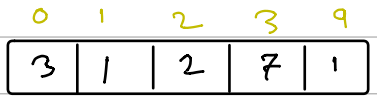
Segment Tree

Efficient Querying of Interval / Range
Efficient Updation of Interval / Range.

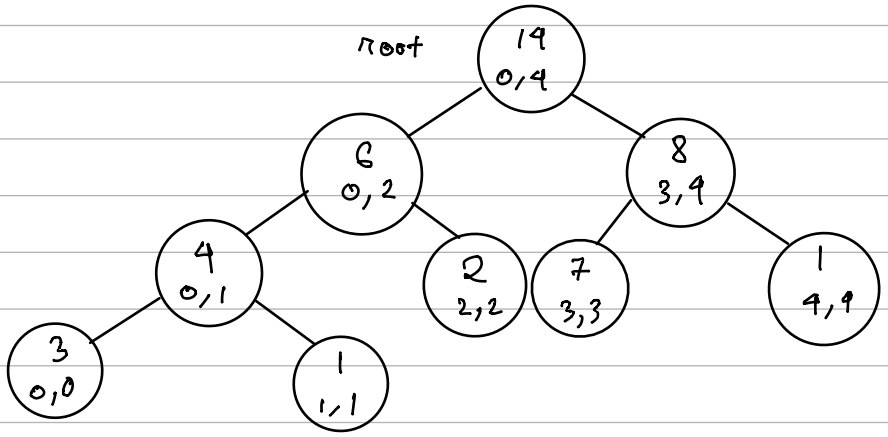
Range Sum :-



root consists of:

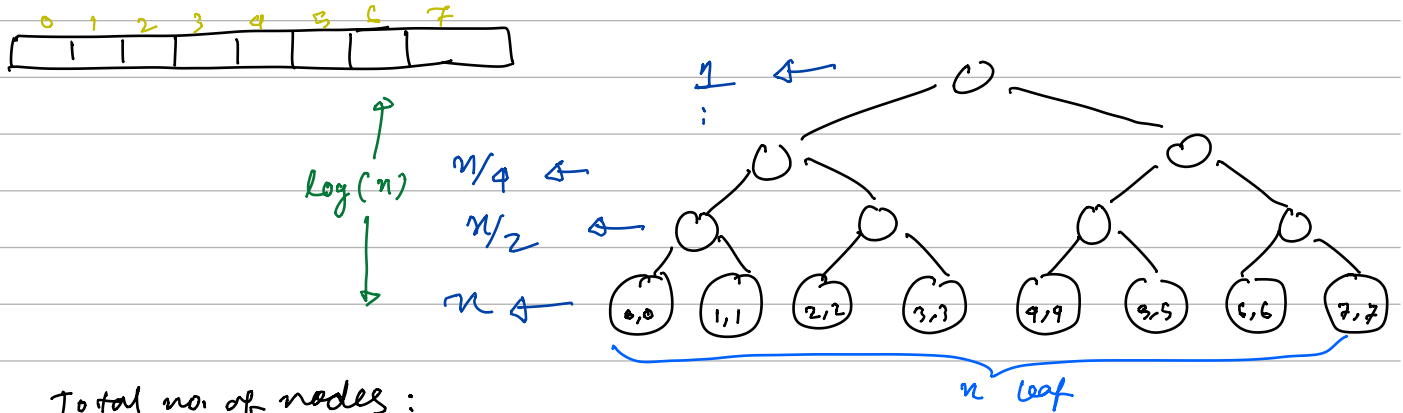
1) value
2) range a, b
3) 2 childs a, mid
   - mid+1, b

mid = a+b/2



Property:

- Binary tree,
- 2 children of all non leaf node.
- Leaf : represent one element of arr.
  root : represent whole arr.
  other nodes : represents an interval.
- height = $\lceil \log_2(n) \rceil$
- It's always a Balanced BST
  height(subtree 1) − height(subtree 2) ≤ 1



Total no. of nodes :

$$N + N/2 + N/4 + \cdots + 1 = N\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) \approx 2N \quad (N = \infty)$$

* Total leaf node = N
interval nodes = N−1

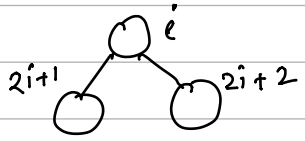
Total N + N−1 = 2N − 1 nodes

* Using Prefix-Sum Sum query problem solution is viable but not for other logic like max, min element in an Interval.

