Segment Tree

- The Segment Tree (or Tree intervals) is a data structure that allows us to store information in the form of intervals, or segments. It can be used for making update/query operations upon array intervals in logarithmical time.

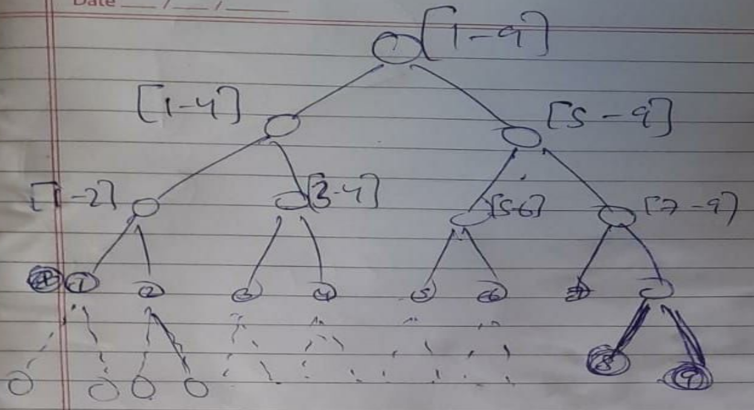
- Segment tree for the interval [i, j] in the following manner:
  - The first node will hold the information for the interval [i, j].
  - If i<j the left and right son will hold the information for the intervals **[i, (i+j)/2] and [(i+j)/2+1, j].**

- A segment trees has only three operations:
    - Construct,
    - Update,
    - Query.

- **Construct tree:** To init the tree segments or intervals values.
- **Update tree:** To update value of an interval or segment.
- **Query tree:** To retrieve the value of an interval or segment.

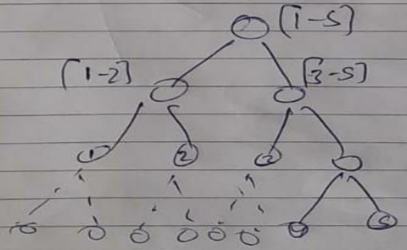
- Segment tree is a strict binary tree. All levels are filled except possibly the last level. Unlike Heap, the last level may have gaps between nodes.
- Memory required in to build segment tree over N size array is O(4*N).
-

Date ___ / ___ / ___



for this case I needed 31 indices to store 9 nodes.

for values of form (2^n+1) worst case occurs



we need 25 indices to store 5 values

diff. more visible for larger values

Code - Build / construct

```
void build(ll a[],ll tree[],ll start,ll end,ll node)
{
    if(start==end)
    {
        tree[node] = a[start];
        return;
    }
    ll mid = (start+end)/2;
    build(a,tree,start,mid,2*node);
    build(a,tree,mid+1,end,2*node+1);
    tree[node] = tree[2*node] + tree[2*node+1];
}
```

function call -> build(a,tree,0,n-1,1);

Code - Query

```
ll query(ll tree[],ll l,ll r,ll start,ll end,ll node)
{
    if(start>r||end<l) return 0;
    if(start>=l&&end<=r) return tree[node];
    ll mid = (start+end)/2;
    ll ans1 = query(tree,l,r,start,mid,2*node);
    ll ans2 = query(tree,l,r,mid+1,end,2*node+1);
    return ans1+ans2;
}
```

function call -> query(a,tree,l,r,0,n-1,1);

Code - Update

```
void update(ll a[],ll tree[],ll node,ll start,ll end,ll idx,ll value)
{
    if(start==end)
    {
        a[idx]=value;
        tree[node]=value;
        return ;
    }
    ll mid = (start+end)/2;
    if(idx<=mid&&start<=idx) update(a,tree,2*node,start,mid,idx,value);
    else update(a,tree,2*node+1,mid+1,end,idx,value);
    tree[node] = tree[2*node] + tree[2*node+1];
}
```

function call -> update(a,tree,1,0,n-1,idx,value);

Question - Bhaiya Ka Safar

1 <= MAXX <= 12 -> total numbers to be selected
1 <= N <= 10^5 -> length of array
1 <= A[i] <= 10^5 -> values of array

Sub-problem - For the given sequence with n different elements find the number of increasing subsequences with MAXX elements.
Link

Why lazy propagation?

