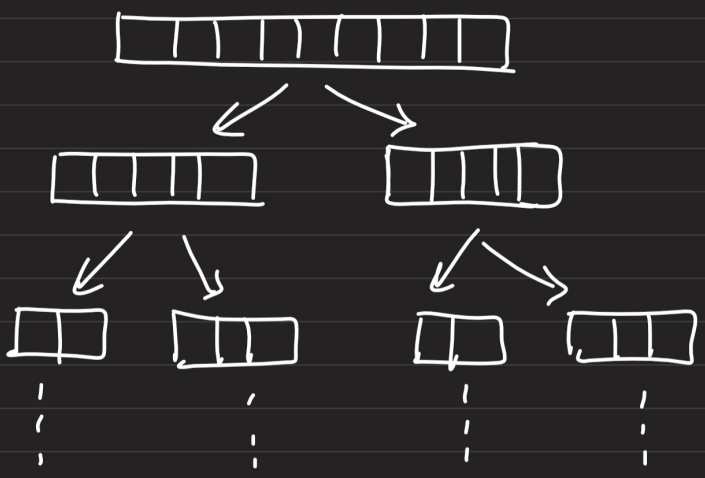
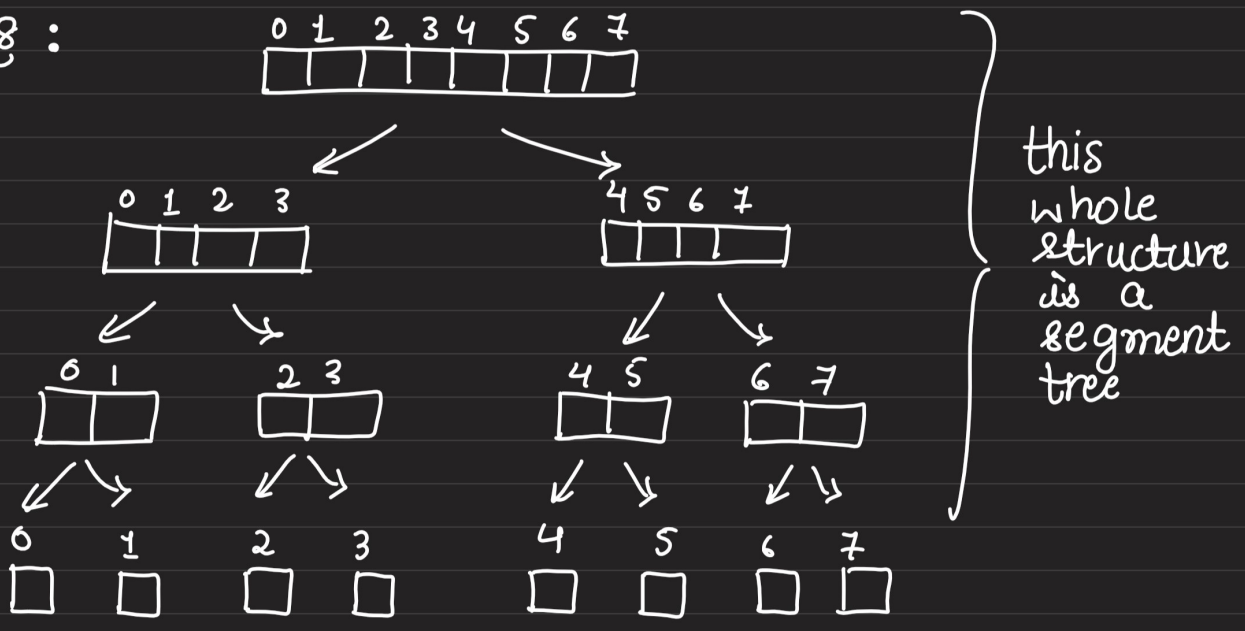
# Segment Trees

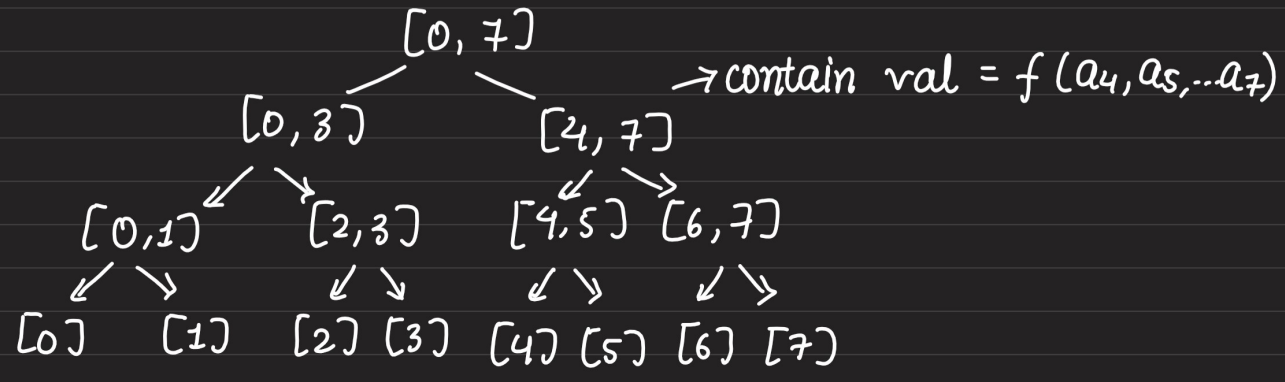
## Saving divide and Conquer states



N = 8 :




## Each node is a Range



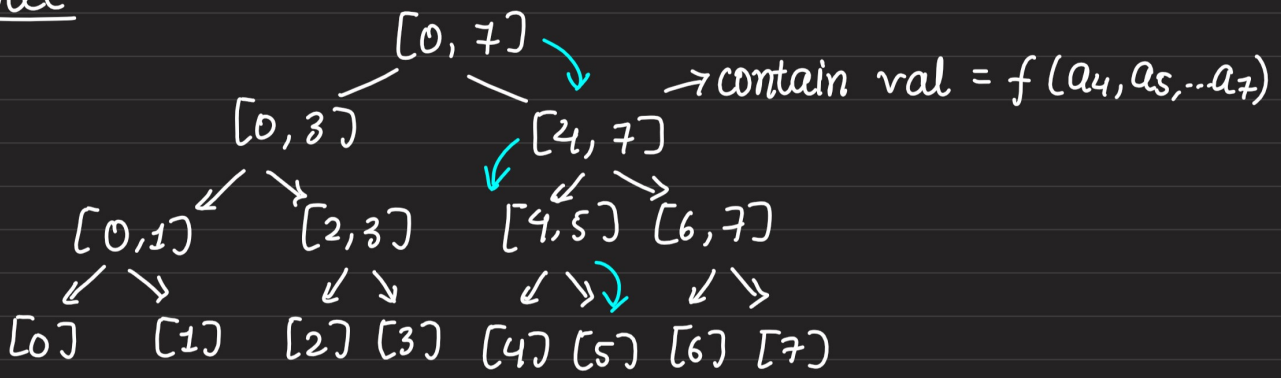
## for N sized array

Height of seg. tree = $O(\log n + 1)$

$\#nodes = n + \frac{n}{2} + \frac{n}{2^2} + \cdots \frac{n}{2^r} = 2n - 1$

$= O(n)$ } fsbl to store all the nodes in an array

## # Tree




To go to node([5]) } → follow blue path

Complexity to travel from root node to child node = $O(\log n)$

Q1 N array , Q queries

1 l r → find $\sum_{i=l}^{r} a_i$ } Query type 1

2 i x → update $a_i = x$ } Query type 2

soln:

Prefix Sum : Type 1 : O(1) p[r] - p[l-1]

Type 2 : O(N)

