Problem A. Picturesque Skyline

Problem Setter: Nafis Sadique
Tester: Md. Mahbubul Hasan
Category: Math, FFT, Divide and Conquer

Let's assume the building heights are a permutation of $1 \cdots N$, we can do that since they are all unique. We can easily see how to form a recursive formula to calculate the number of ways to arrange the buildings. So first of all, if we had to make only one group how would we calculate that? Let's call the function $G(i)$ as the number of ways we can arrange $2i + 1$ buildings in one group. That is simply choosing i buildings out of the $2i$ buildings (since the tallest building must sit in the middle).

$$G(i) = \binom{2i}{i} = \frac{2i!}{i! \cdot i!} \tag{1}$$

Now we are allowed to make multiple groups, let's forget the maximum K buildings restriction for now. So the number of ways we can arrange i buildings is defined by the function $F(i)$.

$$F(i) = \sum_{j=1}^{\lfloor \frac{i-1}{2} \rfloor} \binom{i}{2j+1} \cdot G(2j+1) \cdot F(i-2j-1) \tag{2}$$

$$F(i) = \sum_{j=1}^{\lfloor \frac{i-1}{2} \rfloor} \frac{i!}{(i-2j-1)!(2j+1)!} \cdot \frac{2j!}{j! \cdot j!} \cdot F(i-2j-1) \tag{3}$$

$$F(i) = i! \cdot \sum_{j=1}^{\lfloor \frac{i-1}{2} \rfloor} \frac{1}{j! \cdot j! \cdot (2j+1)} \cdot \frac{F(i-2j-1)}{(i-2j-1)!} \tag{4}$$

$$\frac{F(i)}{i!} = \sum_{j=1}^{\lfloor \frac{i-1}{2} \rfloor} \frac{1}{j! \cdot j! \cdot (2j+1)} \cdot \frac{F(i-2j-1)}{(i-2j-1)!} \tag{5}$$

This format allows us to define two new functions $F'(i)$ and $G'(i)$.

$$G'(i) = \frac{1}{i! \cdot i! \cdot (2i+1)} \tag{6}$$

$$F'(i) = \frac{F(i)}{i!} \tag{7}$$

Or we can write $F'(i)$ differently as:

$$F'(i) = \sum_{j=1}^{\lfloor \frac{i-1}{2} \rfloor} G'(2j+1) \cdot F'(i-2j-1) \tag{8}$$

At the end, the number of ways to arrange N buildings in pyramid-like groups would be $F'(n) \cdot n!$. We can easily calculate $G'(i)$ for all i in $1 \ldots N$. Since we need to print the result modulo 998244353, a prime number, we can get the modular multiplicative inverse of the $i! \cdot i! \cdot (2i + 1)$. We can bring back the limit on the group size K now and reformulate $G'(i)$. We can also simplify it a little bit to make the understanding easier, which would be represented as $G''(i)$.

$$G''(i) = \begin{cases} \dfrac{1}{\frac{i-1}{2}! \cdot \frac{i-1}{2}! \cdot i}, & i \leq K, i \equiv 1 \mod 2 \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

That gives us a new simplified version of $F'(i)$ which we represent as $F''(i)$

$$F''(i) = \sum_{j=1}^{i} G''(j) \cdot F''(i-j) \tag{10}$$

This would have been solved with an standard FFT if we knew the values of $F''(i)$ in advance. Since the value of i is dependent on values smaller than i, there is a well known divide-and-conquer technique called online-fft[1] that can easily handle this problem. We start with N buildings and then we split it into two halves. We can calculate both halves separately. Finally we merge the result of the first half with the second half by calculating the contribution of the first half for each element. We can do that by convolution (FFT). At the end we will have $F''(n)$ from which we get the result as $F''(n) \cdot n!$. Expected complexity is $O(n \lg^2 n)$.

Problem B. Easy String

Problem Setter: Hasinur Rahman
Tester: Raihat Zaman Neloy, Nafis Sadique
Category: Trie, Data Structure

This problem can be solved with Trie and Dynamic Segment Tree. Implementing straightforward Trie will not help actually. We will modify Trie for this problem. In basic Trie we just store the next node of a character. Here, we have to store the next node of a character of specific length (by length we mean continuous occurrence of a character in a string). As we are splitting the nodes according to lengths, we need a data structure which can provide a sum of frequency of strings passing through a range of lengths for each node of Trie. Segment Tree might help here. But we can not build a segment tree for each node of Trie. Here, Dynamic Segment Tree will help. You may take this discussion as a Hint only.

Things to be careful about:

- The query string S1 is not necessarily lexicographically smaller than S2. In that case swapping them would help.
- Memory usage can blow out of limit quite easily. We have to remember that the total number of characters can't be more than $5 \cdot 10^5$.
- Instead of a dynamic segment tree, Binary Indexed Tree using maps can be used. But trying to use maps of size over 10^7 will result in compilation errors, memory limit exceeded error or even runtime errors.

Problem C. Make A Beautiful Array

Problem Setter: Muhiminul Islam Osim
Tester: Raihat Zaman Neloy, Chayan Kumar Ray
Category: Greedy, Longest Increasing Subsequence

The first and most important observation is that It's always better to have array B with a single peak value. Now, for an index i of array A considered as the peak value, we need to calculate the longest increasing subsequence from the subarray $A_1 A_2 \dots A_{i-1} A_i$ and the longest decreasing subsequence from the subarray $A_i A_{i+1} \dots A_{n-1} A_n$. So, we precalculate two arrays called left and right, where $left_i$ is the longest increasing subsequence from A_1 to A_i and $right_i$ is the longest decreasing subsequence from A_i to A_n. Then, for each index i of array A, we can calculate the maximum length of array B if we consider A_i as the peak value. We take the maximum over all indices, and that's the maximum beauty factor we can achieve.

