers' Notes

Teacher-to-Teacher

Comments

My fractals mail

Send fractals mail

Fractals on the Web

You may print and use this [triangle grid paper](#) to help you with this drawing.

Step One.

Start with a large equilateral triangle.

Step Two.

Make a Star.

1. Divide one side of the triangle into three equal parts and remove the middle section.
2. Replace it with two lines the same length as the section you removed.
3. Do this to all three sides of the triangle.

Do it again and again.

Do it infinitely many times and you have a fractal.

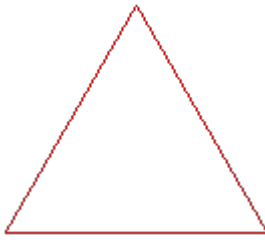
Want to take a long, careful look at what it looks like?

See a few of the steps below.

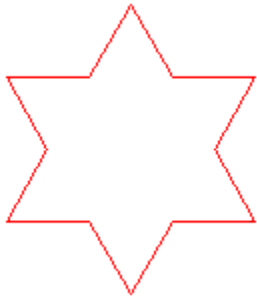
[The Math Forum](#)

Other Math Lessons

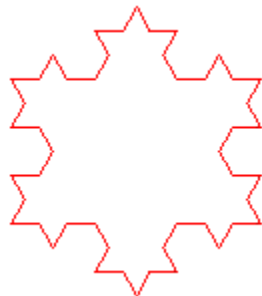
[by Cynthia Lanius](#)



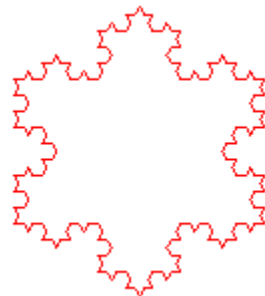
Step One



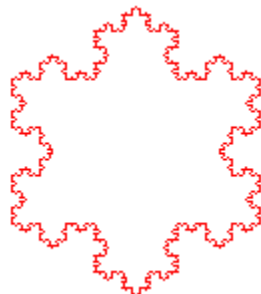
Step Two



Step Three

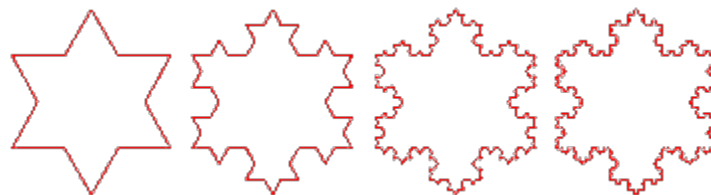


Step Four



Step Five

Let's see them all together



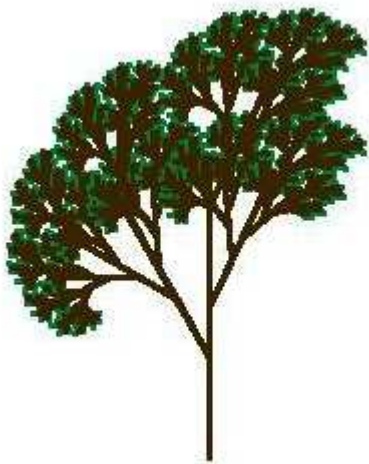
Copyright 1996-2007 Cynthia Lanius

URL <http://math.rice.edu/~lanius/frac/koch.html>

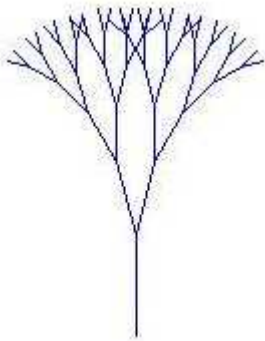
Make a Recursive Tree

This assignment must be done using the BYOB (Build Your Own Blocks) version of BYOB/SNAP. 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!)

