BUET Individual Long 3 : (DS)

<https://vjudge.net/contest/363604>  
([Pass](https://www.youtube.com/watch?v=eTa1jHk1Lxc): Jenny+of+Oldstones)

**Problem A**

* Rafid
* Segment Tree / BIT

Let’s match each closing bracket with the closest opening bracket left to it that is still unmatched. For each of these matches, we shall get a segment which if covered fully by a query range should add 2 to our result. For every index, we shall have two lists; one being the bracket segment ending here and another one the queries that ends here. For each bracket segment, add 2 in the starting index of this segment. And for queries, just do a simple range sum query.

Complexity: O((n + q) lg n)

Code: <https://vjudge.net/solution/24882336>

**Problem B**

Editorialist: Solaiman

DS: Dynamic Convex Hull Set

First of all notice the behavior of the given function. f(i, j) means taking i continuous elements (an element can be taken more than once) from index j to it’s left.

So, optimal solutions of f(3, 4) could be one of a[3]+a[4] and a[4]\*2.

Optimal solutions of f(2, 4) could be one of a[2]+a[3]+a[4], a[3]\*2+a[4] and a[4]\*3. Why are we not considering a[3]+a[4]\*2 as an optimal candidate?

Well, it’s not optimal to take more than one element from other than the leftmost index. We can prove it using the exchange argument. Suppose in an optimal solution we are taking at least one element from range [k, j]. Now let p be the leftmost index where k < p <= j and and we are taking **more than one** element of a[p]. Since, this is an optimal solution, we can safely say that a[k] <= a[p] (otherwise we could have ‘released’ the elements of index k and take those from index p instead!). Now as a[k] <= a[p], we could have taken exactly one element from p and the rest of the remaining elements from k and our solution wil still remain optimal.

So, optimal solutions of f(i, j) will be in the form: ; for some k >= j-i+1. We can write it as,

,

, where pre[j] is the prefix sum of a[] till index j.

For an index k, Let’s denote, and. We can treat these as equations of lines. So, for a given i and j, the answer will be where k >= j-i+1. Each query can be answered in log2N time by keeping a binary indexed tree of dynamic convex hull sets :)

***Dynamic Convex Hull Set***: a DS which stores the lower/upper hull of some lines. You can add lines (m, c) of form y=mx+c to this DS. In log(N) time, you can find the minimum value of all lines for some given x.

Code: <https://gist.github.com/solaimanope/0fb81e12755e14de2db0f149199cfde8>

**Problem C**

Editorialist: Solaiman

DS: Link Cut Tree

*Notice: this problem has a solution using MO too. I will describe only the solution involving LCT here. The complexity of such a solution will be MlogN.*

We add edges to the graph from left to right. While doing this, we will maintain a forest. If i-th edge (u, v) creates a cycle, find the edge with the leftmost index j (obviously, j < i) which is in the path between u and v in the current forest. Remove edge j from the forest. Then, add the edge i to the forest. Now, we will answer all queries for which R = i. Now, for a query (L, R), the answer would be the number of trees in the current forest **plus** the number of edges with index less than L in the current forest. The last part can be done just by keeping a BIT on the index of existing edges in the forest.

Code: <https://gist.github.com/solaimanope/6f35ed598f975ade78a9e3ba39b8bffd>

**Problem D**

* Rafid
* DS: Small-to-Large/DSU on Tree

This problem can be solved offline, since there are no updates. So we can store the queries in respective vertices. We will traverse the tree in post-DFS order so that for a vertex v, we have answered queries for all vertices in its subtree and have the required values in its subtree and can answer queries regarding this vertex.

We can have two arrays- freq (denoting frequency of a color), cnt (frequency of a frequency). We can update these two by small to large merging or the so-called DSU on Tree which can be learnt from here: <https://codeforces.com/blog/entry/44351>. Our answer for any query v, k would be the summation of cnt[j] where j >= k. To achieve this, let cnt be a Binary Indexed Tree or a Segment Tree instead of an array.

Complexity: O(n lg^2 n)

Code: <https://vjudge.net/solution/24906912>

**Problem E**

Editorialist: Solaiman

DS: Binary Indexed Tree, Set

Notice that indices which have value 1 or 2 won’t change in the future. Let’s keep a set of indices where value is greater than 2. When REPLACE l r update is given, we will iterate over the indices in the set which are in the given range and update their values. When a value becomes <= 2, remove this index from the set. Also, we will maintain a BIT to efficiently answer the queries. The editorial on CF says “At first let's notice that this function converges very quickly, for values up to 106 it's at most 6 steps.” So, we won’t traverse each element in the set more than 6 times.

