PREREQUISITES:

Euler tour of a tree, segment trees with lazy propagation, binary lifting

PROBLEM:

Given a tree on N vertices each with a color assigned to it, support Q queries on it, each being one of the following:

- Given u, change the root to u
- Given u and c, set every vertex in the subtree of u to have color c
- Given u and c, count the number of vertices in the subtree of u that have color c

EXPLANATION:

First, let's ignore the root-change updates and try to just solve the other two queries for a fixed root.

Dealing with color changes and queries

For now let's assume the root of the tree is 1.

A useful technique when dealing with updates and queries on subtrees is the Euler tour of a tree.
This allows us to convert subtree updates/queries into subarray updates/queries instead, which are much easier to deal with.

So, build the Euler tour of the given tree. Let B denote the array of colors in Euler tour order. We need to support the following:

- Given c and a range $[L, R]$, set $B_i := c$ for every $L \le i \le R$
- Given c and a range $[L, R]$, count the number of $L \le i \le R$ such that $B_i = c$

There are a few different ways to handle this. One is to use a segment tree with lazy propagation.

Details

Build a segment tree on B.
Each node of the segment tree stores a frequency map of all the elements in the subarray it represents. Let $freq[node]$ denote this map for a given node.

With this, querying a range for the count of c is trivial: it's simply the sum of $freq[node][c]$ across all the $\mathrm{O}(\log N)$ segtree nodes that cover the range $[L, R]$, so each query can be answered in $\mathrm{O}(\log^2 N)$ or such.

Handling updates can be done with the help of lazy propagation.

Suppose we want to set the range $[L, R]$ to c.

Let $[l, r]$ be one range of the segment tree that fully lies within $[L, R]$, corresponding to the node $node$.
Then, do the following:

- For each $x \in freq[node]$ and each ancestor a of $node$, perform `freq[a][x] -= freq[node][x]`
- Set $freq[node]$ to contain the single element c with a frequency of $r - l + 1$, and also increase $freq[a][c]$ by $r - l + 1$ for every ancestor
- Set a flag at this node to lazily propagate this information to its children when it is needed.

While this might make a single update take a lot of time, the total number of operations is bounded by $\mathrm{O}((N + Q)\log^2 N)$ map operations across all updates.

Proof

Each element of some map affects only its $\mathrm{O}(\log N)$ ancestors, and whenever we happen to process it, it is completely deleted.

Building the segment tree itself creates $\mathrm{O}(N \log N)$ map elements, and then each update creates $\mathrm{O}(\log N)$ more.
So, we create $\mathrm{O}((N+Q)\log N)$ map elements in total, each of which costs $\mathrm{O}(\log N)$ map operations to delete.

Note that depending on your implementation, you might need to use a reasonably fast hashmap to not get TLE: for example, the one **described here**.

Handling root changes

Changing the root appears to change the structure of the Euler tour completely, and we can't afford to rebuild it after each change.

However, as it turns out, we don't need to rebuild the Euler tour!

Let's root the tree at 1 and compute its Euler tour, as above. Now, consider two vertices r and u.

- If r does not lie in the subtree of u, then when rooting the tree at r, the subtree of u is the exact same as the current subtree of u.
- If r does lie in the subtree of u, then when rooting the tree at r, the subtree of u is the entire tree except the subtree of v, where v is the child of u that contains r.
  - Finding v is a simple application of binary lifting.

In terms of the Euler tour we first computed,

- The first case just corresponds to a single subarray in it
- The second case corresponds to the entire array except some subarray, which can be thought of as two disjoint subarrays instead

Either way, even if the root is changed we only need to update/query at most two ranges each time, so our time complexity doesn't change!

TIME COMPLEXITY:

$\mathrm{O}((N+Q)\log^2 N)$ or $\mathrm{O}((N+Q)\log^3 N)$ per testcase.

PREREQUISITES:

Hash, Suffix Array, Segment Tree

PROBLEM:

Given two strings **s[0…L-1]** and **t[1…L-1]**, find out the maximum prefix such that they are cyclic equivalent.

EXPLANATION:

Let **F[i]** be the largest **k**, such that **s[i…i + k - 1]** equals to **t[0…k - 1]**. Similarly, **G[j]** denotes the largest **k**, such that **s[0…k-1]** equals to **t[i…i+k-1]**. Both of them could be done by binary search plus hash, or Suffix Array, in **O(LlogL)** time.

With these two auxiliary arrays, we can rewrite our goal in the following form:

For every **0 <= i < L**, we need to find out the maximum **j**, such that **0 <= j < F[i]** and **j + G[j] >= i**. For such pair <**i**, **j**>, the length of prefix is **i + j**, as illustrated in the following figure.

This query for each **i** could be solved by Segment Tree, while each node (**[l, r]**) records the maximum **j+G[j]** of **l <= j <= r**. Therefore, this problem could be solved in a a time of **O(LlogL)**.

PREREQUISITES:

Greedy, **Heavy-Light Decomposition** , Segment Tree with lazy Propagation.

PROBLEM:

Given N lights connected in form of a tree, we are to perform M updates involving flipping lights of two houses. After each query, we try pairing all lights. Print the minimum sum of the distance between each pair we can achieve by optimal pairing after each update.

QUICK EXPLANATION

- Each edge may be included at most once because if included more than once, we can always switch pairs to get this edge excluded by a factor of 2.
- This way, if we flip all edges from included to excluded and vice versa on the path of two houses, the answer is the number of included paths.
- This can be done using Heavy-Light decomposition combined with Segment tree with Lazy Propagation to perform range xor query.

EXPLANATION

Let us the same problem for a chain instead of a tree.

Consider example. Total 10 houses with house 3, 5, 9 and 10 having their lights on. The minimum sum of distances between pairs is 3. Suppose we get an update to flip lights of house 2 and 7. Now, the pairs shall be $(2, 3), (5, 7), (9, 10)$, as can be seen in the image.

We can see, that is is never optimal to keep an edge included more than once in any pairing. Also, notice that the range $[2, 7]$ just got flipped due to update.

Coming back to trees. Assume among all pairs, we have two pairs (a, b) and (c, d) such that paths from a to b and path from c to d pass through edge e with endpoints u and v. We can see, that since both paths cross this edge, one endpoint of each pair lie on one side while another endpoint on another side. So, it is better to pair nodes on each side to themselves. It can be seen, that the sum of the distance between pairs has reduced by two times the number of edges shared in the path between two pairs. Hence, optimal solution, trying to minimize the sum of distance, shall never have any such pair of pairs which have a shared path. This implies, whenever we have an update (a,b), we can just flip included to excluded and vice versa for all edges on the path from a to b. The answer is just Number of included edges.

An example can be found in secret box 😛

Click to view

Consider tree formed by edges 1-2 1-3 1-4 2-5 2-6. Suppose houses 3,4,5 and 6 have their lights on. Then, pairs $(3, 4)$ and $(5, 6)$ are optimal. Pairing $((3, 5), (4, 6))$ and Pairing $((3, 6), (4, 5))$ is not optimal and both include shared path from 1 to 2.

This was all for the idea, coming to **implementation**. Basic knowledge of Heavy Light Decomposition is a must.

See, we basically divide tree into at most $\lceil logN \rceil$ chains, such that we can perform range update over all vertices in time less than the number of vertices in a chain. This way, whenever we have an update, we split the update path into parts such that each sub-path belong to one chain. Each sub-path is just the same as solving the problem on a chain, as explained above. So, we are left with solving this problem for a chain. This is a well-known problem and can be found **here** .

Apart from that, some users face difficulty in handling edges in Heavy Light Decomposition, because an edge is represented by a combination of vertices. A very simple and neat way to tackle this would be as follows.

We have rooted the tree at any vertex, say r. Now, all vertices except r shall have exactly one edge which leads from that vertex to its parent. This is it. We represent edge from a vertex to its parent by that vertex. So, we can translate heavy light decomposition on edges to heavy light decomposition on vertices by modifying our implementation a little.

When updating path from a to b having LCA c, update all nodes on the path from a to c except c, and on the path from b to c except c. This way, we can now use heavy light decomposition on vertices to solve this problem. Details can be found in implementations below.

Learning Material

Heavy Light Decomposition **here** and **here** are good enough with nice practice problems.

For Segment tree with Lazy Propagation, **this** blog is nice. Practice problems can be found **here** .

The proof we used above, is actually n example of exchange argument used for proving/disproving optimality of any greedy solution. Read more about it **here** .

Time Complexity

Time complexity is $O(N * logN + M * log^2N)$, $O(N * logN)$ per chain and the total number of chains being at most $logN$. $O(N * logN)$ preprocessing for LCA finding using binary lifting or Euler tour.

PREREQUISITES

[Segment tree]

PROBLEM:

You are given a sequence of n numbers you are tasked to perform 2 types of query on this sequence.

Query **1 x y** is to find the number of terms in the sequence of the form **6N+1** and store this value as **L** and find the number of terms of the form **6N-1** and store this value as **R**. you have to
print **rangesum** (**min** (**L** %**n** ,**R** %**n**),**max** (**L** %**n** ,**R** %**n**))
Query **2 i val**- update the value of the ith element to val

EXPLANATION:

This problem can be easily solved using segment trees. Let's store the sequence into an array, say **Valarray**.
Let the number of the form **6N+1** be **A**
Let the number of the form **6N-1** be **B**
Now let's define two arays **Aarray** and **Barray** such that

- **Aarray[i]** is **1** if the number at the ith position is of the form **A** else **0**.
- **Barray[i]** is **1**if the number at the ith position is of the form **B** else **0**.

Now we construct three range sum segment trees with **Aarray**,**Barray**,**ValArray**

After this the queries can be easy solved

Query 1:

1 x,**y** can be solved by finding range sum on **Aarray** and **Barray** in the given range(x,y) and let the range sum we got from **Aarray** be **L** and **Barray** be **R**. now solution to this query is
sumrange(min(L%n,R%n),max(L%n,R%n)) on the given **Valarray**

Query 2:

2 i val this query is to update the value of ith index to **val**.for changing this we need to update all the tree segment tree segment trees.while updating the **Aarray** and **Barray** segment tree we need make sure if the **val** is of the form **A** and **B** and the value which is been replaced is of the form **A** and **B**

Update types in Aarray :

- Initial value - of the form **A**
  Replaced value - of the form **A**

- Initial value - not of the form **A**
  Replaced Value- not of the form **A**

```
no  need to update for these two cases
```

- Initial value - of the form **A**
  Replaced value -not of the form **A**

```
Update with difference  -1
```

- Initial value - not of the form **A**
  Replaced value -of the form **A**

```
Update with difference 1
```

Similarly **Barray** updation can be done

This problem can also be solved using fenwick tree instead of segment tree

TIME COMPLEXITY

Time complexity is O(nlog(n))

SOLUTIONS:

Setter's Solution

```
from math import ceil,log
def getmid(s,e):
    return s+(e-s)//2
def constsum(arr,n):
    h=ceil(log(n,2))
    max_size=2*(2**h)-1
    st=[0]*max_size
    constutilsum(arr,0,n-1,st,0)
    return st
def constutilsum(arr,ss,se,st,si):
    if(ss==se):
        st[si]=arr[ss]
        return arr[ss]
    mid=getmid(ss,se)
    st[si]=constutilsum(arr,ss,mid,st,si*2+1)+constutilsum(arr,mid+1,se,st,si*2+2)
    return st[si]
def getsum(st,n,qs,qe):
    return getsumutil(st,0,n-1,qs,qe,0)
def getsumutil(st,ss,se,qs,qe,si):
    if(qs<=ss and qe>=se):
        return st[si]
    if(se<qs or ss>qe):
        return 0
    mid=getmid(ss,se)
    return getsumutil(st,ss,mid,qs,qe,2*si+1)+getsumutil(st,mid+1,se,qs,qe,2*si+2)
def updatevalue(arr,st,n,i,new_val):
    diff=new_val-arr[i]
    arr[i]=new_val
    updatevalueutil(st,0,n-1,i,diff,0)
def updatevalueutil(st,ss,se,i,diff,si):
    if(i<ss or i>se):
        return
    st[si]=st[si]+diff
    if(se!=ss):
        mid=getmid(ss,se)
        updatevalueutil(st,ss,mid,i,diff,2*si+1)
        updatevalueutil(st,mid+1,se,i,diff,2*si+2)

n=int(input())
L=list(map(int,input()))
stsum=constsum(L,n)
r1=[0]*n
r2=[0]*n
for i in range(n):
    if((L[i]+1)%6==0):
        r1[i]=1
    elif((L[i]-1)%6==0):
        r2[i]=1
```

```
str1=constsum(r1,n)
str2=constsum(r2,n)
q=int(input())
for i in range(q):
    a,b,c=map(int,input().split())
    if(a==1):
        l=getsum(str1,n,b,c)%n
        r=getsum(str2,n,b,c)%n
        res=getsum(stsum,n,min(l,r),max(l,r))
        print(str(res))
    else:
        updatevalue(L,stsum,n,b,c)
        if((c+1)%6==0):
            updatevalue(r1,str1,n,b,1)
        else:
            updatevalue(r1,str1,n,b,0)
        if((c-1)%6==0):
            updatevalue(r2,str2,n,b,1)
        else:
            updatevalue(r2,str2,n,b,0)
```

Feel free to share your approach, if you want to. (even if its same 😛) . Suggestions are welcomed as always had been. 🙂

PREREQUISITES

Dijkstra Algorithm and Segment Tree

PROBLEM

Given a directed graph with N nodes and N^2 edges, the cost of directed edge from u to v is given by A_u, where array A is given.

In this graph, exactly M modifications are made, each modification updating the cost of edge from u to v to cost c.

Find the sum of lengths of the shortest paths over all ordered pairs of distinct vertices.

QUICK EXPLANATION

- Treating edges as undirected, find the connected components of nodes whose edge weights are modified.
- For each node in component, we are going to run dijkstra to find minimum distance of two nodes within component.
- In order to execute dijkstra faster, we'd use lazy segment tree to simulate the dijkstra algorithm operations to process multiple edges with same weight faster.
- For each component, we also need to add an external node v with minimum value of A_v to handle case where path within component will have higher distance.

EXPLANATION

Let's start with brute force solution and then improve it.

The brute force approach would be to run floyd warshall all pair shortest path algorithm on given graph, and then taking the sum of distances of ordered pairs. This solution is sufficient for subtask 1.

Observation 1

Consider a node u not having any of its outgoing edge modified. Then we can see that all paths starting at that node have cost A_u, and since there's a direct edge from u to all other nodes with cost A_u, This node contributes $(N-1)*A_u$ to the total cost.

Hence, we can add $(N-1)*A_u$ to the cost and handle nodes which are endpoints of atleast one modified edge.

Subtask 3

By above observation, we can see that we only need to compute sum of distance of pairs starting at atmost $2*M$ candidate nodes.

Let's divide these nodes into connected components (assuming the modified edges were undirected). This way, any shortest path from inside one component to node outside that component would have to use at least one original edge.

Observation 2

Let's write the shortest path from u to v as $u \to x_1 \to \ldots \to x_k \to v$ where u is inside a component and v is outside that component.
Claim: All edges in this path till x_k are modified edges, and edge $x_k \to v$ is an original edge.
Proof: Let us first prove that all x_i lie in the same component as u. Let's assume the first x_i lies outside

component. Then edge $x_{i-1} \to x_i$ ($u \to x_1$ if $i = 1$) must be an original edge. But we can simply use original edge $x_{i-1} \to v$ having same cost $A_{x_{i-1}}$, reducing the cost. Hence, all x_i lies within same component as u.

The edge $x_k \to v$ cannot be a modified edge, since x_k is inside the component and v is outside. Hence, edge $x_k \to v$ is an original edge

Using above observation, if we have distance computed for all pairs of nodes (u, w) within same component, then distance from node u to all outside nodes shall have cost $min(dist(u, w) + A_w)$ over all w in component.

Hence, all we need to do is to compute the distance between all pairs of nodes within component, which can be done using floyd warshall algorithm

Corner Case

Due to modification of edge weights, there might be a case where the shortest path from u to v where u and v are within same component, may include a node w outside the component.

This happens, because due to updates, the direct path from u to v may actually have higher cost now.

Consider example

```
3
1 1 1
1
1 2 5
dist(1, 2) = dist(1, 3)+dist(3, 2) = 2
```

An easy way to handle that is by observing that for all nodes w outside component, it is sufficient to consider the one with minimum A_w, since the cost is $dist(u, x) + dist(x, w) + A_w$ where x is some internal node. This distance is written as $dist(u, x) + A_x + A_w$. The only term depending upon w is A_w, hence it is sufficient to add one node with minimum value of A_w to this component before computing all pair shortest paths.

Subtask 4

All this time, we relied on floyd warshall to compute all pair shortest paths within components of size upto M, which have at most M^2 edges.

Let's try dijkstra here. Since there are total $2 * M$ nodes as endpoint of at least one edge, we need to run dijkstra $2 * M$ times. Each run of dijkstra, if implemented normally, would take $O(M^2)$ time (proportional to the number of edges), which is too slow.

Optimizing dijkstra

Among these M^2 edges within component, only upto M edges are modified ones. Rest all outgoing edges from a node have same weight.

Revisiting dijkstra algorithm, the operations we usually need to support are (assuming implemented with min-heap)

- Finding the node with minimum distance
- Removing a node from our DS
- Update the distance to a node v with min of distance to node u plus edge weight of edge from u to v. (Called the relaxation operation)

Here, the first two operations are performed $O(V)$ time during dijkstra and third operation is performed $O(E)$ time during dijkstra, if V and E denote the number of nodes and edges in graph respectively. With $E = M^2$, we only need to speed up third operation.

Here, for a fixed node u, most of the outgoing edges have weight A_u. Can we process multiple relaxations simultaneously?

Segment Tree

Yes, Let's fix an order of nodes in the component and label them from 0 to $C-1$ where C is the size of component. Let's say we are processing node u, having already computed $dist(source, u)$, having outgoing edges to nodes $c_1, c_2 \dots c_k$ with weights $w_1, w_2 \dots w_k$ in sorted order of c_i.

The relaxations we need to perform are on node c_i with value $dist(source, u) + w_i$ (there are C of them over whole execution of dijkstra, so we perform these naively)
AND
relaxations on all nodes lying in any of the range $[0, c_1 - 1], [c_1 + 1, c_2 - 1] \dots [c_{k-1} + 1, c_k - 1], [c_k + 1, C - 1]$ with value $dist(source, u) + A_u$.

The second type of relaxations have same value $X = dist(source, u) + A_u$. It is equivalent to replacing $dist(source, w)$ with X for all w lying in any of the range.

This operation can be supported by segment tree, where i-th leaf stores the distance to i-th node from source node.

The operations we need to support by segment tree are

- Find position with minimum value (equivalent to finding topmost element of min-heap)
- Removing position (equivalent to popping out topmost element). This can be done by maintaining set of active positions.
- Updating all $dist(source, w)$ with $min(X, dist(source, w))$ for $w \in [l, r]$

These operations can be supported by lazy segment tree, maintaining state of active leaves (initially all are active, and each pop operation marks one of them inactive).

Refer my code for implementation details.

TIME COMPLEXITY

The time complexity is $O(N + M^2 * log(M))$ per test case.

PREREQUISITES:

2-pointer , Segment Tree

PROBLEM:

Given an array we are required to find minimum length subarry such that by changing atmost X elements in that subarray, we can have total sum of array be equal to K.

EXPLANATION:

Given a subarray $A_l, A_{l+1}, ..., A_r$, we first need a way to check if by changing intensities of atmost X elements from that subarray only, can a total array sum of K be achieved. Let the total sum of array be S and sum of subarray be s. Besides, let the sum of smallest X elements(if number of elements is less than X all the elements of subarray is considered) in the subarray be $smls$ and the sum of biggest X(if number of elements is less than X all the elements of subarray is considered) elements be $bigs$. It can be followed that the minimum sum of the array that can be achieved by changing atmost X elements in our subarray will be $from = S - bigs + min(x, lenofsubarray)$. Similarly, maximum sum of the array that can be achieved by changing atmost X elements in our subarray will be $to = S - smls + min(x, lenofsubarray) * n$. So if $from \leq K \leq to$, we can achieve total array sum of K by changing atmost X elements in our subarray.

Now we need to find a way to minimise the subarray length. This can be done by two-pointer techinque. We will have two poninters, i and j, both starting from 0. This represents that $A_i, A_{i+1}, ..., A_j$ is our current subarray under consideration. We will shift the j pointer right until we find a subarray which satisfies our condition. Then, we will shift i pointer right until our condition is not violated. All this while maintaining minimum length of subarray achieved.

The only thing left now is to find a way such that we can efficiently compute smallest and largest X elements in our subarray. This can be done using segment tree. The segment tree will initially be empty. When an element becomes part of our current subarray(due to j pointer shifting right), we will insert that element in our segment tree and when an element is removed from current subarray(due to i pointer shifting right), we will remove that element from the segment tree. Note that in the segment tree, element will be inserted at the position which the element occupies in sorted form of array A. For example, if an element e is at index $i1$ in A but on sorting A, e occupies index $i2$. So when e becomes part of our current subarray, we will make an update for index $i2$ in the segment tree. This way instead of finding sum of smallest X elements in the segment tree, we can equivalently find sum of leftmost X element in the segment tree. Similarly, sum of largest X elements will be equal to sum of rightmost X elements. This is because the elements in the segment tree are always in sorted order.

The sum of leftmost or rightmost X elements can be easily computed by maintainig a segment tree where each node stores 2 values, number of elements and sum of elements in the range represented by the node. As an example, consider we want to compute the sum of leftmost X element in the segment tree. Let the left child of root have values $c1$(count) and $s1$(sum) and right child have value $c2$(count) and $s2$(sum). If $c1 \geq X$, we can recursively query for sum of leftmost X elements at root->left. If $c1 < X$, the answer will be sum of $s1$ and result of query for sum of leftmost $X - c1$ elements at root->right.

PREREQUISITES

Segment Tree, Number Theory

PROBLEM

Given an array A with N integers, you have to answer Q queries, where each query may be one of the following form

- $1\ L\ R\ (R > L)$ which asks you to find $\gcd(A_L, lcm(A_{L+1}, \gcd(A_{L+2}, ..., ((R-L) \bmod 2 == 1? \gcd(A_{R-1}, A_R) : lcm(A_{R-1}, A_R)) \ldots)))$.
- $2\ L\ R\ (R > L)$ which asks you to find the same as above but lcm swapped with $\gcd$ and vice-versa.

EXPLANATION

Different Problem

Let's forget the original problem for this section and try to solve following one.

A function f is defined as $f_{x,y}(z) = min(x, max(y, z))$ for some given x and y. For example, $f_{10,3}(z)$ is a function representing $min(10, max(3, z))$

You are given four integers x_1, y_1, x_2, y_2 and you need to find any two integers x_3 and y_3 such that $f_{x_1,y_1}(f_{x_2,y_2}(z)) = f_{x_3,y_3}(z)$ holds for all possible values of z

Give this problem a thought, as solving this problem is crucial to solving the original problem.

We have $min(x_1, max(y_1, min(x_2, max(y_2, z)))) = min(x_3, max(y_3, z))$.

The issue here is that we need to somehow swap the inner min and max. The **distributive property** comes useful here. We can write $max(a, min(b, c)) = min(max(a, b), max(a, c))$. Applying distributive property on $max(y_1, min(x_2, max(y_2, z)))$, we get

$min(x_1, min(max(y_1, x_2), max(y_1, max(y_2, z)))) = min(x_3, max(y_3, z))$. Now we can apply associative property to rearrange the expression into

$min(x_1, max(y_1, x_2), max(y_1, y_2, z)) = min(x_3, max(y_3, z))$
$min(min(x_1, max(y_1, x_2)), max(max(y_1, y_2), z)) = min(x_3, max(y_3, z))$

Now, we actually compare terms on both sides, which would imply
$x_3 = min(x_1, max(y_1, x_2))$
$y_3 = max(y_1, y_2)$. Or we can write $y_3 = min(x_1, max(y_1, y_2))$ as we know that later we have to take min with x_1
.

Using functional representation, we can see that the values $x_3 = f_{x_1,y_1}(x_2)$ and $y_3 = f_{x_1,y_1}(y_2)$ are the solution to this problem.

Similarly, if we have $g_{x,y}(z) = max(y, min(x, z))$, then we could have solved the problem same way, obtaining $x_3 = g_{x_1,y_1}(x_2)$ and $y_3 = g_{x_1,y_1}(y_2)$

Problem similar to original problem

Replace gcd by min and lcm by max in original problem statement. We have to answer queries of that type now.

Since we have to answer range queries, the possible data structures here are segment tree, sparse table or any other similar structure. One common thing in above structures is how they compute some quantity for whole

range quickly, and we divide the query range into $O(log(R-L))$ ranges, and try to merge the results of those ranges.

So here, assuming we choose segment tree, Consider a node representing interval $[L, R]$. We shall store two constants X and Y such that function $f_{X,Y}$ represents the answer to query for this range. Specifically, $min(A_L, max(A_{L+1}, min(A_{L+2}, ...((R-L) \bmod 2 == 0? \min(A_R, z) : max(A_R, z)) \ldots))) = f_{X,Y}(z)$.

It is basically computing the net effect of applying merging operation with all elements in range quickly.

Let's say we have two nodes representing ranges $[L, M]$ and $[M+1, R]$ represented by functions f_{x_1,y_1} and f_{x_2,y_2}, we need to find values X and Y such that $f_{x_1,y_1}(f_{x_2,y_2}(z)) = f_{X,Y}(z)$ holds. It is precisely same to the problem we discussed above.

Hence, if we can choose appropriate values for leaf nodes, we now know how to merge results of two nodes, thus computing the whole tree before queries and answering queries in $O(log(N))$ time.

We shall store two function parameters at each node. Specifically, we shall store X_1 and Y_1 such that f_{X_1,Y_1} answers our query of type 1. Similarly, we shall store integers X_2 and Y_2 such that g_{X_2,Y_2} answers our query of type 2

Choosing X_1, Y_1, X_2 and Y_2 for leaf nodes
Let's say we have a leaf node representing range $[p, p]$. We need to choose X_1 and Y_1 such that $min(A_p, z) = f_{X_1,Y_1}(z)$. We can choose $X_1 = A_p$ and $Y_1 = -\infty$

We need to choose X_2 and Y_2 such that $max(A_p, z) = g_{X_2,Y_2}(z)$ holds for all z. We can choose $X_2 = \infty$ and $Y_2 = A_p$.

Edge case
When merging two child nodes, we need to check if the left child represent an even number of sub-operations or odd number of operations, as the next sub-operation (min or max) to be applied depends upon the parity of length of range covered by left child.

Specifically, if range length of left child is even, we need to find X_1, Y_1 such that $f_{X_1,Y_1}(z) = f_{x_1,y_1}(f_{x_2,y_2}(z))$. But if the range length of left child is odd, we have $f_{X_1,Y_1}(z) = f_{x_1,y_1}(g_{x_2,y_2}(z))$.

Composition $f_{x_1,y_1}(g_{x_2,y_2}(z))$ is easy to calculate, as the inner two max operations can be merged after expanding. Similarly, $g_{x_1,y_1}(f_{x_2,y_2}(z))$ is evaluated by merging inner min operations.

Relation to original problem

Have a careful read of section [Fundamental theorem of Arithmetic] especially how they write

- gcd operation as minimum of values of exponents in prime factorization of two numbers
- lcm operation has maximum of values of exponents in prime factorization of two numbers

The exponent of prime p_1 is independent of exponent of a different prime p_2.

So, it is possible to solve the original problem by

- Considering all primes one by one
- For prime p, define $B_i = max(j : p^j | A_i)$, i.e. maximum power of p which divides A_i.
- Compute the answer for all primes for each query range using above problem.
- Answers computed for each prime are the exponents of those primes in answer of original query.

Let's assume array $A = [12, 6, 4, 8, 9, 2, 3]$. When solving for $p = 2$, we have $B = [2, 1, 2, 3, 0, 1, 0]$ and for $p = 3$, we have $B = [1, 1, 0, 0, 2, 0, 1]$.Let's say we compute answer A_2 for $p = 2$ and A_3 for $p = 3$.

Then the answer to our original problem query would be $2^{A_2} * 3^{A_3}$

But doing this for every prime for every query leads to time complexity $O(Q * max(A_i) * log(N))$ which is too slow.

Magic of multiplicative functions

Have a look at **this** explanation of Multiplicative functions.

We can replace the min function by gcd and max function by lcm in our previous solution, and compute the answer for all primes simultaneously.

This works because in each prime, gcd and lcm acts on exponents of different primes independently.

Hence, we adapt our segment tree solution by replacing min with gcd and max with lcm to solve the problem.

Implementation caveats

- Earlier we choose initial values for leaf nodes assuming min and max operation. Now $f_{X_1,Y_1}(z) = gcd(X_1, lcm(Y_1, z))$. We need to choose values X_1, Y_1 such that $f_{X_1,Y_1}(z) = gcd(A_i, z)$ for i-th leaf. We can choose $X_1 = A_i$ and $Y_1 = 1$.
- Similarly, we choose $X_2 = 0$ and $Y_2 = A_i$ such that $g_{X_2,Y_2}(z) = lcm(Y_2, gcd(X_2, z)) = lcm(A_i, z)$
- 0 here satisfies the properties of ∞ and 1 works for $-\infty$, as $gcd(1, x) = 1, lcm(1, x) = x$ and $gcd(x, 0) = x$ and $lcm(x, 0) = 0$ (Technically it is not defined, but assuming this is consistent with our problem.
- It is possible that when we distributed lcm operation, we may end up computing lcm of three integers of values close to 10^9, which can result in lcm being close to 10^27 overflowing even long data type. Easy work around is to apply gcd with every term where it can be applied. This issue happens when computing $f_{x_1,y_1}(g_{x_2,y_2}(z))$

TIME COMPLEXITY

The time complexity is $O((N + Q) * log(N) * log(A_i))$ per test case theoretically, but works faster in practice as $log(A_i)$ to compute gcd and lcm is the worst case scenario.

PRE-REQUISITES:

Observations, **Segment Tree & Lazy Propagation** , **Binary Lifting** OR **Persistant Segment Tree**

PROBLEM:

You are given a permutation (array) of size N, answer Q queries of form:

- $L\ R-$ Find maximum length subarray such that the subarray has consecutive numbers.

QUICK-EXPLANATION:

Key to AC- You either need to observe the lemmas or define what operations you need and select appropriate data structure for the problem.

We will explain setter's solution which uses observations and binary lifting. First, observe the following lemmas:

- Call a subarray $[i,j]$ as "good" if all numbers in it can be re-arranged such that they are all consecutive.
- If 2 intersecting subarrays, A and B are good, then their union $A \cup B$ is also good.
- If 2 intersecting subarrays A and B are good, then their intersection $A \cap B$ is also good.

Lets say we have an array $right[i]$ which tells that "$[i, right[i]]$ is the largest good subarray possible starting from i". Similarly define another array $left[i]$. Now, we have this slow, but correct algorithm:

- Start with $i = L$ and $j = R$. Set $ans = 1$.
- We know that array $[i, right[i]\ ]$ is good. If it also lies within range $[L, R]$ of query, then update $ans = max(ans, right[i] - i + 1)$ and set $i = right[i] + 1$. Do this until either $right[i] > R$
- While $j \geq i$ and $left[j] \geq i$, consider subarray $[left[j], j]$ and update answer as $ans = max(ans, j - left[j] + 1)$. Update $j = left[j] - 1$
- If $i > j$, we are done. Else we are yet to consider the subarray $[i,j]$. By lemma 2, since it is an intersection of 2 good subarrays, the subarray $[i,j]$ is also good. Hence we do a final check for $ans = max(ans, j - i + 1)$.

To speed the algorithm up, instead of considering all subarrays one by one, consider them in groups of 2^j at once by binary lifting and update answer considering the largest subarray in that group.

Now only thing we need is to calculate these arrays $left[]$ and $right[]$. This is achieved via segment tree with lazy propagation. Some elaboration on that is given below:

- Say we are updating for index j - this means we will consider all subarrays ending at index j.
- Let $max[i...j]$ represent maximum element in range $[i,j]$. Similarly define $min[i...j]$.
- Notice that for a good subarray $[i,j]$, $max[i...j] - min[i...j] + (i - j) = 0$.

- Maintain 2 monotonous queues, one for maintaining max and one for min. We will use them to appropriately update $max[i..j]$ and $min[i..j]$ parts above for the subarrays.
- The $(i-j)$ part is handled by just adding a -1 in range $[1, i]$ (You will notice that -1 will be added $i-k$ times for an index k such that $k < i$, hence keeping track of this value correctly.
- The same algorithm can be repeated to get the other array, by just reversing the direction of iteration.

EXPLANATION:

We will divide this editorial into 4 sections:

- Discussing and proving the lemmas.
- Discussing how $left[]$ and $right[]$ arrays lead us to a solution.
- Discussing Binary lifting on above.
- Discussing how to calculate the $left[]$ and $right[]$ arrays to wrap up the solution.

This editorial uses point-style format. A little bit of experimentation to keep things clear and maintain stress on important parts. A bit of downside is that the editorial will appear non-trivially lengthier than paragraph-style counterpart (as you might have seen in QE 🙁). So please make sure that you understand each and every point where-ever bullet points are used.

So lets get started 🙂

PS: A good sub-array, referred below has same meaning as in Quick Explanation, i.e. "Call a subarray $[i,j]$ as “good” if all numbers in it can be re-arranged such that they are all consecutive. "

Discussing and proving the lemmas

Observing these lemmas was quite a task - because while they are easy to prove, you'd need to have some eye or amount of effort in coming up with how they are relevant to solving the problem. This section will prove them just for sake of clarity, and so we can freely invoke them in further section(s) without any fear of unknown. 😃

Lemma 1: If two $intersecting$ subarrays, say A and B, are good individually then their union $A \cup B$ is also good.

I will give an intuitional proof here. However, you can also find an alternate proof in the bonus corner for the same.

We know that A and B are intersecting , so their intersection has at least 1 element. In other words, $A \cap B \neq \phi$.

We define 2 variables:

- $A^{'} = A - A \cap B$, in other words, $A^{'}$ has elements belonging ONLY to A and not to B.
- $B^{'} = B - A \cap B$, in other words, $B^{'}$ has elements belonging ONLY to B and not to A.

One thing should be clear, that $A = A^{'} + A \cap B$ and similarly $B = B^{'} + A \cap B$.

Now, I argue the following:

- I start with set $A^{'}$.
- If I append $A \cap B$ at end of it, I get A, which is a good set.
- I also know that if I start with $A \cap B$, and append $B^{'}$ at end of it (or vice versa), I get B which is another good set.
- Now, A is good with all elements consecutive. I add $B^{'}$ at end of it. Since A contains all elements of $A \cap B$, and those elements make a good set if we add elements of $B^{'}$ in the set, then $A^{'} + A \cap B + B^{'}$ is a good set. But we know that $A \cup B = A^{'} + A \cap B + B^{'}$.
- The argument becomes even simpler if we remove the abstraction of "good". The argument simply states that since all elements of set $A^{'} + A \cap B$ are consecutive, and all elements of set $A \cap B + B^{'}$ are also consecutive, then all elements of set $A^{'} + A \cap B + B^{'}$ should be consecutive as well given that repetitions are not allowed!

Hence proved. We can also prove the same via contradiction as in the bonus section.

Lemma 2: If 2 intersecting subarrays A and B are good, then their intersection $A \cap B$ is also good.

The above seems unintuitional at first - and hence it seems a bit complex to prove as well. But I have reaped the easiest way to prove it to you, so read the below carefully:

- We proceed via contradiction. Say $A \cap B$ is NOT good! This means there are some non-consecutive elements in it.
- If the intersection, $A \cap B$ is of size 1, it is good by definition and the proof ends. Else, we continue below with assumption that size of $A \cap B$ is at least 2.
- Recall our definition of $A^{'}$ and $B^{'}$.
- Since A (i.e. $A^{'} + A \cap B$) is a good subarray, this means all the "missing" elements between the non-consecutive elements of $A \cap B$ are supplied by $A^{'}$!
- Similarly, since B (i.e. $B^{'} + A \cap B$) is a good subarray, this means all the "missing" elements between the non-consecutive elements of $A \cap B$ are supplied by $B^{'}$!
- This means those "missing elements" lie in $A^{'}$ as well as $B^{'}$
- But the question guarantees that all elements are distinct! Hence, a contradiction!

Hence our assumption that $A \cap B$ is not a good subarray is false!

Both lemmas are hence, proved.
How the left and right arrays can lead us to solution

Starting here, we will take the first step towards solution.

Lets assume that somehow we got 2 arrays as follows:

- $left[]$: This array is defined as, $left[i]$ will contain the leftmost index such that $[left[i], i]$ is a good subarray. Note that $left[i]$ can be equal to i.
- $right[]$: Similar to above, $right[i]$ will contain the rightmost index such that $[i, right[i]]$ is a good subarray. Note that $right[i]$ can be equal to i.

If we have these 2 arrays, then we can solve the problem with following slow, but correct solution:

- Say my query range is $[L, R]$
- Define two variables, $i = L$ and $j = R$. Set $ans = 1$.
- We know that array $[i, right[i]\ ]$ is good. If it also lies within range $[L, R]$ of query, then update $ans = max(ans, right[i] - i + 1)$ and set $i = right[i] + 1$. Do this until either $right[i] > R$
- Now, while $j \geq i$ and $left[j] \geq i$, consider subarray $[left[j], j]$ and update answer as $ans = max(ans, j - left[j] + 1)$. Update $j = left[j] - 1$. Our limit here is i and not L because we have already processed the range $[L, i]$
- If $i > j$, we are done. Else we are yet to consider the subarray $[i, j]$. We know that $[i, j]$ is an intersection of two good subarrays $[left[j], j]$ and $[i, right[i]]$. Hence, by lemma 2 the subarray $[i, j]$ must also be good! Hence, we update $ans = max(ans, j - i + 1)$ and conclude the result.

Now, why is the above correct? The reason we need to spend time on this, despite any hunch or intuition one might have is, the above algorithm is greedy (more on the bonus corner).

The proof is actually short and simple. We will use proof by contradiction.

- Say query is $[L, R]$. We define i and j as in our algorithm.
- Lets consider the subarray $[i, right[i]]$. The proof can be replicated for other direction, i.e. $[left[j], j]$
- Say there is some index k, such that $i < k < right[i]$ and that $[k, right[k]]$ is actually the largest good subarray (and hence solution to our query).
- By lemma 1, we know that if A and B are two intersecting good subarrays, then their union is also good. Consider $A = [i, right[i]]$ and $B = [k, right[k]]$. Since both A and B are intersecting, as well as good, $A \cup B$ must be good as well! Hence, $[i, right[k]]$ is also a good subarray!
- But if this is the case, then recall the definition of $right[i]$. It is the rightmost index upto which $[i, right[i]]$ is good! Hence, either $right[i] == right[k]$ (which means $[k, right[k]]$ cannot be optimal solution as its smaller than $[i, right[i]]$) or we got a contradiction as if $right[k] > right[i]$, then by definition $right[i]$ should have extended further upto $right[k]$.

With this, we conclude this section. It will help you in further section if you are clear with every point of the algorithm mentioned here. In case of issues, please quote the text and ping 🙂

Applying Binary Lifting on above

Contrary to your expectations, this section is a breeze. The only pre-requisite you need is that you should know basic binary lifting (eg - able to find LCA using that), and the algorithm described in section 2.

In previous section, our solution was correct, but slow. It was slow because it was considering each good subarray one by one, and in worst case (when all good subarrays have size 1), it will take $O(N)$ to answer the query.

What is our bottleneck above? We are considering all good subarrays one by one. What if we consider multiple of them at once? This is where binary lifting comes in. The algorithm will go like-

- Define a variable k. Note that L, R, i, j etc hold the definitions as in quick explanation and previous section.
- Now, earlier if I am currently at index i, I will just check that $[i, right[i]]$ is within query or not. Instead, I will now check for 2^k of them at once now!
- In other words, I will check if all those 2^k good subarrays lie within query or not. If not, I will decrement k until they all lie within query.
- Once I know that now next 2^k groups are lying in the query region, I just need the size of maximum good subarray among them!
- Hence, I re-define the $right[]$ (and hence $left[]$) arrays so that $right[i][k]$ stores a pair of values, which are: $(\text{end point of } 2^{k'th} \text{group, size of longest good subarray in this group})$
- In easier words, I am just storing the required endpoint (so I know which index to process next), and size of longest good subarray among all 2^k subarrays we considered. These two values are stored in $right[i][k]$

Essentially, we will consider multiple good subarrays (eg - $[i, right[i]], [right[i] + 1, right[right[i]]], \ etc$, check size of maximum good subarray among them, and proceed to the next group(s) of good subarrays to process.This processes multiple groups at once and makes our solution fast.

At the end of the day, if you were clear with the brute force algorithm in previous section, you will find we are doing no great feat here- we are just considering multiple groups at once and processing them all at once.

As a closing question to this section, are you able to appreciate the idea of grouping? If yes, then can you also appreciate on how binary lifting can actually be inconsequential to (i.e. not an important part of) the main solution? More about it in the bonus corner 🙂

Discussing how to calculate left and right subarrays

Note - In this section, i and j do not have same meaning as in previous sections. $[i,j]$ will have its traditional meaning, i.e. a subarray starting at i and ending at j.

Now on to the big supper. We know that if we *somehow* have the $left[][]$ and $right[][]$ arrays, then we can very easily solve the problem. But how do we get them?

To keep things simple, this section is divided into 2 sections, each focussing a key step to find these arrays.

Finding binary lifted left and right arrays if we have original left and right arrays

What is the point of this section? The point of the section is that, to find $left[i][k]$ and $right[i][k]$ we require the original $left[]$ and $right[]$ arrays for base case. I want to keep the confusing/tough part for the last because once you get stuck at one point, nothing below that point in editorial makes sense to you.

Note -In this section, I will refer to the original (i.e. without binary lifting) left and right arrays as $left_{old}[]$ and $right_{old}[]$.

Now, suppose we have the original $left_{old}[i]$ and $right_{old}[i]$ arrays as discussed in brute force approach. How can we make the binary-lifted $left[][]$ and $right[][]$ arrays from them?

Without loss of generality, lets discuss calculating $right[i][k]$. The process to get $left[i][k]$ is pretty much the same.

$Base\ Case-$

- If we have the $right_{old}[]$ array, then we can calculate the values of binary lifted $right[i][0]$ easily! Since $right[i][0]$ considers $2^0 = 1$ group, i.e. itself, its endpoint will be $right_{old}[i]$ and size will be $right_{old}[i] - i + 1$. Hence, $right[i][0]$ is calculated for every i.

$Recurrence-$

The relation is actually pretty same. Say we are calculating $right[i][j]$ and that we have all values of level $right[i][j-1]$ for all i and j.

- Recall that $right[i][j]$ stores a pair of 2 things - the ending point of the 2^j good subarrays if we start from i, as well as size of longest good subarray among them. Now, the ending point can be easily calculated as:
- $\text{Ending Index} = right[(right[i][j-1]+1)][j-1]$
- Recall that $right[i][j-1]$ stores ending point of 2^{j-1}'th good subarray starting from i. So $right[i][j-1]+1$ is the starting point of immediately next good subarray. Starting from there, if we advance 2^{j-1} good subarrays more, then we know the ending point of 2^j'th good subarray starting from i. Thats the interpretation of above.
- The maximum element can be calculated as : $\text{Maximum Element} = max(right[i][j-1].maximum, right[(right[i][j-1]+1)][j-1].maximum)$
- Notice that our entire group of 2^j subarrays is a combination of two groups of size 2^{j-1}, from $[i, right[i][j-1]]$ and $[right[i][j-1]+1, right[right[i][j-1]+1][j-1]]$. All that we are doing is that, “Longest size of good subarray in this group of size 2^j = max(longest size if first 2^{j-1} good subarrays, longest size in next 2^{j-1} subarrays)”. That is the interpretation of above expression.

With this the recurrence of binary lifting reaches its conclusion. We have the base case to generate $right[i][0]$, and using that we can generate $right[i][1]$ and then similarly use $right[i][1]$ to get $right[i][2]$ and so on.

Check yourself: When we usually combine groups, we check at 3 places instead of two. Consider a separate problem, you are asked to tell maximum number of consecutive $1's$ in a given range. You will check at 3 places instead of 2, one each for the 2 groups you are combining (for max of answer for those groups), and one more at the point where both groups merge, to see if it gives a new maximum once the groups combine. Why did we not do it here?

Generating original left and right arrays

Congrats if you made it till here. Take a break if you want, you should try to give your full attention to this section. Dont forget to stay hydrated, warm and kind 🙂 .

Lets start with basic first. Its not very hard to observe that if a subarray/sequence is good (i.e. consecutive), then $\text{Max Element - Min Element}$ gives one less than the length of the sequence! Meaning, for a good subarray $[i,j]$, we will have-

$Max[i...j] - Min[i...j] = j - i$ where $Max[i...j]$ represents max element in range $[i,j]$ and similarly $Min[i...j]$ represents the minimum element in the range.

Hence, we deduce that for a good subarray, $Max[i...j] - Min[i...j] + (i - j) = 0$. We will use this to calculate the $left[]$ and $right[]$ arrays.

To do so, we will follow the below algorithm. Note that the below algorithm calculates the $left[]$ array. We can reuse it to get the $right[]$ array as well by reversing the direction.

- We make an array $var[]$. Our plan is, we will process all indices one by one. Say we are processing index j, we will consider all subarrays ending at j. Hence, $var[i]$ (for $i < j$, obviously) will store if $[i,j]$ is a good subarray or not.
- $var[i]$ will achieve the above by storing value of $Max[i...j] - Min[i...j] + (i - j)$. If its 0, then $[i,j]$ is a good subarray.
- We will use segment tree with lazy propagation (to support range updates) to do the above.
- We will make 2 monotone queues, one to keep track of the required range's maximum and other to keep track of required range's minima. If you have trouble seeing why they are needed, check out **this** problem. The queues will perform function similar to stack to tell the correct values of $max[i...j]$ and $min[i...j]$. You can check bonus section for more details. For each of required ranges, we will add $max[i...j]$ to it and subtract $min[i...j]$ from it.
- The only thing remaining is to keep track of is value $(i - j)$. While it seems tricky to keep track of, its actually easy. If we are processing j, we add a -1 to range $[1, j - 1]$ and this part is done! (More explanation at bonus corner)
- All thats remaining is to check that out of all positions where $val[i]$ is 0, which one is the leftmost. This is a trivial segment tree query.

You can additionally check **this** comment for alternate explanation.

And with these, you are completely done with the problem! The good part is, in case you are not sure how to go about implementing it, you will be able to understand setter's code pretty well enough to inspire (and still not copy) the implementation.

With this, we are done with the hardest problem of set.

SOLUTION

Note, setter's $right[]$ and $left[]$ arrays are not both end-point inclusive, i.e. instead of $[i, right[i]]$ being a good subarray , $[i, right[i] - 1]$ will be the good subarray.

Setter

Tester

$$Time\ Complexity = O((N + Q) * LogN)$$
$$Space\ Complexity = O((N + Q) * LogN)$$

CHEF VIJJU'S CORNER 😃

- Prove/Disprove that $left[]$ and $right[]$ array's values exist for every index i. If you disprove it, discuss how it still leads to correct solution.

Solution

If we allow $left[i] = i$ and similarly allow $right[i] = i$, the values exist for all values. Else they do not. Setter's solution allows the above, except that it stores $i + 1$ instead of i for implementation ease.

Proof via contradiction for A U B being good.

Lets assume that A U B is NOT good despite A and B being good. Now, A and B are intersecting as well, so their intersection has at least 1 element. In other words, $A \cap B \neq \phi$.

We define $A^{'}$ and $B^{'}$ similarly as done in editorial. Now, we do the following-

Since A U B is not good, there must be at least 2 elements which are not consecutive! Since A U $B = A^{'} + A \cap B + B^{'}$, following cases arise here:

- Both these non-consecutive elements belong only to $A^{'}$ or only to $B^{'}$
- One of the elements belongs to $A^{'}$ or $B^{'}$ and other to $A \cap B$
- Both elements belong to $A \cap B$

Given that all elements are distinct, if any of the non-consecutive elements belong to $A^{'}$ or $B^{'}$ - we are done with proof. This is because then we can argue that "If in the set of $A^{'} + A \cap B + B^{'}$, these non-consecutive elements come from, say $B^{'}$ alone, then how is B (i.e. $B^{'} + A \cap B$) good?". In other words, if the missing elements between these 2 non-consecutive elements in $B^{'}$ are not contained in $A \cap B + A^{'}$, then they wont be contained in just $A \cap B$ either! Similarly we can argue if both offending elements belong to $A^{'}$.

The second possibility of one of them belonging to $A \cap B$ and other to $A^{'}$ or $B^{'}$ is easily disregarded because then either A (i.e. $A^{'} + A \cap B$) or B (i.e. $B^{'} + A \cap B$) will not be good because of presence of offending elements in them!

The final possibility, is trivially disregarded by lemma 2 🙂

I felt that this proof makes a lot more sense, and is more welcoming if read after intuitions in editorial, and hence I kept it here instead of in main editorial so things remain concise.

Note that one merit(?) of this proof is that it proves both the lemmas at once, and can lead to contestant getting both observations at once (…or perhaps neither at all if he gets stuck!)

Why you need to spend time proving correctness of brute force in section 2

Ultimately the algorithm is saying that "If I am at index i and going right, I check $[i, right[i]]$, which is the largest good subarray starting at i, and update i to $right[i] + 1$. I do not check for any index between i and $right[i]$ because i is directly updated to $right[i] + 1$".

How are we sure that no index in range $(i, right[i])$ [both exclusive] will lead to correct answer? This point usually is a litmus test on if the problem is greedy or if dp or some other algorithm is required. While its true that a lot of people try for proof by AC in long, it cannot be denied that developing such skills to quickly prove/disprove algorithms helps a lot to advance rating greatly and is indispensible in interviews.

Observation at end of section 3

That was a bit of closing question. I guess if you clicked here, you want to know why I said binary lifting might not actually be a major part of solution? Lets go:

Focus on our bottleneck, and in high level terms tell me what we did to solve it.

The bottleneck was that we were processing good subarrays one by one and that made the solution slow. To solve it, we got an algorithm to process “groups” of subarrays at once!

The answer lies there. Binary lifting is not the only algorithm which can divide your data into groups. Can you think of others? I think square root decomposition etc are good candidates. Basically, we took one algorithm to group our data so we can process entire groups at once. So if you have some other algorithm which groups data (and is applicable here), you can also use it! Hence, if you look from a higher view, Binary Lifting is just an “implementation” of the grouping algorithm required, and we can replace it with any other grouping algorithm we deem fit. Our algorithm will still be correct with the new grouping mechanism.

This is what the software industry calls “Plug-in, Plug-out” principle. They keep the parts of their software as independent as possible, so that if they feel that some part needs optimization or rewriting, they simply “plug” it out, perhaps change to a better implementation, and plug it back it. Given that you did not do a design mishap (As in, using some unneeded variables/references from other part of program), this works really flexibly. And this is why companies etc, at least here in India, try to force candidate to complete a function with required logic (where company can control what information you get/use and dependencies on other parts of code you create) instead of candidates writing entire code themselves.

Why we did not need to check at point of merging while binary lifting

Because all our good subarrays are disjoint!

In the problem of given consecutive $1's$, it is possible that on combining 2 groups (simplest example being $[1, 1, 1]$ and $[1, 1]$) we get a new answer.

Whether you have to check at point of union or not is highly problem specific. In our case, we already establish that going over the groups one by one is ideal. In above example, notice that the ideal sequence of $1's$ was separated into two groups to lie partially in both. But in our case, we are including the whole group in range.

Point of union is checked if answer can lie partially in left and right subgroup - but since we are not dividing the good subarrays at all - instead keeping them whole, and given the fact that they are all disjoint, we do not come across this issue. Hence, simply taking max of answer of the 2 groups works.

Use of 2 queues in calculating left and right arrays

These queues are needed to keep track of required range's maxima and minima. Lets see this with an example.

Say my array is $[1, 2, 5, 4, 3, 6]$. Say we are currently at index 5 (1-based indexing), i.e. at element 3. Now, the maximum element in range $[1, 5]$, $[2, 5]$ and $[3, 5]$ is 5. But the maximum element in range $[4, 5]$ is 4.

Hence, to make sure the values of $max[i...j]$ and $min[i...j]$ remain correct in the solution, we use the queues.

Why adding -1 in range 1 to j-1 is sufficient

We want to keep track of value $(i - j)$ here. See that while processing each j, we add -1 to range $[1, j - 1]$. This means:

- At Index $j - 1$, -1 is added only once by while processing j.
- At index $j - 2$, a -1 was added while we processed for $j - 1$. j added another -1. Total reduction is -2, which is desired.
- At index $j - 3$, we added -1 while processing indices $(j - 2, j - 1, j)$ and hence total of 3 is deducted which is the requirement
- …
- At index 1, we added -1 while processing indices $(2, 3, 4, ..., j)$ and hence its added $(j - 1)$ times, which is the required deduction.

If we are at j, then to account for term $i - j$, we want value at $j - 1$ to be reduced by 1, …, value at index 1 to be reduced by $j - 1$, which is what happens above.

Setter's Notes

First we have to establish a few lemmas :

- $Lemma1$: If two intersecting subarrays are good then their union is also good.
- $Lemma2$: If two intersecting subarrays are good then their common subarray is also good.

Now let's try to solve another problem : We are given queries of form (x, y) and we have to divide the subarray (x, y) into minimum number of consecutive good subarrays.

For example if subarray (x, y) looks like $(4, 3, 1, 8, 9, 7)$ then the answer would be 3 : $(4, 3)(1)(8, 9, 7)$. To solve this problem we have to realize that we should take the biggest good subarray we can from the start of (x, y) and then solve the problem for the rest of it (some sort of greedy algorithm). The reason is deducted from the Lemmas introduced above.

Knowing all this, we are left with this greedy problem. To solve it we have to define an array $right[i]$: The maximum index where subarray $[i, right[i]]$ is good. Similarly define $left[i]$. Now we can answer a query like this : while $right[x] \leq y$, we take subarray $[x, right[x]]$ and assign $x = right[x] + 1$. This is correct since we should act greedy as described earlier. So now we have $right[x] > y$. We can do the same thing from the end of the array, i.e while $left[y] \geq x$ we take this subarray $[left[y], y]$ and assign $y = left[y] - 1$. Now if $x > y$ then the problem is solved. Otherwise

we know that $left[y] < x <= y < right[x]$. Considering lemma 2 we deduct that subarray [x, y] is actually also good so we can take the whole subarray as one piece.

This solution solves the problem but it's slow. We can speed it up via binary lifting; Instead of taking good subarrays one by one, we can take them in goups of 2^i resulting in $O(log(N))$ time. Now all we are left to do is calculate $left[i]$ (right[i] is calculated similarly).

We can loop over r from 1 to N and keep track of a segment tree where the $l-th$ index of this segment tree equals to $max(r-l) - min(r-l) - (r-l)$. Then a subarray is good if it's value is equal to zero. We also know that the values are all non-negative so we can keep track of minimum value, then $left[r]$ is the minimum index where we have a minimum (zero) in our segment tree. We can keep track of this segment tree in $O(N * log(N))$ so the problem is finally solved in $O((N + Q) * log(N))$.

There's one more lemma left :

- $Lemma3$: An optimal division of an interval into minimum number of good subarrays always includes a maximum good subarray in that interval. This lemma is derived from lemma 1.

Now that we know how to divide the subarray (x, y) into minimum number of good subarrays, we can also keep track of the maximum size of a good subarray in our binary lifting. Thus the problem is solved in $O((N + Q) * log(N))$. This concludes the solution.

- The alternate solution of this problem via persistent segment tree is not mentioned in editorial because that'd have made the editorial too verbose. However, if you have a command over this data structure, the question just reduces to making the data-structure do a set of operations. In that case, **this** editorial by @tmwilliamlin can help you.

Why is this editorial so verbose bruh

I cant help it!

I know that there are a LOT of places I could have just said "And we do X and use Y" without expanding on how and why - but I feel this is one of those problems which even less proficient coders can get the logic of. This question, hence, can serve as a stepping stone for them in their transition to solving harder problems. I made sure that quick explanation suffices for almost all people who do not need the verbose explanation (and hence they can skip parts they feel they know), but for rest its better to have something here.

Related Problems

- **Pudding Monsters**
- **Learning Dishes** - Binary Lifting
- **Batman and Tree**
- **ADDMUL**

Quote

20

saurav1199 **Mar '20**

Thanks a lot for this amazing editorial.

PREREQUISITES:

Centroid decomposition, point-update range-query data structures

PROBLEM:

You're given a tree on N vertices, vertex i has the value A_i.
For two vertices x and y, let $f(x,y)$ denote the minimum value on the path between them, and $g(x,y)$ denote the maximum.
Find $\sum_{x<y} f(x,y) \cdot g(x,y)$.

EXPLANATION:

First, let's try to solve easier versions of this problem.

For a fixed root vertex r, computing $\sum_{x=1}^{N} f(r,x) \cdot g(r,x)$ can be done easily in $\mathrm{O}(N)$ time using a simple DFS to compute the required values.
However, there doesn't seem to be a nice way to quickly recompute this value when changing roots.

Instead, we'll try to do a bit more: we'll try to compute the sum of $f(x,y) \cdot g(x,y)$ for all pairs of vertices (x,y) whose paths *pass through* r. Let's see how we can do this quickly.

First, compute $f(r,x)$ and $g(r,x)$ for all vertices; as mentioned above, this can be done in $\mathrm{O}(N)$ using a DFS.
For convenience, let $B_x = f(r,x)$ and $C_x = g(r,x)$.

Notice that for a path (x,y) that passes through r, it holds that $f(x,y) = \min(B_x, B_y)$ and $g(x,y) = \max(C_x, C_y)$.
So, we essentially want to compute $\sum \min(B_x, B_y) \cdot \max(C_x, C_y)$ across all valid pairs (x,y).

Once again, let's relax the problem a bit, ignore the "valid pairs (x,y)" condition, and just attempt to quickly compute that summation for *all* pairs.

How?

Let's sort the pairs of (B_x, C_x), so that $B_1 \le B_2 \le \ldots B_N$.

Then, for each i from 1 to N:

- Let's only consider $i < j \le N$
- The sorting guarantees that $\min(B_i, B_j) = B_i$, so we only need to deal with C_j values.
- There are two cases here: $C_i \ge C_j$ and $C_i < C_j$.
  - If $\max(C_i, C_j) = C_i$, we add $B_i \cdot C_i$ to the answer. Overall, the answer increases by $k \cdot B_i \cdot C_i$, where k is the number of indices such that $C_i \ge C_j$.
    Processing them all quickly can be done if we can just find k.
  - If $C_i < C_j$, we add $B_i \cdot C_j$ to the answer. The answer increases by $B_i \cdot S$, where S is the sum of all C_j that are larger than C_i.
    Once again, we can process them all quickly if we can quickly compute S.

So, we need to be able to quickly answer the following questions about the suffix from $i+1$:

- How elements are $\le C_i$?
- What's the sum of elements that are $> C_i$?

If i is iterated in decreasing order from N down to 1, both of these can be answered in $\mathrm{O}(\log N)$ by a fenwick/segment tree built on values.

This allows us to compute the entire sum in $\mathrm{O}(N \log N)$.

Now that we've done this, we have some excess that needs to be removed.
In particular, we need to remove the sum of $\min(B_x, B_y) \cdot \max(B_x, B_y)$ for all pairs vertices whose paths don't pass through r.

In fact, such pairs can be found quite easily.
Let's root the tree at r, and let $c_1, c_2, \ldots, c_m$ be its children.