Building Segment Tree :

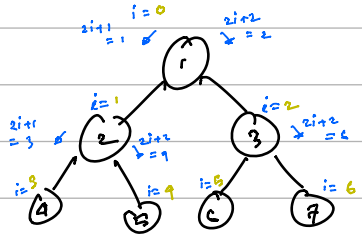
2 Method we can build Segment tree. (any tree)

- TreeNode* root .....
- Using an array



any tree can be represented in an array

if index[P] = i index[left child] = 2i+1
index[right child] = 2i+2



0	1	2	3	4	5	6
1	2	3	4	5	6	7

Recursively

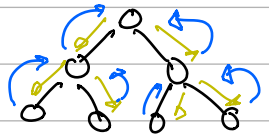
root (idx, l, r)
- buildTree(2idx+2, mid+1, r)
- buildTree(2idx+1, l, mid)

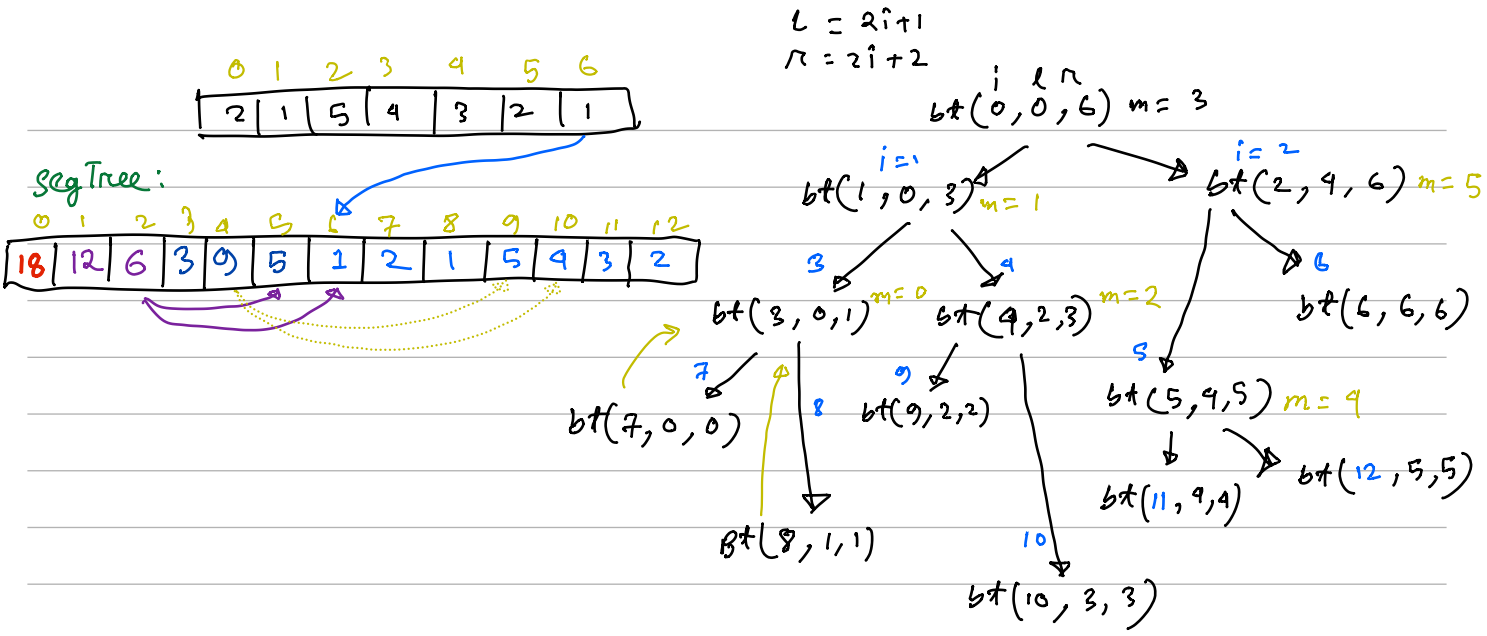
```
void buildTree(i, l, r)
{ if (l == r)
    { segTree[i] = nums[l];
      return nums[l]; }

  mid = l+r/2
  buildTree(2i+1, l, mid)
  buildTree(2i+2, mid+1, r)
  segTree[i] = segTree[2i+1] + segTree[2i+2] }
```