Code: <https://vjudge.net/solution/25034376>

**Problem F**

Editorialist: Solaiman

DS: Merge Sort Tree

We will do a binary search on the answer. After we fix a mid, we will count the number of elements less than or equal to mid in range [i, j]. This can be easily done with Merge Sort Tree! Learn about Merge Sort Tree [here](https://www.commonlounge.com/discussion/fe6ac441785c44d6a959eab662f15adc).

**Problem G**

* Rafid
* DS: Persistent Segment Tree

Help yourself learn about PST. Then read this: <https://blog.anudeep2011.com/persistent-segment-trees-explained-with-spoj-problems/#more-108>

There is an overkill, if you want to do it. Use HLD to get the path segments, then use Persistent Segment Tree to get the kth element in those segments combined. Complexity: O(n lg^2 n). Code: <https://vjudge.net/solution/24914230>

**Problem G (without HLD)**

* Solaiman
* DS: Persistent Segment Tree

Okay, there’s a solution involving persistence but no HLD. Complexity: O(NlogN).

For each node u, suppose we have a (persistent) segment tree built on the frequency array of the weights from root to u. In a segment tree built on some frequency array, you can find the k-th sorted element easily in logN time. Now for each query (u, v), find lca l and it’s parent p. Now, when we are doing binary search inside segment tree, we will do it simultaneously for nodes u, v, l and p :3 When doing frequency count inside a segment tree node, do it like this: f(u)+f(v)-f(l)-f(p).

Code: <https://vjudge.net/solution/25051328>

Problem g alternate 2 : MO diye chap.

**Problem G (with just BIT)**

* Ashiq
* BIT, Binary Search, Offline trick

First, let’s solve another problem. Suppose, you are given a rooted tree and each node has some value and lots of queries of the form : **U X**

*You have to find how many values in the path from root to node U are smaller than X.*

We can solve this problem in **O((q+n)logn)** offline. Let’s start a dfs from the root. When we are at node **U,** we want to maintain all the values from root to node **U** in some data structure. To do this, when you first visit a node, just insert the value in the DS, and when you are leaving, erase that value. If you do this, when we enter the node, only the values from root to **U** will be in the structure. Then, you can calculate the result for all queries corresponding to node **U.** For this particular problem, we can simply use a **BIT / ordered set** as the DS.

**Code Snippet:**

Void dfs(int u, int p) {

// insert value of node u

// solve queries of node u

for(int v : G[u]) {

if(v == p) continue;

dfs(v, u);

}

// erase value of node u

}

We can even, solve these queries as well : **U V X**

*Find how many values are smaller than* ***X*** *in the path from* ***U*** *to* ***V***. This is basically the same as above. You have to just split the path into **3** paths :

1. Root to LCA
2. Root to U
3. Root to V

So, how do we find the **kth** minimum of a path ? Let’s binary search on the **k’th** minimum. To check whether **k’th** minimum is smaller than a particular value **X**, we just need to find how many values on the path are smaller than **X**. But we are not going to binary search each query independently here.

We will binary search all the queries at the same time. For i’th query we shall maintain, lo[i] and hi[i]. Then find mid[i], and we will ask q queries of the form: **U[i] V[i] mid[i] ,** *how many values are smaller than mid[i] in the path from U[i] to V[i]*,in each iteration of binary search. We can find the result for all the queries in **O( (n+q)logn )** in each iteration.

Total Time Complexity: **O((n+q)\*logn\*log(n))**

**Code:** [**https://vjudge.net/solution/24968491**](https://vjudge.net/solution/24968491)

**Problem H**

* Rafid
* Treaps

This is a naive problem on Treaps. Learn about them from <https://cp-algorithms.com/data_structures/treap.html> or <https://www.youtube.com/watch?v=erKlLEXLKyY>. You should need Implicit Treaps.

ps. If you don’t want to use treaps (which I don’t recommend by the way), you could get the new positions for every query. Consider insertions to be already there with a null value, deletions to only make the value null and it could be solved by Segment Tree. However that would be some trouble, as I did here: <https://vjudge.net/solution/24920327>.

**Problem H (Sqrt Decomposition)**

* Ashiq
* Square Root Decomposition

I will just briefly discuss how you can handle erase, insert, replace queries. Let’s split the whole array into **√n** parts. Let’s call each of these parts a bucket. So, each bucket contains approximately **√n** elements.

