COP 4516 Spring 2025 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].

Editor Navigation

There may be an urge to try a greedy strategy, but ultimately what you realize is that there are several options for moves and the lengths of lines can create some interesting cases that thwart most greedy approaches. A better approach is to simply realize that we can use a breadth first search to map out all the possible cursor positions we can reach with 1 move, 2 moves, 3 moves, etc. from our initial cursor position and we can continue searching until we get to our destination. The hardest part of this problem is properly encoding all of the possible moves. Each of the four moves needs special code and can't easily be taken care of with a DX/DY array. Storing a location is fairly simple: each location is an ordered pair of line number and column number. There are few enough of these that a BFS runs in time.

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.

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.

Welcome Party

One possible solution (and my intended solution) is to notice that we can create a bipartite graph by having nodes for each first name starting letter and each last name starting letter as our two sets. Connect a source to all the first name starting letters with capacity 1 and edges from all last name starting letters to a sink with capacity 1. Then connect edges with capacity one for each name going from the starting letter of the first name to the starting letter in the last name. A maximum flow in this graph is proof that each of the people in the set of the max flow edges between the two groups must have their own separate "team". (For example if E→C, J→P and S→M, this is proof that EC, JP and SM must all be on different teams as no pair of them share either a first or last name starting letter in common.) It follows that the answer to the query is the maximum flow in this graph. The original problem authors also intended an alternate solution to work: by limiting last names to 18 possible starting letters, one can try all possible 2^{18} subsets of teams based on the last names. For each of these subsets, it's easy to determine which first name letter teams are needed. (Basically, once we know the last name teams, find all names whose last names can't be included in any group, then for this leftover group, find how many unique first name starting letters there are. This solves the query for one specific subset of last name letter groups. Iterate through all possible last name letter groups and take the least. The run time of this algorithm is 2^{18} (subsets) x 300 (names) ~ 150,000,000 which is pushing it, but just barely within the realm of acceptable runtime.