Segment Tree

Introduction

Suppose we have a problem in which two operations are performed:

1. **Range Query** (**finding sum of elements in a range**).
2. **Update Array element(s)**

For example, in array = [1, 3, 5, 7, 6, 8] we have to perform the following operations:

Operation	Brute Force Time Complexity of Operation
1. Sum (array[0] to array[4])	O(N)
2. array[2] = 6	O(1)
3. array[3] = 9	O(1)
4. Sum (array[1] to array[4])	O(N)

Now, if we have **K** number of queries and **L** number of update operations then the overall time complexity of the solution becomes **O(K*N) + O(L*1)**. This increases with respect to **K**. Therefore, this approach is only effective when **K** is small.

Another approach to solve this is by maintaining another array in which the i^{th} element will store the sum from 0^{th} to the i^{th} element in our original array.

For example, ArrSum[2] = Sum(array[0] to array[2]).

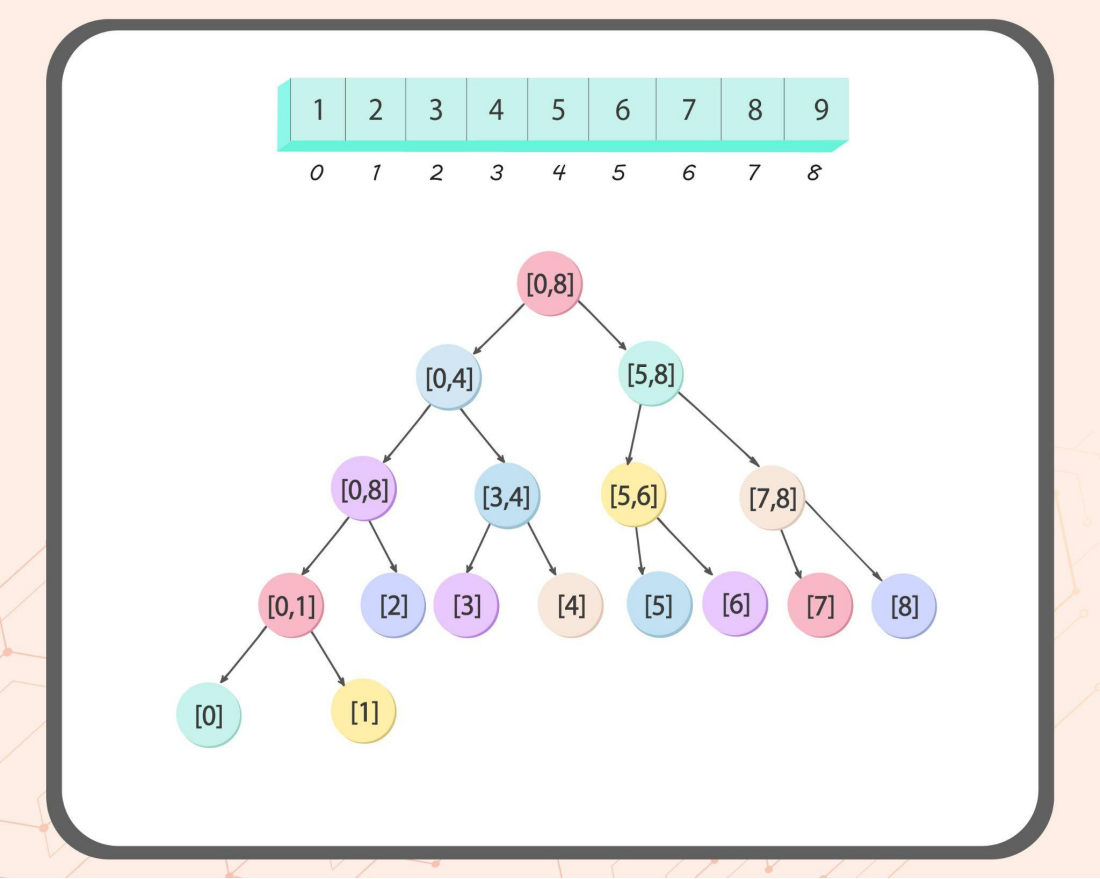
Array = [1, 3, 5, 7, 6, 8]
ArrSum = [1, 4, 9, 16, 22, 30]

Sum(array[2] to array[4]) = Sum(array[0] to array[4]) – Sum(array[0] to array[1])

= ArrSum[4] – ArrSum[1]

Therefore, the time complexity of solving the query reduces to **O(1)** but for update operations, in the worst case we have to traverse the entire "**ArrSum**". Hence, the time complexity of update operation increases to **O(N)**.