**Insert:** Simply loop through the buckets, and you can find in which bucket and at what position you must insert the value in **O(√n).**

**Delete, Replace** can be done similarly.

But there is a catch. What if all the inserts are in the same bucket. Then the size of the bucket gets larger, and insertion will get costly. To prevent this, after every **√q** queries, you should rebuild the buckets in **O(n)**

**Total Complexity : O(n√q + q√n)**

**Problem I**

* Rafid
* Treaps

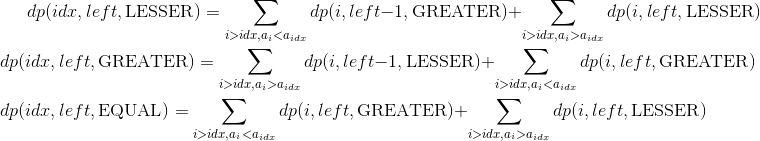
Another problem on treaps. The challenging part is the mathematical merging for a node value. Read about it briefly in Chinese from <https://blog.csdn.net/weixin_38167363/article/details/101894218> ;)

**Problem J**

* Rafid
* DP, BIT

This can be solved with Dynamic Programming. Let’s say dp(idx, left, pre) denotes the result for [idx, n) subarray with “left” peaks where pre denotes the state of the previous value taken before “idx”; the states are LESSER, EQUAL, GREATER. Also, we are saying that we will take the SRM at “idx” since we are starting from here.

Alright, definitions done. Let’s go to transitions.



However, there’s one little detail remaining to be handled. The problem asks for distinct graphs. So, if we consider indices for our dp states, that might lead to some overcounting. What we do is, in transitions we don’t iterate over the indices rather the values in the right side. And, we store dp(idx, left, pre) in a table with indices like (a[idx], left, pre). Basically, we calculate the dp value for “idx” but while storing, we overwrite the “idx” with its array value. And, now for the effective transitions, we can use a Binary Indexed Tree. The result would be the sum of dp(index of the first occurence of an array element, left = k, pre = EQUAL). Make sure, for each distinct element, you add its corresponding dp value once.

Complexity: O(TNK lg N)

Code: <https://vjudge.net/solution/24955569>

Good luck bro :p

:( Shouldn’t have taken this one :3

**Problem N**

Editorialist: Solaiman

DS: Segment Tree, Stack

Let’s process each of the bits separately and count the number of subarrays in which min&max has this bit as 1. Notice that, the **bits lower than k are irrelevant** in deciding max or min now.

For bit k, let’s process the array from left to right. Let’s keep two stacks: one for minimum and one for maximum. When we are at index j, maxStack will store the indices i, where getMax(i, j) gets changed (once again, we are considering k or higher bits to decide max). minStack will also store equivalent information for minimums.

We will maintain a segment tree built on two binary arrays A[] and B[]. A[i] is 1 if getMax(i, j) has k-th bit 1 otherwise 0. B[i] is 1 if getMin(i, j) has k-th bit 1 otherwise 0. In the segment tree, we should be able to set 1 or 0 in some range in array A or B. Also, it should be able to tell us the number of indices i where both A[i] and B[i] are 1.

Now, when we go from j-1 to j, we will know (by popping maxStack) the minimum i for which getMax(i, j) is decided by j. We will set A[i...j] as the k-th bit of p[j]. We will also do similar updates on B[] using minStack. Now the topmost node of the segment tree will tell us the number of indices i such that getMin(i,j)&getMax(i,j) has k-th bit one.

Code: <https://vjudge.net/solution/25009423>

**Problem O**

Editorialist: Ashiqul Islam (Bhai WF treat den :( → nope :) )

Tags: Square Root Decomposition, DSU

First let’s compress all the coordinates. Then, one possible solution is as follows:

Initially the grid is colored white. Iterate over the rectangles in reverse order. During i’th iteration, find if there is any white cell inside the rectangle. Then this color will be seen. Color all the white cells inside this rectangle (if any).

We will do basically the same thing. Let’s split the X-axis in sqrt(max X) parts. For each part, we will keep a boolean array **whole** of size max Y. **whole[y]** is equal to 1 if and only if the whole row at y height in this part is already colored. We will also keep for each part whole\_next array, that points to next row, that is not fully colored.

Now, each of the rectangles occur partially in at most two parts. We will store the y1, y2 for these partial parts as well and compress them in the end.

We will have a 2d array **partial[X range for this part][y occuring partially]** for each of these parts and partial\_next[][] same as before. Partial basically gives the whole picture. Partial[x][y] represents if it’s colored or not. So, we have all our tools.

So, each rectangle is at most two partial parts and some full parts. To update the full parts, just keep jumping from y1 to y2 and keep coloring them.

The partial block part is tricky. The partial grid will come handy. Update partial with brute force and updating next. But as soon as, one row is fully colored, update the whole array as well.   
  