[1] https://codeforces.com/blog/entry/111399

Problem D. Battle of Boberland

Problem Setter: Chayan Kumar Ray
Tester: Nafis Sadique, Mahdi Hasnat Siyam
Category: Data Structure

The problem can be solved by different DS (Trie, Segment Tree, Sparse Table, Prefix Sum etc) using the observation that only the most significant bit (MSB) will always contribute to the answer. N.B: For querying all the elements zero, 0 will always be the answer.

Proof Let's assume, we have all the elements with the same MSB. 2^{msb} will always be the minimum value with which the XOR ($\oplus$) of elements will result in a value less than equals to 2^{msb} (since the binary representation of $2^{msb} \oplus a_i$ will have 0 in the position of MSB) no matter what's other bits are ON in the binary representation of a_i. Adding them to the answer along with MSB will maximize the answer instead of minimizing. Also without considering MSB, it will never satisfy the condition described in theproblem.

Now, let there are elements with different MSBs (msb_0, msb_1, msb_2, msb_3) and their order is $msb_0 < msb_1 < msb_2 < msb_3$. Taking msb_3 will ensure the XOR value (with all the elements containing msb_3) is less than 2^{msb_3}. But it will not satisfy the true condition for other elements with different MSB (XOR of different MSB results greater value for the elements containing other MSB). To satisfy the described condition for these values we have to also consider other MSBs contributing to the answer. We can also consider bits other than MSBs contributing to the answer which will satisfy the less than equal condition but will fail to satisfy the condition of minimizing the answer.

Problem E. Platonic number in a tree

Problem Setter: Mahdi Hasnat Siyam
Tester: Nafis Sadique, Hasinur Rahman
Category: Tree, Sparse Table

For each node u, let's find the first node v in the path from u to root such that $a[v] > a[u]$.

Platonic number of node u is the $a[u] * (depth[u] - depth[v])$ + platonic number of nodes v. In case there doesn't have any node v, platonic number is $a[u] * depth[u]$. Say we maintain two arrays $parent[N][K]$ and $maxA[N][K]$ where $K = \log(n)$. Here $parent[u][i]$ is the 2^i th parent of node u and $maxA[u][i]$ is the maximum value of a of the first 2^i nodes from u to root.

We visit the nodes in dfs order starting from node 1. Say we are in node u and the parent of this node is p. So $parent[u][0] = p$ and $parent[u][i] = parent[parent[u][i-1]][i-1]$. In the same way $maxA[u][0] = a[u]$ and $maxA[u][i] = \max(maxA[u][i-1], maxA[parent[u][i-1]][i-1])$. Now initialize v as u. Loop i over $K-1$ to 0 and jump v to the $parent[v][i]$ if $maxA[v][i] < a[u]$. Finally use the formula above to find the platonic number of node u.

Problem F. Aragorn is back

Problem Setter: Anik Sarker
Tester: Muhiminul Islam Osim, Raihat Zaman Neloy
Category: Dynamic Programming, Data Structure

Idea 1 One observation here is: the squads we will form will never overlap with one another. After this the solution become like the followings:

- Find subarray max sum and keep track of their length.
- Loop over the given array, if a subarray matches the constraints described in the statement, keep them in a vector, sort them and keep a counter of the vector in a map.
- Iterate over the map and find the maximum counter.

Complexity: $O(n \cdot \lg(n))$.

Idea 2 If you don't get the observation from the above array, then you have the hashing algorithm to solve this. In that case, the solution will look like the following:

- Find subarray max sum and find the minimum length of the max sum subarrays. Let's say this length is X.
- Iterate over the array, and from each position, find the hash of the subarray with length X, and keep a counter of that hash values in a map or any sort of stl / data structure.
- Iterate over the counter DS and find the maximum value.

Complexity: $O(n \cdot \lg(n))$.

Problem G. Game of Removing

Problem Setter: S.M Shaheen Sha
Tester: Md. Ashraful Islam Shovon, Raihat Zaman Neloy, Nafis Sadique
Category: Brute Force, Two Pointer, Binary Search

Idea 1 Assume that we want to remove the subarray $B_l \dots B_r$. So, let's fix l first. Then we can try to find the smaller index p_l in array A so that $B_1 \dots B_{l-1}$ is a subsequence of $A_1 \dots A_{p_l}$. Similarly we can fix r as well, we need to find the largest index p_r in the array A so that $A_{p_r} \dots A_n$ contains $B_{r+1} \dots B_n$ as a subsequence. If no such p_l or p_r exists then we consider them as $+\infty$ and $-\infty$. After that we can iterate over all pairs of l & r such that $p_l < p_r$, the length of the smallest such pair is the result.

To calculate these values we can use two pointer technique. Or just simply store them in memory. We can use the observation $p_{l-1} > p_l$ & $p_r < p_{r+1}$, meaning we can iterate over the array from both direction and stop once we found a match. This approach requires $O(n^2)$ complexity, finding p_l or p_r take $O(n)$ complexity.

Idea 2 We can do binary search on the length of the subarray we want to remove. In the binary search, after selecting a length X, we will loop over array B, try to remove all the subarrays of B of length X and want to know if we can get a valid subsequence of A. If yes, then we will try to reduce the subarray length, or will increase our selected length for next testing. Complexity: $O(n \cdot \lg(n))$.

Problem H. Calendar (Please No More)

Problem Setter: Md. Ashraful Islam
Tester: S.M Shaheen Sha, Raihat Zaman Neloy
Category: Implementation

This problem doesn't need much discussion, it was a straight-forward implementation problem.