Space Complexity : O(2N) + O(N) stack memory

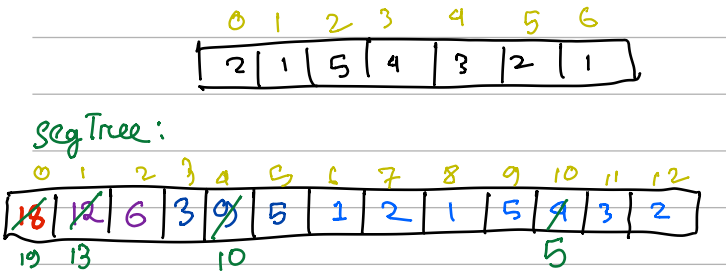
Time Complexity : 2 * O(2N) ≈ O(N)

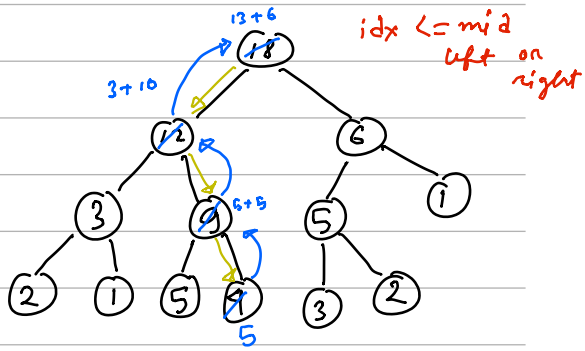
every node is visited twice





Update Query in Segment Tree :-





// update idx 3 = 5

Only traversal of height log n.

Better than prefix sum coz there

update is from i to end.

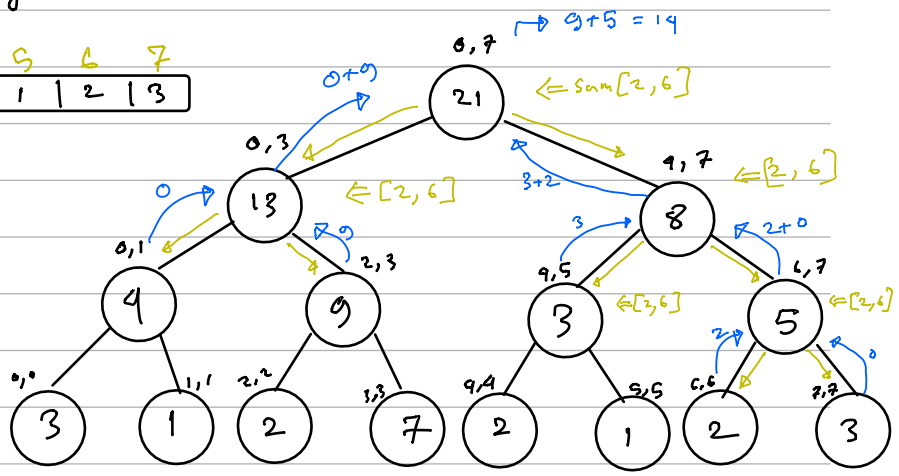
```
void updateSegTree(idx, val, i, l, r)
{   if(l == r)
    {   segTree[i] = val;
        return;  }

    mid = l+r/2
    if(idx <= mid){  // updating left subTree
        updateSegTree(idx, val, 2i+1, l, mid) }
    else {           // updating right subtree
        updateSegTree(idx, val, 2i+2, mid+1, r) }

    segTree[i] = segTree[2i+1] + segTree[2i+2]
}
```

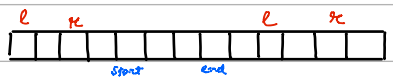
Range Sum Query :-

nums : | 3 | 1 | 2 | 7 | 2 | 1 | 2 | 3 |
0 1 2 3 4 5 6 7

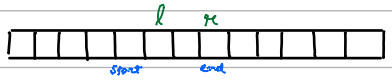
Query Sum [2, 6]

ans = 19

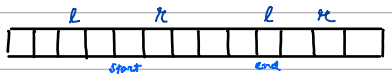


①  & val l, r — Totally Out Of Range

if (r < start || l > end) return 0

②  & val l, r — completely inside Range

if (r <= end && l >= start) return segTree[i];

