

Intro to Recursion

Warm-up/Definitions:

1. What is the general idea behind recursion? That is, if I give you some block in Snap! (and let you see the code that makes it up), what is the quickest way to identify if the block is recursive?
2. What are the two cases to deal with in most recursive problems?
3. In the second of the above cases, what is Dan's 3-part strategy for reaching a solution?
4. What is meant by the idea of the recursive leap of faith?

General Strategy for Approaching Recursive Problems:

1. *Understand what the problem is asking.* There is no use jumping right into a problem without first understanding what it is asking. For example, Dan covered the Fibonacci function in lecture. If you were asked to code this, it is a much better approach to first write down and figure out how Fibonacci works mathematically and then attempt to code it in Snap!, rather than trying to understand how it works along the way.
 - a. What are your inputs and outputs?
 - b. If you can draw a diagram to help visualize the problem, do so!
 - c. If you are dealing with a fractal, identify instances of the previous level in each subsequent level.
2. *Tackle the two cases.*
 - a. In general, start by asking yourself if there is a simple base case. Oftentimes, though not always, this will be the case. Ask yourself, "What is the simplest possible case of this problem?" If you are unable to think of one, then go ahead and move on to the recursive case, as with some problems the base case becomes more clear afterward.
 - b. The recursive case tends to be more challenging, but it too can be conquered, provided you have faith—faith in the recursion. What does this mean? Remember that the whole point of recursion is to break a large problem into smaller versions of itself. When you are in the midst of mapping out a solution to the recursive case, ask yourself the following questions: "Is there some function that would be very useful to me here? Is it possible that this is the very function which I am currently writing?" If it is, you may as well use it! This may sound a bit confusing at first, but the idea will become more clear as you do more examples.
 - i. **A Cautionary Note: Many students, especially in future CS classes (where the recursion problems become harder), fall into the trap of thinking that the recursive leap of faith means that no work is necessary on your part as long as you trust your block (or function, in text-based languages). Then, they will wonder why their code does not work, ever so desperately and naively insisting that they took the leap of faith. Remember the leap of faith is just a tool to help you solve the problem,**

and simply means that you can assume your function works well enough for you to use it in a carefully and properly constructed solution. It does not mean that you can write any somewhat accurate solution and expect it to work just because you took the leap of faith. The best way to think about it is as follows: your block will do its job, but only if you do yours.

3. *Patterns are a thing—find them.*

- a. As you expose yourself to more problems, you will find that many follow similar patterns. Here are a few to keep an eye out for in your two cases.
- b. Common Base Case Patterns
 - i. *List Problems*: Very often, the base case will just involve reporting an empty list.
 - ii. *Math Problems*: The base case will often just be some predefined value. For example, in factorial, $0!$ And $1!$ Are both defined as being equal to 1.
 - iii. *Other Problems*: Just think about the problem outside the context of programming, and chances are you will be able to work it out.
- c. Common Recursive Case Patterns
 - i. *List Problems*: Reporting an item in front of the recursive block called on all but the first item of the list (or just calling the recursive block on the rest of the list, if you do not want to keep the first item).
 - ii. *Math Problems*: Many times, the mathematical function may just be defined recursively, such as Fibonacci.
 - iii. *Other Problems*: If you are struggling, just consider how you would solve the problem if you didn't have to code it, and try to somehow massage that pseudocode into something which uses a recursive call.

Recursive Thinking Practice:




1. Your job is to figure out how many total apples there are in an apple tree. This happens to be a really aesthetic apple tree. Each branch has more branches, which in turn has more branches, and so on and so forth. Now, you could definitely just traverse all these branches one by one and count the apples, but that would take a while, even with your surely amazing climbing ability. You know one thing: Each branch at the first level of the tree has the ability to tell you how many total apples it has (this includes the apples on every eventual branch that stems from it. What should you do?
2. Murtz has kidnapped the rest of the CS 10 staff and is holding them hostage in Sudartja Dai. Rumor has it he is forcing them to do CS 70 problems day and night, and their only source of entertainment and comfort is a stuffed Alonzo. Luckily, Mansi was able to get a message out to you on Piazza, and you saw it before Murtz deleted it. Unfortunately, a lock has been placed on the lab door. The only way in is to solve the following problem: *Design a recursive algorithm which takes in a list which can contain anything. Your algorithm should keep only the Fibonacci numbers from this list, and return a new list with these numbers squared. Assume you have a block “FIB?” which reports if a number is a Fibonacci number or not—but it errors if you pass in anything other than a number. You*

may not use map, keep, or combine. Also, you do not have to describe the code precisely in Snap!—pseudocode is fine.