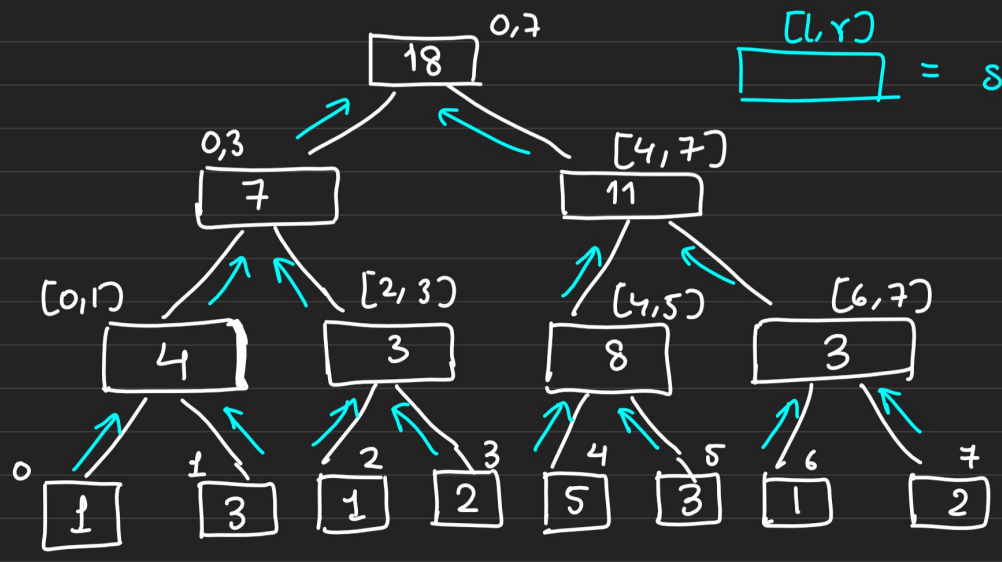
eg:

A = [1 2 3 1 2]
P = [1 3 6 7 9] } After updating we will have to create a whole new P array.

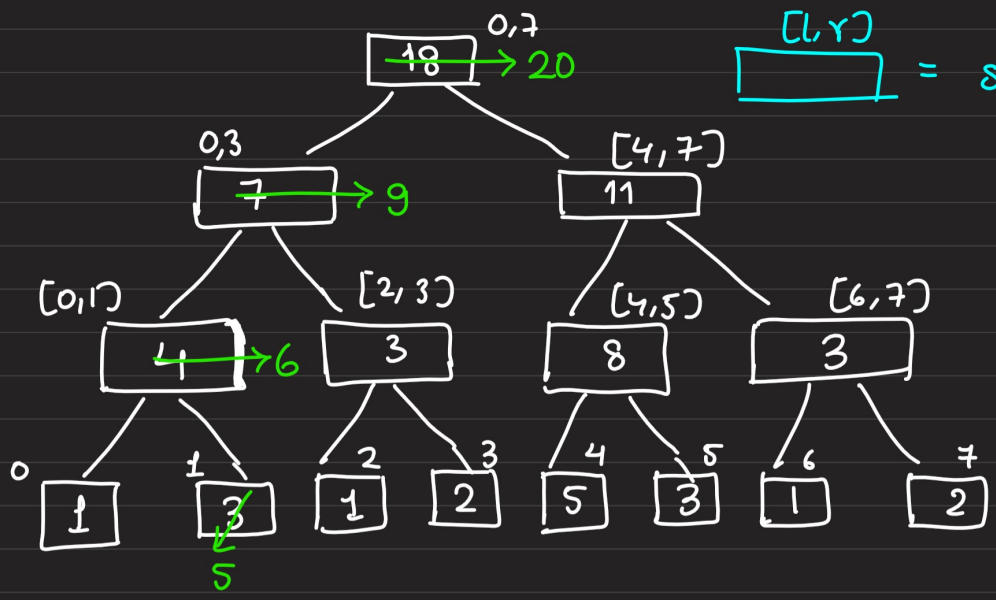
↳ TLE

Segement tree for this array

A = [1 3 1 2 5 3 1 2] n = 8




node [l,r] = $\sum$ node [childs]




# update (arr[1] = 5) → we will travel to node 1 from the top.

#operations = O(log N)

∴ we can handle query of type 2 into O(log N)

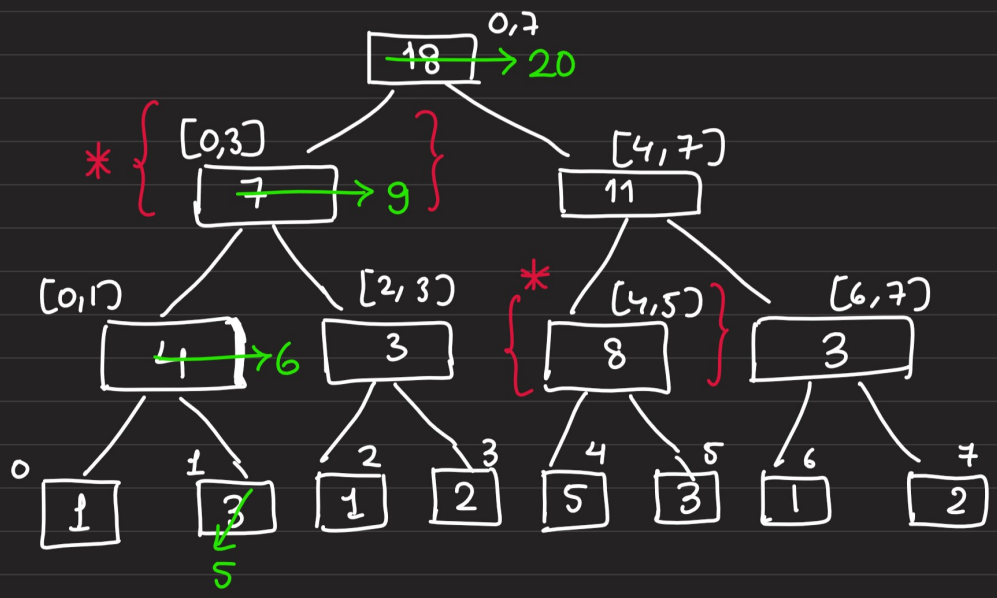
# Query 1

sum ( l=0, r=5 )

Ans = 9 + 8

Q How to find * nodes.?

soln: We want to find the top level nodes in this range




Here, [0,3] ∪ [4,5]

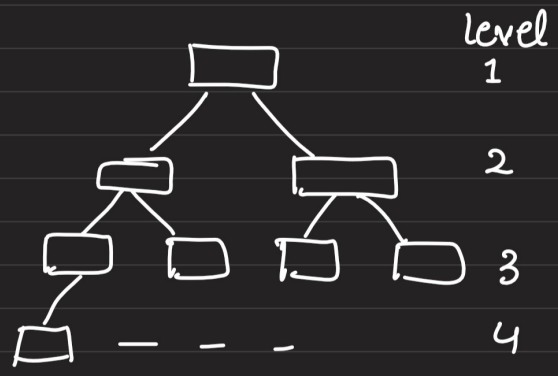
## Theorem

If $[l, r] = \{N_1, N_2, \cdots, N_m\}$ , i.e, $[l, r] = N_1 \cup N_2 \cup \cdots N_m$

then: $M = O(\log N)$

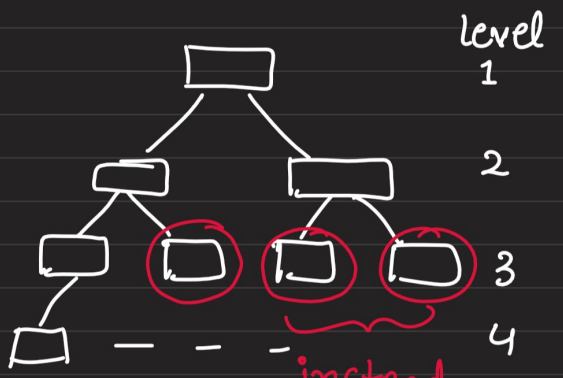
proof: $C_i \rightarrow$ no. of nodes at level $i$ in the solution set.

$$M = C_1 + C_2 + \cdots + C_{\log N}$$




We can always say that $C_i \leq 2$

Suppose $C_3 \geq 3$




} continuous 3 because of continuous range

instead of 2 we can take its parent

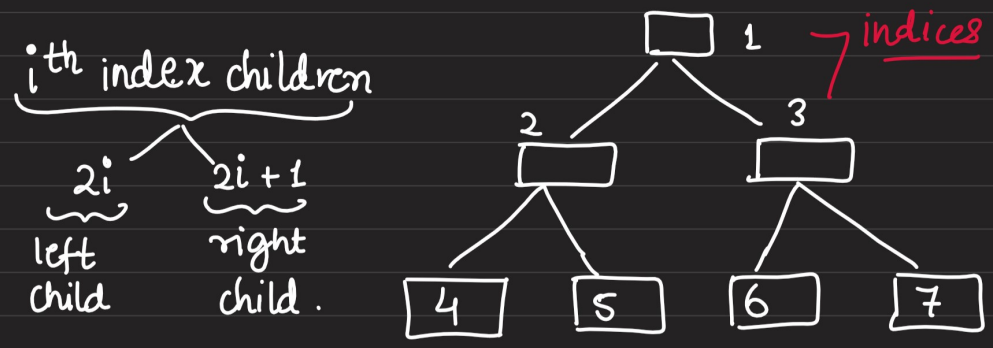
$\therefore$ $M = O(\log N)$

We always try to find the first matching }

Now, sum is also done in $O(\log N)$ time.

