

COP 4516 Spring 2026 Week 4 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: $\text{array}[i] = \text{array}[i-1] + t(i)$, 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 $\text{dp}[i][j]$ be the number of times the string $t[0..j]$ appears as a subsequence in $s[0..i]$. Now, consider calculating $\text{dp}[i][j]$.

If $s[i] == t[j]$, this means that we can build sequences by matching these last letters. The number of these sequences is simply $\text{dp}[i-1][j-1]$, 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 $\text{dp}[i-1][j]$ 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 $\text{dp}[s.length()-1][t.length()-1]$.

Lenny's Lucky Lotto Lists

Let $\text{dp}[i][j]$ store the number of lists of i values with all numbers less than or equal to j . Certainly, $\text{dp}[i][j-1]$ 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 $\text{dp}[i-1][j/2]$. So, the recurrence looks like this: $\text{dp}[i][j] = \text{dp}[i][j-1] + \text{dp}[i-1][j/2]$. As always, care must be taken with base cases and to avoid array out of bounds.

Minesweeper

No recursion is necessary here since you aren't implementing a recursive clear. Instead, you just go through the grid, looking at each '.' square. For each of these, just do a loop to all neighboring squares via a DX, DY arrays for the eight possible directions, counting the number of stars around you. Then just store this result. You just have to watch out for array out of bounds and taking care with char vs. int issues since the board might be a 2D array of characters and you might want to store the character version of each number.

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:

$$dp[i] = \min \text{ over all } j \text{ of } [dp[j] + \text{cost}(j, i)]$$

where $\text{cost}(j, i)$ 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 $\text{cost}(j, j+1)$, $\text{cost}(j, j+2)$, $\text{cost}(j, j+3)$, ..., $\text{cost}(j, n-1)$ 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, $\text{sum}(j+1) = \text{penalty}[j+1]$, $\text{sum}(j+2) = \text{penalty}[j+1] + \text{penalty}[j+2]$ and so forth.

Then the cost function is as follows $\text{cost}(j, i) = \text{sum}(i-1) + \text{dist}(j, i)$.

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.