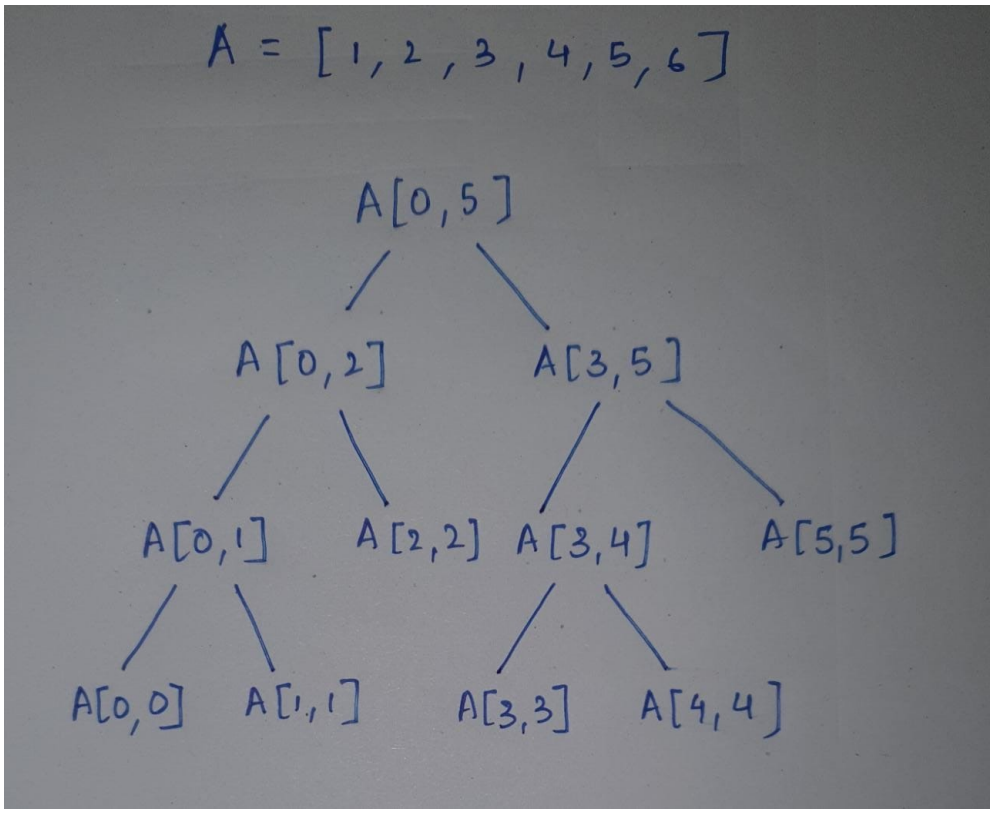
Segment Tree helps us to perform both Query and Update Operations in **O(log(N))** time.



In the Segment Tree, most of the solutions to queries are already present at its nodes, for those queries whose solutions are not directly present like Sum(array[5] to array[7]), we can calculate it by using the [5, 8] and [8] nodes.

Even for the worst-case scenario, that is, traversing through the entire tree, the time complexity of the array will be equal to **O(log(N))** which is equal to the height of the tree.

Building the Segment Tree



Building a Segment tree requires **O(N)** time because there are **N** leaf nodes in a segment tree and the remaining nodes are equal to **N – 1**. Therefore, the total number of nodes = **(N) + (N – 1)** = **2N – 1**.

```
int main () {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

    // segment tree is given the size of 2*size of arr
```

```
    int *tree = new int[18] ;

    buildTree(arr, tree, 0, 8, 1);
}
```

