Edison Ying

Recursion Supplemental Text

The scenario that we have is that you are a cashier, and we are designing a system to tell the cashier the least amount of coins needed for a customer's change. We also want to tell the cashier which coins and how much of those coins will be given back to the customer. There will be an assumption that there is an infinite number of each type of coin available to the cashier.

What is the overall narrative you are attempting to portray?

The overall narrative I am attempting to portray is that we are repeating a task multiple times until we find a case that either works or a case that can no longer work no matter how many times you repeat the task. In my example, with each arrow we are asking the question: Do we have the correct amount of change? If yes, then that branch is complete and we have a way to make our desired change value. If not, then we continue by adding another one of each coin and repeating the same question. In order for us to not get stuck in an infinite loop, if our current value is greater than the desired amount, we can no longer continue. The check marks in my visual represent the base case, which should exist in all recursive functions.

How might your visuals be misconstrued to have a different meaning? What might those alternative meanings be?

(Note that page 1 of visual describes the problem at hand while page 2 visualizes the recursive program). My visual can be interpreted more like a layer by layer approach instead of the actual recursive approach (branch by branch) that our program has. Since numbers could not be used, it would be difficult for others to know which part of the program runs in which order. Recursion goes through one side of the tree before moving on to the next which is why it is great for finding all combinations.

Another way my visual can be misconstrued is what each “layer” represents. Each one of the layers represents adding another coin to the one above. For example, on the left side of layer 2 of the tree, we see that there are 2 nickels and right above it there is 1 nickel. Some people might sum those two values together and interpret that as actually having 3 nickels on the 2nd layer where in reality, the sum of values is what is already being displayed. This misinterpretation is usually more common among programmers as there are many other recursion examples where what is being displayed is not the sum and instead, the entire branch is the sum.

What issues did you run into during:

- **The coding process?**

Debugging was the hardest part of coding my recursive program. Oftentimes the code would return results that were not intended, but it would be very difficult to trace why they were the wrong results. There were only a few lines that needed to be checked, but the difficulty in understanding the path that the function was going made it very difficult to find errors.

Another strange issue that I ran into was the addition of floats in Python. To a human 0.1 + 0.1 = 0.2, but there were cases where it actually equaled 0.1999999 according to the machine which would throw off many cases.

Finding the base case and how the function could be repeatable was also a bit of a challenge. For the base case, it is the fact that the program does not stop, but it has to jump backwards in the tree and return the result to the previous layer that can get complex. When looking at how the function can be repeatable, it is generally a good idea to be efficient when passing through parameters. For example, the layer of the function does not need to be its

separate parameter as it can be calculated using the original parameters. Once you are able to find how the function can be repeated, the programming part is generally easy if you have a bit of experience.

- **The development of the visual above?**

The visual goes the opposite way compared to our actual program, but it conveys the same message as our program. The visual uses a bottom-up approach vs our program having a top-down approach. The reason for this is because it is simply easier to demonstrate how the program works using a bottom-up recursive tree. When designing the visual, I first had to understand what the program was doing and how it was functioning which was a bit complex. I had to print out which values were being calculated first in order to fully grasp the way the recursion algorithm traversed. Being organized on the tree was also an issue as I did not want random arrows pointing to random spots which would make it difficult for someone to understand the visual.

What are some other common pitfalls of recursion that you did not run into?

The most common pitfalls of recursion is the time and memory it requires to function.

If the solution you are using is not the most efficient, you could needlessly calculate many possibilities that would be irrelevant to the solution. For example, in my example, we use recursion to get all possible combinations of numbers that could potentially add up to our desired value. Even though this will always work, for larger values, the time it takes to calculate the results will be very long which is not ideal. The common solution to this is to use memoization or “caching” known values in an array. If we know that we can make 6 cents with a 5 cent coin and a one cent coin, then there is no point for our computer to use 6 one cent coins as it will always require more coins.

The other issue with recursion is the memory that it takes up. Calling a function multiple times will use up more memory than an iterative loop. Every layer in the recursion tree we made has multiple calls of the same function in it. When we are using the top-down approach, each function has to store the results of the function that it calls. This process of storing functions instead of an integer is what uses up so much memory on our computer.

Many languages will also have a maximum recursion depth, which is the maximum amount of times we can call a function in sequence. In Python3, this number is 1000 by default. This is because each function is an object, and storing too many of these functions all at once will use up memory. In most cases where the maximum recursion depth is reached, there is usually an infinite loop which is why languages stop the program to prevent your computer from crashing. However, in rare scenarios where you need more than 1000 call stacks of your function, an iterative solution will likely be much more efficient. (You can modify the maximum recursion depth though when you need to do so, consider using another approach.)

Finally, the complexity of recursive functions can be vastly range in difficulty. Because we as humans like to visualize our programs going in sequential order (most of the time), it can be complex to debug recursive functions as it does not follow the order that we are used to when we use ‘for’ or ‘while’ loops. In fact, a recursive tree usually completes one side of the tree before moving on to the other side instead of a layer by layer approach. This is also the difference between depth first search and breadth first search. However, since I already had some experience with recursion problems, developing the recursive function was not a major issue. Though for many people and their first time creating recursive programs, it would be challenging to grasp the concepts of how recursion works.

Edison Ying

Works Cited

[1] “Find minimum number of coins that make a given value,” *GeeksforGeeks*, Aug. 13, 2015. https://www.geeksforgeeks.org/find-minimum-number-of-coins-that-make-a-change/

[2] “Generate a combination of minimum coins that sums to a given value,” *GeeksforGeeks*, Sep. 15, 2020. https://www.geeksforgeeks.org/generate-a-combination-of-minimum-coins-that-results-to-a-given-value/