COP 4516 Spring 2021 Week 10 Team Contest #2 Solution Sketches

The n Days of Christmas
The n\textsuperscript{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\textsuperscript{th} Triangle Number.

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:

$$dp[i] = \min_{j \lt i} \{ dp[j] + \text{cost}(j, i) \}$$

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 $\text{cost}(j, j+1)$, $\text{cost}(j, j+2)$, $\text{cost}(j, j+3)$, \ldots, $\text{cost}(j, n-1)$ 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.

**Spider-Man’s Diamond Head Dilemma**

The graph is relatively small, so we can just check each requirement in a straight-forward manner. We must check that the graph is bipartite in the manner that is stated and connected. So, first check that no edges connect odds to odds or evens to evens. If this is true, it's bipartite. Then, just run a DFS or BFS to see if the whole graph is connected.

**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 $E \rightarrow C$, $J \rightarrow P$ and $S \rightarrow M$, 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.