In House 1 - Solutions Writeup

Neeyanth Kopparapu

November 2018

1 Introduction

This is probably going to contain some solution thoughts, motivation, etc that would help guide you to the solution if you didn't solve it, or perhaps give you another way to solve the problem. I'm going to assume that you know all the topics I refer to, as they were previous lectures. I will tell you what you need to know, so if you do not go look at the ICT website and learn it before looking at the solution.

I highly recommend trying the problems if you haven't before looking at these solutions. The contest is still open in the group and will accept and evaluate submissions. Try really hard before looking at these solutions, or else you will be stuck in that vicious cycle of giving up.

If the writeup isn't clear, ask the officer who wrote the problem for a better explanation.

2 Problems

A: Lunch Line
Difficulty: Bronze, Author: Arvind Ravipati

This was a straightforward implementation problem. You just had to check that one of the friend values was a 1, and the sum of the time values was less than 120.

B: Fibonacci String
Difficulty: Bronze, Author: Havish Malladi

Another implementation problem. If you do exactly what the problem says, by finding the frequencies of characters and checking if they form the Fibonacci Sequence, you will get the problem right. To calculate the frequencies, you could have done a few things. A few I can think of:

1. Have a 26 length array, where each element stores the frequency of the individual character ($0 \to a$, $1 \to b$, etc.)
2. Use a HashMap (unordered_map) to store frequencies in key-value pairs

C: Candy Fat
Difficulty: Bronze, Author: Arnav Bansal

This was another implementation problem, but is probably a bit more difficult. There are alot of string algorithms to make this work at $O(N \log N)$, but with the constraints you don't need it to. Instead, just check the number of times the candies appear in the search string. A few ways to do this:

1. Take the $\approx n^2$ substrings and check if they are equal to the strings to count frequencies. This is $\approx O(n^3)$.

2. Using a HashMap again, we iterate through each type of candy. During that time, we keep a dummy string that keeps taking characters of the search string until the dummy string contains the candy, in which we increment the sugar level and then erase the dummy and restart. Here is an example with the search string *absabaab* with candy *ab*:

$$a \to \boxed{ab} \to s \to sa \to \boxed{sab} \to a \to aa \to \boxed{aab},$$

   where the boxed is where we delete and restart. This is approx $O(n^2)$.

D: Havish's Money
Difficulty: Silver, Author: Havish Malladi

When the problem mentions finding the sum of the first i elements of an array, you should immediately think of prefix sums to minimize the query of finding the sum (Prefix Sum Lecture). With this, we need to also know the maximum of the first n elements of the array. Thus, we can just have two arrays: 1 that is just the regular array, and one that is an array of tuples, where the first element is the prefix sum of the first i elements, and the second element is the maximum of the first i elements. The entirety of the array can be calculated in $O(n)$ time. Then, because we can calculate the sum of the entire array (total number of money), and then we can iterate through our second array, find how much Arvind and Havish would have for each element, and keep a minimum value to get our answer.

E: Slider Puzzle
Difficulty: Silver, Author: Arvind Ravipati

If you knew what you were doing, this was an easy implementation problem. If you didn't because maybe you didn't take AI this year (like me), you needed to know some Graph Theory (BFS). Lets go over both solutions:

1. **Parity** Essentially you note that because 3 is odd, moving the _ up, down, left, or right does not change the parity of numbers of inversions in the original string. (Look up what this means) Thus, just count the number of inversions and find the parity and see if it is divisible by 2.

2. **BFS Search** Additionally, you could have done a complete search of all possible outcomes and seen if that was one of them. Technically, you could have pre-done it, and just did the $O(1)$ computation of seeing if it is in the set of valid places, but preloading graphs is illegal in contests, so don't do that.

F: Portal Trapping
Difficulty: Silver, Author: Neeyanth Kopparapu

The intent of the problem's solution was to be a Binary Search problem, where you just had to find the number in an array closest to the queried distance, but it turns out that the solution I ran only passed if you did it in C++, even if the original solution was $O(N \log N)$. In an attempt to make it easier I accidentally let $O(N^2)$ solutions pass, so this problem became very easy.
However, let's assume that we were still looking for the $O(N \log N)$ solution. Essentially, because it asks for the closest element of a fixed array, you should think of using binary search. First, you had to compute the Cartesian distance of each point toward the origin, and then just use standard binary search techniques to determine the closest. This problem became very straightforward, except for the fact that you weren't guaranteed the element you are looking for exists, so you had to find the closest, modifying the traditional binary search slightly by instead looking if l and r were the same, find if the indices corresponding to the values of l and r were 1 apart, then doing casework.

G: Russian Tunnels
Difficulty: Hard Silver / Easy Gold, Author: Arnav Bansal

This problem was a connected components problem in which we find the total amount of separated connected components then subtract one from this total to get the number of tunnels needed to connect the whole graph. By separate connect components, two graphs are separate components if there is no path from any node on one graph to another. Therefore, we begin by creating a graph in which we associate each index with a list of all the indices it has a direct path to. We also create a visited set of all the nodes we already visited. We then loop from node 1 to our final node, first checking if that node is already in our visited set. If it is, we ignore it and move onto the next node. If it is not in the visited set, we first add one to our count, which keeps track of the number of components and we then loop through all the children of that node and all their children and add them to our visited set. This means that we have traversed an entire graph and therefore do not need to worry about these nodes. Finally we subtract from our count after finishing the loop for we need one less tunnel to connect all the graphs.

H: Titanic
Difficulty: Gold, Author: Richard Zhan

After reading through the problem and parsing input, it boils down to the following: You are given a bunch of nodes in a graph, where there is a connection between two nodes if there is no wall between them. Given that one of the nodes is like **flooded**, find how many other nodes would be flooded if water can travel between connected nodes. This is essentially what flood fill attempts to find. After parsing all the inputs, a simple flood fill algorithm will determine how many components were flooded.

I: Driving
Difficulty: Gold, Author: Richard Zhan

At first, this problem seems very daunting. Essentially, if you know shortest path algorithms you would instantly think of Dijkstra's algorithm, but clearly this problem is different in that the edge weights are not the same. There are actually 1 very easy way to do this problem, and 1 not so easy way to do this problem. The not so easy way essentially makes you add 5 to each edge of the graph, so that every 2 moves adds 10 minutes, and then when comparing distances at the end, if the number of edges traversed is odd, you simply subtract 5 from the sum. This isn't easy because you run into many comparison issues when trying to find shortest path to other nodes that isn't the final, so I'm not going to go into this problem in detail.
The easier way that involves more typing is that when doing dijkstras, you keep two queues, one for odd length queries and one for even length queries. By doing this, we can keep track of the shortest distances of both odd length and even length everywhere, so this is just a typical Dijkstra's algorithm now. We just make sure that when looking at an even length, we add the child to the odd length query, and vice versa, making sure to add the 10 minutes when nessecary.

J: Gift Stacking
Difficulty: Gold, Author: Neeyanth Kopparapu

This problem is a simple and fairly common dynamic programming problem. First, we begin by generating all three states of the box. As stated in the problem, there are only three states and not six for rotations of the box are considered the same state. Essentially, this problem boils down to the DP version of the largest increasing subsequence problem after sorting by base area. First, you sort the array by base area and then using a variant of the LIS problem, where the DP step is

$$f(i) = max(f(j) + h(i))\Big|_{j<i \quad w_j>w_i \quad l_j>l_i}.$$