# Implementation

Representation : root node → index 1

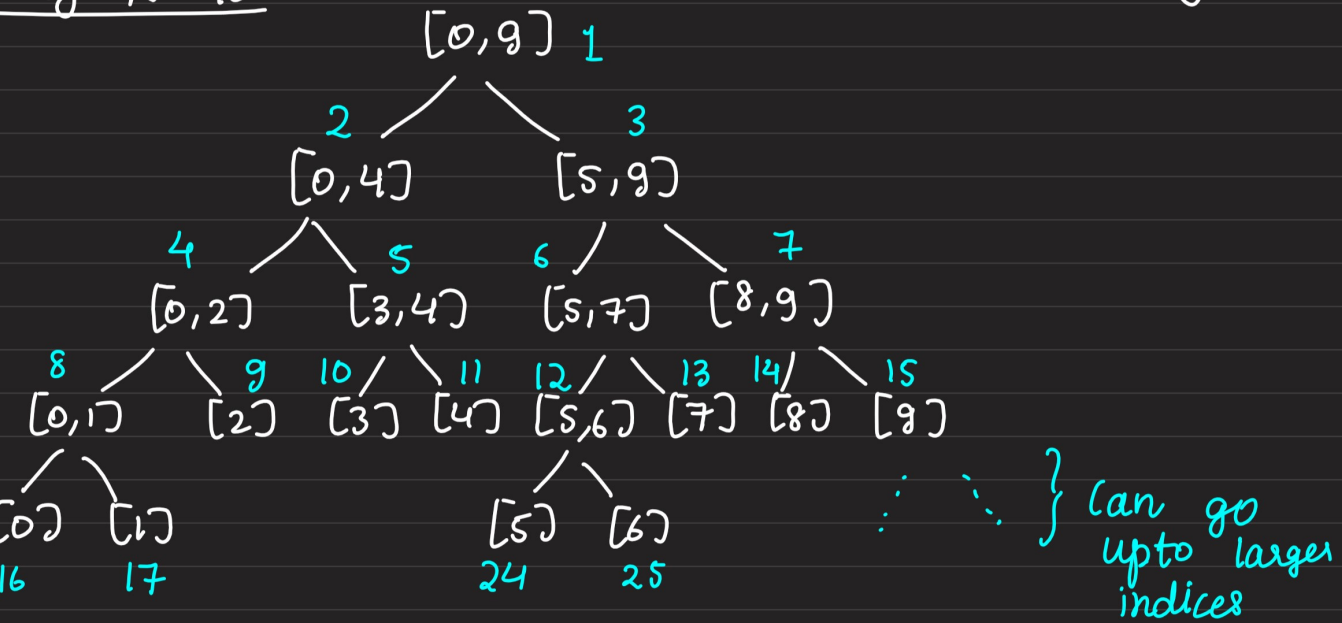



We can flatten the tree into a single array.

seg_tree :  } size = 4n

why

say N = 10




So, for N sized, for safety always have size as 4N.

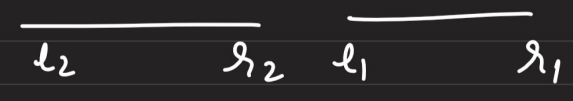
# Check if ranges $[l_1, r_1]$, $[l_2, r_2]$ is intersecting or not.

Soln: Check non intersecting is easier (less cases)

(1) 

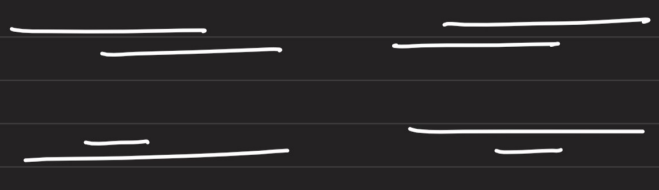

$r_1 < l_2$

OR

(2) 


$r_2 < l_1$

Intersecting cases



} larger in number.

# Building

```
//node represent the range [start, end]
void build(int node, int start, int end)
{
    if(start==end)
    {
        //leaf node
        tree[node] = a[start];
        return;
    }
    int mid = (start + end)/2;

    //build left child
    build(2*node,start,mid);

    //build right child
    build(2*node+1,mid+1,end);

    tree[node] = tree[2*node] + tree[2*node+1];

    return;
}
```

} node represents [start, end] → indices of a[].

} for leaf node, tree[node] = a[start].

→ DnC

} first building left and right child.

} then building the parent.

# Updating in logN time

```
//Node represents -> [start, end]
//Update A[i] -> X
void update(int node, int start, int end, int i, int x)
{
    if(start == end){
        a[start] = x;
        tree[start] = x;
        return;
    }
    int mid = (start+end)/2;

    //Left child 2*node = [start, mid]
    //Right child 2*node+1 = [mid+1, end]
    if(i <= mid) update(2*node,start,mid,i,x);
    else update(2*node + 1,mid+1,end,i,x);

    //updating current node
    tree[node] = tree[2*node] + tree[2*node+1];
    return;
}
```

} Index i is to be updated to value x.

} once we found the leaf node representing i → update the value.

} updating the required child.

} updating the parent.

# Processing Query

```
//Node represents -> [start, end]
//Find the sum from [l,r]
lli query(int node,int start, int end, int l, int r)
{
    if(end<l || start > r)
    {
        //[start,end] not intersecting [l,r] -> no need for this
        return 0;
    }
    //[l.................r] => st-end is subrange of l-r
    //[   st.......end     ]
    if(start>=l && end<=r)
    {
        return tree[node];
    }

    //[start, end] is partially intersecting
    int mid = (start+end)/2;
    return (query(2*node,start,mid,l,r) + query(2*node+1,mid+1,end,l,r));
}
```

} If range is not intersecting the curr range, return 0.

} If range is totally inside the required range

} for partially intersecting search for intersections in both subtree.

NOTE: If query is called more than 2 times in recurrence, then time complexity becomes O(N)

# $T(n) = T\left(\frac{n}{2}\right) + O(1)$

$O(n^{\log_2 1}), O(1)$ } same $\Rightarrow \log(n)$

# $T(n) = 2T\left(\frac{n}{2}\right) + O(1)$

$O(n^{\log_2 2}), O(1)$ } $O(N)$

Q2. N size array, Q queries

$1 \le N, Q \le 10^5$

- 1 l, r → $\min(a_l, a_{l+1}, \ldots, a_r)$
- 2 i x → $a_i = x$

Soln: Very small modification, instead of sum, take minimum.

---

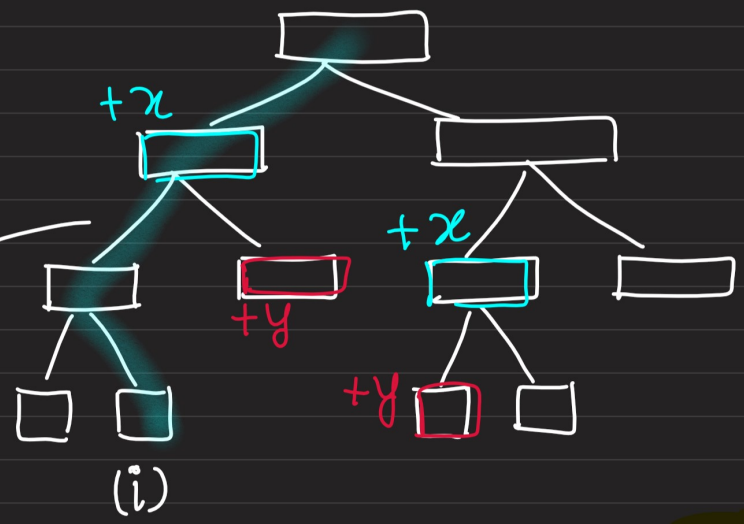
Q3 N array, Q queries $N, Q \le 10^5$

- 1 l r x → $(a_l + x, a_{l+1} + x, \ldots a_r + x)$

  $A_i \; t = x \quad \forall \; i \in [l, r]$
- 2 i → find $A_i$

Segment tree of increments } Initially all zero.