What we need to remove is then the value of all pairs that lie in the subtree of c_1, all pairs in the subtree of c_2, $\ldots$, all pairs in the subtree of c_m.
This is quite easy to do: after all, we already have a way to compute the sum of all pairs of vertices, given the list of their values. So, simply run the above on each of the respective sets, and subtract it from the answer.
This also takes $\mathrm{O}(N \log N)$ time in total, since the sum of the sizes of the sets is $N - 1$.

Now that we've done this, we can move on to a solution to the original problem.

Notice that if we root the tree at some r and compute this value, we are left with several smaller trees (one corresponding to each neighbor of r); which can be solved with the same algorithm.

It's well-known from the centroid decomposition algorithm that choosing r to be the centroid each time leads to a depth of $\mathrm{O}(\log N)$, and at each level of this recursion we do a total of $\mathrm{O}(N \log N)$ work leading to a solution in $\mathrm{O}(N \log^2 N)$ overall.

TIME COMPLEXITY

$\mathrm{O}(N \log^2 N)$ per test case.

PREREQUISITES:

Lazy propagation, segment tree / fenwick tree

PROBLEM:

There are N enemies standing from left to right with strengths $S_1, S_2, \ldots, S_N$.

M bullets are to be fired from the left side such that only one bullet is fired at a time. The ith bullet $(1 \leq i \leq M)$ has power P_i and capacity C_i.
This means that the ith bullet affects the first C_i **alive** enemies from the left side and decreases the strength of each of those C_i enemies by P_i.

Once an enemy's strength becomes less than or equal to 0, the enemy dies and its strength becomes 0.

Find the strength of all the enemies after all M bullets are fired.

EXPLANATION:

To approach any data structure problem, follow the two steps:

1. Find the queries required to solve it.
2. Use the appropriate data structure to answer every query.

We can approach this problem in two ways:

Way 1:

- Query 1: Finding the location of C_i^{th} alive enemy starting from the left - Solve it using a Fenwick array and doing **binary search** on it in $\mathrm{O}(\log N)$ or use segment tree with range sum for doing the same in same complexity. Total time for all queries will be $\mathrm{O}(M \log N)$.
- Query 2: Reducing the power of the first C_i alive enemies by P_i - Solve it using lazy propagation over the segment tree in $\mathrm{O}(\log N)$. Total time for all queries will be $\mathrm{O}(M \log N)$.
- Query 3: Finding the enemies whose energy has turned ≤ 0 and thereby updating the fenwick array used to answer query 1 accordingly - This can be done by having a range minimum segment tree to answer query 2 and going only into those nodes where the minimum value ≤ 0 after the update. The important observation here is that we only end up reaching $\mathrm{O}(N)$ leaves, therefore total time will be $\mathrm{O}(N \log N)$. Make sure to update the strength of the enemy which has been eliminated to `LLONG_MAX`.

Way 2: (Setter's approach)

We will calculate the final strength of an enemy iteratively from left to right. Bullet will hit an enemy until its strength is greater than 0.

Let's say total x bullets hit an enemy i. Therefore the sum of power of all x bullets should be $\geq S_i$.

We can find this x (using segment tree or fenwick tree) in the power array P.

We then decrease the count of all x bullets by one in the count array C.

If at any moment, count of any bullet becomes zero (we can find this using range minimum query) we remove this bullet or simply make its power zero.

Like this we can calculate the strength of all the enemies.

COMPLEXITY ANALYSIS:

$\mathrm{O}((N+M)\log N)$ for first and $\mathrm{O}((N+M)\log M)$ for second approach.

PREREQUISITES:

Stacks, Range minimum/maximum queries, (optionally) Binary search

PROBLEM:

You have an array of length N. You can pick any subarray of size 2 or more of this array, after which Chef removes 2 elements from it. Your score is computed as the sum of the remaining elements.
Maximize your score, given that Chef always chooses elements to minimize your score once a subarray is chosen.

QUICK EXPLANATION:

- Chef always picks the largest and second largest element of the subarray
- Iterate over the position of the second largest element
- Once the second largest is fixed, the largest element is either to its left or its right - try both cases. The positions of these elements can be precomputed with a stack
- Also compute the positions of the second closest larger element to the left/right
- Once all these positions are known, the best answer for a given position can be computed by a couple of range minimum/maximum queries

EXPLANATION:

The very first thing to notice is that once you have chosen a subarray, Chef will always remove the largest two elements from it.
So, our task is to find a subarray such that its sum after removing the largest two elements from it, is maximum.

Of course, this can be done in $\mathrm{O}(N^3)$ by iterating over all subarrays and then finding their maximums, and even optimized to $\mathrm{O}(N^2)$ by smartly keeping track of subarray sums and maximums while iterating. This is too slow for our purposes, and it is not immediately obvious how to optimize it to run any faster.

Let us instead approach the problem differently: suppose we fixed which two elements were being deleted, say A_L and A_R where $L \le R$. Then, the maximum sum of a subarray containing these two points can be broken into 3 parts:

- The sum of elements from positions $L+1$ to $R-1$
- The maximum sum of a subarray ending at $L-1$
- The maximum sum of a subarray starting at $R+1$

However, we are fixing the positions of the maximums - hence, none of the 3 parts can contain any element which is larger than $\min(A_L, A_R)$.

This also leads us to another observation:
Suppose $A_L < A_R$. Then, R must be the smallest index $> L$ such that $A_L < A_R$.
Similarly, if $A_L > A_R$, L must be the largest index $< R$ such that $A_L > A_R$.

Hence it seems useful to know, for each element, what the next/previous greater element is.
It is well known that computing the next/previous greater element can be done for all indices of an array in $\mathrm{O}(N)$ time using a stack: see **this** for more information.

Armed with this information, we note something further: upon fixing the second largest element of the array, the largest element is either going to be the next greater or the previous greater.
In other words, we have reduced the number of objects we are iterating on from $\mathrm{O}(N^2)$ to $\mathrm{O}(N)$. If we are able to calculate the maximum subarray sum for each of these (L, R) pairs in something like $\mathrm{O}(\log N)$, that would solve the problem - which is exactly what we will do.

Recall that the subarray was broken into 3 independent parts. Let us try to solve for each of those separately.

The middle part

Upon fixing L and R, every element between them must be taken, no matter what.
So, this simply comes out to be the sum of the subarray between $[L+1, R-1]$ which is easily calculated by, say, prefix sums.

For the remaining two parts, I assume that $A_L < A_R$ - the other case proceeds similarly.

The left part

We want the largest possible subarray sum ending at $L-1$ such that its elements don't exceed A_L.
Let L_1 be the index of the previous greater element of index L. For the subarray to not have an element exceeding A_L, it must equivalently not contain index L_1.
So, we are looking for the largest sum of a subarray of the form $[i, L-1]$ where $L_1 < i < L$.
In terms of prefix sums, the sum of any such subarray is $pref_{L-1} - pref_{i-1}$ where $pref_i = A_1 + A_2 + \cdots + A_i$ and $pref_0 = 0$.
Note that $pref_{L-1}$ is a constant, so maximizing the sum is equivalent to picking the minimum value of $pref_{i-1}$ for $L_1 < i < L$.
This is the classical **range minimum query** problem which can be answered in a host of ways - for our purposes, we will assume $\mathrm{O}(\log N)$ with the help of a segment tree.

The right part

At first glance, this case is symmetric to the one above - if we know the first index larger than R (say R_1) such that $A_{R_1} > A_L$, the largest subarray sum starting from $R+1$ and ending before R_1 can be found via a segment tree query again (this time, a range maximum query instead).

What if such an index doesn't exist?

There is no problem in such a case - it just means that every subarray starting from $R+1$ can be considered, which is equivalent to saying that $R_1 = n+1$.

However, recall that we have only previously computed the index of the next greater element of R - there is no guarantee that R_1 should also be this index, it might be less.

Example

Consider the array

```
1 3 2 4
```

The next larger element of `1` is `3`, and the next greater element of `3` is `4`. However, the R_1 we are looking for corresponding to `1` is `2`, not `4`.

So, the problem here is how exactly to find R_1 - if we are able to do that, the problem is functionally solved.

It turns out that there are several ways to do this:

Method 1 (Stacks) (Setter's Solution)

Note that what we want to compute is essentially just the second nearest greater element to the left/right.
To do this, It is possible to extend the algorithm used to compute the nearest greater element.
Here is some pseudocode to compute the next greater element:

```
stack s
for i from 1 to n:
    while s is not empty and a[top(s)] < a[i]:
        next_greater[top(s)] = i
```

```
        pop s
    push i onto s
```

The idea here is that `s` contains a (monotonic) sequence of indices whose next greater element hasn't been found yet.
Extending that idea, computing the second next greater element only needs another stack - whenever something is popped from the first stack, it can be pushed into the second, to signify that its next greater element has been found, and whatever pops it from the second stack will be its second next greater element.
However, note that in order to maintain monotonicity of the second stack, we cannot just push values into it as soon as we pop from the first. Instead, at each index, we must store the sequence of elements popped, and then push them in reversed order.

Pseudocode is as follows:

```
stack s1, s2
for i from 1 to n:
    while s2 is not empty and a[top(s2)] < a[i]:
        next_greater_2[top(s2)] = i
        pop s2

    stack rem
    while s1 is not empty and a[top(s1]) < a[i]:
        next_greater[top(s1)] = i
        push top(s1) onto rem
        pop s1

    while rem is not empty:
        push top(rem) onto s2
        pop rem
    push i onto s1
```

This still runs in $\mathrm{O}(N)$ time overall because the cost of pushing/popping amortizes over all indices.
Method 2 (Binary search + RMQ) (Tester's and Editorialist's Solution)

it is possible to binary search on the position of R_1, with the help of (yet again) range maximum queries.
Binary search on the range $[R+1, N+1]$, and:

- Suppose the current range is $[lo, hi]$.
- Query for the largest element in $[lo, mid]$, suppose it is x.
- If $x > A_L$, set $hi = mid$
- Otherwise, set $lo = mid + 1$

At the end of this, lo is the R_1 we are looking for.

The complexity of this part is $\mathrm{O}(\log^2 N)$ assuming a segment tree is used for the range maximum query, which may or may not pass depending on how well the implementation is handled.
However, it is possible to improve the complexity to $\mathrm{O}(\log N)$ by combining the binary search into the query function of the segment tree. This technique is sometimes called implicit binary search/walking down the segment tree.
Implementation details can be found in the comments of **this blog**

Implementation Details

The solution outlined above takes some liberty with details in order to not clutter the main idea. In particular, there are some cases where you would like the next/previous greater element to be equal instead of strictly larger, as in the case of the array

```
1000 -1000000 10 5 5 10 -1000000 1000
```

Here, the best subarray to choose is `10 5 5 10` giving a final answer of 10, which the above solution doesn't quite do because it would always pair a 10 with a 1000 which is suboptimal.

The fix

Note that for a given index, we compute two things: the next/previous greater element, and the second next/previous greater element.
By far the simplest way to fix this problem is to compute the first greater element allowing equality, and the second with a strict inequality.
Making this change is just a matter of changing < to <= (or vice versa, depending on implementation) in the popping criteria.

Why does this work?

Consider a case which looks like:

```
----a------b---x---x--x----c------d-----
```

where $x < a, b, c, d$ and - represents something smaller than x.
Suppose we fix one of the x between b and c to be the second largest element.
Then, there are several possibilities for the subarray which can be chosen:

- The subarray has to contain at least one of b, c or another x; else the one we fix would not be the second maximum
- The subarray cannot contain both b and c at the same time, and cannot contain a or d no matter what.
- If the subarray contains neither b nor c, it is completely contained between them and will contain some 2 consecutive x's. This case will be covered when fixing the first x as the second largest element, and checking for the largest element on the right (since we allowed equality)
- If the subarray contains b, it must contain the leftmost x. This case will be covered when fixing the first x and checking for the largest element on the left
- If the subarray contains c, it must contain the rightmost x. This case will be covered when fixing the last x and checking for the largest element on the right.

Also, make sure to properly handle cases when the something doesn't exist (i.e, when there is no larger/second larger element).

TIME COMPLEXITY:

$\mathrm{O}(N \log N)$ or $\mathrm{O}(N \log^2 N)$ per testcase, depending on implementation.

PROBLEM:

Given the heights and widths of the boxes in a single line
Find Longest Increasing Subsequence from the left end such that the sum of widths of all the boxes chosen is maximum i.e distance of Amit.
Find Longest Non - Decreasing Subsequence from the right end such that the sum of widths of all the boxes chosen is maximum i.e distance of Saurav.
Print the name of Winner and distance traveled by him, in case of a tie print the name of Amit and distance traveled by him.

Prerequisites:

Stronghold on the concept of Weighted LIS, Dynamic Programming, and Segment Tree.
A little amount of brain to be applied while solving or understanding the editorial 😉

EXPLANATION:

The idea is the same as finding the LIS but just to make sure that every time the weights of respective indexes to be updated. This can be implemented in various ways, one is the seg tree.
The ith index tells that if we will choose I as the last element of LIS then we will get a total s[i] score.
Firstly, we will compress the array of heights and build a segment tree on it where the nodes will contain the max value(which will initially be 0). Then we will iterate over the array and will store the number of total elements(id) less than it (by using lower_bound), then we will make a query of 1 to id-1 to get the max value in this range (mx), then our current answer will be max(score[i]+mx, value at that node), which will be updated to our current index in the segment tree.
After N iterations, The max element in the entire range will be our answer.

##Another approach:
Maintain pairs (height[i], i) and sort them by heights, and now maintain a BIT Tree over N elements of the array, where the ith index of the BIT Tree denotes the ith index of the original array.
Now for every index 1<=i<=N, in pairs(described above), update the BIT array at index pair[i].second with value pair[i].first and dp[i] = Prefix sum of the elements of the BIT Tree from 1 to pair[i].second.
The answer will be the max(dp[i]) , 1<=i<=N.

PREREQUISITES:

Segment-tree (**Segment tree | Efficient implementation - GeeksforGeeks**)

PROBLEM:

Given array Ai and 2 queries.
In query 1, find the sum of subarray of the given range.
In query 2, updating the array.

QUICK EXPLANATION:

Brute force approach: Traversing the given range for every query which will have the time complexity O(q*n).
Efficient Approach: Using segment tree concept which will have the time complexity O(q+n)*logn.

EXPLANATION:

Efficient Approach:
First converting the given array to alternative negative terms as given in question that the alternate classroom will have rude professors.
Then building the segment tree.
For query 1: Just calculating the sum of the given range and if the initial index of given range is odd-indexed then the sum must be multiplied by -1 as odd indexed classroom contains rude professors.
For query 2: Simply update the segment tree.

PREREQUISITES:

Euler Tour, Segment Trees, Lazy Propagation

PROBLEM:

Given queries, you have to either increase the values of all nodes in a subtree by some value, or find the number of nodes with an odd value in a subtree.

EXPLANATION:

The first observation we can make is since only the parity of the numbers matters, and not the numbers themselves, we can represent all values using bits, with 1 representing Odd and 0 representing $Even$.

The second observation to be made is, during the Update queries, adding an even number to all values does not change their parity. So update queries with an even value can be ignored.

If the Update query has an odd number, all the bits in the subtree must be flipped.

We can **flatten the tree into an array** and use a **Segment Tree** to flip all bits in a range and also find the number of set bits in any range.

Since the Segment Tree takes $O(logn)$ time per query, the overall time complexity is $O(N + QlogN)$.

PREREQUISITES:

persistent segment tree, sqrt decomposition

(Note: I can't find wikipedia articles for persistent segment tree, but **this** seems like a good tutorial to read. You can also google "persistent segment tree" for this topic. The editorial assumes familiarity with persistent segment tree.)

PROBLEM:

There are n intervals $[l_i, r_i]$ where $1 \le l_i \le r_i \le n$. You have Q queries, where each query is given CNT distinct points $x_1, x_2, \ldots, x_{CNT} (1 \le x_i \le n)$, and you need to output the number of *good* intervals. An interval is *good* if it crosses an odd number of query points. An interval $[l, r]$ cross a point x if $l \le x \le r$.

QUICK EXPLANATION:

For a query of CNT points:

- if $CNT \ge \sqrt{\frac{n}{\log n}}$, then we use an $O(n)$-time brute force: we can count, for each interval $[l_i, r_i]$, the number of points that it crosses, in a total time of $O(n)$. Then we simply report the answer.
- if $CNT < \sqrt{\frac{n}{\log n}}$, then we use the following $O(CNT^2 \log n)$ algorithm: first we sort the points by ascending order, say they are $x_1, x_2, \ldots, x_{CNT}$. We then enumerate the first and the last point x_i, x_j that's crossed by the interval, ensuring that $j - i$ is even(so there are odd points crossed); and look up how many intervals satisfy this condition. For an interval $[l, r]$, this condition is equivalent to $x_{i-1} + 1 \le l \le x_i$ and $x_j \le r \le x_{j+1} - 1$, which can be maintained in a data structure to support $O(\log n)$-time queries.

EXPLANATION:

subtask 1

This subtask can be solved by straightforward brute force in $O(N \cdot sum(CNT)) = O(N^2)$ time.

subtask 2

a special case: $CNT = 1$

In this special case, each query is a point $x \in [1, N]$, and we'd like to find the number of intervals which crosses x. However, we'll consider the following form of query, which will be useful later: given L_1, R_1, L_2, R_2, how many intervals $[l_i, r_i]$ are there such that $L_1 \le l_i \le R_1$ and $L_2 \le r_i \le R_2$?

To answer such query, let's build a persistent segment tree. The x-th root is a segment tree, and for a node which represents $[L, R]$ in this tree, it stores the number of intervals with $l_i \le x$ and $r_i \in [L, R]$. It can be constructed by the following pseudocode:

```
sort the intervals such that l_1<=l_2<=...<=l_n
j = 1
for i = 1 to n
    (root of i-th tree) = (root of (i-1)-th tree)
    while (l_j <= i)
        insert r_j into (root of i-th tree)
        j = j + 1
```

Let $f(a, L_2, R_2)$ be the number of intervals with $l_i \le a$ and $L_2 \le r_i \le R_2$. We can query $f(a, L_2, R_2)$ in $O(\log n)$ time: we simply query the sum of $[L_2, R_2]$ in the a-th tree. Then we can answer the above query (L_1, R_1, L_2, R_2): the answer is simply $f(R_1, L_2, R_2) - f(L_1 - 1, L_2, R_2)$.

the solution

We set a parameter $S = O(\sqrt{\frac{n}{\log n}})$.

If $CNT \le S$, then we sort the points as $x_1 < x_2 < \cdots < x_{CNT}$, and enumerate L, R such that $1 \le L \le R \le CNT$ and $R - L$ is even. We can find the number of intervals such that $[l_i, r_i] \cap \{x_1, \ldots, x_{CNT}\} = \{x_j : j \in [L, R]\}$. This is simply done by a query $(x_{L-1} + 1, x_L, x_R, x_{R+1} - 1)$. This takes $O(CNT^2 \log n) = O(CNT\sqrt{n \log n})$ time.

If $CNT > S$, the query can be answered by brute force: we let $s[i]$ equal to the number of x_j's such that $x_j \le i$, and the number of points $[l_i, r_i]$ crosses is just $s[r_i] - s[l_i - 1]$. This takes $O(n) = O(CNT\sqrt{n \log n})$ time.

The total time complexity is $O(n\sqrt{n \log n})$.

PREREQUISITES:

Subtree updates/queries on a tree; segment trees with lazy propagation

PROBLEM:

You're given a tree with N vertices; the i-th has value v_i.
Support the following updates and queries on it:

1. Given u and p, multiply the values of all values in the subtree of u, by v_i.
2. Given u and p, fully divide the values of all values in the subtree of u, by v.
   Here, fully divide means "while x is divisible by p, divide it by p".
3. Given u, count the number of primes that divide all values in the subtree of u.

it is known that $p, v_i \leq 700$.

EXPLANATION:

The third query suggests that looking at the prime factorizations of the values is helpful.
Looking at the how the updates affect values in terms of their factorizations:

- The first update means "all values in the subtree of u now have p as a prime factor".
- The second update means "all values in the subtree of u no longer have p as a prime factor".

This means the prime factorizations of each element are rather boolean in nature: for each element, we only need to know which primes exist; we don't care about their powers.

Since $p, v_i \leq 700$, there are only 125 primes we care about in total.
So, let's represent the prime factorization of each element by a 125-bit bitmask.
Let M_i denote the mask for vertex i.
This bitmask can be represented using two 64-bit integers in most languages (or, in C++, a `bitset<125>`).

Translating our updates and queries to this bitmask:

- Update 1 means "set a certain bit to 1 for all masks in the subtree of u".
  This can be represented by a bitwise OR operation.
- Update 2 means "set a certain bit to 0 for all masks in the subtree of u".
  This can be represented by a bitwise AND operation.
- Update 3 means "find the number of bits that are set in all masks in the subtree of u".
  This can be thought of as "find the bitwise AND of all masks in the subtree of u, then count the number of set bits".

We now need to perform these efficiently.

First off, as always we can transform subtree operations into subarray operations by flattening the tree via an Euler tour.
Now, we need to support the following. We have an array M of length N. Then,

- Given (L, R, x), set $M_i \to M_i \mid x$ for all $L \leq x \leq R$.
- Given (L, R, x), set $M_i \to M_i \ \& \ x$ for all $L \leq x \leq R$.
- Given (L, R), compute the bitcount of $(M_L \ \& \ M_{L+1} \ \& \ \ldots \ \& \ M_R)$.

Since we're performing lazy propagation, we need to be able to combine updates; and this is somewhat non-trivial because there are two completely different types of updates.

However, it is indeed possible to do so.
For each bit, there are three possible updates to it:

- Keep it the same
- Set it to 1
- Set it to 0

So, each update is a *ternary* state.
Combining these ternary states, however, can be done using just binary operations as follows:

- Each update type consists of two bitmasks x and y.
  By default, x has all its bits set and $y = 0$.
- Then, to push an update (x_1, y_1) from a parent node in the segment tree to its child whose update is currently (x_2, y_2), do the following:
  - Set $x_2 \to (x_2 \ \& \ x_1) \mid y_1$
  - Set $y_2 \to (y_2 \ \& \ x_1) \mid y_1$

This process of performing the AND update before the OR one is always correct, and can be verified manually against all 3×3 possibilities of combining updates for a single bit.

Since we're using purely bitwise operations, this is now just standard lazy propagation on a segment tree whose time complexity is $\mathrm{O}(Q \log N)$, which is fast enough.

TIME COMPLEXITY

$\mathrm{O}(N + Q \log N)$ per testcase.

PREREQUISITES:

Segment Tree, **Segment Tree Beats**

PROBLEM:

Chef is preparing a banquet of N dishes for Sir Anton and Sir Trygub. Unfortuantely, Sir Anton and Sir Trygub only told the Chef three days before the banquet that they do not like food that is too spicy or too numb.

For the i-th dish, Chef has assigned 4 scores:

- SA_i — the spiciness felt by Sir Anton
- NA_i — the numbness felt by Sir Anton
- ST_i — the spiciness felt by Sir Trygub
- NT_i — the numbness felt by Sir Trygub

Find the minimum value of $\max\limits_{i \in A}(SA_i) + \max\limits_{i \in A}(NA_i) + \max\limits_{i \in T}(ST_i) + \max\limits_{i \in T}(NT_i)$ over all (A, T), such that, $A \subseteq [1, N]$,$T \subseteq [1, N]$, $1 \leq |A|, |T|$ and $A \cup T = [1, N]$.

EXPLANATION:

We perform the following strategy:

- Fix the values of $\max\limits_{i \in A}(SA_i)$ and $\max\limits_{i \in A}(NA_i)$. Denote the fixed values as MSA and MNA respectively.
- For all dishes j such that $SA_j \leq MSA$ and $NA_j \leq MNA$, put them into A.
- The remaining dishes are put into T.

This strategy leads to an obvious $O(N^3)$ algorithm:

- Loop over MSA.
- Loop over MSB.
- Construct the set T according to the previous strategy.
- Calculate $\max\limits_{i \in T}(ST_i)$ and $\max\limits_{i \in T}(NT_i)$.

To speed up this process, we use the following strategy:

- Let MNA be the index of some data structure D, where D_{MNA} stores $MNA + \max\limits_{i \in T}(ST_i) + \max\limits_{i \in T}(NT_i)$, where T is the set of dishes j that have $SA_j > MSA$ and $NA_j > MNA$.
- Sweep line over decreasing MSA.
  - When we reach some dish i such that $SA_i = MSA$, update the data structure D for indices from $NA_i + 1$ to ∞. Specifically, we need to update the values of $\max\limits_{i \in T}(ST_i)$ and $\max\limits_{i \in T}(NT_i)$ for these indices.
  - Then we query the minimum value of D.

We can see that this data structure needs to support maximizing range update and max range query. There are multiple ways to do this:

- Apply direct segment tree beats.
- Precompute using stacks to transform each maximizing range update to many addition range updates (this strategy is used in the setter's solution).

TIME COMPLEXITY:

Time complexity is $O(N \log^2 N)$ per test case.

PREREQUISITES:

Two-Pointer and Segment Tree with Lazy Propagation.

PROBLEM:

There are N chefs and M types of dishes, ith chef can cook a dish of type F_i also each chef has K friends. When some chef cooks a dish, he can ask his friends or the chefs who cook the same type to evaluate the dish. The chef cannot evaluate his own dish. You have to count the number of pairs L and R and allow the chefs $L, L+1, \ldots, R$ to cook their dishes and have them evaluated as described above each chef from 1 to N evaluates at least one dish.

QUICK EXPLANATION:

- Suppose if range $[L, R]$ satisfies the condition then all the ranges which contain this range will satisfy the condition. i.e. all the ranges $[X, Y]$ such that $X \leq L$ and $Y \geq R$.
- So will use the Two-Pointer technique to compute the number of segments by fixing the left border and finding the shortest right border such that segment $[L, R]$ satisfies the condition. And we will add $(N - R + 1)$ to our answer.
- We will create a modified array such that all the chefs with dish type 1 are put first, chefs with dish type 2 are put next, chefs with dish type 3 are put next, and so on. In this way, we can update the chefs with the same dish type in $O(\log N)$ as they form a continuous segment using segment tree with lazy propagation.

EXPLANATION:

Let us propose a claim. Suppose if range $[L, R]$ satisfies the condition then all the ranges which contain this range will satisfy the condition. i.e. all the ranges $[X, Y]$ such that $X \leq L$ and $Y \geq R$.(think why?)

Now first try to do this problem with two-pointer in an inefficient way. To count the number of segments we can follow the strategy as :

- Maintain a frequency array which tells the number of times a chef i has been invited. Initially, all values are zero.
- Maintain a $count$ variable which tells the number of which has been invited and it is equal to 0 initially as no chef has been invited yet.
- Let us fix the left border. Suppose for each i where i is the left pointer we count the minimum R, where R is the right pointer such that segment $[i, R]$ satisfies the condition. Then we can add $(N - R + 1)$ ranges to our answer(it is the number of ranges starting at i and ending at index $\geq R$).
- Initially let the right pointer is at 0.
- Iterate over the left border of the range. Suppose we start at index 1. we will add all the chef's friends and those with the same dish type to the frequency array(except chef itself as he cannot evaluate his own dish). `We will increase the count variable only when the value or frequency for some chef changes from 0 to 1` then we increase the right pointer until right pointer hits N or the value of $count$ variable becomes N. Suppose it becomes N when the right pointer is equal to j then we will add $(N - j + 1)$ to our answer. Now when we shift our left border from i to $i + 1$ we will subtract all his friends and those with the same dish type as chef i `We will decrease the count variable only when the value or frequency for some chef changes from 1 to 0. Also, note you will not add the friends and other chefs for the chef i+1 until your right pointer > left pointer because we have already invited his friends and chef with dish type same as dish type of chef i+1 as we have moved our right pointer.`
- In this way we can compute the answer.

Now the only problem in this approach is that the number of chefs with the same dish type as the current chef can be very large which leads it to $O(N^2)$ solution in the worst case. So we need an efficient way to update and

check whether all chefs have been invited at least once or not. For this, we will use Segment Tree with Lazy Propagation.

We will create a modified array such that all the chefs with dish type 1 are put first, chefs with dish type 2 are put next, chefs with dish type 3 are put next, and so on. In this way, we can update the chefs with the same dish type in $O(\log N)$ as they form a continuous segment, and as the number of friends which (might have different dish type) is ≤ 3, so we can do an update for each of them individually. `(NOTE! suppose you update the range of dish type for the chef i, then you have to update again for chef i to subtract the effect of the "range dish update" as it is mentioned in the question that chef cannot evaluate his own dish` . So basically the indication`(depends on the implementation of segment tree)` that all chefs have been invited at least once can be done by querying over the whole range. This modification basically converts our previous $O(N^2)$ solution to $O(N * \log N)$ solution.

TIME COMPLEXITY:

- Both Left and Right Pointer changes $O(N)$ time and each chef i, adding and deleting his friends and other chefs with the same dish type requires $O(\log N)$.
- So total time complexity per test case is $O(N * \log N)$.

PREREQUISITES:

Fenwick Trees (Binary Indexed Trees), Segment Trees, Range Updates

PROBLEM:

Given an array A consisting of size N. Process Q queries of the following types:

$1\ L\ R\ X$ - Add $(X+i-L)^2$ to A_i, for all i in $[L,R]$

$2\ i$ - Output A_i for $i \in [1,N]$

QUICK EXPLANATION:

The given queries can be easily processed with binary indexed tress or segment trees.

EXPLANATION:

Processing type 1 queries

Let's look at the first query:
$1\ L\ R\ X$ - Add $(X+i-L)^2$ to A_i, for all i in $[L,R]$
Here we are adding $(X+i-L)^2$. Let $y = X-L$. Now lets expand this term:
$(X+i-L)^2 = (X-L+i)^2$
$= (y+i)^2$
$= y^2 + 2*y*i + i^2$

Now lets say we performed two queries of type 1 with $y = y_1$ and $y = y_2$ respectively. Let's see what value we added to A_i after the two queries. We add following to A_i.
$(y_1^2 + 2*y_1*i + i^2) + (y_2^2 + 2*y_2*i + i^2)$
$= (y_1^2 + y_2^2) + (2*y_1 + 2*y_2)*i + (1+1)*i^2$

Similarly after we perform k queries of type 1, A_i will be increased by:
$(y_1^2 + y_2^2 + ... + y_k^2) + (2*y_1 + 2*y_2 + ... + 2*y_k)*i + (1+1+...+1(k\ times))*i^2$

If we can maintain following sums $\sum y^2$, $\sum 2*y$ and $\sum 1$ for each i then we can easily answer what would be the value of A_i after performing these queries. These values can be maintained by using three binary indexed trees or segment trees. Let the data structures B_1 maintain the sum $\sum y^2$, B_2 maintain the sum $\sum 2*y$ and B_3 maintain the sum $\sum 1$.

Now we can break the query of type 1 in following steps:

- Add y^2 to the range $[L,R]$ in B_1.
- Add $2*y$ to the range $[L,R]$ in B_2.
- Add 1 to the range $[L,R]$ in B_3.

These are simple update queries which can be performed in $O(logn)$ time in both segment trees and binary indexed trees.

Answering type 2 queries

As we saw in the previous section, after k queries of type 1, A_i will be increased by:
$(y_1^2 + y_2^2 + ... + y_k^2) + (2*y_1 + 2*y_2 + ... + 2*y_k)*i + (1+1+...+1(k\ times))*i^2$

$= \sum y^2 + i * \sum 2 * y + i^2 * \sum 1$

Let the point query $q_1(i)$ return the value at index i in B_1, $q_2(i)$ return the value at index i in B_2 and $q_3(i)$ return the value at index i in B_3. These are simple get queries for the value at index i which can be easily processed in $O(logn)$ time in both segment trees and binary indexed trees.

Now the value added at index i would be:
$q_1(i) + i * q_2(i) + i^2 * q_3(i)$

So answer to the query 2 i would be $A_i + q_1(i) + i * q_2(i) + i^2 * q_3(i)$.

Algorithm

- Keep 3 fenwick arrays or segment trees B_1, B_2, B_3.
- For queries of type 1
  - Now let $y = X - L$.
  - For B_1 do a range update: add y^2 for all i from L to R.
  - For B_2 do a range update: add $2 * y$ for all i from L to R.
  - For B_3 do a range update: add 1 for all i from L to R.
- For queries of type 2
  - Let point query in i^{th} DS on index j be represented by $q_i(j)$
  - Answer to the query is $A_i + q_1(i) + i * q_2(i) + i^2 * q_3(i)$.

TIME COMPLEXITY:

$O(QlogN)$ for processing Q queries as each query takes $O(logn)$ time.

PREREQUISITES:

BIT , Binary search , Coordinate compression , Two pointers , Inversion count .

PROBLEM:

Given an array A, let array B be composed of the inversion counts of every subarray of A. Find the median of array B. In case of multiple medians, output the left one.

HINTS:

Not the full solution. Just some hints to help you if you are stuck at some point. As soon as you encounter a hint that you had not thought of, try solving the problem on your own.

Hint 1:

There can be $O(N^2)$ inversion counts over all the subarrays. We need to find which one of them is the median. The median will be the one for which there are $\frac{(M+1)}{2} - 1$ subarrays which have an inversion count less than or equal to it, where M is the size of B.

As there can be $O(N^2)$ possible candidates for this, we can perform a binary search on them to find the median.

Hint 2:

For our current binary search query, we'll need to find how many subarrays have an inversion count less than or equal to it.

Try to think in the direction of: how many subarrays starting at an index i will satisfy this.

Hint 3:

All subarrays starting at i and ending at or before a j which satisfy this, only if the inversion count of $[i,j]$ is less than or equal to our query, and of $[i,j+1]$ is greater than our query number.

There will be $(j-i+1)$ subarrays starting at i which satisfy this. Add this to the count of subarrays, move on to the next i, and try to extend j for it, in a sliding window fashion.

But how will be find the inversion count for a window efficiently?

Hint 4:

We can use a BIT for this. As $A[i]$ can be as high as 10^9, we can use coordinate compression. Refer to this to understand how inversions can be counted for an array using a BIT.

When we extend the window, we query the BIT for the inversion count for this index, and then update the BIT for this index. When we move on to the next i, we remove the contribution of $A[i]$ from our inversion count, and decrement its value in the BIT.

Try to think about the reason for this, and try to implement the solution on your own if you have understood it.

QUICK EXPLANATION:

show

Perform a binary search on all possible inversion counts and find the smallest one for which the number of subarrays with inversion counts less than or equal to it is $\geq \frac{(M+1)}{2} - 1$, where M is the size of B. To do this efficiently, find the length of the largest subarray starting at an index i, such that its inversion count is less than or equal to our query number, and add this length to our subarray count. Do this for every i, efficiently using sliding window technique with a Binary-Indexed Tree.

EXPLANATION

show

A **squad** consists of one or more consecutive soldiers in the array.
$\implies$ A squad refers to a **subarray** of the array.

The **strength** of a squad is the sum of the number of soldiers to the left of the current soldier with a greater height than him, for every soldier.
$\implies$ Strength of a squad is the **inversion count** of that subarray.

Now, let's rephrase the problem to:

Given an array A, let array B be composed of the inversion counts of every subarray of A. Find the median of array B. In case of multiple medians, output the left one.

Finding the median shall require us to possess the array B. B will have the inversion count of every subarray of A. Since there are $N * (N+1)/2$ subarrays of an array of size N, i.e., $O(N^2)$ subarrays, we can't do this. We will have to find a better way.

Let's take sample example 3 to better visualise:

$N = 4$
$A = \{4, 3, 2, 1\}$
$B = \{0, 0, 0, 0, 1, 1, 1, 3, 3, 6\}$ (in sorted order).

B is made up of inversion counts of the subarrays of A. **Inversion count of any subarray of A will always be in the range $[0, N * (N-1)/2]$.**

why

Inversion count will be 0 when the subarray is sorted in ascending order.

Inversion count will $N * (N-1)/2$ when the largest subarray, i.e., the whole array is sorted in descending order.

Other inversion counts between 0 and $N * (N-1)/2$ will depend on the relative order of the elements in the subarrays.

Median of B is its middle element when it is sorted. Let the size of B be M.
$\implies$ The median will be the $\frac{(M+1)}{2}$ th element of B
$\implies$ There should be $\frac{(M+1)}{2} - 1$ elements in B before the median.
$\implies$ $\frac{(M+1)}{2} - 1$ subarrays of A should have an inversion count $\leq$ the median. $\ldots (1)$

(Here we are considering 1 -based indexing of the arrays. Also, division here implies integer division, i.e., considering the floor value of the actual division. Note that the observations above related to the median will work when size of B is odd, as well as when the size of B is even.)

From (1), we can see that our median will be an inversion count in B such that exactly $\frac{(M+1)}{2} - 1$ subarrays of A have an inversion count less than or equal to it. Thus, we can check this condition to determine if a particular inversion count is the median or not.

But there can be $O(N^2)$ inversion counts. How to check all of them?

We don't need to check all of them. Instead, we can **perform a binary search on the inversion counts**, and find the smallest inversion count which satisfies: The number of subarrays which have an inversion count less than or equal to this number is $\geq \frac{(M+1)}{2} - 1$. Since all numbers in B will satisfy this after a certain point, the smallest of them will be the median.

Now, we know that we need to perform a binary search on inversion counts to obtain the median. But how to check the condition? How to find the number of subarrays of A which have an inversion count less than or equal to a particular number?

To do this, we shall use a **Binary-Indexed Tree** or a BIT, coupled with the **two-pointers** (also known as sliding window) technique. Instead of a BIT, we can also use a segment tree, but BIT takes less space, and is easier to implement.

As the elements in A can be as large as 10^9, and we are only interested in the relative order of the numbers, we shall perform a **coordinate-compression** on them to make it possible to operate on them using a BIT. Before proceeding further, make sure to learn how to find the **inversion-count** of an array using a BIT.

We shall now be learning how to apply the two-pointer technique with a BIT to solve this problem.

Since we want to find the number of subarrays with an inversion count less than or equal to a given inversion count, let's iterate over the starting positions of the subarray.

For every i **we shall be considering all subarrays that start at** i, and end at j, where $i \leq j \leq N$. Start with the subarray (i, i). Increment the compressed position of $A[i]$ by 1 in the BIT. Now find how many elements are greater than $A[i]$ in our subarray, by finding the sum of values in all positions greater than our compressed position in the BIT.

Keep extending j as long as the inversion count remains less than or equal to our query inversion count. When j can no longer be extended without obtaining an inversion count greater than our query number, we know that all subarrays that start at i, and end at or before j satisfy the condition that their inversion counts are less than or equal to our query number. Thus, the number of subarrays starting at i which satisfy this condition is:

- $j - i + 1$

We can add this to our subarray count, keep j as it is and proceed to the next i.

Before proceeding to the next i, we will have to remove the contribution of $A[i]$ from our inversion count. $A[i]$ impacted every number to the right of it in our window, which is smaller than it. To obtain the contribution, we can get the sum of values all positions in the BIT which are less than the compressed position of $A[i]$ and subtract this count from our current sliding inversion count. Moreover, as $A[i]$ will be removed from our current window, we will have decrement its value in the BIT by 1. After this, we can proceed to the next i.

Thus, by using a BIT and two-pointers technique, we can calculate the number of subarrays which have an inversion count less than or equal to query number, and using binary search, find the smallest one which satisfies the condition described earlier. Thus, we'll obtain our required median.

So, to solve the problem:

1. Perform a binary search on all possible inversion counts, and find the smallest one which satisfies the condition: the number of subarrays of A which have an inversion count less than or equal to this count is $\geq \frac{(M+1)}{2} - 1$, where M is the size of B.
2. To check the condition, use two-pointer technique with a BIT. For every starting position i of the subarrays, find the highest position j such that the inversion count of the subarray (i,j) is less than or equal to the query inversion count. Add $(j - i + 1)$ as the count of subarrays which begin at i and satisfy the condition.
3. Move on to the next starting position of subarrays, i.e., the next i, while keeping j same as its previous value. Don't forget to remove the contribution of the previous i from our window before moving on to the next i.

TIME COMPLEXITY:

show

The time complexity is: $O(N * log^2 N)$

Why?

There can be $O(N^2)$ possible inversion counts. We are performing a binary search on them. But $log(N^2) = 2 * logN$. Thus, there is a $logN$ factor.

For every search query, we are using the two-pointers technique to iterate over the whole array, and for each position, we are perfoming some updates, or asking some queries on the BIT. Each update/query takes $O(logN)$, and there are N indices. This adds an $NlogN$ factor.

Thus, the final complexity is $O(N * log^2 N)$.

It takes $O(N)$ space for the BIT as well as for A.

SOLUTIONS:

Setter
Tester
Editorialist

Feel free to share your approach if it differs. **You can ask your doubts below. Please let me know if something's unclear.** I would LOVE to hear suggestions 🙂

Quote

---

myselfsk **May '20**

Can anyone post the solution in python?

---

rahul_g **May '20**

You set complete tree to zero in each test case, that makes your complexity $\mathrm{O}(T \cdot maxn \cdot log^2 n)$.

---

wild4stone **May '20**

can anyone explain why is this giving TLE :
[CodeChef: Practical coding for everyone]

The solution described above is O(N*logN^2) and same as per the editorial

---

sayan_1999 **May '20**

@akashbhalotia Sir,I would really like to know how you yourself came up with the solution within the tight time limits of the contest?

---

sanjay_thiru **May '20**

This is an amazing question. Learnt quite a lot. I have commented the tester's solution. Just posting here so that it will be useful for someone. If there is anything wrong in my understanding, kindly please point out.

https://www.codechef.com/viewsolution/33353455

---

rishup_nitdgp **May '20**

Your implementation looks ok to me. I think…TLE is due to recursive segment tree.

---

sayan_1999 **Jul '20**

@rishup_nitdgp if you can kindly answer. What if using the binary search, we land upon a value that satisfies the criteria but that actually does not occur in the array B? How do you ensure that will never happen?
Thanks in advance…

PROBLEM:

Given an array arr consisting of not necessarily distinct numbers and 2 types of queries:

- to update an element at a given index
- to find greater element nearest to given index in its right such that it has not been appeared in the left of given index(i).

Prerequisites:

- knowledge of segment tree and its applications.
- A little amount of brain to be applied while solving or understanding the editorial 😉

Hint:

1 Point

A simple brute force logic will satisfy the given constraints.

99 Points

Try to construct a logic by using segment tree and with the help of some basic container like set.

QUICK EXPLANATION:

It is a kind of a good implementation of segment tree in a given range to find next greater element where the range will from $[i\ ,\ n-1]$, where we can construct a segment tree of max values and can traverse through the left branch of segment tree to find the greater element and if not present we can check for right branch at that time and then continue to explore rest of the left branches. To take care that next greater element should not repeat before i , we have to make a record of those index and then to build segment tree after these records to ensure correct answer.

EXPLANATION:

First of all I would like to share an approach for finding greater element nearest to given index in right direction (here for some time lets ignore the condition that our answer should not repeat before i).

Now , lets say the array on which we are doing segment tree operations be Arr (initial array)for this we should construct a binary tree of max numbers on this array.For finding next greater element we should start exploring left branches of tree (as they are nearest to $Arr[i]$) by checking the $condition$ that the respective max element is greater than $Arr[i]$ or not, if not at that time we will check our right branch of tree and as soon as our $condition$ (the respective max element is greater than given element or not) satisfies we will continue to explore our rest of the left branches till the leaf node occurs.
By this approach we can easily get the greater element(>$Arr[i]$) nearest to i in $[i\ ,\ n-1]$, but now we are not ensured that our answer is correct or not (for this we have to ensure that it should be not occur in $[0\ ,\ i]$).

For this lets make a new array crr where $crr[j]$=$arr[j]$ if $arr[j]! = arr[k]$ where k is in the range $[0, j-1]$ for every j in the range $[0, i-1]$ (because only unique elements will contribute in crr to avoid duplicacy of elements).
By doing this ,we should replace crr with arr and perform the rest same segment operations to find obtain our correct answer.

Now all concern is about update query. We have to manage update query in such a way that crr should be updated and then by this we can perform some update operations on segment tree.
For this we have to maintain a record of indexes of all same type of elements (which I have done by a $map < long, set < ll >$ in my solution in cpp). where $mp[arr[j]]$ will contain all indexes h such that $arr[h] = arr[j]$ for

h in the range $[0, n-1]$.
You will be wondering why am I doing this 😉 ,this will help us in our update query.

Now see during an update (lets say for $arr[b] = c$) here we have to first find $arr[b]$ and to check whether b was the first index where arr[b] was appeared ,if it is then $crr[b] = -1$ and to find next occurence of $arr[b]$ (lets say $b^{'}$), then $crr[b] = arr[b]$.

After this we will update $arr[b]$ to c and insert b to $mp[arr[b]]$ and now again check whether b was the first index where arr[b] was appeared ,if it is then $crr[b] = arr[b]$ and to find next occurence of $arr[b]$ (lets say $b^{'}$), then $crr[b] = -1$.

By this we can see that at max 4 updations are to be done crr so,for this we can call our update function of segment tree at max 4 times in a single query .

Hence by all this we can ensure that our answer will always be correct.As the complexity of our code is $O(nlogn + 4logn + logn)$ = O(nlogn) approximately, which will satisfy the required constraints.

Sorry for the long editorial,but I hope the solution had cleared your doubts 😉 , but if there is any confusion or query regarding the editorial or approach feel free to ping me 👍!!

Prerequisites

Tree flattening using DFS, Segment Tree, Trie, Implementation

Problem

Given a tree rooted at node 1 whose every node has a value associated with it, implement a technique to support the following types of operations:

- Type 1: Given two nodes a and b and a value x, apply bitwise OR operation on the values of the nodes in the simple path from a to b.
- Type 2: Given a node u and a value y, find the maximum value which can be obtained by applying bitwise XOR operation on any of the values of a node which lies in the subtree of u.

Quick Explanation

Flatten the tree using a **dfs traversal** . After this, the subtree of any node can be represented by a contiguous range on the flattened tree.

It is important to note that the overall number of bit flips due to the update operation can never exceed $O(N * 17)$ [17 = Bit length of the values] because bitwise OR operation can only set a bit if it wasn't set already. This observation is important because for every update, we can now iterate over only those values which will get affected.

Finding the maximum possible XOR value is a rather standard problem and can be solved by maintaining tries. You can study and practice the technique at **link1** and **link2**

Explanation

Flatten the tree using dfs traversal as explained above. After flattening, every subtree can be represented by a single contiguous range. Then, build a segment tree on the flattened tree.

Going to the easy part first. Let's handle the query operations. Every query operation requires us to find the answer in a contiguous range. To solve this, maintain a **trie** at each node of the segment tree. The trie should contain all the values present in the covered range (simply think of a **merge sort tree** . Instead of the linear arrays at each node, maintain tries to store the values). On average, queries on segment trees involves $O(lgN)$ nodes and each node (trie) will take $O(17)$ time to compute the maximum XOR value. Therefore, the time complexity of each query operation is $O(lgN * 17)$.

Next, comes the update part. As proved earlier, we only need to find the indexes (or tree nodes) which will be affected by the particular update operation. We can then naively iterate over all those nodes, and update them in the required tries. To find the affected values, we can maintain 17 separate DSU (**Disjoint set union**) type data structures. The i-th index of the j-th DSU will point to the nearest ancestor of node i whose j-th bit is 0. As the values of the nodes get updated, update the DSU structure accordingly. It can be updated efficiently using path compression technique. Refer to the solution(s) given below for implementation details. Coming to the time complexity, for each update, there are $O(lgN)$ tries to modify (point update complexity in Segment Tree), each trie takes $O(17)$ time to update (Height of each trie). There are $O(N * 17)$ sucessful update operations overall. Therefore, the overall time complexity comes to $(N * 17 * lgN * 17)$.

Time Complexity

The overall time complexity of the solution is intended to be $O(Q * lgN * 17 + N * 17 * lgN * 17)$.

PREREQUISITES:

Persistent segment tree, polynomial hashes

PROBLEM:

You are given array of n integers and q queries. For each query you have to check if sorted elements from $[a;b]$ and $[c;d]$ have at most one mismatch on pairwise comparison.

QUICK EXPLANATION:

Use persistent segment tree to maintain hashes of elements which occur on each prefix. Then check if there is exactly one mismatch of hash values.

EXPLANATION:

In this particular problem we want to check similarity of sets of numbers so it is good idea to mark set of numbers with polynomial hash $\sum_i x^{a_i}$. Since we want to check equality of hashes from subsegments, it would be useful to use well-known technique used in k^{th} order statistics or order of key on segment queries. We maintain persistent segment tree in such way that tree built over first m elements of array contains number $occ_i x^i$ in i^{th} leaf. Where occ_i is the number of occurences of number i on m^{th} prefix. Now when we maintained such segment tree, we have to check whether occ_i for $[a;b]$ and $[c;d]$ differ in either zero or two consequent positions:

```
/*
Checks if occ[l..r] for [a;b] matches with occ[l..r] for [c;d].
mod = 0: two consequent mismatches allowed
mod = 1: single mismatch on suffix allowed
mod = 2: single mismatch on prefix allowed
va, vb = vertices of occ[l;r] for (a;b]
vc, vd = vertices of occ[l;r] for (c;d]
*/
bool check(int va, int vb, int vc, int vd, int mod = 0, int l = 0, int r = maxn)
{
    if(r - l == 1) // single mismatch allowed
        return abs((occ[vb] - occ[va]) - (occ[vd] - occ[vc])) <= 1;
    int m = (l + r) / 2;
    bool ch1 = (hash[L[vb]] - hash[L[va]]) == (hash[L[vd]] - hash[L[vc]]); // is full matc
    bool ch2 = (hash[R[vb]] - hash[R[va]]) == (hash[R[vd]] - hash[R[vc]]); // is full matc
    bool any1 = (hash[L[vb]] - hash[L[va]]) || (hash[L[vd]] - hash[L[vc]]); // any non-nul
    bool any2 = (hash[R[vb]] - hash[R[va]]) || (hash[R[vd]] - hash[R[vc]]); // any non-nul
    if(mod == 0)
    {
        if(ch1 == 0 && ch2 == 0)
            return check(L[va], L[vb], L[vc], L[vd], 1, l, m) &&
                   check(L[va], L[vb], L[vc], L[vd], 2, m, r);
        else if(ch1 == 0)
            return check(L[va], L[vb], L[vc], L[vd], 0, l, m);
        else
            return check(R[va], R[vb], R[vc], R[vd], 0, m, r);
    }
    else if(mod == 1)
    {
        if(ch1 == 0 && any2)
            return 0;
        else if(any2)
```

```
            return check(R[va], R[vb], R[vc], R[vd], 1, m, r);
        else
            return check(L[va], L[vb], L[vc], L[vd], 1, l, m);
    }
    else
    {
        if(ch2 == 0 && any1)
            return 0;
        else if(any1)
            return check(L[va], L[vb], L[vc], L[vd], 2, l, m);
        else
            return check(R[va], R[vb], R[vc], R[vd], 2, m, r);
    }
}
```

As you can see, the code above finds both mismatch positions in $O(\log C)$.

The other elegant idea to simplify the solution is to maintain sum of elements and sum of squared elements on prefixes. Then, assuming that arrays differ in numbers p and q, we can get $p - q$ and $p^2 - q^2$ which will give us an opportunity to find p and q and then simply check by hashes that corresponding segments are equal if we disregard p and q.

PREREQUISITES

Suffix array, LCP array, Segment tree or Fenwick tree.

PROBLEM

Given a string S, a substring of string S is said to be unique if it has a number of occurrence in S exactly equal to 1, Given queries of the form
$L\ R\ K$, find the lexicographically smallest substring $S[x, y]$ of length at least K such that $L \le x \le y \le R$.

EXPLANATION:

Let $\forall\ i \ni 1 \le i \le N$, p_i be the minimum index such that $S[i, p_i]$ is a unique substring. Then all the substrings $S[i, p_i + 1]$, $S[i, p_i + 2]$, $S[i, p_i + 3]$, … , $S[i, N]$, must be a unique substring.

We require Suffix array and LCP array to find p_i. **Here** is a nice video tutorial with practice problems.
Let SA be the suffix array of string S.
Let $\forall\ i \ni 1 \le i \le N$,
l_i = $max(LCP(SA[i-1], SA[i]), LCP(SA[i], SA[i+1])$ …(1)
If l_i is not equal to the length of ith suffix, $p_{SA[i]}$ = $SA[i] + l_i$, else $p_{SA[i]} = INF$
Note: For index 1 and N, we actually only take the LCP's with index 2 and $N-1$ respectively.

We'll be discussing a few lemmas after which you can convince yourself with the above claim:

Lemma 1: The substring $S[SA[i], SA[i] + l_i]$ is a unique substring if l_i is not equal to the length of ith suffix.
Proof: SA denote the suffixes sorted in lexicographical order,
i.e. $S[SA[i-1], N] < S[SA[i], N] < S[SA[i+1], N]$. We prove the above lemma by contradiction.
Let $S[SA[i], SA[i] + l_i]$ be not a unique unique substring, i.e. there exist another substring of length $l_i + 1$ such that it is equal to $S[SA[i], SA[i] + l_i]$. Since $S[SA[i], N] < S[SA[i+1], N]$,
$l_i + 1 \le LCP(SA[i], SA[i+1])$, which is clearly a contradiction since from relation (1), $LCP(SA[i], SA[i+1]) \le l_i$.

Lemma 2: There does not exist $x_i < l_i$, such that $S[SA[i], SA[i] + x_i]$ is a unique substring.
Proof: Let $x_i = l_i - 1$ and $S[SA[i], SA[i] + l_i - 1]$ be a unique substring, i.e. there does not exist another substring of length l_i such that it is equal to $S[SA[i], SA[i] + l_i - 1]$.
But from relation (1), atleast one of $LCP(SA[i-1], SA[i])$, $LCP(SA[i], SA[i+1])$ must be l_i, which clearly denotes that there exists another substring of length l_i which is equal to $S[SA[i], SA[i] + l_i - 1]$ so its a contradiction again.

Let ord_i be the inverse SA_i at each index. It can be calculated simply as:

```
for i in [1, N]:
    ord[SA[i]] = i
```

Let's consider a list of pairs seg such that seg_i = (p_i, i). Sort these pairs in non-decreasing order of p_i. We also require **min**-segment tree or fenwick tree.
Now let's do a sweep-line for i in $[1, N]$. After sorting seg, let (x, y) pair denote seg_j. For all such segments such that $seg_j.x \le i$, update segtree at index $seg_j.y$ with $ord[seg_j.y]$. For all queries (L, R, K) such that $R = i$, query for minimum in segment tree in the range $[L, R - K + 1]$, since we require the substring to be of length at least K.

Why querying minimum in the range $[L, R - K + 1]$ works ?
Since the substrings are unique now, each of them can be uniquely determined and compared based on ord_i

values.

The pseudo code for this part:

```
Let st = min_segment_tree
j = 0
for i in [1, N]:
    while(j < N and seg[j].x <= i):
        st.update(seg[j].y, ord[seg[j].y])
        j++
    for query such that query.R = i:
        mn = st.min_query(query.L, query.R - query.K + 1)
        if(mn != INF):
            ans[query.id] = {SA[mn], max(SA[mn] + query.K - 1, p[SA[mn]])}
        else:
            ans[query.id] = {-1, -1}
```

SOLUTIONS:

PREREQUISITES:

Segment tree,bitset

PROBLEM:

Finding no of integers with odd frequency in a given range and updating values at given indices.

OBSERVATION 1:

Tap to view

All values in the array are less than or equal to 2000.

OBSERVATION 2:

Tap to view

We don't need the exact count of how many times an element occurs. We only need to know the parity of the frequency of each element. This allows us to think on the lines of a boolean array that can be used to store the parity of all frequencies. But then we will have to traverse the boleen array for each query this gives us a time complexity of O(Q*2000) for answering the queries. Solution with this complexity is not intended to pass. We can get better performance if we use bitsets instead of a boolean array. Bitsets are faster and will occupy less space. Bitsets support bit operations as well , this will help us later on in building our solution.

EXPLANATION:

Tap to view

We can build a segment tree to answer the queries and perform update operations in O(log(N)) time. Each node of the segment tree will store a bitset . Let any Node K of the segement tree represent range [L,R] then the bitset of this node will have set bits only at positions of the elements which occur odd number of times. For eg if 3,8,11 occur odd number of times then the 3rd,8th and 11th bits will be set in the bitset. With a bit more observation we can see that to combine two ranges into one i.e to combine ranges like [L,X]+[X+1,R] into [L,R] we use can use the XOR operation. While combining two ranges to get the answer for a bigger range we can see that any element can occur odd number of times in the bigger range if and only if the element occurs odd no of times in exactly one of the two smaller ranges. For eg. to combine [L,X]+[X+1,R] into [L,R] only the elements which occur odd number of times in exactly one of the two smaller ranges will also occur odd number of times in the range [L,R].

For queries of type 0 we need can just update the segment tree twice at index L and index R. This will take O(2*log(N)) time.

We can break the queries of type 1 into 2 parts:

1. L<=R: This is simple we will query the segment tree for the range [L,R] and output the number of set bits in bitset.
2. R<L: This query can be broken down into two parts first we query the segment tree for [L,N-1] and then query for [0,R]. We then combine the two queries just like we combined two ranges in the segment tree using the XOR operation and finally output the no of set bits in the bitset.

Bonus:

Queries where R<L can also be answered in just 1 query to the segment tree.

Overall Time Complexity:O((N+Q)log(N))

PREREQUISITES:

Segment trees with lazy propagation

PROBLEM:

Given N points (X_i, Y_i) on the plane, where $X_i < X_{i+1}$, support the following queries and updates:

- Given `L R K B`, set $Y_i := KX_i + B$ for each $L \le i \le R$
- Given `L R`, report whether the points in $[L, R]$ are collinear

EXPLANATION:

First, let's look at the collinear query. It can be rephrased in terms of slopes as follows:

- Define S_i to be the slope between the i-th and $(i+1)$-th point. That is, $S_i = \frac{Y_{i+1} - Y_i}{X_{i+1} - X_i}$.
- Then, the query $[L, R]$ really just asks if $S_L = S_{L+1} = S_{L+2} = \ldots = S_{R-1}$.

Computing the initial values of S_i is easy, so we just need to figure out how to maintain them across the given updates.

Let's break down what an update `L R K B` looks like in terms of the S_i.

- First, for each $L \le i < R$, we essentially just set $S_i = K$.
- The only other points that change value are S_{L-1} and S_R (if they exist).
- Finding out what they change to can be done if we are able to query for their respective y-coordinates. So, we also need to maintain the y-coordinates of the points across queries.

Note that the y-coordinate of a point depends solely on the last (K, B) update applied to it, so we can simply maintain the (K, B) values for each point. Each update is then just a range set update, which can be done using a segment tree.

From our discussion above, we see that maintaining the S_i values needs a data structure that can:

- Set a value of S_i to a range
- Set a value of S_i to a point (which can just be subsumed into the first point, treating a point as a length 1 range)
- Query for whether a range has all equal elements

The last part can be done by, for example, computing the maximum and minimum values in a range, and checking if they're equal.
So, we need a data structure that supports range set updates, and range min/max queries. Once again, this is just a segment tree with lazy propagation.

In order to not run into precision issues, it is recommended to maintain the S_i values as pairs of (numerator, denominator) instead of directly as floating-point numbers.

TIME COMPLEXITY

$\mathrm{O}(N + Q \log N)$.

PREREQUISITES

Segment tree

PROBLEM

Handle point updates, and maximum subarray sum queries in the given range after performing an arbitrary number of cyclic right shifts.

QUICK EXPLANATION

Use a segment tree that supports maximum subsegment sum range queries and point updates, to find the maximum circular subarray sum in a given range.

EXPLANATION

If we take $c = 0$, the answer is just the maximum subarray sum in that range. If we have a better answer which can be obtained by taking $c \neq 0$, then the elements of that shifted subarray will wrap around the unshifted array.