Sometimes problems will ask you to update an interval from l to r, instead of a single element. One solution is to update all the elements one by one. Complexity of this approach will be O(NlogN) per operation since where are N elements in the array and updating a single element will take O(logN) time.

To avoid multiple call to update function, we can modify the update function to work on an interval using lazy propagation.

What is lazy propagation

Do work only when needed. How ?

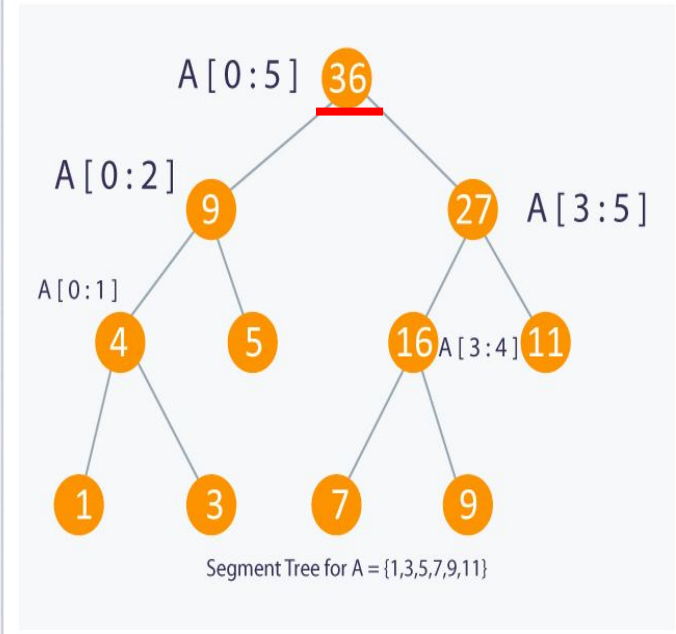
When we need to update an interval, we will update a node and mark its child that it needs to be updated and update it when needed. For this we need an array lazy[] of the same size as that of segment tree. Initially all the elements of the lazy[] array will be 0 representing that there is no pending update. If there is non-zero element lazy[k] then this element needs to update node k in the segment tree before making any query operation.

To update an interval we will keep 3 things in mind.

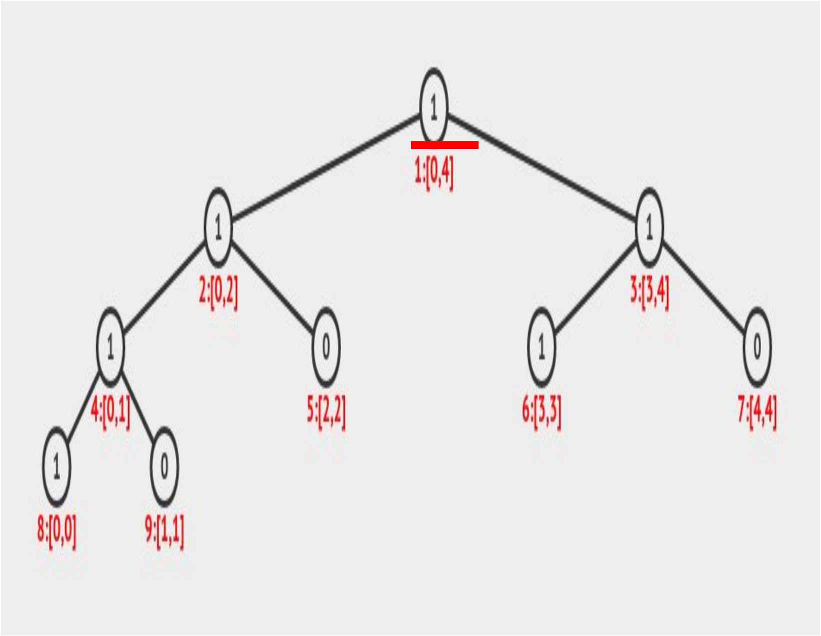
1. If current segment tree node has any pending update, then first add that pending update to current node.
2. If the interval represented by current node lies completely in the interval to update, then update the current node and update the lazy[] array for children nodes.
3. If the interval represented by current node overlaps with the interval to update, then update the nodes as the earlier update function