We increment x in those l, r subsets




2 i

find sum of increment in total path.

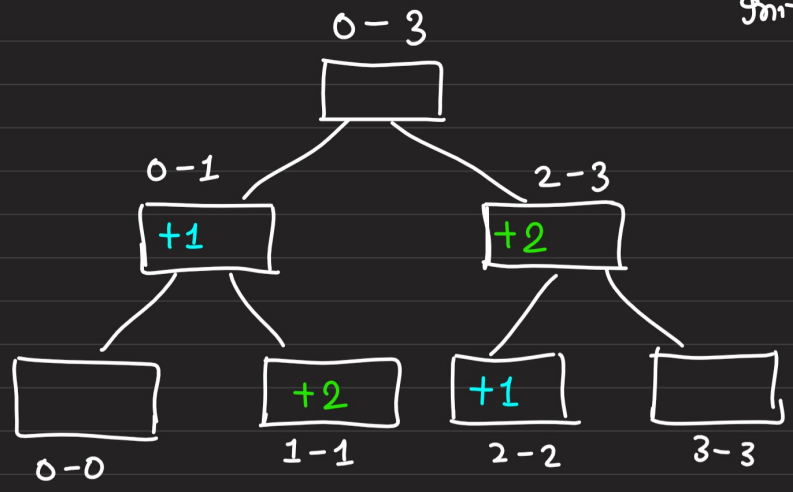
↳ total increment for the child node i.

Shakil

# Solution Idea

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| arr : | 0 | 0 | 0 | 0 |

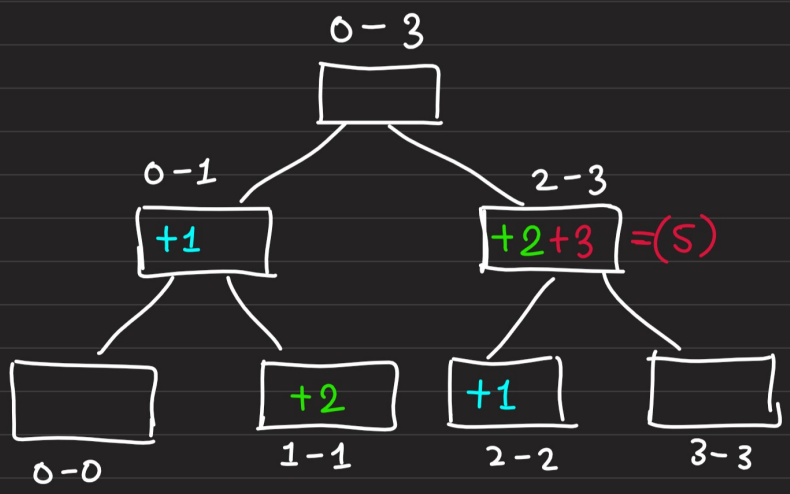
Initially all elements zero.




| | | | | |
|---|---|---|---|---|
| | 1 | 0 | 2 | 1 |
| | 1 | 1 | 3 | 2 |
| A) | 2 | 2 | | |
| | 1 | 2 | 3 | 3 |
| B) | 2 | 3 | | |

1 l r x } Instead of adding x to (r-l+1) elements, we update O(log N) nodes whose union is [l, r]

for Query A, val[2] = sum of values of all the ancestors
= 3




for Query B, val[3] = 5

Actually, sum of all the ancestor values → gives us the change in value of a node.

No Build is required, since, initially all the values are already zero.

```
//Node represents -> [start, end]
//Find the  from [l,r]
void update(int node,int start, int end, int l, int r, int x)
{
    if(end<l || start > r)
    {
        //[start,end] not intersecting [l,r] -> no need for this
        return; //no need to update anything
    }
    //[l.................r] => st-end is subrange of l-r
    //[   st.......end     ]
    if(start>=l && end<=r)
    {
        tree[node]+=x;
        return;
    }

    //[start, end] is partially intersecting
    int mid = (start+end)/2;
    update(2*node,start,mid,l,r,x);
    update(2*node+1,mid+1,end,l,r,x);
}
```

} updating the increment in the ranges

## Query Processing

```
//No Build required as intially all are already 0
//Node represents -> [start, end]
//Update A[i] -> X
lli query(int node, int start, int end, int i)
{
    if(start == end){
        return tree[node];
    }
    int mid = (start+end)/2;

    //Left child 2*node = [start, mid]
    //Right child 2*node+1 = [mid+1, end]
    if(i <= mid) return tree[node] + query(2*node,start,mid,i);
    else return tree[node] + query(2*node + 1,mid+1,end,i);
}
```

} returning the total increment

} adding up increments going down the ranges.

this gives us the total increment at i. → 

```
cout << a[i] + query(1,0,n-1, i);
```

# Segment tree:

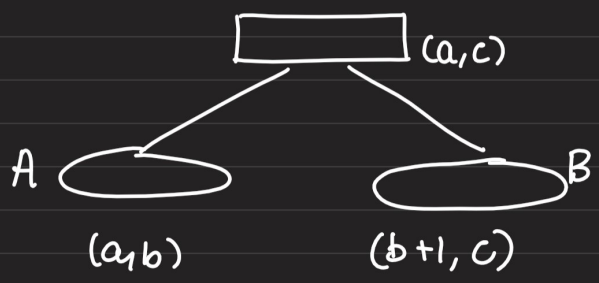
- i) Point update, Range Query
- ii) Range update, Point Query
  - we save Δ (change in Metric)
  - query(i) → total Δi
- iii) Range update, Range Query.
- iv) Point update, Point Query.

Q4. N → arr[]

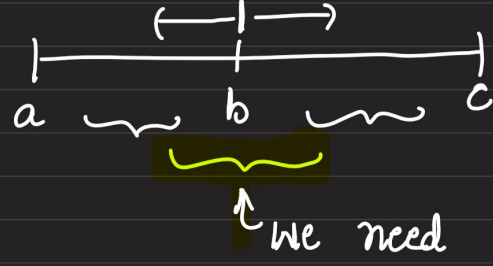
Queries, Q : 1 x V → update arr[x] = V

2 l r → find max sum subarray in [l,r]

Soln:




We know ans for (a,b) & (b+1,c). Using this, how can we find ans for (a,c)

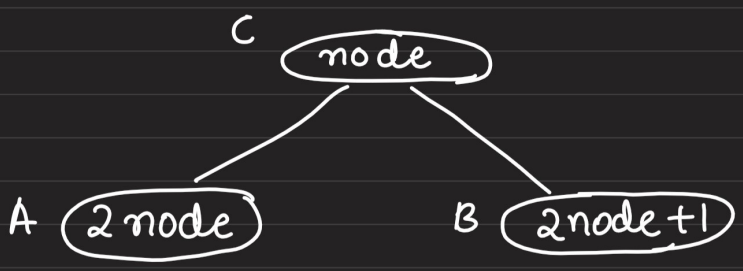



} We also need to find the best subarray intersecting at b/b+1.

↑ We need to compare this as well.