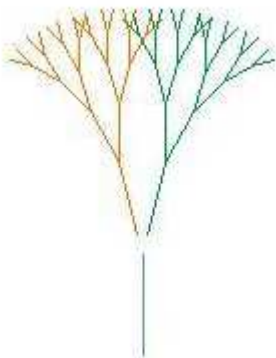
Our goal is to draw a tree like this:



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:



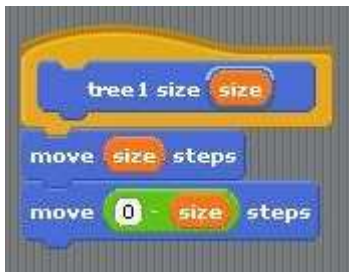
We're going to create a TREE block in BYOB BYOB/SNAP. It'll start with a MOVE block to draw the trunk of the tree,

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.

Tree1	Old Tree2	New Tree2a

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

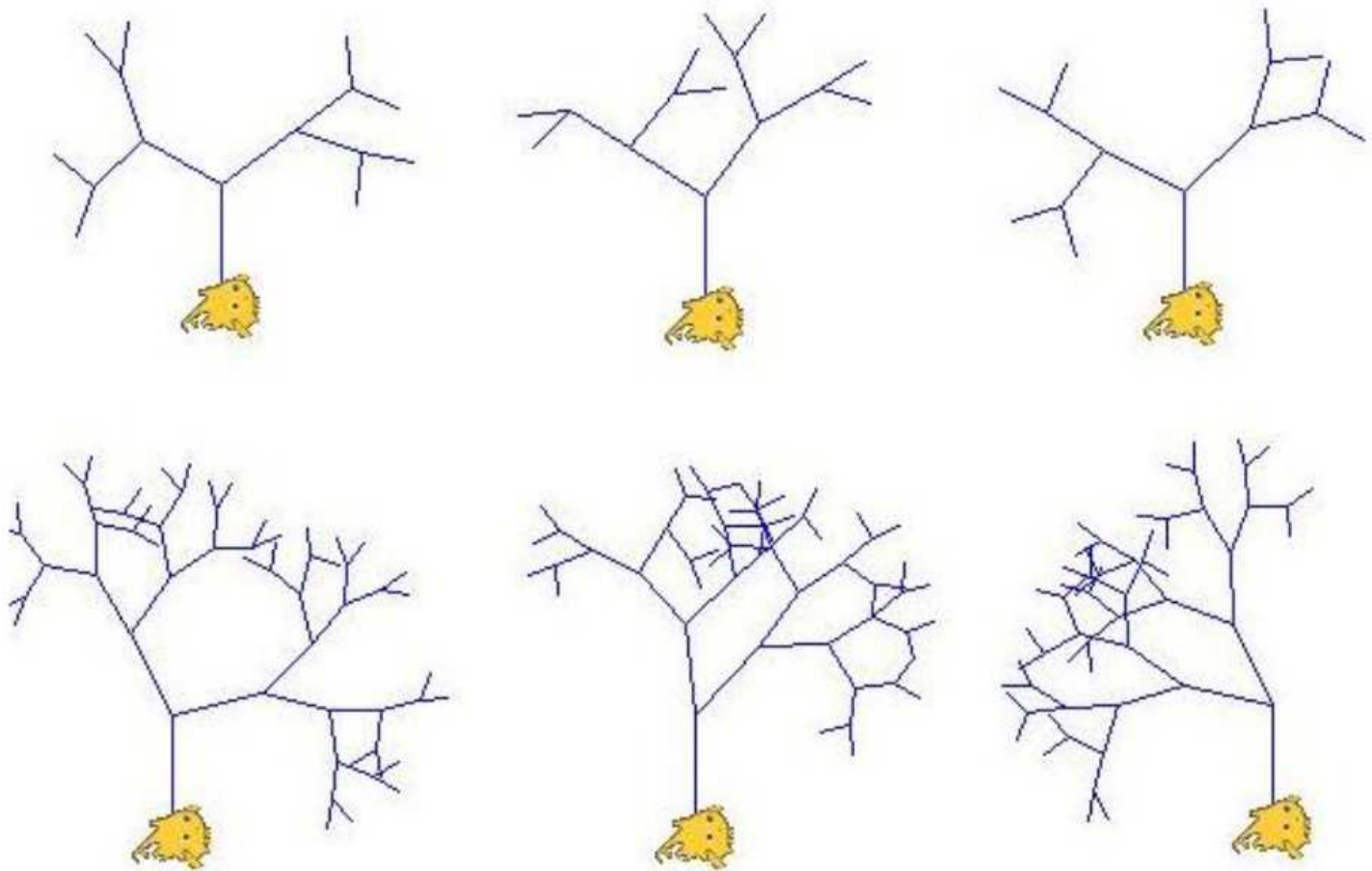
Change your tree in the following ways. You should probably try these one at a time so that you can always tell which change you made caused a new bug.