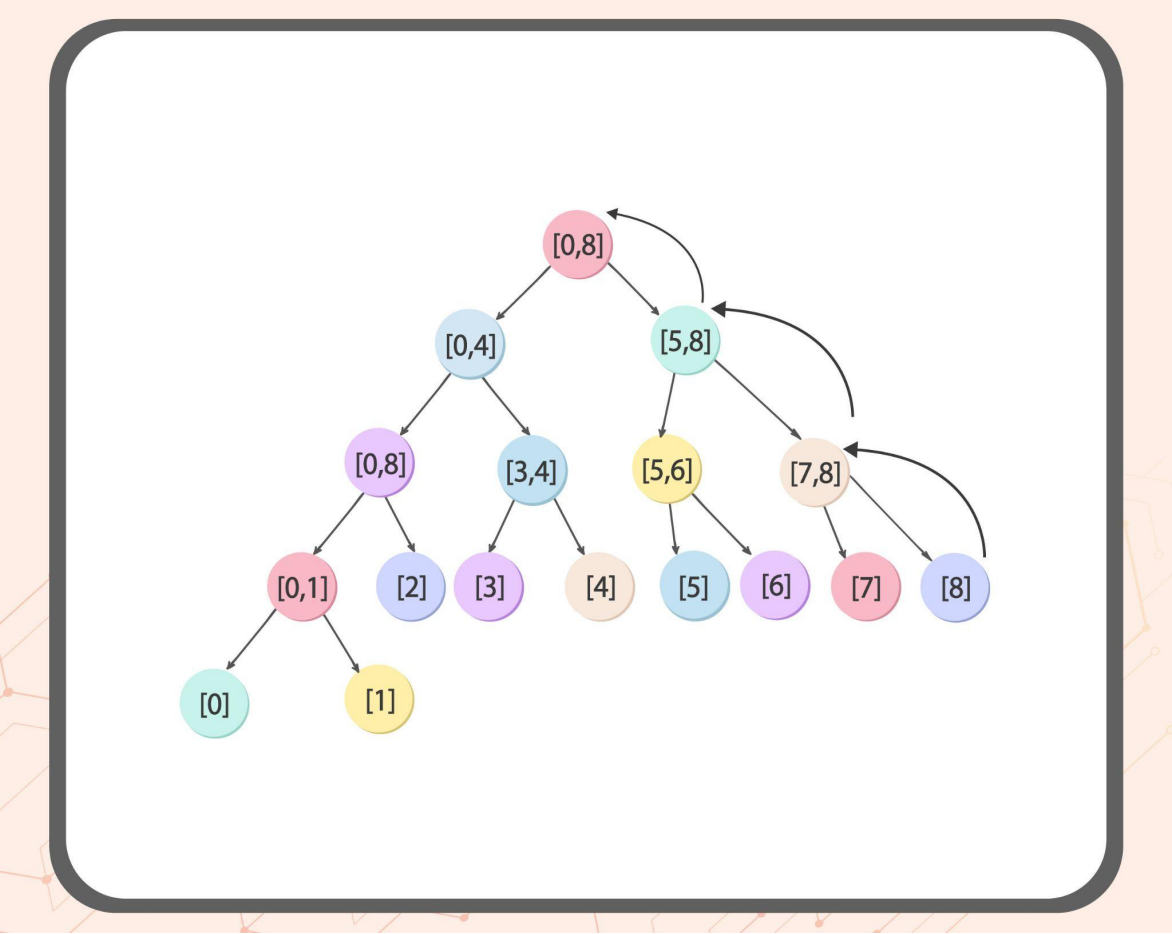
```
void buildTree(int* arr, int* tree, int start, int end, int treeNode){

    // BASE CASE
    if(start == end){
        tree[treeNode] = arr[start];
    }

    int mid =(start+end)/2;
    // two recursive calls to build the left half and
    // right half respectively
    buildTree(arr, tree, start, mid, 2*treeNode);
    buildTree(arr, tree, mid+!, end, 2*treeNode+1);
    // the answer to the recursive calls gets stored in
    // the treeNode
    tree[treeNOde] = tree[2*treeNode] + tree[2*treeNode + 1]
}
```

The size of the segment tree is twice the size of our initial array and the indexing starts from 1 instead of 0.

Here there are two recursive calls happening for every ***treeNode*** and the sum of their answers gets stored inside the ***treeNode***.

Updating the Segment Tree

First we find mid, then decide either to go left or right, that is, if the index where we have to update the value is greater than mid then we go right, else we go left.
Even if the values at other nodes get affected by updating a single node, we only travel the whole height of the tree in the worst case scenario, hence, the time complexity for update operation is **O(log(N))**.

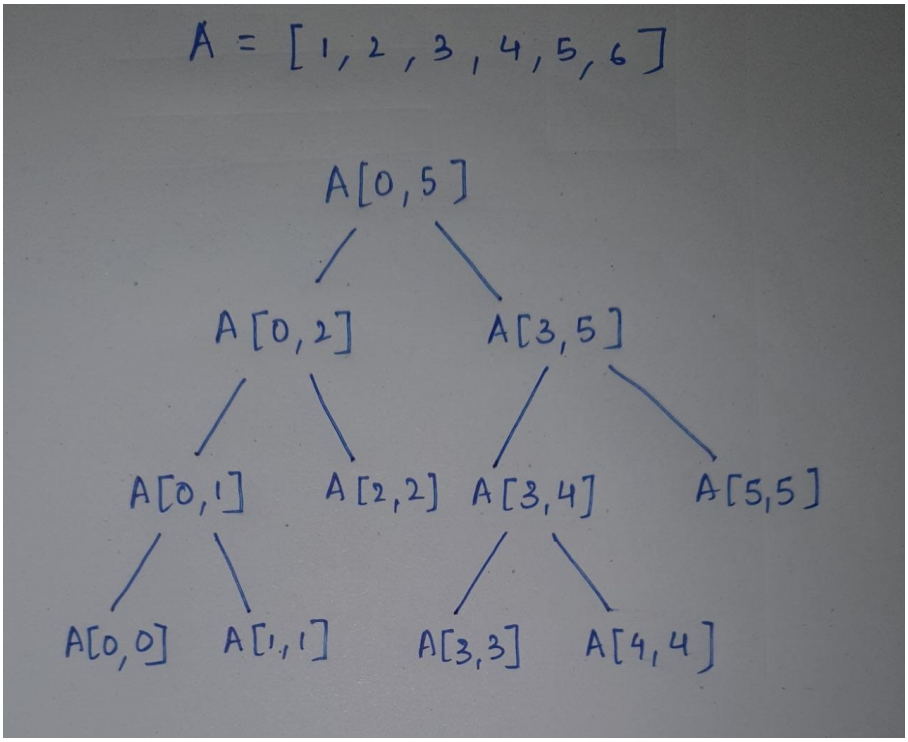


Example: For the update function at [8] or 9th index of the segment tree, only the values at 4 nodes need to be changed.

```
void updateTree(int* arr, int* tree, int start, int end, int treeNode, int idx
, int value){

    int mid = (start+end)/2;
    if(idx > mid){
        updateTree(arr, tree, mid+1, end, 2*treeNode+1, idx, value);
    } else {
        updateTree(arr, tree, start, mid, 2*treeNode, idx, value);
    }
    tree[treeNode] = tree[2*treeNode] + tree[2*treeNode+1];
}
```