increase interval [0:2] by 2

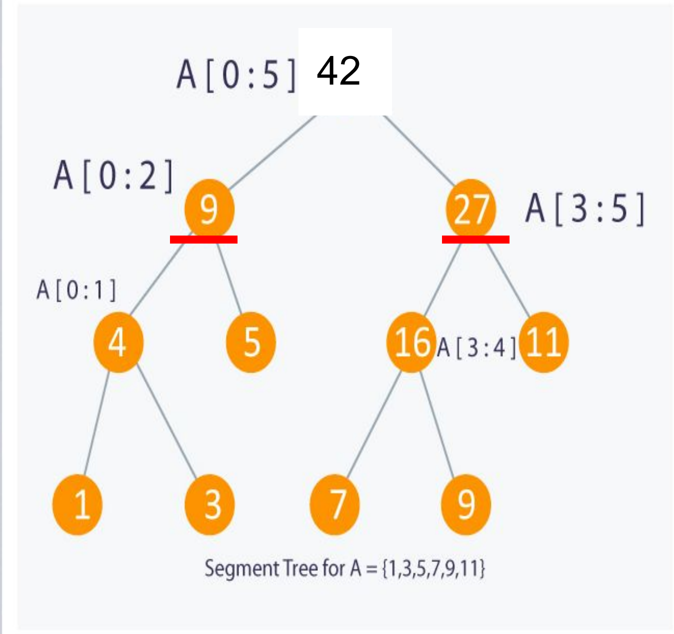
SEGMENT TREE



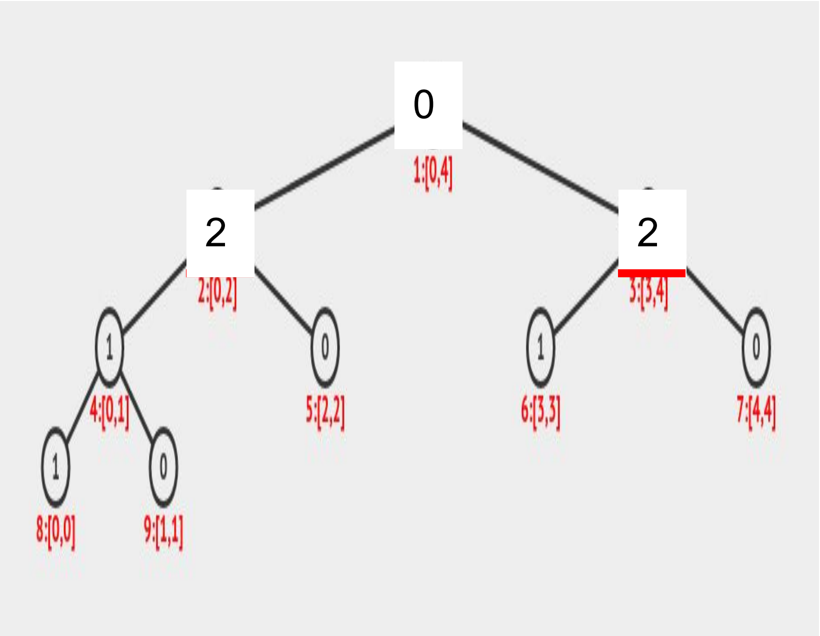
LAZY TREE



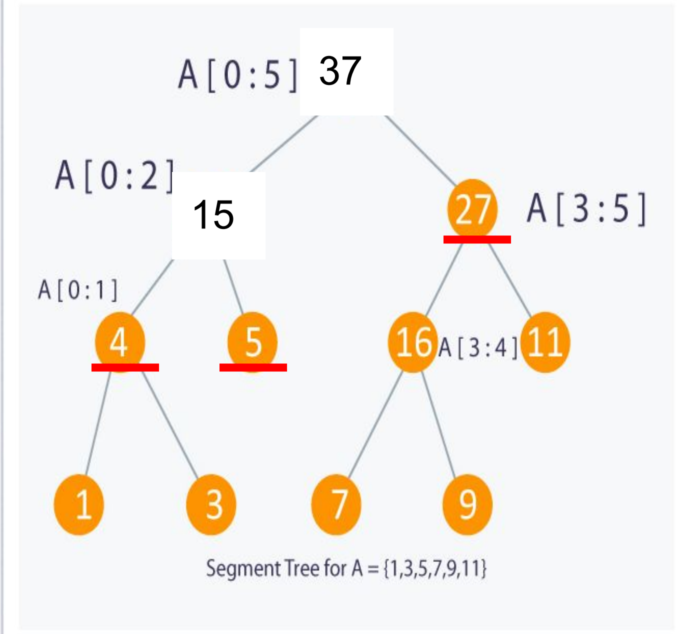
SEGMENT TREE



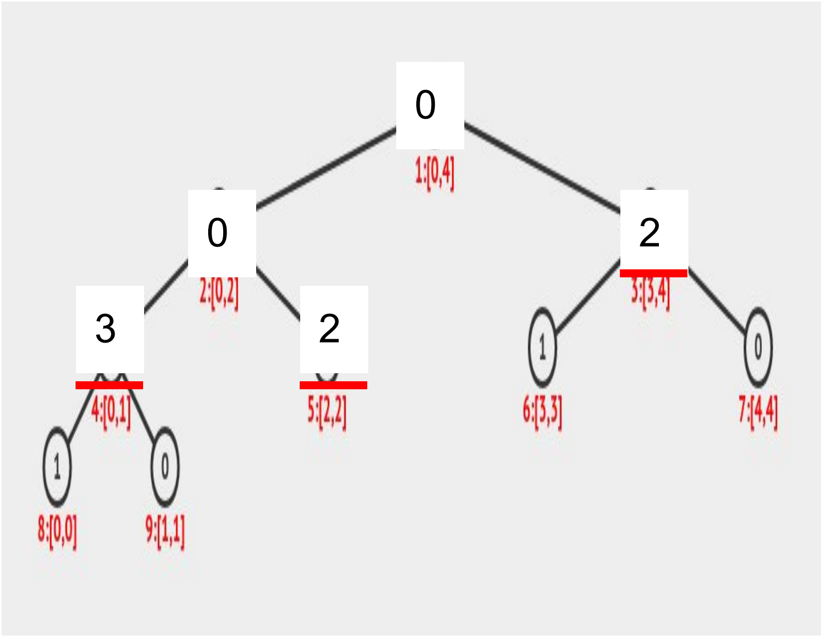
LAZY TREE



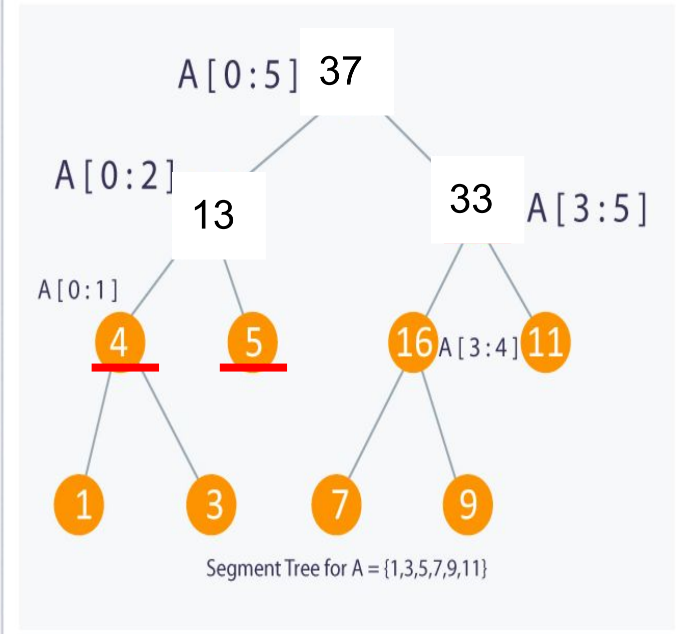
SEGMENT TREE



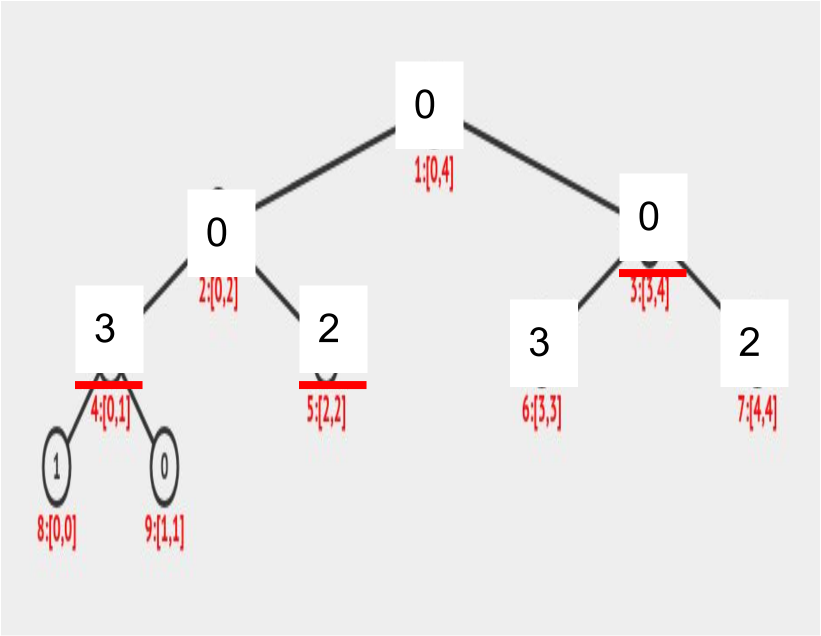
LAZY TREE



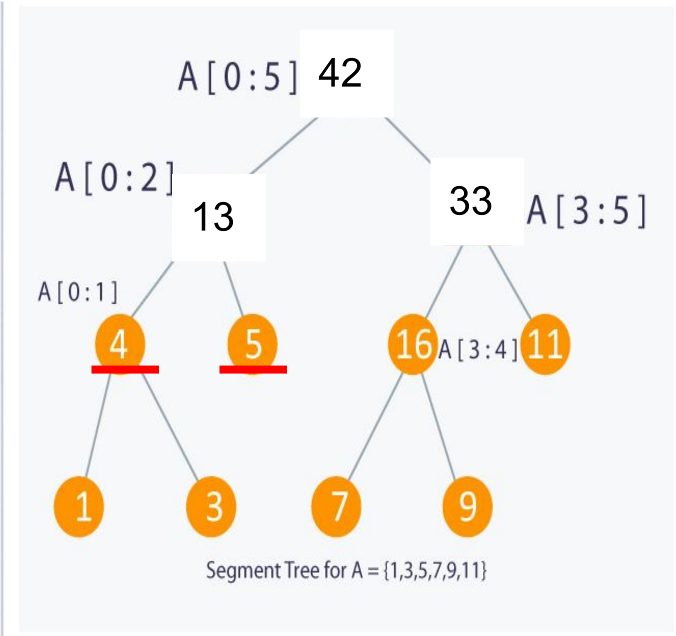
SEGMENT TREE



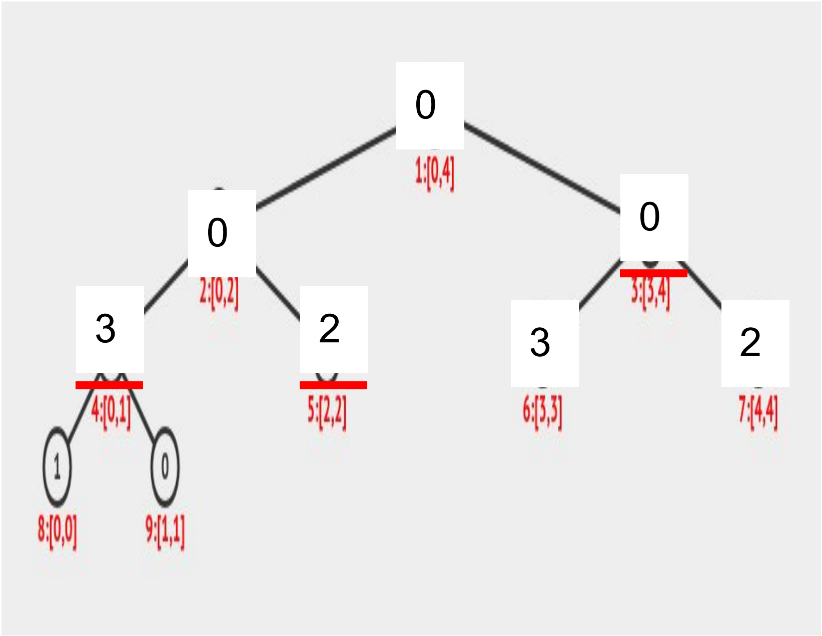
LAZY TREE



SEGMENT TREE



LAZY TREE



code - update

```
 void updateRange(int node, int start, int end, int l, int r, int val)
{
    if(lazy[node] != 0)
    {
        // This node needs to be updated