Challenge Problem: Summer 2018 Midterm #15

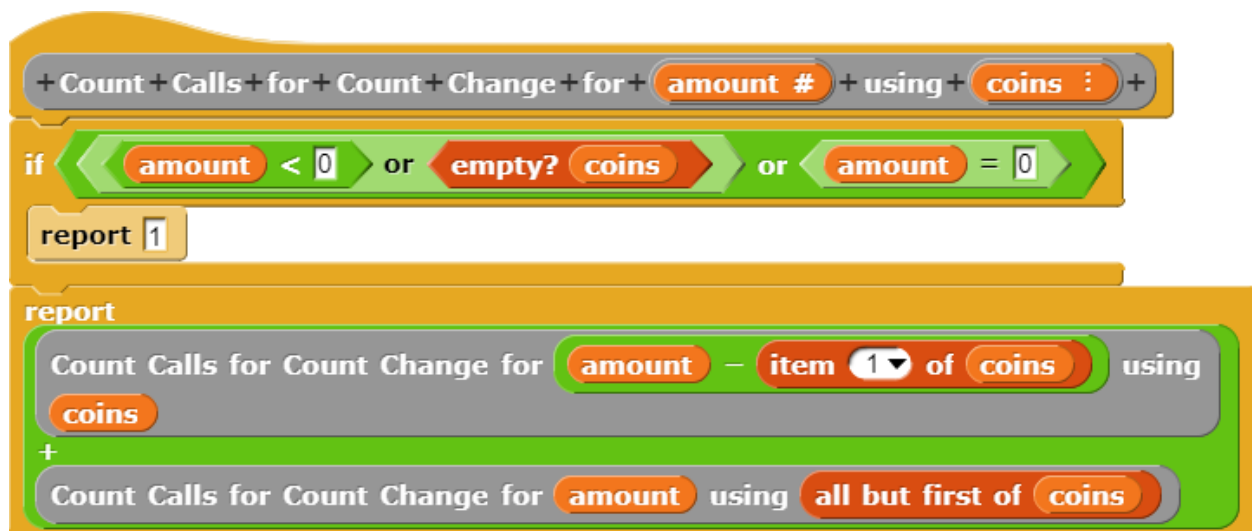
Question 15: Count Change It Up (9 points total, 25 min.)

As we're sure you'll agree, Count Change is a pretty great function. But unfortunately its runtime is terribly inefficient. To demonstrate this, we've tried to write the block "Count Calls for Count Change." It takes as input an amount (in cents) and a list of coins, and should output the number of calls (recursive or non-recursive) to Count Change required to compute a result for these inputs. For some example calls, see the table below:

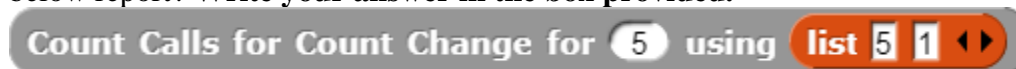
Function Call	Explanation
	Since counting change for amount=0 is a base case, this only requires one call to Count Change (the initial call).
	This requires three total calls: the initial call, and recursive call with amount=0 and coins = [1], and the recursive call with amount = 1 and coins = [].
	Yeah, so...it's inefficient.

Question continued on next page....

Below is our attempt to implement Count Calls for Count Change. Unfortunately, though, it has a bug.



a) (3 pts) Ideally, if Count Calls for Count Change is working properly, what should the call below report? **Write your answer in the box provided.**



b) (3 pts) Using the buggy implementation above, what will the call below report?



c) (3 pts) Below, describe how you can modify our buggy version of Count Calls for Change so that it works properly. **Note: It is possible to fix the code with a very simple modification. If your answer is unnecessarily long or complex, it may not receive full credit.**