Node structure → maxsum, lsum, rsum, total_sum

lsum → max sum prefix, rsum → max sum suffix

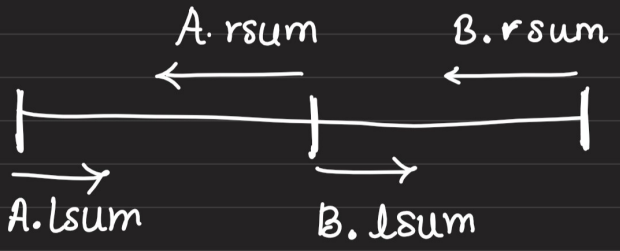



$$msum(node) = \{ \underbrace{A.msum,\ B.msum,\ A.rsum + B.lsum}_{\text{take max across all}} \}$$

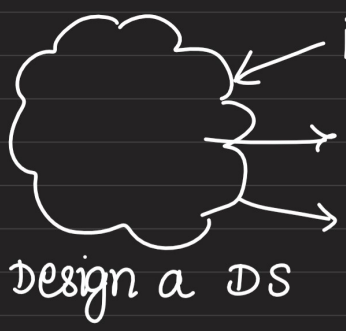
$$lsum(node) = \max(A.lsum,\ A.total\_sum + B.lsum)$$

$$rsum(node) = \max(B.rsum,\ B.total\_sum + A.rsum)$$

$$C.total\_sum = A.total\_sum + B.total\_sum.$$



Q4



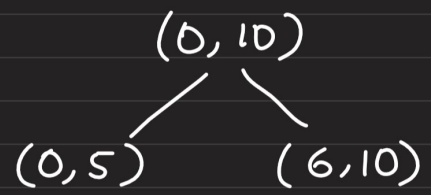

Create a set with random access.

Q queries
+ x – add x to the set
– x – remove x if exist
? k – find $k^{th}$ element (k ≤ size())

Constraints

$N \leq 10^5$ , $Q \leq 10^5$ , $X \leq 10^5$

sol$^n$:

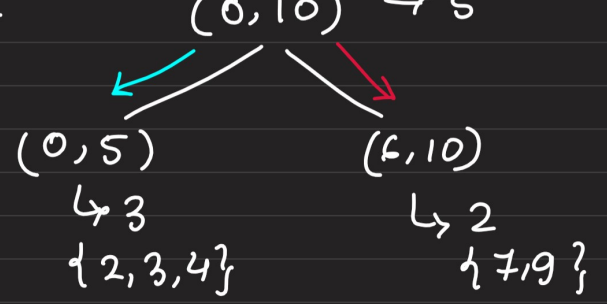



(l, r) : #of values in set in range (l, r)

eg:

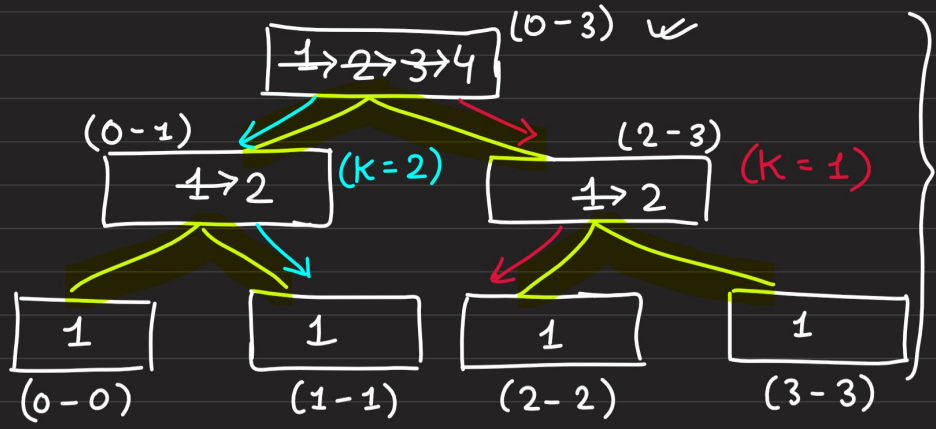
2 3 7 9 4

finding



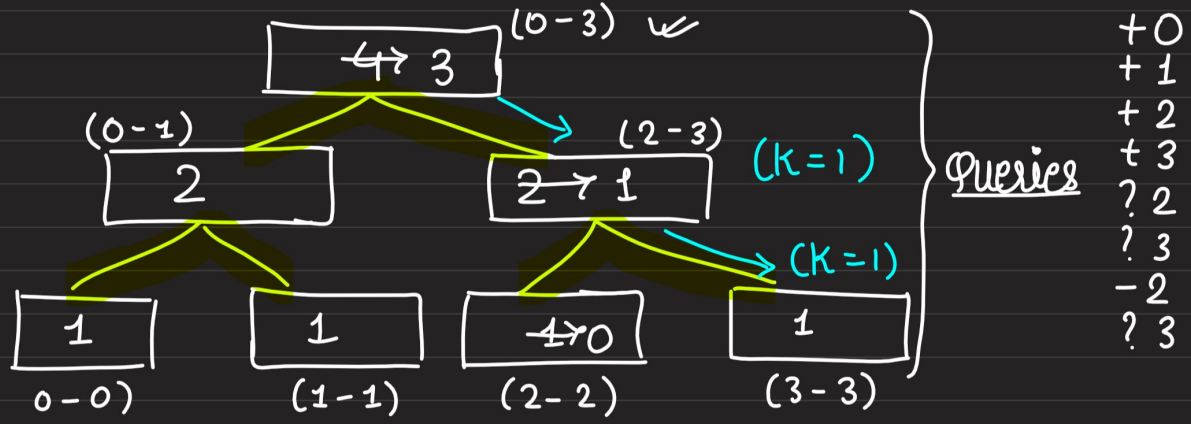

we can prune down the range.

- say k = 2 , which side will the 2nd element lies Move on the left side. k remains same
- say k = 4 , Move on the right side., then find $(4-3)^{th}$ element , k changes to 1.




?2 : follow blue line . Ans = 1.

?3 : follow Red line . Ans = 2




-2 ? go to leaf node (2-2), do a -1, & the update parents.

? 3 : follow blue line. Ans = 3.

```
find_kth (node, st, end, k)
{
    if (l == r)
    {
        return l;
    }
    if (k ≤ tree[2*node])
        return find_kth (2node, st, mid, k)
    else
        return find_kth (2node+1, mid+1, r, k - tree[2node])
}
```

Q5 N people with all distinct height in a line.

h[i] → heights of persons

Person 1
Person 2
⋮
Person N

we don't know which height is of which person.

for every index i, we know how many persons before it has height greater than him.

(ii) → $\sum_{j=0}^{i-1}$ truth [arr[j] > arr[i]] } given for every i

soln: h[] = {1, 2, 3, 4, 5}

taller[] = {0, 1, 2, 0, 1}

- 0 → 1st person have no taller persons before it.
- 1 → last person have 1 taller person before it.

corresponds to ↓

arr[] = {3, 2, 1, 5, 4} } Ans

height in correct order

Given height and taller array, find arr[].

lets start from the back :-

Taller[] = {0, 1, 2, 0, 1}

h[] = {1, 2, 3, 4, 5} sorted

## lets start from the back

Taller[] = {0, 1, 2, 0, 1}  h[] = {1, 2, 3, 4, 5} sorted

↓

only one taller 6e4 it } ⇒ have to be the 2nd largest. i.e, 4.

Now, remove 4 from h[].

Taller[] = {0, 1, 2, 0}  h[] = {1, 2, 3, 5}

↳ 1st tallest = 5 } h.remove (5)

Taller[] = {0, 1, 2}  h[] = {1, 2, 3}

↳ 3rd tallest = 1 } h.remove (1)

⋮

finally, we get required ans[]

★ We need a data-structure to find the $K^{th}$ tallest element.

for this we need segment tree.

↳ last problem.

# Coordinate Compression

Q6 


Queries $N, Q \leq 10^5$

DS contains elements only from the arr given.

1 (x)

if x is present in DS, remove it else add it.

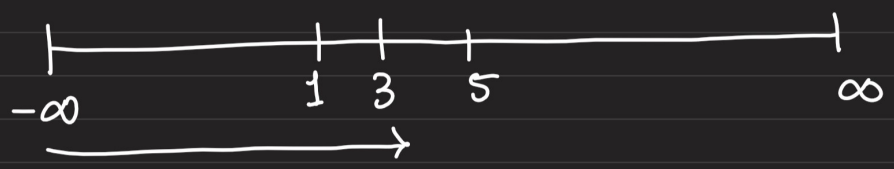
2 k

no. of element in set $\leq k$.

