**Problem 1: Collaborators**

**Problem 2: Fair and Square**

a)

The maximum amount a friend can get is the total sum, which occurs in the scenario where only one coin of denomination is given. In that case, one of the two friends will get the coin while the other gets nothing.

b)

Ideally, we would want to divide the total into two equal parts, . If we think of the problem in this manner, it becomes a version of the Knapsack problem, since have to choose coins in such a manner so as to get as close as possible to without crossing it. As such, in the best case, there will be no difference between the amount the two friends will get, meaning the total amount can be equally divided. In the worst case, the difference will be the denomination of the coin with the greatest value, meaning everything except that coin could be divided equally. In part a, we essentially saw this worst case.

**Problem 3: Parenthesization? Nope!**

a)

The running time for the recursive version of this problem can be found through analysis.

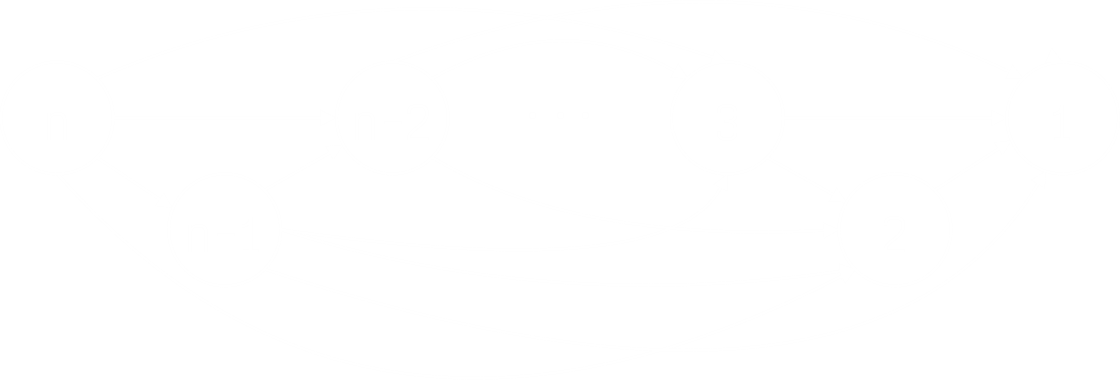
From this pattern, it is easy to see that . We can analyse this further to find the exact time complexity.

Following this, we will eventually reach the base case, which is . Thus,

b)

The -th call makes calls to each of the smaller values of . As such, every number from to is dependant on every number smaller than itself. Thus, there is no possibility of cycles forming. The only independent call is , which returns . As such, that is the base case.

The DAG for the algorithm will look somewhat like this:



**Problem 4: Longest Paths**

This problem can be solved using a prefix approach if we considered the line of capsicums to be a sequence.

Algorithm

For the -th plant, we have two choices. We can either choose to keep the plant or we can choose to uproot it.

If we choose to keep it, then we get to keep the capsicums it has grown. However, we cannot keep the -th plant, which we must uproot. Thus, by keeping the -th plant, the total number of capsicums we have for the first plants is plus the maximum number of capsicums that can be saved for the first plants.

If we choose to uproot it, then we lose the capsicums with it. As a result, we can keep the -th plant if we want to. Thus, the total number of capsicums we have for the first plants is the same as the maximum number of capsicums that can be saved for the first plants.

We have to find the maximum number of capsicums that can saved between these options. Thus, . Of course, we will start from the last possible plant, meaning the first function call will be , assuming the plants are numbered from to .

We also need to consider the base case. There are two possible base cases here, and . is just the maximum possible number of capsicums that can be saved for the first plant. Obviously, this is . , which is called from , is the maximum possible capsicums that can be saved for plants. Again, the answer is quite obvious – .

Of course, we are memoizing the results of all function calls so that repeated function calls can just use the saved results.

Algorithm Proof

At every call to , we compare the only two possibilities for the -th plant, keeping it or not keeping it. We take the maximum value obtained between these two choices. Since this process is repeated for every value of , we find the maximum value obtained for each possible set from to . Thus, we can find the maximum value obtained from the first plants.

Time Complexity

There are a total of subproblems, one for each of the plants. At each subproblem, we are only performing a comparison between two values obtained from calls to other subproblems. As such, the time complexity of an individual subproblem is . This makes the total time complexity of the algorithm . The algorithm is linear in terms of time complexity.