Query on the Segment Tree

Example : A = [1, 2, 3, 4, 5, 6] **Query: Sum(2 to 4)**



- At A[0, 5] : the value of this node doesn't give us the answer to our query. Hence, we have to call recursion, now our query requires a sum

from 2 to 4th index so it is partially inside A[0, 2] and A[3, 5]. Therefore, a recursion call is made to both the left and right subtree.

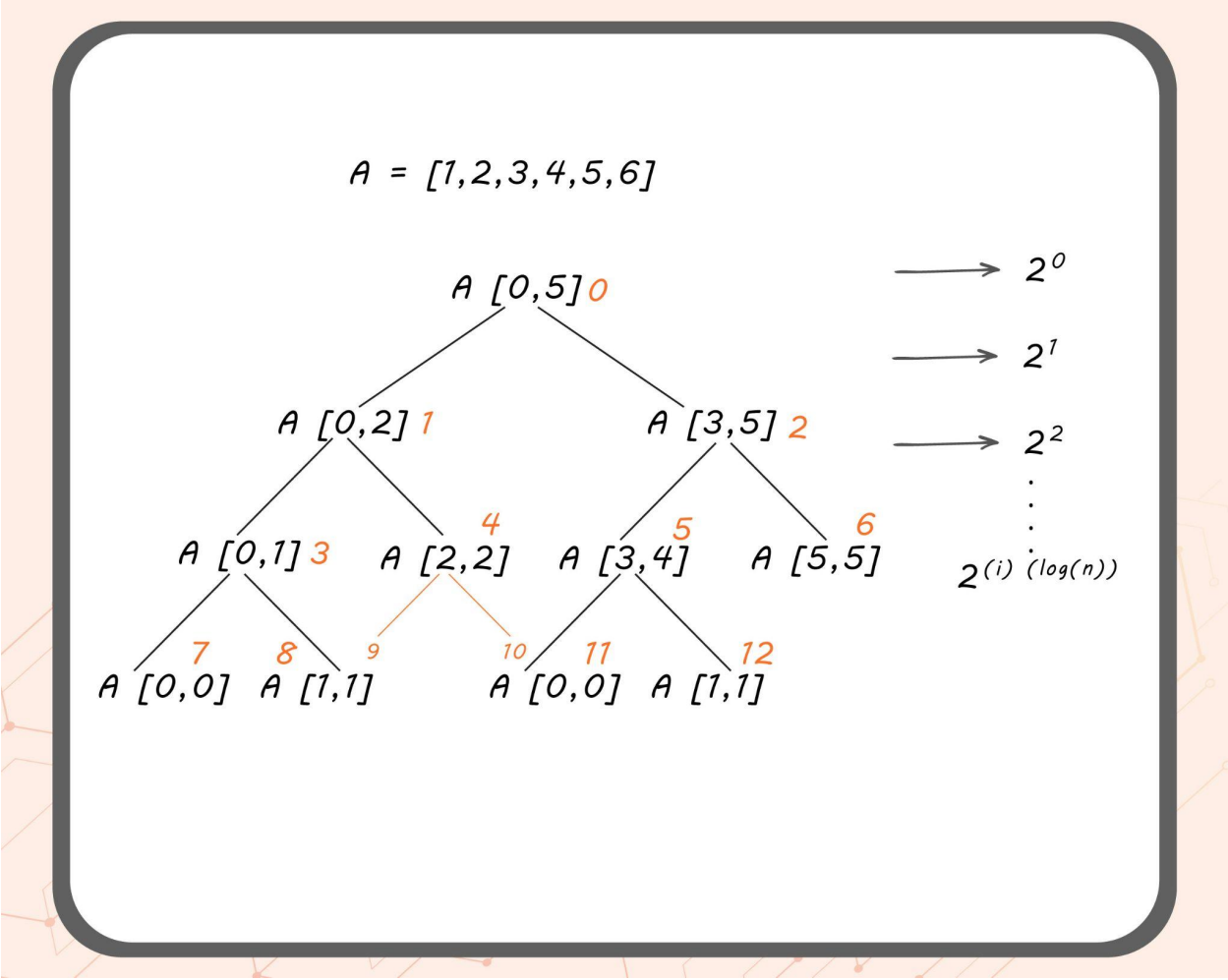
- At A[0, 2] : recursion call to A[0, 1] is not made because 0 and 1 are completely outside the range of our query and the answer of A[2, 2] which is completely inside the range of our query is returned using a recursive call.