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

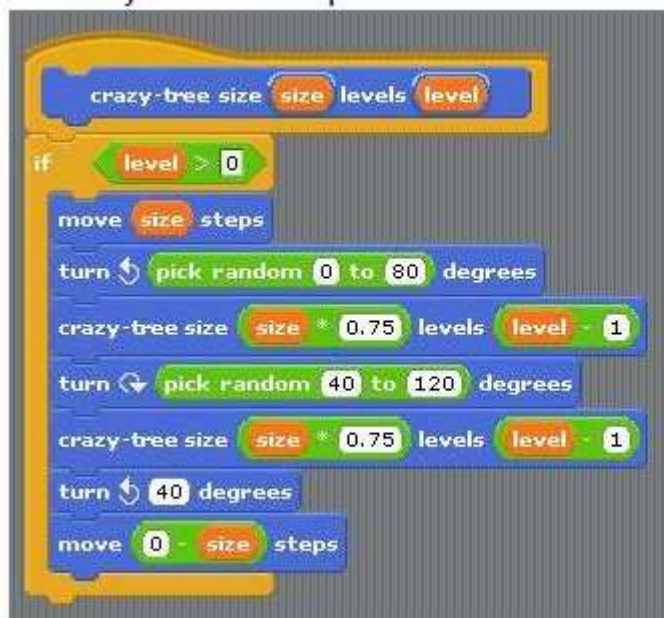
Note: If your character goes off the screen your image might get messed up because you'll tell your character to change their y-position but the character literally won't be able to.

Random Tree

We want to make a tree that looks more random. Here are some pictures of different runs of the program with different levels so that you can see what we were thinking.



Unfortunately our attempt (shown below) was unsuccessful (see “tree” on the right). Modify the crazy-tree code to produce the random trees as shown above.