③  & val l, r — Overlaping

return Query (leftvalue) + Query (Right value)

→ return query (start, end, 0, 0, n-1)

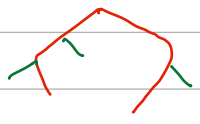
```
int query(start, end, i, l, r)
{   if (l > end || r < start)
        return 0
    if ( l >= start && r <= end)
        return segTree[i]
    return query(start, end, 2i+1, l, mid)
         + query(start, end, 2i+2, mid+1, r) ; }
```

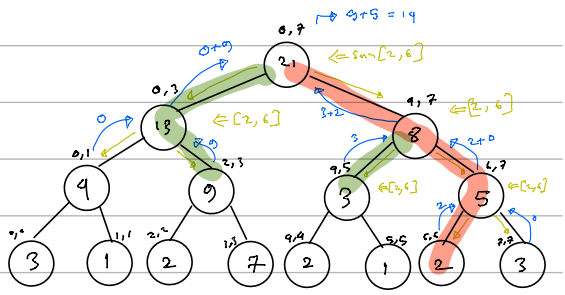
→ Time Complexity :-

max depth is traversed at max twice. so

$O(\log n + \log n)$

$\approx \log(n)$

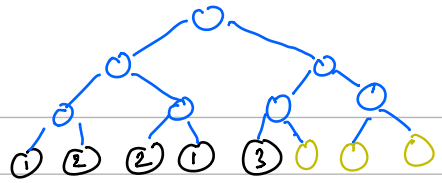




For Q queries $O(Q \log n)$

n = 5 1, 2, 2, 1, 3

↳ normal SegTree size = 9 (2×5−1)



but every pair may not match ;

take the next 2's Power after 9 = 16, then it'll match.

next 2's power after 9 is ≈ 9×2 = 18

For safety assign 2(2n−1) ≈ 4n size for the SegTree.

0 Range Update + Lazy Propagation :

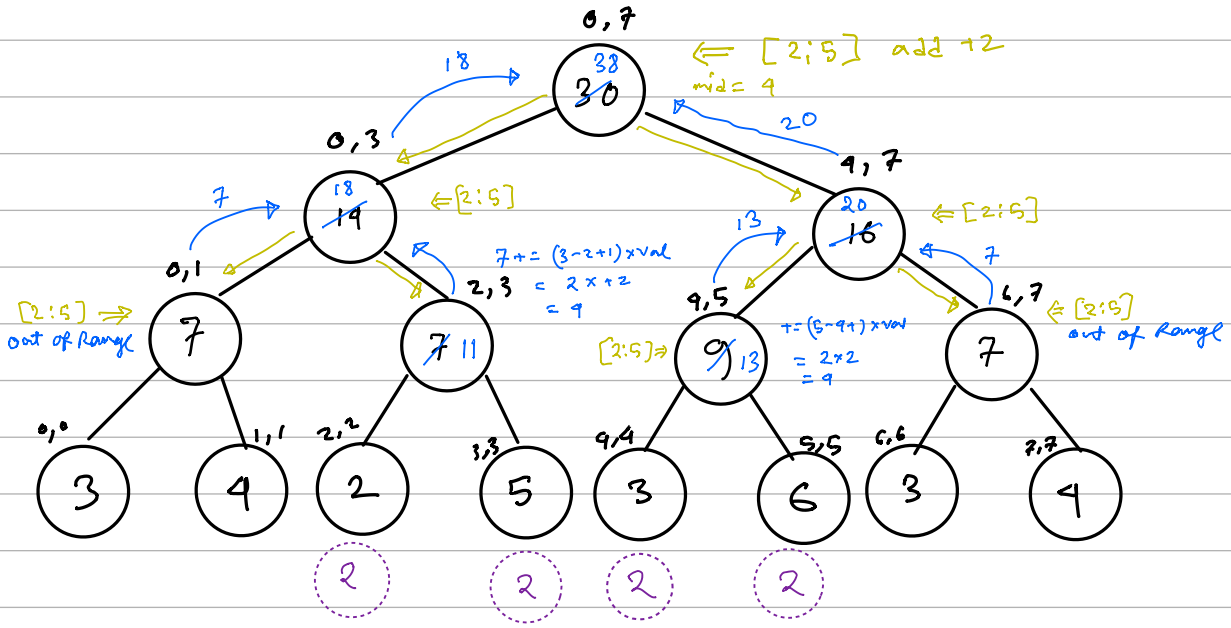
nums: 3, 4, 2, 5, 3, 6, 3, 4 update range[2:5] add +2