- At A[3, 5] : recursion call to A[5, 5] is not made because 5 lies completely outside the range of our query while the answer to A[3, 4] which is completely inside the range of our query is returned using a recursive call.

```
int query(int *tree, int start, int end, int treeNode, int left, int right){
    // completely outside the given range
    if(start > right || end < left){
        return 0;
    }
    // completely inside the given range
    if(start >= left && end <= right){
        return tree[treeNode];
    }
    // Partially inside and partially outside
    int mid = (start+end)/2;
    int ans1 = query(tree, start, mid, 2*treeNode, left, right);
    int ans2 = query(tree, mid+1, end, 2*treeNode+1, left, right);

    return ans1 + ans2;
}
```

Size of the Segment Tree

Earlier we made the segment tree to be of size **2N,** because we were storing **2N** nodes. This doesn't always work because like in the figure shown below A[3, 3] and A[4, 4] are supposed to be stored at the 11th and 12^{th} index rather than the 9th and 10^{th} index.



- Total number of nodes = $2^0 + 2^1 + 2^2 + ... + 2^{ceil\ (log(N))a}$

$$= (1)(2^{ceil\ (log(N))\ +1} -1)\ /(2-1)$$

= $2*2^{\text{ceil (log(N))}}$

= **4N** (approx)

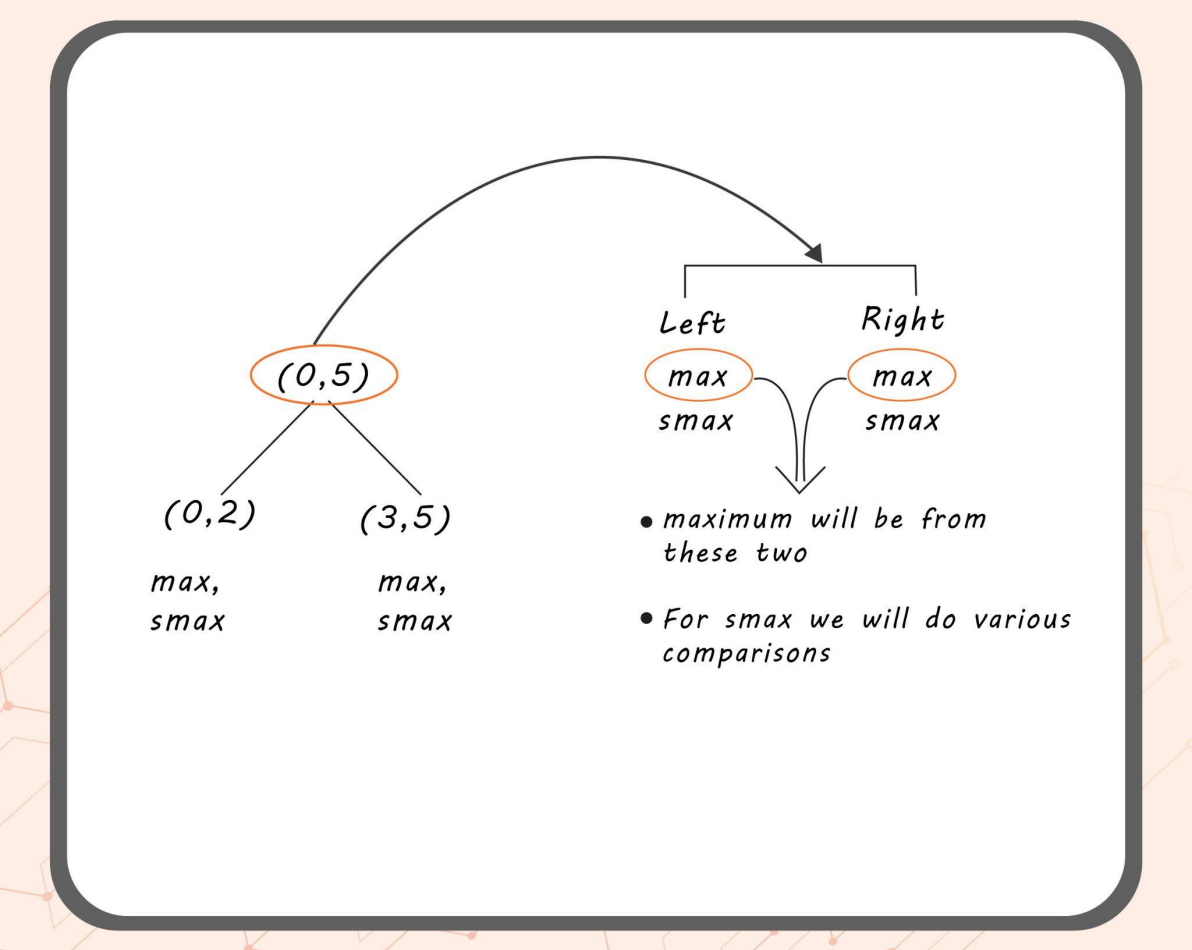
Maximum Pair Sum

Problem Statement: Suppose we are given an array = [2, 3, 1, 5, 7, 6] and we have to find a pair that gives us the maximum sum.

