Problem A. ViString

First, declare a variable $cnt = 0$. Start processing the string by checking every element from the beginning to the end. If the current element is *'v'* and $cnt = 0$, then increment cnt by 1. If the current element is *'i'* and $cnt = 1$, then increment cnt by 1. If the current element is *'s'* and $cnt = 2$, then return "YES". After you are done looping through the array return "NO". Time complexity is $\mathbb{O}(n)$.

Problem B. heXORei

XOR multiplication is associative and commutative. So, we don't care about the order of the numbers in the permutation. For each n number x, we have to check the xor of all numbers in the sequence without x and return the minimum xor among n possibilities.

The intended run time is $\mathbb{O}(n)$. So we must find the xor of n-1 numbers efficiently: Firstly, compute the xor of n numbers. Let's say it is equal to A. For each number x, xor of the rest of the n-1 numbers will be equal to $(A \oplus x)$ because the inverse of xor operation is xor operation. Assume arr_i is the $i-th$ element of the array. So, answer $= min_{i\in[n]}(A \oplus arr_i)$.

Note that $p_1 \oplus p_2 \oplus ... \oplus p_{n-1} \oplus p_n \oplus p_n = p_1 \oplus p_2 \oplus ... \oplus p_{n-1}$

Time complexity is $\mathbb{O}(n)$.

Problem C. Sugar Cane Agriculture

This problem has a very short solution, but it is not quite obvious to find.

Let us first consider all fields where both w and h are divisible by three. Then, by splitting the field into 3x3 squares, we have a simple optimal solution; if we place a sprinkler in the middle of every 3x3 square, we can plant 8 seeds around each sprinkler. The number of sprinklers is $\frac{w}{3} \cdot \frac{h}{3} = \frac{wh}{9}$, and the number of seeds $wh - \frac{wh}{9}$.

Now let's remove one row from the field so that h is still divisible by three, but $w \equiv_3 2$. In particular, let's look at the first 3 columns. We notice that it's easy to place the first $\frac{w-2}{3}$ sprinklers using the same 3x3 square idea above; however, we need an entire sprinkler to water the remaining 2x3 rectangle. The number of water sprinklers is the same as if we enlargened the field by one row; from w times h to $w+1$ times h. Hence, the number of sprinklers is $\frac{(w+1)h}{9}$, and the number of seeds $wh - \frac{(w+1)h}{9}$.

This can be extended to any grid size; we enlargen it until both w and h are divisible by three, calculate the number of sprinklers on the new grid, and then use that to calculate the number of seeds. The enlargened grid size is $3\lceil\frac{w}{3}\rceil$ times $3\lceil\frac{h}{3}\rceil$; hence the number of sprinklers is $\frac{3\lceil\frac{w}{3}\rceil \cdot 3\lceil\frac{h}{3}\rceil}{9} = \lceil\frac{w}{3}\rceil \cdot \lceil\frac{h}{3}\rceil$, and the number of seeds $wh - \lceil\frac{w}{3}\rceil \cdot \lceil\frac{h}{3}\rceil$.

Problem D. Fishing

Let l_i be the left most position we can throw the net so that we catch fish a_i. Then a_i would be the right most fish caught. In the optimal solution there will be some fish which is the right most caught fish, and we try all. We can binary search the position l_i, and then binary search the indecies of the left most and right most fish which are caught if we throw the net at position l_i. This give us an $O(n \log n)$ solution.

There is also a solution that works in linear time which uses the method sliding window.

Problem E. Cities

To check whether Alice will visit a particular city v, we can remove v from the graph, and check whether there is a path from a to b. This is a connectivity check which can be done in $O(n+m)$ using e.g. depth-first search (DFS). In the implementation, we don't actually remove v, but start a DFS from a and check whether it reaches b while ignoring all edges that lead to vertex v. We can do this check for all cities except a and b and count how many of them disconnect a and b. This solution runs in $O(n \cdot (n+m))$.

There is a $O(n+m)$ solution that uses a modified version of the algorithm for finding articulation points. Note that all cities which Alice is guaranteed to visit are articulation points, but not all articulation points are guaranteed to be visited. In particular, we are interested in those articulation points that lie on the direct path between a and b. Formally, in our DFS starting from a we are guaranteed to visit a city v if: (1) v is an articulation (2) b is in the subtree of v. The first condition is part of the articulation point algorithm, the second one is true if $v = b$ or if it is true for any of the children.

Problem F. How to Become a Millionaire

There is a dynamic programming solution to this problem. The question to ask to fill $dp[i][j]$ is: If you play optimally, assuming you start with $10,000 \cdot j$ how many ways are there to become a millionaire in i rounds?

For $i = 0$, the DP table is zero except for $j = 100$, where it is 1.

For any other i, iterate through all possible bets; for each possible outcome and the new amount of money c, sum $dp[i-1][\frac{c}{10,000}]$. The new DP value is then the max of these sums across all possible bets.

The solution can then be found in $dp[n][100]$.

Problem G. Buildings

The first thing to notice is that any trip that consists of more than two vertices, can also be viewed as a trip of two vertices where we start at the first vertex and end at the second vertex. Therefore we consider all trips of length 2.

Consider what is the optimal trip u, v such that the path between them contains w. If we pick some w as consideration and find the optimal answer. Then we now don't need to consider any pairs that go through w, thus we can remove w from the graph and solve for the rest of the graph. As we want a fast solution, we would want the subgraphs created to not have a large size. Thus we pick w to be the centroid of the tree. A centroid of a tree is a vertex, such that if we remove it no subgraph has a size more than $n/2$, where n is the number of vertices. Centroid decomposition or the process of repeatedly removing the centroid takes $O(n \log n)$.

Now we need to consider how can we find the optimal path that goes through w.

If you have a building with height h_u and a building with height h_v and the distance between them is d. Then you can fly from u to v if and only if $h_u - h_v >= d$, considering ordered pairs. We can rewrite $d = d_u + d_v$ where d_u is the distance from u to w. We thus get $h_u - h_v \geq d_u + d_v$. We can rewrite this as $h_u - d_u \geq h_v + d_v$, and thus each term is only relant to itself.

So we create an array of A of $h_u - d_u$ and B of $h_u + d_v$, which are both sorted. Now we need for each index in A to find the value in B which is greater than to equal to that value of A and contains the minimum h_u. This can be done in multiple ways, for example with a sliding window. Sorting the array will take $O(n \log n)$, thus the total time complexity is $O(n \log n^2)$.