COP 4516 Spring 2025 ChatGPT #1: Tries (Solution Sketches)

Dot Game Dominator

Once again a greedy strategy can be used here. You want to eat the largest dot that is strictly smaller than your current size, since at each step, this makes your size grow the most. No competing sequence which eats a smaller dot can beat this greedy strategy. To implement this, a TreeSet can be used, since the class TreeSet has a method lower. However, be careful though - TreeSets don't store duplicates and an error will be caused in situations where there are multiple dots of the same size. To avoid this error, create a TreeSet of objects, where each object stores both the size of the dot and its ID number (you assign this as you read in the dots to be unique). Alternatively, it's very difficult to make data where the number of dots you eat is very large. Or if it is, where it takes you a long time to find the largest dot smaller than you. Thus, a relatively nieve implementation which maintains a sorted list of the dot sizes will pass the data that was generated for this problem.

Ground Game

This question is the banger in the set. Keep a variable that tracks how many levels underground the player is and keep a second variable that stores the maximum levels underground the player has gone. When processing the input, ignore characters moving left(<) and right(>), add one to the count when moving down (v), and subtract one to the count when moving up (^). After each change to the counter, see if the new value is greater than the previously seen maximum. If so, update the maximum.

Strange Lottery Simulator

A regular trie where you store the number of words stored in each subtrie can handle the query asking for the number of participants with a particular prefix. However, if the names can be reversed, that same trie is not helpful. However, what can be done is that two ***different*** tries can be maintained simultaneously. Notice that reversing a name twice just brings it back to its original form. Thus, Whenever a name is added, add it to a trie forward as is and ALSO add its reversed version to a trie reverse. (So, "sharon" would get added to the trie forward AND "norahs" would get added to the trie reverse.) Finally, whenever you receive a query, you just need to know if the current state of the names is regular or reverse. Just keep either a boolean variable or an integer that toggles between 0 and 1, and then answer the query in the appropriate trie based on the value of the variable storing the current state of the names (forward or reverse).

Tree Sales

We first note that a single query could be very expensive if we just stored each person's sales in their node in the tree structure. Then, to answer a single query, we would have to visit each node in the subtree of the queried node, which could be up to 100,000 steps and this could happen 100,000 times. (In reality, it wouldn't be this big, but we could still get close to 50000^2 operations total over 100,000 operations.) Thus, we need an idea to speed up a query. What if we store the ***sum*** of ALL nodes in a subtree at the root of that sub tree? So, the old storage method might look like what's drawn on the left and we can store tree sums instead as shown on the right, to speed up our ability to answer queries:

Now, we have O(1) answers to queries (as long as we can access a node in O(1) time). But…how long does it take to update all the values for the tree on the right? Basically, a single sale affects each node on the ancestral path of the node making the sale. So say the node that currently has 11 makes a sale of $20, here are the edits to both tree structures:

Basically, we just had to add 20 to the tree on the right to the nodes storing 11, 37, 55 and 100, respectively. Thus, the new run time of updating the tree for a sale is O(h), where h is the height of the tree. Luckily, the height of the tree is limited to 100 in the problem specification, so each update can be performed in 100 steps and each query in a single step. This gives us a total run time no worse than 100 x 100000 (max operations) = 10^7 steps, which is easily viable for a programming contest.