Explanation:

Basic Approach: If we need a pair that has the maximum sum then that clearly means the elements in the pair are maximum and second maximum in the array.
Therefore, we can sort this array or use a two pointer approach to solve this question.

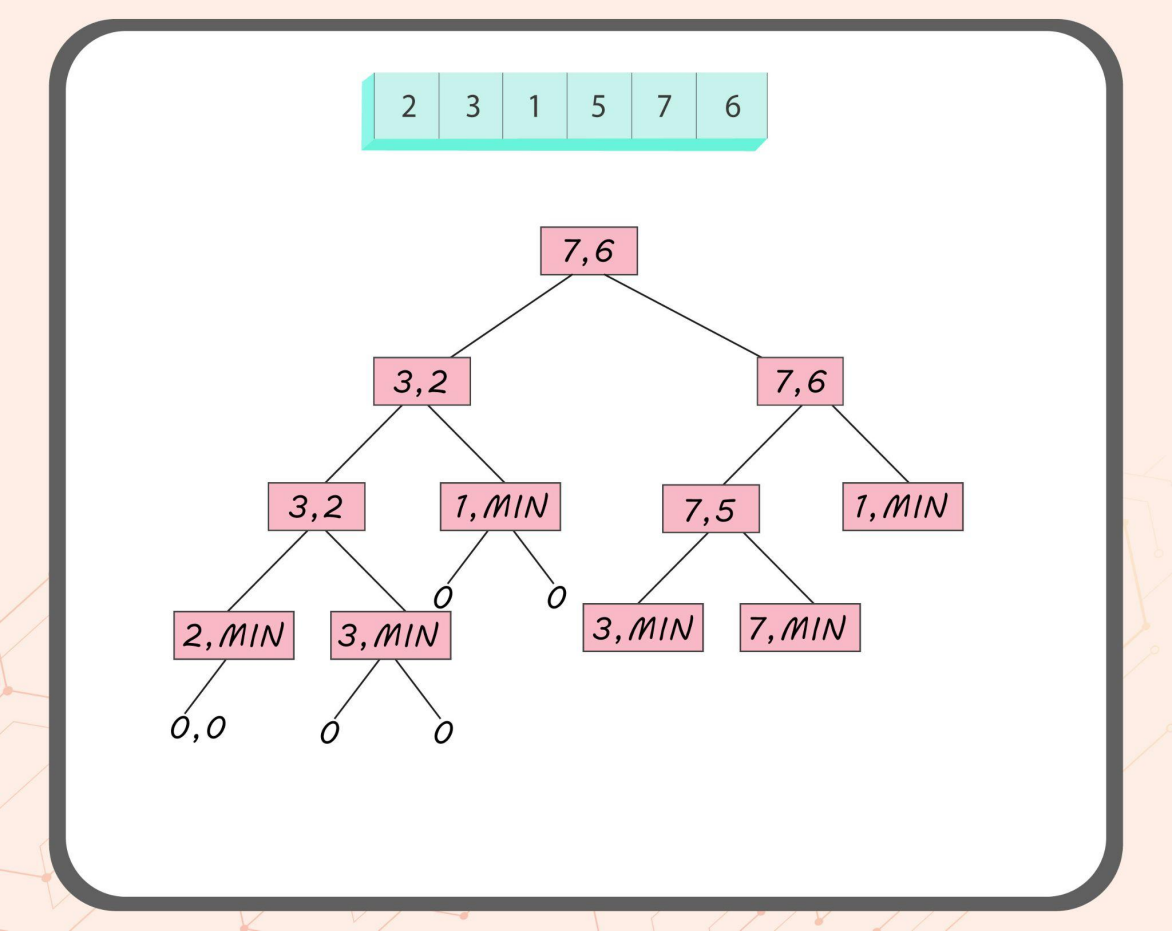
Segment Tree Approach: We can use the segment tree in the above problem by calculating the maximum(**max**) and second maximum(**smax**) at each node of the segment tree.



For the node that stores the answer of (0, 5), we have one left node and one right node and we have the max and smax from both left and right nodes.

Now, the max of their parent node will be from the max of the left or max of the right node.
If the right node's max is chosen then smax of the parent node will be the max of the left node or smax of the right node.
And vice versa.

The tree after the build will look like this:



The code is pretty similar to what we have done until now so try to do it on your own.

Lazy Propagation

Earlier we used to update the value on a single index but we use lazy propagation when we have to update in a range of indexes.

When there are many updates and updates are done on a range, we can postpone some updates (avoid recursive calls in update) and do those updates only when required.

