COP 4516 Spring 2024 Week 15 Team Final Contest Solution Sketches

Problem A: Advertising UCF

To answer the queries, we must know the number of 'U's, 'C's and 'F's within a range of the current string. The most straight-forward way to answer these queries efficiently is to have a Binary Index Tree for each letter (so 3 BITs in all). Use the given input string to initialize each of the three bits. Whenever a letter is changed, subtract 1 from that index in the old letter's bit and add 1 to the index of the new letter's bit. For queries, just query the same range in all three bits. If the answer is the same in each, print "YES", otherwise, print "NO".

Problem B: Divisibility

The key here is that at most 6 '_' characters will be included in the input string. These characters can be filled in at most 10^6 ways. This means a brute force solution is viable. One can run a brute force solution similar to an odometer by adding in the value of fixed slots and trying all 9 or 10 options in blank slots and adding up the number of ways in which the filled in mod value is 0. Alternatively, one can pre-compute the value under mod of the fixed digits and then try all possible values for the empty slots and see if they sum to the necessary value or not. (For example, if the fixed digits contribute 23 mod 57, then we would way the open digits to contribute 34 mod 57.)

Problem C: Eden Math

This is the second easiest problem in the set. You can code up general base conversion having the letters, 'T', 'I', 'M', 'R' and 'L' hard-coded, or since the data is so small, you can just have five if statements or something of that nature that check for each bit individually.

Problem D: How Many Students in the Class?

This is the easiest problem in the set. Just take three times the first input and subtract the second input, since the class "loses" a student for each team of size two.

Problem E: How Many Integers in Range?

There are always b – a + 1 integers in the range [a, b]. Thus, we can map the problem to finding the smallest and largest multiples of d in the range [L, R]. If a number X is positive, the largest multiple of d that is less than or equal to it is X/d via integer division. But the formula is a bit different if X is negative. It's possible the integer division gives a number that is one too big. Check with an if statement and subtract OR come up with a suitable formula. The other sub-problem that needs to be solved is the smallest multiple of d greater than or equal to an integer X. A similar strategy can be employed here to solve this sub-problem (either a formula with integer division or check an answer and add 1 if necessary.)

Problem F: Sokka's Market Math

All that matters about each price is the cents. Since the input numbers are so big, it's important to read them in as strings and just parse out the last two characters. (Trying to read them as doubles will lead to precision errors.) Once we have this done, then the problem is simply asking for a slight modification to the Knapsack algorithm. Instead of calculating the maximum value of a subset that adds to some given weight, we're looking for the ***number of subsets*** that add to a particular value. We solve this similarly to regular knapsack. When considering a new item, we want to look at either "including it" or "not including it." Simply add the number of ways to do both to get to the new answer to solve the problem. The run-time should be about 100n, where n is the number of items, since the outer loop runs n times (once per each item), and the inner loop just runs 100 times, since there are only 100 different possible "endings" to prices. Mod needs to be used since the sum of two prices might exceed 100 but the appropriate array index would be less than 100.

Problem G: PeekQueue

There are two different judge solutions. The first one utilizes a stack and runs in O(n) time while the second uses a TreeSet and runs in O(nlgn) time. Either runs in time, but an $O(n^2)$ brute force solution gets a time limit exceeded verdict. The stack maintains the relevant tallest people as we process the data left to right. When a taller person is added to the back, we first pop off each person shorter than them from the stack, until we get a person of equal or greater height. That person is our "stopper" and the difference of our indexes in the queue (minus 1) represents how many people the new person added to the back of the queue and can see. Repeat this process as each new person gets added to line. (Each person gets pushed and popped at most once, hence the O(n) run time.) The other judge solution uses a TreeSet and adds people into the set one by one, from tallest to shortest. At any point in time, all the "people" in the TreeSet are taller (or of equal height) to the person being added next. We can do a lower query on the treeset to find the largest index less than your index of a person already in the set (so your nearest stopper), and then perform the same calculation previously described.

Problem H: Red-Black Tree

A tree only has two possible two colorings. Since a tree is connected with no cycles, once you pick a color for the first node, it will fix the colors for the rest of the nodes. One way to think about it is to run a BFS and always color new nodes the opposite color of their neighboring node. Run this from any node and get a count for how many nodes are "red" and how many are "black", initially arbitrarily assigning colors. You just want to paint the more frequent color the with the paint that is cheaper, so all that is required is one BFS that counts the number of nodes at an even distance from the start and odd distance from the start. Then, take the larger of these values and multiply by the cheaper cost and the smaller of these values and multiply by the larger cost. Don't forget to use longs, since we can have 10^5 nodes each with cost 10^5.

Problem I: Grandest Social Gathering

One solution is to keep one map that just maps names to frequencies, followed by an array of ordered sets, where sets[m] stores each name that appears m times and a third set which stores each unique frequency of names. The first map can answer the basic query of how many of a name there is. To figure out the next group, the other two data structures are required. Normally, if there are 10 John's, then we can go to sets[10] and get the next key higher than "John". But if none exists, we need to know the next frequency less than 10 that exists. Then we can go to that set and find the first name in it.

A more direct solution is to store a TreeSet of objects, where the objects are sorted in the manner described in the problem. In addition, the original map of names to frequencies is still needed to quickly handle queries.

Problem J: Triangle: Acute, SKalene

Since the values of x and y can be so large, we want to avoid doubles. If a triangle has three unequal side lengths, then the squares of those side lengths are unequal as well. Secondly, for a triangle to be acute, the square of its longest side has to be strictly less than the sums of the squares of the two shorter sides. If the two sums were equal, the triangle would be a right triangle. As the included angle increases, the opposite side length increases. Thus, our break point is the right triangle and we simply require that the square of the longest side be strictly shorter than a hypothetical hypotenuse to the two shortest sides. Since $n \leq 200$, we can just try brute force, going through all possible triangles, calculating the square of the lengths of the three sides (let these be asq, bsq, and csq) and then checking if asq < bsq and bsq < csq and asq+bsq > csq.