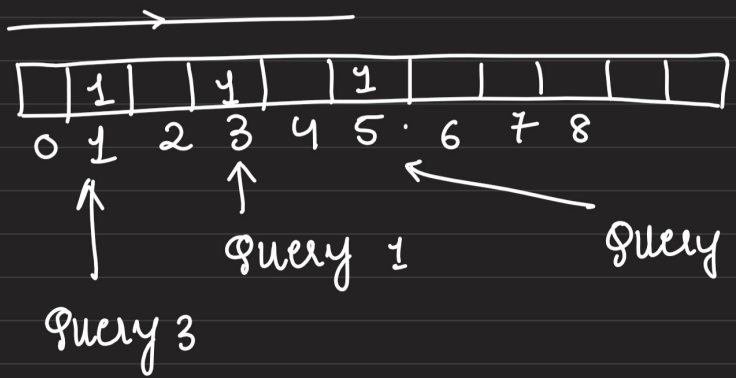
eg: Q

1 1 } DS {3, 5, 1}
1 2 }
1 3
2 4 → print 2
1 3 DS {3, 5}
2 4 → print 1.

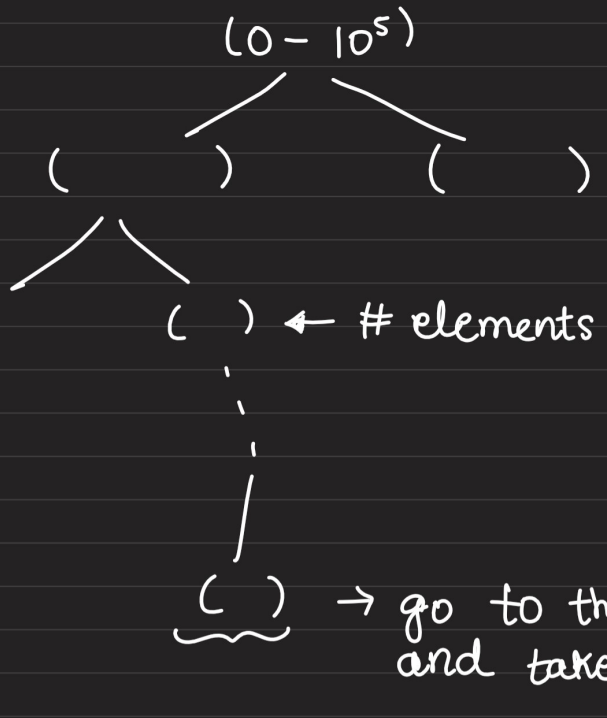
sol$^n$:




Query Ans = prefix_sum.




prefix sum - upto k.




for find_k → find no. of elements in $[0, k]$

This approach will work good for $arr[i] \leq 10^5$.

**What if $arr[i] = 10^9$ ?**

↳ we can't really build this segment tree.

**Coordinate compression**

↳ mapping values of array into values $[0, N]$

| 3 | 5 | 1 | $10^9$ |
|---|---|---|---|

↓ replacement value

$1 \rightarrow 0$
$3 \rightarrow 1$
$5 \rightarrow 2$
$10^9 \rightarrow 3$

} we will maintain the property that if $a < b$ then $map[a] < map[b]$

Now, we can create seg_tree with root representing range $[0, n-1]$.

We will map all values in array $\leq k$.

then instead of findi. numbers in range $[0, K]$, we will find numbers in range $[0, map(num)]$, where num is the number just less than or equal to K.

Q7 $arr[] = \{3\ 5\ 7\ 1\ 2\ 1\}$, $A[i] \leq 10^9$, $D \leq 10^9$

$D = 2$. Find longest subsequence $P_1, P_2, \ldots, P_x$ s.t $abs(P_i - P_{i+1}) \leq D$

$N \leq 10^5$

soln: $Arr = \{a_1\ a_2\ a_3\ a_4\}$

# DP approach

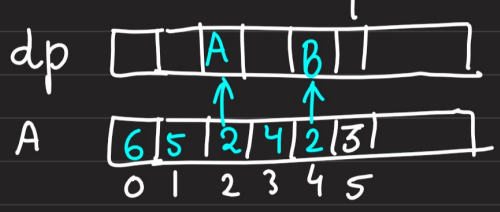
$dp(i)$ = longest subsequency ending at $i$

$dp[i] = max(dp(j) + 1, 1)$

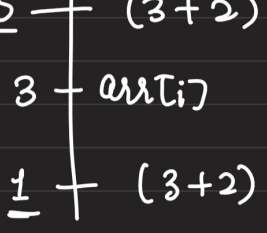
$j < i$ & $|A[i] - A[j]| \leq D$

Time Complexity : $O(N^2)$

Optimization :




$D = 2$




$arr[i] - D \leq arr[j] \leq arr[i] + D$

$j \leq i$

Best seen

$x \rightarrow$ what is the best subsequence length ending at value $x$ (not index)

BS(2)
= max(A, B)

for calculating ans for 3 (dp[5]), we can take max among the max of previous values in range of [1, 5] + 1.

We want max of range [arr[i] - D, arr[i] + D]

↓

This can be done using segment trees.

for each value found, we update our segment tree.

But $\underbrace{A[i], D \leq 10^9}$

for this we will use coordinate compression.

# Lazy Propagation

Q1. N  $N, Q \le 10^5$

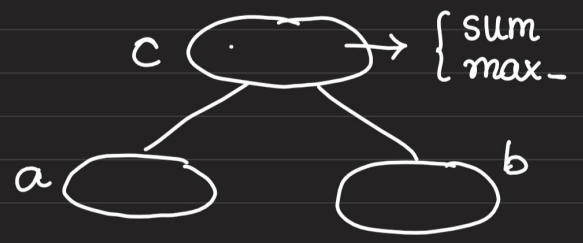
Q queries

1 l r → find sum of elements in range [l, r]

2 l r → find max of elements in range [l, r]

Range Update { 3 l r v → A[i] = v for all i ∈ [l, r]

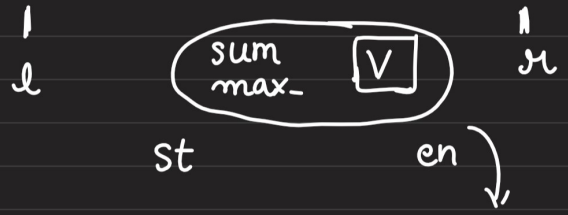
soln:




# c.sum = a.sum + b.sum

# c.max = max(a.max, b.max)

Range update ?? (3 l r v)

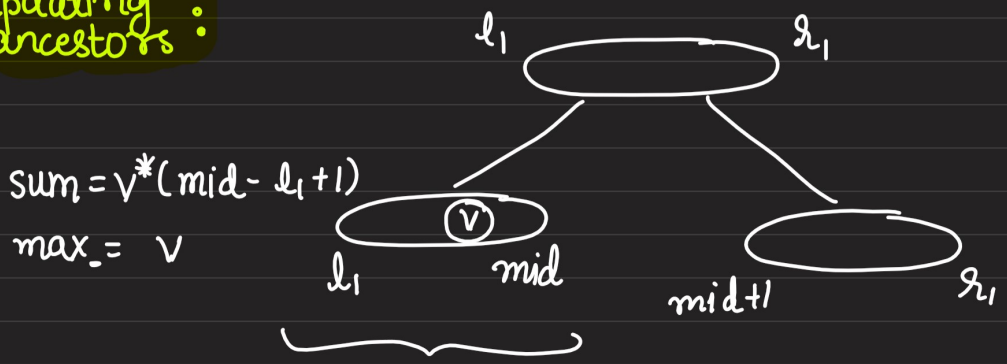



$l \le st \le en \le r$ } every leaf should change.

TLE

We will mark lazy, with value V.
For marking lazy → time complexity O(logN)

Updating ancestors :



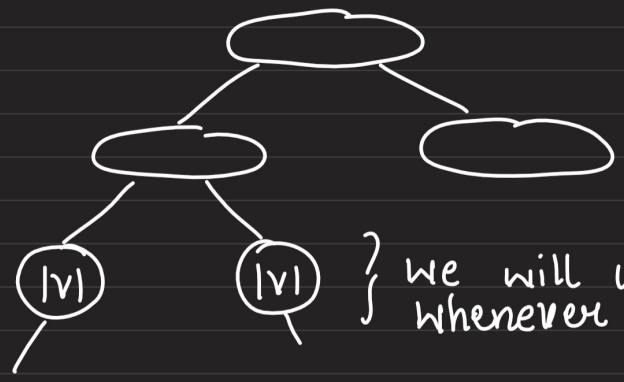

updating this node will give correct update for all the ancestor } this is called push logic.