For example, consider the following array:
A = [2, -2, -2, 2]
With $c = 0$, max subarray sum is 2. But a better choice, in this case, would be to choose ans = A[0] + A[3] = 4, which “wraps” around the array. We can do so because we can choose some arbitrary c for which the wrapped around sequence turns into a contiguous sequence after shifts ($c = 1$ in this case).

In other words, the problem is the same as finding the maximum subarray sum in a cyclic array in a given range.

What can be the maximum sum of such wrapped around sequence? We can observe that the sum consists of a prefix of A and a suffix of A. We need to maximize **prefix_sum[i] + suffix_sum[j]** such that $i < j$. This means that we should leave out a contiguous part from the middle, and take the left and right parts (left being the prefix, right being the suffix).

Now, what should we leave in the middle to maximize the answer? Of course, we should leave the part corresponding to the minimum subarray sum.
Therefore, **ans = max(maximum subarray sum, sum - minimum subarray sum)**
where **sum** is the sum of subarray corresponding to the given range.

It only remains to show how to query maximum subarray sum and minimum subarray sum in a given range $[L, R]$ with point updates. This is a **standard problem** that can be solved using segment tree, which uses the idea of maintaining and combining answer at each nodes along with the best prefix sum, suffix sum, and the sum of the segment.
Note: The mentioned **standard problem** uses terms *subarray* and *subsegment* interchangably.

Once we know how to use segment tree to find subsegments with maximal sum, we can apply the same knowledge to find subsegments with minimal sums. We construct a segment tree on the array obtained after negating every element of the array, i.e. $A[i] = -A[i] \qquad \forall 1 \leq i \leq n$
To find minimum subsegment sum in given range, we can query for the maximum subsegment sum on this segment tree instead. Suppose the res is the result we get making this query, then we can update answer as:
ans = max(maximum subarray sum, sum - res)

PREREQUISITES:

Trees, Segment tree, **DFS order**

PROBLEM:

You are given a rooted tree with N nodes. If you select a node u, then all nodes in u's subtree with distance $\leq K$ will be covered. For each K from 1 to N, find the minimum number of nodes to select to cover all nodes and return the sum.

QUICK EXPLANATION

A greedy solution of always selecting the K-th ancestor of the deepest uncovered node is correct. We can prove that the sum of answers is $O(N \log N)$, so we will perform each greedy step in $O(\log N)$ time with segment tree operations, making the total time complexity $O(N \log^2 N)$.

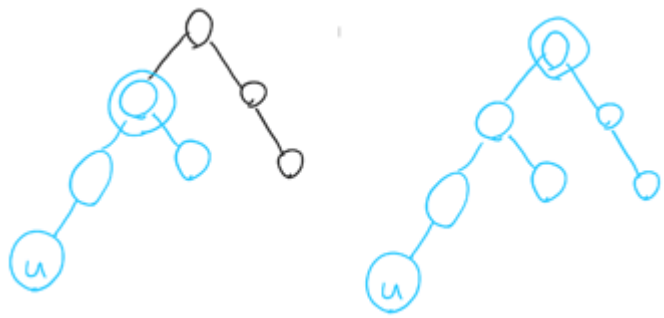
EXPLANATION:

Let's consider the deepest uncovered node u. Somehow, we need to cover it, so one of the $K+1$ ancestors of u should be selected (including u). Which of those $K+1$ ancestors do we select?

Observation 1. If v is an ancestor of u with distance $\leq K$, then after selecting v, all nodes in v's subtree will be covered.

Proof

u is the deepest uncovered node, so all other uncovered nodes in the subtree of v are closer to v and given that v has a distance $\leq K$ from u, v will also have a distance $\leq K$ from all uncovered nodes in v's subtree.



Observation 2. It is always optimal to select the K-th ancestor (or the root if there is no K-th ancestor) of the deepest uncovered node.

Proof

It is always better to select a higher ancestor that can still cover u because the uncovered nodes of a higher subtree is a superset of the uncovered nodes in the lower subtree.

Using observations 1 and 2, we can formulate a simple greedy solution, shown below:

- While there exists an uncovered node:
  - Find the deepest uncovered node and let it be u.
  - Let v be the K-th ancestor of u or the root if u has no K-th ancestor.
  - Select v and cover all nodes in v's subtree.

Observation 3. The sum of answers is $O(N \log N)$.

Proof

This means that if we can somehow perform each greedy step in $O(\log N)$, then we will have a total time complexity of $O(N \log^2 N)$.

Finding the K-th ancestor of u in $O(\log N)$ is standard and the most common way to solve it is using **Binary Lifting**. This leaves us with 1. finding the deepest uncovered node and 2. covering the entire subtree of v efficiently.

Subtree queries are a sign that we should use DFS preorder to reduce the subtree queries into range queries on an array. After applying DFS preorder, our queries are 1. finding the node with maximum depth on the entire array and 2. setting a range of nodes to -1 (to simulate covering the nodes). To support these two queries in $O(\log N)$, we just need a segment tree which supports range maximum queries and range set updates.

There's one last thing: At the start of calculating the answer for each K, we need to undo the changes from covering the nodes while calculating the answer for the previous K. Note that rebuilding the segment tree naively for each K is not an option because it will cause the entire solution to be $O(N^2)$. One simple (if you already know persistent segment trees well enough) solution is to just use a persistent segment tree. Another way you could uncover the nodes is to store the changes you make to the segment tree array and recover them before calculating the next K. Check my implementation for more details.

PREREQUISITES:

Treap , Segment Tree

PROBLEM:

Support the following operation on a treap which is a a binary search tree according to keys, and a max-heap according to weights:

1. insert a node with key k and weight w,
2. delete a node with key k,
3. find the distance between two nodes with key k1 and k2.

QUICK EXPLANATION:

Refer to the next section.

EXPLANATION:

Since the treap is a tree, the distance computation between two nodes can be reduced to the problem of finding lowest common ancestor of two nodes, and depth of a node. If the lowest common ancestor of node u and v is w, then

dist (u, v) = h (u) + h (v) - 2 h (w).

Lowest common ancestor (LCA):

In this section we discuss how to compute the lowest common ancestor of two nodes in a dynamic treap. The treap is a binary search tree according to the keys, hence, if we make an array of all treap nodes and sort them according to the key values, the array will represent the in-order of the treap.

Suppose we have two nodes u and v, and their indices is this in-order array are ku and kv (ku < kv). It is easy to note that the index kw of the lowest common ancestor node w of u and v must lie between ku and kv, i.e., ku < kw < kv. Further, the index kw will have the highest weight in the subarray [ku, kv]. Hence, using the in-order array one can easily find the lowest common ancestor of two nodes by doing a Range maximum query. for the range [ku, kv]. However, this will work only when the treap is static. In our problem the treap is changing, and hence its in-order array will also keep changing.

Even though the treap is changing, we can do an offline processing of the queries to find the set of all possible keys that will ever exist in the treap, and create an array with all keys present. Initially, we assign zero weight to all elements in the array. When a node (k, v) is inserted in the treap, we update the weight of the corresponding element in the array from 0 to v. Similarly, when a node (k, v) is deleted, we update the weight the corresponding element in the array from v to 0. In order to find the lowest common ancestor of u and v, we need to find the element in the subarray [ku, kv] with highest weight. Note that, now the array is fixed, we are only updating the values of its elements.

Hence, now the problem of finding lowest common ancestor is reduced to find the range maximum in a dynamic array. This can be achieved using a segment tree , such that both update and lowest common ancestor query take O (lg N) time, where N is the size of the array.

Depth Computation:

In this section, we discuss how to compute the depth of a node in a dynamic treap. First, we show how to compute the depth in a static treap, and then discuss how to modify our approach to handle the case of dynamic treap.

Let us start with an example. Suppose the treap contains the following nodes:
(1, 8), (2, 10), (3, 7), (4, 6), (5, 12), (6, 9), (7, 8), (8, 11)

The in-order array will be contain the elements in the same order as shown above. The node (5, 12) has the highest weight, hence, it will be the root of the treap. The left subtree of the treap will contain nodes (1, 8), (2,

10), (3, 7), (4, 6), while the right subtree of the treap will contain (6, 9), (7, 8) and (8, 11).

Lets look at the left subtree. Here (2, 10) is the node with highest weight, hence it will be the root, with left subtree containing (1, 8) and right subtree containing (3, 7) and (4, 6).

Observe that two nodes u and v can not be part of the same subtree, if there exist a node between them in the in-order array having weight higher than that of the two nodes. Because this node with higher weight (actually the node with the highest weight between the two lighter nodes) will become the root, and the two nodes with lower weights will go to different subtrees. Further, if there are no nodes in the in-order array between u and v, which has higher weight than both of u and v, then u and v will be in the same subtree having either u or v (whichever has higher weight) as root.

Based on the above observation, we can easily compute the depth of a node using the in-order array. If we want to compute the depth of a node v, which is at position kv, then we need to find how many indices kw exist such that the node w at index kw has higher weight than that of v, but no other nodes in the subarray [kv, kw] has higher weight than both of v and w. The number of such w's will be the depth of the node v.

Let us see how to compute the depth of (3, 7) in above example using this approach. There are only two possible subarrays [kv, kw]:
(2, 10) to (3, 7)
(3, 7) to (5, 12)

Hence, the depth of (3, 7) is 2. The immediate parent of (3, 7) will be (2, 10) and the next ancestor will be (5, 12).

The following snippet computes the depth of a node v at index kv using the in-order array A.

```
int depth(int kv) {
    int d = 0;
    int mx = A[kv].weight;

    // Look for subarrays of the form [kv, -].
    for (int i = kv + 1; i <= N; ++i) {
        if (A[i].weight > mx) {
            ++d;
            mx = A[i].weight;
        }
    }

    // Look for the subarrays of the form [-, kv].
    mx = A[kv.weight];
    for (int i = kv - 1; i > 0; --i) {
        if (A[i].weight > mx) {
            ++d;
            mx = A[i].weight;
        }
    }

    return d;
}
```

So, now we know how to compute the depth of a node in the treap. However, the above snippet is not very efficient as it takes O (N) to compute the depth. Next, we discuss how to reduce the complexity of depth computation.

Data structure to reduce the complexity of depth computation:

In the above snippet, there are two traversals to compute the depth of a node. We only consider the first traversal, the other one can be done in a similar way. So the problem is that we are given an array, and for a given index k, we want to find out how many times the maximum of subarray [k, r] will change where r is varying from k + 1 to N.

The problem can be solved using a segment tree. We create a segment tree corresponding to array, where each node in the segment tree corresponds to a subarry. The leaf nodes correspond to the subarrays of size one, and

the parent of two nodes corresponds to the subarray which can be obtained by merging the subarrays corresponding to the two children.

Each node of the segment stores some information corresponding to its subarray. More precisely, this information is the “prefix maximum” function of the subarray. The “prefix maximum” function is already described in the editorial of **problem**. For a given array A, we define the “prefix maximum” function PM as follows: PM(i) = max(A[0], A[1], …, A[i]).

Clearly, the PM() function is a step function, and we only need to store the positions where it makes step and the value of the function at these positions.

Using this information the number of of subarrays [k, r] can be computed easily. In order to compute the number of subarrays [k, r] for a given k, we need to start from the leaf node in the segment tree corresponding to the subarray [k, k], and move towards the root, while maintaining mx, and d as in the above snippet. This is also shown in the snippet below:

```
int right_depth(segtree *tree, int k) {
    int d = 0;
    node *n = tree->get_leaf(k);

    // the highest value in the subarray corresponding to node n.
    int mx = n->max();

    while (n->parent) {
        node *p = n->parent();

        // We are only interested in right subarrays.
        if (n is right child of p) {
            n = p;
            continue;
        }

        // how many elements in the PM function of the right child are greater
        // the maximum value seen so far.
        d += p->right->rank(mx);

        // update the maximum value.
        mx = max(mx, p->right->max());
        n = p;
    }

    return d;
}
```

Since the height of segment tree is O (lg N). The above code will make at most O (lg N) iterations. Also in each iteration we need to compute the rank of an element. If the segment tree nodes stores the PM() function using a sorted array of steps, then this can be done in O (lg N) time as well, resulting in O (lg^2 N) complexity of the above operation.

Now let us see if we can avoid storing the PM() function at each node in the segment tree. The only information that we about the PM() function is the number of elements in it which are greater than a given value h. Next, we show how to compute this information without storing the whole PM() function. We store only two values in a segment tree node:

1. The maximum element of the corresponding subarray represented by mx, and
2. number of steps in the PM() function represented by len.

```
struct node {
    int rank(int h);

    int mx;
    int len;

    // Use pointers for simplicity.
```

```
    node *parent;
    node *left;
    node *right;
};
```

Suppose we want to calculate the rank of a given value h, i.e., the number of elements in the PM() function which are greater than h. If h = 0, then this will be the same as the number of steps, i.e., len. If h >= mx, then this value will be zero. Now consider the case when 0 < h < mx.

In this case we look at the two children of the node, which correspond to the smaller subarrays. There could be two possible cases:

1. h >= left->mx. In this case no element in the left subarray is greater than h, so we can ignore it, and the answer will be the rank of h in the right subarray.
2. h < left->mx. In this case all the elements of the PM() function of this node will be included except the ones which are smaller than h. Note than none of these smaller elements will be from the right subarray. This is because all the elements of right subarray which are present in the PM() function must be greater than left->mx, which in turn is greater than h. Hence, we only need to remove the elements from the left subarray, i.e., the answer would be (len - left->len + left->rank(h)).

Note that, in both cases we have reduced the problem to a smaller subarray which has half of the size of the original subarray. Hence, the complexity of this approach would be O (lg N).

```
// Returns number of elements of the PM() function which are
// greater than h.
int rank(int h) {
    if (h >= mx)
        return 0;

    if (h == 0 || node is a leaf)
        return len;

    // Case 1.
    if (left->mx <= h)
        return right->rank(h);

    // Case 2.
    return (len - left->len) + left->rank(h);
}
```

Now, we have seen how to compute the depth of a node in a static treap. Let us see how to modify our approach so that it also works for a dynamic treap. The only thing that we need to care about is how to update the mx and len value of a node, when the value of an element in the subarray changes. This can be done easily, we just need to start from the leaf node corresponding to size one subarray, and move towards root while updating all the nodes in the path.

```
// Updates the value at position k to val.
void update(segtree *tree, int k, int val) {
    node *n = tree->get_leaf(k);
    n->mx = val;

    n = n->parent;
    while(n) {
        n->mx = max(n->left->mx, n->right->mx);
        n->len = n->left->len + n->right->rank(n->left->mx);
    }
}
```

Hence, an update can be done in O (lg^2 N) time.

TIME COMPLEXITY:

O (N lg^2 N)

PREREQUISITES:

Disjoint Set Union, Dynamic Connectivity

PROBLEM:

Initially given is a prime p and a matrix M of size $N * N$.

The matrix is required to be $curious$, i.e., once it is completely filled,

- Each cell should contain an integer between 1 and $p - 1$ inclusive .
- For each non trivial square **submatrix** A of M (a submatrix containing more than 1 cell) , its determinant $|A|$ should be a multiple of p.

You are required to handle Q queries . After each addition or delete query in the matrix M, report the number of possible ways to fill up the empty cells in M to make it $curious$ modulo $(1e9 + 7)$.

QUICK EXPLANATION:

- It can be shown that the $2nd$ curious condition is equivalent to the matrix having rank 1 with respect to modulo p. In a rank 1 matrix, each row is a multiple of every other row and each column is a multiple of every other column . Hence there is a cyclic dependency between the cells, i.e., $1 \leq a, b, c, d \leq N$, $M_{a,b} * M_{c,d} \equiv M_{a,d} * M_{c,b} \; mod \; p$, thus if any 3 of these cells have values in them, the 4th cell necesaarily needs to have a particular value to obey the rank 1 matrix condition .
- There are actually a total of $2N - 1$ individual cells whose values independently can determine the values to be filled in all other cells for the matrix to have rank 1 . Intuitively just consider the $1st$ row and $1st$ column filled, a total $2N - 1$ cells . These are actually the independent constants in a graph, the count of free constants available decide the count of ways to form a curious matrix, $(p - 1)^f, f$ being the current count of free constants.
- The problem can be modelled into a graph problem. Consider $2N$ nodes in the graph, each node representative of a row or a column, nodes 1 to N for the rows and $N + 1$ to $2N$ for the columns . For every filled cell $M_{x,y}$ containing value v, consider a directed edge from node x to node $y + N$ with weight v and another directed edge from node $y + N$ to node x with weight $1/v$ (inverse of v w.r.t prime p) . Now to obey the rank 1 matrix condition, the products of all edge weights in a cycle in the graphymust be 1 . The count of free constants is decided by the number of disconnected components in graph subtracted by 1.
- The addition query alone could have been handled in $O(logN)$ by a DSU. However to handle deletion queries, dynamic connectivity comes into play .

EXPLANATION:

Counting ways to get a curious matrix from an initially empty matrix

Let's count the number of ways to fill an empty matrix to make it $curious$.

If we were asked to just satisfy the first condition only, the number of ways would have been $(p - 1)^{N*N}$. Does the count remain same if we were asked to incorporate the $2nd$ condition as well ? No, it doesn't since by the second condition, there is certain dependency imposed on certain cells once some other cells are filled . This dependency is what is enforced by the **rank** 1 property of the matrix .

Proving the $2nd$ Curious condition is equivalent to rank 1 requirement

Consider a, b such that $1 \leq a, b \leq N - 1$, consider the $2 * 2$ submatrix formed by rows a and $a + 1$ and columns b and $b + 1$ as,

$$A = \begin{bmatrix} M_{a,b} & M_{a,b+1} \\ M_{a+1,b} & M_{a+1,b+1} \end{bmatrix}$$

The curious condition requires the determinant of the submatrix, $|A|\ mod\ p$ = 0, i.e., $M_{a,b} * M_{a+1,b+1} \equiv M_{a+1,b} * M_{a,b+1}\ mod\ p$. This implies, $\frac{M_{a,b}}{M_{a+1,b}} \equiv \frac{M_{a,b+1}}{M_{a+1,b+1}}\ mod\ p$. Thus the bottom row of submatrix A is a multiple of the top row, now varying b from 1 to $N-1$, it can be found that $(a+1)$-th row is a multiple of a-th row in the matrix M. It can also be said the other way round, i.e., a-th row is a multiple of $(a+1)$-th row This can be generalised to show that each row is a multiple of every other row in a curious matrix . Similarly, this can also be shown for columns too . Hence, considering modulo prime p, the rank of any curious matrix is 1 .

Now we understand the cyclic dependency that is created among the cells . Taking cue from the rank 1 property we just proved, we use the fact there are $N+(M-1)$ free constants in a Rank 1 matrix of dimension $N*M$, the N-components of the first column vector (let's call them $x_0, x_1 \ldots x_{N-1}$), and the multiplier of each of the other column vectors with respect to the first (let's call them $y_1, y_2 \ldots y_{m-1}$). Hence the number of ways to fill up an empty matrix to make it $curious$ is $(p-1)^{2N-1}$, since the answer is the number of values each constant can take to the power of number of such constants.

To better understand the notion of rank 1 matrices and free constants, look watch this **video** .

Transformation to a Graph problem

Since, we have already established that the rows are multiples of each other and columns are multiples of each other too, we have to somehow effecitively store the ratios between them to handle the queries .

Consider the matirx in this state ,

$$\begin{bmatrix} .. & .. & .. & .. & .. \\ .. & M_{a,b} & .. & M_{a,d} & .. \\ .. & .. & .. & .. & .. \\ .. & M_{c,b} & .. & .. & .. \\ .. & .. & .. & .. & .. \end{bmatrix}$$

At this stage, we know c-th row is a multiple of a-th row by a factor of $M_{c,b}/M_{a,b}$, also d-th row is a multiple of b-th row by a factor of $M_{a,d}/M_{a,b}$. We attempt to maintain this ratios between these rows and columns together in a graph such that :

- The graph contains $2N$ nodes, each node representative of a row or a column, hence $2N$ of them, x-th row denoted by node x, y-th row denoted by node $y+N$.
- For every filled cell $M_{x,y}$ containing value v, consider a directed edge from node x to node $y+N$ with weight v and another directed edge from node $y+N$ to node x with weight $1/v$ (inverse of v w.r.t prime p) .

On careful observation on the construction of the graph, we can infer that the ratio between a-th row and c-th can be obtained as the product of the edge weights in the path from node c to a . Notice that there can be many directed paths from c to a, but the product of each of those paths should be same . By construction the reverse path would have a product that is an inverse of the product of edge weights in original path . Hence, we can say, it is needed to satisfy that product of edge weights in each directed cycle in the graph should be 1. Another crucial observation is that the count of free constants in the matrix being constructed is equal to the number of components in the graph subtracted by 1.

Let's now look at how to handle the queries.

Now on an addition query $x\ y\ v$,

- If there was earlier a path between x and $y+N$, it needs to obey that the product of all the edges in the path from node x to node $y+N$ should be v .

- If there wasn't a path earlier, we add an edge as described above. In notion of the rank 1 matrix count of the free constants decrease by 1 since number of components decrease and hence after this query the number of ways to obtain curious matrix decreases .

Answering the query

If the query is valid, i.e., there exists a way to fill up the remaining cells of the matrices , the answer is calculated as, $(p-1)^f$, f being the current count of free constants. Count of free constants can be found by getting the current number of components in the graph. Report the answer $modulo\ 1e9+7$.

Efficiently handling addition queries through Disjoint Set Union

Both the above checks can be handled faster by maintaing a disjoint set union structure . Note usually the DSU's are associated with undirected edges, we too can work with undirected edges only since between any two nodes, the weight in one of the directed edge is just inverse of the the directed edge in opposite direction , this can be maintained using undirected edges only, just maintain the edge that is directed from a node to its parent in the DSU structure .

For an addition query $x\ y\ v$,

- To check if there is a path between x and $y+N$, we just need to check if the two nodes are in same subset of DSU or not .
- If the nodes are in the same component, find p_x, the product of all edges from x leading up to r_x, the root of the subset containg the node x and p_y, the product of all edges from $y+N$ leading up to r_y,the root of the subset containg the node $y+N$ (All products are found modulo prime p) . Note that we have actually simulated the reverse path from $y+N$ to r_y . Since for obeying rank 1 matrix rule, the product of the path from x to $y+N$ was required to be v and hence it must hold that, $v \equiv \frac{p_x}{p_y}\ mod\ p$.
- If the nodes are not in the same component, we need to connect r_x and r_y, the roots of the subsets containing x and $y+N$. We use smaller to bigger merging and accordingly take care of the edge weight we put between them . If r_y is made a child of r_x, the edge connecting r_y to r_x (we only maintain the edge in upward direction) is assigned an edge weight, $w = \frac{p_x}{v*p_y}\ mod\ p$, while if r_x is made a child of r_y, it is assigned an edge weight, $w = \frac{v*p_y}{p_x}\ mod\ p$. The number of components get decreased and in sense the free constants as well.

But note that deletion queries cannot be handled efficiently just by DSU . Hence so far we have solved subtask 1 and 2 . For subtask 1, it is not needed to handle deletions efficiently .

Efficiently Handling Deletion Queries through Dynamic Connectivity

Finally, Handling delete queries is a simple albeit programmatically tedious modification, what follows is a standard technique described here: **Deletion in $O(logN)$**. We need to add delete operations to the DSU, for which we use a Segment Tree and a regular DSU as described above which does and undoes operations on a stack (by storing all changes made after each operation and undoing them later). We loop over the query list and pair up the set and unset queries, and the ranges in which the value is set, we add it to the Segment Tree. Then we perform a DFS on the Segment Tree, passing the DSU around, and this allows all the operation to be applied as deleted in the stack-order. Since path compression is not possible with deletion (roll backs), merging smaller into bigger is why we used in DSU as well .

For subtask 3, it is actually not needed to maintain edge weights, just maintaing the edges is fine . Complexity wise it's same as in subtask 4, just the careful handling of edge weights is what is not needed in the subtask .

For subtask 4, we use everything discussed so far to efficiently answer the queries in $O(Qlog^2Q)$.

If you are new to dynamic connectivity, it's recommended to try the problem **Extending Set of Points** and look at the attached **editorial** .

Also give a try at **Dynamic connectivity contest** .

PREREQUISITES:

Segment Tree, Coordinate Compression

PROBLEM:

You are given an unweighted directed graph whose edges are given by:

- For each i, where $1 \leq i \leq M$, and for each pair of vertices $u \in [A_i, B_i]$, $v \in [C_i, D_i]$, there is an edge from u to v.

Find the shortest distance between vertices X and Y.

QUICK EXPLANATION:

- First, we compress all the vertices given in the input, so that the number of vertices becomes $O(M)$.
- Build a graph as shown in the figure below, of size equal to the number of compressed vertices. The tree edges are weighted 0. Then for each valid i, decompose $[a_i, b_i]$ into subsegments in the *lower half* and draw an edge from each of them to each subsegment of $[c_i, d_i]$ in the *upper half*. All these edges should be of weight 1. In this way, the graph consists of $O(M \log^2 M)$ edges. Now, we can find the answer by running 0-1 BFS from source $[x, x]$, and finding the distance to $[y, y]$. This solution takes $O(M \log^2 M)$ time.
- Instead of drawing an edge from each subsegment of $[a_i, b_i]$ to each of $[c_i, d_i]$, we can define an extra vertex v_i and draw an edge from each subsegment of $[a_i, b_i]$ to v_i, followed by an edge from v_i to each subsegment of $[c_i, d_i]$. The graph now consists of a total of $O(M \log M)$ edges. We can then run 0-1 BFS as before, but divide by 2 to get the final answer. Thus, the problem is solved in $O(M \log M)$ time.

EXPLANATION:

Firstly, we can compress the input so that instead of dealing with N vertices, which can be upto 10^9, we only need to deal with $O(M)$ vertices in our graph, as there are only $4M + 2$ vertices to compress. Let us represent the number of vertices after compression to be K.

Motivation

We can start by naively building the graph as given in the problem, followed by a BFS. The problem with this approach is that for each valid i, we require $O(M^2)$ edges to be added to the graph in the worst case. Clearly, we need a structure which can represent that same information with far fewer edges. We can recognize that a **segment tree** is well-suited for such a role. For instance, let us build a segment tree of size K. Then for each valid i, we can decompose $[a_i, b_i]$ into $O(\log K)$ subsegments of the segment tree, and draw an edge from all those subsegments to all subsegments of $[c_i, d_i]$. After some experimentation with similar kind of structures, we can realize that the following graph works for us.

Solution

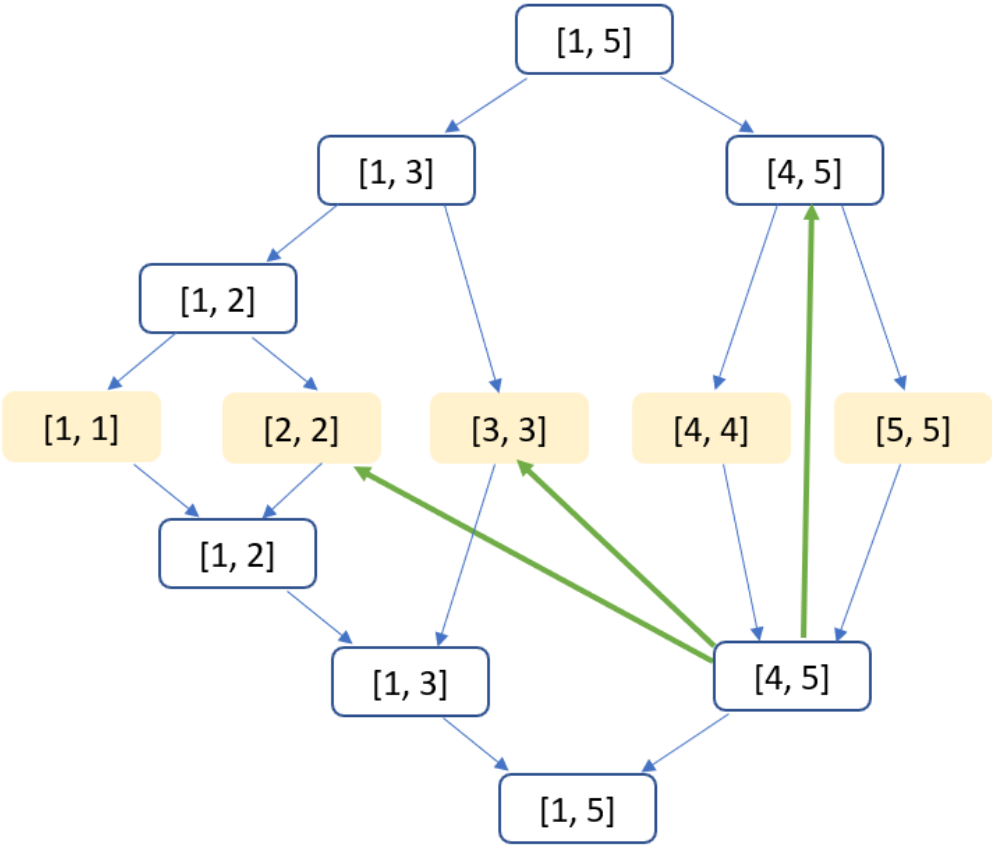


Figure: The graph for $K = 5$. Tree edges of weight 0 are shown in blue. For an example, edges of weight 1 from $[4, 5]$ to $[2, 5]$ are shown in green.

Let us construct this graph, i.e. the two segment trees of size K joined together as shown in the figure. We initially assign all the tree edges weight 0. Now, we start iterating over each valid i. As we do in any segment tree, we can partition every segment $[a, b]$ into $O(\log K)$ subsegments of the segment tree. Thus, for every $[a_i, b_i]$ we draw an edge from each of its subsegments in the *lower half* to each subsegment of $[c_i, d_i]$ in the *upper half*. We draw these edges with weight 1. So why does this graph work?

We show that if there exists an edge from vertex p to q in the original graph, then and only then there exists a path of distance 1 from segment $[p, p]$ to $[q, q]$ in our constructed graph. Note that if there is an edge from p to q, there must be an i such that $p \in [a_i, b_i]$ and $q \in [c_i, d_i]$. Thus, there must be a subsegment of $[a_i, b_i]$ in the segment tree which includes p, let us call it $[a_i^{'}, b_i^{'}]$, and a subsegment of $[c_i, d_i]$ including q, which we can call $[c_i^{'}, d_i^{'}]$.

It is easy to see that all segments in a segment tree that include p, lie on the path from $[p, p]$ to the root. Thus, we can move downwards from $[p, p]$ in our graph and reach $[a_i^{'}, b_i^{'}]$ in the lower half of the graph. Further, as we had drawn an edge from each subsegment of $[a_i, b_i]$ to each subsegment of $[c_i, d_i]$, there exists an edge from $[a_i^{'}, b_i^{'}]$ in the lower half to $[c_i^{'}, d_i^{'}]$ to the upper half. So we can move to $[c_i^{'}, d_i^{'}]$ and from there on we can move downwards in the tree until we reach $[q, q]$. Thus, we have shown that a path exists from $[p, p]$ to $[q, q]$. Also, notice that we used exactly 1 unit of distance in doing so. The only-if part of the proof is very similar.

Hence, any path from x to y of length l in our original graph corresponds to a path from $[x, x]$ to $[y, y]$ of the same distance l in our constructed graph, and vice versa. So how can we find the shortest path from $[x, x]$ to $[y, y]$? The simplest way is to use **0-1 BFS** with source vertex $[x, x]$, which is possible since all our edges are of weight 0 or 1. Alternatively, we can use Dijkstra's but that will result in an extra log factor in the time complexity.

Time complexity

Remember that the compression phase took $O(M \log M)$ time. Now, as building a segment tree of size K requires $O(K)$ edges our initial graph consisted of $O(K)$ edges as well, or equivalently $O(M)$ edges. Then, for

each i we divided the segments $[a_i, b_i]$ and $[c_i, d_i]$ into their $O(\log M)$ subsegments, and we defined an edge from each subsegment of $[a_i, b_i]$ to each of $[c_i, d_i]$.

This means that for every i, we inserted $O(\log^2 M)$ edges in our graph. The total edges in our graph becomes $O(M + M \log^2 M)$, and it is not difficult to see that the time taken was $O(M \log^2 M)$ as well. Further, as 0-1 BFS takes $O(V + E)$ time, this part also takes $O(M \log^2 M)$ time. Hence, our final time complexity is $O(M \log^2 M)$.

Further reducing the time complexity

We can further reduce the time complexity to $O(M \log M)$ very simply. For each valid i, instead of drawing an edge from each subsegment of $[a_i, b_i]$ to each of $[c_i, d_i]$, we can define an extra vertex v_i and draw an edge from each subsegment of $[a_i, b_i]$ to v_i, followed by an edge from v_i to each subsegment of $[c_i, d_i]$. As before, we draw these edges of weight 1. Note that we only used $O(\log M)$ edges here instead of $O(\log M^2)$ previously.

With this modification, we can easily show that any edge from p to q in the original graph now corresponds to a path from segment $[p, p]$ to $[q, q]$ of distance 2 in our new graph. Indeed, the only change in the previous proof is that the distance of $[a_i^{'}, b_i^{'}]$ to $[c_i^{'}, d_i^{'}]$ is now 2, but otherwise the same path exists. Thus, we can run 0-1 BFS as before from $[x, x]$ to $[y, y]$ in this graph, but divide by 2 to get the final answer. Since the graph now consists of only $O(M \log M)$ edges, our final time complexity is $O(M \log M)$ as well.

PREREQUISITES:

Persistent Segment Trees, Small-to-Large-Merging

PROBLEM:

Danya has a rooted tree with $N \leq 10^5$ nodes (numbered 1 through N). Node 1 is the root and for each i ($2 \leq i \leq N$), the parent of the i-th node is p_i. For each valid i, the i-th node also has a value a_i.

Danya really loves his tree and wants to play with it. You should answer his Q queries. In each query:

- You are given two integers X and D.
- Consider the set S of all nodes v that lie in the subtree of node X (including X) such that the distance between X and v does not exceed D.
- The answer to this query is the number of different values a_v among all $v \in S$.

Note that there are 5 such test-cases, and that the queries are online.

QUICK EXPLANATION:

I promise this will be quick!

The solution hinges on the fact that if we can maintain for every subtree, the depth of the highest node of every colour inside its subtree, we can just count using a persistent segment tree how many of them are less than D. To merge these segment trees fast, traverse on the smaller segment tree and merge. Note that this does not handle when the depth of a color gets updated. Rather, if there were two different depths of the same colour in the two segment trees, we would have accounted for two different depths for it. To correct this, we also need to delete the occurrence of that colour when it becomes “deactivated”. To find the first ancestor where a particular node (and thereby the depth of the colour associated with that node) we can use dfs start and end time. The fact that there will be only $O(n)$ such removals necessary helps keep the complexity $O(nlogn)$.

EXPLANATION:

Observation 1

First off, one might be tempted to think “But oh! we are dealing with subtrees so why not Euler Tour, DFS array, Etc Etc”. The catch is the D variable. And that makes everything related to subtree-being-a-contiguous-range-in-array a *tad* hard to do.

But....

One idea you might have is, can’t we just add the distance variable like another “dimension” in our search parameter, and then find for distinct values in a range using Segment trees? (more detail about the technique **here**). Well, maybe. But its really hard to do. If you can figure out a fast enough way to do so, do mention in the comments.

Observation 2

In general, a good idea is to simplify our problem and try to solve the problem in that simple circumstance. What can be a better simplification here than to do away with D. Note that we don’t want to again go back to thinking about subtree-ranges transforms, since we already know its somewhat hard to do. Nonetheless… what can we do?

A possible solution after Observation 2

Well, since there is no D, and we are counting number of distinct colors in a subtree, a popular technique for this is called "Small to large" merging. (Click **here** and **here** to learn more).

Offline or online?

Well, it is possible to store the answer for all vertices and answer queries online, but that feels like cheating. Instead, can we make it actually online?

Can you?

Turns out, you can answer queries online. Just make whatever data structure you are using to store the "set" of all colors **persistent**.
In particular, when you are doing the dfs and merging subtrees, while inserting elements into the "set" of the heavy child, you get back a new version which you then assign to the current node. In this way, you still maintain the "live" version of the data structure at each node, and can truly answer queries online! **Note** that "set" doesn't necessarily mean STL Set, but rather a set-like data structure of your choice.

Implementation 1

Whenever we are using persistence, it is a good idea to use a segment tree as the underlying data structure. Hence, in the above case, we maintain the *set* in the form of a Segment Tree. I will leave the details of on what array we are building the segment tree as an exercise 😛

Implementation 2

A question might be, won't segment trees take a lot of space? For example, if we built a segment tree on the identity array for color, (where if we insert a node of color C, we just do $Arr[C]++$), wont we need to store a segment tree of size 10^5 at each node? Won't that be taking us huge amounts of memory? Oh… persistance… But wait! We are using persistence to only merge segment trees. What if it was a star graph, ie, node 1 connected to all nodes $2, 3, 4 \ldots N$. Wouldn't that take $O(N^2)$ memory? Well, yes.

HOWEVER, we can reduce the memory usage effectively. The trick is to use **dynamic segment trees**.

Observation 3

Let the depth of u be X. Then any node in subtree of u with depth at most $X + D$ is valid. Seems rather … uniform right?

Observation 4

Whenever we are counting something, it is a good idea to "Fix" a certain property of an element, using which we count it. What can be such a "property" of the colours/node containing the colors, that we can utilise to keep track of the depth/height of the colors, and whether it is inside D range of the given node u.

Using the above observations

We can just store the least depth occurence of every color! If we maintain only those numbers, then it's just finding which all nodes have depth $< Y$ for some Y when querying for a subtree. (Don't forget that whatever

we maintain, we will use persistent data structures so that we can do online queries).

How can we maintain?

It seems appropriate to use a segment tree on the frequency table on depths of nodes. Right? RIGHT?

Okay I'll explain

Here's what I mean. Let the underlying array of the segment tree be called **Arr**. Let's say we insert the node u which has color c. Then the operation that we want to support is $Arr[depth[u]]$++. Notice that we aren't directly using the value c. However, it is needed because we want to ensure that only 1 occurance of a node with color c exists, and so that the $depth[u]$ for that node u is the minimum possible (since we want to maintain the highest occrurence of a color in a subtree) Hence, when merging two subtrees, let's say in the data structure of the heavy child, the highest node of color c was v, and in the light child the best occurence is u, and $depth[v] > depth[u]$. Then, we do the following:

- $Arr[depth[v]] - -$
- $Arr[depth[u]] + +$

To answer for a query node u, we go to its version of the segment tree, and just get range sum $[depth[u], depth[u] + D]$. (Think why)

Initial Attempt at Complexity Analysis

The complexity analysis is simple for this. We use the idea from small to large, and since inserting an element into a segment tree costs $O(logn)$, we get a total complexity of $O(nlog^2n + qlogn)$.

Making merging faster

A faster way to merge, instead of just putting elements one by one into the segment tree, is by traversing both the segment tree and merging. Claim is that this will make merging the two segment trees fast! (Proof given below). The only catch, while merging, we just merge the nodes – we don't make any updates. To explain what I mean here: Let's say we are merging two Segment trees T_1 and T_2. Let there exist $u \in T_1$ and $v \in T_2$, such that the color of both u, v is c. Then, after merging, our segment tree will contain both the occurrences. How do we delete that extra occurrence like in our previous implementation? ITS HARD right? I mean, now you don't keep track of which color contributed to which depth – you just keep track of the depths in the segment tree. Maybe you can keep track of the nodes and their colors separately as well… but can you? Won't that ruin the complexity, since just to maintain that information by small-to-large, we need $O(nlog^2n)$ time?

How to handle repeated occurrence

If we can get when is the first ancestor when a node will need to get "removed" (rather which is the first ancestor where is it will get replaced by another occurrence of the same color, which was in a different subtree, with lesser depth). Formally, for every node u of color c we need to find first ancestor x, such that $depth[x]$ is greatest possible, and there exists a node v of color c such that $depth[u] > depth[v]$ and $LCA(u, v) = x$. Then, after we merge the segment trees at node x, then for all such node us, we update the segment tree as $Arr[depth[u]] - = 1$. This will help keep everything correct. (Think why this is equivalent to our previous merging one by one. Think about how we are doing the same additions and subtractions just in a roundabout way). Since there are only $O(n)$ such deletions required, we incur a total time of $O(nlogn)$ for the mergings.

How to find these ancestors?
Hint 1:

Process each color one by one.

Hint 2:

Use depths and euler tour.

Hint 3:

Only LCA of the nodes matter.

Hint 4:

LCA of adjacent nodes are all that matter.

Details

For every color, sort according to increasing depth. Now let's say you are processing the xth element, and **you have already inserted everything with lesser depth** into a Data Structure. Now, let's say that inside this data structure, everything is **sorted by DFS startTime of the nodes**. Then, we just need to check for LCA with the previous and next nodes for x. You can just use a **set** for the data structure as sets provide iterators to move to next and previous elements easily.

Merging two Segment Trees

Now, lets backtrack a bit and focus on the merging of segment trees part, and how to do it. Well, what we essentially do is traverse both the segment trees simultaneously. Now, we do this as long as there are corresponding nodes in both the trees. Let's say the trees be T_1 and T_2, and we are currently at node X. Let's say the left child of X was **null** in T_1, and some node Y in T_2. Then after merging, we just attach Y as the left child of X in the updated version, instead of going further down. However, if both left children are not **null**, then we recursively merge them, and add them as the left child. Note that, this ensures that we will visit at most the nodes of the smaller segment tree, and probably less.

But what is the complexity?

First, notice that every time we visit a node during merging, we create a new node due to persistence. Hence, if we can put an upper bound on the number of nodes created, we can get a complexity. Again, remember that when we call our merge function with trees T_1 and T_2, we only traverse those nodes which are present in both trees. For a moment, let's forget about persistence. Then, you basically traverse two nodes and "merge" them into one. Effectively what happens is, if T_2 was the segment tree of a light child, you won't visit the nodes of T_2 again, as only the tree of the heavy child is pushed upwards. Hence, since for **every two corresponding nodes** that you visit in T_1 and T_2, you **"discard"** one node. Also note that unless both the corresponding nodes exist, you don't visit either – you just attach the child pointers to the merged node. How many total nodes will be created and visited in this scenario? (Note that we aren't considering persistence here). It's $O(NlogN)$ since we are essentially inserting N numbers overall and in dynamic segment tree, that corresponds to $O(NlogN)$ memory. Hence, we can essentially discard only $O(NlogN)$ nodes, which means that the total number of visited node can be $O(2NlogN)$ which is $O(NlogN)$. Now, let's get back to persistence. Won't that be creating many new nodes? No. Note that persistence only creates a new node when we visit a node. We just saw how we only ever visit $O(NlogN)$

nodes. Hence, we will only be creating $O(NlogN)$ extra nodes due to persistence. Hence, overall our complexity becomes $O(NlogN)$.

[**NOTE:** : I have first proved the bound without using persistence, and then elaborated on what changes when we use persistence.]

Potential Functions

Note that I have explained in an informal language, what could have been written concisely using Potential Functions. Consider the potential function P = sum of sizes of the Segment Trees. We will not be considering persistence. When we do an $insert$ operation, we add $logn$ to P. Hence, max value of P can be $NlogN$. Whenever we visit two corresponding nodes in T_1 and T_2, we subtract 1 from P. Hence, we only ever visit $2NlogN$ nodes, worst case. Since in persistence we build additional nodes for every visited node, we have $O(NlogN)$ extra nodes. Hence, our complexity is bounded by $O(NlogN)$.

For more discussion on merging, and related problems, click **here** .

Final Complexity

So what is the final complexity? The merging of segment trees take $O(NlogN)$ time, and we need to do an additional $O(N)$ deletions, so even that takes $O(NlogN)$ time. Hence, the total complexity is $O(NlogN)$.

SOLUTIONS:

PREREQUISITES:

Segment tree .

PROBLEM:

Given an array of numbers A, support the following two queries:

1. for a given range [L, R], reduce each number in A[L…R] by its smallest prime factor,
2. for a given range [L, R], find the number in A[L…R] with largest smallest prime factor.

QUICK EXPLANATION:

Use a segment tree while maintaining the segments in which each number is equal to one.

EXPLANATION:

We are given an array A of size N, and we are supposed to maintain two kind of range queries on this array:

1. Update the numbers in a range A[L…R].
2. Query some information about the numbers in the range A[L…R].

Let us first solve a simpler version of the problem, in which we support the following queries, and then we will come back to the original problem:

1. Update a number A[x]
2. Query some information about the numbers in the range A[L…R].

This seems like a classical **segment tree** problem. We can create a segment tree for the array. As explained **here** in the segment tree each node maintains some information about the elements A[k * 2^x, (k + 1) * 2^x). In our case, this information would be the largest smallest prime factor of the numbers in the range.

```
struct info {
  // The node corresponds to A[low..high]
  int low;
  int high;

  // The largest value of the smallest prime factors
  // of the number in A[low..high].
  int value;

  // The children nodes of this node containing
  // information about the ranges
  // [low, mid], and [mid + 1, high].
  info* left;
  info* right;
};
```

Once we update a number A[x], the information in all nodes containing the index x will be updated. There are at most O (lg N) such nodes, hence the update operation can be done in O (lg N) time. For the query operation about the range [L, R], we find the list of nodes whose ranges is contained in [L, R], and take the maximum of their values. Once again, there are O (lg N) such nodes, hence the query operation can also be implemented in O (lg N) time. For the implementation details of update and query, please see **here** .

Now, let us see how we could solve the original problem. One possibility could be that, when we get the range modification entry, we modify each element in the range one by one, and hence perform (R - L + 1) single element update queries. This approach would be to slow, as it makes the complexity of range update query to O ((R - L + 1) lg N). Next, we discuss that many of these single element update queries can be ignored, and the number of actual performed queries is much lower.

Note that, if the value of an element A[x] is 1, then update query on A[x] does not have any effect on the segment tree. We call such queries as degenerate queries, and can discard them. Also note that a number M

can be written as a product of at most O (lg M) prime numbers, Hence, we can perform at most O (lg M) non-degenerate modification queries on A[x] = M, all the latter queries will be degenerate. This means that if all numbers in the array are smaller than M, then a most O (N lg M) non-degenerate single element update queries would be performed. Each single element update query takes O (lg N) time, and hence all update queries can be performed in O (N lg M lg N) time.

However, now the problem is how to find the non-degenerate single element update queries for the range [L, R]. For this purpose, we maintain another information in the segment tree node, which is the number of elements in the range which are larger than one.

The following snippet shows how to use this information so that we only perform single element update queries, without iterating through all elements in the range. The idea is similar to the lazy propagation.

```
struct info {
  // The node corresponds to A[low..high]
  int low;
  int high;

  // Number of elements in A[low..high], which are larger than 1.
  int count_non_degenerate;

  // The largest value of the smallest prime factors
  // of the number in A[low..high].
  int value;

  // The children nodes of this node containing
  // information about the ranges
  // [low, mid], and [mid + 1, high].
  info* left;
  info* right;
};
```

Now, while performing the range modification query, we can discard all subranges which have all elements equal to one.

```
void update (info* nd, int L, int R) {
  // This node has a range disjoint with [L, R]
  if (nd->low > R || L > nd->high) continue;

  // All elements in this range are 1, nothing to do.
  if (nd->count_non_degenerate == 0) continue;

  if (nd->low == nd->high) {
    // Perform the single element update query
    ...
    return;
  }

  update(nd->left, L, R);
  update(nd->right, L, R);
}
```

This means all the range update queries can be performed in O (N lg M lg N), while the range query time is O (lg N) per range.

TIME COMPLEXITY:

O (N lg M lg N + Q lg N), where N is the size of the array, M is the bound on the elements of the array, and Q is the number of queries.

PREREQUISITES:

Segment Tree with Lazy Propagation, basic math.

PROBLEM:

Given an array A of length N and an integer M, find the value of $\sum_{i=1}^{N-M+1} \sum_{j=1}^{N-M+1} F(i,j)$ where $F(p,q) = \sum_{i=0}^{M-1} A_{p+i} * A_{q+i}$

QUICK EXPLANATION

- By a couple of transformations, we can reduce the given expression to $\sum_{i=0}^{M-1} A[i, i+N-M]^2$ where $A[i,j]$ is the sum of elements from i-th element to j-th element both inclusive.
- We can maintain a segment tree with M leaves, each leaf storing values $A[k, k+N-M]$. Each non-leaf node shall store the sum of values as well as the sum of square of values.
- We need to handle range update, updating all leaves which have position pos in its interval while keeping the sum of squares.
- The answer for each query shall be the sum of squares of the root node.

EXPLANATION

The given sum seems a pain, so let's try to simplify it.

We have $S = \sum_{i=0}^{N-M} \sum_{j=0}^{N-M} \sum_{k=0}^{M-1} A_{i+k} * A_{j+k}$

Reordering summations and taking non-dependent terms out.

$$S = \sum_{k=0}^{M-1} \sum_{i=0}^{N-M} A_{i+k} * \sum_{j=0}^{N-M} A_{j+k}$$

Writing the sum of subarray $[l, r]$ as $A[l, r]$, we get

$$S = \sum_{k=0}^{M-1} \sum_{i=0}^{N-M} A_{i+k} * A[k, k+N-M]$$

$$S = \sum_{k=0}^{M-1} A[k, k+N-M] * \sum_{i=0}^{N-M} A_{i+k}$$

$$S = \sum_{k=0}^{M-1} A[k, k+N-M] * A[k, k+N-M]$$

$$S = \sum_{k=0}^{M-1} A[k, k+N-M]^2$$

We get a nice expression via transformations. Hurray! This allows us to solve subtask 1 by computing the above expression using **sliding window** technique.

But update are still a pain. Each update affects many terms of the above summation, we need to handle it better.

Writing $B_k = A[k, k+N-M]$, we need to find $\sum_{k=0}^{M-1} {B_k}^2$ while updating B_k for each update.

In updating position pos, all B_i are affected which contain A_{pos}. By basic observation, we can find that A_p is included in all B_i such that $max(0, p-(N-M) \le i \le min(p, M-1)$.

Now, we have reduced the problem to,
Given an array, handle the following operations

- Update range $[L, R]$ by delta.
- Find the sum of squares of values in the array.

Anyone who have heard of **Segment tree** shall know what's about to happen 😛

For the segment tree, only the push function is a bit different due to squares of terms. Rest details can be found easily.

Let us store the sum of values as well as sum of squares of values at each node. Say the sum of values at node i is S_i and the sum of squares of values at node i is SQ_i.

Suppose each node in the range $[L, R]$ needs to be increased by D. Suppose old values of S_i and SQ_i are given by $sumX$ and $sumX2$ respectively.
The new S_i is given as $sumX + (R - L + 1) * D$
The new SQ_i is given as $SQ_i = \sum_{p=L}^{R} (B_p + D)^2 = \sum_{p=L}^{R} {B_p}^2 + \sum_{p=L}^{R} D^2 + 2 * D * \sum_{p=L}^{R} B_p$

$SQ_i = sumX2 + D^2 * (R - L + 1) + 2 * D * sumX$

Hence, we can push updates in $O(1)$ time, thus solving the problem.

The final sum of squares after each query shall be stored as SQ_{root}

TIME COMPLEXITY

The time complexity is $O(N + M * log(M))$ per test case.

PREREQUISITES:

Data Structures on Trees, Stack, Greedy

PROBLEM:

A "mountain range" can be modeled by a polyline in a 2D plane that passes through N points $p_1 = (1, h_1), p_2 = (2, h_2), \ldots, p_N = (N, h_N)$, in this order, where h_i is a positive integer for each valid i. This polyline divides the set of points $\{(x, z) : 1 \le x \le N\}$ into two regions. The "solid" part of the mountain range is the "bottom" region, i.e. the region that contains the x-axis.

Each point p_i is initially assigned the value a_i. Chef can glide from point to point, and the total value of a journey is just the sum of the values of all points visited. However, there are some restrictions,

- Chef can only glide from a higher den to a strictly lower den, i.e. if he glides from a den at p_i to a den at p_j, then $h_i > h_j$ must hold.
- Chef can only glide in one direction and never look back, i.e. if he starts travelling in the direction of decreasing x, he must keep travelling in that direction until the end of the journey, and if he starts travelling in the direction of increasing x, he must keep travelling in that direction until the end of the journey too.
- Chef cannot travel through solid mountain, i.e. he can only glide from a den at p_i to a den at p_j if the line segment connecting the points p_i and p_j does not intersect the "solid" part of the mountain range. However, it may touch the border of the mountain range, i.e. the polyline itself.

You must process Q queries. There are two types of queries:

- `1 b k`: The value of point p_b should be changed to k instead for all future queries.
- `2 b c`: Find the maximum possible value of all journeys that start at p_b and end at p_c

QUICK EXPLANATION:

Suppose that $b \le c$.

1. It is always optimal to visit as many points as possible on the journey.
2. Suppose we want to plan a journey from $p_b \to \cdots \to p_c$. Let i be the greatest integer less than c such that $h_i > h_c$, i.e. p_i is the point nearest p_c to its left that is higher up than it. Then, it is always optimal to visit p_i, i.e. the journey goes $p_b \to \cdots \to p_i \to p_c$. Call i the "optimal previous" of c.
3. For the journey $p_b \to \cdots \to p_c$, we should start at c, then visit its optimal previous, then visit *its* optimal previous, and so on, until we arrive at b.
4. For each $1 \le c \le N$, connect c with its optimal previous. This creates a rooted forest.
5. If b is not an ancestor of c, then the journey is impossible. Otherwise, the answer is the sum of the a_i of all nodes on the path from c to b.
6. Point updates and path sum queries can be handled by flattening the tree, rudimentary heavy-light decomposition, etc.

For when $b > c$, create another rooted forest that represents motion in the other direction.

EXPLANATION:

To simplify things, let's assume that $b \le c$ for all queries of the second type. So, Chef is always traveling with increasing x-coordinates.

Since all values are positive, a journey from b to c can always be improved if we can squeeze in another point to be visited. So, let's try to determine a point that we *for sure* will visit on the way from $p_b \to \cdots \to p_c$.

Let i be the greatest integer less than c such that $i < c$ and $h_i > h_c$, i.e. p_i is the point nearest p_c to its left that is higher up than it. Call i the "optimal previous" of c. Since no other point between i and c is higher than h_c, we know it is always possible to travel from p_i to p_c.

We call i the optimal previous because we can show that it is always optimal to include p_i in Chef's journey. Consider any optimal journey from p_b to p_c that does not include p_i. We can then improve the score of that journey by first passing through p_i and then ending on p_c.

This reduces the problem then to finding the maximum possible value of all journeys that start at p_b and end at p_i. But then, by the same argument, we know we have to pass through the optimal previous of i, and so on.

So, we now can build the journey with the following algorithm. Start at c, then visit its optimal previous, then visit *its* optimal previous, and so on, until we end up at b. If we don't end up at b, then the journey is impossible.

Each point has a unique optimal previous, if it has one at all. Determining the optimal previous of all points can be done in $\mathrm{O}(N)$ using a stack.

If we connect each point to its optimal previous, we end up defining a rooted tree (or rather, a forest). Checking if it is possible to get from p_b to p_c is a matter of checking if b is an ancestor of c in its tree. Then, the value of the maximum possible journey is simply the sum of the values of all nodes on the path from b to c.

What about when $b > c$? Simple—just create two different forests, one for left-to-right motion, and one for right-to-left motion. Maintain updates on both, and use the appropriate one when querying.

Handling point updates and path sum queries can easily be maintained using your favorite tree data structure—you could flatten the tree, or use Heavy Light, or whatever you want.

The runtime is thus $\mathrm{O}(\log n)$ (with Tree Flattening) or $\mathrm{O}(\log^2 N)$ (with Heavy-Light Decomposition) per query. We are okay with letting $\mathrm{O}(\log^2 N)$ solutions pass if they are optimized (both the setter's Heavy-Light and Tree Flattening solution finish in 0.20s).

I will briefly discuss one method for solving this, which is by flattening the tree, since it is the simpler one of the two.

For our problem, we may get a forest, so create a dummy node 0 that is root of all our trees.

Now, we perform a DFS tree traversal starting from the root. We maintain a global `timer` which starts at 0, and maintain two arrays `entr` and `ext`, corresponding to the **entry** and **exit** time of each node.

- When you enter a node `u`, record `entr[u]=timer;` then increment `timer += 1;`
- When you exit a node `u`, record `ext[u]=timer` then increment `timer += 1;`

Our tree has $n + 1$ vertices, so each of the integers from 0 to $2n + 1$ is associated with a particular node, on either entry or exit. You can imagine tracing the tree traversal with a continuous line, then unwrapping and flattening it out (seriously, draw a picture, it will help).

Now, note that the `entr` and `ext` of all nodes in the subtree of some `u` must appear between `entr[u]` and `ext[u]`, which can be more easily visualized if you have a picture of the traced path.

How can we check if `b` is an ancestor of `c`? That's equivalent to `c` being a descendant of `b`, i.e. `c` exists in the subtree of `b`. So, we just check if `entr[b] <= entr[c] and ext[c] <= ext[b]` (actually it doesn't matter if we use `entr[c]` or `ext[c]` for each check).

How about the path queries? We create an array `flat`, then set `flat[entr[u]] = a[u]` and `flat[ext[u]] = -a[u]`. If `b` is an ancestor `c`, then the sum of all nodes on the path from `b` to `c` is just the sum of `flat[entr[b]]` until `flat[entr[c]]`. Why?

This range contains all the nodes that we visit before `c`. If it is on the path directly between `b` and `c`, then only its `entr` is in the range and **not** its `ext`, because we only exit each of those nodes after we are done with all its descendants (and `c` is one of them). However, if a node is **not** on that path, we must have entered and then

exited it before arriving at `c`. That means **both** its `entr` and `exit` are in the given range, and they will *cancel each other out* when summed.

Here is another way to visualize this. Consider the stack of recursive calls for our DFS. When you enter a node `u`, push `u` onto the stack. When you exit a node `u`, pop `u` from the stack. When you reach `c`, the only nodes on the stack are those on the path from `b` to `c`.

Now, instead instead of pushing `u` onto stack, push `a[u]`. And, instead of popping `u` from the stack, push `-a[u]`. Then, we see that the sum of the values of the nodes on the path from `b` to `c` is equal to the sum of some **consecutive** range of "recursive calls". This range is what `entr[b]` and `entr[c]` keeps track of.

When doing a point update on `u`, make sure to update both `flat[entr[u]]` and `flat[ext[u]]`.

We can maintain point updates and range sum queries with either a segment tree or a Fenwick tree, giving $\mathrm{O}(\log n)$ time per query.

Question: Suppose you should start at `b` but you can choose whatever ending point `c` that you want. How can answer those queries quickly?

Question: What do you do for path queries from `u` to `v` where one is not the ancestor of another?

Question: Why does this trick not work with range max or range min queries?

PREREQUISITES:

Online Algorithms , **Range trees** like **Segment Tree** or **Fenwick Tree** , **Order Statistics Tree** (like GNU Policy Based Data Structure (PBDS) in C++) or various Decomposition techniques.

PROBLEM:

You are given with a sequence A consisting of N integers denoting unique IDs. You have to perform Q queries (online) in which you have to insert a unique integer into A and provide rewards equal to the number of elements in A smaller than this currently inserted unique integer. Find the total number of rewards provided after all queries are performed.

EXPLANATION:

The first sub-task can be solved with a naive implementation where each query is processed in $O(N)$ time by linearly traversing A each time to count the integers smaller than the current cat's worthiness and hence each test is solved in $O(N.Q)$ time. The constraints are small enough for this sub-task, so this solution is viable.

The real challenge here is the second sub-task which will, again, force us to use some wonderful data structures to solve this online problem. This problem can be solved using various different strategies as mentioned in the *Prerequisites* part of this editorial. The solution discussed in this editorial is based on Binary Indexed Tree (Fenwick Tree) implementation.

