1. Triplet

What are the possible solutions to the problem?

In my opinion, there are several solutions with time complexity from O(n^2) to O(logn). For example, treating each element as a middle one and checking its left and right values to find triplets, merge sort, and tree based algorithms.

How do you solve this problem?

The first thing that came to my mind is that I can take every element as a middle in the possible triplet, and by linear search, I found all the smaller left and bigger right elements, finding all combinations with them, and added to the triplet's counter. But it was too slow with the time complexity of O(n^2), so I decided to write in Python for my convenience. I tried solving with the "bisect" library, but still slow. Seeing the constraint of n ≤ 10^6 and learning algorithms in the class, I thought about the Merge sort. Also, that we need both the value and its position, made me think that it is a good method. So, I created a 2D array with numbers and their positions, and started two functions for the left and right sides, like the default Merge sort procedure, which we learn in the lecture. After dividing them, the left_merge function merges the sorted left and right halves of the array, and also counts for each number from the right half that how many smaller numbers have already been processed from the left. This count is then added to the sums array at the number's original index, which is the smaller of the left side. The right_ merge goes through the sorted left half and counts how many numbers in the entire right half are larger than it. Then adds it to sums and does merging. So we get lefts and rights, and we find how many combinations we can do with them, and that is the triplet count. Eventually, I got O(nlogn) solution, but it didn't pass the last test, so I tried to implement library sys, but I couldn't, even though it is standard library. I tried tokens for I/O and a buffer array for fast results, but didn't work. So I switched to Java again after seeing hint from Announcement and also added that buffer method that would avoid creating new merged [] every time in functions. While transferring code from Python to Java, I also noticed that I can't work with same number_position array because leftMergeSort would make different order from original so I made new copy to avoid confusion. Also I thought that Scanner or Buffered would not be enough, so after studying different kind of fast I/O in Java, I implemented FastReader, which instead of reading every element, it reads whole line and divides it.

Why is your solution better than others?

Maybe because I used already learnt the algorithm, as it was said in the instructions, avoided too much data by writing single array for the code without making new one each time, and passed all cases.

2. Yumao’s Multi-Year Rock Jumping Competition

What are the possible solutions for the problem?

I asked two questions to myself. First one is “Is it possible to achieve a bottleneck of at least x by removing at most M rocks?” and the only answer I could’ve thought of is just what was suggested in the instructions: Binary Search The second one is “Is value x feasible?” and I thought of Greedy Algorithms, especially Dynamic Programming, which I was discussing recently with my roommate, but it was a bit slow with O(n^2). So I just used the greedy check.

How do you solve this problem?

Firstly, I understood the template, like where I can text and the meaning of already written code. Then, in the hint, where said “Define a yes/no predicate P(x): “Is value x feasible?” Make sure P is monotonic in x.” I thought that Y would be x and P(x) function, which finds the maximum of the minimums of year x. So, I started writing P(x) first and then just found the best minimum through the loop. Starting with my logic of binary search, we make our first guess as a mid value, which is half of the whole length. Then in a loop, we check the difference between our standing point and where we gonna jump, and if it's less than mid, then it is too short for us and we skip it and add 1 to the number of removed stones. If it’s more than mid, then it's what we were seeking, and we save the index, and we also break if removed is more than M to save time. And outside of “for” loop, we check again for its feasibility, and if it’s feasible, we take mid as the maximum of minimums. After checking all the needed values in D[], we find the best value. That is our P(x). Then, to find the lowest throughout the years, I just compared them with each other and picked the best. Then I scored 80/100, and I needed to optimize it. After asking Mr. Yumao, I understood that I can skip years that are not feasible. For that, I found year zero’s value and put it as the best bottleneck, and in the second loop, we start with year 1. And before Binary Search itself, we check it for quick feasibility compared to year 0. We just go through the D[] and count jumps less than the best bottleneck, and if it’s more than M, then it is just feasible, and we just skip that year. Also, I added FastReader which I used in the first problem.

Why is your solution better than others?

It’s only 124 lines, and I also implemented the fastest algorithm that I know for this question. Skipping years and some arrays really optimised this program, even though the worst case is still O(Y*N*logL). But it went through all the cases and passed them.