In House 2 - Solutions Writeup

Neeyanth Kopparapu

February 2019

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.

This contest was meant to be **much harder** than the other contests. When writing it we intended for the maximum score to be 5 or 6 questions. I can speak for the entirety of the officer core when I say that I believe you all really got a lot better at algorithmic CS. Last contest, the highest score was a 9/10, and this contest it was an 8/9.

2 Solutions

Note: This is in order of difficulty, not in the ordering of the problems.
Problem E - Havish's Exponents (25 Solves)

This problem seems fairly straightforward - just use Math.pow right? Like all built-in methods, Math.pow is slow: it is $O(P)$ where P is the power. Additionally, if you were just to use Math.pow, you would have to calculate $(10^6)^{10^6}$, which doesn't fit into an int, long, long long, or anything (it would probably break python too). With this, we look towards a "DP"-oriented power system, where we use what we have at a current state to find the next power. Using squares, we can do it efficiently in logarithmic time instead of linear time. Additionally, it's important to note that the modulus operation % is very very quick, and can be use liberally to ensure you don't overflow.

Problem G - Permutations (21 Solves)

Another math problem. If you did this in python you suck. This was meant to teach good modular arithmetic programming practices, but oh well.
To all of the math people, you probably know this. By Fermat's Little Theorem, we know that $a^{p-1} \equiv 1 \mod p \rightarrow a^{p-2} \equiv p^{-1} \mod p$ for any prime p and integer a. Additionally, the quantity $\frac{a}{b} \equiv a \times b^{-1} \equiv a \times b^{p-2} \mod p$. Using the same power formula from the above, we know can do division and multiplication easily mod p.

Now, to calculate the actual value, note that typically there are $N!$ ways to arrange N objects, but this is only if they are all different. Take the example in the sample case: $GLEE$ has 12 ways, but $4! = 24$. So where did the factor of 2 come from? Note that if we actually let the two $E's$ be different, then we would have 24 arrangements of GLE_1E_2. But, there are 2! ways to arrange the 2 E's, and thus we have to remove a factor of 2!. In general, if there are k multiples of any letter, we need to divide the total amount by $k!$. So we just have to find the frequencies of the characters and then divide $N!$ by each of the frequencies through the factorial operator to get our answer, dividing using the modular division described above.

Problem I - Snake City (20 Solves)

After reading the problem, it is evident that it is a graph theory problem. What more, because the edges are bi-directional, if a friend "has" the secret, then that friend has the capability to share it with everyone in his connected component for free. Thus, all we need to do is share the secret with one person in each connected component. Obviously to minimize cost, we do it based on the friend that has the least cost to spread the secret.

Problem C - Winner Winner (19 Solves)

Because of the constraints of the problem, we literally just had to implement the problem. The implementation is just a lot of work, but isn't that difficult $\rightarrow$ you kinda just do it.

Problem F - Grade Bumps (14 Solves)

There are many ways to solve this question. First, it is important to recognize because of the length of the decimal we must store it as a string and not as a double. In order to maximize the amount we get from rounding, we find the leftmost digit that is greater than 4, and round it. You keep doing this until you have the maximum decimal. Note that this isn't $O(N^2)$ partially because you only have check digits that are updating in the rounding again to see if they are greater than 4.

Problem H - Battlecode (7 Solves)

Honestly, I thought this problem was easier than Grade Bumps, but whatever. Note that because we are looking at Manhattan distance, its a lot easier to determine distance now instead of Pythagorean distance, specifically because the Manhattan distance between two points on a grid is equivalent to the length of the shortest path from one to the other. Now, there are a lot of ad-hoc optimizations to the naive $O(PQ)$ implementation of the full search, but consider this simpler solution $O(N^2)$. Have a queue of states, each state indicating the starting castle, and the current node of the search. Perform a multi-sourced BFS, where the queue starts with P states of the starting castle and the starting castle. Now, the first enemy castle we hit is the closest one, and we have the starting castle that corresponds and we can determine the Manhattan distance to finish the problem.

Problem B - Favorite Words (9 Solves)

We can think of the trades as edges in a graph, where each node is a letter. By using Floyd-Warshall, we can find (in a very small time) the quickest way from one letter to another. Thus, we now know the cost of transforming one word to the other (assuming they are the same length). Just find the sum of the costs of the shortest path from the ith character in one word to the ith character in the other. To find the lexicographically smallest word, for each index find the smallest character on the path from the initial character to the final, and you can construct the smallest word.