Fenwick tree is a range tree which supports logarithmic time range sum queries (which are used in the solution). According to the constraints, the worthiness of each cat lies in the range $(1 \leq W \leq 10^9)$. So, a Fenwick tree can be constructed to store range sums for all unique IDs where each unique ID has a count of 1 and hence we can find the range sum for the range $[1, W]$ for each cat before it's insertion in A, which will give us the number of IDs less than it's ID and hence the reward. Summing up all the rewards will finally lead us to the ultimate answer for each test.

For a space-time (not Einstein's!) efficient implementation and ability to support dynamic (online) insertions efficiently in the Fenwick tree (because W is large), **Hash Map** data structure has been used. So, we'll be able to process each query in $O(\log_2 W)$ time which is fast enough.

SOLUTION:

Setter's Solution

```
#include <bits/stdc++.h>

using namespace std;

constexpr long MAXW = 1'000'000'001L;

unordered_map<long, long> fenwick;

inline void add(long x, long y) {
    while (x < MAXW) {
        fenwick[x] += y;
        x += x & -x;
    }
}

inline long get(long x) {
    long res = 0L;
    while (x) {
        res += fenwick[x];
```

```
        x -= x & -x;
    }
    return res;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
    int tt;
    cin >> tt;
    while (tt--) {
        int n;
        cin >> n;
        fenwick.clear();
        for (int i = 0; i < n; i++) {
            long x;
            cin >> x;
            add(x, 1L);
        }
        int qq;
        cin >> qq;
        long ans = 0L;
        while (qq--) {
            long x;
            cin >> x;
            x ^= ans;
            ans += get(x);
            add(x, 1L);
        }
        cout << ans << '\n';
    }
    return 0;
}
```

ALTERNATE SOLUTION:

This problem can be solved with lesser code using GNU PBDS in C++ which is an ordered set implemented using Order Statistics Tree. It supports operations like `order_of_key` which tells the expected position (0-indexed) of an element in sorted order in the set. And as all the IDs are unique so `order_of_key` for an element will tell the number of elements currently in the set which are smaller than it. This operation works in $O(\log_2 X)$ time where X is the current size of the set.

So, this solution runs in $O((N+Q)\log_2(N+Q))$ time per test with $O(N+Q)$ auxiliary space.

Alternate Solution

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>

using namespace std;
using namespace __gnu_pbds;

template <typename T, typename C = less<T>>
using indexed_set = tree<T, null_type, C, rb_tree_tag, tree_order_statistics_node_update>;

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
```

```
    cout.tie(nullptr);
    int tt;
    cin >> tt;
    while (tt--) {
        int n;
        cin >> n;
        vector<int> a(n);
        for (int i = 0; i < n; i++) {
            cin >> a[i];
        }
        indexed_set<long> s(a.begin(), a.end());
        int qq;
        cin >> qq;
        long ans = 0L;
        while (qq--) {
            long x;
            cin >> x;
            x ^= ans;
            ans += s.order_of_key(x);
            s.insert(x);
        }
        cout << ans << '\n';
    }
    return 0;
}
```

COMPLEXITY:

Time complexity: $O((N+Q)\log_2 \max\{W\})$ per test.
Space Complexity: $O(N+Q)$ auxiliary space per test.

Feel free to share your approach. If you have any queries, they are always welcome.

PREREQUISITES:

Point update/range query structures (**segment trees**/**fenwick trees**), basic combinatorics

PROBLEM:

There are N rocks, the i-th of which is at position X_i and has weight W_i.

Rocky will eat a subset of these rocks starting from the one with least weight, then moving to the one with second least weight, and so on.
Find the expected distance travelled by Rocky if any subset is equally likely to be chosen.

EXPLANATION:

First off, each subset has an equally likely chance of being chosen, and there are 2^N subsets.
So, to find the expected value, it's enough to compute the total distance traveled by Rocky across all subsets, and then divide this by 2^N.

As is common with several problems of this type (i.e, find the sum of something across all subsets), instead of fixing a subset and computing the desired quantity for it, we can fix a quantity and see how many subsets it occurs in.

In our case, the quantity we're interested in the distance between points, so let's look at that.
In particular, we care about distances between *adjacent* (in terms of weight) points in the chosen subset; since the total distance covered is the sum of these distances.

This quickly gives rise to a solution in $\mathrm{O}(N^2)$, as follows:

- Let's fix two indices i and j (without loss of generality, say $W_i < W_j$), and count the number of subsets in which Rocky travels directly from X_i to X_j.
- In order for this direct travel to happen, the chosen subset cannot contain any weights lying strictly between W_i and W_j.
  On the other hand, any weights $< W_i$ or $> W_j$ can be freely chosen.
- This gives us $2^{W_i-1+N-W_j}$ subsets in which this direct travel happens, each for a distance of $|X_i - X_j|$.

With i and j fixed, this is an $\mathrm{O}(1)$ (or $\mathrm{O}(\log MOD)$) computation, and we have our quadratic solution.

To optimize this, let's fix the value of j, and look at the structure of all i that we need to compute for.

For now, only consider $X_i < X_j$; the other side can be computed similarly.

For a fixed i, the contribution is $(X_j - X_i) \cdot 2^{W_i-1+N-W_j}$.
Here, 2^{N-W_j} is a constant independent of i, so let's ignore it for now and multiply it in the end.
The remaining part can be split into $(X_j \cdot 2^{W_i-1})$ and $-(X_i \cdot 2^{W_i-1})$.

We want the sum of this across all i such that $X_i < X_j$ and $W_i < W_j$.
Notice that all we really need is the sum of all 2^{W_i-1} and the sum of $X_i \cdot 2^{W_i-1}$, which are both quantities depending purely on i.

This looks something like a 2D range query, but it can in fact be done with a simple segment tree.

Let's iterate over values of W_j from 1 to N (note that we're iterating from small to large *weight*, not *index*).
Doing this automatically satisfies the constraint $W_i < W_j$ whenever we do a query, so we only need to take care of the $X_i < X_j$ condition.

This means we can simply query for all indices $< j$ (since the input guarantees that $X_1 < X_2 < \ldots < X_N$).
Knowing the 2^{W_i-1} and $X_i \cdot 2^{W_i-1}$ values for these indices, this reduces to a simple range query.

In order to ensure that we don't compute these values for indices with weights $> W_j$, set the values at all indices to be zero initially, and update an index's value only *after* it has been processed as a W_j.
This way, when doing the range sum query, the contribution of higher weight indices is 0, which is what we want.

$X_i > X_j$ can be handled similarly, the range query is on a suffix instead of a prefix.
Don't forget to divide by 2^N in the end.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PREREQUISITES:

BIT/Segment Tree, LCA, Euler Tour on trees, Sorting

EXPLANATION:

Quick Explanation:

Sort the queries and the values in the ascending order. We will solve queries offline. Do an ETT over the tree and calculate IN and OUT for each node. We will use BIT to answer queries.

Initially, all nodes are 0. Let's Answer the query in ascending order.

For each query, all nodes with value less than equal to w of that query are added to the BIT(positive value at IN of that node and negative value at OUT). After no such nodes are left, We can answer the query. Bit query of IN[u] will give us summation over node 1 to u. Note that we have only added values less than W of the current query. Use LCA and some basic maths to answer the query.

Detalied explanation:
To be updated.

PREREQUISITES:

Euler Tour , Bit/segment Tree

QUICK EXPLANATION:

Find the in time and out time for each node (village) using Euler tour. Take a $temp$ array and store positive value of each node at index $=$ it's in time and negative value at index $=$ it's out time. Use Bit or segment tree on $temp$ to answer the queries. For type 1 query, just find range sum on $temp$ from index 0 to $out[x] - 1$. For type 2 query, update index $in[x]$ by $+u$ and $out[x]$ by $-u$. Keep the value of node 1 zero throughout the queries. Skip all its updates.

EXPLANATION:

The problem is a trivial range query and point update question which can be easily solved using segment tree or Fenwick tree (Bit). We just need to convert the tree into an array and then use Bit or segment tree on it.
We use the Euler Tour technique to store the in time and out time of each node. We will use these values as indexes. In the $temp array, at in value of each node, we will store the positive value of the node at the out time, we store the negative value of the node.
Realize that the answer to the queries is as follows:
For type 1 query, just find range sum on $temp$ from index 0 to $out[x] - 1$. For type 2 query, update index $in[x]$ by $+u$ and $out[x]$ by $-u$.
The implementation is very trivial. We just have to be careful about node 1. We keep its value 0 throughout the queries and skip all updates on it.

PREREQUISITES:

Depth first search, Segment/fenwick trees, Binary search

PROBLEM:

You have a tree T, each node of which may or may not be activated.
Each node also has an energy, initially 0.
At time i, each activated node gives i energy to itself and its neighbors.

Answer Q queries of the form (u, T, K), with the answer being the minimum time after which at least T nodes on the path $1 \to u$ have an energy of $\geq K$.

EXPLANATION

This editorial will be for the hard version of the problem, with $K \leq 10^{18}$.

Let's focus on a single query (u, T, K). How to solve it?

To do that, let's look at how the energy of a vertex changes.
For vertex x, let m_x be the sum of A_v across all vertices v such that $v = x$ or v is a neighbor of x.
Then, vertex x will receive exactly $m_x \cdot i$ energy at time i.
In particular, at time i, vertex x will have $m_x \cdot (1 + 2 + \ldots + i) = m_x \cdot i \cdot (i + 1)/2$ energy.

So, suppose we know the m_x values for all vertices. To answer a query (u, T, K):

- First, note that the number of vertices with energy $\geq K$ is a non-decreasing function of time, so we can binary search on it to find the first time this number exceeds T.
  Be careful when choosing the upper limit of the binary search
- This requires us to be able to quickly calculate the number of nodes on the $1 \to u$ path that have $\geq K$ energy at a given time t.
- This means we want the count of nodes x such that $m_x \cdot t \cdot (t + 1)/2 \geq K$.
  t is a constant so by bringing this to the other side we get an equation of the form $m_x \geq c$ for some constant c.
- Keeping all the values of m_x from the root to u in an appropriate data structure (for example, a segment tree) allows us to find this count in $\mathrm{O}(\log N)$.
- So, a single query can be solved in $\mathrm{O}(\log N \log M)$ where $M = 10^9$.

This solves a single query. Now we extend this to multiple queries.

Note that a query at u only depends on the m_x values on the $1 \to u$ path. So, we can do the following and solve queries offline:

- Group queries by their u.
- Run a dfs on the tree.
- When you enter a vertex u, insert m_u into the data structure you are using.
- Then, solve all queries involving u.
- Next, continue on with the dfs to u's children.
- Finally, when exiting u, remove m_u from the data structure.

You can see that this process ensures that whenever answering a query at u, the data structure contains only those m_x values for x on the $1 \to u$ path, which is exactly what we want.

It is also possible to solve the queries online (for example, with a persistent segment tree).

PREREQUISITES:

Depth first search, Segment/fenwick trees, Binary search

PROBLEM:

You have a tree T, each node of which may or may not be activated.
Each node also has an energy, initially 0.
At time i, each activated node gives i energy to itself and its neighbors.

Answer Q queries of the form (u, T, K), with the answer being the minimum time after which at least T nodes on the path $1 \to u$ have an energy of $\geq K$.

EXPLANATION

This editorial will be for the hard version of the problem, with $K \leq 10^{18}$.

Let's focus on a single query (u, T, K). How to solve it?

To do that, let's look at how the energy of a vertex changes.
For vertex x, let m_x be the sum of A_v across all vertices v such that $v = x$ or v is a neighbor of x.
Then, vertex x will receive exactly $m_x \cdot i$ energy at time i.
In particular, at time i, vertex x will have $m_x \cdot (1 + 2 + \ldots + i) = m_x \cdot i \cdot (i + 1)/2$ energy.

So, suppose we know the m_x values for all vertices. To answer a query (u, T, K):

- First, note that the number of vertices with energy $\geq K$ is a non-decreasing function of time, so we can binary search on it to find the first time this number exceeds T.
  Be careful when choosing the upper limit of the binary search
- This requires us to be able to quickly calculate the number of nodes on the $1 \to u$ path that have $\geq K$ energy at a given time t.
- This means we want the count of nodes x such that $m_x \cdot t \cdot (t + 1)/2 \geq K$.
  t is a constant so by bringing this to the other side we get an equation of the form $m_x \geq c$ for some constant c.
- Keeping all the values of m_x from the root to u in an appropriate data structure (for example, a segment tree) allows us to find this count in $\mathrm{O}(\log N)$.
- So, a single query can be solved in $\mathrm{O}(\log N \log M)$ where $M = 10^9$.

This solves a single query. Now we extend this to multiple queries.

Note that a query at u only depends on the m_x values on the $1 \to u$ path. So, we can do the following and solve queries offline:

- Group queries by their u.
- Run a dfs on the tree.
- When you enter a vertex u, insert m_u into the data structure you are using.
- Then, solve all queries involving u.
- Next, continue on with the dfs to u's children.
- Finally, when exiting u, remove m_u from the data structure.

You can see that this process ensures that whenever answering a query at u, the data structure contains only those m_x values for x on the $1 \to u$ path, which is exactly what we want.

It is also possible to solve the queries online (for example, with a persistent segment tree).

PREREQUISITES

Dynamic Programming, Segment Tree

PROBLEM

You have an array A containing all the natural numbers from 1 to N twice. Find the number of **ordered pairs** of **disjoint** subsequences (P, Q) of A such that they are equal.

EXPLANATION

Let,

- $dp_i :-$ denote the number of **ordered pairs** of subsequences (P, Q) that can be formed taking elements from index 0 to i.
- $f_i, s_i :-$ denote the indices of the first and the second occurrence of A_i.
- $t_i :-$ denote the number of **unordered pairs** of subsequences (P, Q) ending at indices f_i and s_i respectively.

Now, let's say we have already computed dp_{i-1}.

Case 1: If $i = f_i$,

No additional pair of (P, Q) can be formed by including the element at index i. So we can simply say, $dp_i = dp_{i-1}$.

Case 2: If $i = s_i$,

We have to add all the ordered pairs of subsequences (P, Q) ending at f_i and i to dp_{i-1}. So we can say,

$dp_i = dp_{i-1} + 2 \cdot t_i$.

Let's try to find t_i now. This comprises of 3 parts:

- **Part 1:** We can add A_{f_i} and A_i respectively at the end of all the **ordered pairs** of subsequences (P, Q) formed from indices 0 to $f_i - 1$.
- **Part 2:** We can add A_{f_i} and A_i at the end of all the **unordered pairs** of subsequences (P, Q) which end at f_x and s_x repectively, such that, $0 \le f_x < f_i < s_x < i$.
- **Part 3:** An additional subsequence pair $([A_{f_i}], [A_{s_i}])$.

The **first part** is simply dp_{f_i-1} and as f_i is the first occurrence of A_i, we can also say $dp_{f_i-1} = dp_{f_i}$ *(Case 1)*.

Now, for the **second part**, you have f_i and i as fixed and you need to sum all the values of t_x where, $f_x < f_i < s_x < i$. So, let's say we have maintained an array T (say) till index $i - 1$ with the value t_x at the index of the first occurrence and $-t_x$ at the index of the second occurrence of all the elements having $s_x \le i - 1$.

The sum of this array T from $j = 0$ to $j = f_i - 1$ will give us the value of the second part because in case of $s_x < f_i$, the values t_i and $-t_i$ are simply going to cancel each other out.

For the condition of $s_x < i$ to be fulfilled at all times, we will update the array T only as we iterate over i.
We **cannot** pre-compute the values of T, otherwise, the summation will also include values where $f_x < f_i$ but

$s_x > i$.

Mathematically,

$$T \begin{cases} T_{f_x} = T_{s_x} = 0, & \text{if } i-1 < s_x \\ T_{f_x} = t_x \text{ and } T_{s_x} = -t_x, & \text{if } i-1 >= s_x \end{cases}$$

and, $t_i = dp_{f_i} + \sum_{j=0}^{j=f_i-1} T_j + 1$.

Finally, we can update the array T as follows:
$T_{f_i} = t_i$ and $T_i = -t_i$

These range sum and point update queries in T can be handled efficiently using a Segment Tree.

Note: No updates in T were required in **Case 1** as $i < s_i$.

PREREQUISITES:

Binary Indexed Trees/Segment Trees
Mathematics and Number Theory

PROBLEM:

Given fixed parameters R, p1, p2 (p1,p2 are primes) and array A, perform three operations:

* 0 S D X Y: Add an AGP with the start term of S, the common difference of D, common ratio of R from the X-th to the to Y-th element of A.

* 1 X g: A[X] = $(A[X])^g$ modulo p2.

* 2 X Y: Output (A[X] + … + A[Y]) modulo p1.

Array A can have upto 10^5 elements and upto 10^5 queries.

EXPLANATION:

We can see AGP as an infinite AGP. To add an AGP from [X,Y] is equivalent to adding 2 infinite AGP's which are:
(a) Start Term = S, Common Difference = D and Starting from X.
(b) Start Term = −(S + (Y − X + 1) * D) * R^{Y-X+1}, Common Difference = −D * R^{Y-X+1} and starting from Y+1.

Now the solver can use any data structure such as BIT/segment tree. But since BIT are easy to implement and efficient, I will discuss the approach using BIT.

Supose that using BIT, we somehow solve queries 0 and 2, how can we solve query 1 efficiently. Here we keep two BIT, one is modulo p1 (say BIT1) and other modulo p2 (say BIT2). For query 1 ie. 1 X g

```
// read respective values at X
var1=read(BIT1,X)
var2=read(BIT2,X)

// update to make both indexes 0
update(BIT1,X,-var1)
update(BIT2,X,-var2)

// update to add new values
update(BIT1,X,var2^g mod p2)
update(BIT2,X,var2^g mod p2)
```

Now, we have to answer queries 0 and 2. Let us analyse the situation after just one addition of query 0 (which is added at X, as we have broken 1 AGP into 2 infinite AGp's).

Let us find sum of all elements from 1 to J.

$$Sum = S + (S + D) * R + ... + (S + (J - X) * D) * R^{J-X}$$

Multiplying with R, we get

$$Sum*R = S*R+...+(S+(J-X-1)*D)*R^{J-X}+(S+(J-X)*D)*R^{J-X+1}$$

Subtracting above 2 equations, we get

$$Sum*(R-1) = -S-D*(R+R^2+...+R^{J-X})+(S+(J-X)*D)*R^{J-X+1}$$

$$Sum*(R-1) = -S-\frac{D*R*(R^{J-X}-1)}{R-1}+(S+(J-X)*D)*R^{J-X+1}$$

Simplifying, we get

$$Sum * (R-1)^2 = -S*(R-1) - D*(R^{J-X+1} - R) + (S*R^{J-X+1}*(R-1)) + (J-X)*D*R^{J-X+1}*(R-1)$$

Let $BADD = -S*R + S + D*R$

Let $T = -D*R^{-X+1} + S*R^{-X+2} - S*R^{-X+1} - X*D*R^{-X+2} + X*D*R^{-X+1}$

Let $Z = D*R^{-X+2} - D*R^{-X+1}$

Note:

(a) BADD, T and Z are independent of query point J.
(b) $Sum * (R-1)^2 = BADD + T*R^J + Z*J*R^J$

By using Fact (a), we can store BADD, T and Z in the BIT (as it is independent of query point) in the update query 0 and use fact (b) for query of type 2.

Solution is not yet complete as we need to augment our data structure with yet another field ADD, which keeps the constant addition of the value in the index. This is used intially and for query 1. So, the output of query 2 changes from (b) to:

(c) $Sum = ADD + (BADD + T*R^J + Z*J*R^J) * INVERSE((R-1)^2))$

Wait, the solution is not yet over! What if $(R-1)^2$ is not coprime to prime numbers, then we won't be able to find INVERSE of it and hence our solution will fail. In this case, the problem transforms to AP as R mod prime = 1 mod prime.

Solve for AP in the similar idea using data structures.

Complexity of solution is O(N * log(N)) intially, plus O (log(N)) for each query. So overall complexity is O((N+Q)* log(N)).

See setter's solution for implementation details using the above idea.

ALTERNATE SOLUTION:

(as told by @triveni)
There was no need of computing any kind of inverse modulo or such things at all to solve this problem, using the method that I followed. Even if p1 and p2 are not primes, this problem can be solved very easily using the same approach. I used Segment Tree data Structure. In my opinion, the hardest part to solve this problem is to find the sum of the AGP in efficiently. It is said in the editorial, we need to compute Modulo((R-1)**2).

But actually we can do something better here.

sum(i) = S + (S+D)*R + (S+2D)*R^2 + (S+3D)*R^3 +…+ (S+(i-1)*D)*R^(i-1)

=>sum(i) = (S + S*R* + S*R*^2 + S*R^3* + S*R*^4 +…) + (0 + D*R* + 2*D*R^2 + 3D*R^3* + 4*D*R^4 +…)

=>sum(i) = S(1 + R + R^2 + R^3 + R^4 +…) + D(0 + R + 2*R^2* + 3R^3 + 4R^4 +…)

=>sum(i) = S * X(i) + D * Y(i)

Now, what one can do is pretty simple. Just pre-compute the value of X(i) and Y(i) for all 1<=i<=N.

Hence, it becomes pretty easy. Here is my solution using segment tree approach [CodeChef: Practical coding for everyone].

This method doesn't assume anything about primeness of p1 and p2.

PREREQUISITES:

Segment Tree

PROBLEM:

You are playing a game, where entering at time l and exiting at time r costs $k \cdot (r - l)$ coins. You can enter and exit the game at most once.
There are n items in the game, the i th of them is valued at v_i coins, and can only be collected if $l \le x_i \le y_i \le r$.
What is the maximum profit you can make?

QUICK EXPLANATION:

- It is optimal to enter only at some x_i, and exit at some y_j.
- Consider all possible y_j in increasing order, and for each of them compute the maximum profit. The final answer is the maximum over all of these.
- Calculating the maximum profit can be done with any data structure supporting range max query and range sum updates, for example a segment tree.

EXPLANATION:

First, note that it is optimal to exit only at some y_i - staying any longer would incur a cost of k coins for each unit of time, which is not optimal.
The same applies to entering - it is optimal to do so at some x_i.

Subtask 1
$n \le 20$, so an $\mathrm{O}(n \cdot 2^n)$ bruteforce over all possible subsets of items to choose will work fast enough.
Given a fixed non-empty set of items $(x_{i_1}, y_{i_1}, v_{i_1}), \ldots, (x_{i_r}, y_{i_r}, v_{i_r})$, let $L = min(x_{i_1}, x_{i_2}, \ldots, x_{i_r})$ and $R = max(y_{i_1}, y_{i_2}, \ldots, y_{i_r})$. The maximum profit obtained while getting all of them is $v_{i_1} + v_{i_2} + \ldots + v_{i_r} - k \cdot (R - L)$.
Compute this value over all possible subsets and print the maximum of this and 0 (where we enter and exit at the same point and pick up nothing)

Subtask 2
Suppose, for the moment, that all the y_i are distinct.
Let us sort the items by y.
Now, for each $1 \le i \le n$,
Suppose we want to exit at y_i. What's the maximum profit possible?
Given that the items are sorted by increasing y, exiting at y_i means that it's impossible to pick up any of items $i+1, i+2, \cdots, n$, so we only need to worry about items $1, 2, \cdots, i$.
Now suppose we enter at time x. What's the maximum profit?

- The cost of entering and exiting is $k \cdot (y_i - x)$
- For each $1 \le j \le i$, we can add v_j to the profit if $x_j \ge x$ (because we know that $y_j \le y_i$).

But, as mentioned before, it's enough to consider x to be one of the x_j where $j \le i$, because entering at any other time cannot be optimal.
This gives an $\mathrm{O}(n^3)$ solution: for each $1 \le i \le n$, iterate over $1 \le j \le i$ and find out in $\mathrm{O}(n)$ the maximum profit if we enter at x_j and leave at y_i. However, this is not fast enough to pass this subtask yet.

One can observe that this solution can be improved to $O(n^2)$ by noting that, when we fix y_i, it's enough to traverse the x 's we have, in descending order. Doing that allows us to keep a running sum of the v_j which are possible to get, without needing a third loop to do so.
This can be implemented by, for example, keeping a sorted (multi)set of pairs (x_i, v_i). Iterating over this in descending order is easy, and insertion is $\mathrm{O}(logN)$, leading to an overall $O(n^2)$ solution.

How to deal with the fact that the y_i might not be distinct? Simply ignore it!
Even if they're not all distinct, the above process will give the correct answer, simply because the maximum profit possible when exiting at any y will be calculated correctly when the last item with $y_i = y$ is being processed.

Subtask 3
Quadratic is too slow here. Let's look at what we're actually computing.
The profit for entering at x and leaving at y_i is given by $-k \cdot (y_i - x) + \sum_{\substack{1 \le j \le i \\ x_j \ge x}} v_j$

Rewrite this as $-k * y_i + k * x + \sum_{\substack{1 \le j \le i \\ x_j \ge x}} v_j$.

$k * y_i$ is a constant depending only on y_i, so we can ignore it for now.
What about the rest? It depends purely on x, so if we were able to update this value for each x after processing an item, maybe we could do something to get the maximum of them all.
And that's exactly what we will do!

Suppose we had a really large array A, indexed by the integers -10^9 to 10^9.
Initially, let $A[i] = k * i$.
Whenever we process an item (x, y, v), do a range update and add v to the range $[-10^9, x]$
Now when we fix y_i and starting point x, what's the cost of entering at x? It is exactly $A[x] - k * y_i$
So, our problem reduces to finding the maximum $A[x]$ where $x \le y_i$ (equivalently, the max on the range $[-10^9, y_i]$), and being able to add a value to a range.
A segment tree with lazy propagation supports both these operations in logarithmic time, so that is our data structure of choice.
(Don't know how to perform these operations using a segment tree? **Codeforces EDU** is for you)

We are left with only one issue now: the initial array. It is way too big for us to explicitly create it, even if all subsequent operations on it are fast.
How to overcome this?

Editorialist's Approach

The editorialist used a dynamic segment tree/implicit segment tree (**click** for more details), i.e, a segment tree where the only nodes created are the ones used in some query or update.
Since the only points we care about querying are the x_i, create an implicit segment tree with the values of all points set to $-\infty$. Then, for all x_i, set the value at x_i to be $k \cdot x_i$.
After this, update and query the tree as mentioned above, and the problem is done.

Setter's Approach

The setter chose a different approach, and instead used coordinate compression. Map the smallest x_i to 1, the second smallest to 2, and so on. This maps all relevant points to $[1, 10^5]$, and now we can build a normal segment tree on this array and update/query it as necessary. Finding the range to query/update requires binary search on the compressed coordinates.

Both methods use the fact that we only care about $\mathrm{O}(N)$ points.

TIME COMPLEXITY

$\mathrm{O}(N \log M)$ or $\mathrm{O}(N \log N)$ depending on implementation, where $M = 2 * 10^9$

PREREQUISITES:

Segment Tree, Lazy Propagation, Euclid's GCD, Two-Pointers

PROBLEM:

You are given an array A of size N and Q queries each of which is of following type:

- `l r x y` Replace the `A[l+i]` with `x + i * y` for each $0 \le i \le r - l$

The beauty of an array is given by the maximum of $\gcd(A_l, A_{l+1}, \ldots, A_r).(r - l + 1)$ for all $0 \le l \le r \le n$, i.e. compute for each sub-array the product of its gcd and length and then take the maximum over all values computed for every sub-array.

Print the beauty of the given array and then the beauty of the array after performing the first i queries for each $i = 1, 2, \ldots Q$.

QUICK EXPLANATION:

Given any two numbers A, B, either $\gcd(A, B) = \min(A, B)$, otherwise $\gcd(A, B) \le \frac{\min(A,B)}{2}$.
Therefore consider all the sub-arrays starting from some index i, there will be only $O(\log C)$ where $C = 10^9$ distinct $\gcd$ values among all the sub-arrays beginning at index i.

To calculate the maximum over all sub-arrays, we can use the divide and conquer approach.
To merge the left and right halves, we need to iterate through the suffix of the left half and the prefix of the right half.
By compressing those prefixes and suffixes with the same $\gcd$, we need to only iterate through $O(\log C)$ different sub-arrays in the left and right halves and so merging can be done by considering each pair, finding the cost of the combined arrays and updating the answer. The time complexity per merge is $O(\log^2 C)$.

To accommodate the queries, we can build a segment tree with lazy updates over this D&C solution which immediately leads to an $O(N \log N \log^2 C)$ solution, which can be a bit difficult to fit into the time limit.

The clever observation to optimize the above is that while merging two adjacent sub-arrays into one, if the $\gcd$ changes, i.e. the $\gcd$ is lesser than the $\gcd$ of both the sub-arrays then the cost of the combined sub-array is not greater than the cost of both the sub-arrays considered individually. Thus we need only find those (suffix, prefix) pairs that have the same $\gcd$ value. Since the prefix and suffixes have $\gcd$'s in sorted order we can use a two-pointer approach to find them all and implement the merge procedure in $O(\log C)$ time. This gives us an $O(n \log n \log C)$ algorithm which can comfortably fit into the time limit.

EXPLANATION:

Let's first consider the problem without queries, which is much easier.

On the GCD operation

The first observation is that when applying the $\gcd$ operation on two numbers, the result is either equal to the minimum or is at most half of the minimum.
This is because $\gcd(A, B)$ is a factor of both A, B. If it is not equal to B, then it must be a proper factor of B. A proper factor of B will be lesser by at least one prime factor, and the smallest of all primes is 2.

Remember that the $\gcd(A, B)$ operation takes up to $O(\log B)$, actually we can even write it as $O(\log \frac{B}{g})$ where g is the result of the $\gcd$ operation. This will come in handy when analyzing the runtime of the algorithms that we develop.

Divide and Conquer strategy

In many cases when we want to minimize a function over all sub-arrays, it might be useful to think about a Divide and Conquer style algorithm. If the merge part can be done efficiently, then we get a nice runtime complexity and another hint that such an algorithm is applicable is the presence of range queries in the question. Such Divide and Conquer solutions can (in most cases) be adapted to construct a segment tree with efficient point/range updates and point/range queries.

Following the Divide and Conquer strategy, we split the array into two halves, compute the answer for each and then merge them. We will discuss here how to merge efficiently.

To combine, we need to only consider those sub-arrays that cross the midpoint of the array, which can be represented as a union of a suffix of the left half and a prefix of the right half.

In each half, the prefixes and suffixes have only $O(\log C)$ distinct $\gcd$ values, and we only need the sub-arrays with maximum length among those distinct values. So we can compress them.

To merge, if we naively consider every possible prefix and suffix, we will have $O(\log^2 C)$ distinct pairs of sub-arrays to consider. Also note that if we fix the prefix of the right half and iterate through the suffixes of the left half, the total cost of performing the $\gcd$ operation is $O(\log C)$ total for each prefix.

This is because if the $\gcd$ values are $g_1, g_2, \dots g_k$ then the total number of operations is $O(\log \frac{C}{g_1} + \log \frac{g_1}{g_2} + \dots \frac{g_k}{1}) = O(\log C)$.

Another way to see this is by tracking the modulo operations done while applying the $\gcd$ algorithm. The modulo operation can only be applied $\log(C)$ times until the value reduces to 0 and so the total cost of performing all the $\gcd$'s for a fixed prefix is $O(\log C)$.

So in total, the complexity of the merge operation is $O(\log^2 C)$.

Segment Tree Implementation Details

If we store the answers computed for each divide and conquer iteration, along with the compressed prefixes and suffixes, we get a segment tree and we can perform the range updates by using lazy propagation.

More formally for each node in the segment tree, we maintain the following

- The set of distinct prefix $\gcd$'s and their maximum lengths (stored and $(\gcd, \text{length})$ pairs)
- The set of distinct suffix $\gcd$'s and their maximum lengths (stored and $(\gcd, \text{length})$ pairs)
- The maximum beauty of all sub-arrays which are completely within the interval of the node of the segment tree

And then to merge the left node L and the right node R and obtain the resultant node S, we do the following:

- Set prefix of S as the prefix of L and extend it by visiting the prefixes of R
- Set suffix of S as the suffix of R and extend it by visiting the suffixes of L
- Set maximum beauty as the maximum of beauties of L and R
- For each suffix of L and prefix of R, compute the result of combining them and update the maximum beauty

Range Updates

The range update is change a sub-array of the array to an arithmetic progression. Notice that for an arithmetic progression, $A, A+D, A+2D, \dots A+xD$, the $\gcd$ of any sub-array of size 2 will be $\gcd(A+iD, A+(i+1)*D) = \gcd(A+iD, D) = \gcd(A, D)$, therefore, the $\gcd$ of any sub-array of size at least 2 is also $\gcd(A, D)$.

So the distinct prefixes are $\{(A, 1), (\gcd(A, D), sz-1)\}$. Be careful of the case when $sz = 1$.
Similarly, distinct suffixes can also be obtained.

In the usual style of performing segment tree with lazy updates, we will find the $O(\log n)$ disjoint intervals that make up the given update range $[l, r]$ and we can simply forget about the values stored previously and update them with the new values calculated above. We can mark the children as lazy and store the arithmetic progressions that have to be added there. Refer to the code below for more details.

Complexity analysis

Let's analyze the time complexity of each merge operation.

For extending the prefix and suffix of the left and right nodes, suppose we ended up with k different $\gcd$ values, namely $g_1, g_2, \dots g_k$, then we performed in total k called to $\gcd$ function (i.e. Euclid's algorithm), however they are all factors of each other. Therefore the total cost of performing $\gcd$ for all of them is $O(\log \frac{C}{g_1} + \log \frac{g_1}{g_2} + \dots \frac{g_k}{1}) = O(\log C)$.

Similarly, if we fix the suffix in the left node L and iterate through the prefixes of R, we will take only $O(\log C)$ operations for each suffix, this gives a total of $O(\log^2 C)$ complexity for each merge operation.

Combined with the segment tree updates the total complexity of this approach is $O(N \log N \log^2 C)$ which can be difficult to fit into the constraints. But the bottleneck is the merge operation above, which we will now optimize.

Optimizing the merge operation

Suppose we have two adjacent sub-arrays with $(\gcd, \text{size})$ pairs as (g_1, L_1) and (g_2, L_2) respectively, then the combined sub-array has $(\gcd(g_1, g_2), L_1 + L_2)$. Suppose $\gcd(g_1, g_2) < \min(g_1, g_2)$, i.e. we get a new distinct $\gcd$ by combining the two, then the beauty of the resulting array is $\gcd(g_1, g_2) * (L_1 + L_2)$.

Suppose $L_1 \le L_2$, then $\gcd(g_1, g_2) * (L_1 + L_2) \le \frac{g_2}{2} 2 * L_2 = g_2 L_2$. So merging two sub-arrays with different $\gcd$'s can be avoided and since the $\gcd$ is decreasing across each suffix and prefix, we can find all pairs whose $\gcd$'s are equal by using a two-pointer approach in $O(\log C)$.

With this idea, our solution is $O(N \log N \log C)$ which can comfortably fit into the time limit.

TIME COMPLEXITY:

TIME: $\mathrm{O}(N \log N \log C)$
SPACE: $\mathrm{O}(N)$

PREREQUISITES:
[Binary Search](), [Segment Tree]()

PROBLEM
Given with two arrays $a[]$ and $cost[]$ of length N. $cost[i]$ is the cost of incrementing/decrementing $a[i]$ by 1. You will be provided with Q queries.

$i\ x\ y \rightarrow$ This updates value of a[i] with x and $cost[i]$ with y.

After each query you have to output the minimum cost to make all elements in the array equal by performing the increment/decreament operation on any element any number of times.

As answer could be large output the answer modulo $10^9 + 7$.

SOLUTION:

Hint 1

Think for the same problem, assuming cost of incrementing/decrementing is always 1.

Hint 2

Find if there is a specific point, such that cost of equalizing all its elements will be minimum every time.

Hint 3

Think of reducing this Problem to a problem where cost of incrementing/decrementing is always 1.

Solution

We can observe that, if cost of unit incrementing/decrementing values is always 1, then, we will get the total cost minimized if we make all elements equal to the median of the array.

Now, we have to reduce this problem, to a problem similar to the one where cost for unit incrementing/decrementing is always 1.

To achieve this, we can have $cost[i]$ occurrences of $a[i]$ in the array and make cost for every element as 1. This is equivalent to the original problem as it will ensure that cost of unit incrementing/decrementing a specific value x will be

(cost of unit incrementing/decrementing a single element) * (number of occurrences of x).

In the reduced problem, we know that, cost of incrementing/decrementing a single element by 1 is **1** and number of occurrences of x in the array is **cost[i]**. Therefore, the cost of unit incrementing/decrementing a specific value x will be **cost[i]** which is indeed same as the original problem.

Now, our problem is reduced to find the median of the modified array (which contains $cost[i]$ occurrences of $a[i]$). But we can observe that the size of such modified array can be very large. Therefore, we cannot simply create a modified array and find the middle element.

So how to find the median of the modified array?

There are a couple of ways to do the same. We can use binary search with segment tree to find the median according to the number of occurrence of each $a[i]$. Let say sum is the total number of elements in the modified array. To find the median of the modified array we need to find an element $a[i]$ such that the number of elements less than $a[i]$ is equal to number of elements greater than $a[i]$ in the sorted modified array. Therefore, if the number of elements which are less than $a[i]$ (including $a[i]$ in the sorted modifies array) is less than $(sum + 1)/2$, then our median lies on the right side of $a[i]$. Else median lies on the left side of $a[i]$. To find the number of elements less than $a[i]$ efficiently, we could use sum-segment tree or fenwick tree.
This median finding algorithm will take $O(log^2 n)$ per query.

But, we can also optimize it to $O(logn)$ per query using only segment tree with extra computations.

After, finding the median by any of the algorithm, we can easily find the cost to make all elements equal to the median by quering on a sum-segment tree.

TIME COMPLEXITY:
If we use binary search with segment tree to find the median in the modified array. We need to query in the segment for every iteration of binary search. Therefore, the Total Complexity will be:

$$O(q \times log^2 n)$$

Whereas, as discussed we can reduce this complexity to $O(q \times logn)$ using extra computation on the segment tree (mentioned in the setter's solution).
Though, we allowed $log^2 n$ per query to pass.

SETTER'S SOLUTION
https://www.codechef.com/viewsolution/62724395

TESTER'S SOLUTION
https://www.codechef.com/viewsolution/62724255

TESTER'S SOLUT
https://www.codec

Quote

Invitation to Alohomora [Editorial Update]

PREREQUISITES

Dynamic Programming, Segment Tree, Randomization

PROBLEM

Given a **Randomly generated** Permutation of length N and an integer K, find the maximum length of the longest subsequence of given permutation such that there are at most K pairs of neighboring elements such that left element is greater than right element. (Referring to these as special pairs)

QUICK EXPLANATION

- It is easy to do a trivial $O(K * N^2)$ DP by maintaining state (i, k) storing the length of longest subsequence ending at i, which has at most x pairs where the first element is greater than second
- We don't need to check transitions from position j to position i where $i < j$ if
  - $P_i < P_j$ and there exists p such that $i < p < j$ and $P_j < P_p < P_i$
  - $P_j > P_i$ and there exists p such that $i < p < j$ and either $P_j < P_p$ or $P_p < P_i$
    In both cases, we can include the element at position p to obtain a longer subsequence with the same number of special pairs. With random permutation, the number of transitions per state is reduced to $O(log(N))$, leading to complexity $O(K * N * log(N))$
- If the whole permutation contains up to K pairs, then the whole permutation is the required subsequence, otherwise, we expect the special pairs to be spread almost evenly, implying the number of special pairs for subsequence ending at position i must be $i * K/N$
- So, let's maintain a wide diagonal of the DP table instead of maintaining the whole DP table since the probability of the special pairs being concentrated towards ends is exponentially low.

EXPLANATION

Slow solution

Let us use a simple Dynamic Programming approach here, where state (i, k) denotes the length of the longest subsequence of first i elements in a given permutation, such that there are exactly K special pairs.

Transitions are relatively straight forward, we can write $f(i, k) = 1 + \max_{j<i}(1 + f(j, k^{'}))$ where $k^{'} = k$ if $P_j < P_i$ and $k^{'} = k - 1$ if $P_j > P_i$. This recurrence correspond to trying to extend subsequences ending at each position $j < i$ to include element at position i.

The longest subsequence is the maximum in the whole DP table. The initial state can be defined by filling the DP table with 1 s

Computing the answer for each state requires $O(N)$ transitions, and there are $O(N * K)$ states. This solution has time complexity $O(K * N^2)$ and space complexity $O(K * N)$, both of which are too high for given constraints

Reducing the transitions

Let us try to reduce the number of transitions required per state. While computing values for states at position i, we don't need to check states at position j where $j < i$ if either of the following is satisfied

- $P_j < P_i$ and there exists a p such that $j < p < i$ where $P_j < P_p < P_i$
  OR
- $P_j > P_i$ and there exists a p such that $j < p < i$ where $P_j < P_p$ or $P_p < P_i$

In both cases, it is beneficial to include element p when extending subsequence ending at position J to include the element at position i since we can include the element at position p in subsequence to obtain a longer subsequence with the same number of special pairs, so for position i, checking position p is sufficient.

From first condition, the set of positions checked having $P_j < P_i$ are a subsequence $i_1 < i_2 < \ldots < i_l$ such that $P_{i_j} > P_{i_{j+1}}$ is satisfied for $1 \leq j < l$.

From second condition, let's find largest p such that $P_p < P_i$. All positions $j < p$ don't need to be checked. Let's say positions $p < i_1 < i_2 < \ldots i_l < i$ are the positions to be considered. Then again, $P_{i_j} > P_{i_{j+1}}$ is satisfied for $1 \leq j < l$.

In both cases, the number of transitions from state i is bounded by **number of suffix maximums** for subarray $P_{1,i-1}$, where a suffix maximum of an array is position x such that $P_x > P_y$ for all $y > x$.

Let's visualize reversing this subarray, and the number of transitions is bounded by the number of prefix maximums of a given permutation.

This question is already answered, and by the proof mentioned **here**, we can see that the expected number of transitions to be processed from a given position becomes $log(N)$.

Hence, the solution now has time complexity $O(K * N * log(N))$ and space complexity $O(K * N)$

Reducing the size of DP table

Let us compute the number of special pairs in a given permutation. Say there are C such pairs. If we have $C \leq K$, then the whole permutation is a valid subsequence.

Otherwise, we can greedily prove that it is beneficial to pick a subsequence with exactly K such pairs, since selecting fewer pairs is not optimal.

Core Observation
The crucial observation here is that since the permutation is random, we can expect the K pairs to be evenly distributed in the range from 1 to N.

So, based on the above observation, the optimal subsequence must have approximately $i * K/N$ special pairs for elements up to position i.

We attempt to maintain DP table values which we expect to be relevant. For position i, we expect states (i, x) to be relevant as $|x - i * K/N|$ is smaller. So, let's choose a bound B and discard states (i, x) if $|x - i * K/N| > B$. We can visualize it as preserving a B width diagonal from the bottom left corner to the top right corner.

For example, considering permutation `2 7 6 3 4 1 8 5` and $K = 5$, the DP table is generated as follows

```
1 2 2 2 3 1 4 4
1 2 3 3 4 4 5 5
1 2 3 4 5 5 6 6
1 2 3 4 5 6 7 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
```

By using bound $B = 2$, we keep the following table

```
1 2 2 2 - - - -
1 2 3 3 4 3 - -
1 2 3 4 5 5 6 -
- 1 3 4 5 6 7 7
```

```
- - - 4 5 6 7 8
- - - - 1 6 7 8
```

This way, we intend to store only $O(B)$ states per position, reducing time complexity to $O(B * N * log(N))$ and space complexity to $O(B * N)$

Bound B, if chosen too high, shall lead to TLE or MLE, and if chosen too tight, you might actually miss a permutation P depending upon the values in the DP table we discarded. The probability of this happening with a sufficient bound is exponentially low, and becomes even lower as N increases, by a proof similar to **Chernoff Bound**.

For this problem, choosing $B = 70$ was sufficient, and TL was kept wide enough to even allow $B = 300$ to get accepted. During test generation, Some random tests requiring $B = 138$ were found as well.

Implementation

Dynamic Programming is straightforward. Computing the list of transitions for each state can be done by building a segment tree, where position x stores p if $P_p = x$ and $p < i$ and we are computing transitions for position i.

We can repeatedly query the rightmost position of an element containing value in range $[lo, P_i - 1]$ where $lo = 0$ initially, and we update $lo = P_q + 1$ if q is the position returned. This process runs time proportional to the number of transitions, and thus is executed $O(N * log(N))$ times.

PREREQUISITES:

Segment Tree, Deque.

PROBLEM:

Given a cyclic array A of length N, with current position being the first element, Perform following operations.

- $!$ Shift all elements to the right and make the last element as the first element.
- $?$ print the maximum number of consecutive 1s in the array. In case the answer is more than K, print K.

SUPER QUICK EXPLANATION

- The value of K is irrelevant. We just need to calculate the longest contiguous sequence of 1s after any number of shifts.
- If we append an array to itself, the problem reduces to, Given a sequence of length $2 * N$, we need to answer the maximum length of the contiguous sequence of 1s in the range [pos, pos+n-1].
- We can use Segment Tree to calculate this, storing information for every node in segment tree $[l, r]$ (Max length of consecutive ones in range l to r, Number of 1s at prefix of [l, r], Number of 1s at suffix of [l, r] and whether the segment contains a zero at all.)
- We can maintain a shift pointer which, on every shift query, moves one step to the left, and in case it is already at position 0, it moves to position $N - 1$. This way, answer to every query is same as the answer to query $[shuft, shift + N - 1]$ to above segment tree.

EXPLANATION

First of all, let's see the brute force solution.

We can actually shift the array in $O(N)$ time and for answering queries, we can iterate over the whole array again in $O(N)$ time, Giving Overall Time complexity $O(Q * N)$ which shall time out.

Now, Let us focus on a faster solution.

There are two (or even more) solutions, out of which, I am going to share a more general one, which has wider applications than the other, the Segment Tree solution.

In Segment Tree, we usually store some information for a node, and we should be able to combine information of two child nodes to get the same information for the parent node.

Segment Tree is simple, so, I'll be focusing more on finding the answer using segment tree. Segment tree resources are listed at the end.

Let us call maximum length of contiguous 1s in a range $[l, r]$ as range max of $[l, r]$.

Now, For this problem, to answer the query, we need the range max for a segment.

Suppose we have it calculated the range max of two segments $[l, mid]$ and $[mid + 1, r]$ and we need to combine this to calculate range max for segment $[l, r]$.

But, the catch here is, that we cannot directly assume that maximum length of contiguous 1s in the range $[l, r]$ is **just** the maximum of range max of both children.

Consider example: The array is 1 1 0 1 1 1 0 0

Assume l = 1, r = 8 and mid = 4. We see, that range max for $[l, mid]$ is 2 and $[mid + 1, r]$ is 2, but range max of segment $[l, r]$ is 3. This happened because the left segment's suffix merged with the right segment's prefix to form a larger segment. The left child had a suffix length 1, and the right child had prefix length 2, which merged

to form a segment of length 3. Hence, we also need to store the length of contiguous 1s at the start of the segment as well as at the end of the segment.

Also, we need another variable telling whether a segment contains at least one 0 or not. The reason is, that if a segment does not contain 0 at all, this means that prefix and suffix are same as the length of a segment. This case needs to be handled specially.

Now, we are ready to merge the information of two children to calculate the same information for the parent node.

We need to consider all four cases, whether the left child contains 0, and whether the right child contains 0 or not. So, Here we go.

- If both left and right child do not contain 0, we see, that the range consists of all 1s, hence, prefix, suffix as well as range max are equal to the length of the segment.
- If only left child contains 0, we can see, that by merging, prefix of left child become prefix of parent, suffix of parent is the length of right child plus suffix of left child, and range max is maximum of range max of both segments, and the segment length formed by merging suffix of left child with whole right segment.
- If only right child contains 0, we can see, that prefix of the parent is formed by merging whole left child with the prefix of right child, and suffix of the parent is just the suffix of right child. The range max is the maximum of range max of both segments, as well as the maximum of segment length formed by merging whole left child with the prefix of the right child.
- Now, Both child contains 0, Then we see, that prefix of left child become prefix of parent, suffix of right child become suffix of parent, and range max of segment is maximum of range max of both segments, and the segment formed by merging suffix of left child and prefix of right child.

Proving the above is not hard, and this gives us a very neat and simple solution to solve the problem.

Let us append the array to itself, so we get a new array B of length $2 * N$ with property that $B[i] = B[i + N] \forall i \in [0, N-1]$. Now, Consider one by one, the segments $[0, N-1]$, $[1, N]$, $[2, N+1]$ and so on. You can see, that it is basically the different shifts of the array. So, we can change the rotation to shifts in the array B.

We can reformulate the problem as.

Given an array B of length $2 * N$, we need to find maximum length of contiguous sequence between range $[P, P + N - 1]$ for a given P

This can be easily answered using the above segment tree.

But as the problem says, we should report a segment only of length $\leq K$, we print K if the length of the sequence we find is greater than K.

This completes another Segment Tree problem. But we still need to practice segment trees, so here we go.

An excellent blog on Segment tree **here** .

A nice set of problems to practice we may find **here** .

Alternative Implementations

An alternate solution could be based on observation, that after handling the case in which array consists of 1 only, there will be at least two disjoint segments consisting of 1s only, considering First and Last elements to be neighbors. At any moment, due to shifting, only one of these segments can be split (divided due to the border). We can see, that we can find the maximum length of contiguous sequence containing 1 by considering the position of the shift, and making cases on basis of the position of largest two segments consisting of 1s only, which we can find beforehand.

Time Complexity

Time complexity is $O((N + Q) * logN)$. $O(N * logN)$ for building Segment Tree while $(Q * logN)$ for processing answer queries.

PREREQUISITES:

Segment Tree, Binary Search

PROBLEM:

You are given array $A_1, A_2, \ldots, A_n$. You have to process queries of two types:

- Find the number of students who cut in line if the array of discontent is $[A_l, A_{l+1}, \ldots, A_r]$ - that is, the subsegment of the array A;
- Add a number W to $[A_l, a_{l+1}, \ldots, A_r]$ - that is, the subsegment of the array A.

You also have to find the number of students who cut in line using the initial aa as an array of discontent.

EXPLANATION:

Subtask 1:

Here $q = 0$

The student that enters the queue, at last, will have the value of discontent 0, as after that no student is going to enter the queue. Also, all the students that are the right to this student will have the discontent value greater than 0. As their discontent value will increase by 1.

Hence, we can make a conclusion that the rightmost index whose value is 0 is the person that enters last in the queue. Also, observe that if the last student is at the end of the queue then this student didn't cut the line.

Now we will remove the last student from our queue and decrement 1 from every student right to it. We will keep doing it till there is no student left in the queue.

Time Complexity

$O(N * log(N))$

Solution

```
#include <iostream>
#include <vector>

using namespace std;

struct SegTree {
    vector<int> t, mod;
    int sz = 1, n;

    void build(int v, int l, int r, vector<int> &a) {
        if (r - l == 1) {
            t[v] = a[l];
            return;
        }

        int m = (l + r) / 2;
        build(v * 2 + 1, l, m, a);
        build(v * 2 , m, r, a);
        t[v] = min(t[v * 2 + 1], t[v * 2]);
    }

    void push(int v) {
```

```
        t[v * 2 + 1] += mod[v];
        t[v * 2] += mod[v];
        mod[v * 2 + 1] += mod[v];
        mod[v * 2] += mod[v];
        mod[v] = 0;
    }

    int get_first_zero(int v, int l, int r) {
        if (r <= l || t[v] > 0)
            return -1;
        if (r - l == 1)
            return l;

        push(v);
        int m = (l + r) / 2;
        int res = get_first_zero(v * 2, m, r);
        if (res == -1)
            res = get_first_zero(v * 2 + 1, l, m);
        return res;
    }

    int get_first() {
        return get_first_zero(1, 0, n);
    }

    int get_val(int v, int l, int r, int L, int R) {
        if (r <= L || R <= l)
            return 1e9;
        if (L <= l && r <= R)
            return t[v];

        push(v);
        int m = (l + r) / 2;
        return min(get_val(v * 2 + 1, l, m, L, R), get_val(v * 2, m, r, L, R));
    }

    void update(int v, int l, int r, int L, int R, int val) {
        if (r <= L || R <= l)
            return;
        if (L <= l && r <= R) {
            t[v] += val;
            mod[v] += val;
            return;
        }

        push(v);
        int m = (l + r) / 2;
        update(v * 2 + 1, l, m, L, R, val);
        update(v * 2, m, r, L, R, val);
        t[v] = min(t[v * 2 + 1], t[v * 2]);
    }

    void update(int l, int r, int val) {
        update(1, 0, n, l, r, val);
    }

    int get_val(int l, int r) {
        return get_val(1, 0, n, l, r);
    }

    SegTree(int _n, vector<int> a) {
```

```
        while (sz <= 2 * _n)
            sz *= 2;
        n = _n;
        t.resize(sz);
        mod.resize(sz);
        build(1, 0, n, a);
    }
};

int solve(int n, vector<int> a) {
    SegTree val(n, a);
    vector<int> tmp(n);
    for (int i = 0; i < n; ++i)
        tmp[i] = i;
    SegTree num(n, tmp);
    int pos = val.get_first_zero(1, 0, n);
    int ans = 0;
    int now_size = n;
    while (pos != -1 && now_size > -2) {
        if (num.get_val(pos, pos + 1) != now_size - 1)
            ++ans;
        val.update(pos, pos + 1, 1e9 + 228);
        val.update(1, 0, n, pos + 1, n, -1);
        num.update(pos + 1, n, -1);

        --now_size;
        pos = val.get_first_zero(1, 0, n);
    }
    return ans;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    int n;
    cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; ++i)
        cin >> a[i];
    cout << solve(n, a) << '\n';
    int q;
    cin >> q;
    while (q--) {
        int t, l, r;
        cin >> t >> l >> r;
        if (t == 1) {
            vector<int> b;
            for (int i = l - 1; i < r; ++i)
                b.push_back(a[i]);
            cout << solve(b.size(), b) << '\n';
        } else {
            int val;
            cin >> val;
            for (int i = l - 1; i < r; ++i)
                a[i] += val;
        }
    }
```

```
}
```

Subtask 2:

Let's go from right to the left, maintaining maximum discontent (x). It is clear that the last student got up correctly, then $val = a[n]$. Now if the $(n-1)^{th}$ student enters the queue correctly then the value of $a[n-1]$ should be the same as that of $a[n]$. But what if the $(n-1)^{th}$ student enters the queue by cutting the line then it will increase the discontent value of nth student.

Hence we say that, if $a[n-1] < x$, then $(n-1)^{th}$ person got up incorrectly. Also, we decrease our x by 1. (since we took into account one who did not get up correctly), otherwise, he got up correctly and $val = a[n-1]$, We repeat the process till we reach the leftmost end of an array.

From this solution, it is easy to show the following fact: Let $c_i = a_i - i$. Then the answer is the number $i : i > a_j$, for all $j > i$.

Time Complexity

```
O(N*Q)
```

Solution

```
#include <iostream>
#include <vector>

using namespace std;

int solve(int n, vector<int> a) {
    int val = a.back();
    int ans = 0;
    for (int i = n - 2; i >= 0; --i) {
        if (a[i] < val)
            --val, ++ans;
        else
            val = a[i];
    }
    return ans;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    int n;
    cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; ++i)
        cin >> a[i];
    cout << solve(n, a) << '\n';
    int q;
    cin >> q;
    while (q--) {
        int t, l, r;
        cin >> t >> l >> r;
        if (t == 1) {
            vector<long long> b;
```

```
            for (int i = l - 1; i < r; ++i)
                b.push_back(a[i]);
            cout << solve(b.size(), b) << '\n';
        } else {
            int val;
            cin >> val;
            for (int i = l - 1; i < r; ++i)
                a[i] += val;
        }
    }
}
```

Subtask 3:

We can make sqrt decompositions.

- We will store all the maxima in the block in sorted order. We will go to the left and will apply a binary search in the block.
- Updates can be easily done by rebuilding the side bocks and adding W to every block in between.

Time Complexity

$O(N * sqrt(N * log(N)))$

Solution

```
#include <iostream>
#include <vector>
#include <algorithm>

#define all(a) a.begin(), a.end()

using namespace std;

const int K = 650;
const int N = 1e5 + K + 228;
const int INF = 1e9 + 228;

vector<int> records[N / K];
int add[N / K];
int A[N];

int n;

void build(int k) {
    int last_max = -INF;
    records[k].clear();
    for (int i = min((k + 1) * K - 1, n - 1); i >= k * K; --i) {
        if (A[i] > last_max) {
            records[k].push_back(A[i]);
            last_max = A[i];
        }
    }
}

int get(int l, int r) {
    int last_max = -INF;
    int ans = r - l + 1;
    while (r >= l) {
```