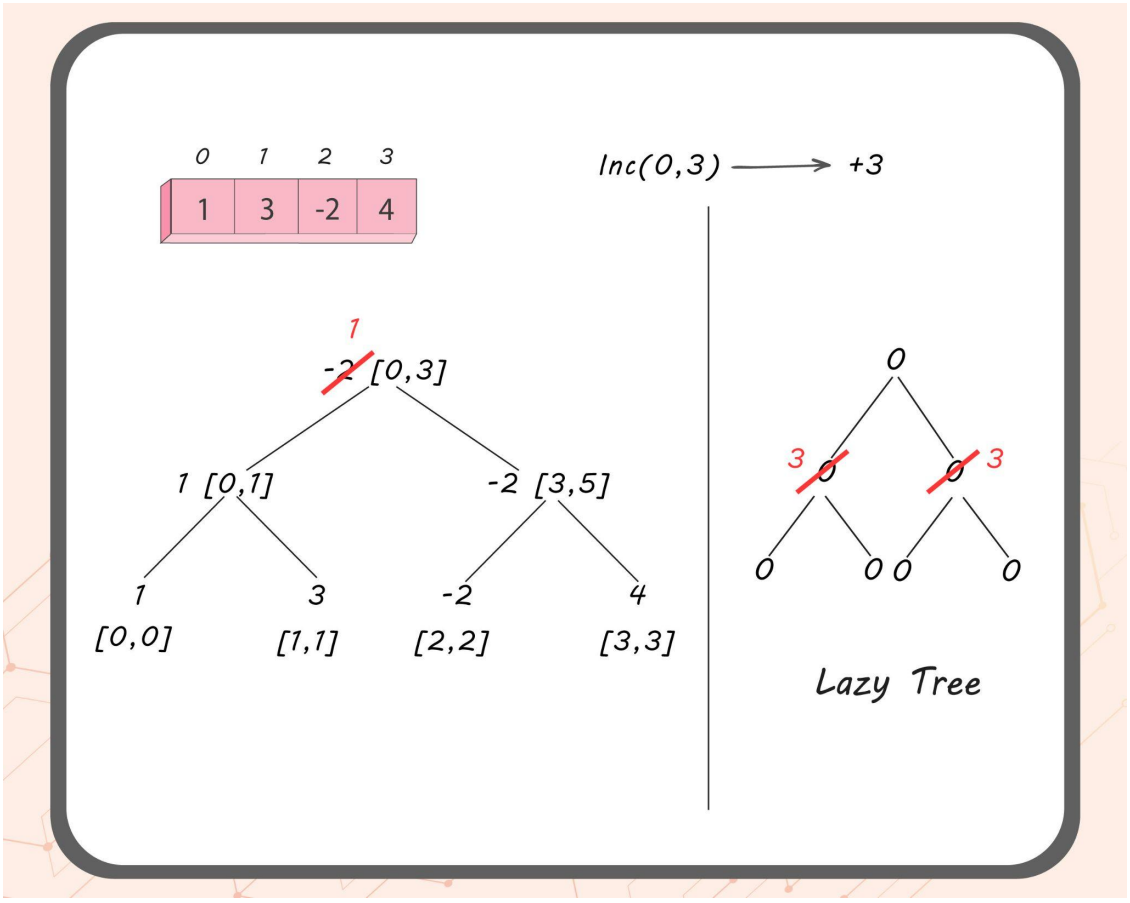
We make a lazy tree with the same structure as our segment tree. A value of 0 at any node represents that there are no pending updates on that node. A non-zero value represents the amount to be added to the node in the segment tree before making any query to the node.