SegTree :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
~~30~~ 38	~~14~~ 18	~~16~~ 20	7	~~7~~ 11	~~9~~ 13	7	3	4	2	5	3	6	3	4	

Lazy :

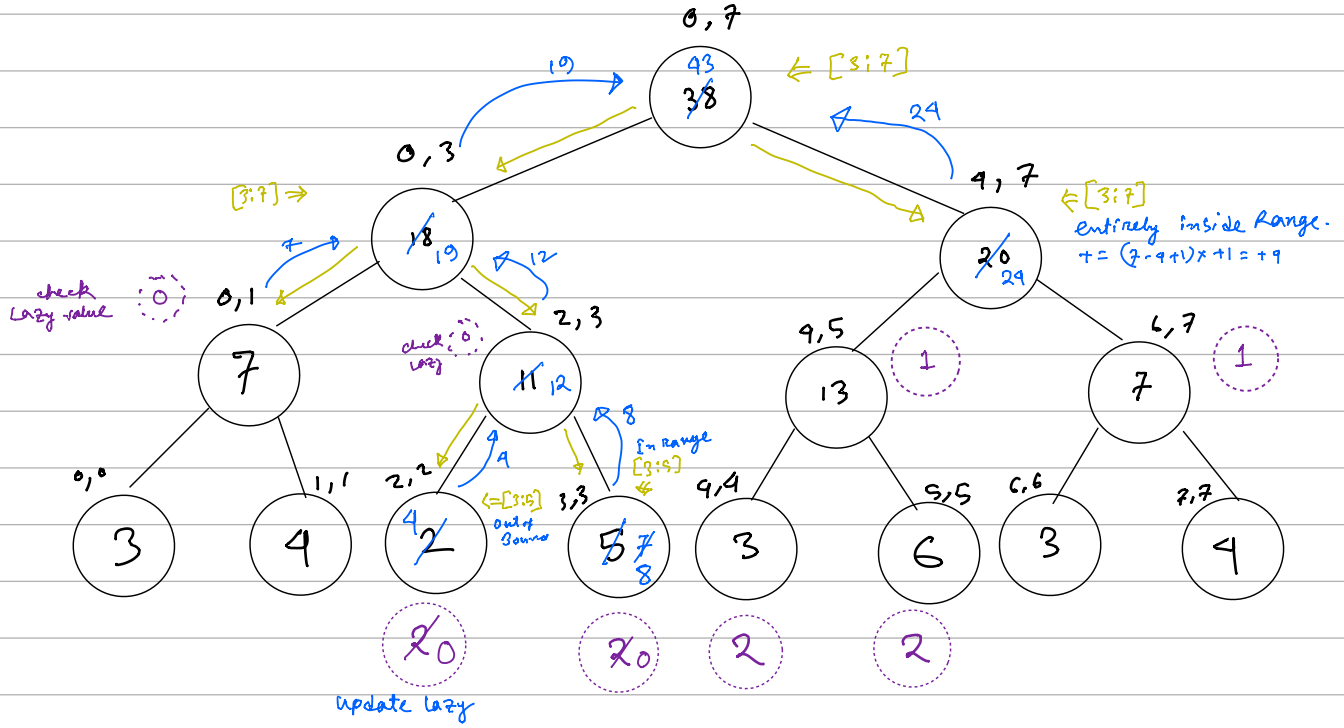
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	~~0~~ 2	~~0~~ 2	~~0~~ 2	~~0~~ 2	0	0	



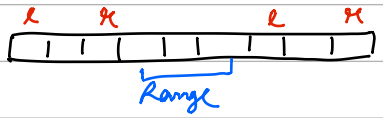
nums: 3, 4, 4, 7, 5, 8, 3, 9 (0 1 2 3 4 5 6 7) update range[3:7] add +1

segTree: 38 | 18 | 20 | 7 | 11 | 13 | 7 | 3 | 4 | 2 | 5 | 3 | 6 | 3 | 4 | (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14)
43 10 12 9

