COP 4516 Spring 2023 Week 7 Final Individual Contest Solution Sketches

Problem A: Anya's Bracelets

This is the easiest problem in the set. Since there is geometric growth, the number of times Anya uses the machine will ever exceed about 30 (since 2^{30} is greater than a billion). Thus, there is time for each test case to simulate the process and stop when there are enough bracelets.

Problem B: Confused Village

The data structure that quickly allows one to access an arbitrary item, change its value and insert it back into the data structure is a balanced tree. In this problem, we need quick access to the minimal health value (of fighting villagers), as well as access to an arbitrary villager's health value. Since the health values are distinct, these can actually be used as keys in a TreeMap (or map in C++). Map each villager's health value to their index. In addition, store the data forwards in a list or array, where health[i] is the health of villager i.

To do queries of type 1: To find a villager in the tree map, first find that villager's health in the list/array. Then, use their health as the key into the map. In this way, we remove this mapping from the TreeMap, subtract out the health points, and reinsert the villager, **if they still have positive health**, back into the TreeMap with the key of this new health value. The parallel change is made to the health array.

To do queries of type 2: Just find the minimal positive health value in the TreeMap, get its corresponding value (villager index), output it, and then remove this entry from the map.

Problem C: Lior's Dictionary

Here was the original intended solution:

Store the dictionary as a trie.

Write a recursive function to solve the problem for each input query. The recursion will traverse the trie, marking unique nodes in the trie that are reachable by following the input query by deleting at most five letters. Once the nodes are marked, we can count how many there are. (We don't want to add the sum of each recursive call because the same word can be formed in different ways via different sets of deletions.)

The recursive function takes in which letter in the search string "we are on", k, as well as how many letters have already been "deleted", del, from the search string. If the number of deleted letters is less than 5, we potentially have two options:

1) Stay at this node, skipping letter k, and adding one to our deleted letters (this is effectively deleting letter k), we pass k+1 and del+1 to this recursive call.

2) As long as there is a link from this trie node on the letter querystr[k], then recursively follow that link, move to letter k+1, but del will stay the same.

In our base case, if k is the length of the string (meaning we've processed each letter in the input string), we "mark" the node if it's a node for the ending of a valid word (ie. if flag is set to 1).

After the recursion is fully done running, the number of unique nodes marked represents the number of words in Lior's dictionary that match the query string.

After the problem was created, we discovered an easier alternate greedy solution. We chose to keep the problem in the set and reward students who discovered this solution:

Given a query string, for example, "restauran" and a dictionary word, "star", loop through the query string, greedily matching characters in the dictionary word. Basically, it's best to match a letter whenever both letters are the same. If the two letters don't match, you can "delete" the character from the query string (ie move forward). If the number of deletions is less than 5 when you complete the query string, then you can do it. Note that the query string must be at least as long as the dictionary word (and no more than 5 letters longer than it), to have a shot at working. Here is how the greedy algorithm matches with each word:

```
restauran
star
```

The reason this works is that we want to maximize our matches. Any competing algorithm that doesn't match a letter the first time there is an opportunity can not "BEAT" the greedy algorithm, because the greedy algorithm will have all the future opportunities (and maybe more) to match letters.

Problem D: Uber Driver

The key to this problem is noticing that we need access to the shortest distance between many, many pairs of nodes, and that we have to reuse that information many times, since we have such a large number of queries. All of this points to the idea of pre-computing every possible shortest distance, and then using that chart as a look up table to answer every query quickly (by adding at most 9 shortest paths to get the total driver path length.) The algorithm that computes the shortest distance between all pairs of points is Floyd-Warshall. So the solution is as follows:

1. Run Floyd-Warshall's once for the input graph for a case, storing all pairs of shortest distances.

2. Process queries. For each query, just add up the shortest distances in the Floyd-Warshall's table for the places to visit. (So if we are going from 2→8→7→4, calculate d[2][8] + d[8][7] + d[7][4], where d[i][j] stores the shortest distance between vertex i and vertex j.)

Problem E: Longest Upwards

Every string can be decomposed into several disjoint upwards. Consider this example:

gi|epqr|dd|c|ahy|mew|eoxz

We can loop through the string a single time, seeing if str[i] < str[i+1]. If so, a streak continues. If not, a new streak begins. Whenever a streak finishes, see if its length is greater than the previous longest. If it is, update your answer. If the streak is the same length, see if it comes first alphabetically, compared to the previous longest streak. If so, update.

If we trace through this algorithm with the example above, we would first store "gi" as the longest upwards. Then, it would be replaced by "epqr". Then the next several upwards are too short (shorter than length 4). Finally, when we get to "eoxz", we compare this to "epqr" and see that "eoxz" comes first alphabetically, and update our answer to this.