COP 4516 Spring 2023 Week 10 Team Contest #2 Solution Sketches

The n Days of Christmas

The n^{th} Triangle Number is the sum of the first n positive integers. It's equal to $\frac{n(n+1)}{2}$. This question asks you to find the sum of the first *n* Triangle Numbers. It's best to pre-compute these sums (there are a million possible queries) and then answer each query. Note that the millionth Triangle Number itself easily overflows int, so longs must be used. When pre-computing, just store results in an array and build results as follows: array[i] = array[i-1] + t(i), where t(i) represents the i^{th} Triangle Number.

Counting Sequences

The solution to this problem is similar to a solution of the longest common subsequence problem. Let the input strings be called s and t, respectively. Let dp[i][j] be the number of times the string t[0..j] appears as a subsequence in s[0..i]. Now, consider calculating dp[i][j].

If s[i] == t[j], this means that we can build sequences by matching these last letters. The number of these sequences is simply dp[i-1][j-1], since we must build the previous j letters of t from subsequences of the first i letters of s.

In all cases, old appearances of t[0...j] should still count. There are dp[i-1][j] of these, and these should be added in in all cases.

Finally, care should be taken care of in initializing the DP table and making sure no array out of bounds errors occur. The correct result should be stored in the entry dp[s.length()-1][t.length()-1].

Lenny's Lucky Lotto Lists

Let dp[i][j] store the number of lists of i values with all numbers less than or equal to j. Certainly, dp[i][j-1] stores many of these lists. In fact, it stores all of these lists that don't contain j. Thus, we must just add to it the number of lists that do contain j. Well, if our list ends in j, and this number must be at least twice the previous number, then the maximum value of the rest of the list is j/2. Thus, we want to add the number j to a list of size i-1 with maximum value j/2. This value is simply dp[i-1][j/2]. So, the recurrence looks like this: dp[i][j] = dp[i][j-1] + dp[i-1][j/2]. As always, care must be taken with base cases and to avoid array out of bounds.

Name Tag

The data is small enough that we can just try all cyclic rotations of the input string and simply keep the one that comes up first alphabetically. In Java, the String method substring is very useful in quickly coding a solution that calculates all string rotations. Of course, compareTo will be used as well to easily identify which string comes first alphabetically.

Robot Challenge

Let dp[i] represent the minimum score ending at target i. There are several options for ending at target i. In fact, all options comprise of any target j < i, being the previous target we visited. We want to try all of these j's and see which one is best. In some sense we have:

dp[i] = min over all j of [dp[j] + cost(j, i)]

where cost(j, i) is the cost of moving directly from target j to target i.

Specifically, this cost is the distance between targets j and i, plus the sum of the penalties of the missed targets.

Notice that to calculate dp[i], we must loop through all possible values of j, thus, this is at least O(i) work, where i ranges from 1 to n. BUT…for each different value of j, if we calculate the cost function from scratch each time, this is an extra cost of O(i-j). Notice that the sum of i-j over all possible values of j actually equals $O(i^2)$ and that the sum of $O(i^2)$ as i ranges from 1 to n is really $O(n^3)$. With n = 1000, this is too slow.

Thus, we need a speed up to get this to run in time. The key idea is improving the calculation of the cost function. It would be nice if this could be calculated in O(1) time. One thing we can do is simply precompute this function for all pairs j and i. Consider fixing j. We can in turn, calculate cost(j, j+1), cost(j, j+2), cost(j, j+3), …, cost(j, n-1) each one just taking O(1) time because we can build the answer from the previous answer. An easy way of looking at this: consider calculating a cumulative sum of penalty points, starting at target j+1. Thus, sum(j+1) = penalty[j+1], sum(j+2) = penalty[j+1] + penalty[j+2] and so forth.

Then the cost function is as follows cost(j, i) = sum(i-1) + dist(j, i).

So, to solve this problem, pre-compute the cost function and store the results in an array. (Or you can precompute the cumulative sums of penalty points and calculate the cost function on the fly.) Then, the DP solution stated above runs in $O(n^2)$ time because the pre-computation takes $O(n^2)$ time and the DP itself takes $O(n^2)$ without the extra time for recomputing costs.

Tricky Tolls

The key to this problem is to realize that the relevant information necessary to solve a subproblem is knowing which toll booth you are at, and at which minute (0 to 59) you arrive at that toll booth. We can define f(booth, minute) to be the minimum cost to arrive at toll booth booth at the designated minute, or we could define it to be the minimum cost to move from toll booth booth at the designated minute to the end of the drive. The former designation is probably easier to use if a traditional dynamic programming solution is implemented (nested for loops) while the latter designation is probably easier if memoization is utilized where the function f is written recursively with a memo table to speed it up. When solving the problem for a single state, we only have two choices: move immediately or wait at the toll booth till the change of the half hour to get a different toll. In the solution, try both options and just take the one with minimum cost.