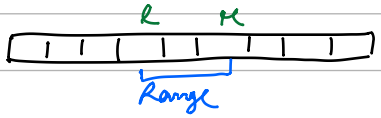
lazy: 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 0 | 0 |
1 1 0 0



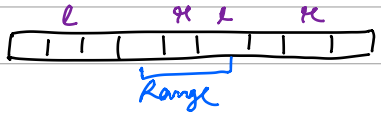
→ First update segTree[i] += lazy[i] & propagate lazy[i] to children if exist.

① 

r < start || l > end
↳ Return

② 

l >= start && r <= end
↳ segTree[i] += val & propagate lazy to children.

③ 

updateRange(l, mid),
updateRange(mid+1, r),
segTree[i] = segTree[2i+1] + segTree[2i+2]

```
void updateRange(start, end, i, l, r, val, segTree, Lazy)
{
    if(Lazy[i] != 0){
        segTree[i] += (r-l+1) * lazy[i];
        if(l != r){ // if i is leaf node it doesnt have child.
            lazy[2i+1] += lazy[i];
            lazy[2i+2] += lazy[i]; } }
        lazy[i] = 0;

    if(l > end || r < start){ // completely out of Bound
        return; }

    if(l >= start && r <= end){ // Completely inside Range
        segTree[i] += (r-l+1) * val
        if(l != r){ // if i is leaf node it doesnt have child.
            lazy[2i+1] += lazy[i];
            lazy[2i+2] += lazy[i]; } }
        return;

    // overlaping case
    mid = (l+r)/2
    updateRange(start, end, 2i+1, l, mid, segTree, Lazy)
    updateRange(start, end, 2i+2, mid+1, r, segTree, Lazy)

    segTree[i] = segTree[2i+1] + segTree[2i+2];
}
```

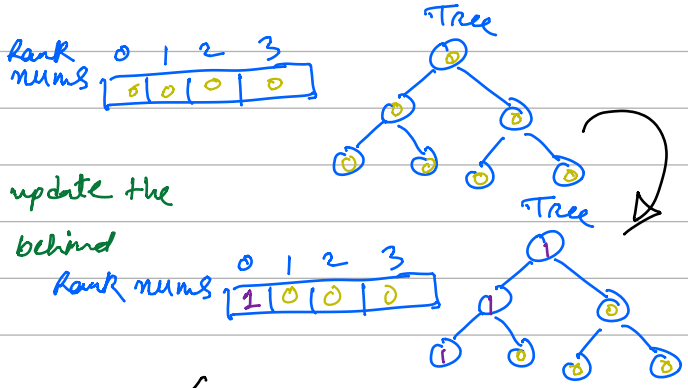
#LeetCode/Hard

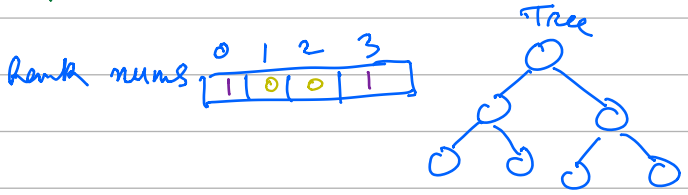
Count all Smaller after Self:

nums = [5, 2, 6, 1] ans = [2, 1, 1, 0]

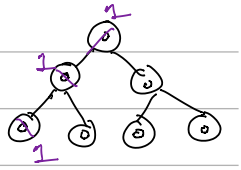
rank array of nums [5, 5, 2, 6, 1] is [2, 2, 1, 3, 0]

The tree represent normal range sum of the rank of nums array. Initially 0, the update the rank nums as we process the nums from behind and update the tree.