```
        if (r % K == K - 1 && r - K + 1 >= l) {
            ans -= (records[r / K].end() - upper_bound(all(records[r / K]), last_max - add
            if (records[r / K].size())
                last_max = max(last_max, records[r / K].back() + add[r / K]);
            r -= K;
        } else {
            if (A[r] + add[r / K] > last_max)
                last_max = A[r] + add[r / K], --ans;
            --r;
        }
    }
    return ans;
}

void update(int l, int r, int val) {
    while (l < r) {
        if (l % K == 0 && l + K <= r) {
            add[l / K] += val, l += K;
        } else {
            int nxt = min(l - l % K + K, r);
            for (int j = l; j < nxt; ++j)
                A[j] += val;
            build(l / K);
            l = nxt;
        }
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    cin >> n;
    for (int i = 0; i < n; ++i) {
        cin >> A[i];
        A[i] -= i;
    }
    for (int i = 0; i < n; i += K)
        build(i / K);
    cout << get(0, n - 1) << '\n';

    int q;
    cin >> q;
    while (q--) {
        int t, l, r;
        cin >> t >> l >> r;
        if (t == 1) {
            cout << get(l - 1, r - 1) << '\n';
        } else {
            int val;
            cin >> val;
            update(l - 1, r, val);
        }
    }
}
```

Subtask 4:

Author's Sketch

We implement the $query(v, w)$ function - the number of maxima is greater than w. ($cnt[v]# - count maximum on segment, $max[v]$ - maximum in segment). If $max[v->R] > w$, then we do not care what is in the left segment (we took it into account at the vertex). Then $query(v, w) = query(v->r, w) + cnt[v] - cnt[v->r]$ (count maximum in left son) If $max[v->R] <= w$, then there are no suitable maxima at this vertex, then $query(v, w) = query(v->l, w)$. With this function, it is easy to combine the segment, and therefore respond to the segment. And the modifier on the segment is a regular push in SegmentTree.

TIME COMPLEXITY:

$O(Q * log^2 N)$

PREREQUISITES:

Segment-tree(Lazy propagation), matrix-multiplication

PROBLEM:

The problem is self-explanatory.

- For a query of Type 1, you will be given two numbers L and R and a character C. You are supposed to replace every character in S from position L to R with C.
- For a query of Type 2, you will be given 4 numbers L,R, p and q. Now, for every character in S from position L to R, perform the following operations in the order left to right:
  - If the character is $'A'$, then replace p with $(p - q + MOD)\ \%\ MOD$ and q with $(p + q)\ \%\ MOD$.
  - If the character is $'B'$, then replace p with $(p + q)\ \%\ MOD$ and q with $(q - p + MOD)\ \%\ MOD$.

EXPLANATION:

Let the transformation matrix for characters 'A' and 'B' be

$\begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$ and $\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$ respectively. This is because:

$$\begin{bmatrix} a & b \end{bmatrix} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} a - b & a + b \end{bmatrix}$$

and

$$\begin{bmatrix} a & b \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} a + b & b - a \end{bmatrix}$$

Next, we build a segment tree of size N, where each node is a 2×2 matrix which is the multiplication of its two children. The $i - th$ leaf node is $\begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$ if $S[i]$ ='A' or $\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$ if $S[i] =$ 'B'.

To answer the query, we simply do the standard range query operation on the segment tree and obtain the transformation matrix. Multiply the initial p and q with this matrix will give us the answer.

To update the segment tree(with lazy propagation) we create two arrays storing the powers of each transformation matrix upto N and do the lazy update on the segment tree.

TIME-COMPLEXITY:

$O((N + Q)log(N))$

PREREQUISITES:

Lazy Propagation, Segment tree

PROBLEM:

The princess of Shamakhi is trying to find the source of eternal youth so she can restore her lost beauty. It turns out that she needs to collect the tears of a thousand beautiful maidens. To do that, she will accomplish a serious mission with three Bogatyrs, more precisely, update one of their assigned arrays within Q queries, and after each update, she wants to know the power of the Bogatyrs.

More formally, there are three Bogatyrs, each with their own array of N integers — named A, B, and C respectively. The princess of Shamakhi has Q updates of the form (t, pos, val), whose details are as follows:

- If $t = 1$, then assign $A_{pos} = val$
- If $t = 2$, then assign $B_{pos} = val$
- If $t = 3$, then assign $C_{pos} = val$

After every update, you need to find the power of the Bogatyrs (P), which is calculated as follows:

$$P = \max_{1 \le L \le R \le N}(a(L-1) + b(L,R) + c(R+1))$$

Here, the functions a, b, and c are defined as:

- If $1 \le x \le N$, then $a(x) = \sum_{i=1}^{x} A_i$, otherwise $a(x) = 0$.
- if $1 \le x \le y \le N$, then $b(x,y) = \sum_{i=x}^{y} B_i$, otherwise $b(x,y) = 0$.
- if $1 \le x \le N$, then $c(x) = \sum_{i=x}^{N} C_i$, otherwise $c(x) = 0$.

Can you help the princess of Shamakhi to restore her lost beauty by finding the power P of the three Bogatyrs after each update?

EXPLANATION:

Let array P keep the prefix sums for array A, more specifically $P_i = A_1 + ... + A_i$, and array M keep the prefix sums for array B, more specifically $M_i = B_1 + ... + B_i$ and array S keep the suffix sums for array C, more specifically $S_i = C_i + ... + C_N$.
Then the power of the Bogatyrs is the maximum value of $P_{l-1} + (M_r - M_{l-1}) + S_{r+1}$ for some $1 \le l < r \le N$.
Let us simplify the formula, $P_{l-1} + (M_r - M_{l-1}) + S_{r+1} = (P_{l-1} - M_{l-1}) + (M_r + S_{r+1})$.
Let us compute two new arrays D, E, where D_i = $P_i - M_i$ for $1 \le i \le N$, $E_i = M_i + S_{i+1}$ for $1 \le i \le N$. The problem now reduces to maximize the value of $D_l + E_r$ for $1 \le l < r \le N$. This can be done with the help of segment tree, where each node of segment tree keeps the maximum possible value of $D_l + E_r$ (satisfying $l < r$), maximum value of D array in the range, maximum value of E array in the range. Now it remains how to efficiently update these D, E arrays for each type of query.
Updating some position of A array influences the suffix of D. More specifically if we update A_{pos} with val then each value in the suffix D starting from pos is increased by $val - A_{pos}$(decreased if this value is negative).
Updating some position of B array influences the suffix of D and E array.
More specifically if we update B_{pos} with val then each value in the suffix D starting from pos is increased by $B_{pos} - val$ (decreased if this value is negative) and suffix E starting from pos is increased by $val - B_{pos}$ (decreased if this value is negative)
Updating some position of C array influences the prefix of E array. More specifically if we update C_{pos} with val then each value in the prefix of E ending at $pos - 1$ is increased by $val - C_{pos}$(decreased if this value is negative).
As a result, we need to keep two separate lazy propagation arrays for D and E arrays to deal with range addition updates for D and E.

The answer after each query would be the maximum value of $D_L + E_R$ stored in the root node of the segment tree

For details of implementation, please refer to the solutions attached.

PREREQUISITES:

Ordered-Sets

PROBLEM:

You are given N rectangles of different dimensions and colours. Each of the rectangles has three properties, the colour C_i, the length L_i and breadth B_i. The length L and the breadth B of these rectangles are even and all the rectangles are centred at the origin. The rectangles are placed one over the other. The rectangle i, is obscured by the rectangles j if $j > i$, that is only the colour of the top rectangle is visible in the overlapping region.

Given the colours and the dimensions of the rectangles, find out how many unit squares of each colour are visible in the final figure.

QUICK EXPLANATION:

As the figure is always symmetric about x and y axis, we need to just calculate the area for a quadrant and then multiply by 4 at the end.
Instead of processing from rectangle 1 to rectangle N, here we go from rectangle N to rectangle 1. So instead of placing the rectangles at top one by one, we place them at the bottom of others.
When we keep on adding rectangles this way, we observe that the outline of the current figure forms a staircase-like structure. So while placing a new rectangle we can find the portion of the rectangle that would be visible once it is placed behind the current figure.
We sum over these visible areas as per the different colours to get the area of each colour in the final figure.

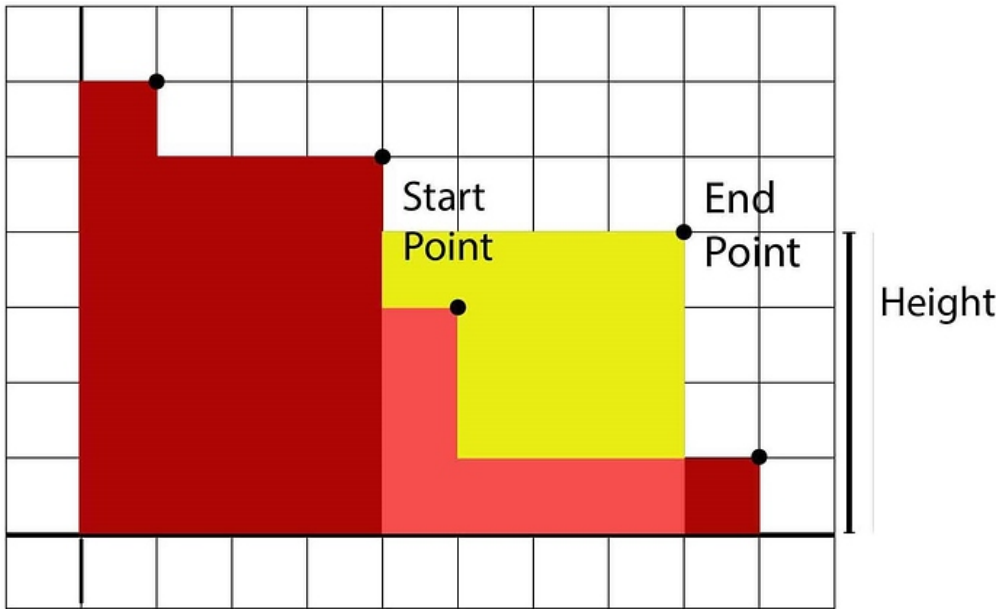
EXPLANATION:

As mentioned above, we would be adding rectangles in reverse from N to 1. At any instant, the outline of the figure forms a staircase structure.

So lets say, at some moment I have placed i rectangles and now I am placing $(i+1)^{th}$ rectangle behind them. Some of the portion of this rectangle will be hidden behind other rectangles and some part visible after placing it. If a portion of the rectangle is visible, then I can say that it will be visible in the final figure as well, as the next rectangles will be placed behind these (i+1) rectangles.

Therefore, for every rectangle, while placing it, I just need to find the area of the part of the rectangle that will be visible after placing it behind the previous rectangles. I add this area to the count of the respective colours for each rectangle and get the final answer.

Now comes the question, how to find the area of the $(i+1)^{th}$ rectangle that is visible in the final figure.



Refer to the figure above.
Now , I am adding a rectangle to this structure, as the structure is forming a staircase-like structure, I can binary search on it to get the starting point a at which the visible portion of this new rectangle starts i.e. a previously added rectangle whose height is just greater or equal to current rectangle' height in the outline.
Let the length of rectangle be b.
There may be two cases here:

- $b \leq a$: The rectangle will be completely hidden. It will not contribute to the final figure.
- $b > a$: Some portion of the rectangle would be visible in the final figure.

Now in the case $b > a$. We have to consider the contribution of this rectangle in the final figure.
The contribution of this rectangle is

$(b - a) \times height$ - Area of the staircase outline from a to b

Then we update the staircase strucure after inserting this rectangle.

Implementation

For maintaining the staircase structure, I have used an ordered set of pairs with (x,y) coordinates of top-right corner for every rectangle in it.

This structure satisifies the property $x_1 \leq x_2 \leq x_3 \leq x_4 \leq \cdots \leq x_k$ and $y_1 \geq y_2 \geq y_3 \geq \cdots \geq y_k$.

To add a new rectangle (x,y) beneath these k rectangles, we find the lower_bound of the indices. Now we iterate over the rectangles which lie inside the new rectangle area and remove them one-by-one. As we remove them, we also calculate the area thay had occupied which we subtract from the current rectangles area.

Time Complexity: $O(N \times log(N))$

PREREQUISITES:

Sweep Line , Binary Indexed Tree , Segment Tree

PROBLEM:

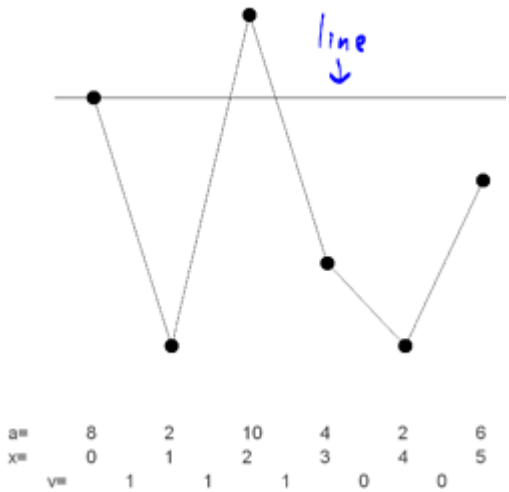
There are N points, the i-th point is at (i, A_i). For all $0 \le i \le N-2$, segment i connects points i and $i+1$. You are given Q queries (l, r, y), meaning that you need to calculate the number of segments in $[l, r)$ that intersect with the horizontal line with height y.

QUICK EXPLANATION:

Segment i will intersect with the line if $\min(A_i, A_{i+1}) \le y \le \max(A_i, A_{i+1})$. We will sweep over y while maintaining the segments that could be intersected in a data structure while answer queries.

EXPLANATION:

Segment i will be active if it intersects with the horizontal line at y for some query (even if the segment is not within $[l, r)$). Let the value of a segment be 1 if it is active and 0 if it is inactive. The values of the segments are represented by the array v shown below:



The answer for a query is the number of active segments in $[l, r)$, or the sum of the values of the segments in $[l, r)$. In order to answer the queries efficiently, we will use a sweepline algorithm. The idea of this sweepline algorithm is to imagine a horizontal line sweeping from $y = -\infty$ to $y = \infty$. At any point in time, the values of the segments should be updated for the current horizontal line. Whenever the horizontal line reaches y_i for some query i, we will answer query i.

There are three types of events:

1. For each segment i, when the line reaches $y = \min(A_i, A_{i+1})$, the value of the segment i will become 1.
2. For each query i, when the line reaches y_i, we will answer query i by adding up the values of the segments in $[l, r)$.
3. For each segment i, when the line goes over $y = \max(A_i, A_{i+1})$, the value of the segment i will become 0.

Since the sweepline starts from $y = -\infty$ and moves to $y = \infty$, we should process the events in increasing order of y. The only thing which remains is to calculate the sum of the values of the segments in $[l, r)$ efficiently for each query. This can be done with any data structure which supports 1. update $x_i = v$, given i and v 2. find the sum of x_i in a given range. A binary indexed tree or a segment tree will work.

The total time complexity is $O(N \log N)$.

PREREQUISITES

Segment Tree, Bit Manipulation

PROBLEM:

There is an array A consisting of N elements and we are given Q queries to perform which would be of the following 3 types-

- $?\ L\ R$ - Output the number of unique elements that Chefina doesn't like in the complete array. The condition for liking the element is that the frequency of the element is odd in the given index range $[L, R]$.
- $*\ i\ X$ - Replace the ith index element with X.
- $!\ i\ Y$ - Replace the ith index element with the GCD of A_i and Y.

EXPLANATION

We have two approaches for the problem, the first one being a naïve one as it will only work for subtask 1 and will give TLE verdict for subtask 2. So, we require a better approach to the problem.
We will require an understanding of **Segment Trees** and **Bitsets** to solve this problem.

Naïve Approach:

Using frequency array, with a worst case time complexity of $\mathrm{O}(Q * N)$

Naïve Approach

We can create a frequency array to store the frequency of the elements and also keep track of the number of unique elements in another variable.

- For 1st Query:

We can iterate over the given range $[L, R]$ for each query and create a frequency array for each query, and further count the number of elements that appear an odd number of times and subtract it from the unique elements count.

- For 2nd and 3rd Query:

We can simply replace the values. If all the queries are of type 1, then it will consume a lot of time.

Hence, we need to come up with a better solution.

Better Approach:

We will be creating a segment tree to store the number of elements that appear an odd number of times in a particular segment.

Segment tree overview

At the first step, a segment tree is formed. We follow 1-base indexing in the segment tree.
If the root node is at i^{th} index then-

- Left Child will be at $2 * i$ position and
- Right Child will be at $2 * i + 1$ position.

A recursive function will be used to build the segment tree in $\mathrm{O}(N)$ time.

In the segment tree, each node represents the answer for the corresponding range using **bitset**. Since the range of A_i is $[0, 10^3)$, we can use a bitset of size 1000 bits.

The leaf node represents an element of the array (the Xth bit will be set at leaf node if the element in the array is X). The parent node is obtained by taking **XOR** of its children nodes.

But why XOR?

If an element appears an odd number of times in the left range as well as the right range, then in the complete range it will appear an even number of times represented by resetting the bit to 0.
Following this logic, it is clear that if the frequency of an element has same parity in left and right ranges, then its corresponding bit will be 0, otherwise it will be 1. This is nothing but the **XOR** of both sides. Bitsets in C++ provide us with this bitwise functionality and we can simply take the XOR of the left and right child to compute the answer for parent node, (or the current range).

Time taken will be $\mathrm{O}(\frac{D}{32})$, where D is the size of the bitset.

For 1st Query:

The node would be a D bit binary number (or a bitset) for the range of that node where,

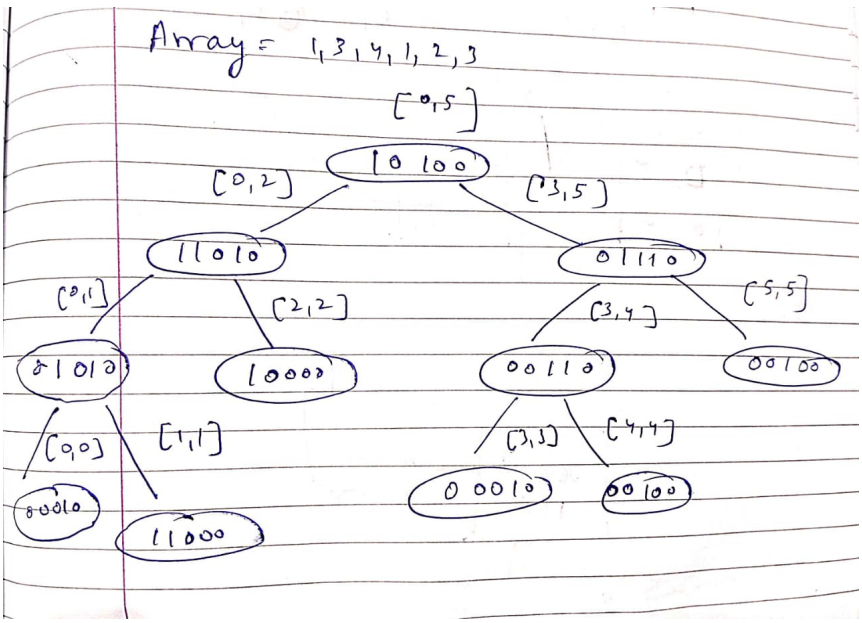
- 0 at i^{th} index would represent that the element "i" is either not present in that range, or is present even number of times, whereas
- 1 at i^{th} index represents that the element is present an odd number of times.

Bitset follows 0-base indexing from right to left.
For example, in a 4-bit binary number 0010, this represents even occurrences of $3, 2, 0$ and odd occurrences of 1.

The range of the tree: It is the part from which the query is executed.

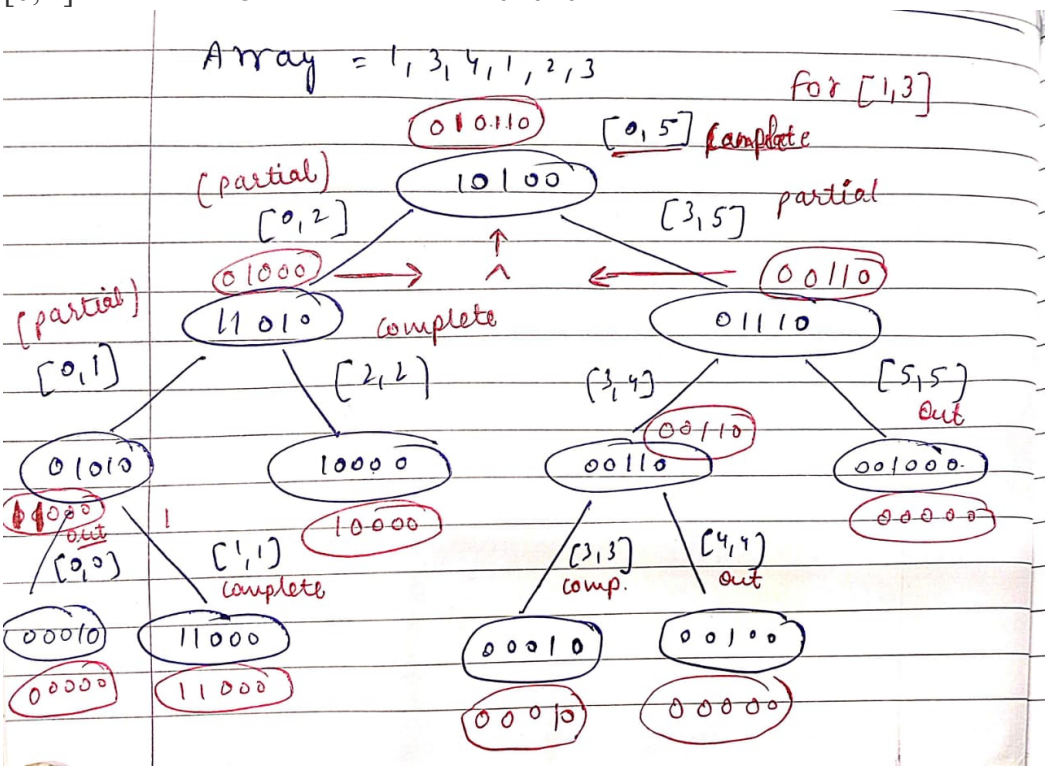
Example

$a[]$=$[1, 3, 4, 1, 2, 3]$ size from $[0, 5]$ (this is the range) i.e root of tree;



Left subtree will store the answer for the range $[0, 2]$ and the right subtree for $[3, 5]$ (these are the ranges).

In this way, we continue to form the segment tree. The leaf node for the range $[0, 0]$ i.e the first element is 00010, and $[1, 1]$ represents 01000 (wrongly mentioned in the figure as 11000). The parent element i.e $[0, 1]$ will be the XOR of its children i.e 01010.



To solve the query,
Frequency bitset is found for the given range. For instance, the range is $[1, 3]$, as shown in the above example. Then we have 3 possible cases.

- **Case 1**: (No overlap)
  When it lies outside the range of the parent node:
  In this case, return the value $[00000...000]$ via bitset.
- **Case 2**: (Complete overlap)
  When it lies completely inside the range of the parent node:
  Return the bitset present here, without exploring further children.
- **Case 3**: (Partial overlap)
  In this case take the **XOR** of the value of both nodes.
  In the example above, we have $[1, 3]$
  Starting from the root node $[0, 5]$, the left subtree is $[0, 2]$ and right subtree is $[3, 5]$.
  Hence $[1, 3]$ lies partially in both. Therefore, continue till leaf node and take respective XOR till we get the answer for both ranges $[0, 2]$ and $[3, 5]$. Finally the output will be the XOR of the two children.

For 2nd and 3rd Query:

To update the tree,
If we have to replace the value of any element in the array then it would mean that a leaf node of that element has to be updated and hence there has to be changes for all those nodes that appear from that leaf node all the way up to the root node.
So after replacing the value of the leaf node, we go on to update its parent node by taking the XOR of the child nodes and so on till the root node (similar to the first query).
This can be done in a bottom-up fashion for both the queries as they require only updation and are virtually the same. Additional $\mathrm{O}(log(maxA_i))$ work has to be done for calculating GCD.

PREREQUISITES:

Segment Tree, Greedy, Square Root Decomposition

PROBLEM:

Given an array of N positive integers. You have to answer Q queries, each requiring you to determine the maximum possible GCD over all subsequences (of length > 1) of a specified range of the array.

EXPLANATION:

Hint 1

Observe that it is sufficient to select only 2 elements in the specified range to achieve the maximum possible gcd.

Why?

It is known that:

$$\gcd(a, b) \geq \gcd(\gcd(a, b), c) = \gcd(a, b, c)$$

Therefore, since the gcd never increases on adding another element, it makes sense to select the least possible number of elements.

Hint 2

Armed with this observation, write a naive bruteforce to solve the problem in $O(N^2)$ per query.

Solution

For each element in the range, compute the gcd with every other element in the range. The answer is then the maximum over these computed values.

This is too slow to pass the subtasks apart from the 1st one.

Hint 3

Let $d(x)$ represent the number of divisors of positive integer x. Then, for a fixed a, there are exactly $d(a)$ possible values of $\gcd(a, b)$ over all valid b.

Thus, instead of iterating over every pair of elements, can you iterate on the divisors of every element (in the range) to calculate the maximum possible gcd?

Solution

Let D_x represent the set of divisors of integer x.

Let S be the multiset of $\{D_{A_l}, D_{A_{l+1}}, \ldots, D_{A_r}\}$. Then, it is easily verifiable that the maximum possible gcd is the largest number in S *that occurs more than once*.

Precomputing divisors for all A_i **can be done** in $\approx O(N\sqrt{\max A_i})$. Then, the above method can determine the solution in $O(N \cdot d)$ per query, where $d \leq 100$ represents the maximum number of divisors (for the given constraints).

This is an improvement to our naive bruteforce, but will only pass the first two subtasks.

Hint 3

Offline (aka, no updates) range queries is a significant pointer towards sorting the queries first and then solving.

Solution for subtask 3; Unrelated to complete solution

Standard application of **Mo's Algorithm** . The time complexity is $O((N+Q) \cdot d \cdot \sqrt{N})$, which works for the first 3 subtasks.

Solution

Start by sorting all queries in increasing order of their right (R value) bound.

We iterate over array A from 1 to N. Let i represent the current index we are processing. Also, let v_x represent the maximum gcd over all pairs of elements in the subarray $A[x..i]$. Then, answer to any query of the form $[l, i]$ equals v_l.
Initially, v_x is 0 for all x.

Here's how we can efficiently update array v in each iteration -
Traverse over all divisors of A_i - call the current divisor d. Consider the largest $p < i$ such that $d \mid A_p$. Then, update $v_x = \max(v_x, d)$ for all $x \leq p$.
It is easy to see (by recursion) that at the end of the above operation, the updated array matches the aforementioned definition of v.

How do we implement this?
The value of p can be efficiently determined in $O(1)$ by maintaining a sliding window during our iteration. Range updates (and point queries) to v can be done in $O(\log N)$ using a segment tree with lazy propagation.

The total time complexity is therefore:

$$O(N \cdot d \cdot \log N + Q \cdot \log N)$$

This however, TLE's on exactly one test in the last subtask!
To remedy this, observe that the number of updates is a magnitude more than the number of queries. Thus, we could switch to a data structure with faster updates and slightly slower queries - **Square Root Decomposition**.
For this, we must appropriately convert our operations to be point update (giving $O(1)$ per update) and range query (with $O(\sqrt{N})$ per query), which is left to the reader as a simple exercise.

TIME COMPLEXITY:

Precomputing divisors of each integer takes $O(N\sqrt{\max A_i})$.

Sorting the queries takes $O(Q \log Q)$. Building array v for each index i takes $O(d)$, where d is the number of divisors of A_i. Determining the answer to each query takes $O(\sqrt{N})$ with range queries. The total time complexity is therefore

$$O(N \cdot d + Q\sqrt{N})$$

per test case.

SOLUTIONS:

Editorialist's solution can be found **TBD**

Quote

---

rahul_246 **Mar '22**

Editorialist's solution link is not working , kindly check it once !!

---

shaily_adm **Mar '22**

It'll be updated in some time.

---

beruboiv **Apr '22**

Still not updated.

---

beruboiv **Apr '22**

Can anyone please explain why my solution is failing?

https://www.codechef.com/viewsolution/62514092

jatin0308_adm **Apr '22**

Hey beruboiv ☺️

I think your segment tree implementation have an error , I wrote the code with same logic but with Fenwick Tree so you can go through it 🙂 .

```
#include <iostream>
#include<bits/stdc++.h>
#include<deque>
#include<algorithm>
#include<math.h>
#include<sstream>
#include<stdio.h>
#include<bitset>
#include<string>
#include<vector>
#include<unordered_map>
#include<queue>
#include<set>
#include<fstream>
#include<map>
#define int long long int
#define ld long double
#define pi 3.14159265358979323846264338327950288419 71
#define MOD 1000000007
#define MOD1 998244353
#define print(vec) for (int i = 0; i < vec.size(); i++) cout << vec[i] << " "; co
using namespace std;
int dx[4] = {1, 0, -1, 0};
int dy[4] = {0, 1, 0, -1};
int inf = 1e18;
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().time_si
        return splitmix64(x + FIXED_RANDOM);
    }
```

```
};
#ifdef __SIZEOF_INT128__
ostream& operator << (ostream &os, __int128 const& value) {
    static char buffer[64];
    int index = 0;
    __uint128_t T = (value < 0) ? (-(value + 1)) + __uint128_t(1) : value;
    if (value < 0)
        os << '-';
    else if (T == 0)
        return os << '0';
    for (; T > 0; ++index) {
        buffer[index] = static_cast<char>('0' + (T % 10));
        T /= 10;
    }
    while (index > 0)
        os << buffer[--index];
    return os;
}
istream& operator >> (istream& is, __int128& T) {
    static char buffer[64];
    is >> buffer;
    size_t len = strlen(buffer), index = 0;
    T = 0; int mul = 1;
    if (buffer[index] == '-')
        ++index, mul *= -1;
    for (; index < len; ++index)
        T = T * 10 + static_cast<int>(buffer[index] - '0');
    T *= mul;
    return is;
}
#endif
int add(long long a, long long b) {return ((a % MOD) + (b % MOD)) % MOD;}
int subtract(long long a, long long b) {return ((a % MOD) - (b % MOD)) % MOD;}
int mult(long long a, long long b) {return ((a % MOD) * (b % MOD)) % MOD;}
int add1(long long a, long long b) {return ((a % MOD1) + (b % MOD1)) % MOD1;}
int subtract1(long long a, long long b) {return ((a % MOD1) - (b % MOD1)) % MOD1;
int mult1(long long a, long long b) {return ((a % MOD1) * (b % MOD1)) % MOD1;}
int expo(int a, int b, int mod) {
    int res = 1;
    while (b > 0)
    {   if (b & 1)
            res = (res * a) % mod;
        a = (a * a) % mod;
        b = b >> 1;
    } return res;
```

```
}
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}
int mminvprime(int a, int b) {
    return expo(a, b, b + 2);
}
const int maxn = 1e5 + 12;
vector<int> a, d[maxn], sofar(maxn);
int n;
vector<int> c;
void update(int ind, int val)
{
    while (ind > 0)
    {
        c[ind] = max(c[ind], val);
        ind = ind - (ind & (-ind));
    }
}
int query(int ind)
{
    int s = 0;
    while (ind <= n)
    {
        s = max(s, c[ind]);
        ind = ind + (ind & (-ind));
    }
    return s;
}
int32_t main() {
#ifndef ONLINE_JUDGE
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
#endif
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    int tt;
    cin >> tt;
    for (int i = 1; i < maxn; ++i)
    {
        sofar[i] = 0;
        for (int j = i; j < maxn; j += i) d[j].push_back(i);
    }
    while (tt--)
```

```
    {
        int q;
        cin >> n >> q;
        a.assign(n + 1, 0);
        for (int i = 1; i <= n; ++i)
        {
            cin >> a[i];
        }
        c.assign(n + 1, 0);
        vector<pair<int, int>> qrs[n + 1];
        for (int i = 1; i <= q; ++i)
        {
            int l, r;
            cin >> l >> r;
            qrs[r].push_back({l, i});
        }
        vector<int> ans(q+1);
        for (int i = 1; i <= n; ++i)
        {
            for (auto x : d[a[i]])
            {
                if (sofar[x]) update(sofar[x], x);
                sofar[x] = i;
            }
            for (auto [l, ind] : qrs[i])
            {
                ans[ind] = query(l);
            }
        }
        for (int i = 1; i <= q; i++) cout << ans[i] << "\n";
        for (int i = 1; i <= n; ++i)
        {
            for (auto x : d[a[i]]) sofar[x] = 0;
        }
    }
    return 0;
}
```

---

beruboiv **Apr '22**

Thanks @jatin0308_adm ,

I understood my mistake, it was a fault in segment tree logic only.

But now I am getting a TLE, is there a way to remedy this?

https://www.codechef.com/viewsolution/62739656

jatin0308_adm **Apr '22**

Hey @beruboiv ☺️ ,
I think this is due to tight constraints and you are logic is not the good fit so try reduce the number of operations(I think this is due to implementation of 2 seg trees).

beruboiv **Apr '22**

Thanks! I will look into that.

PREREQUISITES:

Prefix sums, Segment tree

PROBLEM:

You are given an array B of length N and Q queries of two types:

1. Given x and y, find the maximum sum of $B[l, r]$ such that $l \le x \le y \le r$.
2. Given x and v, update B_x to v.

QUICK EXPLANATION:

Maintain a segment tree of prefix sums and another segment tree of negative prefix sums. Query 1 is the range max of a suffix of the first segment tree added to the range max of a prefix of the second segment tree. Query 2 is a range add update on the suffix of both segment trees.

EXPLANATION:

Let C_i be the sum of the first i elements of B (the prefix sum array of B). Whenever we process query 2, all C_i such that $i \ge x$ increase by $v - B_x$ (since all prefixes with at least x elements will contain the x-th element of B and be updated), so we can maintain all C_i by using a data structure which supports range add updates, such as a segment tree.

The sum of $B[l, r]$ is equivalent to $C_r - C_{l-1}$. So, we want to find the maximum value of $C_r - C_{l-1}$ given that $l \le r$, $l \le x$, and $y \le r$. Notice that $x \le y$, so we can ignore the condition $l \le r$. Now, l and r are independent, so we can focus on maximizing C_r and $-C_{l-1}$ independently.

Finding the maximum value of C_r given that $r \ge y$ equivalent to a range maximum query on $C[y, n]$, which we can perform efficiently with a segment tree. Finding the maximum value of $-C_{l-1}$ is not much different: we will have another segment tree maintaining the negative prefix sums, and to answer queries of type 1, we just need to perform a range maximum query on $-C[0, x-1]$.

PREREQUISITES

Segment Tree

PROBLEM

Consider a tree with N nodes, rooted at node 1.
The value of the ith node $(1 \le i \le N)$ is denoted by A_i.

Chef defines the function $MEX(u, v)$ as follows:
Let B denote the set of **all** the values of nodes that lie in the **shortest path** from u to v (including both u and v).
Then, $MEX(u, v)$ denotes the **MEX** of the set B.

Find the **maximum** value of $MEX(1, i)$, where $1 \le i \le N$.

QUICK EXPLANATION

- Maintain a set of integers not yet found on the current path from the root to u for some node u.
- If A_u is present in the set, we shall remove it at the start of DFS, process its subtree, and then add back A_u to set.
- The MEX on a path from the root to u is the minimum element present in this set which the ordered set can answer fast enough.

EXPLANATION

We need to compute the MEX of values on the path from node 1 to node u for all u and take its maximum. Let's denote S_u as the set of values on the path from 1 to u. Then we have $B_u = \{A_u\} \cup B_p$ and $B_1 = \{A_1\}$...

So, if we process the nodes of trees in DFS order, we only need to add and remove each value exactly once. The value would be added when DFS enters node u, and the value A_u would be removed when DFS exits node u.

Hence, we have to support adding an element, removing an element, and computing MEX of the current set.

There are multiple possible solutions for this problem, I'd discuss two for now.

Clever solution, simpler implementation

Instead of maintaining the set of values found on the path from 1 to u, let's maintain the set of values **not** found on the path from 1 to u. This helps us because now, the MEX is simply the smallest value present in this set.

Now, when DFS enters a node u, we check if this special set contains A_u. If it contains, we would remove A_u from the set at the start.

Now, MEX for the current set is the minimum element in this special set. Also, we can make recursive calls to solve for all nodes in subtree of u.

Lastly, If A_u was removed at the start of DFS, we add back A_u to this set as we exit node u.

The implementation of this is added in Editorialist solution 1.

Generic overkill solution using segment tree

We maintain the frequency of each element in the segment tree. Specifically, leaf x denotes the number of occurrences of x. Our segment tree should support

- Increase/decrease frequency of an element x by 1.

- Find MEX of elements currently present.

The MEX of elements would be the leftmost leaf with 0 frequency, which can be found using a technique called tree descent, described **here**.

The implementation of this is added in Editorialist solution 2.

TIME COMPLEXITY

The time complexity of both solutions is $O(N * log_2(N))$ per test case.

PREREQUISITES:

segment tree

PROBLEM:

You are given N segments where each segment is of the form $[L_i, R_i]$ $(1 \le L_i \le R_i \le N)$.

It is ensured that for any two segments X and Y, one of the following conditions hold true:

- $X \cap Y = X$,
- $X \cap Y = Y$, or
- $X \cap Y = \emptyset$ (empty set)

Note that $\cap$ denotes the intersection of segments.

In other words, for any two segments, either one is completely inside the other or they are completely disjoint.

The ith segment $(1 \le i \le N)$ is assigned a value V_i.
A subset $S = \{S_1, S_2, \ldots, S_k\}$ having k segments is considered *good* if there exists an integer array B of size k such that:

- All elements of the array B are **distinct**.
- B_i lies in the segment S_i.

Let the score of a *good* subset be defined as the sum of values of all segments of the subset. Find the **maximum** score of a *good* subset.

EXPLANATION:

First sort the segments given in the input in increasing order of their length (number of elements in them). Now iterate over the segments in the ascending order of their length. Now, Suppose we are at some segment $[L, R]$. There are two cases:

- There is a point in $[L, R]$ which has not been assigned to any interval yet. Then, we can freely assign it to this interval.
- Every point has been assigned to some segment. Then, find the lowest value point in the interval and check if it is more profitable to assign it to $[L, R]$.

Both the cases listed above can be combined into one by assuming that every point is initially assigned a value of 0. The calculation of the minimum and its position in a range along with point updates can be done efficiently using a segment tree.

For details of implementation please refer to the solutions attached.

TIME COMPLEXITY:

$O(Nlog(N))$ for each test case.

PREREQUISITES:

BINARY SEARCH, TRIES, SEGMENT TREES

PROBLEM:

Given Q queries (l, r, x), find the maximum value of $a_j \bigotimes x$ where $l \leq j \leq r$.

EXPLANATION:

The first subtask is trivial : You can just iterate through all elements in range for each query to find the answer.

The third subtask has $x = 0$, which reduces the problem to Range Maximum Query, which can be solved with Sparse Table or Segment Trees in $O(n \log n)$.

Now, let's see how to solve the second subtask. One way to solve this is by building a trie (though it's not neccesary and the same solution can be applied to the sorted array of numbers instead). Build a trie for all the numbers in range $[l, r]$. We walk down the trie from the root. Whenever we have a choice of whether to go left or right, we pick the choice that maximizes the final xor value, i.e. if the current bit of x is 0, choose the path that is labelled 1 and vice versa. Thus, each query can now be solved in $O(\log n)$ time.

We can generalize this idea to solve the whole problem. What we need here is a segment tree of tries (or segment tree of sorted arrays is sufficient in this case). We can build a trie for each segment of the segment tree in $O(n \log n \log a_i)$ time and for each query, we need to query the trie in $O(\log n)$ segments. Thus, the whole solution would work in $O((n + q) \log n \log a_i)$ time.

PROBLEM:

There are N companies in your area. Let A_i be the net-worth of the ith company.
After the pandemic, they start acquiring each other. A company can only acquire its adjacent company (the nearest company to its left or to its right, provided it exists).

Let's say, if company-1 of net-worth x acquires its adjacent company-2 of net-worth y then company-2 disappears and company-1 net-worth changes to $x - y$.
This goes on until only one company remains.

Now you need to answer Q queries of the following type :

$L\ R$

You need to find the maximum net-worth of the last remaining company if the companies in the range L to R (both inclusive) start acquiring each other.

NOTE: Each query is independent to each other and after each query the initial array will be retained in order to process the next query.

PREREQUISITES:

- Maths
- RMQ

DIFFICULTY:

MEDIUM

EXPLANATION:

Firstly, let's see calculate the answer for a single query

Each company in the query range will either be added or subtracted to the final answer.
Now, observe that any combination of addition and subtraction is valid except adding all or subtracting all.

Now to maximize the answer we have a few cases :

- If only positive net-worth companies are present, we add everything to the result except the minimum element which we subtract.
- If only negative net-worth companies are present, we subtract everything to the result and add the minimum element.
- Now if we have a mix of positive and negative net-worths, then we simply add the absolute values of all the net worths.
- Also note, if we have 1 element in the range then the answer would the element itself and not the absolute value.

We have solved the problem for a single query. Let's do it efficiently.

To calculator the sum of absolute values in a range we could use a prefix sum array.
And to find the minimum elements in a range we could use a segment tree or sparse table.

See setter and tester codes for implementation.

PREREQUISITES

String Suffix Structures, Segment Tree

PROBLEM

Given an array A of N integers, compute the maximum possible score among all non-empty subarrays of given array. The score of subarray $A_{l,r}$ is given as $score(l, r) = (\sum_{i=l}^{r} A_i) * (\text{occurrences})$, where occurrences is the number of occurrences of subarray $A_{l,r}$ in array A.

QUICK EXPLANATION

- Build suffix tree on given array A as the text string. Each path corresponds to a distinct subarray, where the number of leaves in subtree denotes the number of occurrences (Assuming the last integer is distinct).
- Then do a DFS on this tree, maintaining the sum of values on the path from the root to the current node, and the number of leaves in the subtree of each node.
- Since there can be multiple values written on a single edge, we need to support query $q(l, r)$ as the maximum sum of the non-empty prefix of subarray $A_{l,r}$ to process all of them simultaneously, which can be done using segment tree.

EXPLANATION

Intuition

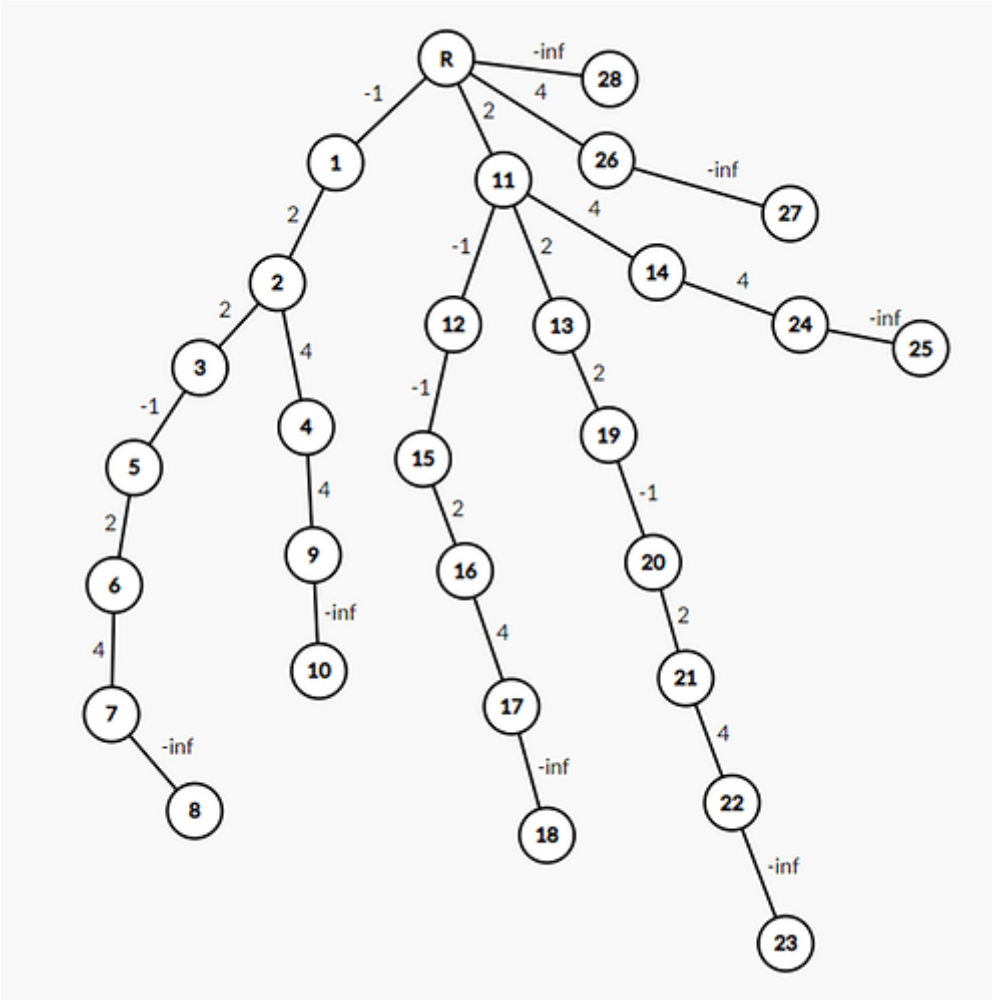
Counting the occurrences of a substring in the original string is a well-explored domain in string problems, so let's consider our array as a string of integers, each index representing a character in the string.

We need to consider all subarrays of the given array A, so let's create a trie, and add all suffices of A in this trie.

This structure is called **Suffix Trie** and is slightly different from the suffix tree.

Throughout the editorial, I'll consider $A = [-1, 2, 2, -1, 2, 4, -\infty]$ as the given array. I have added $-\infty$ at the end which I'll explain later.

The Trie of all suffices of this array is as follows. Ignore Node labels, only edge labels matter, and node R is root.



Claim: Each node in suffix trie represents a subarray of the given array A.
Proof: Each subarray $A_{l,r}$ can be written as the prefix of l-th suffix of the original array.

So, for each node, we need to compute the sum of subarray (which is the sum of edge labels on the path from the root to this node) and the number of times it appears in the original array.

Claim: The number of times some subarray appears in A is exactly the number of leaves in the subtree of the node corresponding to that subarray, in suffix trie.
Proof: The number of leaves in the subtree of node u in suffix trie means the number of suffices of A which have subarray represented by node u as their prefix. Each suffix corresponds uniquely to a start point, and each suffix contributes to exactly one occurrence.

Hence, in the above example, subarray $[-1, 2]$ appears twice, and subarray $[2]$ appears thrice, but subarray $[2, -1]$ appears once.

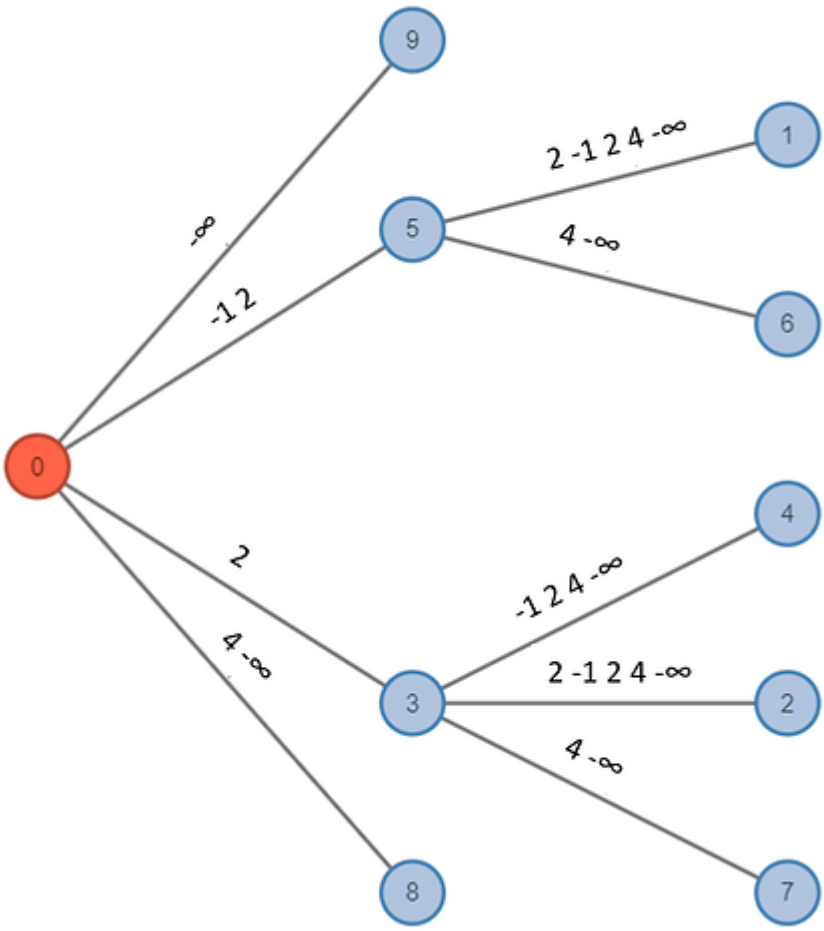
Hence, we can build suffix trie, and for each node, we can calculate the number of leaves in its subtree. The sum of subarray can be calculated along with DFS, or by prefix sums, solving the problem in time proportional to the number of nodes in the tree.

The number of nodes in suffix trie may be up to N^2, so this solution doesn't work for the final subtask.

Faster intuition

Assume that cnt_u denotes the number of leaves in the subtree of node u in suffix trie.

Let's merge the nodes which have only one child with their child and join their edge labels. We get the following tree, which is the **suffix tree** of A



The number of nodes in the above tree is $O(N)$ since there are at most $N+1$ leaves (including $-\infty$), and each node has at least two children, so there cannot be more than N internal nodes.

But, there is a new problem now. Earlier, we were able to check all subarrays by establishing the correspondence between each node and subarray. But now, some subarray may end inside an edge. For example, subarrays $[2,1]$, $[-1,2,2,-1]$ end within an edge, so these would be missed if we apply the same idea again.

Let's say a subarray $A_{l,r}$ is considered to end on edge e if traversing through the suffix tree from the root by following the path with labels from l to r, the last traversed edge is edge e.

For example, for subarray $[-1]$, $[-1,2]$, the last traversed edge is $(0,5)$, while last traversed edge for subarrays $[2,-1], [2,-1,2], [2,-1,2,4], [2,-1,2,4,-\infty]$

Claim: All the subarrays **ending at same edge** appear the same number of times in array A.
Proof: WLOG assume there are two characters in the edge label, say a and b. So suffix trie must have some nodes u, v and w such that there's an edge from u to v with label a and from v to w with label b, and node v doesn't have any other child.

Since node w is the only child of node v, $cnt_v = cnt_w$, which implies subarray ending at a appears same number of times as subarray ending at b.

Hence, this allows for a new approach. For each edge from u to v, we know that all subarrays ending at this edge appear cnt_v times in the original array. We need to compute maximum sum subarray for each subarray ending at edge e

Segment tree Time

Let's assume we want to compute this for edge $e = (u, v)$. All the subarrays ending at e must have all values on labels on the path from the root to u as the prefix, only then we can reach this edge.

Considering all subarrays $A_{l,r}$ ending at edge e, we can see that the right ends from a consecutive segment.

For example, for edge $(3, 4)$, subarrays appearing on that edge are $$A_{2,3}, A_{2,4},A_{2,5},A_{2,6}$, and each of these subarrays appear $cnt_v = 1$ times in original array.

Claim: The right ends of subarrays ending at the same edge are consecutive values in some range.

For edge $(3, 4)$, the left end is a node 2, and the right ends of subarrays form range $[3, 6].

Let's say, for edge e, this interval is $[l', r]$. This means, that subarrays ending at edge u have prefix $A_{l,l'-1}$ followed by some non-empty prefix of subarray $A_{l',r}$

For edge $(3, 4)$, $l' = 3$, so each subarray consist of $A_{2,2}$ followed by some non-empty prefix of $A_{3,6}$

Hence, we need to compute for edge e, $\max_{i=l'}^{r} S_{l,r} = S_{l,l'-1} + \max_{i=l'}^{r} S_{l',r}$. $S_{l,l'}$ is computed during DFS, or by prefix sums.

Now we have query, for some interval (l, r), find $\max_{i=l}^{r} S_{l,r}$, the largest valued non-empty prefix.

This is where the segment tree comes in. Let's suppose node i in tree stores information of interval $[L_i, R_i]$. max_i is the maximum value of $\max_{i=L_i}^{R_i} S_{l,i}$, the largest valued non-empty prefix in range $[L_i, R_i]$. and $sum_i = \sum_{i=L_i}^{R_i} A_i$.

It is easy to see that if node x is parent of node $2 * x$ and node $2 * x + 1$ in segment tree, then we can write $sum_x = sum_{x*2} + sum_{x*2+1}$ and $max_x = max(max_{2*x}, sum_{2*x} + max_{2*x+1})$.

Prove why above works, if for leaf nodes, we have $sum_x = max_x = A_{L_i}$

Concluding

Hence, the segment tree allows computing the largest subarray sum ending at each edge. We also know the number of occurrences of all subarrays ending at each edge. We can multiply and take maximum.

Why $-\infty$ was added at the end

Since $-\infty = -10^13$ doesn't appear in the array, adding it to the end makes sure that all the leaves are explicit nodes. Consider suffix tree for string `abcab`, the two leaves are implicit, which would not allow us to procced normally.

If this value is included in the subarray, the subarray has only one occurrence, and a very small value, so this doesn't affect the maximum answer.

Resources

[Suffix Tries and Suffix Trees]
[Suffix Tree Implementation]

TIME COMPLEXITY

The time complexity is $O(N * log(N))$ per test case.

Pre-requisites:

Segment Trees, BIT

Problem:

Given a rooted tree, each of whose node is associated with an integer, “health”. You have to perform two operations:

- decrease “health” of all descendants of a node by a given quantity.
- Report number of descendants of a given node with positive “health”.

Explanation:

Trees are hard to think about. In many tree problems involving data structures, the first step is to form one(or several) linear structures, and phrase the problem in terms of these linear structures.

In the present problem, we need to perform some operation/query on all descendants of a node together.

> When dealing with a rooted tree data-structure problem where operations/queries are made on all descendants of tree nodes, the tree can be linearised by writing the vertices down in the order given by pre-order traversal of the tree.

Pre-order traversal of the tree gives numbering to vertices such that all descendants of any internal node get consecutive numbers, and form a single range. This is because after visiting any vertex, dfs first visits all descendants of the vertex, and then all the remaining vertices.

Therefore, once we arrange vertices by their preorder traversal order, all descendants of any node become a subarray of this array. The new problem becomes:

> Given an array of integers("health"s of nodes), perform the following two operations:
>
> - decrease the value of all elements in a given subarray by a given number.
> - report the number of positive elements in a given subarray.

There are mainly two intuitive approaches to this problem:
i) An offline solution in **O(N log^2 Q + Q log Q)** time,
ii) An online in **O((Q + N)log N)** time.

Originally the Setter had only the offline solution in mind until others came up with an online one. For futher discussion, assume each decrement operation is given by four parameters (**time, dec, l, r**), where **time** is index of the operation(can be 1, 2, … Q), **l,r** give the subarray(l, r inclusive) which should be decreased by **dec**. Also assume that each query is represented as (**time, l, r**), meaning of terms being similar.

Offline solution

Lets first assume that all chefs die, by adding one more operation in the end which decreases the health of all chefs by 10^9. Suppose that I want to find out the death time of a single chef, chef **A**.

The most intuitive way to go about would be to write down all the decrement operations that effect chef **A**, along with the point of time at which they occur. Now, add decrement value of all those operations in chronological order until it exceeds the health of chef **A**. Thus I would get the time at which he died.

The method can be speeded up with binary search. That means, I would predict a point of time **t1** and try to find if chef **A** was alive at time **t1**, by summing up all decrement operation before time **t1**. To sum up all decrement values upto a given point of time efficiently, a Binary Indexed Tree(BIT) can be used. Thus, we could find out the time at which a given chef dies in **O(log^2 Q)** time, assuming we have build the BIT for storing all operations that effect this chef.