Time Complexity: **O(n√n \* inverse Ackermann(n))**

Memory Complexity: **O(n√n)**

**Problem P**

Editorialist: Solaiman

DS: Segment Tree

If you apply modulo operation to some number x, x' = x mod p, there are two cases either x' = x (when x < p) or x' < p <= x (when x >= p). Now, think about the second case, which changes the current value of x. When the second case applies, the value of x gets **at least halved**. So, if you keep applying modulo operation to some number x, how many times it’s value will be changed? At most logx times. After that, x will become zero and no further change will be applied to it. We will use this idea to solve the problem. Write a segment which maintains sum and max. When we are doing an update modulo p, if we see that current node’s max < p we will proceed no further. In this way, we will visit only the **relevant indices** where the values are **actually being changed**. The complexity will be NlogNlog109.

Code: <https://gist.github.com/solaimanope/26822f9dfef1e32065cfc94d557f22a1>

**Problem Q**

Editorialist: irshad

DS : link cut tree

The task is to find the sum of depths of nodes of the tree. In other words, the sum of subtree sizes. Let’s process all the nodes from 1 to n in increasing order. To add node i as the root we will have to cut some edges in the current tree and add a few back. The total number of cuts and links for adding one node might be O(n). but the number of cuts and links throughout all the nodes is O(nlogn). We can maintain subtree sizes by a link cut tree and dynamically add and remove edges. Finding the edges which need to be cut can also be found by keeping the maximum and minimum index nodes of a path in the splay tree. Each LCT operation (access, cut, link) takes O(logn).

The overall complexity is O(n log^2 n)

**Problem R**

* Rafid
* Segment Tree: Lazy Propagation

This here is a cute problem. The challenging part is the update. Let’s fix a node structure for now. We shall keep these values in a node of segment tree:

* max - denoting the overall maximum in the corresponding range.
* cnt - denoting the frequency of maximum in the range.
* sec - denoting the second maximum in the range.
* sum - the summation of the values in the range.

Let’s consider an update with a value “t” in a node. We shall consider only three cases.

1. t >= max
2. sec < t < max
3. t <= sec

Notice that 1 can be ignored since it’s not changing anything. For 2, we can do a simple lazy update, since only the max and sum would be changed. The changes should look like this:

sum -= (max - t) \* cnt

max = t

We should look into 3 carefully. What if, for a node with condition 3, we traverse both of it’s childs recursively and carry this on until we reach a leaf or a node with condition 1 or 2? Would that be a lot? There’s a nice observation to be made that whenever we are dealing with this condition, we are actually erasing at least one element from our node. The max element! Our new max would be t nonetheless and also we will have a new second max. How does this help us?

Consider the values in the node to be a, b, c, d, e where a > b > c > d > e.

* If we update with t1 <= b, that would be a condition 3 and the new order would be t1 > c > d > e.
* Another update with t2 <= c would give t2 > d > e.
* Another with t3 <= d would provide t3 > e.
* Only one further t4 <= e can be made providing only t4.

So in a node with n distinct values, we can only make updates with condition 3 at most n - 1 times, since every time at least an element is erased.

Since every element is in at most lg n nodes and we can only erase them once, the overall complexity would be O((n + q) lg n).

Code: <https://vjudge.net/solution/25016158>

**Problem S**

* Rafid
* Segment Tree

Read the editorial from here: <https://codeforces.com/blog/entry/45310>. Nicely explained.

**Problem T**

* irshad
* Segment Tree

As the numbers in the array are distinct we can treat the array as a permutation. All numbers to the left of the maximum are connected to it which is a prefix of the array. The first prefix which is disconnected from the rest of the array is such a prefix where every number is strictly greater than every number in the rest of the array. Without this prefix, the rest of the array is still a permutation where the next connected component is also a prefix. Thus each component is a consecutive part of the array.

We do not need to use all possible edges. Only the edges connecting two consecutive elements of the permutation is enough because the union of these edges will cover the entire component segment. We can find the immediate smaller and larger numbers by a set and process each query by erasing the current value and inserting the new one. Erasing a number will delete at most 2 edges and insert at most 1 edge. Inserting a number will delete at most 2 edges and insert at most 2 edges.

We will keep only the edges where x+1 is to the right of x in the permutation.

We can build a segment tree on an array of size 2\*n-1 where there are n-1 free spaces in between nodes. Initialize nodes with 1 and free spaces with 0. Adding an edge u-v means adding 1 on a range [ pos[u] , pos[v] ] and deleting an edge means subtracting 1. All we need to do is find what is the minimum and its count. If the minimum is 0 then the number of components is cnt[0] + 1. Else there is only 1 component.

Overall complexity : O(nlogn)