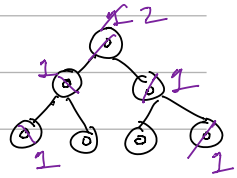


Process from end [5, 2, 6, 1] [2 1 3 0]

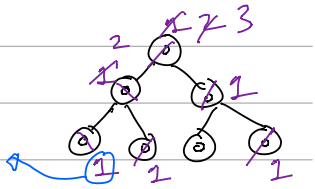


1) rangeQuery(start = 0, end = 0-1)
ans return = 0
- updateTree(idx = 0)

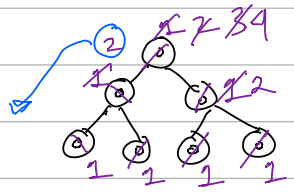
2) [5, 2, 6, 1] [2 1 3 0]
ans = rangeQuery(start=3, end=2)
↳ return 1
- updateTree(idx = 3)



3) [5, 2, 6, 1] [2 1 3 0]
ans = rangeQuery(start = 0, end=0)
↳ 1
updateTree(idx = 1)

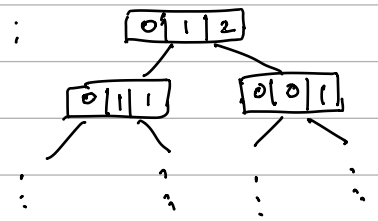


4) [5, 2, 6, 1] [2 1 3 0]
ans = rangeQuery(start = 0, end = 1)
↳ 2
updateTree(idx = 2)

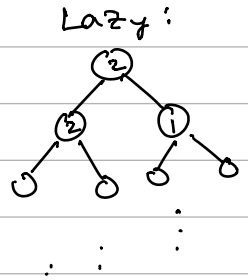


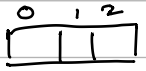
Q. queries on multiple of 3

Seg Tree :

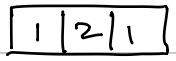


Lazy :

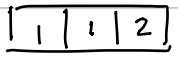


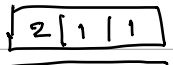


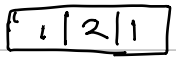
no. of element %3 = 0 or 1 or 2. ✓



numbers : 6, 4, 5, 7

add +1 7, 5, 6, 8 :  (Rotation) Lazy ①

add +1 8, 6, 7, 9 :  (Rotation) Lazy ②

add +1 9, 7, 8, 10 :  (Rotation) Lazy ⓪

Q. K increasing Sequences

count no. of increasing Subsequences of length K.

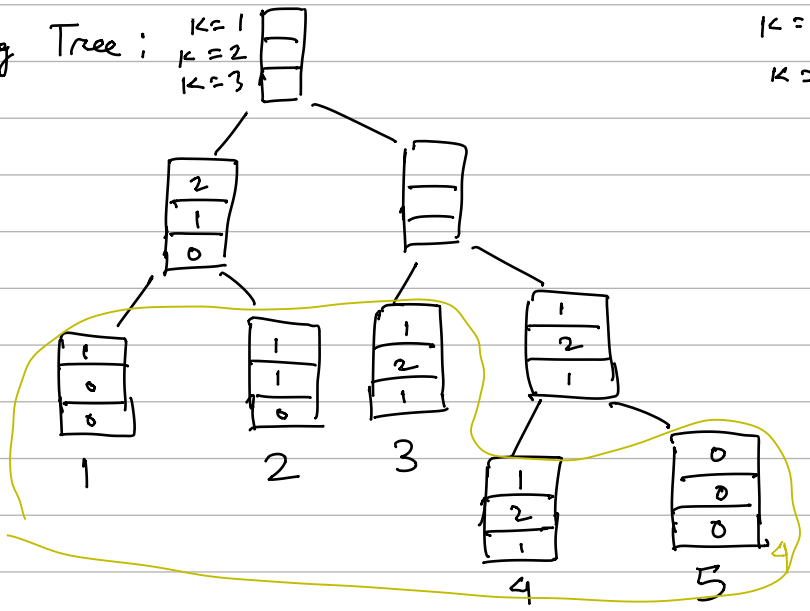
DP soln : TLE

nums : 1 2 4 3 5

K = 1	1	1	1	1	1
K = 2	0	1	2	2	4
K = 3	0	0	1	1	5

Ans for K = 3

Seg Tree : K=1 K=2 K=3



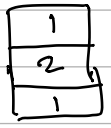
These nodes are in their natural sorted order.

update and Query is in order of input nums.

✓ update 1's value  update 2's value Query 1's value 

✓ update 4's value Query [0 to 3]

use this



✓ then update 3 use Query [0 to 2]