```
function get-time-of-death(int i):   // index of chef
    low = 1
    high = Q
    while(low < high)
        t1 = (low + high)/2
        if (healthOfChef[i] &leq; getDecSum(t1)) // get sum of all decrements performed up
            high = t1   // I am dead at time t1
        else
            low = t1+1  // I am alive at t1
    return low
```

However, we still have to get hold of all operations that effect chef **A**, and use them to construct our BIT. To do this efficiently, we can use the fact that all operations affect some continuous subarray of our array. If Chef **A** comes immediately after chef **B** on the array, the operations that effect chef **A** are going to be *almost same* as those that effect chef **B**. Only those operations whose range either ends at **B**, or begins at **A** have to be taken care of. Rest of the operations are going to be common. Here is a pseudo code showing how to accomplish this

```
BIT <- empty  // stores no operation
for i = 1 to N
    for each operation (time, dec, l, r) whose range (l,r) starts at i          // that i
        update-BIT(time, dec)
    death-time[i] = get-time-of-death(i)
    for each operation (time, dec, l, r) whose range (l,r) end at i             // that i
        update-BIT(time, -dec)
```

Great ! I got the time-of-death of all chefs. How do I go about answering all queries now ?
At this point, one can use ideas from a **previous editorial** (offline processing section) to completely solve the remaining part. However, I will reiterate the points here.

In current setting, all queries ask for number of alive chefs in a subarray at a given point of time. Suppose, for now, all queries asked for number of alive chefs in the entire array at any given point of time. This would be easy to do, because for all points of time we could store the number of chefs who died at that point of time. To answer a query at time **t**, we can sum up number of chefs died upto time **t**. Another BIT can be used to speed up this part.

Also, If I want to add one more chef to my array, who dies at time **t1**, it can be done easily by updating the BIT. I will borrow the following general tip from a previous editorial.

> If a problem can be solved efficiently for an array, and more elements can be also be added to the array efficiently, without hurting the ability to solve the problem efficiently, then the problem can also be solved (offline) for arbitrary prefixes(or suffixes) of the array.

We can use the above to find the number of dead chefs at time **t** in arbitrary prefixes of the array. But this would be sufficient to answer queries about arbitrary subarrays as well, because any query can be written down as two prefix queries: query(**time, l, r**) = prefix-query(**time, r**) - prefix-query(**time, l-1**). That means to report number of chefs alive at time **time** in the subarray given by **(l,r)**, find the number of alive chefs in the prefix ending at **r** at time **time**, and number of alive chefs in the prefix ending at **l-1** at time **time**.

To find number of alive chefs in arbitrary prefixes:

```
sort all queries (time, i) in increasing order of i.
Initialize a BIT, will all zeros
for i = 1 to N
```

```
    update BIT with death-time[i]
    for all queries of the form (time, i):
        report answer of query = BITquery(time)
```

Refer to setter's solution for further etails on implementation.

Online Solution

Online solution goes exactly how you would expect it to. For each update, it first decreases all values in the subarray, and then figures out which Chefs have died because of this update. It also maintains all alive chefs in order to answer all queries. A single segment tree can be used to solve this problem.

To efficiently decrease all values in a subarray, we would need lazy propagation. Also, to efficiently determine chefs who have died due to an update, each node(of our segment tree) should store the descendant leaf of minimum health(this leaf will be the first to die if all descendants are poisoned). The overall structure of a node of our segtree is:

```
struct node {
    int __health-of-least-healthy-chef__, __decrement__ , __alive-count__;
    // _Actual health_ of a leaf(in segtree) can obtained by subtracting __decrement__ val
    // This is called _lazy propagation_.
}
```

A node can be updated by simply updating **decrement** and **health-of-least-healthy-chef** parameters. decrement would need to be *propagated down* before updating any node.
After performing a normal update, the following function can be used to remove dead chefs:

```
function remove-dead-chefs(int root)
    if tree[root].health-of-least-healthy-chef > 0
        return // no dead chef below me
    if is-leaf(root)
        set __alive-count__ of root to 0, __health-of-least-healthy-chef__ to infinity
    else
        propagate the value of __decrement__ to both children, set __decrement__ of root t
        remove-dead-chefs(root * 2);    // some dead chef exists and needs to be removed
        remove-dead-chefs(root * 2+1);  // so check both children
        update __health-of-least-healthy-chef__ and __alive-count__ parameters of root
```

Query and update operations are very standard. Refer Editorialist's solution for details. The complexity of this method is **O(N lof N + Q log N)**, because each update/query takes **O(log N)**, and whenever a chef dies, **O(log N)** time to spent to figure out who has died and update the tree to reflect this.

Setter's Solution:

Can be found **here**

Tester's Solution:

PREREQUISITES:

Segment Tree with Lazy Propagation, Stacks.

PROBLEM:

Given a permutation P of length N, find the value of the following sum.

$\sum_{i=1}^{N} \sum_{j=i}^{N} getMin(i,j) \wedge getMax(i,j)$ where $\wedge$ denotes AND operation and $getMin(i,j) = min(P_i, P_{i+1} \ldots P_{j-1}, P_j)$ and $getMax(i,j) = max(P_i, P_{i+1} \ldots P_{j-1}, P_j)$.

EXPLANATION

Let us solve the simpler problem instead, where instead of AND operation, we have multiplication operation.

The problem becomes the same as finding the sum of product minimum and maximum value in subarray for all subarrays.

Let us group the subarrays by their endpoints. Suppose cur hold the sum of values for subarrays ending at position p. We want to calculate the same value, for all subarrays ending at position $p+1$ using this. First of all, the subarray $(p+1, p+1)$ is easy, just add $P_{p+1} * P_{p+1}$ to cur. Now notice how minimum and maximum values are affected by extending subarrays (i,p) to $(i, p+1)$ for $1 \le i \le p$.

If $P_p < P_{p+1}$, then let us find rightmost position q such that $P_q > P_{p+1}$. If there's no such position, then assume $q = 0$. P_{p+1} affects only the maximums for subarrays which start in range $[q+1, p]$ and end at p. Now, for a subarray (i,j), the maximum value changes from x to y, it's contribution to cur changes by $(y-x) * min(i,j)$, since cur includes $x * min(i,j)$ and we now want to include only $y * min(i,j)$.

We want to do this for all subarrays which start in the range $[q+1, p]$ and end at p. Also, the value of y is always P_{p+1}. Let us divide interval $[q+1, p]$ into disjoint exhaustive set of intervals such that for any two positons a and b in same interval, $getMax(a,p) = getMax(b,p) = x$. For each sub-interval (l,r), we add $(y-x) * sumMin(l,r)$ where $sumMin(l,r) = \sum_{i=l}^{r} min(i,p)$.

For getting $sumMin(l,r)$ we can maintain a segment tree, such that each position i currently stores $getMin(i,p)$. When we move from position p to position $p+1$, we find the leftmost position q such that $getMin(q,p) > P_{p+1}$ and assign all positions with value P_{p+1}. This is a standard application of Lazy propagation in Segment Tree which you may read about **here** .

For keeping a record of such intervals, a stack is a nice data structure. Suppose we consider positions from left to right and whenever we consider position p, we keep removing positions in the stack from the top, until either the stack is empty, or the element at the topmost position in P is greater than current element. After this procedure, we insert the current position into the stack. **This stack holds the invariant that values at positions are always in strictly increasing order.** This way, stack gives us the intervals, as well as automatically helps us to find position q.

We can similarly repeat this process where $P_p > P_{p+1}$ where only the minimums get affected and we have another segment tree for maximum values and a stack for the minimum value positions.

This allowed us to solve the problem where we have multiplication operation.

Coming back to the original problem, we can convert the AND operation into a $log(N)$ subproblems involving multiplication operation, handling each bit individually for the purpose of calculating the sum. For each bit, we need to maintain a separate segment tree, where bit at position i is set if the minimum (maximum) of range (i,p) has current bit set. For each bit, we have to handle updates separately.

This concludes the problem. Refer implementations in case anything is still unclear.

TIME COMPLEXITY

Counting the number of intervals made by the stack, we can see that total N insertions are made in whole process and deletions cannot exceed insertions, so they are also up to N only. The number of updates to each segment trees also do not exceed $2 * N$ and there is $log(N)$ segment trees and each update takes $O(log(N))$ time, time complexity does not exceed $O(N * log^2(N))$.

So time complexity is $O(N * log^2(N))$ per test case.

PREREQUISITES:

Inversion counting, Segment trees (with lazy propagation)

PROBLEM:

You're given an array A. For each $1 \le i \le N$ independently, find the following:

- If you can change A_i to any **real number**, what's the maximum possible number of inversions in A?

EXPLANATION:

Let $x_1 < x_2 < \ldots < x_k$ be the *distinct* values be present in A.

The main observation required to solve this task is as follows:
Suppose we set $A_i = x$, then there are only about $2k + 1$ 'interesting' values of x, namely:

- One of the x_i; or
- Something less than x_1 (they're all equivalent, so we can treat this as $x_1 - 1$); or
- Something greater than x_k (once again they're all equivalent, so we can consider this to be $x_k + 1$); or
- Something strictly between x_i and x_{i+1} for some i.
  Once again, we can consider this to just be $x_i + 0.5$.

Let's apply this information to solve the problem.

Subtask 1

The easiest subtask allows for pretty much brute-force computation.

As discussed above, there are at worst $2N + 1$ distinct values of x we care about.
So, for each index i:

- Set $A_i = x$ for each of the 'interesting' x.
- Compute the number of inversions in the resulting array.
- Print the maximum across all x.

Even if inversions are computed in $\mathrm{O}(N^2)$ each time, this will run in $\mathrm{O}(N^4)$ time overall which is enough to pass this subtask.

Subtask 2

N is larger now, though still small enough that $\mathrm{O}(N^2)$ or similar can pass.
Earlier, for each i, we set $A_i = x$ and computed the number of inversions in $\mathrm{O}(N^2)$.

Let's optimize the second part.
When we set $A_i = x$, notice that inversion pairs that don't involve index i don't really change.
So, we don't really need to recompute the total inversion number each time.

In particular,

- Let inv be the number of inversions in the original array A.
  Let B_i be the number of inversions involving index i.
- When we set $A_i = x$,
  - Suppose there are y elements before index i that are larger than x.
  - Suppose there are z elements after index i that are smaller than x.
  - Then, the number of inversions in the new array is exactly $\mathrm{inv} - B_i + y + z$.

Of these quantities, inv can be precomputed in $\text{O}(N^2)$, and B_i for all i can be computed at the same time (whenever you see $i < j$ with $A_i > A_j$, add one to both B_i and B_j).
So, if we were able to quickly compute y and z, we'd have a very quick algorithm.

There are several ways to do this, for example:

- Let P_i be a sorted list of the elements $[A_1, A_2, \ldots, A_i]$.
  Similarly, let S_i be a sorted list of $[A_i, A_{i+1}, \ldots, A_N]$.
- Then, when fixing $A_i = x$, y and z can be computed by binary searching on P_{i-1} and S_{i+1} respectively; to find the number of elements in the prefix/suffix that are larger/smaller than x.

Since $N \le 1000$, computing all the P_i and S_i lists using $\text{O}(N^2)$ memory is not an issue.

We're binary searching $\text{O}(N^2)$ times for a solution in $\text{O}(N^2 \log N)$ overall, which is enough to get AC.
It's possible write the solution to run in $\text{O}(N^2)$ as well (using two-pointers), but that isn't necessary to get AC.
Subtask 3

We can no longer independently try every x for each position.
However, we can still maintain enough information for each one.

Recall that when fixing $A_i = x$, we had the number of inversions be $\text{inv} - B_i + y + z$; where y was the number of elements larger than x before i, and z was the number of elements smaller than x after i.

First off, inv and all the B_i can be computed in $\text{O}(N \log N)$ time: this is a standard application of mergesort or segment/fenwick trees or (in C++) the **order-statistics set**.
A couple of articles detailing this can be found **here** (segment tree approach) and **here** (mergesort approach).

Now, let's define C_x to be the corresponding $y + z$ value for when we set $A_i = x$.
If we knew all the C_x values for index i, we'd just need to take the maximum of them, say M, and the answer would be $\text{inv} - B_i + M$.

Suppose we knew all the C_x values for index i.
Let's see how they change when we move to index $i + 1$.

- First, for any $x > A_{i+1}$, A_{i+1} was contributing a count of 1 to C_x. This will no longer be the case because index $i + 1$ will no longer be to the right.
  So, we need to decrement C_x for all $x > A_{i+1}$ by 1.
- Next, index i will newly become a 'left' index.
  This means, any x such that $x < A_i$ will get 1 added to its C_x value.
- These are the only changes!

Notice that our changes are essentially "subtract 1 from a range of C_x" and "add 1 to a range of C_x"; along with being able to know their maximum after these two operations.
That can be done quickly with a segment tree! (using lazy propagation, of course)

So, at each index we have two range updates and one range query (which is just a query for the whole range).
This can be achieved in $\text{O}(N \log N)$ time, which is good enough for us.

There are a couple of things to take care of:

- First, we need to set the initial C_x values for index 1.
  This is not hard: for each $i > 1$, we must add 1 to C_x for all $x > A_i$; which comes out to $N - 1$ more range updates — our segment tree can handle this.
- More pressingly, we have not-necessarily-integer values of x; as well as very large values of x. Building a segment tree on them is somewhat hard.
  To get around this, we can use coordinate compression.
  Recall that the distinct elements in A were $x_1 < x_2 < \ldots < x_k$.

Let's relabel them into $1, 3, 5, 7, \ldots, 2k-1$. This keeps the comparison relation between elements, while also giving us 'space' for real numbers:

- Anything less than x_1 can be represented by 0
- Anything more than x_k can be represented by $2k$
- Anything between x_i and x_{i+1} can be represented by $2i$

After compression, we have an array of size $2N$, and normal segtree stuff can be done on it.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PREREQUISITES

Range Minimum Query , Simple Math

PROBLEM

You are given **N** matchsticks arranged in a straight line, with their **rear ends** touching each other. You are also given the rate of burn for every matchstick (possibly same) in number of seconds it takes to burn it out. If a matchstick is lit from both ends, it burns out **twice** as fast - taking **half** as much time.

Answer several queries of the following type efficiently

All the matchsticks in the range **L** to **R**, inclusive are lit from their front ends simultaneously. Find how much time it takes for **all** the matchsticks to burn out.

QUICK EXPLANATION

For each query, the operation performed plays out in the following way

- All the matchsticks in the range **L** to **R** are lit from their front ends.
- The matchstick that burns **quickest** in the range **L** to **R** burns out and **ignites** all the other matchsticks on their **rear** ends.
- The matchticks in the range **L** to **R** now burn out **twice** as fast.
- **All** the other matchsticks burn out at their **original** rate.

We can find the time taken in all the steps above using only the following pieces of information for the segment **L** to **R**

- Quickest rate of burning for a match in the range **L** to **R**.
- Slowest rate of burning for all matches in the range **L** to **R**.
- Slowest rate of burning for all matches outside the range **L** to **R**.

EXPLANATION

For a given range **L** to **R**

- Let **m** denote the minimum time taken by some matchstick in the range **L** to **R** to burn out
- Let **M** denote the largest time taken by some matchstick in the range **L** to **R** to burn out
- Let **M**' denote the largest time taken by some matchstick outside the range **L** to **R** to burn out

The time taken by each of the steps in the scenario described above is as follows

- The matchstick that burns quickest, burns out
  - Takes time **m**
- The following things happen in parallel
  - The matchsticks in the range **L** to **R** now burn out twice as fast
    - Takes time **(M - m) / 2**
  - The matchsticks outside the range
    - Takes time **M'**

Thus, the time taken for all the matches to burn out completely is

```
m + max( (M-m)/2 , M' )
```

It remains to find efficiently the **minimum** time some matchstick will take in a range, and the **maximum** time some matchstick will take in a **range**.

Such queries can be answered in **O(N log N)** time by using segment trees . Refer to this topcoder tutorial for a wonderful writeup with code samples on how to get about writing a segment tree. The topcoder tutorial also describes a **O(N sqrt(N))** approach as well which will also work within the time limits for this problem.

Two segment trees must be constructed. One to answer queries of the type "**minimum in a range**", that returns the time it takes for the fastest burning matchstick to burn out. Another to answer queries of the type

"**maximum in a range**" to find **M** and **M**' as defined above. Note that **M**' will itself be the maxmimum of two ranges, **1 to L-1** and **R+1 to N** respectively.

A lot of solutions were stuck in the caveat that it is required to **always** print the answer in a **single** decimal place. Note how the answer will either be **integer**, or contain a single decimal digit (**5**). In case the answer is **integer**, it is required to print a trailing decimal followed by **0**.

PREREQUISITES:

[Segment trees with lazy propagation]

PROBLEM:

An array B is called good if, when sorted, B_i divides B_{i+1} for every i.

You're given an array A and Q queries on it. For each query (L, R), find the number of good subarrays of $[A_L, A_{L+1}, \ldots, A_R]$.

EXPLANATION:

Let's first analyze the structure of a good array.
Consider a good array B. Without loss of generality, we can assume B is sorted.

Now, for each i, either $B_i = B_{i+1}$ or $B_i < B_{i+1}$.
In the first case, divisibility is trivially satisfied.
In the second case, note that B_{i+1} is at least $2B_i$, since it should be a multiple of B_i that's larger than B_i.

In particular, this means that if B has K distinct elements, the largest of them is at least $2^{K-1} \times B_1$, since each time we move to a higher element the value at least doubles.

In our case, we know $A_i \le N \le 2 \cdot 10^5$.
This means that any good subarray can have at most 18 distinct elements, since $2^{18} = 262144 > 2 \cdot 10^5$.

Solving for $[1, N]$

For now, let's ignore queries entirely and focus on counting the number of good subarrays of a given array.

Let's fix the left endpoint L, and count the number of valid right endpoints R.
Note that if $[L, R]$ is good, then so is $[L, R-1]$: this follows from the fact that if a divides b and b divides c, then a divides c.

In particular, the set of valid R for this L form a contiguous range starting at L, so it suffices to find the right endpoint of this range.

Since the range can only contain upto 18 distinct elements, there's a rather simple way of doing this:
Start with $R = L$.
Then, move R to the next position that contains a new element and check if the resulting set of elements satisfies the divisibility condition.
if it does, jump to the next new element and continue; otherwise $R - 1$ is the position we're looking for.

To do this fast, we need to be able to quickly find the next new element.
This can be done as follows:

- Let S be a set that contains pairs of (position, element), sorted by position. Initially, S is empty.
- Iterate i from N down to 1.
- At i, if A_i exists in S (at some other position), delete it. Then, insert (i, A_i) into S.

S now contains the nearest occurrence of every distinct element that occurs at index $\ge i$, sorted by position. This is exactly what we want.
Now we can simply iterate across S to find the next distinct element, as detailed at the start of the section.

This way, we do N set insertions and deletions, and then at each index upto 18^2 operations (more specifically, upto $\log^2 N$ operations) to find the appropriate R, for a total of $\mathrm{O}(N \log^2 N)$ time.

Answering queries

Of course, applying the above method directly to each query is going to be too slow, since both N and Q are large.

However, notice that we can in fact reuse a lot of information if we solve our queries offline.
That is, first we'll read all the queries, then at a given index i we'll answer all queries whose left endpoint is i.

Consider a new array B of length N. Initially, $B_i = 0$ for every index.

Let's run the algorithm from the previous section. When at a fixed i, first find its optimal right endpoint r_i as before.
Then, add 1 to the range $[i, r_i]$ of B.

Note that B_j now represents the number of indices $\leq j$ (but $\geq i$) such that j is a good right endpoint for this index (since we add 1 to it exactly as many times as it occurs in $[x, r_x]$ for some x).

This means, to answer a query (i, R), we can simply take the sum $B_i + B_{i+1} + \ldots + B_R$.
Note that this is valid only when we haven't yet processed index $i - 1$, which is why we need to solve queries offline.

Maintaining B is theoretically quite simple: we need to support adding 1 on some range, and getting the sum of some range. This is a standard problem that is solved by segment trees with lazy propagation.

TIME COMPLEXITY

$\mathrm{O}(N \log^2 N + Q \log N)$ per test case.

PREREQUISITES:

Dynamic Programming, Binary Search, Range Minimum Queries, Coordinate Compression

PROBLEM:

Given array A and integer K. Determine the largest value s such that, it is possible to partition A into K contiguous blocks, where the sum of elements of each block is $\geq s$.

EXPLANATION:

Let s be the largest integer such that it is possible to partition A into K contiguous blocks, where the sum of elements of each block is $\geq s$. This value is the answer we are looking for.

Step 1: Use binary searching to find the value of s.

Why? How?

Let $solve(d)$ represent whether it is possible to partition A into K blocks, where the sum of elements of each block is $\geq d$. Let $solve(d)$ equal 1 if it's possible to partition, and 0 otherwise.

Observation 1: If $solve(d) = 1$, then $solve(d') = 1$ for all $d \geq d'$.
(The proof is trivial and left to the reader as an exercise)

Observation 2: If $solve(d) = 0$, then $solve(d') = 0$ for all $d' \geq d$.

Proof

We prove by contradiction.
Say there exists $d' \geq d$ such that $solve(d) = 0$ and $solve(d') = 1$.

By the previous observation, $solve(d') = 1$ implies $solve(x) = 1$ for all $d' \geq x$. But $solve(d) = 0$ - a contradiction and we are done.

The two observations imply the monotonic structure of the function $solve$, sufficiently allowing us to binary search to find the answer.

Let d be the currently considered value in the binary search. Now, we want to check if it's possible to partition A into K blocks, each whose sum $\geq d$.

Definition: *Call an array of elements good, if the sum of its elements $\geq d$.*

Let dp_i be the set of all positive integers k, such that it is possible to partition the first i elements of A into k good subarrays. I claim that dp_i is a (possibly empty) set of consecutive integers, for all i.

Proof

Since the proof has a lot in common with the logic of the solution, reading the rest of the editorial first will make the proof much more clearer.

Let $pref(x) = A_1 + A_2 + \cdots + A_x$.

Computing dp_i is straightforward, and can be described mathematically as:

$$dp_i = \{x + 1 \mid x \in dp_j \ \forall\, j < i \text{ and } pref(j) \leq pref(i) - d\}$$

Let's define $G_i[v]$ as the set of integers x, such that it is possible to partition the first j elements of A into x good subarrays, where $j < i$ and $pref(j) = v$.

$$G_{i+1}[v] = \begin{cases} G_i[v] \cup dp_i, & \text{if } pref(i) = v \\ G_i[v], & \text{otherwise} \end{cases}$$

This recurrence is derived from a simple (yet important!) observation, the proof of which is left to the reader as an exercise.

Finally (the last definition, bear with me), let's define $H_i[v] = G_i[v] \cup H_i[v-1]$. That is, $H_i[v]$ is the union of all sets $G_i[v']$ with $v' \leq v$. It is easy to deduce that $H_i[v']$ is always subset of $H_i[v]$.

Informally, $H_i[v]$ can be described as the set of integers x, such that it is possible to partition the first j elements of A into x good subarrays, where $j < i$ and $pref(j) \leq v$.

Then, we can rewrite the set relation of dp_i (mentioned at the start) as:

$$dp_i = \{x + 1 \mid x \in H_i[pref(i) - d]\}$$

Thus, if we prove that $H_i[v]$ is a (possibly empty) set of consecutive integers for all v, then dp_i is also a (possibly empty) set of consecutive integers, completing the proof.

We prove by induction.
Assume for a particular n, $H_n[v]$ is a set of consecutive integers, for all v. Let us prove by inducting on n.

$$\begin{aligned} H_{n+1}[v] &= G_{n+1}[v] \cup G_{n+1}[v-1] \cup \dots \\ &= \begin{cases} (G_n[v] \cup G_n[v-1] \cup \dots) \cup dp_n, & \text{if } pref(n) \leq v \\ (G_n[v] \cup G_n[v-1] \cup \dots), & \text{otherwise} \end{cases} \\ &= \begin{cases} H_n[v] \cup dp_n, & \text{if } pref(n) \leq v \\ H_n[v], & \text{otherwise} \end{cases} \end{aligned}$$

$H_{n+1}[v] = H_n[v]$ implies our proposition holds for $H_{n+1}[v]$ (since by our assumption, it holds for $H_n[v]$). So, lets only focus on the former case, which can be expanded as:

$$H_{n+1}[v] = H_n[v] \cup \{x + 1 \mid x \in H_n[pref(n) - d]\}$$

One of $H_{n+1}[v], H_n[pref(n) - d]$ is a subset of the other. A bit of case work shows that, the union of two overlapping sets of consecutive integers, after shifting all values of one set by 1, is still a set of consecutive integers. Therefore, the proposition holds true for $H_{n+1}[v]$, establishing the inductive step.

The trivial case of $n = 0$ has $H_0[v] = \emptyset$ for all v, which matches our proposition.

Thus proved.

Why is this claim important? The claim implies that the set of values of dp_i is made up of all integers in some range $[L_i, R_i]$. Then, A can partitioned into K good subarrays only if $L_N \leq K \leq R_N$.

All that remains now is to calculate L_i and R_i efficiently.

Suboptimal solution

A straightforward DP recurrence relation gives:

$$L_i = 1 + \min L_j$$
$$R_i = 1 + \max R_j$$

where $j < i$ and $\sum_{x=j+1}^{i} A_x \geq d$.

This can be calculated in $O(N^2)$, enough for subtask 2, but not viable for the original constraints.
Optimised solution

Let $pref(x) = A_1 + A_2 + \cdots + A_x$.

The suboptimal solution can be rewritten as:

$$L_i = 1 + \min L_j$$
$$R_i = 1 + \max R_j$$

where $j < i$ and $pref(i) - pref(j) \geq d \implies pref(j) \leq pref(i) - d$.

Let $mini_x$ represent the set of all L_j such that, $j < i$ and $pref(j) = x$. Similarly, let $maxi_x$ represent the set of all R_j such that, $j < i$ and $pref(j) = x$.
Then, from the above stated recurrence constraint, it is easy to see

$$L_i = 1 + \min_{x \leq pref(i) - d} (\min mini_x)$$
$$R_i = 1 + \max_{x \leq pref(i) - d} (\max maxi_x)$$

This can be done easily using range minimum queries on $mini$ and $maxi$, typically using a Segment Tree/Fenwick Tree with coordinate compression (since the range of x in $mini_x$/$maxi_x$ is $[-1e14, +1e14]$).

The time complexity of this approach is $O(N \log N)$, which is efficient enough to AC for the given constraints!

To summarise, binary search for the answer - let the midpoint value we are currently trying be d. For this particular d, calculate $[L_i, R_i]$ for all i, in subquadratic time. Then, A can be partitioned into K subarrays, each with sum of elements $\geq d$, only if $L_N \leq K \leq R_N$. Return the result and continue the binary search until it converges.

PREREQUISITES:

Segment Tree and Implementation would do.

PROBLEM:

Given N people numbered 1 to N and M conflict pairs of people, answer Q queries of format. Given K pairs in a query namely $[L_i, R_i]$ inviting all people in range $[L_i, R_i]$ to party. Determine if all people invited to party get along or not.

QUICK EXPLANATION

- The brute solution to answer each query in $O(N+M)$ time can be easily written. This solution takes $O(Q*(N+M))$ in the worst case if all queries consist of a single interval of people invited and shall time out.
- We can develop another solution answering each query in $O(K^2)$ time by iterating over every pair of intervals in the query and checking if both of the intervals contain a pair of people who do not get along. If it holds true for any pair of intervals in query, answer is NO, otherwise YES. This solution too won't work as it may take time to answer if there are too many intervals in few queries.
- To find out whether any pair of an interval of people contains any conflict pair, we shall use segment tree operations.
- Now combine the above two solutions, using the first solution when K is large, and the second solution when K is small, to fit the overall solution into the time limit.

EXPLANATION

The solution to this problem is based on merging two slower solutions to fit the time limit, so let us discuss both solutions.

Brute Solution:

To answer each query in $O(N+M)$ time, we can simply maintain an array included, marking all the elements present in intervals and then iterating over all conflict pairs to check if both persons in any pair are included or not. This solution performs well when there are few queries with a high number of intervals.

Segment Tree Solution:

We can also build a solution which, for each query check all pairs of intervals to see if the first member of any conflict pair lies in the first interval and second person of conflict pair lie in the second interval of selected pair. If we tried to maintain a set of conflict pairs, it shall time out. We need something smarter.

Let us store these queries and answer them all together efficiently.

What we can do is, to build a range max Segment tree initially filled with -1 and start considering each person one by one. While considering yth person, Consider all conflict pairs (x, y), $x < y$ and update these positions with value y.

Now, whenever we have an ith interval in query $[L, R]$ where R is the person being considered, we can check all intervals ending before L and check if range max of any interval is always strictly smaller than L. The reasoning is as follows.

Considering only first i persons, when we reach the ith person, we update all positions having a conflict with the ith person with value i. Now, If any interval has range maximum x if implies that in that interval, there is a person having a conflict with the xth person. So to check if any conflict pair is there, we iterate over all intervals ending before the current interval and check if this interval has a conflict with any person having index $\geq L$. (Meaning people in the current interval). This compares all unordered pairs of intervals and thus, runs in $O((N+M)*log(N)+\sum K^2)$ to answer all queries.

This solution won't work due to square factor and shall time out when there are few queries with a large number of intervals.

Merging solutions:

Let us define a limit SQ. We use brute solution when $K \geq SQ$ achieving worst case when $K = SQ$ taking $O(Q * (N + M))$. This may seem large, but here, $Q * SQ \leq 2 * 10^5$

We shall use the second solution when $K < SQ$ as our second solution is good in answering queries when K is small. If all queries are answered by this solution, it takes $O((N + M) * log(N) + \sum K^2 * log(N))$ time. But, since $K < SQ$, $\sum K$ cannot exceed $2 * 10^5 * SQ$.

By choosing a reasonable value of SQ, we can achive $O((N + M) * Q/SQ + (N + M) * log(N) + Q * SQ * log(N))$.

Implementation hints:

It is useful in the second solution to sort the intervals beforehand so you only need to check intervals indexed up to the current interval. Additionally, do not forget to consider the case when a conflicting pair lies inside a single interval only in the second solution. This can be easily handled by considering the pair of an interval with itself too.

There are quite a number of problems merging solutions, but I don't remember any at present, so please share the link of any problem you know using a similar idea.

PREREQUISITES:

Segment tree, Prefix Sums, Combinatorics

PROBLEM:

You are given an array A of length N, which initially is a permutation of integers from 1 to N.
Find the number of ways to perform N operations wherein each operation we choose any index i and remove A_i from A (which decrements the size of A by 1) ensuring that the array doesn't contain any consecutive monotone triples. Print answer modulo $10^9 + 7$.

EXPLANATION:

Hint 1

At any point, we can choose to remove one amongst at max 4 points, 2 from the start, and 2 from the end.

Hint 2

Try coming up with an expression involving the last 3 elements left in terms of the removed $N - 3$ elements.

Hint 3

Suppose the indexes of the last 3 remaining elements are represented as `L`, `M` and `R`, the number of ways to get:
`L` as the left of the 3 remaining elements is independent of indexes belonging to the set $\{M + 1, M + 2, ..., N\} - \{R\}$ and
`R` as the right of the 3 remaining elements is independent of indexes belonging to the set $\{1, 2, ..., M - 1\} - \{L\}$.

Solution

Suppose after the removal of x elements our array is represented as A_x and the size of A_x = sz_x. We can choose to remove one amongst at max 4 points to reach A_{x+1}:

1. First (,i.e, A_{x_1}) (Easy to see)
2. Second (,i.e, A_{x_2})
3. Last (,i.e, $A_{x_{sz_x}}$) (Easy to see)
4. Second last (,i.e, $A_{x_{sz_x-1}}$)

Proof

Let the present state of the array be represented as B.
Also, note that B must not contain any monotone triples. Consider the first 5 elements of B which must be related as:

1. $B_1 > B_2 < B_3 > B_4 < B_5$ or,
2. $B_1 < B_2 > B_3 < B_4 > B_5$.

Why other elements can't be removed

Suppose the first relation holds (proof for the second will be similar).
After removing B_3 one of the following relations will hold:

1. $B_1 > B_2 > B_4 < B_5$ or,
2. $B_1 > B_2 < B_4 < B_5$.
   both of which have a 3 monotonic triple. Same will hold for other elements as well.

So we have a contradiction, so we can't remove anything except for the mentioned 4 elements.

When second and second last elements can be removed

Let's talk about the second element, the same will follow for the second last element as well.
So this holds when for condition:

1. $B_1 < B_3$
2. $B_1 > B_3$

Now B_3 becomes our second element which can be removed when for condition:

1. $B_1 > B_4$
2. $B_1 < B_4$

...

Terminating Condition

This removal is possible until some index x where for condition:

1. $B_1 < B_{x+2}$
2. $B_1 > B_{x+2}$

when x is even, and

1. $B_1 > B_{x+2}$
2. $B_1 < B_{x+2}$

when x is odd.
,i.e, we can't remove index $x+1$ without removing index 1.

Suppose the indexes of the last 3 remaining elements after removing the remaining $N-3$ elements are represented as `L`, `M` and `R`, the number of ways to get:
`L` as the left of the 3 remaining elements is independent of indexes belonging to the set $\{M+1, M+2, ..., N\} - \{R\}$ and
`R` as the right of the 3 remaining elements is independent of indexes belonging to the set $\{1, 2, ..., M-1\} - \{L\}$.
which means contribution from the removal of a prefix is independent of the contribution from the removal of the suffix.

So we can fix our `M` and find the number of ways to get (`L`, `M`, `R`) as the last 3 elements as follows:
Iterate from index(i) 2 to $N-1$ and consider it as our `M`.
Find the number of ways in which we can get an index $j < i$ as `L` and
let the sum of the number of ways over all valid j:

1. Where $A_j < A_i$ be `mulL[i][0]`.
2. Where $A_j > A_i$ be `mulL[i][1]`.

Find the number of ways in which we can get an index $k > i$ as `R` and
let the sum of the number of ways over all valid k:

1. Where $A_k < A_i$ be `mulR[i][0]`.
2. Where $A_k > A_i$ be `mulR[i][1]`.

Finally the triplet (j, i, k) is valid if either both A_j and A_k are smaller than A_i or vice-versa.

Our answer = $(\sum_{i=2}^{N-1} ((\texttt{mulL[i][0]} \cdot \texttt{mulR[i][0]} + \texttt{mulL[i][1]} \cdot \texttt{mulR[i][1]}) \cdot \binom{N-3}{i-2})) \cdot 3!$.

where $\binom{N-3}{i-2}$ represents the number of ways to choose $i-2$ positions where we remove an index $i <$ `M` out of the total $N-3$ removals and $3!$ represents the arrangement of last 3 remaining elements since they can be removed in any order.

We explain the computation part only for prefixes, computation for suffixes can be done similarly by reversing the array.

Calculating mulL array

Suppose we represent removing the first element as `op1` and removing the second element as `op2`.
Define `ways[i]` as the number of ways to remove the prefix of length i.
Define `range[i]` as maximum index $j(> i)$ such that if index i is the first element, elements in the range $[i+1, i+2, \ldots, j]$, can be the second element if we only choose `op2` and not `op1`.
So:
`ways[0]` = 1
`ways[i]` = `ways[i - 1]` + $\sum_{j=1}^{i-1}$ `ways[j - 1]` (where $j < i$ and `range[j]` $\geq i$). (for $i \geq 1$)

- First term - After removing $i-1$ length prefix, perform `op1` operation.
- Second term - After removing $j-1$ length prefix, perform $i-j$ `op2` operations and then perform `op1` operation.

Calculating range array

Define:
`forward[i][0][0]` as the smallest even index $j > i$ such that $A_j < A_i$.
`forward[i][1][0]` as the smallest odd index $j > i$ such that $A_j < A_i$.
`forward[i][0][1]` as the smallest even index $j > i$ such that $A_j > A_i$.
`forward[i][1][1]` as the smallest odd index $j > i$ such that $A_j > A_i$.

if no such index exists, value = `INF`.

`range[i]` = min(min(`forward[i][i & 1][1]`, `forward[i][1 - (i & 1)][0]`) - $2, N$)

The expression comes from our previous discussion that how long we can keep removing the second element. We can calculate the `forward` array using the segment tree in $\mathrm{O}(N \log_2 N)$ and the `range` array in $\mathrm{O}(N)$ after that.

Calculating ways array

`ways` array can be calculated in $\mathrm{O}(N)$ using prefix sums and range update query.

Finally,

$$\texttt{mulL[i][0]} = \sum_{\substack{j=1 \\ A_j < A_i}}^{i-1} \texttt{ways[j - 1]}$$

$$\texttt{mulL[i][1]} = \sum_{\substack{j=1 \\ A_j > A_i}}^{i-1} \texttt{ways[j - 1]}$$

which can be calculated in $\mathrm{O}(N \log_2 N)$ using segment tree after calculating the `ways` array and in $\mathrm{O}(N)$ using prefix sums and range update query.

ALTERNATE IMPLEMENTATION:

Check the tester's code for reference.

COMPLEXITY ANALYSIS:

So final complexity is $\mathrm{O}(N \log_2 N)$.

PREREQUISITES:

Segment Tree, Priority Queue

PROBLEM:

There are *n* planets in a sequence from 1, 2,…,n and each planet has a certain number of trees with some heights (heights may be same or different).

There are *q* queries and each query maybe of the following two types :

- Type 1 : $1\ l\ r$: First integer specifies that the query is of type 1. Next two integers l and r specify a range of planets from l and r inclusive. You have to remove a single tree having the highest height among all the trees present in planets from range l and r. You have to output the height of that tree.
- Type 2 : $1\ p\ h$: First integer specifies that the query is of type 2. Next two integers p and h specify the planet number and height of a new tree respectively. You have to add the tree to the specified planet p.

NOTE :

- If not a single tree exists for a particular range of planets given in any query of type 1, then output 0 for that particular query.
- There may exist multiple trees with same height on a planet.
- Given a range of planets in any query of type 1, if there are multiple trees with same maximum height then remove the tree present on the lower numbered planet. **For example** : If there are two planets, planet 1 having 1 tree of height 6 and planet 2 also having 1 tree with height 6 and a query is given with l = 1 and r = 2 then remove the tree with height 6 on planet 1.

EXPLANATION:

The solution is a combination of segment tree and priority queue.
Create an array of n max priority queues representing each planet and store the heights of trees on a planet in its respective priority queue.
Pop the first element from each priority queue and store it in an array.
This array will be used to construct a max segment tree, where each node of the array consists of a pair (height, num) where height is the maximum height of the tree of planet num.
Now we have the required data structures ready for processing the queries :

- Type 1 : rangeQuery the segment tree with range l and r to obtain the tree with maximum height. Print the height and then using num, pop the priority queue of planet num and update the segment tree with the height obtained after popping the priority queue. If the priority queue is empty, then update the segment tree with zero.
- Type 2 : rangeQuery the segment tree with range p and p to obtain the tree with maximum height on planet p. If the queried height (x) is lower then h then add x back to the priority queue of planet p and update x with h in the segment tree. Else, just add h to the priority queue of planet p.

This forms the solution to the problem.

PREREQUISITES:

Tries, binary lifting, subtree updates and point queries on a tree.

PROBLEM:

You have N strings, all initially with value 0. Process Q queries/updates:

- Given (i, k, x), add x to the value of all strings whose length-k prefix equals $S_i[1:k]$.
- Given (i, k, T), add a new string to the list that equals $S_i[1:k] + T$ with value 0.
- Given i, print the value of the i-th string.

EXPLANATION:

Let's tackle a slightly easier version of the problem first, without the second query type - i.e, without adding new strings.

Given that we're dealing with strings, and in particular prefixes of strings, the obvious data structure that comes to mind is a trie.
So, let's put all the S_i into a trie.

Now, for the queries:

- Type 1: given (i, k, x), add x to all strings whose length-k prefix equals $S_i[1:k]$.
  Notice that, if we find the vertex u in the trie corresponding to the position of the k-th element of S_i, all valid strings are exactly those that lie in the *subtree* of u.
  So, we'd like to add x to the values of all vertices in this subtree.
  Finding u quickly is not very hard; you can even save that information when you create the trie and just look it up when needed.
- Type 3: given i, print the value of the i-th string.
  Note that the previous query type had us perform several subtree updates; so all we need to do is to quickly get the current value of the endpoint of this string.

After building the trie, our problem has really turned into a problem on a tree: we'd like to add values to certain subtrees, and get the value of certain vertices.
This is a classical problem.

How do I solve it?

Perform an **Euler tour** of the tree, which "flattens" subtrees into subarrays.

Now, subtree updates and vertex queries instead correspond to subarray updates and point queries on an array - which is a well-known problem that can be solved in several ways.
For example, you can use a segment tree with lazy propagation; or even just a normal segment tree/BIT built on the prefix sums of the array instead.

Now, let's try to incorporate the second query type as well.
Once again, let's construct a trie of all the S_i.
The type 2 query (i, k, T) then corresponds to the following:

- Find the position of the k-th character of S_i in the trie.
- Then, insert T into the trie, starting from this position.

The second part is easy once we've done the first, so let's focus on the first instead.
Suppose the length of S_i is L.
Then, the k-th character of S_i from the front is the $(L - k + 1)$-th character of S_i from the back.
Moving backwards in the trie is easy; it corresponds to just moving to the parent.

So, if know the vertex u which is the endpoint of string S_i, we're really just looking for the $(L-k+1)$-th ancestor of u, which can be found quickly using binary lifting.

However, notice that this process cannot really be done online, since it would change the structure of the trie (and hence the Euler tour will change).

Instead, we can process all insertions *offline*.

That is,

- Read all the queries initially, but don't do anything with them just yet.
- First, process all type 2 queries only, to build a single large trie.
- Build the Euler tour of this trie; and a segment tree on the tour.
- Next, go over the queries once again, but this time:
  - Type 1 queries correspond to subtree additions, as before.
  - Type 3 queries correspond to point queries, as before.
  - Type 2 queries however, correspond to point-set updates.
    Notice that since we're using the single megatrie, some values might've been added to vertices even though they technically don't exist yet.
    To counter this, you can reset the values of such elements to 0, which can still be done using the segment tree.

TIME COMPLEXITY

$\mathrm{O}(S \cdot (26 + \log S))$ per testcase, where S is the sum of lengths of all input strings.

PREREQUISITES

Number Theory, Segment Tree

PROBLEM

Given an array A of length N, answer queries of the following form.

Given interval L, R, consider all subsequences of values in subarray $A[L : R]$ and write down their product. The answer to the query is the smallest positive integer not written down. The product of empty subsequence is 1.

QUICK EXPLANATION

- Only the powers of primes matter, we can discard the rest of the elements.
- Processing the queries offline, and maintaining the right end pointer at p, let's maintain function $f(x)$ denoting the rightmost position r such that there exists a subsequence of subarray $A[r, p]$ with product x. We need to maintain this $f(x)$ only for prime powers as well.
- To answer query L, R, we need to find smallest x such that $\min_{y=1}^{x-1} f(y) \geq L$ and $f(x) < L$ which can be found using binary search, if we store $f(x)$ as the value at x-th leaf in segment tree.

EXPLANATION

Crucial Observation

Only the powers of prime numbers matter. Answer to any query can only be of the form p^a, where p is prime and $a \geq 1$.
Proof: WLOG assume answer to some query is $X = p^a * q^b$ where p and q are prime and $a, b \geq 1$. So we assume that X doesn't appear as a product of some subsequence in the query range.

This means that values p^a and q^b both appear as products of a subsequence in the query range. Let's consider subsequence S_1 with product p^a and subsequence S_2 with product q^b. Since $gcd(p^a, q^b) = 1$, subsequences S_1 and S_2 do not have any common element.

Let's take the union of S_1 and S_2, and find its the product. The product of $S_1 \bigcup S_2$ shall be $p^a * q^b$, which is same as X. Hence, we have a contradiction since X is the product of subsequence $S_1 \bigcup S_2$.

Therefore, the answer to some query can only be of the form p^a where p is a prime and $a \geq 1$

Upper Bound on Answer

Considering a query on range L, R, we can see that there can be at most $R - L + 1$ primes in a range. In the worst case, all N entries of the array are distinct primes. The MEX in this case can be equal to the value of $(N + 1)$-th prime. Hence, the value of $(N + 1)$-th prime, which is 1299721 for $N = 10^5$, shall form an upper bound on the answer. Let's call $MX = 1299721$

Handling Prime Powers

Now we can discard all 1s and all elements which are not prime powers. Let us fix a right end R. The answer to the query L, R shall be the smallest prime power x such that it is not possible to make product x from a subsequence of subarray $A[L : R]$.

To check whether we can make product x from a subsequence of subarray $A[L:R]$, let's determine the largest position $R^{'}$ such that a subsequence of subarray $A[R^{'}:R]$ has product x. Let's do this for each prime power x.

Defining $f_R(x)$ as the largest position $R^{'}$ such that subarray $A[R^{'}:R]$ has a subsequence with product x. $f_R(x) = -1$ if there's no such $R^{'}$.

Assuming 1-based indexing, We know that $f_0(x) = -1$ for all x. Let's figure out a way of computing $f_R(x)$ from $f_{R-1}(x)$. Only new element to be added is $A_R = p^a$.

$f_R(x) = f_{R-1}(x)$ if x is not a power of p. So we only need to update powers of p. We can see that following updates happen

- $f_R(p^a) = R$
- $f_R(p^a * x) = max(f_{R-1}(p^a * x), f_{R-1}(x))$ where $x = p^b, b \geq 1$

The first update happens only for $x = p^a$, and the second update happens for powers of prime p. Since we don't care about $x > MX$, there can be $log(MX)$ such positions, where the second update happens. So we can update one by one.

Handling Queries

Now that we have $f_R(x)$ computed and we need to answer query on subarray $A_[L:R]$, we need smallest x such that $f_R(x) < L$ and $f_R(y) \geq L \forall y < x$.

Earlier, we visualized stored $f_R(x)$ in an array, but we need a data structure to handle point updates and answer prefix minimum queries. Segment Tree is a perfect data structure.

Hence, Let's build a segment tree where x-th leaf stores $f_R(x)$, and the query answers the minimum of a range. By binary searching, we can find smallest x with $f_R(x) < L$. The segment tree needs to have at most MX leaves.

TIME COMPLEXITY

The time complexity is $O(MX * log^2(MX))$ per test case.

PREREQUISITES:

Mo's algorithm on tree or Heavy-Light Decomposition , Randomization.

PROBLEM:

You are given a tree with N nodes, where a value is assigned to each node, given in array A. You have to process following updates and queries.

- Update value assigned to node p to X.
- Consider all values assigned nodes on the path from L to R on the path from u to v. Say the number of nodes is L. Print Yes, if the values under consideration form a permutation of natural numbers in the range $[1, L]$.

EXPLANATION

First of all, assign random numbers > 0 to each value from 1 to N and for each value in the array A, replace x by the random value assigned to x.

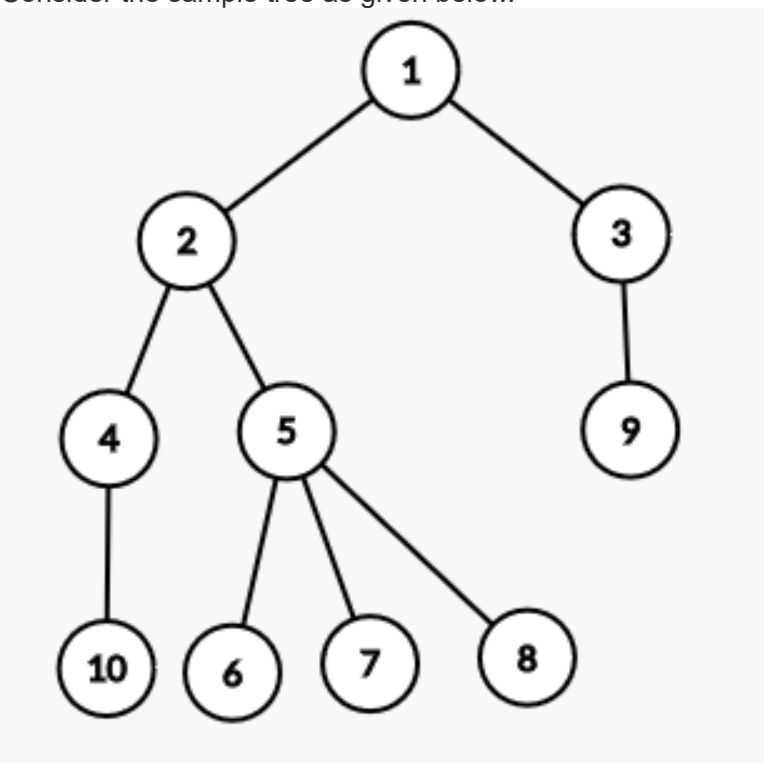
Let us consider we need to answer query (u, v) such that there are L nodes on the path from u to v. Consider the xor of all values on the path from u to v. If the values on path form a permutation of first L natural numbers, then this xor value shall be equal to the xor of values assigned to each i from 1 to L. Xor of values assigned from 1 to i can be easily precomputed using prefix xor for each i from 1 to N. So, the answer to our query becomes Yes, if the xor of numbers on the path from u to v is the prefix xor up to position L.

This gives us the means to check if values of nodes on a path form the permutation or not, if we can somehow implement a data structure capable of updating value for a position, and finding xor of values on a path.

We can either use Heavy-Light Decomposition but there's a simpler and efficient solution using Euler tour and Mo's algorithm on trees.

Since the inverse of xor operation is xor operation itself, we can see, that xoring with a number twice doesn't affect the initial value.

Consider the sample tree as given below.



The Euler tour for this Tree shall be

1 2 4 10 10 4 5 6 6 7 7 8 8 5 2 3 9 9 3 1

We can see, that for query (u, v) (Assuming $ST_u \leq ST_v$ where ST denote start times and EN denote end times.), if LCA of u and v is u, then interval $[ST_u, ST_v]$ contains all nodes on path from u to v exactly once, while all other nodes either twice or not at all (ST being . Since xor operation cancels itself, presence of any node twice negates its presence, and we are only left with nodes on the path from u to v. This way, we can get xor of value on the path from u to v as xor of an interval.

Similarly, if LCA of u and v is not u, we need to consider LCA node separately, and remaining nodes on the path from u to v appear exactly once in the interval $[EN_u, ST_v]$.

For updates, we just need to update the start and end position of a node with the random value assigned to value given in update in the segment tree.

PREREQUISITES:

- Euler Totient Function
- Sieve of Eratosthenes
- DFS
- Euler Tour of Tree
- Range Data Structure (Segment, Fenwick)

DIFFICULTY:

EASY-MEDIUM

PROBLEM:

Given a function $F(x)$, where $F(x)$ is defined as -

$$F(x) = \sum_{i=1}^{x} i * gcd(i,x)$$

You are given a tree rooted at Node 1, you need to answer queries, the queries are of two types -

$Type-1 \rightarrow 1\ i$ - Find the sum of F(x) over all the nodes in the subtree rooted at node i, where x is the value of the node.

$Type-2 \rightarrow 2\ i\ val$ - Update the value of the node i to val.

EXPLANATION:

First of all we need to simplify the given function $F(x)$, where $F(x)$ -

$$F(x) = \sum_{i=1}^{x} i * gcd(i,x)$$

Brute force approach to compute $F(x)$ is to iterate over all possible values of nodes (1 <= Value <= 10^3) and store the results computed for each such value, so that these can be reused later.

How To Simplify $F(x)$?

For efficient computation of $F(x)$ over values of nodes according to the original constraints we need to do something better.

For simplification of $F(x)$, go through the below links where I have attached the process of how to compute $F(x)$ in an efficient manner.**Proof**

Now after simplification $F(x)$ reduces to -

$$F(x) = (x/2) \times (\sum_{d|x} (x/d) \times \Phi(d) + x)$$

where $\Phi(d)$ is Euler's phi function.

This function can be easily calculated for every 1 <= x <= 10^6 in O($NlogN$) time using Sieve of Eratosthenes.

For each query -

Now , we need to calculate sum of $F(x)$ over all the nodes in the subtree rooted at Node i.
In order to calculate the answer for a query , we already have the precomputed values of $F(x)$, now as a part of brute force -

$Type-1$ We can perform a dfs every time for a query and take sum of $F(x)$ over all the nodes in the subtree of the nodes given in the query i.e O($V+E$) time.
$Type-2$ Updation of a given Node can be done in O(1) time.

But , in this way , we need to perform DFS for each query i.e O($V+E$) for each query which is not fast enough!! 🙁

In order to calculate this value efficiently, we can do **EULER TOUR** of the tree. 😃

And now the problem reduces to finding sum over a given range of intervals for each given node in the query, which can be easily done by building a segment tree or fenwick tree for the intervals in the Euler Tour of the tree as a subtree will always denote some contiguous subarray of the Euler array which stores the values for $F(x)$ for the corresponding node built from Euler Tour , so now for every query -

$Type-1$: For a given node we can easily find the range sums over a given interval that corresponds to the node given in the query. Time Complexity: O($logN$)

$Type-2$: Since we are updating an element over Euler array over which the segment tree is built , updating a element takes O($logN$) time too. 🙂

Pre-Requisites : Segment-Tree , Number Theory , Maths.

EXPLANATION

Problem was pretty straight forward that you want to find product in a range of array and update the elements in the queries of type 2.

where in each update operation you will multiply the number present at any given index by the given number . And in queries of type one you have to answer.

whether the product of numbers in the range is a perfect square or not.

In both of the queries your main point is that you have to find product of numbers in the given range.Thus as per constraints if you will find the product
and try to store in any data type then it will overflow. So to preserve the number you were given another observation that the highest prime factor of any
number of a array or in query is less than 100.

Making use of above given fact this problem can be solved very easily. What we can do is that each array index we can store a vector of length 100 which
will store the prime factorization of the given number for example if number is 12 then vectors initial values will be like {0,0,2,1,0,0,0…} since prime
factorization of 12 =223 so at index 2 we inserted power of 2 and at index 3 we inserted 1 because power of 3 is 1.

This generation of prime factorization
of a number takes log(n) time only you can study about this from here **https://codeforces.com/blog/entry/7262**

Now suppose our array is of 2 size and it is like
12 3. After prime factorization it looks like:-

{0,0,2,1,0,0},{0,0,0,1,0,0}; two vectors will be of full 100 size only but all further zeroes are truncated. now you might know that for finding the product

of two numbers you basically sum the powers of their primes so we find the sum of powers of primes for a particular prime number till 100 index and thus when
we sum these 2 given vectors then it becomes {0,0,2,2,0,0} now if in this final produced vector if all numbers are even(divisible by 2) then the product is a
perfect square. if any value in the produced product vector is odd then the number is not a perfect square.

Thus your main task is to find the sum of powers of a particular prime number in the given range in a query of type 1 from l to r and this is also not a tough task.

For range summation queries and index update query you can use any of the fenwick or segment trees with a vector of length 100 as a node because all primes
are less than or equal to 100. keeping this sum is not a difficult task. finding this sum in a range can be done easily just return vector as answer of a query.

We will use segment trees only for finding the sum of powers of a particular prime in a given range and for update query with multiplying the value at index
i with val we can do easily just by prime factorizing the given value and adding the corresponding powers of this index's prime factor…

For example value at index 3 is 3({0,0,0,1,0,0}) thus after multiplying this value with 9({0,0,0,2,0,0}) then the value becomes 27 which is nothing but
{0,0,0,1,0,0}+{0,0,0,2,0,0}={0,0,0,3,0,0};

Thus this problems becomes very easy all you have to implement is with segment trees that range summation of powers of particular
prime number and source to study that. **https://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/**

PREREQUISITES

Segment-trees , Euler tour

PROBLEM

Given a tree where all nodes are unmarked and rooted at node 1. For a given query you need to find the closest marked node which lies in its subtree and after answering the query you set the node in the query to be marked.

EXPLANATION

First, we will be maintaining an " IN " array and an "OUT" array which will basically store the In time and out - time of the nodes when we do a depth-first search on the given tree. This trick is called **EULER TOUR** .
Now let's consider we are currently answering the query for Node Q . Which has

```
IN[Q] = X

OUT[Q] =Y
```

so for all the descendant nodes of this node will have IN times within the range $[X, Y]$.
so let's maintain an array where the indexes of the array will indicate IN times and will store the level of the node and node number as a pair of integer. Now we can find the minimum within the range $[X, Y]$ and find the node which is closest using a segment-tree or a Fenwick tree by applying "**find_min**" function within the range, and we will update the Q_{th} position of the array with the level of Node Q and Node number after answering the query.

TIME-COMPLEXITY

$$O(N + Qlog(N))$$

For each query we are applying a find_min function within a particular range , and before that we had to do DFS on the tree .

PREREQUISITES:

Dfs, HLD, Segment tree, Lazy Propagation

PROBLEM:

There are 3 type of queries.

- In the first query you have to count the number of black and white nodes and repaint the path with the color having the maximum frequency. (If both have same frequency then do nothing).
- In the second query you have to color the subtree with black.
- In the third query you have to print the maximum value between the sum of white nodes and black nodes.

EXPLANATION:

Here we'll be using the concept of **HLD**.

Let us denote color **white as 0** and color **black as 1**.

So from this convention to calculate the **total number of black nodes**, we have to calculate the sum of the path between x and y.Now calculating the LCA of the nodes, we can get the total $path\ distance$.So,we have $path\ distance$ and $the\ number\ of\ black\ nodes$.

$The\ number\ of\ white\ nodes$ = $path\ distance$ - $Number\ of\ black\ nodes$.

For the query of type 2, We'll be adding the concept of **Eular tour** in our HLD.So that for each subtree update we know its **in_time and out_time**.

For the query of type 3, You can update the array of your decompossed tree up to your choice.In this part we'll be updating the segment tree values as **negative** when we have to color it **white** and **positive** when we have to color it **black**.For the $modify\ path\ sum$ value between the path x and y we'll be getting all the **black and white nodes** together.

So for that, we'll be having a segment tree made of the original values **(positive values)** and each query we get the $path\ sum$ values in between the path of x and y.

Now in the $path\ sum$ value we have all the **positive values** denoting all nodes as **black** and in the $modify\ path\ sum$ value we have both **black and white nodes**.

From this to get the sum of values of black nodes and white nodes:

1. ($path\ sum + modify\ path\ sum$) / 2 = $all\ the\ black\ nodes$
2. ($path\ sum - modify\ path\ sum$) / 2 = $all\ the\ white\ nodes$.

TIME COMPLEXITY:

The time complexity for the problem is: $O(n\ logn + q\ log^2\ n)$

PREREQUISITES:

Hashing, Segment Trees

PROBLEM:

We have a board on which 2 type of queries can be performed.

1. In this we can place a board of some color on a section of the board, the dimensions of which will be mentioned.
2. We have to determine if 2 cells are similar, $i.e$ we have to check whether both the cells have the same colors and of the same frequency.

EXPLANATION:

For the update queries we need to lazily update all the cells starting from top left to bottom right with the hash of the color. For the type 2 query, we need to check if the hash of the two cells are equal.
For this purpose we would maintain a $2D$ segment tree (create a segment tree on every row and then create a segment tree on the already created segment tree).

For the type 1 query, go through all the cells of the $2D$ segment tree that cover the range (*both row wise and column wise*) and update it with the hash. This would take $O(logM.logN)$ time ($i.e$, for $logN$ rows , update $logM$ columns).

For the type 2 query, go through all the rows of the segment tree whose ranges lie in the path from root to the range of the cell mentioned in the query $(X0, Y0)$. Now, for every such row , similarly go through the columns and if you find non zero values in them , push them down to its children, till you reach a cell with no children. In such a case , add it to your current hash value.

Do the above step for both the cells $(X0, Y0)$ and $(X1, Y1)$ and compare their hashes. If those are equal print YES else print NO.

TIME COMPLEXITY :

$O(M.N + Q.logM.logN)$

PREREQUISITES:

PREFIX SUM, FENWICK TREE, SEGMENT TREE

PROBLEM:

You are given a circular array $A(|A_i| \leq 10^9)$ of $N(1 \leq N \leq 2 * 10^5)$ integers. $A_{i,j}$ defined as an array formed using elements of A from i^{th} index element to j^{th} index element(both inclusive) in a **clockwise** direction.

A function $f(A_{i,j})$ defined as the count of a subarray of array $A_{i,j}$ whose sum is less than $K(|K| \leq 10^{15})$. You have to find the sum of $f(A_{i,j})$ for all valid i and j under modulo $10^9 + 7$.

QUICK EXPLANATION:

- The solution is explained here in $1-$ based indexing.
- Let, first solve the problem for the non-circular array. For that find the number of subarrays that end at i^{th} index and have the sum less than K. It can be done by finding the count of indexes of the prefixes that have ended before it and have the sum such that if we subtract that from the ith index prefix then it evaluates less than K.
- Suppose there are D_i prefixes $d_{i,1}, d_{i,2}, ..., d_{i,D}$ are of the type above mentioned for index i and the contribution of all the subarray who ends at index i is then $\sum_{j=1}^{D_i} d_{i,j} \ * (N - i + 1)$.
- Then the answer to for non-circular array can be written as $\sum_{i=1}^{N} \sum_{j=1}^{D_i} d_{i,j} \ * (N - i + 1)$.

EXPLANATION:

Let's solve the problem in two parts:

1. Calculating the contribution of subarray which constructed by eliminating the two prefixes of the given array.

   As there are $(n * (n + 1))/2$ subarray of this type so we could use the same technique mentioned in the quick explanation. Let's suppose two prefixes ending at i^{th} and j^{th} has sum s_i and s_j such that $j < i$. Then how many subarrays (consider all possible subarray including circular one) consist of this as it's a subarray. The answer to this is given by the following sum:

   - $1 + 2 + ... + (N - i + j + 1)$
   - $((N - i + j + 1) * (N - i + j + 2))/2$
   - $((N - i + 1) * (N - i + 2) + (2 * N - 2 * i + 3) * j + j * j)$

   If there are cnt possible values for j then the answer is given by:

   - $(cnt * (N - i + 1) * (N - i + 2) + (2 * N - 2 * i + 3) * \sum_{j \subset k} j + \sum_{j \subset k} j * j)$

   We need to do this process for each index and this can be efficiently done by Fenweek Tree or segment Tree.

2. Calculating the contribution of subarray which constructed by the concatenation of non-empty suffix and non-empty prefix of the given array.

   To do that we calculate the valid prefixes for each valid i such that if we consider the suffix from $(i + 1)$ to N such that sum is less than K. We can do this in the same way as described above. For better understanding refer to the setter solution.

COMPLEXITY:

TIME: $O(NlogN)$
SPACE: $O(NlogN)$

PREREQUISITES:

Segment Tree, Sweep Line

PROBLEM:

Let $f([x_1, x_2, \cdots, x_k])$ be the minimum number of subarrays such that each x_i belongs to exactly one subarray, and it's possible to arrange the subarrays in a way that makes x increasing. For example, $f([5, 4, 1, 2, 3]) = 3$ since we can divide x into three subarrays: $[5], [4], [1, 2, 3]$ then rearrange them to $[1, 2, 3], [4], [5]$.

You are given a **permutation** P of length N. You are also given Q queries of the form L, R. For each query, find the value of $f([P_L, P_{L+1}, \cdots, P_R])$.

EXPLANATION:

For each query $[L, R]$, observe the following:

- In the optimal division, we only separate two consecutive elements into two subarrays if they are **not** consecutive values. For example, in $f([1, 3, 2])$, the optimal division is $[1], [3], [2]$, and we separate 1 and 3 because they are not consecutive values (there is 2 in between). We call such pair of consecutive elements *bad*.
- Calculating $f([P_L, P_L + 1, \ldots, P_R])$ is equivalent to counting the number of bad pairs in that range, since we only separate into a new subarray at each bad pair.

Therefore, the problem now turns into calculating the number of bad pairs in each query (inversely, counting the number of good pairs). This is still difficult to do directly, but notice the following:

- Suppose for some i such that $P_i < P_{i+1}$, we calculate $U_i < i$ to be the largest index such that $P_i < P_{U_i} < P_{i+1}$, and $V_i > i + 1$ to be the smallest index such that $P_i < P_{V_i} < P_{i+1}$. Then for any query L, R, (P_i, P_{i+1}) is a *good* pair if and only if $U_i < L \leq i$ and $i + 1 \leq R < V_i$. This is because any query with $L \leq U_i$ will include U_i in its subarray, and therefore make the pair bad; similar things can be said about queries with $R \geq V_i$.

We can find U_i and V_i for all pairs using segment tree (details is left as exercise for readers). Having all these information, the problem becomes counting the number of indices i such that the rectangle $[U_i + 1, i] \times [i + 1, V_i - 1]$ covers the point (L, R). This is a classic problem that can be solved using sweep line and segment tree.

TIME COMPLEXITY:

Time complexity is $O((N + Q) \log N)$ per test case.

PREREQUISITES:

Observation, **RMQ** or **Segment Tree** .

PROBLEM:

Given a string S of length N, you have to answer Q queries of the following form.

For each query, given two integers l and r, determine whether there exists any substring of String $S_l, S_{l+1}, \ldots S_r$, which is dominated by any character c. A string is dominated by a character c if c occur strictly more than half the length of string.

EXPLANATION

This editorial would be mostly observation, with little to code.

First of all, let's assume $S[l, r]$ denote substring $S_l, S_{l+1}, \ldots S_r$.

Let's assume substring T with length x is dominated by some character c. WLOG assume x is even, this can be extended to odd x too. Let's split this substring into two equal parts of length $x/2$ each, let's call them left and right part.

Assume both left and right part are not dominated. Then, the maximum number of times c can occur in both strings is at most $x/4$ each, hence, we can have at most $x/4 + x/4 = x/2$ occurrences of c at most, which implies that string T is not dominated by any character.

But this contradicts our assumption that T is dominated by character c.

This leads us to the conclusion that if a substring S is dominated, either left or the right half of the string is also dominated.

Hence, if we consider all substrings of $S[l, r]$ with $length >= 6$, we can always have a string, it just suffices to check whether left or right half of string is dominated or not.

For strings of length 4 and 5, we can check whether they contain any substring of length 3 which is dominated or not as there cannot exist any substring of length 4 or 5 which is dominated, but no substring of such string of length 3 is dominated. (Similar proof as above). This cases arose because problem imposes the restriction on the minimum length of the dominated string.

Hence, to conclude from above, if a dominated substring T of length $x > 3$ exists, there also exists a substring with length 3 as a substring of T which is dominated.

So, all we need to do is consider each start point p and check whether substring $S[p, p + 2]$ is dominated or not. Let's precompute this.

Let's answer queries now. For each query, we are concerned for substring $S[l, r]$. If the length of this substring is less than 3, the answer is NO directly. Otherwise, we can consider all start points in the range $[l, r - 2]$ and if any string is dominated, substring S contains a dominated string, otherwise, no dominated string exists in this substring.

Noticing carefully, we need a range data structure, which can determine the range OR of range $[l, r]$. Any RMQ data structure such as sparse Table and Segment Tree would work.

Bonus What if there are updates, each update changing a single character at a specified position to c.

TIME COMPLEXITY

Time complexity is $O((N + Q) * log(N))$ per test case.

PREREQUISITES:

Probability, segment tree

PROBLEM:

Chef has an array A of N elements and he will perform the following operation K times:

1. Select a subarray uniformly at random.
2. Reverse the subarray.

Help him find the expected value at every index of the array A after he performs the operation K times.

Print the values as $(P.Q^{-1})$ mod $10^9 + 7$, where Q^{-1} denotes the modular multiplicative inverse of Q (modulo $10^9 + 7$).

QUICK EXPLANATION:

We can first solve the problem for a single swap, and then repeat the algorithms K times for final answer.

$expVal[i] = P(A[i]$ is not swapped$)*A[i] + P(i$ and j are swapped$)*A[j])$

EXPLANATION:

N^2 solution for a single swap:

There are 2 cases that we need to handle:

1. For 2 indices i and j, such that i <= j, the number of subarrays that if swapped, lead to i and j being swapped are min(i, $N - j + 1$) (As the end points of subarray should be equidistant from both i and j.

2. For an index i, the number of subarrays that if swapped, do not affect i are $totalSubarrays - i * (N - i + 1)$. ($i$ is in all the subarrays that have the left end $\leq i$ and right end $\geq i$). Note that i swapped with itself is considered in the first case.

This gives an O(N^2) solution for a single swap.

To reduce this to O($Nlog(N)$), we need to reduce the complexity of first step:

1. For an index i, consider all indices j, such that j >= i. We can divide the expression min(i, $N - j + 1$) into 2 parts:

   1. min(i, $N - j + 1$) = i

   2. min(i, $N - j + 1$) = $N - j + 1$

   For the 1st case, the range of j is $i \leq N - j + 1$, ie, $j \leq N - i + 1$. We can update the expected value of all these indices using range update on segment tree, ie update expected values from $i + 1$ to $N - i + 1$ by $i * A[i]$ (ie, $numSubarrays * A[i]$).
   For the 2nd case, we can update them when we iterate over to them.

2. For an index j, consider all indices i, such that j > i. We need to update the expected value at index j, by all the elements in 2nd part of the above step, ie all i such that $i > N - j + 1$. So now we need to update the expected value at index j by $(N - j + 1) * A[i]$ ($N - j + 1$ is the number of subarrays). This can be done by a range sum query over i. ie, $sum = query(N - j + 1$ to $j - 1)$,

$$expectedVal[j] += sum * (N - j + 1).$$

The above steps are for $i \leq j$, so have to reverse the array and do the same. (taking care of equality)

The 2nd part (index not swapped) is same as that of O(N^2) algorithm.

Now, we have to divide by total number of subarrays and print the solution in required format.

This is O($Nlog(N)$) solution for 1 swap. Now we have to replace the array A by the $expectedVal$ array, and repeat the algorithm $K - 1$ times.

PREREQUISITES:

Segment Tree /Fenwick Tree , Cummalative Sum Techniques, DFS and Maths

PROBLEM:

You have given a tree with N nodes. There are K special nodes called police stations which are situated at nodes $V_1, V_2, \ldots, V_K$, each special node has some *radius of efficiency* R_i. The ith police station increases the security level of node X by $max(0, R_i - distance(X, V_i))$. Initially, the security level of each node is 0. For each node, you have to compute the final security level of the node after considering the effect of each police station on it. Also for any station, it is observed that S_{V_i} = R_i, where S_X is the final security level of X.

QUICK EXPLANATION:

- For any 2 stations V_i and V_j the distance between them is $> max(R_i, R_j)$.
- From the above observation, you will observe the condition $R_1 + R_2 + \cdots + R_K \leq 2 * N$ to be true.
- Root the tree at node 1
- Observe what will happen when you increase the security level of the nodes with $val + d$ and subtracted the corresponding depths for the subtree of station X, where val is the radius of station X and d is its depth.
- It is mentioned that for each station we will increase the security level of other nodes by $max(0, R_i - distance(X, V_i))$. So what can we do if $depth[X] > depth[station] + R_i$. so to handle this properly we need only those values whose magnitude is $\geq depth[X]$. To compute their sum we can maintain a data structure($SegmentTree$ or $FenwickTree$) which stores all the updates which are done on the **ancestors** of this node X including itself and provide the range sum of the values which is $\geq depth[X]$. Also, note suppose you do 2 operation of above types which are valid for node X(i.e. magnitude of their value is $\geq depth[X]$) so in that case, you have to subtract the $depth[X]$ of the node 2 times from there sum. So we have to maintain another data structure($SegmentTree$ or $FenwickTree$) which stores all the updates which are done on the **ancestors** of this node X including itself and provide the range sum of the count of values which is $\geq depth[X]$.
- ```
  ans[X] = sum_of_values_greater_than_depth[X] - (depth[X] * count_of_values_greater
  ```
- Let us represent an update as $(X, depth[X] + R, count)$ where the update is applied to the node X and value to be added is $(depth[X] + R)$ and it is added $count$ times.
- To compute the answer for node X you will need only the update of the station which is either ancestor of X or X itself. So we will maintain only 2 data structure for all the nodes and as we move down the DFS we will perform all the updates associated with node X, $(X, value + depth[station], count)$ to introduce its effect to nodes in its subtree, and when we move up we will perform the update as $(X, value + depth[station], -count)$ to negate its effect.
- To update the nodes for the security increase for station X, outside of its subtree you can do the following operation.
  - To update the nodes of the subtree of station X for the effect of station X we can add an update to node X as $(X, depth[x] + R, 1)$.
  - To update the nodes of the subtree of $Par[X]$ for the effect of station X we can add an update to node $Par[X]$ as $(Par[X], depth[Par[X]] + R - 1, 1)$ and as we don't want to effect subtree of X with this update we add an update to node X to negate its effect as $(X, depth[Par[X]] + R - 2, -1)$ (note the count is -1). Then we will assign X = $Par[X]$ and R = $R - 1$, and continue the process until $R > 1$ and there is some parent of current X.
  ```

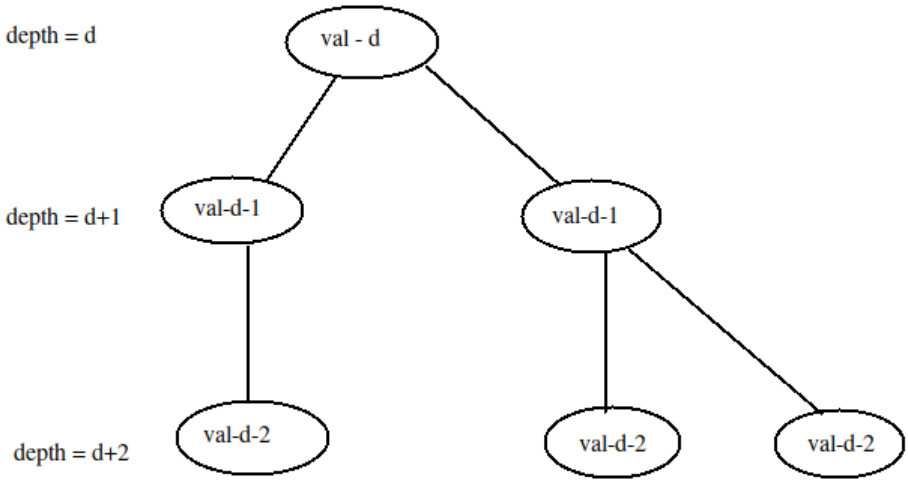
EXPLANATION:

If you observe the line "She already finished her job and noticed an interesting coincidence: for each valid i, $S_{V_i} = R_i$" where S_X represents the security level of node X. You can deduce that for any 2 stations V_i and V_j the distance between them is $> max(R_i, R_j)$. If you try to exploit this condition a little bit more you will observe the condition $R_1 + R_2 + \cdots + R_K \leq 2 * N$ to be true.