**Updating Subtree :** First update the value of node marked lazy, & update ancestors

↓

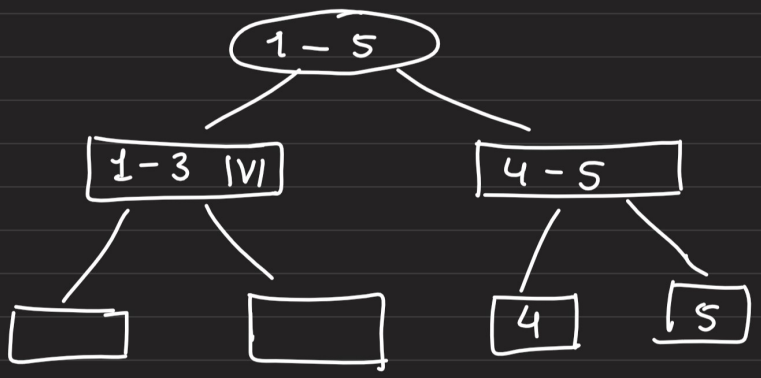
Now, mark its childs to lazy.

→ dont pass it to the whole subarray just the childs.



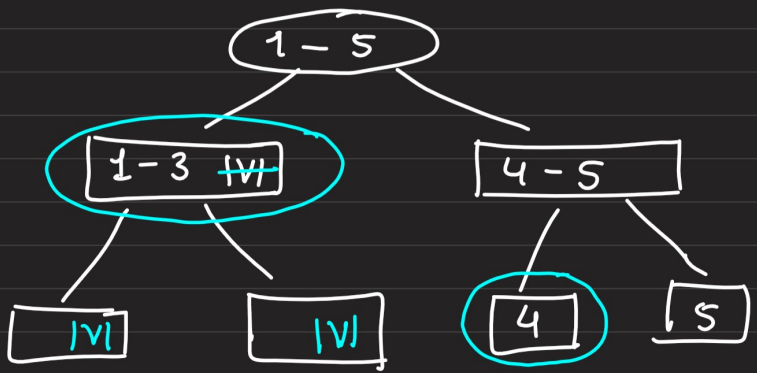

} we will use this whenever required.

Query




We want to query 1 - 4

go to 1 - 3, if node is lazy, update it & make it non-lazy. Mark their childs as lazy.



## When 2 apply lazy ?

2 key Problem property

i) Correction of range using push. You just need the update value to update the property of range No required of subtree information.

ii) Lazy should be passable. (Lazy merge)

# Sweep Line

Q1 N intervals. Find total length covered under segments where atleast K intervals intersect.

$[l_1, r_1]$
$[l_2, r_2]$
:
$[l_n, r_n]$

eg: K = 2, N = 5

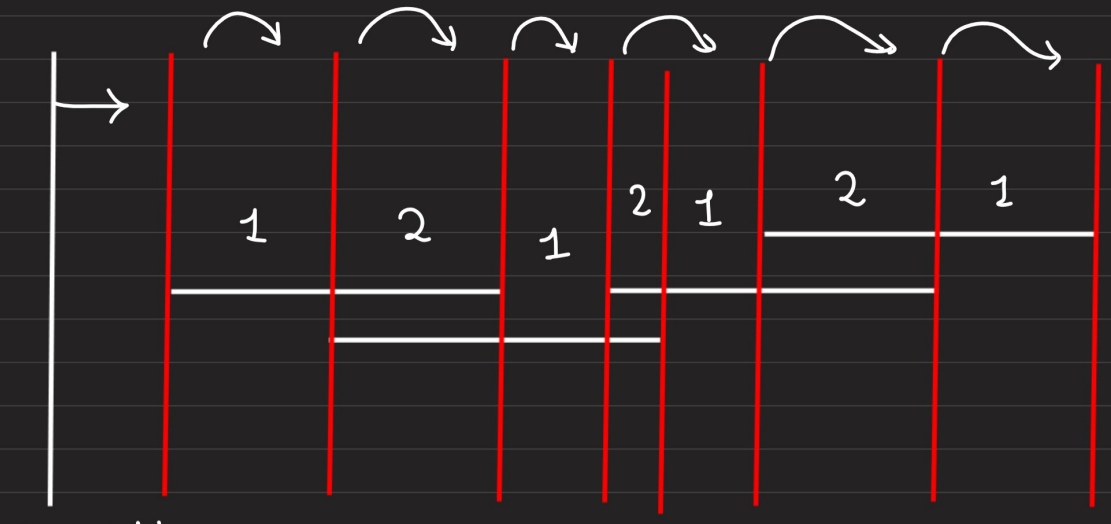

find sum of green lies.

↳ tells us about # intersecting intervals

Obs. Numbers are changing only at the end points.

Either start/end point of an interval are of our interest.

Total ≡ 2N point (st, end) of each interval




event points are in increasing order.

## Implementation :-

```cpp
void solve()
{
    lli n,k;cin>>n>>k;
    vector<ii> events;

    for(int i=0;i<n;i++)
    {
        int l,r; cin>>l>>r;
        events.push_back({l,0});
        events.push_back({r,1});
    }
    sort(events.begin(),events.end());

    int cur_intersecting_lines =  0;
    int ans = 0;
    //Sweeping
    for(int i=0;i<(int)events.size();i++)
    {
        if(events[i].second == 0)
        {
            cur_intersecting_lines++;
        }else{
            cur_intersecting_lines--;
        }

        if(cur_intersecting_lines>=k && i+1 < (int)events.size())
        {
            ans += events[i+1].F - events[i].F;
        }
    }
    cout<<ans<<'\n';
}
```

→ O(n)

→ O(2n log(2n))

→ O(2n)

TC :

O(nlogn)

Q2 N segments

$[l_1, r_1, z_1]$ $z_i \rightarrow$ value of the segment.

$[l_2, r_2, z_2]$

$\vdots$

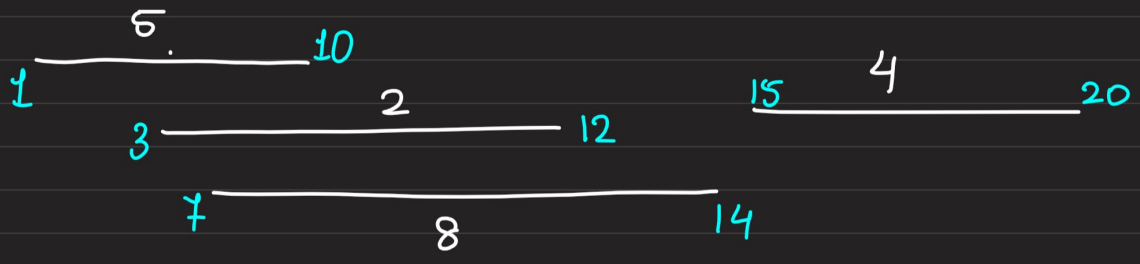
$[l_n, r_n, z_n]$

Q queries

$x_1, x_2, \ldots x_Q$

For $x_i$ query, find the largest value among all the segments in which $x$ belongs.

eg:




$x = 4$, belongs to $[1, 10, 5]$, $[3, 12, 2]$ } Ans = 5

$x = 11$, belongs to $[3, 12, 2]$, $[7, 14, 8]$ } Ans = 8

$1 \leq N, Q < 10^5$

$1 \leq l_i, r_i, z_i, x_i \leq 10^9$

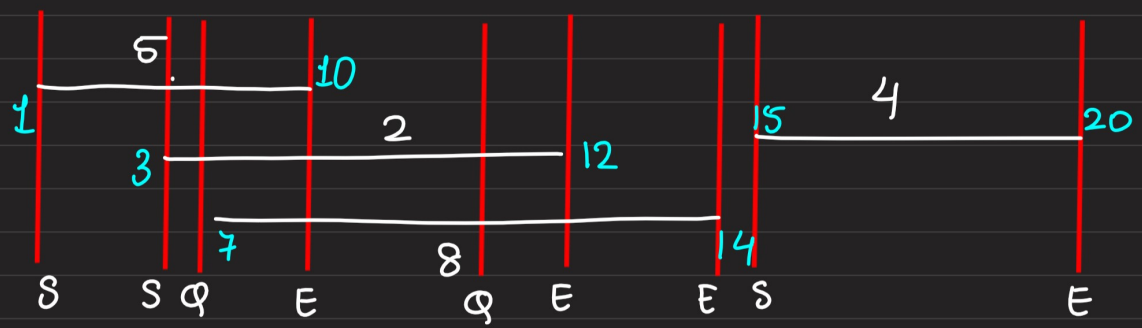
$(l_i \leq r_i)$

} Constraints

Soln: Q queries } Offline processing.

