1 Problem A. Range Sum Query Strikes Back

You can do pretty much the same as in the problem about minimum. The only difference is that you don't need to initialize b_i with INF but with 0.

2 Problem B. Lesser Elements with Insert

Discussed in the lecture notes, the code was shared

3 Problem C. Sum Problem

The solution is very similar to the problem B. You can also calculate prefix sums for the array you store. Another implementation fact is to check that you don't count any element twice. You can delete the duplicates from the array a but when you're dealing with elements from *delayed* you should also check that they are not in a. You can use unordered_set, binary search or any other option to solve this problem.

Last but not the least you should also keep the variable $lstAns$ which is equal to 0 if the last query was of type + or to the answer if the query was of type ?. You need it to use in the formula described in the "Input format" section.

4 Problem D. Fabrosaurs

If the updates were in a single point but not on the range we could store some kind of map/set for each block in SQRT decomposition and then use it to check if there is an element X in this block.

Now, let's see how we can modify the solution to support the range updates. The main idea is to keep so called promises. What is the promise? Well, if we want to update some range and some block is completely inside the updated range we don't want to increase each element in the block by X. We can create a promise $pr_i = X$ which says that we want to increase **each** element in the block by X. So, now while updating we can update the promises for the blocks and simply update the elements in the beginning and in the end.

Now, when we want to answer the query and we want to check if there is an element X in the block i we should check if there is an element $X - pr_i$ in the map/set we keep for this block.

Be very careful with the data structure you use as the Time Limit is quite tough. I recommend unordered_set or fast hash table (you can read here: https://codeforces.com/blog/entry/60737).

5 Problem E. Madness and Courage

It was quite a hard problem and, to be honest, I underestimated its complexity. Here's the main idea of the solution.

Let's define $health_{i,x}$ — what is the minimum initial value of health the warrior with attack x should have in order to be able to beat first i monsters.

We will firstly calculate all $health_{1,x}$, then all $health_{2,x}$, then all $health_{3,x}$, etc. until $health_{n,x}$. We can notice that $health_{i,x} - health_{i-1,x} = c_i \cdot \lceil \frac{d_i}{x} \rceil$. One can notice that there are no more than $2\sqrt{d_i}$ different values of $\lceil \frac{d_i}{x} \rceil$. So, if we maintain $health$ as a set of segments with equal value, the number of segments wouldn't increase by more than $O(\sqrt{C})$ after adding one monster (we can see that C is actually $200,000$ as it is the upper bound on d_i).

Now let's try doing the same idea we used in problem F (problems are not sorted by difficulty :-)). We will split the monsters into blocks, process the blocks one by one and for each hero

we will firstly find the block where it fails and then go through this block again separately for each such hero. For each block we will add $O(\sqrt{C}\sqrt{n})$ segments.

The only question that is left is how to keep the segments for the *health* array. It is fine to keep them in the vector in a proper order (as segments don't intersect) and carefully rebuilding this vector from scratch after we added a group of monsters (by finding out the segments where $\lceil \frac{d_i}{x} \rceil$ has equal values for each i in this block).

The complexity should be something like $O(n\sqrt{N}logC)$ as we need for each hero to find its segment after each block.

6 Problem F. Candies Eating

This problem is also not an easy one. Let's firstly solve an easier problem.

Imagine, we have the same queries but we want to determine how much candies are left in each pile in the end. To do it we can create a special array *add*. Now for each query (l, r, x) we will increase add_l by x and decrease add_{r+1} by x. After doing it for each query and calculate prefix sums on this array we can see that $pref_i$ is exactly how many candies were eaten.

Now back to the initial problem. Let's split the queries into blocks of size $\sqrt{Q}$. Now let's process all the queries from the first block as described above. This way we understand which piles were eaten during the first block. Now for each of these piles **separately** we will go through the queries of the block and apply it one by one to find the exact query when this pile is eaten.

For all other piles we just apply this block and move to the second block and do the same. Then to the third, fourth, etc.

Let's calculate the complexity. Firstly, processing each block is $O(N)$ so the total complexity is $O(N\sqrt{N})$. Also, for each pile we go through one block to find the exact query when it was eaten so it is another $O(N\sqrt{N})$. Thus, the total complexity is also $O(N\sqrt{N})$.

7 Problem G. Little Elephant and Array

This problem was described in the lecture notes. The only additional thing is to notice that we should only check the x's which are not greater than n. So, we don't need a map, we're fine with a simple array.

8 Problem H. Powerful array

You can solve it using Mo in a pretty much the same way. The formulas are slightly different though but not hard anyway.