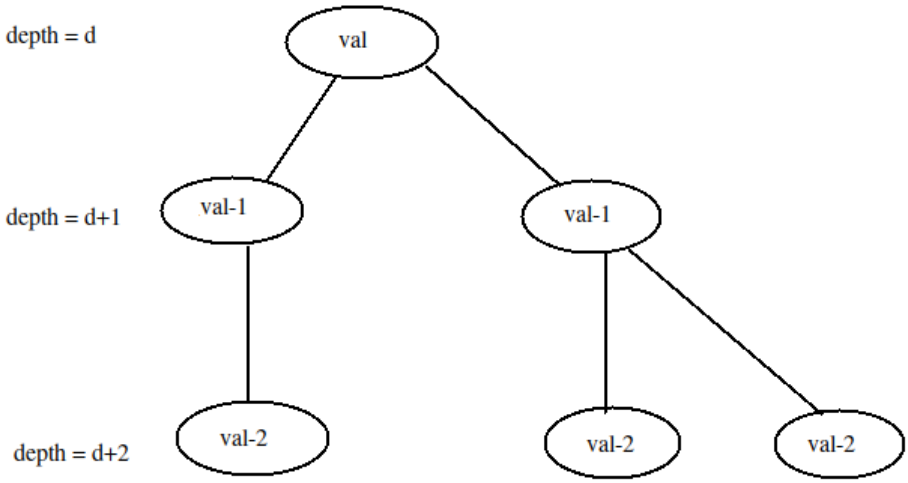
Proof

Now let us see how we can exploit this condition to solve our problem.

Let us first root the tree at node 1 so we can introduce a parent-child-subtree relation in the tree.
Suppose we have a way to increase the value of subtree of node X by val. Now observe what will happen to the values of the node if we subtract it by there corresponding depths.



Now, what if instead of updating the value of subtree of node X by val, we have updated it with $val + d$, where d is the depth of node X and subtracted the corresponding depths.



Doesn't it seem similar to the update we are trying to do in the problem if X is considered as a station and val as R_i?

Now there are some conditions to note before applying the above operation.

- It is mentioned that for each station we will increase the security level of other nodes by $max(0, R_i - distance(X, V_i))$. So what can we do if $depth[X] > depth[station] + R_i$. so to handle this properly we need only those values whose magnitude is $\geq depth[X]$. To compute their sum we can maintain a data structure($SegmentTree$ or $FenwickTree$) which stores all the updates which are done on the **ancestors** of this node X including itself and provide the range sum of the values which is $\geq depth[X]$. Also, note suppose you do 2 operation of above types which are valid for node X(i.e. magnitude of their value is $\geq depth[X]$) so in that case, you have to subtract the $depth[X]$ of the node 2 times from there

sum. So we have to maintain another data structure($SegmentTree$ or $FenwickTree$) which stores all the updates which are done on the **ancestors** of this node X including itself and provide the range sum of the count of values which is $\geq depth[X]$.
So the final formula for a node X to compute its value is

```
ans[X] = sum_of_values_greater_than_depth[X] - (depth[X] * count_of_values_greater_t
```

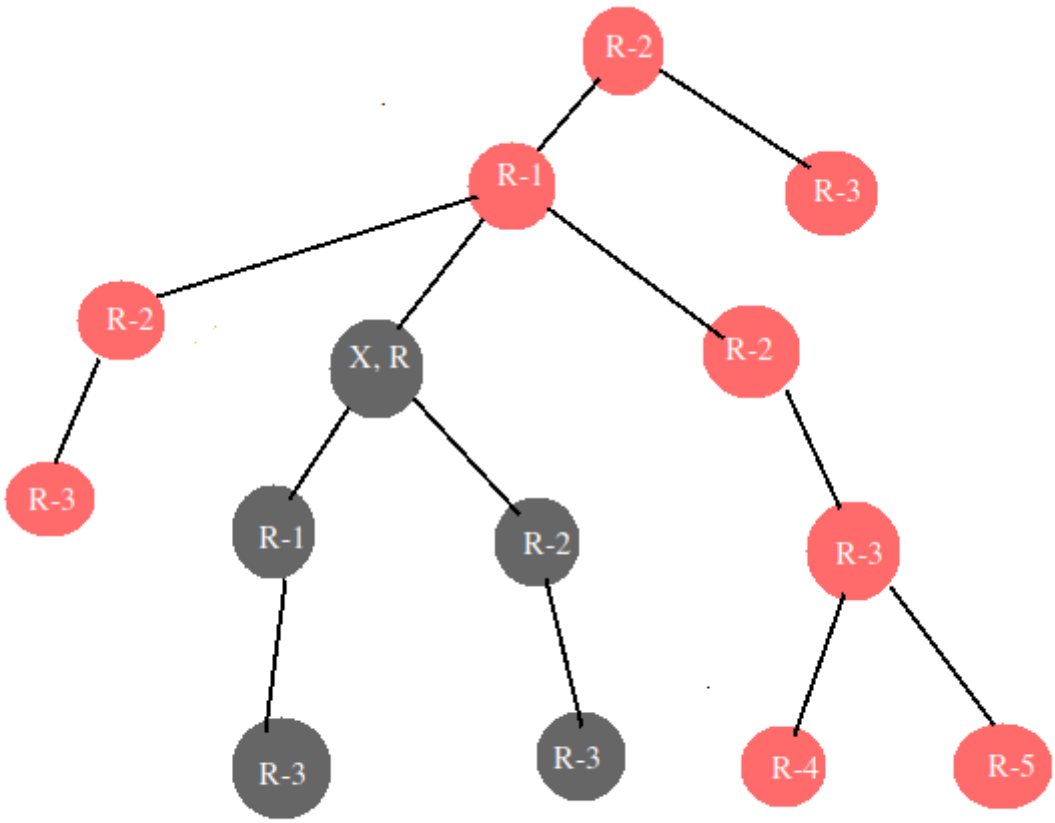
- So each update can be represented as $(X, value + depth[station], count)$ where the count is the number of times the $(value + depth[station])$ is added and X for whose subtree we are applying this update. So for station this update will look like $(V_i, R_i + depth[V_i], 1)$

- Another difficulty which it posses is how to maintain 2 data structures for each node and how to update them,
For this, you will note that to compute the answer for node X you will need only the update of the station which is either ancestor of X or X itself. So we will maintain only 2 data structure for all the nodes and as we move down the DFS we will perform all the updates associated with node X, $(X, value + depth[station], count)$ to introduce its effect to nodes in its subtree, and when we move up we will perform the update as $(X, value + depth[station], -count)$ to negate its effect as the answer for all the nodes in its subtree is computed.

```
for(auto i : updates[node]) { // adding the effect of updates for the node, as it will eff
    long long value = depth[node] + i.first;
    long long count = i.second;
    sum_depths.update(value, count*value); // as this value will impact only those nodes w
    count_depths.update(value, count);
}

for(auto i : updates[node]) { // removing the effect of updates for the node, as answer fo
    long long value = depth[node] + i.first;
    long long count = -i.second;
    sum_depths.update(value, count*value); // as this value will impact only those nodes w
    count_depths.update(value, count);
}
```

- Now you might be wondering that updating subtree is ok, what about nodes outside the subtree of station V_i having radius R_i. As its parent should be increased by $R_i - 1$ and so on(you can see the figure). The black node represents node X and its subtree and Red nodes represent nodes outside the subtree of X.



Note you update these nodes as follow

- To update the nodes of the subtree of station X for the effect of station X we can add an update to node X as
  $(X, depth[x] + R, 1)$.
- To update the nodes of the subtree of $Par[X]$ for the effect of station X we can add an update to node $Par[X]$ as $(Par[X], depth[Par[X]] + R - 1, 1)$ and as we don't want to effect subtree of X with this update we add an update to node X to negate its effect as $(X, depth[Par[X]] + R - 2, -1)$ (note the count is -1). Then we will assign X = $Par[X]$ and R = $R - 1$, and continue the process until $R > 1$ and there is some parent of current X.
- You might think why it will not get TLE because here we exploit the condition we proved above i.e. $R_1 + R_2 + \cdots + R_K \leq 2 * N$.

```
updates[station].push_back({radious, 1}); // as we add values to its subtree.
while((parent[station] != -1) && (radious > 1)) {
    radious --; // as its parent is at distance 1 from its child
    updates[parent[station]].push_back({radious, 1}); // as we add values to subtree of pa
    if(radious > 1) { // but we don't want to add the values to subtree of station again a
        updates[station].push_back({radious-1, -1});
    }
    station = parent[station];
}
```

So by performing the update as we go down and negating the update of node X as we move up the DFS and by applying the $formula$, we can compute the answer.

TIME COMPLEXITY:

- For each node X we are doing $O(\log N)$ operation to compute the answer. So total time to compute the answer for all nodes will be $O(N * \log N)$.
- Also we are doing several updates in $O(\log N)$. The total number of updates are in order of $O(N)$. So total time to perform an update is $O(N * \log N)$.
- So total time complexity per test case is $O(N * \log N)$.

Pre-requisites: Binary Search, Sparse Tables, Segment Tree

Problem:

You are given ascending-sorted array **A[]**, consisting of **N** integers. You are to calculate the answers for **M** queries of the following type: You are given two integers **T** and **D**. You should find the minimal integer **L** such that there is some integer **R** (**L ≤ R**) for which the following conditions are satisfied:

A[L] + D >= A[L + 1],

A[L + 1] + D >= A[L + 2],

…

A[R - 1] + D >= A[R],

A[R] <= T < A[R + 1] (we assume, that **A[N + 1]** is infinitely big).

Explanation:

The first observation, that we should make, is that there is the only **R** for each query **(T, D)**, that could satisfy to the conditions from the statement.

It isn't hard to be sure about that. As far as **A[]** is an ascending-sorted array, let's define **R** as the largest integer, such that **A[R] <= T**. Then **T** is less than **A[R + 1]**(otherwise **R** is not the largest integer that we are looking for).

Please, note that as far as **A[1] <= T**, such **R** always exists.

How to find **R** efficiently? We can use a binary search!

Here is a pseudocode, that shows the implementation of the algorithm of searching **R**.

```
R = 1
LEFT_BOUND = 2
RIGHT_BOUND = N
while ( LEFT_BOUND <= RIGHT_BOUND )
begin
    MID = (LEFT_BOUND + RIGHT_BOUND) / 2
    if ( A[ MID ] <= T )
    begin
        R = MID
        LEFT_BOUND = MID + 1
    end else
        begin
            RIGHT_BOUND = MID - 1
        end
end
```

This subtask works in **O(log N)** time.

OK, now we know how to find **R**. But we were asked to find **L**, not **R**.

So, let's do it.

Firstly, let's fix **R**. We already know, that it's unique for each query and there is no way to choose another **R**, so we need to adapt **L** under **R**.

Let's understand, that if **L = R**, then the conditions from the statement are satisfied. The problem is that **R** may be not the minimal such **L**, that satisfies. On the other hand, we can observe, that if for **L = X** the conditions are satisfied, then for **L = X + 1** the conditions are satisfied as well.

More formally, let's consider boolean function **G(X)**, **1 <= X <= R**. **G(X)** is true if and only if the conditions are satisfied for **L = X**, otherwise it's false. This function is monotone.

Here is a pseudocode, that shows the implementation of the algorithm of searching **L**.

```
L = R
LEFT_BOUND = 1
RIGHT_BOUND = R - 1
while ( LEFT_BOUND <= RIGHT_BOUND )
begin
    MID = (LEFT_BOUND + RIGHT_BOUND) / 2
    if ( G( MID ) == true )
    begin
        L = MID
        RIGHT_BOUND = MID - 1
    end else
        begin
            LEFT_BOUND = MID + 1
        end
end
```

This subtask works in **O(log N * < complexity of calculation of G >)** time.

So, the last subtask, that we should discuss, is how to calculate **G** efficiently.

Firstly, let's forget about this condition: **A[R] <= T < A[R + 1]**. It doesn't play any role as far as **R** is fixed and has already satisfied to that condition.

Let's assume, that we are calculating **G(X)** right now. What should we check?

In order to make **G(X)** true, the following conditions must be satisfied:

A[X] + D >= A[X + 1],

A[X + 1] + D >= A[X + 2],

…

A[R - 1] + D >= A[R].

Let's work a little bit with the inequalities. I.e. let's rewrite them in such a way, that **D** is the only summand to the right of the comparison sign:

A[X + 1] - A[X] <= D,

A[X + 2] - A[X + 1] <= D,

…

A[R] - A[R - 1] <= D.

Let's build up array **B[]**, such that **B[i] = A[i + 1] - A[i]** for each **1 <= i < N**.

Then,

B[X] <= D,

B[X + 1] <= D,

…

B[R - 1] <= D.

We can finally rewrite the inequalities like:

max(B[X], B[X + 1], …, B[R - 1]) <= D.

Well, now it's quite a well-known problem.

G(X) is true if and only if the maximal number of array **B[]** on the range **[X, R - 1]** is not greater than **D**.
We can use a sparse table in order to answer this query efficiently. If you don't know sparse tables at all, don't

be upset. The constraints in this problem are quite soft, you can use a simple segment tree.

Nevertheless, the solution, which uses a sparse table instead of a segment tree, works much faster and has a better complexity.

Here is a pseudocode, that shows the implementation of building up array **B[]** and making sparse table **P[][]** based on **B[]**.

```
for i from 1 to N - 1 do
begin
    B[i] = A[i + 1] - A[i]
    P[0][i] = B[i]
end

for power from 1 to CEILING( LOG_2( N - 1 ) ) do
begin
    for i from 1 to N - 1 do
    begin
        if ( i + 2 ^ power - 1 <= N - 1 ) do
        begin
            P[ power ][ i ] = max( P[ power - 1 ][ i ], P[ power - 1 ][ i + 2 ^ ( power -
        end
    end
end
```

Here is a pseudocode, that shows the implementation of function **G()**.

```
boolean function G(L)
begin
    LEFT_BOUND = L
    RIGHT_BOUND = R - 1 (we assume, that variable R is global)
    LIMIT = D (we assume, that variable D is global too)
    POWER = FLOOR( LOG_2( RIGHT_BOUND - LEFT_BOUND + 1 ) );
    MAX_VALUE = max( P[ POWER ][ LEFT_BOUND + 2 ^ POWER - 1 ], P[ POWER ][ RIGHT_BOUND - 2
    if ( MAX_VALUE <= LIMIT ) do
    begin
        return true
    end
    return false
end
```

The Tester and the Setter have another solution, which uses off-line processing of queries. If you are interested, you can check them out.

PREREQUISITES:

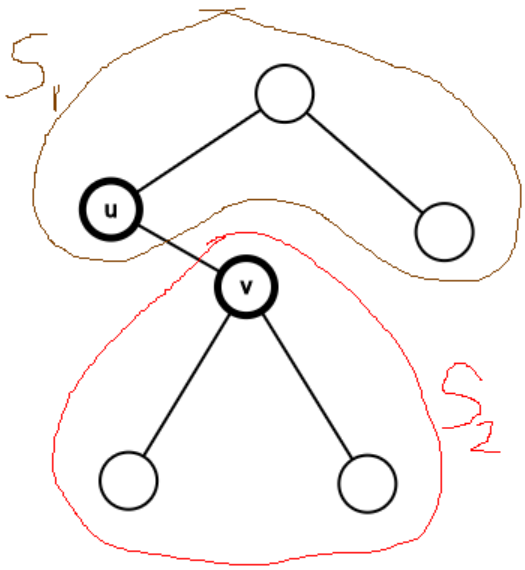
Euler Tour, Segment Trees

PROBLEM:

Given a rooted tree with N nodes. Initially, all nodes have colour black. You are also given Q queries of the following form:

- Determine the number of paths passing through edge $u \rightarrow v$, with both endpoint nodes having the colour black.
- Invert the colour of all nodes in the subtree of node u.

EXPLANATION:



(Leonardo gave the world 'Mona Lisa'; here's my masterpiece 😛)

Let b_1 and b_2 correspond to the number of black nodes in S_1 and S_2 (from above picture), respectively. For queries of type 1, the answer is equal to

$$b_1 * b_2 \implies ((b_1 + b_2) - b_2) * b_2$$

where $b_1 + b_2$ is equivalent to the number of black nodes in the whole graph.

Thus we have reduced the problem to:

- Invert the color of all nodes in the subtree of u
- Determine the number of black nodes in the subtree of u (and subtree of node 1 - to find the total number of black nodes).

This can be solved efficiently using a Segment Tree (with lazy propagation) on the Euler Tour of the rooted tree.

How?

Run Euler Tour on the tree; let tin_u and $tout_v$ be the entry and exit times of node u in the tour.

Consider an array A such that $A_{tin_u} = 1$ if node u is black. Initially, $A_i = 1$ for all valid i.

- For a flip operation (query type 2) on the subtree of u, invert the value of all elements of A in the range $[tin_u, tout_u]$.
- To query the number of black nodes in the subtree of u, sum all elements of A in the range $[tin_u, tout_u]$.

Both operations can be done in $O(\log N)$ per query, using segment trees with lazy propagation.

TIME COMPLEXITY:

Running Euler Tour takes $O(N)$. Building the segment tree takes $O(N \log N)$. Each query takes $O(\log N)$. The total complexity is thus:

$$O(N + N \log N + Q \log N) \approx O((N + Q) \log N)$$

SOLUTIONS:

Editorialist's solution can be found **here**

Experimental: *For evaluation purposes, please rate the editorial (1 being poor and 5 excellent)*

1 2

Quote

Show results

divyanshu_1712 **Sep '21**

> infinitepro:
>
> $b1*b2 \Longrightarrow ((b1+b2)-b2)*b1$

You mean bi*b2 = ((b1+b2)-b1)*b1

PREREQUISITES:

DP, Segment Tree/Fenwick Tree

PROBLEM:

You are given a $N \times M$ matrix A, filled with **distinct** numbers. You can start at any cell. If you are currently at cell (x_1, y_1), then you can jump to any other cell (x_2, y_2), if and only if the following two conditions are satisfied:

- $A_{x_2,y_2} > A_{x_1,y_1}$
- Manhattan distance between A_{x_2,y_2} and $A_{x_1,y_1} > S$.

Let V be the set of the cells visited by you.

Maximize the value of $\sum_{(x,y)\in V} A_{x,y}$.

EXPLANATION:

Let's process the cells in the increasing order of the values. Hence, now we need to find the maximum values in the cell which have been processed and which has a manhattan distance greater than S from the cell where we are standing.

We can solve it by Dynamic Programming. Let's define our DP as:

- $DP(x, y)$ is defined as the maximum value that we can achieve when all the cells whose value is less than $A_{x,y}$ have been processed.
- Transition of DP will be as we saw above i.e:

$$DP_{x,y} = A_{x,y} + maximum$$

where $maximum$ is the maximum value in the cell which have been processed and which has a manhattan distance greater than S from the cell where we are standing.

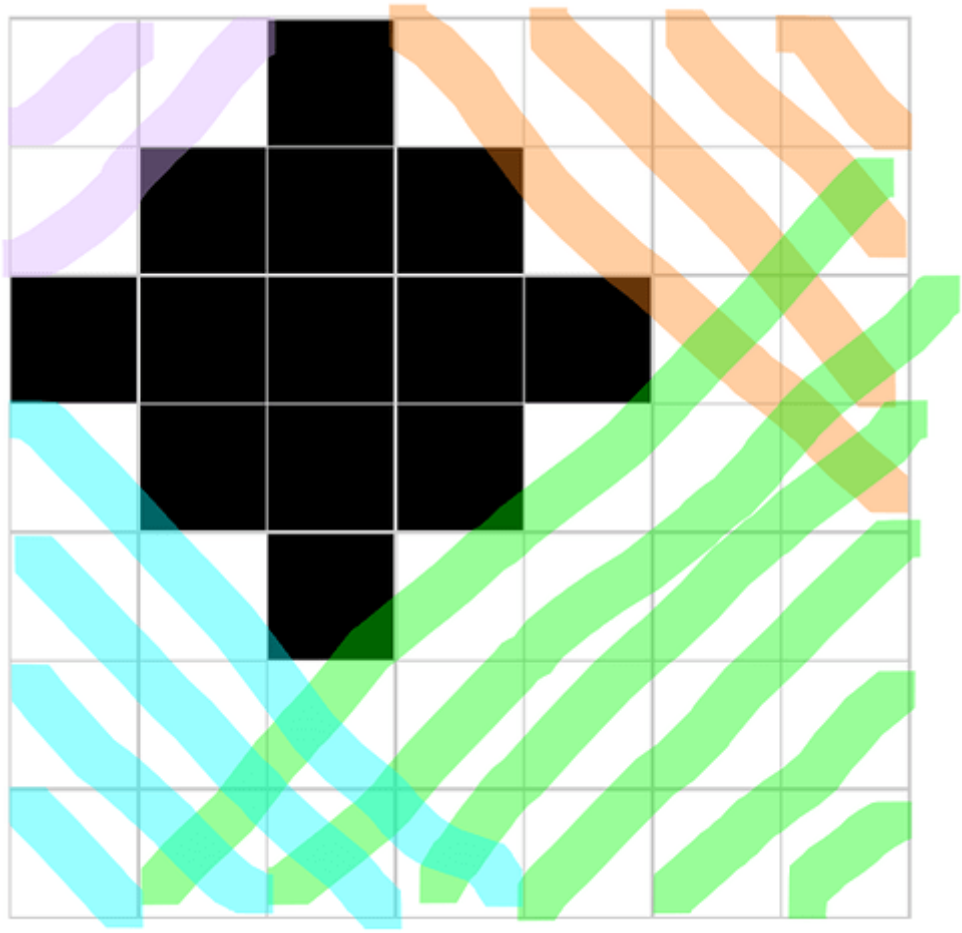
If we iterate over all the cells to find which cells are valid, then it will require $O((N * M)^2)$ complexity which is too high to pass the constraints. Can we do better? Let's see:

Let's consider the given matrix in the four kinds of diagonals.

- All Diagonals that start from the top-left corner.
- All Diagonals start from the top-right corner.
- All Diagonals start from the bottom-left corner.
- All Diagonals start from the bottom-right corner.

Suppose we are at cell (x, y). Now all the cells whose Manhattan Distance is less than or equal to S from the cell (x, y) form a diamond. Credits to **czhang2718** for the image below.

Image



As we can see in the figure, there are 4 types of diagonal which are defined above. Now in order to optimize the above DP let's make a segment tree or Fenwick tree for all four types of diagonal. You can create Fenwick Trees in such a way that the starting index in the tree is the first cell from where the diagonal begins.

Now, instead of traversing all the cells, we can simply find the maximum prefix in all the four types of Fenwick trees we created. Finally, we can update the value of the cell in all of the 4trees. Hence now our transition of DP will be as follows:

$$DP_{x,y} = A_{x,y} + maxprefix(T_1, T_2, T_3, T_4)$$

where T_1, T_2, T_3, T_4 are the Fenwick trees that we created.

This helps in reducing the time for transitions from $O(N * M)$ to $O(4 * log(N + M))$.

Our answer will be the maximum value that we find in the DP.

TIME COMPLEXITY:

$O(N * M * log(N + M))$ per test case

PREREQUISITES

segment tree, sqrt decomposition, offline processing of queries, binary search

PROBLEM

You are given an array A of integers of size N. A block is defined as maximal consecutive subarray with equal values. For example, if $A = [20, 10, 10, 7, 7, 7, 10]$, then there are four blocks in the array, i.e. [20], [10, 10], [7, 7, 7] and [10].

You are given Q queries, each consisting of L, R, K, you have to find number of blocks in the subarray $[L, R]$ that are of length at least K.

Solving first subtask

For solving first subtask, you can find the length blocks in a given subarray in the range L to R in a single iteration over the subarray $[L, R]$. Thus you can also find the count of blocks having length at least K in $\mathrm{O}(|R - L + 1|)$ time, which can be $\mathrm{O}(n)$ in the worst case. So, the worst case complexity of answering all the queries will be $\mathrm{O}(n * Q)$, which will take around $3000^2 = 10^7$ operations, sufficient to run under a second which is good enough for the first subtask.

You can not afford to iterate over all the elements in the range of the query for answering it. You have to either somehow do some preprocessing or build some sort of data structure that can efficiently answer the query in time asymptotically better than $\mathrm{O}(n)$.

A smart solution of $\mathrm{O}(\log n)$ complexity per query

Let us take the example given in the statement, $[20, 10, 10, 7, 7, 7, 10]$. We will create another array B as follows. B_i will denote the number of suffixes of $A[1..i]$ that have equal elements. Intuitively, it denotes the length of the overlap of block consisting A_i with $A[1..i]$. For our above example, the B array will be $[1, 1, 2, 1, 2, 3, 1]$. If we would have to find the number of blocks with length at least K in the entire array, we can notice that this will be precisely equal to number of B_i's equal to K. This is true because if there is a block is of length X, then we will have written numbers $1, 2, 3, \ldots, X$ written as B values at the corresponding indices of block. So, this block will be considered for each of the K's of the query where K is in the range 1 to X. In our example, you can see that there are two blocks of length at least 2 (i.e. the block [10, 10] and the block [7, 7, 7]), which is same as number of 2's in the array B. The number of blocks of length 1 are 3, which is also same as number of 1's in the array B.

Now, what if our query does not span the whole array? Let the query be generic range $[L, R]$. There can be three type of blocks that we will might have to consider.

```
- The blocks that lie completely inside.
- The left side block lying partially in range
- The right side block lying partially in the range.
```

For example, for the arary $[20, \mathbf{20, 10, 10, 7, 7}, 7, 10]$ and the range $[2, 6]$, the block [10, 10] lies completely inside the range, and the left side block $[20, \mathbf{20}]$ lies partially inside the range. Similarly the right stable block $[\mathbf{7, 7}, 7]$ also lies partially inside the range.

For easiness in the explanation, let us assume that there are blocks of type 1 present in the query. Later we will learn how to deal with the other case too.

You can see that for the blocks that are completely inside the range, the number of B_i equal to K will give the correct number of blocks of length K in the range $[L, R]$. What's the case with left and right side partial blocks?

The contribution of right side block will also be counted correctly in this. This is because, all the values in the range $1, 2, \ldots B_R$ are present in the range $[L, R]$. But for the left block, the B_i's don't accurately represent the B_i values if we would have considered the only the subrray $[L, R]$ in the consideration. In the later case, we should have started the B values from $1, 2$ onwards. For example, $B_2 = 2$, but that doesn't mean that block $[20]$ should be counted as of length 2. This block is of length 1. So, it could happen that the left side block is of not length at least K. We handle this case separately by checking whether the left side block is indeed counted in the answer when we calculate the number of K's in the range $[L, R]$, and whether it should be really counted or not. Identifying which is the case is a matter of easy bookkeeping observations that can be made.

Now, remains the case when there is a no block that lies completely inside the range $[L, R]$. This special case is very easy to deal as one just have to check the lengths of left and right blocks.

For a given fixed array, the queries of finding the number of elements in a range L, R that are equal to K can be done in $\mathrm{O}(logn)$ time after preprocessing of the array. For each number x in the array, you can create a sorted list of indices where x appears. After that finding number of elements in the range $[L, R]$, will require you to do two binary searches to find the smallest index $\geq L$ and largest index $\leq R$.

Thus, each query will have $\mathrm{O}(\log n)$ time complexity. Overall time complexity of the solution will be $\mathrm{O}(Q \log n)$.

Alternate view of the problem.

Let us create another array B that denotes the lengths of the blocks of the array A. Number of elements in B will be equal to the number of blocks of the array A. For $A = [20, 10, 10, 7, 7, 7, 10]$, the array B will be $[1, 3, 2, 1]$. From the previous discussion, you have seen that finding the answer for the blocks that completely lie inside the range is difficult. Processing the two left and right blocks is just a matter of small bookkeeping. So, we will see how to find the number of blocks of length at least K that lie completely inside the range $[L, R]$.

Using two binary searches, we can identify the range l, r such that $B[l..r]$ denote the length of blocks that lie inside the range $[L, R]$. We have to find number of elements in the range $[l, r]$ that are at least K.

So, we have to solve the following problem.

Given an array a. You have to queries $[l, r]$, find the number of elements in the range that are greater than or equal to K.

Solution using segment tree

The above problem can be solved by many standard data structures. You can create a segment tree whose each node will store the sorted list of the values in the range. Space and time complexity of creating this structure will be $\mathrm{O}(n \log n)$, where n is number of elements in array a. Answering a query K can be done by going through the nodes in the desired range and doing a binary search in each node of the segment tree to find the number of values in the node that are at least K. This algorithm provides you a time complexity of $\mathrm{O}(\log n^2)$ per query.

Solution using sqrt decomposition.

You can also apply a sqrt decomposition algorithm for it. Divide the array a into $\sqrt{(}n)$ buckets. For each bucket, also maintain a sorted list of values in the bucket. When you have to answer a query in the range $[l, r]$, there are three kind of scenarios that can exist.
- The buckets that lie completely inside the range.
+ You can go through each of the buckets and for each bucket do a binary search to find the number of values in the bucket that are at least K. Number of suck buckets will be $\sqrt{n}$.
- The buckets lie partially in the range.
+ There can be at most two such buckets, the left and right buckets that intersect with the range $[l, r]$. You can simply iterate over the values of these buckets that lie in the range and find the count of values that are at least K. As each bucket is of length $\sqrt{(}n)$. Its time complexity will still be $\mathrm{O}(\sqrt{n})$.

Time complexity of answering each query is $\mathrm{O}(\sqrt{n}\ \log n)$.

This can be optimized to $\mathrm{O}(\sqrt{n})$ by making the following observation. As we have to answer queries offline, we are allowed to shuffle or sort them in any way. We sort them w.r.t K values in the increasing order. For each bucket, we maintain a pointer that points to the current value $\leq k$ in the sorted list of values of the bucket. You can notice while processing a query this pointer will always move in the forward direction for a particular bucket. Hence, in the worst case, the above algorithm will give a amortized $\mathrm{O}(\sqrt{n})$ time complexity per query.

PREREQUISITES:

Segment trees

PROBLEM:

You should perform queries on a sequence A. In each query, you either change an element A_i or need to find the smallest K such that $|A_L - K|, |A_{L+1} - K|, \ldots, |A_R - K|$ is sorted in non-decreasing order (or determine that no such K exists).

EXPLANATION:

Let's reformulate a condition $|A_i - K| \le |A_{i+1} - K|$. There are three distinct cases:

- $A_i = A_{i+1}$: the condition is always true
- $A_i < A_{i+1}$: we need $K \le (A_{i+1} + A_i)/2$
- $A_i > A_{i+1}$: we need $K \ge (A_{i+1} + A_i)/2$

It's easy to see because the condition says simply "K cannot be closer to A_{i+1} than A_i".

In addition, we're looking for $K \ge 0$ and clearly won't ever need $K > M = 10^9$ (M is the upper bound on elements of A). That means we can create and keep updating a new sequence of ranges $[l_1, r_1], \ldots, [l_{N-1}, r_{N-1}]$, where:

- if $A_i = A_{i+1}$, then $l_i = 1$ and $r_i = M$
- if $A_i < A_{i+1}$, then $l_i = 1$ and $r_i = (A_{i+1} + A_i)/2$
- if $A_i > A_{i+1}$, then $l_i = (A_{i+1} + A_i)/2$ and $r_i = M$

For each query of type 1, at most two of these values need to be updated. These ranges determine allowed values of K, so in a query of type 2, we just need to find $x = \max(l_L, \ldots, l_{R-1})$ and $y = \min(r_L, \ldots, r_{R-1})$. Then:

- if $x > y$, there is no possible K
- if $x \le y$, any K in the range $[x, y]$ satisfies the condition; we're looking for the minimum, which is $K = x$

Finding the minima and maxima of ranges, along with updating single elements, is done efficiently by several well-known data structures like segment tree.

PREREQUISITES:

prefix-sum, suffix-sum, sliding window, deque, sparse table, segment tree

PROBLEM:

Given N machines generating chocolates given in a form of array A. Find the maximum amount of chocolates spider man can get if he can take at most $3K$ chocolates by throwing 3 webs each of K size and get all the chocolates in the range selected. He cannot take a chocolate once taken. Spider-man **cannot** his throw on a machine more than once. But he is now given some magic power through which he can revive atmost L machines ($L \leq K$).

QUICK EXPLANATION:

Select a window of K size as a middle subarray and then select one left subarray of size K having maximum sum and then select one right subarray of size K and keep taking maximum of sum of three. For choosing left and right subarray for each middle subarrays there are $K - L + 1$ possibilities by giving a gap of $K - L$ (by taking maximum of all subarrays before left one and maximum of all before right one) between end of pre_max subarray and start of suf_max subarray. We can take max of all these possibilities for each middle array but for selecting this maximum value we can optimise it using deque, sparse-table or segment-tree for getting final time complexity of $O(N)$ or $O(NlogN)$.

EXPLANATION:

Spider-man can choose one subarray of size K and throw his web at it now according to that middle subarray he can choose the maximum sum subarray of size K in his left and his right. Thus making total choosen subarrays equal to 3 now we can choose all such possible subarrays and keep taking maximum of them.

Choosing left subarray and right subarray is the main key here. If we observe carefully then for all chosen middle subarrays we can consider all window of size $K - L$ inside this middle subarray and take sum of pre_max of sum of all subarray of size K ending just before the start of this window and suf_max of sum of all subarray of size K starting just after the end of this subarray and keep taking max of sum of this chosen middle subarray, left subarray and right subarray.

We can pre compute pre_max and suf_max of sum of subarrays of size K by sliding window easily now for second objective we can precompute all subarrays by sliding window of size $K - L$ and taking sum of pre_max ending just before the start of this window and suf_max staring just after the end of this window. Let's call it sub2.

$sub2[i] = preMax[i] + sufMax[i + k - l]$ such that $i + K - L \leq N - K$.

Now, when calculating final answer while traversing middle subarray we can take max of all such possible window inside this subarray in $O(1)$ time using deque. By doing pre computation of $O(NlogN)$ it can be done using segment-tree in $O(logN)$ and in $O(1)$ using sparse table.

For this we follow these steps-

- Create a prefix max array having maximum sum of all subarray ending before index i.
  We can do this by sliding window of size K from $i = 0$ to $i = N - K$ and keep on taking maximum of all windows before and at $i + K - 1$.

- Create a suffix max array having maximum sum of all subarray starting from index i.By doing similar thing of sliding subarray of size K for all $i = 0$ to $i = N - K$ and keep taking maximum of all the windows starting after and at i.

- Create sub2 array such that $sub2[i] = preMax[i] + sufMax[i + k - l]$ such that $i + K - L \leq N - K$.
- Select each possible middle subarray and take keep on taking maximum of sum of middle subarray and maximum of sum of all previous subarray and maximum of sum of all forward subarray. We can do this by deque using **this** way of selecting max of subarrays in a range. We can do same using sparse table using **this** way or segment tree using **this** way.

We cannot choose such subarray under given conditions if $3K - L \geq N - 1$. Since we need at least this size of subarray.

PREREQUISITES:

Segment Trees, LCA queries on Trees

PROBLEM:

You are given a tree with N nodes and Q queries of the following types

- `1 u` Mark the node u as special
- `2 u` Mark the node u as not special
- `3 K` Determine whether there exists a node in the tree (not necessarily special) whose maximum distance to one of the special nodes is exactly `K`

For each query of type `3`, print `0` if the answer is `false` and `1` otherwise. Initially none of the nodes are marked as special.

QUICK EXPLANATION:

Firstly we build a data structure to compute $\text{dist}(u, v)$ between any two given vertices in $O(1)$ time.

For every query of type `3`, it is sufficient to consider only the pair of special nodes that are farthest apart in the tree and the end points of the diameter of the tree. If one of these distances between the four pairs of points is at least K and K is at least $\frac{\text{dist}(u,v)}{2}$ then the answer is `true`, otherwise the answer is `false`.

To include queries `1` and `2`, we need to be able to find the pair of special nodes that are farthest apart quickly and with updates. To do these operations, we can maintain a segment tree, where in each node of the segment tree that has interval $[L, R]$ we store the only the pair of special nodes in the interval $[L, R]$ that are farthest apart.

For merging it is sufficient to consider the farthest pair from each of the pairs of special nodes computed in the children. There are only $O(1)$ such pairs, so merging can be done in $O(1)$.

EXPLANATION:

Queries of type 3

We first look at how to answer operations of type 3 quickly and worry about updates later.

Let's first setup the tree for answering LCA and distance queries between pairs of nodes in $O(1)$ time.

Fast LCA Queries

We first build the **Euler Tour** of the input tree.

Then the LCA of two nodes u and v can be obtained by looking at the interval $[L, R]$ in the Euler Tour of the tree where L is the first occurrence of node u and R is the first occurrence of node v. (Here assume that u occurs first in the Euler Tour, otherwise we can just swap the nodes). The node which has the lowest depth in the interval $[L, R]$ is the LCA of the two nodes u, v.

Now it is sufficient to find build an interval range minimum query data strucutre on the Euler Tour of the tree. If we use sparse tables, we can get a constant query time.

For more details refer **this** blog.

The total preprocessing time is $O(N \log N)$ and query time for LCA is $O(1)$. The preprocessing can even be reduced to $O(N)$ but we need not bother with that optimisation since the constant factors are large anyway.

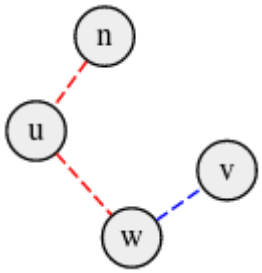
The distance between two vertices $\text{dist}(u, v)$ can then be calculated as $\text{depth}(u) + \text{depth}(v) - 2 * \text{depth}(\text{lca}(u, v))$, which also takes $O(1)$ time.

Starting with the most naive solution, let's find the maximum distance from each node in the tree to any of the special nodes by the following brute force algorithm (we use $\text{maxDist}(n)$ to denote the distance of the farthest special node to node n):

```
maxDist := An array of size n, initialised to -1
for each node u from 1 to n:
    for each node v that is marked special:
        maxDist[u] := max(maxDist[u], dist(u, v))
```

Since the distance queries are $O(1)$, the above code takes $O(n^2)$ time, since there could be $O(n)$ special nodes, however we can show that we need to consider only certain $O(1)$ special nodes.

For example suppose w is a special node that lies on the path between two other special nodes u, v (see diagram below), then any path ending at w can be extended to one of u, v yielding a higher distance. Therefore we can ignore such special nodes while performing the second iteration in the brute force, since they never contribute to the answer.

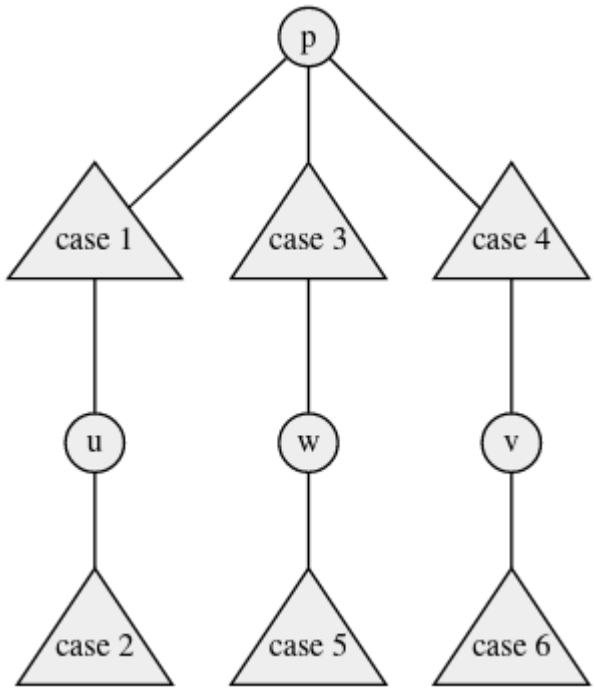


We can extend this idea to see that we need to only consider two special nodes. You should try to find them yourself before reading Observation 1.

Observation 1. Suppose u, v are two special nodes such that $D_s = \text{dist}(u, v)$ is maximum among all pairs of special nodes, then for all nodes n in the tree $\text{maxDist}(n) = \max(\text{dist}(n, u), \text{dist}(n, v))$

Proof

Let u, v be the special nodes with maximum distance between them, and let w be any other special node.
It will be helpful to think of the tree in the manner shown below.



In the diagram node p is the closest node to w that lies on $\text{path}(\text{u, v})$. The triangles represent sub-trees. The diagram can be thought of as a general representation of any tree.

In the preceding paragraphs, we already saw the case when w lies on the path between u, v. Notice that here we have assumed that that is not the case. Also since by u, v is the farthest set of special nodes, w cannot lie in `case 6` or `case 2` (why?). In other words we can assume that p is always different from u, v, w since the other cases are easy to deal with.

We will now argue that given any node n, wherever it occurs in the tree, one of u, v are always farther from n than w.

Note that :

- $\text{dist}(u, v) = \text{dist}(u, p) + \text{dist}(v, p)$
- $\text{dist}(w, v) = \text{dist}(w, p) + \text{dist}(v, p)$
- $\text{dist}(w, u) = \text{dist}(w, p) + \text{dist}(u, p)$

Using $\text{dist}(u, v) \geq \text{dist}(u, w)$ and $\text{dist}(u, v) \geq \text{dist}(v, w)$
we get that $\text{dist}(p, w)$ is the **shortest** among $\text{dist}(p, u), \text{dist}(p, v)$. This fact will be useful for us.

- **Case 1:** $\text{dist}(p, v) \geq \text{dist}(p, w)$ so if n lies in these sub-trees, v is farther than (or the same distance as) w .
- **Case 2:** By assumption, passing through u and then to v gives the maximum distance
- **Case 3:** This is the most tricky case. Consider the last common point between $\text{path}(n, w)$ and $\text{path}(n, v)$, i.e. the point from which the path to v and w from n starts diverging. Let this point be m. Upto m, the path is identical and the distances are same. After m, the maximum possible distance to w is $\text{dist}(p, w)$, and the minimum possible distance to v is $\text{dist}(p, v)$. Thus v is farther than w.
- **Cases 4 & 6:** Same as cases 1 & 2, but with u, v swapped, so the same arguments follow.

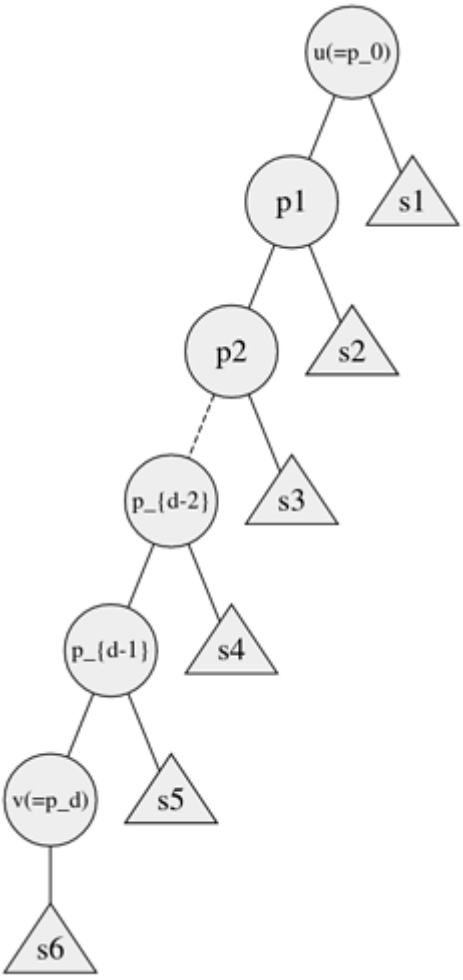
We can see that irrespective of where the node n lies, one of u, v is always farther when compared to w, as desired.

From observation 1, it is clear that to answer the query, we need to determine if there exists a node n such that $\max(\text{dist}(n, u), \text{dist}(n, v)) = K$. however the following observations and case work will help us consider each of

the nodes separately.

Observation 2. There is no solution if $K \leq \frac{D_s}{2}$

Proof



Observe that at least one of the paths from $\text{path}(n, u)$, $\text{path}(n, v)$ contain either the left half or the right half of $\text{path}(u, v)$ completely. Therefore $\text{maxDist}(n) \geq \frac{D_s}{2}$, so if $K < \frac{D_s}{2}$ it is not possible for $\text{maxDist}(n)$ to equal K.

Observation 3. There is always a solution if $\frac{D_s}{2} \leq K \leq D_s$

Proof

Refer the diagram in observation 2. p_K is the desired node. Note that $p_D = v$.

Observation 4. If $K > D_s$ then it is sufficient to find any node that has a distance of **at least** K from one of the two special nodes u, v computed above.

Proof

Suppose the farthest special node from n is u and has a length of at least K. We can shrink the path to size K by removing nodes at the other end of the node u. Since $K \geq \frac{D_s}{2}$, the path we've obtained must contain at least one half of $\text{path}(u, v)$ and so u still remains as the farthest point from n and this completes the construction.

Now it is sufficient to find the farthest node from u and v. Similar to observation 1, we can show that it is sufficient to consider only the endpoints of the diameter of the tree to find the farthest node from each of u, v. The following generalisation of Observation 1, will be useful.

Observation 5. Suppose S and T are two subsets of nodes in the tree. Then for each node in s, the maximum of distance of s to any node t belonging to T is equal to one of $\text{dist}(s, T_u), \text{dist}(s, T_v)$, where $\text{dist}(T_u, T_v)$ is

maximum among pairs of nodes belonging to T.

Note that observation 1 is as special case of observation 5 with $T =$ the set of special nodes and $S =$ the set of all nodes in the tree.

Now consider $S = \{u, v\}$, and $T =$ set of all nodes in the tree. Suppose T_u, T_v are the end points of the diameter of the tree (i.e $\text{dist}(T_u, T_v)$ is maximum among all pairs of nodes in the tree), then from the above Observation 5, the maximum distance from u is one of $\text{dist}(u, T_u), \text{dist}(u, T_v)$ and similarly for node v. If any one of the four pairs of distances are greater than K, then by observation 4 the answer for our query is `true`.

Revisiting the brute force algorithm, we see that we can replace the first for loop, with just the endpoints of the diameter and the inner for loop with the two special nodes u, v and then compute the maximum of all desired pairs of nodes in $O(1)$.

Now we can answer every query of type `3` in $O(1)$ time.

Queries 1 & 2

We need to find a way to compute the farthest special points, when the set of special points are updated by either addition or deletion of a node. We can do this by using a segment tree.

[details=Segment Tree Implementation Details]

Consider a segment tree where in each node $[L, R]$ we store the pair of special nodes which are farthest apart among all pairs of special nodes such that both of them lie in the range $[L, R]$.

Let's consider an internal node of the segment tree corresponding to an interval $[L, R]$ with left child l and right child r, and $M = \lfloor \frac{L+R}{2} \rfloor$.

Using Observation 5 with $S =$ the set of special nodes in the left child and $T =$ the set of special nodes in the right child we find that it is sufficient to look only at the farthest pairs of nodes among the nodes stored in the left and right children. There are at most 4 nodes and $\binom{4}{2} = 6$ pairs to consider. So merging elements in the segment tree can be done in $O(1)$ by computing the distances for each pair and choosing the one that is maximum.

Therefore we can use the usual segment tree with updates. This gives a total of $O(\log N)$ time to support each operation of type `1` and `2`.

Finally note that if a segment tree node does not contain any special nodes, we can have a pair of dummy invalid nodes which always give a distance of `-1` (or any invalid distance). For more details refer the source code below.

[\details]

COMPLEXITY:

TIME: $\text{O}((N + Q) \log N)$
SPACE: $\text{O}(N \log N)$

PREREQUISITES:

Data structures

PROBLEM:

Given an array of N stones each of some height.
For each query, we are given two positions and we have to output the total number of stones visible from both positions.

EXPLANATION:

While maintaining a stack we can easily determine the number of stones visible from each position in linear time.

Thereafter we can calculate the height and position of the maximum stone present between stones A and B.

It is not tough to see that the number of stones visible from this position is also the number of stones that are visible from both positions A and B.

As for the implementation details we can use a segment Tree to determine the maximum between A and B.
Also we maintain a map where for each height we store a pair of position and size of stack (which corresponds to the number of visible stones from this position).

Finally we can just use lower_bound to find out the exact position of the maximum.

PREREQUISITES:

BIT, segment tree.

PROBLEM:

Determine minimum amount of minutes required to paint a fence consisting of N parts, in green color. Each part is initially painted either red or green. We can chose an index X, and flip colors in range $[X, \min(X+K, N-1)]$ in one minute.

QUICK EXPLANATION:

Consider the index where first red is occurring as T, flip the colors in range $[T, \min(N, T+K-1)]$. Keep repeating this step until each part is colored green.

EXPLANATION:

First lets see how to solve the problem, later we can see the proof of correctness.

Consider the index at which first red is occurring and call it T, flip the colors in range $[T + \min(N, T+K-1)]$. Keep repeating the given procedure unless whole array is green. After each iteration value of T will be increasing, so the given algorithm will terminate in less than or equal to N steps.

Each iteration may take $O(N)$ amount of time, hence the given algorithm will be taking $O(N^2)$ time. As the N can be 10^5 in the given problem, we need to optimize our solution.

We can solve the problem in $O(N \log N)$ time. Range flip can be seen as adding 1 to the range and value of color can be obtain using sum at that particular index modulo 2. If the $\text{sum}\%2$ is 0, then color at that index will be same as the initial one or flipped otherwise. The range updates and queries can be done in $O(\log N)$ time using **BIT** or **segment tree** .

O(N) Solution:

It can be done in $O(N)$ as well. Adding 1 to range $[L, R]$ can be seen as adding 1 to index L and adding -1 to index $(R+1)$. When we want to retrieve the value at specific index, we need to take the prefix sum upto that index. See sub problem 1 of this **editorial** for more details.

Proof:

The given procedure can be proved correct using the following lemmas.

1. Order of the updates doesn't matter.
2. Starting index of any 2 updates will be different, as 2 updates at same starting position will cancel each other.
3. If we sort the minimum update sequence and suppose first update happens at index i, then at each index in range $[0, i-1]$ color must be green and at index i, color must be red.

PROBLEM:

You are given an array A of N non-negative integers.
Let $A(i,j)$ denote the subarray
$\{A[i], A[i+1], ..., A[j]\}$,
$1 \leqslant i \leqslant j \leqslant N$.

- The maxshift of any subarray $A(i,j)$ is denoted by $D(i,j)$ and is defined as:
  $D(i,j)$ = $\sum_{k=i}^{k=j}$ ($max - A[k]$), where max denotes the maximum element in $A(i, j)$.
- The minshift of any subarray $A(i,j)$ is denoted by $d(i,j)$ and is defined as: $\$
  $d(i,j)$ = $\sum_{k=i}^{k=j}$ ($A[k] - min$), where min denotes the minimum element in $A(i,j)$.
- We also define the strength of $A(i,j)$ as:
  $S(i,j) = D(i,j) + d(i,j) + \sum_{k=i}^{k=j} A[k]$

You need to answer Q queries of the following types:

$Type\ 1$: $1\ x\ y$ ($1 \leqslant x \leqslant N$)
Change the element at position x to y. **Note that this change will persist**.

$Type\ 2$: $2\ \ell\ r$ ($1 \leqslant \ell \leqslant r \leqslant N$)
Output $S(\ell, r)$.

Prerequisites:

- knowledge of segment tree and its applications.

QUICK EXPLANATION:

The expression for the strength of the array can be reduced to :
$(r - l + 1) * (max - min) + sum$
All these three parameters can be found using a segment tree.