$x_1 \rightarrow$ ans 1
$x_2 \rightarrow$ ans 2
⋮
$x_q \rightarrow$ ans q

} We will process answer in our order & then we will print it in req. order.




events

[l, start] } at start, we put the value of the interval in our data structure.
[r, end]
[x, query] } at query event point, we will find max value from our DS.

at end, we remove the value of the interval.

- We need to take query in sorted manner.
- Data-structure, we can use ⇒ multiset.

Checking edge case :


} We need to make sure that x's ans should be max of both segments

So, if : $x = l_i$ ⇒ first add the segment, then query, then remove.

```cpp
void solve()
{
    lli n,q;cin>>n>>q;
    //[l,0,z], [x,1,index], [r,2,z] → Insert/Query/Remove
    vector<pair<ii,lli>> events;
    vector<lli> ans(q);
    for(int i=0;i<n;i++){
        int l,r,z; cin>>l>>r>>z;
        events.push_back({{l,0},z});
        events.push_back({{r,2},z});
    }
    for(int i=0;i<q;i++){
        lli x; cin>>x;
        events.push_back({{x,1},i}); //we need to keep the index of the orignal query
        //to answer them in sorted manner
    }
    sort(events.begin(),events.end());
    multiset<lli> st;
    //Sweeping
    for(int i=0;i<(int)events.size();i++){
        if(events[i].F.S == 0) //start
        {
            st.insert(events[i].S);
        }
        else if(events[i].F.S == 2) //end
        {
            st.erase(st.find(events[i].S));
        }
        else //query
        {
            if(!st.empty()){
                ans[events[i].S] = *(st.rbegin()); //nlogn
            }
        }
    }
    for(int i=0;i<q;i++)cout<<ans[i]<<'\n';
}
```

Time complexity

$O((N+Q)\log(N+Q))$

we took $[l,0,z]$, $[Q,1,ind]$, $[r,2,z]$ because for edge case, we want to first insert, then query and then remove.

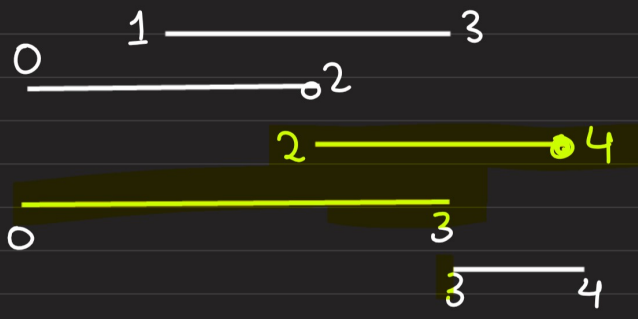
if $l = Q = r$ } we uses 2nd val for sorting, i.e, 0, 1, 2.

# Solving Interval - 1

Find minimum no. of intervals to cover the whole range [0, L).


Ans = 2

[0,4), [3,8]




Ans = 2

We can take interval that is farthest from the curr. taken interval.

0 → 2 active intervals [0, 2), [0, 3)

larger, r so taken

Moving, till that larger r, → taking the interval that extends farthest from r.

## Logic :

Sort all the intervals : [0,2)[0,4) [2,5) [3,6) [3,8) [5,8)

Initially, covered point c is 0. So the first interval with L>c is [2,5). Till here, we take the interval that covers maximum distance, including start=0 i.e., interval [0,4), and the count of the interval is 1.

Points covered till now are c=4. Again take the first interval with L>c i.e.[5,8). In between, take the interval that covers maximum distance after [0,4) and starts before or at 4. Here such an interval is [3,8). Again count of interval increments by now.

The points covered till now are 8 which is the length of the total number line. So the value of count = 2 is the minimum number of intervals required. The intervals are [0,4) and [3,8).

TIME COMPLEXITY: O(NlogN)

```cpp
bool comp(ii a, ii b)
{
    if(a.F==b.F) return a.S > b.S;
    return a.F<b.F;
}

void solve()
{
    lli n,L;cin>>n>>L;
    vector<ii> intervals;
    for(int i=0;i<n;i++)
    {
        lli l,r; cin>>l>>r;
        intervals.push_back({l,r});
    }
    sort(intervals.begin(),intervals.end(),comp);

    lli r = intervals[0].S;
    multiset<lli> ms;

    if(r==L){
        cout<<"1\n";
        return;
    }

    lli ans = 1;

    for(int i=1;i<n;i++)
    {
        if(intervals[i].F <= r)
        {
            ms.insert(intervals[i].S);
            if(intervals[i].S==L)
            {
                ans++;
                break;
            }
        }
        else
        {
            r = *ms.rbegin();
            ans++;
            ms.insert(intervals[i].S);
            if(intervals[i].S==L)
            {
                ans++;
                break;
            }
        }
    }
    cout<<ans<<'\n';
}
```

Implementation

# Kickstart 2021

## Problem

You have just heard about a wonderful festival that will last for $\mathbf{D}$ days, numbered from 1 to $\mathbf{D}$. There will be $\mathbf{N}$ attractions at the festival. The $i$-th attraction has a *happiness rating* of $\mathbf{h_i}$ and will be available from day $\mathbf{s_i}$ until day $\mathbf{e_i}$, inclusive.

You plan to choose one of the days to attend the festival. On that day, you will choose up to $\mathbf{K}$ attractions to ride. Your *total happiness* will be the sum of happiness ratings of the attractions you chose to ride.

What is the maximum total happiness you could achieve?

## Input

The first line of the input gives the number of test cases, $\mathbf{T}$. $\mathbf{T}$ test cases follow.

The first line of each test case contains the three integers, $\mathbf{D}$, $\mathbf{N}$ and $\mathbf{K}$. The next $\mathbf{N}$ lines describe the attractions. The $i$-th line contains $\mathbf{h_i}$, $\mathbf{s_i}$ and $\mathbf{e_i}$.

## Output

For each test case, output one line containing `Case #x:` $y$, where $x$ is the test case number (starting from 1) and $y$ is the maximum total happiness you could achieve.

## Limits

Memory limit: 1 GB.
$1 \leq \mathbf{T} \leq 100$.
$1 \leq \mathbf{K} \leq \mathbf{N}$.
$1 \leq \mathbf{s_i} \leq \mathbf{e_i} \leq \mathbf{D}$, for all $i$.
$1 \leq \mathbf{h_i} \leq 3 \times 10^5$, for all $i$.

## Test Set 1

Time limit: 20 seconds.
$1 \leq \mathbf{N} \leq 1000$.
$1 \leq \mathbf{D} \leq 1000$.

## Test Set 2

Time limit: 90 seconds.
For at most 10 test cases:

- $1 \leq \mathbf{N} \leq 3 \times 10^5$.
- $1 \leq \mathbf{D} \leq 3 \times 10^5$.

For the remaining cases, $1 \leq \mathbf{N}, \mathbf{D} \leq 1000$.

Almost same as problem 1.

Just maintain topk sum Data structure.

↳ then use sweepline.

Q3 N intervals → $[l, r]$ $N \leq 10^5$ $l_i, r_i \leq 10^9$

find
(i) union of all
(ii) intersection of all

sol$^n$:


Intersection is always $[\max(l_i), \min(r_i)]$ and if $\max(l_i) > \min(r_i) \Rightarrow$ no intersection.

$\max(l_i)$: Things thats start late
$\min(r_i)$: end as early as possible.

→ O(N)

Union → Sweep line

Sweep line (maintain how many lines, it is intersecting)

- Event points → start and end of the segments.