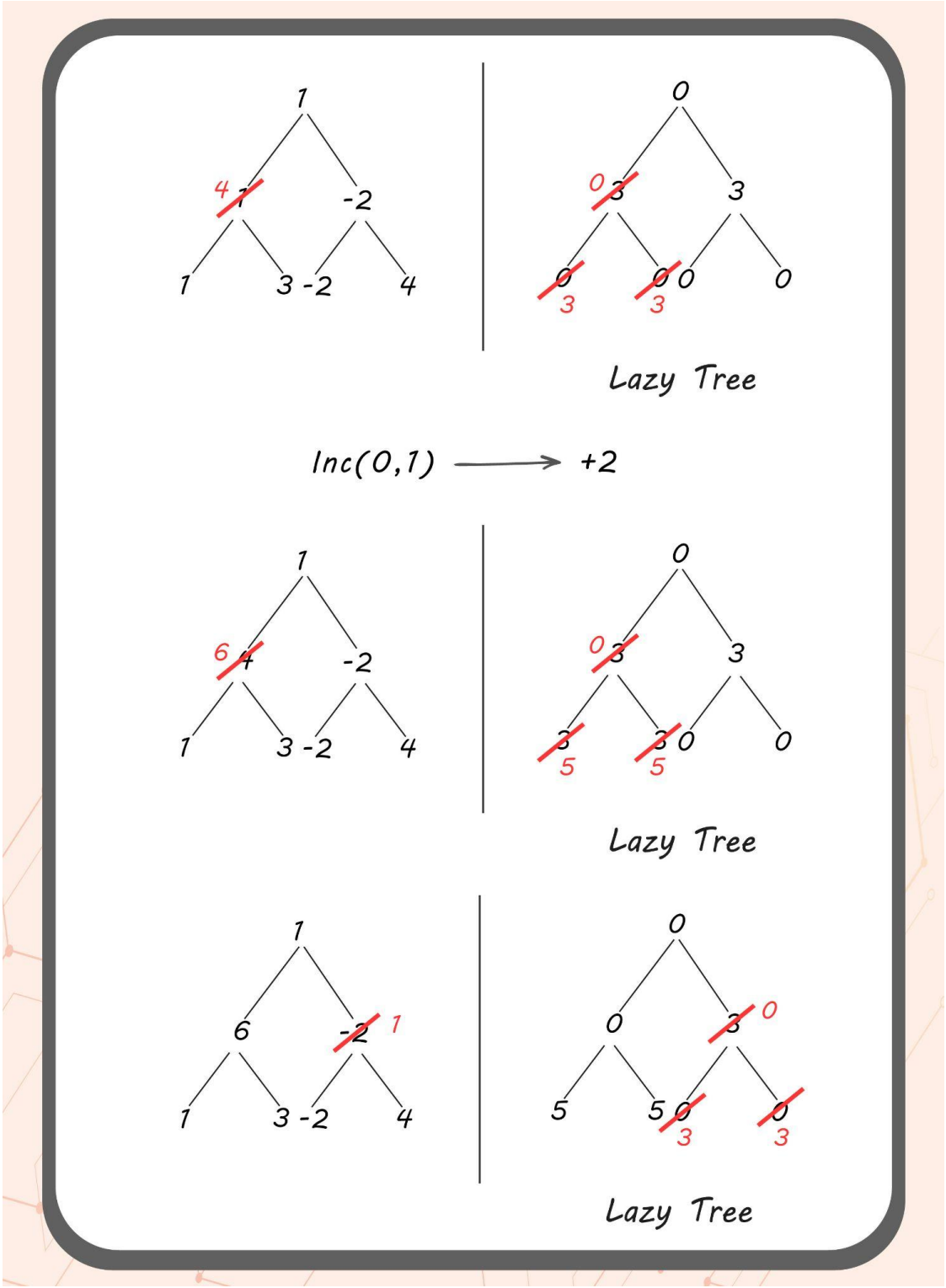
Algorithm:

// To update segment tree for change in array
// values at array indexes from l to r.

updateRange(l, r)

1. If the current segment tree node has any pending update, then first add that pending update to the current node.
2. If the current node's range lies completely in the update query range:
    a. Update current node
    b. Postpone updates to children by setting lazy values for children nodes.
3. If the current node's range overlaps with the update range, follow the same approach as above.
    a. Recur for left and right children.
    b. Update current node using results of left and right calls.


0 1 2 3
1 3 -2 4
Inc(0,3) → +3
1
-2 [0,3]
1 [0,1]
-2 [3,5]
1
[0,0]
3
[1,1]
-2
[2,2]
4
[3,3]
0
3 0
0 3
0 0 0 0
Lazy Tree


Lazy Tree
Inc(0,1) → +2
Lazy Tree
Lazy Tree

Code :

```
#include<bits/stdc++.h>
using namespace std;

void buildTree(int* arr,int* tree,int start,int end,int treeNode){

    if(start == end){
        tree[treeNode] = arr[start];
        return;
    }
    int mid = (start+end)/2;

    buildTree(arr,tree,start,mid,2*treeNode);
    buildTree(arr,tree,mid+1,end,2*treeNode+1);
    tree[treeNode] = min(tree[2*treeNode],tree[2*treeNode+1]);
}

void updateSegmentTreeLazy(int* tree,int* lazy,int low,int high,int
startR,int endR,int currPos,int inc){

    if(low > high){
        return;
    }

    if(lazy[currPos] !=0){
        tree[currPos] += lazy[currPos];

        if(low!=high){
            lazy[2*currPos] += lazy[currPos];
            lazy[2*currPos+1] += lazy[currPos];
        }
        lazy[currPos] = 0;
    }

    // No overlap
    if(startR > high || endR < low){
        return;
    }

    // Complete Overlap
```

```
    if(startR<= low && high <= endR){
        tree[currPos] += inc;
        if(low!=high){
            lazy[2*currPos] += inc;
            lazy[2*currPos+1] += inc;
        }
        return;
    }

    // Partial Overlap

    int mid = (low+high)/2;
    updateSegmentTreeLazy(tree,lazy,low,mid,startR,endR,2*currPos,inc);

updateSegmentTreeLazy(tree,lazy,mid+1,high,startR,endR,2*currPos+1,inc);
    tree[currPos] = min(tree[2*currPos],tree[2*currPos+1]);
}

int main(){
    int arr[] = {1,3,-2,4};
    int* tree = new int[12]();
    buildTree(arr,tree,0,3,1);
    int* lazy = new int[12]();
    updateSegmentTreeLazy(tree,lazy,0,3,0,3,1,3);
    updateSegmentTreeLazy(tree,lazy,0,3,0,1,1,2);

    cout<< "Segment Tree" <<endl;
    for(int i=1;i<12;i++){
        cout<<tree[i]<< endl;
    }

    cout<< "Lazy Tree" <<endl;
    for(int i=1;i<12;i++){
        cout<<lazy[i]<< endl;
    }
    return 0;
}
```