Base Cases

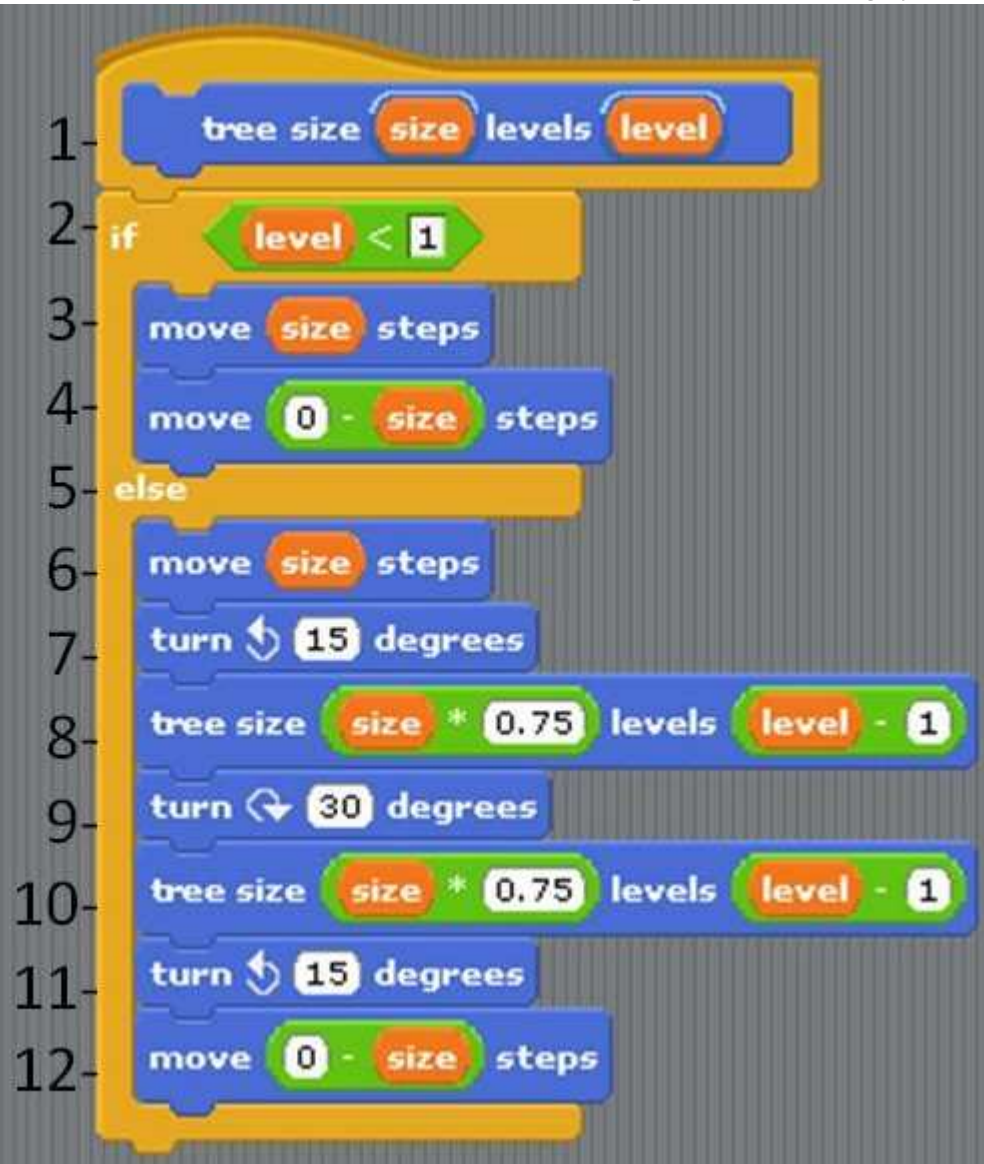
Now that you have some experience with some recursion, we're going to point out some of the anatomy of a recursive procedure.

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.)



[Curriculum \(/bjc-course/curriculum\)](#) / [Unit 7 \(/bjc-course/curriculum/07-recursion\)](#) /

[Lab 2 \(/bjc-course/curriculum/07-recursion/labs/02-recursive-bunnies\)](#) /

Recursive Bunnies

Use BYOB to create a recursive procedure to count the number of bunny ears given a number of bunnies entered. Use the Factorial or OddNumSum program as a guide. Remember, there are two ears per bunny (unless you have “different” bunnies).

Your code **MUST** be recursive - You should have a function (block) to count the ears that calls itself!

HINT

Looking at Factorial in our example and OddNumSum we can see that our recursive call should look something like:

The Number of Ears + BunnyEars (Number of Bunnies - 1) where BunnyEars is my recursive function block name and I have a variable to represent “The TOTAL Number of Ears” and the “Number of Bunnies” and the “Number of Ears” would be 2...

because we want to add 2 (or 3) ears to the total of the smaller group of bunnies. Right?

EXTRA POINTS

To earn extra points modify your script code so that:

The ODD number bunnies have 3 ears and the EVEN number bunnies have 2 ears.

For example: I have 5 total bunnies. This means that bunnies #1, 3, and 5 have 3 ears each for a total of 15 ears and bunnies # 2 & 4 have 2 ears each for a total of 4 ears. So my output would say “There is a total of 19 ears.”

By the way, you can have 2 recursive calls inside your script as long as there is conditional execution. In other words, If _____ do the recursive call this way Else do the recursive call that way.

Turn in Your Work

- Create a page titled “Recursive Bunnies” under your Classwork category and describe your logic in creating your recursive procedure.
- Upload your Recursive Bunnies file (.ypr)
- Add the Screenshots to a Word doc with your description. Submit on Moodle.