        tree[node] += (end - start + 1) * lazy[node];    // Update it
        if(start != end)
        {
            lazy[node*2] += lazy[node];                  // Mark child as
lazy
            lazy[node*2+1] += lazy[node];                // Mark child as
lazy
        }
        lazy[node] = 0;                                  // Reset it
    }
    if(start > end or start > r or end < l)              // Current segment
is not within range [l, r]
        return;
    if(start >= l and end <= r)
    {
        // Segment is fully within range
        tree[node] += (end - start + 1) * val;
        if(start != end)
        {
            // Not leaf node
            lazy[node*2] += val;
            lazy[node*2+1] += val;
        }
        return;
    }
    int mid = (start + end) / 2;
    updateRange(node*2, start, mid, l, r, val);        // Updating left
child
    updateRange(node*2 + 1, mid + 1, end, l, r, val);   // Updating right
child
    tree[node] = tree[node*2] + tree[node*2+1];        // Updating root
with max value
}
```

Time Complexity = O(logN)

code - query

```
int queryRange(int node, int start, int end, int l, int r)
{
    if(start > end or start > r or end < l)
        return 0;         // Out of range
    if(lazy[node] != 0)
    {
        // This node needs to be updated
        tree[node] += (end - start + 1) * lazy[node];            // Update
it
        if(start != end)
        {
            lazy[node*2] += lazy[node];         // Mark child as lazy
            lazy[node*2+1] += lazy[node];    // Mark child as lazy
        }
        lazy[node] = 0;                 // Reset it
    }
    if(start >= l and end <= r)             // Current segment is totally
within range [l, r]
        return tree[node];
    int mid = (start + end) / 2;
    int p1 = queryRange(node*2, start, mid, l, r);         // Query left
child
    int p2 = queryRange(node*2 + 1, mid + 1, end, l, r); // Query right
child
    return (p1 + p2);
}
```

Time Complexity = O(logN)