EXPLANATION:

First of all, let's analyze the expression for strength. Let the array elements be $A[1], A[2], ...A[n]$. For any subarray $\{A[i], A[i+1], ..., A[j]\}$, the maxshift can be written as :

$$D(i,j) = max - A[i] + max - A[i+1] + max - A[i+2]..... + max - A[j]$$

$$D(i,j) = max * (j - i + 1) - \sum_{k=i}^{k=j} A[k]$$

Similarly, the expression of minshift can be written as :

$$d(i,j) = A[i] - min + A[i+1] - min + A[i+2] - min..... + A[j] - min$$

$$d(i,j) = \sum_{k=i}^{k=j} A[k] - min * (j - i + 1)$$

Therefore,

$$S(i,j) = max * (j - i + 1) - min * (j - i + 1) + \sum_{k=i}^{k=j} A[k].$$

$$S(i,j) = (max - min) * (j - i + 1) + \sum_{k=i}^{k=j} A[k].$$

So, for each thpe 2 query (which represents a subarray), we need three parameters - min, max and the sum of that subarray. We cannot iterate over the segment to get these values because the overall time complexity of the solution would be $O(Q * N)$, which will give TLE.

Instead, we can use segment trees to get these values and the overall time complexity will be reduced to $O(Q * logN)$ which will satisfy the given constraints.

PREREQUISITES:

Binary Indexed Tree (fenwick tree) / Segment Tree

HINT

Solve the subtask3 and try to reduce the original problem to this special case.

EXPLANATION

Let A be the given array (zero indexed).

Let sum(A,i…j) = A[i] + … + A[j]

Let avg(A,i…j) = sum(A,i…j) / (j-i+1)

Subtask1:
For each pair of indices i,j (i <=j) check if average of elements between i and j is greater than or equal to K.

```
count=0
for i = 0 to (N-1)
    for j = i to (N-1)
        if avg(i..j) >= K
            set count = count+1
output count as answer
```

This solution runs in O(N^3) and hence solves only the 1st subtask.

Subtask2:
You can optimize the previous solution to run in O(N^2) by making a very small observation.
To compute avg(i…j) you need to compute sum(i…j). You can compute sum(A, i…j) by just adding A[j] to sum(A, 1…(j-1)). Let us look at a refined algorithm using this.

```
count = 0
for i = 0 to (N-1)
    sum=0
    for j = i to (N-1)
        sum = sum+A[j]
        if sum >= K*(j-i+1)
            set count = count + 1
output count as answer
```

Subtask3,4 and 5:
Subtask3 has an additional condition of K = 0. i.e., we have to find the number of sub-arrays whose average is >= 0. It suffices to finding the number of sub-arrays whose sum >=0. We shall now reduce our problem to this special case.

Create a new array B of length same as A and set B[i] = A[i] - K for all i.

Claim: avg(A,i…j) >= K if and only if sum(B,i…j) >= 0

Proof:
First we prove avg(A,i…j) >= K implies sum(B,i…j) >= 0

(A[i] +… +A[j])/(j-i+1) >= K

A[i] +… +A[j]) >= K*(j-i+1) = K + … + K (j-i+1times)

(A[i] - K) + … + (A[j] - K) >= 0

B[i]+…+B[j] >= 0

We can prove sum(B,i…j) >= 0 implies avg(A,i…j) >= K by just going in the backward direction

> So the problem now boils down to
> finding the number of sub-arrays in B
> whose sum >= 0

Consider the PrefixSum array of B, let us call it P. (it will be of same length as B).

For each i we will have P[i] = B[0] + … + B[i]

We can compute this in O(n) time.

Note that sum(B,i…j) = P[j] - P[i-1] (imagine P[-1] to be 0)

Consider the following algorithm to compute our answer.

set count = 0

for i = 0 to (N-1)

- if P[i] >= 0 increment count by 1
- Find the number indices j such that P[j] <= P[i] and j < i, let us call the value x
- increment count by x

Output count

To find the number indices j such that P[j] <= P[i] and j < i, if we naively iterate through all indices < i our algorithm will end up being a O(n ^ 2) algorithm which is no better than our solution for subtask2. But it turns out that we can answer this particular query in O(log n) time by using an appropriate data structure.

Initially the datatstructure will not have any elements.

The idea is simple while processing a particular index i we will query the datastructure for the number of prefixSums in the data structure present already and less than or equal to current prefixSum. This has to done in O(log n) time.

We will add P[i] to the datastructure. This has to be done in O(log n) time.

Hence giving a runtime of O(n log n)

There are at least three data structures which will help you.

- Binary Indexed Tree - See setter's and tester's solution
- Segment Tree - See editorialist's solution
- Balanaced Binary Search Trees - We can use an augmented balanced BST similar to Order Statistic Tree (**Order statistic tree - Wikipedia**). But implementing a balanced BST during a contest is very tough job and hence we will skip this approach.

Refer **http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees#read** to learn about Binary Indexed Trees

Refer to **http://www.topcoder.com/tc?d1=tutorials&d2=lowestCommonAncestor&module=Static** and **link text** to learn about segment trees. But these might be overkill to solve this problem.

Since we have values between -10^9 and +10^9 we can't use a BIT or a segment tree, hence we work with the positions of the values in the original array to index our data structures. Look at the solutions to see how to overcome this.

Also check out this question on quora about BIT vs Segment Tree **http://www.quora.com/Data-Structures/How-does-one-decide-when-to-use-a-Segment-Tree-or-Fenwick-Tree**

SOLUTIONS

Setter's Solution: **SUBARR.cpp**

Tester's Solution: **SUBARR.cpp**

Editorialist's Solution: **SUBARR.java**

Setter's and Tester's Solution Explanation

You can't use a BIT directly to compute the answer as we can't use our prefix sums to index the BIT data structure as they can be arbitrary integers and we can't have an array of such huge size. Here is how the setter's and tester's solution overcome this.

Create n pairs <sum, index> and now sort them on increasing value of sum, if sum is equal then use index. Create a BIT data structure of size n (we store data based on indices). Process the pairs in sorted order when we are processing a pair P all the pairs processed will have sum < P.sum or (= P.sum and index < P.index) . So we have sufficient information to perform the step "Find the number indices j such that P[j] <= P[i] and j < i, let us call the value x" in our algorithm proposed earlier. The value of x is nothing but the count of all indices < P.index stored in the BIT which can be simply found in O(log n) time by querying the BIT. Now update the BIT with the value P.index. Note that we don't process the prefix sums in the order we proposed in our algorithm but since we process all the prefix sums we still get the same answer.

Editorialist Solution Explanation

The structure used is a standard segment tree. Since the values of our prefix sums can be arbitrary and can't directly use the segment tree to represent these value what we have done instead is sort the array and index the elements from 0 to (N-1) (could be less if the array had duplicates) and use these indices to refer to the array values and vice versa. At the last level of the complete binary tree representing segment tree the leaf nodes represent the indices 0 to N-1 and rest of the leaves can be simply ignored. Each leaf stores the count of number of times the value corresponding to that leaf has been inserted in the segment tree so far. Each internal node stores the sum of counts stored in all the leaf nodes which fall under the left subtree of that node. Now let us see how we can perform insert and the query operations.

When we want to increase the count of a value P into the segment tree, we get the corresponding index i of P in the sorted array. Then we locate the ith node from the left in the last level of the segment tree and increase the value of that node. Now we traverse up to the root by moving one level up at a time and since there can be only parent in the tree this path will be unique. When we are travelling up this path all we need to see if we have reached the current path from the left or the right. If we came from left that indicates the node we processed initially was in the left sub tree of the current node so we simply increment its count by 1. If we came from the right side we simply do nothing as the node we processed lies in the right sub-tree.

When we want to query for number of elements less than equal to P in the segment tree, we get the corresponding index i of P in the sorted array. We have a variable ans initialized to 0. Then we locate the ith node from the left in the last level of the segment tree and add the value of that node to ans. Then we traverse up to the root just like we did in the insert step. But this time we do just the opposite, if we are coming from the left we do nothing and if we are coming from the right we simply add up the value to ans and do nothing when we come from the left. Now ans we have count of all the elements less than equal to P. If you are convinced why this works just draw up a tree take few examples and see.

PREREQUISITES:

Partial Sum Arrays, Binary Search or Segment Tree (or sliding window with heaps).

PROBLEM:

Given an array A of length N and an integer K, find the number of beautiful subarrays where a subarray $S_{l,r} = A_l, A_{l+1} \dots A_{r-1}, A_r$ is beautiful if following holds.

- Define array B as $S_{l,r}$ concatenated concatenated with itself m times, where m is the smallest integer such that $m * (r - l + 1) \geq K$ and sort B.
- Let $X = B_K$, the Kth smallest element in B. Let F be frequency of X in $S_{l,r}$.
- A subarray is beautiful if F is present in $S_{l,r}$.

QUICK EXPLANATION

- We need to check each subarray separately. For each subarray, we can see that $m = \lceil K/(r - l + 1) \rceil$.
- Now, When we append $S_{l,r}$ m times and sort it, the array looks like, a m occurrence of the smallest element in $S_{l,r}$, m occurrences of the second smallest element in $S_{l,r}$ and so on. So, we know, that element at a k position in B is actually the element at $\lceil k/m \rceil$ position when $S_{l,r}$ is sorted.
- Finding xth element in $S_{l,r}$ can be done using binary search over prefix arrays or using segment trees.
- Checking frequency of X is doable with partial sums. Then we can check if F is present in the array or not again using partial sums or segment tree.

EXPLANATION

So many prerequisites for the first problem? The editorialist must be having fun with us Yes, He's having fun :D.

Testers solution is using Segment trees while editorialist solution uses prefix arrays and binary search which you may refer below.

Testers solution

We check for each subarray whether it is beautiful or not and increment answer if subarray is beautiful.

We need to find B_k where B is the array obtained by appending $S_{l,r}$ to B till $|B| < K$ and sorting array B. We can observe, since we append the same array m times, there shall be m occurrences of an element on B for every occurance of any element in $S_{l,r}$. Also, We know, that the smallest m such that $m * (r - l + 1) \geq K$ is $m = \lceil K/(r - l + 1) \rceil$. So, when we sort B, First m elements of B are same as smallest element of $S_{l,r}$, next m elements are second smallest element of $S_{l,r}$ and so on. We need to find xth smallest value in $S_{l,r}$ such that $(x - 1) * m < K$ and $x * m \geq K$.

Suppose we make a segment tree ith element denoting the frequency of i in $S_{l,r}$ where we can update the frequency of any element and query the number of elements having value in the range (l, r). If we want to find the xth element in range (l, r) represented by a node, we can move down the node by checking if left child node has, say z elements in its range, If $z >= x$, we know that our required element lies in range represented by left child. Otherwise, we move to the right child and try to find the (x-z)th element in the range given by the right child. More about such a segment tree in the box.

Click to view

Each leaf node representing range $[x, x]$ store the frequency of x in the array. The non-leaf nodes representing range $[l, r]$ contains the number of values which lie between $[l, r]$. While querying for the xth smallest element at a node representing range $[l, r]$, if the left child has at least x values, we know xth smallest lie in left sub-

range. But if the left child has less than x values, say z values, we know that after counting z smaller elements, the xth element in the range $[l, r]$ is same as $(x - z)$ element in the range given by the right child.

Moving forward, we have found $X = B_K$, now we check the freqency of X, say F in $S_{l,r}$ using segment tree. Now, We check the frequency of F in subarray again with the same frequency segment tree and if F has a frequency at least one, we increase the number of beautiful subarrays.

Click to view

About Making such frequency segment tree, We fix a starting point say l, and consider subarrays $S_{l,l}$ first, then $S_{l,l+1}$ and so on. We can see that we just need to increase the frequency of A_r in frequency tree having all elements of subarray $S_{l,r-1}$. Hence, we can fix every possible start point and thus solving the problem.

Editorialist solution

Editorialist proceeds the same way as the tester, except for the part of finding out B_K. We make prefix arrays PRE[x][r] denoting the number of values in range $[1, r]$ which have value $\leq x$.

Now, We can binary search on the number of elements smaller than the current element to find out the xth element such that $(x - 1) * m < K$ and $x * m \geq K$. For each iteration, we can compute in $O(1)$ time the number of elements in the range $[l, r]$ smaller or equal to the current mid element, hence finding the B_K in $O(log(MAX))$ time.

Now, checking the frequency of an element x in range can also be done from same prefix arrays.

Click to view

Frequency of x in range $[l, r]$ is $(PRE[x][r] - PRE[x][l]) - (PRE[x - 1][r] - PRE[x - 1][l])$.

Another solution

Another possible solution is to iterate over the length of subarrays. For a given length, m does not change, and thus, $P = \lceil K/m \rceil$ remain same. We can maintain two heaps, first being a max heap holding smallest p elements and other heap holding remaining elements. Whenever we move to the next starting position, we remove one element (at the previous starting point) and add another element (the rightmost element which got included), updating the heaps accordingly and finding the B_K easily. Remaining is left as an exercise.

People say I write long and scary editorials. I believe they are worth it! What do you say?

Time Complexity

Time complexity is $O(N^2 * log(N))$ per test case. (Can you find a faster solution?)

PREREQUISITES

Euler Tour, Segment Tree, Greedy

PROBLEM

Given a tree with N nodes and an integer K.A subtree is defined as a connected subgraph of the tree. That is, a subtree is another tree that can be obtained by removing some (possibly none) vertices and all edges incident to those vertices from T.

A subset S of vertices is called good if every subtree containing all the nodes in S has at least k nodes.

Find the **size of the smallest good subset**.

QUICK EXPLANATION

- For $K = 1$, we have to select only 1 node.
- If the diameter of the tree has X nodes, then we can select its endpoints as the required subset for all $K \leq X$
- Otherwise, after selecting endpoints of diameter in the subset, we select leafs furthest from any selected vertex greedily, till the number of nodes in induced subtree cover is at least K.

EXPLANATION

Intuition and Idea

Let's handle the edge case with $K = 1$ beforehand, as we can select any 1 node.

Now, let's see what is the largest number of nodes we can cover using a good subset of two nodes. The number of nodes in the subtree is nothing, but the number of nodes on a simple path between two nodes. By the definition, the diameter of the tree is the longest simple path present in the graph. So if the diameter of the tree is X nodes, then the subset containing endpoints of diameter is good for all $K \leq X$

Now, let's say $K > X$, so we need to select at least 3 nodes.

Claim: We only select leaf nodes in a good subset
Proof: Let's assume there's a non-leaf node u inside the chosen subset. Two cases arise

- Node u has a direct neighbour not present in the subtree.
  In this case, we can select that neighbour, increasing the size of the subtree covered by at least 1. hence, the choice of node u is not optimal in this case
- Node u has no direct neighbour not covered in the subtree
  In this case, Since node u has at least 2 neighbours, both of which are already covered, then node u is covered in subtree irrespective of including u in the subset. So we can remove node u from the subset, reducing subset size by 1 while keeping subtree size the same.

Claim: Let's say an endpoint of diameter (a, b) is selected as the root, It is optimal to choose the root node and leaves greedily, based on the number of nodes, which shall be included in subtree by choosing this leaf.

Intuition:
I'd share my intuition on why this claim works, as complete proof is a bit tricky

- Firstly, choosing one leaf over other do not, in any way restrict us from choosing some other leaf subsequently.

- Let's say we choose leaves a, b, c, d in this order by greedy, where a and b are diameter. See the tree image below for reference. After this, the gain by adding node c and node d shall be $G_c + G_d - G_{lca(c,d}$. If we change the order to a, b, d, c, the substree spanned after adding 4 nodes is same, but subtree spanned after three nodes is $D + G_d$ instead of $D + G_c$ where D is length of diameter. Since greedy choose c before d, then $G_c \geq G_d$, so if some leaf shall be added to subset, it is optimal to add them in the decreasing order of G_u.

Lastly, Let's compare two orders a, b, u, w and a, b, v, w, where (a, b) is diameter, and first order is made by greedy approach, and second optimal. After processing diameter, we have $G_u \geq G_w \geq G_v$, and we have $G_w - G_{lca(u,w} \geq G_v - G_{lca(u,v)} \implies G_w \geq G_v$.

Note that the values G_x used are the values just after processing diameter.

The number of nodes added in first case is $S_1 = D + G_u + G_w - G_{lca(u,w)}$ and in second case, it is $S + 2 = D + G_v + G_w - G_{lca(v,w)}$. We aim to prove that $S_1 \geq S_2$ or $S_1 - S_2 \geq 0$.

$S_1 - S_2 = D + G_u + G_w - G_{lca(u,w)} - (D + G_v + G_w - G_{lca(v,w)}) \implies S_1 - S_2 = G_u - G_{lca(w,u)} - G_v + G_{lca(v,w)}$

We already have $G_u \geq G_w$, and We have $$G_{lca(u, w)} \geq G_{lca(u, v)}$ implying node w is in subtree of node $lca(u, v)$ which implies $G_{v,w} \geq G_{u,v}$.

Now, by considering nodes in order a, b, w, and then choosing greedily, we can see that since only one node can be choosen, we must choose the node with largest $G_u - G_{u,w}$, which is leads to same gain as picking nodes in order a, b, u, w. So this completes a rough intuition on why greedy works. The complete proof can be read in paper below.

Proof: The rigorous proof for this can be seen in **this** paper, with special emphasize on section 2.4 on Tree networks.

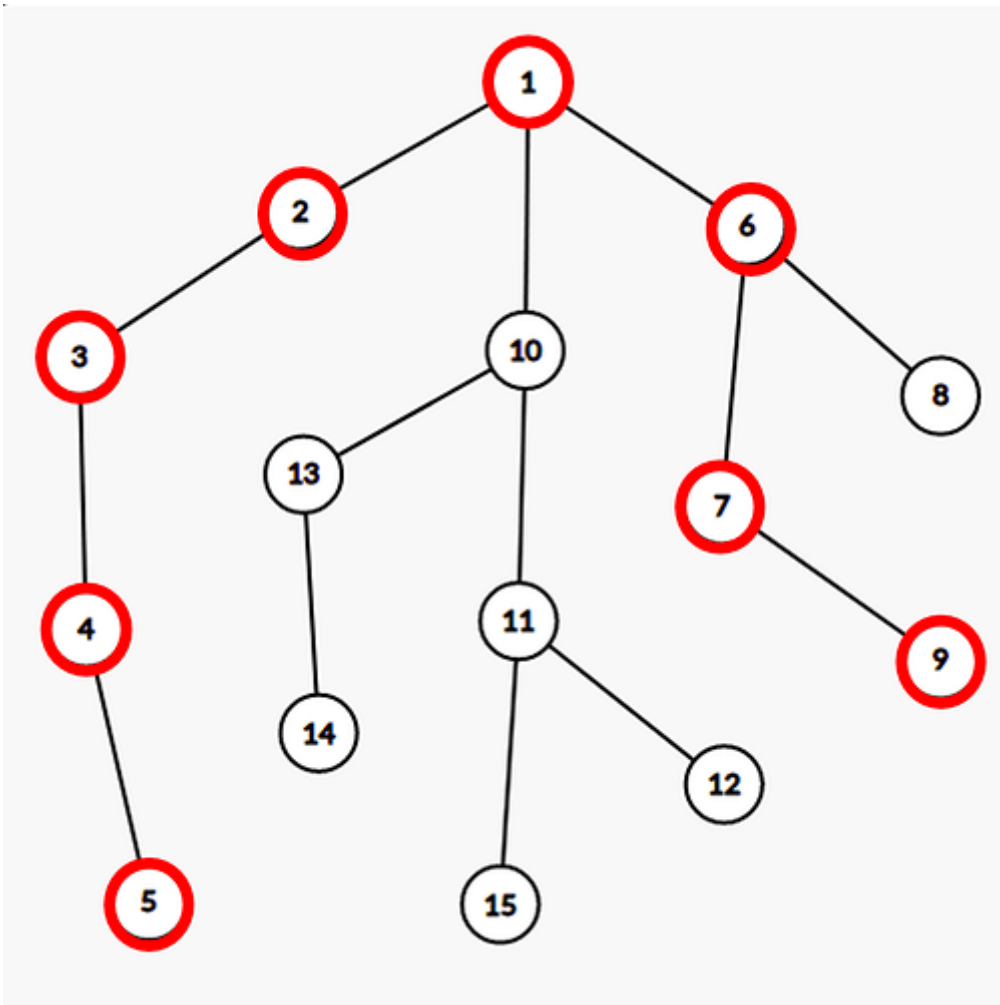
Implementation

After finding diameter, let's root the tree at one end of diameter. The benefit of this is that since root of this tree is included in subset, then if we choose to include node u in subset, all ancestors of node u shall be included in subtree automatically, saving tedius implementation.

Let S denote the good subset selected till now, inc_u holds a boolean value, determining whether node u is inside subtree or not, and G_u denotes the number of new nodes, that shall be added to subtree if node u is selected.

We need a way to maintain G_u for all nodes while moving some nodes from not included to included status.

Let's see the example below.



The path highlighted is the diameter of tree, and is already included in induced subtree by nodes 5 and node 9.

Let's compute G_u for all non-included nodes. We have $G_{10} = G_8 = 1$, $G_{11} = G_{13} = 2$ and $G_{12} = G_{14} = G_{15} = 3$

Let's say node 12 is added to subset. Now, we need to update G for all nodes not in induced subtree.

Nodes 10, 11 and 12 shall get added to induced subtree. So the resulting values would be $G_8 = G_{15} = G_{13} = 1$ and $G_{14} = 2$ We assume $G_u = 0$ if u is included in subtree.

Claim: If node u is added to subset, then G_v shall be reduced by $G_{lca(u,v)}$ for all nodes v.
Proof: All nodes on path from $lca(u, v)$ to nearest included nodes are the ones which get excluded from G_v by reducing G_v.

Hence, let's consider nodes 10, 11 and 12 in this order as candidates for $lca(12, v)$.

- For node 10, Both nodes 13 and node 14 has lca with 12 at node 10. So we reduce G_{13} and G_{14} by $G_{10} = 1$ each.
- For node 11, Only node 15 has lca with node 12 at node 11. Hence, G_{15} is reduced by G_{11}
- There's no node with $lca(v, 12) = 12$

Hence, while considering node x, we iterate over all children of x not included in subtree, and reduce them by G_x.

If we build an euler tour on tree, then these transform to range decrement queries. Finding u with maximum value of G_u is just the argmax query, the position of maximum value. For removing some value, we can decrement them with N, so that they are never considered again.

Lastly, since each node shall be move from non-included to included only once, we can process nodes (like we processed 10, 11 and 12) one by one.

TIME COMPLEXITY

Diameter can be computed in $O(N)$. Segment Tree operations take $O(log(N))$ per query after $O(N)$ construction. Each node is processed only once, so the number of processed nodes is $O(N)$ and the number of segment tree operations is proportional to the number of edges, which is $O(N)$ as well.

Hence, the time complexity is $O(N * log(N))$ per test case.

PREREQUISITES:

Segment trees with lazy propagation

PROBLEM:

You have an array A and an integer K.
Let $f(l, r)$ denote the number of distinct integers in the subarray $[A_l, A_{l+1}, \ldots, A_r]$.

Find

$$\sum_{l=1}^{N} \sum_{r=l}^{N} f^k(l, r)$$

EXPLANATION:

First, let's try to solve this problem for $K = 1$, and then generalize that solution to higher K.

Solving $K = 1$

Our task is to compute the sum of the number of distinct elements across all subarrays.
While counting the number of distinct elements in a range is easy in $\mathrm{O}(N)$, and even answering Q queries about different ranges is doable in $\mathrm{O}((N + Q) \log N)$ (see **here** for several ways to do it, both online and offline), querying for every subarray is still too slow.

However, we can slightly modify one of the ways to answer queries offline to achieve what we want.

Let's fix r, and then let D be an array of size N such that $D_l = f(l, r)$.
If we know D for r, we can quickly update it for $r + 1$ using the following observations:

- If A_{r+1} occurs in the subarray $A[l, r]$, then $f(l, r + 1) = f(l, r)$; i.e, there's no need to change D_l.
- Otherwise, $f(l, r + 1) = f(l, r) + 1$; i.e, D_l increases by exactly 1.
- Further, notice that if D_l remains unchanged, so does D_{l-1}; and if D_l increases by 1, so does D_{l+1}.

Essentially, this induces a range update on D: a certain range of values is increased by 1, and everything else remains unchanged.
Finding which range increases isn't too hard either: we just need to find the previous occurrence of A_{r+1}.

In particular, we have the following algorithm:

- Let D be as described above, initially containing all zeros.
- We'll consider r from 1 to N.
- At r, let p be the position of the previous occurrence of A_r.
- Then, add 1 to the range $[p + 1, r]$ of D.
- Finally, add $D_1 + D_2 + \ldots + D_r$ to the answer.

To do this quickly, we need a data structure that allows us to quickly add a value to a range, and find the sum of values in a range.
This can be done using a segment tree with lazy propagation.

We do N updates and N queries, so $K = 1$ is solved in $\mathrm{O}(N \log N)$.

Solving $K > 1$

Now, we need to generalize this solution to higher K.

Let's do almost exactly the same thing; except that now, $D_l = f^K(l, r)$.

Our updates and queries are also exactly the same; the only difference is in the part where we propagate updates down the segment tree.
In particular, we need to update the sum of K-th powers of several values, if all those values are increased by x.

Suppose we know $\sum_i a_i^K$, and we want to compute $\sum_i (a_i + x)^K$. That can be done as follows:

$$\begin{aligned}\sum_i (a_i + x)^K &= \sum_i \sum_{j=0}^{K} \binom{K}{j} a_i^j x^{K-j} \\ &= \sum_{j=0}^{K} \binom{K}{j} x^{K-j} \sum_i a_i^j\end{aligned}$$

Notice that the second summation is simply the sum of j-th powers of all the a_i, and is information that doesn't depend on x.
If we maintained that information, this update can be done in $\mathrm{O}(K)$.

So, in each node of the segtree we keep the sum of the j-th powers of the values in it, for each $0 \le j \le K$.
When pushing an update, simply update all of these $K + 1$ sums as mentioned above: this takes $\mathrm{O}(K^2)$ time — $\mathrm{O}(K)$ for each.

This gives us a solution in $\mathrm{O}(K^2 N \log N)$.

TIME COMPLEXITY:

$\mathrm{O}(K^2 N \log N)$ per testcase.

PREREQUISITES:

Segment Tree, Hall's Theorem

PROBLEM

You're given a row of M seats. N people want to buy tickets in this row. Person i wants exactly S_i number of tickets, all of which should be in the range $[A_i, B_i]$. It's guaranteed that $A_i \geq A_{i-1}$ for $2 \leq i \leq N$.

For each i ($1 \leq i \leq N$), calculate the largest j ($i \leq j \leq N$) such that all people $i \ldots j$ can be accommodated together and given tickets according to their requirements.

QUICK EXPLANATION (HINTS):

Hint 1

We can build a bipartite graph in which the nodes on the left side denote people, and the other side represents seats. Notice that the problem is now a specific case of the bipartite matching problem. We only need to determine if a left-saturating matching exists and not rebuild the entire assignment. Does this remind you of Hall's theorem?

Hint 2

What makes this setup of matching different from the general bipartite matching problem? How can Hall's theorem be modified to handle this case more efficiently?

Hint 3

Notice that all nodes on the left have an edge to a continuous range of nodes on the right.

Hint 4

Try to find a sufficient condition for checking if we can accommodate a continuous range of people together.

Hint 5

For an interval of seats $[l, r]$, let's denote by $F(l, r)$ the number of people p such that $A[p] \geq l$ and $B[p] \leq r$. If there exists an interval of seats $[l, r]$ such that $F(l, r) > r - l + 1$, then there exists no matching which satisfies all people. How do we implement this?

Hint 6

Time to use the condition that $A_i \leq A_{i+1}$.

Hint 7

Use a segment tree to process the j pointer while scanning over i pointers in order.

EXPLANATION

Assume that every person wants precisely one ticket. If they want more, let's create a duplicate person with the same $A[i]$ and $B[i]$ values for the number of tickets that they want. A greedy strategy to optimally assign seats to a set of people is:

- Sort the people by their $A[i]$ (which is already done actually in the input)
- For each seat s from 1 to M in order (preferably after doing some compression), assign this seat to the person p who still doesn't have a seat ticket and has $A[p] \leq i$ and $B[p] \geq i$. If there are multiple such people, choose the one with the smallest value of $B[p]$.

The intended solution takes an entirely different approach.

Note that this is a bipartite matching problem, with the additional constraint that the elements on the left side (people) always have edges to a continuous range of nodes on the right side (seats). Hall's theorem can take a stricter form in this case. For an interval of seats $[l, r]$, let's denote by $F(l, r)$ the number of people p such that $A[p] \geq l$ and $B[p] \leq r$. If there exists an interval of seats $[l, r]$ such that $F(l, r) > r - l + 1$, then there exists no left-side-saturating matching (a matching in which all people get a seat and each set gets assigned to at most one person). Let's call this $[l, r]$ a *violating range* of seats. If there is no such range, a left-side-saturating matching definitely exists.

How do we use this efficiently?

I'll first describe a way to check whether a given set of people can be accommodated together. We just need to check if a *violating range* of seats exists. Let's say it is $[l, r]$. If such a range exists, we can see that we can also find a *violating range* in which both l and r appear somewhere in the arrays A or B. Otherwise, we can increase l until that is true and decrease r until that is true, and it will still be a *violating range* since $F(l, r)$ remains the same and $r - l + 1$ will decrease. So, let's compress the ranges of seats.

Make a range-add range-max segment tree over the compressed seats, with values initially filled as $-X$, where X is the compressed seat's actual (uncompressed) index. We loop over the people in decreasing order of $A[i]$, and then add $+S[i]$ to all positions with actual seat indices >= $B[i]$ in this segment tree. Then, we query for the maximum in the segment tree. Let it be mx. If $mx > 1 - A[i]$, we've confirmed that there exists a *violating range* of seats. This is because $mx = k - r$, where k is the number of people whose range of compressed seats lies completely inside $[A[i], r]$. Since $mx > 1 - A[i]$, this implies $k > r - A[i] + 1$. Congrats, we've found a *violating range* $[A[i], r]$! If we didn't, then congrats! We can assign these seats to satisfy all people.

How do we use this to solve the problem at hand? For each i, let $ans[i]$ denote the answer we are looking for. To calculate $ans[i]$, we need to find the smallest suffix of people that need to be deleted such that the remaining suffix of people starting at i can be accommodated together. Let's first calculate the smallest suffix that needs to be deleted such that there exists no r such that $[A[i], r]$ is a *violating range.* Let this minimum suffix to be deleted be the one starting at position $j + 1$. Then, $ans[i] = min(ans[i + 1], j)$. That means we can calculate the j for positions N to 1 in decreasing order and keep moving the j pointer backwards as needed. This is true because if the j required for position i is greater than the one required for $i + 1$, it doesn't matter. Because if we include those greater positions, then there will exist a *violating range* starting at position $A[i + 1]$. So we can answer using two pointers.

Time Complexity: $O(N * \log N)$

PREREQUISITES:

Segment trees/Fenwick trees

PROBLEM:

Alice, Bob, and Charles participated in N contests; and obtained A_i, B_i, C_i points in the i-th one, respectively.

A person gets *first place* in a range $[L, R]$ of these contests if their total score in this range is not less than the score of the other two.
For each person, find the longest range of contests in which they obtained first place.

EXPLANATION:

Let's solve the problem for Alice; the same process can be repeated for Bob and Charlie.

Consider some range $[L, R]$ of contests.
Alice's score is not worse than Bob's in this range if and only if $A_L + A_{L+1} + \ldots + A_R \geq B_L + B_{L+1} + \ldots + B_R$.
This can be rewritten as $(A_L - B_L) + (A_{L+1} - B_{L+1}) + \ldots + (A_R - B_R) \geq 0$.

So, if we create an array X with $X_i = A_i - B_i$, we want $X_L + X_{L+1} + \ldots + X_R \geq 0$.
Subarray sums naturally lend themselves to prefix sums, so let's look at those.
If pX denotes the prefix sum array of X, then we want $pX_R - pX_{L-1} \geq 0$, i.e, $pX_R \geq pX_{L-1}$.

Similarly, we can construct arrays Y and pY corresponding to $A_i - C_i$.

Our task is to thus find the longest range $[L, R]$ such that $pX_R \geq pX_{L-1}$ **and** $pY_R \geq pY_{L-1}$.

In particular, if we fix a value of R, we then want to find the smallest $L \leq R$ such that these conditions hold.
There are ways to do this directly using persistent of 2D data structures, but there's also a simpler solution using more well-known (and easy-to-code!) structures.

Let's iterate over values of R in increasing order of pX_R.
This ensures that the condition $pX_R \geq pX_{L-1}$ is automatically satisfied, so we only need to deal with $L \leq R$ and $pY_R \geq pY_{L-1}$.

We now want to find the smallest value of $L \leq R$ such that $pY_R \geq pY_{L-1}$.
That can be done using a segment tree and binary search, as follows:

- Consider a range-minimum segment tree built on the array pY.
- With the help of this tree, we can use binary search to find the first L in the range $[1, R]$ whose pY_{L-1} value is no larger than pY_R.
  - This can be done by querying for the minimum value in the suffix $[1, \text{mid}]$ of the binary search and checking whether it's below pY_R or not.
- Since we're iterating over R in increasing order of pX_R, we can simulate this in the segment tree by first initializing everything to ∞ (some large number) so that querying positions with larger pX-value doesn't affect the minimum.
- After fixing R and finding the optimal L, update position R into the segment tree so that it can contribute to future queries.

At each index, we do a binary search which has segment tree queries within it.
This leads to a complexity of $\mathrm{O}(N \log^2 N)$.

This can be further optimized to $\mathrm{O}(N \log N)$ by combining the binary search into the segment tree query; in the technique sometimes known as **segment tree descent or walk**.
This is not necessary to get AC, however.

PREREQUISITES:

Segment Trees.

PROBLEM:

The problem basically is :
Store a dynamic list of pairs. You have to answer queries which give you a pair to store in the list and asking you to retrieve a pair having the minimum 2nd value among all the pairs having 2 times the first value of the pair given to store currently.
In case 2 pairs have the minimum 2nd value, return the one having max 1st value then.

QUICK EXPLANATION:

Make a segment tree over a 1-D table of multisets. Make range queries over the table and retrieve the smallest values stored in that range of multisets.

EXPLANATION:

This is a kind of medium-level implementation question requiring you to use a segment tree(also possible in fenwick tree) to get the desired value.

Make a N sized array of multisets where ith element will represent the multiset storing the price of the phones having performance equal to i.
Building a segment tree over this array will allow us to make queries over a range [L,R] to get the cheapest phone having performance in the range [L,R]. The segment tree should be made of pairs to store the location of the minimum value too incase 2 phones of different performance have the same cheapest price.

Total time complexity is in the order $O(NlogN + Mlog(N + M))$

PREREQUISITES:

DFS Tree, Segment Tree with Lazy Propagation

PROBLEM:

You are given two rooted trees T_1 and T_2 consisting of N nodes each, numbered from 1 to N. Any vertex on the path from the root to a vertex v is an ancestor of v.

A vertex numbered w is said to be a common ancestor of a pair (u_1, v_2), if:

- w_1 is an ancestor of u_1 and w_2 is an ancestor of v_2.
- $depth_{w_1}$ = $depth_{w_2}$, that is, w_1 and w_2 are at equal depths from their respective roots.

Your task is to find the maximum number of ancestors common to a pair of vertices among all such pairs from T_1 and T_2.

QUICK EXPLANATION:

- Linearize the first tree $i.e.$ T_1 and build a segment tree with nodes in DFS order.
- Now, run the DFS for the second tree $i.e.$ T_2.
- If there is a node that matches in depth with its counterpart in the tree T_1, then all nodes in its subtree will count this node as a common ancestor.
  - Hence we increment each node of this subtree as the number of common ancestors for all the nodes of this subtree will increase.
  - At each stage, we find the node which has maximum number of common ancestors.
  - Finally, while leaving the node we undo the increment.
- The maximum number of common ancestor is equal to the the maximum number of common ancestor that any node has at any stage.

EXPLANATION:

For any vertex to be common ancestor, a should be present at the same depth in both of the trees $i.e.$ T_1 and T_2, and there should exists a pair (u_1, v_2) such that vertex is ancestor to both of the nodes in their respective trees.

We will try to explore tree T_2, if we find any node x that matches in depth with its counterpart in tree T_1, then all nodes of its subtree will count this node as an ancestor. It means there can exists a pair as:

$$(S_{1x}, S_{2x})$$

where S_{1x} denotes the set of vertex, such that each node of this set is subtree of x for tree T_1 and similarly define S_{2x} for tree T_2.

Also, we can observe that the vertices that were common ancestor before arriving at node x, will not be affected due to tree T_2.

Proof

Let's say $C_1, C_2, ...C_M$, be the common ancestors at node y.It means $C_1, C_2, ...C_M$ are the ancestors of node y.

While doing DFS on tree T_2, we can arrive to any node only and only through its parent, it means that all the ancestors of the parent node will also be the ancestors of its child node. So, if we arrived at node x through node y, it means that $C_1, C_2, ...C_M$ are also the ancestors for the node x.

Hence, every time we can choose x from S_{2x} to form a pair, as choosing x results in maximizing the number of common ancestors. Since all the common ancestor are still valid and choosing x gives us one additional vertex as common ancestor $i.e.$ node x itself.

$$(S_{1x}, x)$$

Hence, we can say that our answer completely depends on S_{1x}. $i.e.$ on subtree of node x of tree T_1. It means that the number of common ancestors will be affected due to tree T_1.

Why

Let's say $C_1, C_2, ...C_M$, be the common ancestors at node y.It means $C_1, C_2, ...C_M$ are the ancestors of node y.

Since, we are not doing DFS on tree T_2, it means when we arrive at node x, it doesn't guarantee us that we arrived at this node through its parent. It means that y can be or can't be the ancestor of node y. Hence, we say that $C_1, C_2, ...C_M$ can be or can't be the common ancestor for node x.

In order, to do so we will check each time for all nodes of S_{1x}, the number of common ancestor it has and finally find out the maximum number of ancestor any node has in this set.

We can use segment tree for the same, so whenever we encounter a node x that matches in depth with its counterpart in the tree T_1, then we increment all nodes of the subtree x in our segment tree as all the nodes will have one more common ancestor $i.e\ x$. During, each update we check for the maximum number of ancestors of any node of the subtree x. Finally when leaving the node, we undo the increment for the all the nodes of subtree x.

If we linearize our Tree T_1 and build the segment tree in the DFS order, then the subtree of any node x will always be continuous. Hence, we can say that we are incrementimg/decrementing in a range.

Hence, we can use Segment Tree with Lazy propagation, as explained **here** and **here** and at many places I didn't link here.

Rest is implementation, which you may refer from solution.

TIME COMPLEXITY:

$O(N * log(N))$ per testcase

PREREQUISITES:

Euler tour of a tree, Segment trees

PROBLEM:

You're given a tree on N vertices. Vertex i has color C_i.
Let F_x denote the frequency of color x in the **entire tree**, and S_u denote the set of all colors appearing in the subtree of u.

For each vertex u, compute

$$\prod_{x \in S_1 \setminus S_u} F_x$$

modulo M.

EXPLANATION:

If M were prime, this would be a rather easy exercise in small-to-large merging.
Unfortunately, M isn't guaranteed to be prime — this throws any approaches involving division out of the window.

Let's perform an Euler tour of the tree, so that subtrees now correspond to subarrays.
Let $[L_u, R_u]$ be the subarray corresponding to the subtree of vertex u.
This makes the set of vertices *outside* the subtree correspond to the union of the ranges $[1, L_u - 1]$ and $[R_u + 1, N]$.

In particular, suppose that we are able to ensure the following:

- For any color c that appears in S_u, F_c doesn't appear outside $[L_u, R_u]$
- For any color c that doesn't appear in S_u, F_c appears exactly once outside $[L_u, R_u]$

Then we can simply take the product of the values on ranges $[1, L_u - 1]$ and $[R_u + 1, N]$ to obtain our answer!
Finding the product of a range can be done in $\mathrm{O}(\log N)$ using a segment tree, so we just need to figure out how to maintain the two invariants above.

That can be achieved with a bit of cleverness when performing a DFS.
Consider an array A of length N, such that $A_i = 1$ for all i initially.
Next, for each color x, select **exactly one** vertex u such that $C_u = x$ and set $A_{L_u} = F_x$.

This ensures that the frequency of each color appears exactly once throughout the whole array.
We only need to focus on moving it around appropriately.

Let's perform a DFS. When at vertex u with color x,

- First, recursively compute the answer for all the children of u.
- Then, suppose pos_x is the current position of F_x in the array.
  Set $A_{\mathrm{pos}_x} = 1$, and $A_{L_u} = F_x$.
  Finally, query for the product of ranges $[1, L_u - 1]$ and $[R_u + 1, N]$; the product of these two values is the answer for u.

This works because:

- First, we already ensured that each F_x value occurs exactly once in the tree.

- Next, if color x doesn't appear in the subtree of u, then F_x will surely lie outside the range $[L_u, R_u]$ and hence be included in the product.
- Finally, if color x *does* appear in the subtree of u, F_x will be set at the position of some occurrence of color x in the subtree of u — in particular, it will be inside the range $[L_u, R_u]$, and hence will not be included in the product of the outside.

Point updates and range queries can be handled in $\mathrm{O}(\log N)$ using a segment tree, and so we're done.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PREREQUISITES:

Segment Tree, basic geometry, and sweep.

PROBLEM:

Given N points (x_i, y_i) in two-dimensional cartesian space, and N right isoceles triangles having $(l_i, 0)$ to $(r_i, 0)$ as their hypotenuse, find the number of points lying on or inside each triangle.

QUICK EXPLANATION

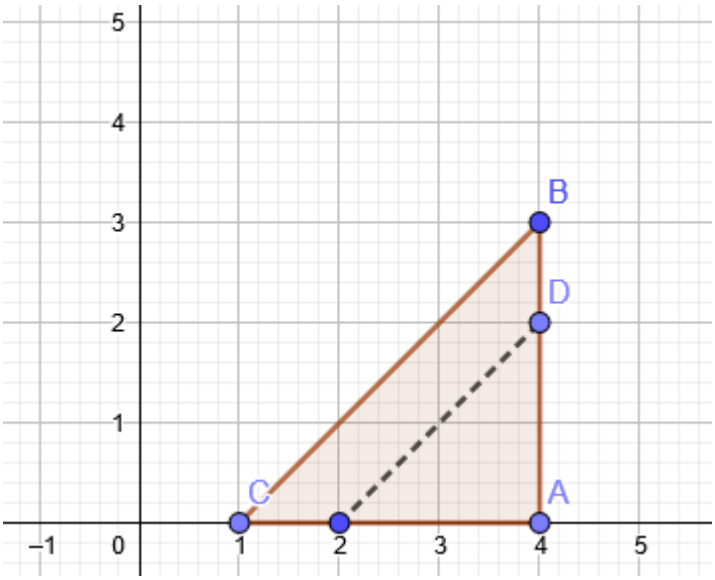
- We can divide the triangle (l_i, r_i) into two right triangles, each of which has it's base at x-axis. We shall consider the left and right half of the triangle separately.
- For point (x_i, y_i) and left triangle $(a, 0)$, $((l_i + r_i)/2, 0)$ and $((l_i + r_i)/2, (r_i - l_i)/2)$, the point lie inside the triangle if and only if $x_i \leq (l_i + r_i)/2$ and $x_i - y_i \geq l_i$.
- We can sort the points by x_i and left half of triangles by $(l_i + r_i)/2$, all the point with $x_i \leq (l_i + r_i)/2$ shall appear before this triangle. We just need to add $x_i - y_i$ into a DS and for the triangle, we need to query the number of values in DS greater than or equal to l_i
- Similarly, for right triangle, we get conditions $x_i \geq (l_i + r_i)/2$ and $x_i + y_i \leq r_i$. We can use DS to add points $x_i + y_i$ into DS and querying for number of points with value $[(l_i + r_i)/2, r_i]$ after considering points with $x_i \leq r_i$ and subtracting the number of points with values $[(l_i + r_i)/2, r_i]$ after considering points with $x_i \leq (l_i + r_i)/2$
- Be careful about points lying just on the border of triangles.

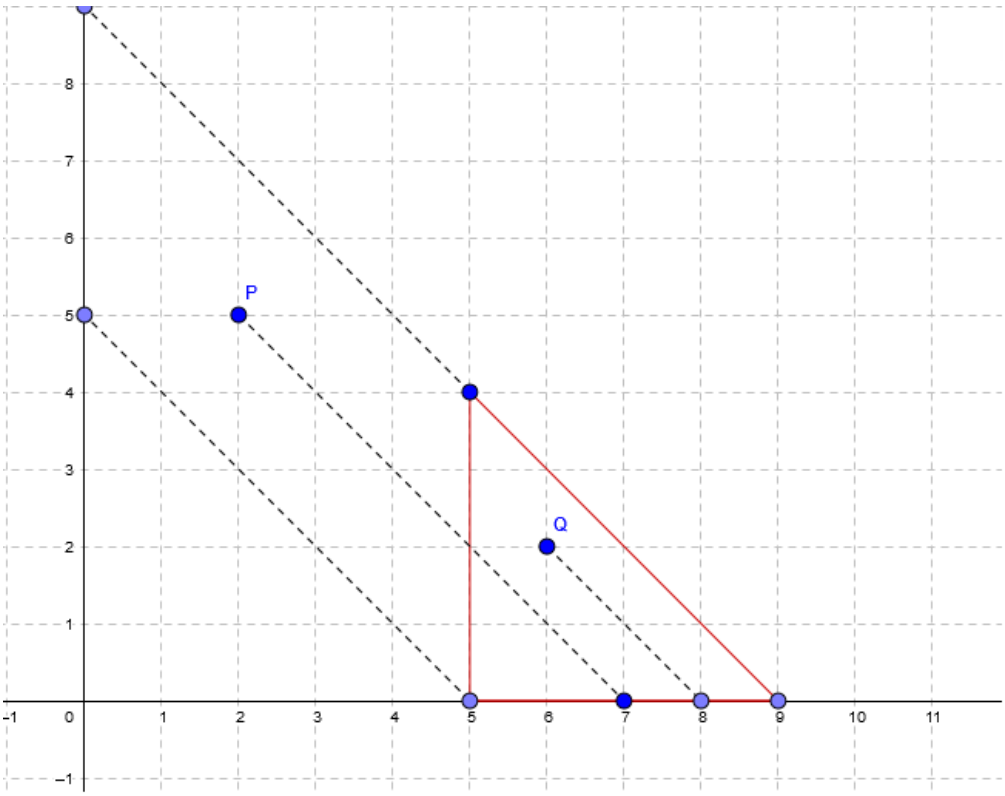
EXPLANATION

Scary quick explanation, right?
Well, that was all we need to do.

Let's split the triangles into two equal halves and try to count the number of points in each half separately and then add them while taking care not to include any point twice.

Refer the following images for a rough idea (Please ignore size, my first time with graphing tools, also suggest good graphing tools please).





Let's consider only left triangles for now. It is each to see that coordinates of the left triangle are of form $(a_i, 0)$, $(b_i, 0)$ and $(b_i, b_i - a_i)$

Considering the equation of each side of the triangle or even by observation, we can easily deduce that a point (x_i, y_i) lies in this triangle if $x_i \leq b_i$, $y_i \geq 0$ and $x_i - y_i \geq a_i$.
Since all points have $y_i \geq 0$, we can ignore that condition.

Let's order the points by x_i and left triangles by b_i in increasing order. Hence, some prefix of the point shall satisfy condition $x_i \leq b_i$.

The only condition left is the condition $x_i - y_i \geq a_i$. Let's assume a data structure supporting insertion of integer values and querying the number of values within a range.

We can use a segment tree to add values $x_i - y_i$ for each point and whenever we get a left triangle, we query for the number of values in range $[a_i, b_i]$.

Hence, we have counted the number of points in the left triangle.

For right triangle, we have two approaches, one is to replace all x coordinates with $10^6 - x$ and apply the idea for the left triangle.

The other one is a bit different. See the second image.
The coordinates of right triangle is of form $(a_i, 0)$, $(a_i, (b_i - a_i))$ and $(b_i, 0)$. We can derive conditions $x_i \geq a_i$, $y_i \geq 0$ and $x_i + y_i \leq b_i$

Once again we can use the segment tree, adding values $x_i + y_i$ to the segment tree. After considering all points with $x_i \leq b_i$, we can query for number of values in range $[a_i, b_i]$. But this shall also include point P as shown in the image. We can exclude that by subtracting the number of values in range $[a_i, b_i]$ after considering all points with $x_i \leq a_i$

Tips For such problems

- It is helpful to scale the whole grid by a factor of 2, to avoid dealing with half values.
- I know the above editorial is far longer and could be shorter, but Just as we derived conditions here, we can virtually derive the condition for any geometrical shapes.
- Try using test cases with points on the border, points close to the border for debugging.

TIME COMPLEXITY

The time complexity is $O((N+Q)*log(N+Q)) per test case.

PREREQUISITES

Tree, DFS, RMQ

PROBLEM

You are given a rooted tree with N nodes numbered $1, 2, \ldots, N$. Node 1 is the root node. Some of the nodes have a token in them. In one move, you can choose a non-root node that has a token, but its parent doesn't, and move the token from this node to its parent. What is the maximum number of moves you can make?

Note: When a token is moved out of a node, the node becomes empty, and other tokens will be able to move there.

QUICK EXPLANATION:

We can run DFS on tree as, $DFS(s)$: First call DFS for all children v of s. Now If node s initially had a token, then we can do nothing otherwise find the deepest token in its subtree and shift it to node s.

EXPLANATION:

The idea is that we can take the deepest token that has a free ancestor and move it to the closest (deepest) free ancestor.

Proof

Let us prove it by contradiction:

Let's look at an optimal solution that doesn't move this token. This token had more ancestor vertices than ancestor tokens.

So in the end:

- It has a free ancestor, which is definitely not optimal we can just move this token there as our goal is to maximize the number of moves.

Hence, to do so we will go through all the vertices from the deepest. If there is a token in it we will try to find the closest free ancestor and move this token there.

To simplify the implementation, we can do DFS on the tree as

$DFS(s)$: Such that all the nodes of subtree which are rooted at node s have been explored. Now If node s initially had a token, then we can do nothing otherwise find the deepest token in its subtree and shift it to node s.

Subtask 1:

$T \leq 10, N \leq 17$

Since the value of N is so small we can try every possible combination to shift which token to which node etc and find such a combination that maximizes the number of moves.

But yes definitely it is an overkill solution.

Solution

```
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define pii pair<int, int>
```

```
#define F first
#define S second
#define all(c) ((c).begin()), ((c).end())
#define sz(x) ((int)(x).size())
#define ld double

template<class T,class U>
ostream& operator<<(ostream& os,const pair<T,U>& p){
    os<<"("<<p.first<<", "<<p.second<<")";
    return os;
}

template<class T>
ostream& operator <<(ostream& os,const vector<T>& v){
    os<<"{";
    for(int i = 0;i < (int)v.size(); i++){
        if(i)os<<", ";
        os<<v[i];
    }
    os<<"}";
    return os;
}

#ifdef LOCAL
#define cerr cout
#else
#endif

#define TRACE

#ifdef TRACE
#define trace(...) __f(#__VA_ARGS__, __VA_ARGS__)
template <typename Arg1>
void __f(const char* name, Arg1&& arg1){
    cerr << name << " : " << arg1 << std::endl;
}
template <typename Arg1, typename... Args>
void __f(const char* names, Arg1&& arg1, Args&&... args){
    const char* comma = strchr(names + 1, ',');cerr.write(names, comma - names) << " : " <
}
#else
#define trace(...)
#endif

long long readInt(long long l,long long r,char endd){
    long long x=0;
    int cnt=0;
    int fi=-1;
    bool is_neg=false;
    while(true){
        char g=getchar();
        if(g=='-'){
            assert(fi==-1);
            is_neg=true;
            continue;
        }
        if('0'<=g && g<='9'){
            x*=10;
            x+=g-'0';
            if(cnt==0){
                fi=g-'0';
```

```
            }
            cnt++;
            assert(fi!=0 || cnt==1);
            assert(fi!=0 || is_neg==false);

            assert(!(cnt>19 || ( cnt==19 && fi>1) ));
        } else if(g==endd){
            if(is_neg){
                x= -x;
            }
            if(!(l<=x && x<=r))cerr<<l<<"<="<<x<<"<="<<r<<endl;
            assert(l<=x && x<=r);
            return x;
        } else {
            assert(false);
        }
    }
}
string readString(int l,int r,char endd){
    string ret="";
    int cnt=0;
    while(true){
        char g=getchar();
        assert(g!=-1);
        if(g==endd){
            break;
        }
        cnt++;
        ret+=g;
    }
    assert(l<=cnt && cnt<=r);
    return ret;
}
long long readIntSp(long long l,long long r){
    return readInt(l,r,' ');
}
long long readIntLn(long long l,long long r){
    return readInt(l,r,'\n');
}
string readStringLn(int l,int r){
    return readString(l,r,'\n');
}
string readStringSp(int l,int r){
    return readString(l,r,' ');
}
template<class T>
vector<T> readVector(int n, long long l, long long r){
    vector<T> ret(n);
    for(int i = 0; i < n; i++){
        ret[i] = i == n - 1 ? readIntLn(l, r) : readIntSp(l, r);
    }
    return ret;
}

const int SN = 1000;
// O(N 2^N)
int get(int n, vector<int> par,  string type){
    vector<vector<int>> children(n);
    for(int i = 1; i < n; i++){
        assert(par[i] >= 0 && par[i] < i);
        children[par[i]].push_back(i);
```

```
    }
    vector<int> depth(n);
    vector<int> masks(n);
    int sum = 0;
    vector<vector<bool>> isAncestor(n, vector<bool>(n, 0));
    vector<vector<bool>> isOK(1 << n, vector<bool>(n, 0));
    for(int i = 1; i < n; i++){
        depth[i] = depth[par[i]] + 1;
        sum += (type[i] - '0') * depth[i];
        isAncestor[0][i] = true;
        int u = i;
        while(u != 0){
            isAncestor[u][i] = true;
            u = par[u];
        }
    }
    for(int mask = 0; mask < (1 << n); mask++){
        for(int i = 0; i < n; i++) if(mask >> i & 1){
            isOK[mask][i] = true;
            for(int j = 0; j < i; j++) if((mask >> j & 1) && isAncestor[j][i]) isOK[mask][
        }
    }
    const int INF = 1 << 29;
    vector<vector<int>> dp(n, vector<int>(1 << n, INF));
    for(int s = n - 1; s >= 0; s--){
        if(type[s] == '1') masks[s] |= 1<<s;
        for(int v : children[s]){
            masks[s] |= masks[v];
        }
        dp[s][0] = 0;
        for(int submask = masks[s]; submask; submask = (submask - 1) & masks[s]){
            for(int i = 0; i < n; i++) if(isOK[submask][i]){
                int cost = depth[s];
                int mask = submask ^ (1 << i);
                for(int v : children[s]){
                    cost += dp[v][mask & masks[v]];
                    cost = min(cost, INF);
                }
                dp[s][submask] = min(dp[s][submask], cost);
            }
        }
    }
    return sum - dp[0][masks[0]];
}

int main(){
    int t; cin >> t;
    int sn = 0;
    while(t--){
        int n; cin >> n;
        string type; cin >> type;
        vector<int> par(n);
        for(int i = 1; i < n; i++){
            cin >> par[i];
            par[i]--;
        }
        cout << get(n, par, type) << endl;
    }
}
```

Subtask 2:

The sum of N over all test cases doesn't exceed 2000.

So during DFS if we are at some node say s (such that its subtree is already explored) which doesn't have a token. Then we need to find the deepest node in its subtree which has a token so that we can shift that token to this node.

We can find this deepest node by doing DFS again on this subtree since the value of N is small enough and it does allow us to do DFS again. Once we found that node we shift that node token to node s adding the number of moves that were needed to our answer.

Subtask 3:

The sum of N over all test cases doesn't exceed 10^5.

The idea is the same $i.e$ during DFS if there is a node that doesn't have a token then we will simply find the deepest node that has a token on its subtree and will shift that token to this node.

But finding the deepest node in the subtree by traversing each node again will result in TLE as the value of N is large enough this time.

To optimize it, we can maintain the depths of nodes with the help of multiset + offset, use small to large merging.

This results in a $O(N * log^2 N)$ solution which will be good enough to pass this subtask.

Solution

Subtask 4:

Original Constraints

As the value of N is large enough to get the TLE verdict for our $O(N * log^2 N)$ solution. We can optimize it further to the $O(N * logN)$ solution by using Euler Tour and Range Minimum Query.

We can simply do Euler Tour in the given tree and build the RMQ structure of this Euler tour. So when we are at node s which doesn't have a token on it then we can make a query on the subtree and can find the deepest token using RMQ.

Hence we are able to optimize our solution to $O(N * log(N))$.

TIME COMPLEXITY:

$O(N * log(N))$ per test case

PREREQUISITES:

Basic observations, segment tree or fenwick tree

PROBLEM:

You have to update your array based on the given values in each query,and
output certain soldiers strength

EXPLANATION:

(approach using segment tree)
Create two different arrays for each army and initialize it with 1
(this arrays signify multiplication factor of each soldier)

now update the range based on queries
update range of winning kingdom by increasing their factor by 1 and
update range of losing kingdom by decreasing their factor by 1.

Both these operations can be done in O(Logn) time

and finally output the soldiers current strength by quering the multiplication factor in O(Logn) time and multiplying it with initital strength of that soldier

ALTERNATE EXPLANATION:

The same thing can be done using Fenwick tree

PREREQUISITES:

sweep line, segment tree

PROBLEM:

Given **N** segments on Cartesian plane, each one is either vertical or horizontal, count the number of intersection points between any pair of the segments.

EXPLANATION:

General explanation

Let's imagine vertical line moving from very far left (say X=-inf) to very far right (say X=inf) -let's call this line as "sweeping line"-, at every moment this line is intersecting with some subset of horizontal segments so we can keep track with this subset as long as the sweeping line is moving, now whenever that line meets vertical segment, say it's endpoints are (X1,Y1) and (X2,Y2) such that X1=X2 and Y1 < Y2, then the answer to its query is the number of horizontal segments which are now intersecting with the sweeping line and their Y-coordinates is between Y1 and Y2.

so if we can find a way to quickly count number of horizontal segments in the subset which have Y-coordinates between any given range then we can answer the query of all vertical segments. however, we also need to support add/erasing segments from the subset.

note that we can do the same idea but with a horizontal sweeping line to answer the queries of horizontal segments.

representing the sweeping line and its movement technically

We want to find a way to store the subset of horizontal segments which are currently interesting with the sweeping line and support quickly adding new segment to this set and support removing existing segment from this set.

This can be done by consider the sweeping line as an array **A** of length $10^5 + 1$ if there's currently segment intersecting with sweeping line at Y-coordinate equal to **Y** then we have **A[Y] = 1** otherwise, we have **A[Y] = 0**, when we want to answer a query to vertical segment then we just need to find the sum of elements in this array inside a given range, so can use segment tree or Fenwick tree data structure.

to simulate the movement of the sweeping line we are only interested in some events that will occur during the movement of the sweeping line which are

- The sweeping line meets with a vertical segment, because we need to answer its query
- sweeping line intersect with new horizontal segment
- sweeping line stop intersecting with a horizontal segment

so we create an array of events that will store all events of any of the above 3 types and sort them according the X-coordinates where it will happen, then we simply do the events in that order.

Corner Cases

- The problem said "at least two other directions to continue their trip" so we should intersecting points that has extensions in 3 or 4 directions but not 2, that means an intersection point can be endpoint of at most of segment but not two, so for simplicity we can include the case that has 2 extensions then exclude them alone, so for now an intersecting can be an endpoint of 2 segments
- when we sort the events in case two or more events have same X-coordinates then we put first events of adding new horizontal segments first, then the events of querying vertical segments and lastly the events of erasing horizontal segments, this is because an intersection point can be an endpoint of a segment.

- also because an intersection point can be an endpoint of a segment, then the range sum query should be inclusive.