Problem D - Havish's Homework Help (2 Solves)

This problem deals with range sums and point updates, which is best done using what is known as a Fenwick Tree (or BIT), and many of you recognized this. Unfortunately, this isn't a very easy problem after that. Note that the numbers we are given are in **binary**, and we are looking for the remainder mod 3. Note that for even powers of 2,

$$2^{2k} \equiv 4^k \equiv (4-3)^k \equiv 1^k \equiv 1 \mod 3.$$

, and for odd powers it is $2^{2k+1} \equiv 2 \mod 3$. Thus, all we have to do is initially put the binary digits into our array based on the remainder they give mod 3, so even indices (even powers of 2) stay the same, and odd powers of 2 (odd indices) get doubled (alternatively they get negated). Then, we can sum and get a remainder mod 3?. Unfortunately, we are not done yet. Because the binary string doesn't always start at the rightmost index, an index of k in the actual array doesn't always mean the digit represents the 2^k power in the query, it actually represents 2^{k-r}. However, we can note that we can do the entire sum, and divide the entire sum by 2^r to get the same effect, and like above we've realized that 2^r is either 1 is r is even, or -1 if r is odd, and we can finish the problem.
There are other ways to do this problem. A square root bucketing approach (although not optimal, as it runs in $O(n\sqrt{n})$ time instead of $O(n \log n)time)$ passes in these time constraints. You could use a segtree as well, but those are really your only options.

Problem A - Problem Writing (0 Solves)

I don't think we expected anyone to solve this problem. Note that with the extremely small bounds, literally anything would work, **except** for $O(N!)$. Sadly, note that the naive solution is $O(N!)$ in the sense that we permute the array $0, 1, 2, \cdots n$ and then check if this permutation is valid, where the integer in the ith index would be assigned to the ith problem topic.
Just like how DP can be used to turn exponential problems into polynomial ones, there is a technique called **bitmask** DP that turns factorial problems into exponential ones (obviously not all of them). It works on the same principal, except instead of a state being like a single item, it is a subset of the array $0, 1, 2...$ (note that there are 2^N subsets of the array). We expect this solution to run in approximately $O(N \times 2^N)$.
Essentially, every subproblem $DP[x_1, x_2, x_3 \cdots x_k]$ is the number of ways to assign the first k interns to this subset of the topics. Let $a[i][j]$ be true if the jth problem topic is tolerable by the ith intern. Let S be the set of the DP subproblem described above. Using the formula

$$DP[S] = \sum_{i=1}^{k} DP[S - x_i] \text{only if} a[k][x_i]$$

we can find the final answer, which is the $DP[1, 2, 3, 4, \cdots N]$. This formula works because the only way the previous subset can contribute to this sum is if the kth person is able to pickup the new topic which is the difference from the new subset to the previous subset.
A lot of bit manipulation is required to do bitmask DP. To that end, I would recommend learning and being familiar with boolean algebra if you want to do well in bitmask DP. Also, bit operations are incredibly quicker than addition and subtraction. These are the ones you need to know:

- $>>$ is bit shift.
- & is bitwise and.
- | is bitwise or.
- ^is bitwise xor. (This is a caret)
- ~is bitwise not. (This is a tilde)

As an example, if you have a binary string x and want to remove the 2^i given that x has a 1 in the ith position, instead of doing $x - (1 << i)$, consider doing $x\&\tilde{}(1 << i)$ instead. ((x) and not (2^i))

3 Ending Thoughts

Judging by the spread of solves in the contest, I think people need to consider trying every problem at least once. Especially in a 3 day contest with an unlimited amount of incorrect submissions, it can't hurt. Especially with problems like BattleCode, there are many solutions that just required a few minutes of thought, compared to more solved questions like Favorite Words that were harder (in my opinion). Make sure to try every problem in a USACO contest, it would suck if you realize that the solution to a problem was easy but the reason you didn't promote is because you didn't attempt a problem.

On another note, although these questions weren't professionally written, I would advise not asking questions about the statements, such as bounds on constants that don't affect complexity, ranges of string values, etc. that don't really have an effect on the problem. Try to solve a problem in the most general case, and please read the question before asking questions! In USACO and other contests, they aren't as generous about answering questions.

If you ask any of the ICT officers for advice on how to get better at USACO, they should all say the same thing: practice. Good practice involves just trying the problem, instead of skimming problem then solution and repeat.