Planning Document – Nicolas Sanchez Assignment 2: Tree House Walking

Planning Phase

The planning phase of this program was by far the longest stage while also not having that much to talk about. It contained me just reading through the pdf and first trying to understand what it was asking, then how I was going to get the main permutation function into code. Trying to understand the goal of the program wasn't too bad, but because at the beginning of this I didn't fully understand how to write a permutation function, I spent a long time trying to figure out how I was going to translate everything into code. Eventually, I got it after watching a million videos on youtube about permutations with recursion and staring at my code for hours (ms paint is super helpful).

Assistance Received

Surprisingly, I didn't ask any of the TA's for help on this assignment. This was mostly because when I did need help, I had no idea what to ask. Once I understood what I didn't understand about permutations and what the program was asking, I pretty much got it all.

Debugging Phase.

Here's the thing with this code. It's fairly simple to write once you understand it, but if you don't write it efficiently, it's going to be extremely slow. Because we are working with factorials, the increase in calculations for the number of trees used is insane. The number of combinations increases exponentially. But luckily, because of how the question is written, there is a way around that.

1. The first major optimization. You do not want to calculate distances for pairs that are the same. This optimization knocks out half of the distance calculations done in the pre-comp function.
    a. A pair of trees (2, 2) would be a wasteful calculation because they are the same tree and the value would just be 0. So skip over pairs where x = y.
    b. A pair of trees (1, 2) and a pair (2, 1) are the exact same thing. We only need to calculate this once in order to get the distance of this pair. So skip over pairs where x > y.
2. The second major optimization, and the more important one, is preventing the creation of permutations that are the same thing. These knock out a ton of repeated calculations.
    a. The first part of it checks the tree when finding a pair. It applies a pair formatting rule that says, if the x > y value, skip this tree. This is done to prevent the calculation of a repeated pair. For example, (1, 2) and (2, 1). Every pair will have a match like this, so we skip over the others and only keep 1.
    b. The second part of this checks each tree after each pair. It works the same way but instead of checking the y, you check to see if the x value in the current pair is greater than the x value in the previous pair. If it is not, skip this tree. For example, (1, 2) (3, 4) and (3, 4) (1, 2). The second one is the same, so we get rid of it by applying a second formatting rule.

Testing Phase

My testing phase was mostly targeting the runtime of my program. I made test cases based on the number of trees in order to test how fast my program could calculate them. This time I could only think of one obscure test case. I took the sample in and just changed the order of the tree coordinates to confirm That I would still get the correct answer. This test case found a bug in my code that revealed that my entire sum function was incorrect.

After making my code look nice and removing any nonsense whitespace, I went through my program like usual and traced it all to reinforce my understanding of my own code. This stage was a lot faster this time because the code is pretty short. After this, I just did a final test to make sure everything was working fine.