Segment Trees

Sum Queries on an Array

Given an array A of length N, there are Q queries. Each query specifies two integers l, r and requests the sum of all array elements whose indices lie in the range $[l, r]$.

We can easily solve this by computing the prefix sums of the array. The answer to a query l, r is just $\mathrm{prefixSum}[r] - \mathrm{prefixSum}[l-1]$.

Computing the prefix sums takes $O(N)$ time and answering each query takes $O(1)$ to answer. Total complexity is $O(N+Q)$.

Sum Queries with Updates on an Array

What if in between answering the queries, there are also U updates which specify two integers i, v and require us to add v to $\mathrm{A}[i]$?

If we still want to use prefix sums, then for each update we would have to rebuild the prefix sum which could take $O(N)$ time in the worst case. The complexity for this would be $O(N+Q+UN)$. We need a better solution. We'll try using something known as square root decomposition.

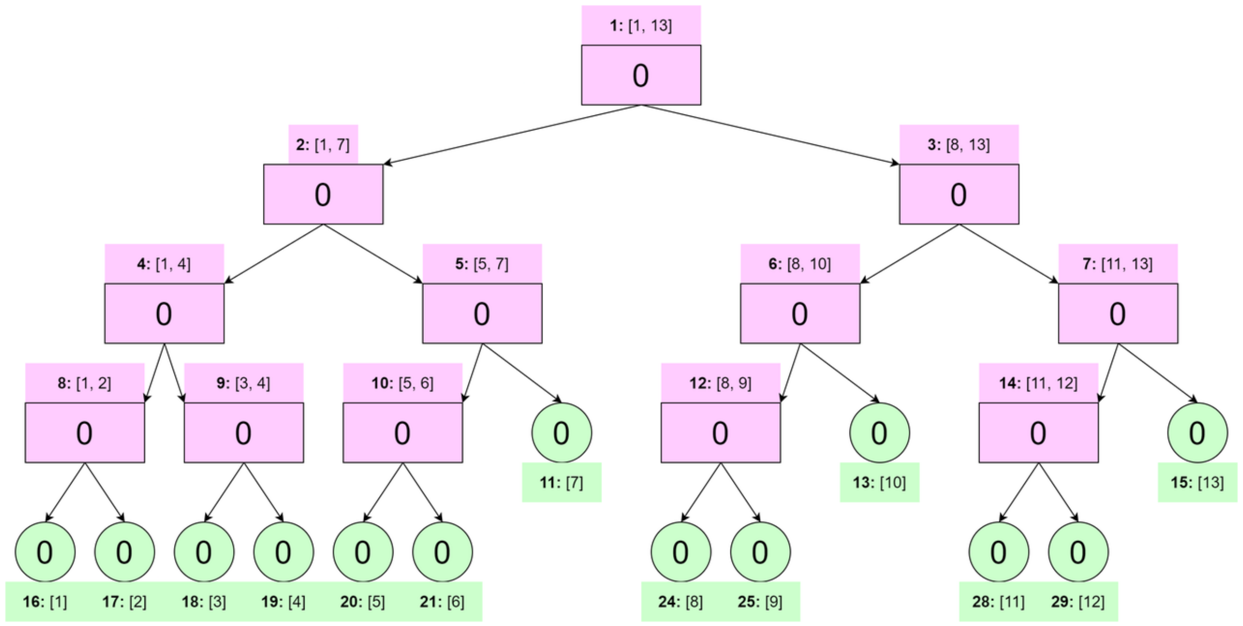
Split the array into $\sqrt{N}$ blocks of length $\sqrt{N}$ and compute prefix sums for each block separately. That is, treat each block as a separate array and compute prefix sums only for the elements in it, ignoring all other elements.

Now, if we have to update an element then, we only need to update the prefix sums of the block it is in. That would take $O(\sqrt{N})$ time. What about when querying? Given a range $[l, r]$ we can check the overlap of this range with each block and taking the sum of elements in the overlapping range will take only $O(1)$ time as we have computed the prefix sums already. In the worst case, we have to check this with all $\sqrt{N}$ blocks. The total worst case time for each query is then $O(\sqrt{N})$.

Our complexity is now $O(N+Q\sqrt{N}+U\sqrt{N})$.

Using Segment Trees

We can actually do better with a data structure called (here, at least) segment trees. This time let us try something different. Let's say $N=13$. Our array range is $[1,13]$. We'll maintain the sum of all the elements in the range. Now, split it into two: $[1,7],[8,13]$. We will also maintain the sum of elements for these two ranges. We keep splitting until we reach a point where the range covers just one index. Assume that initially $\forall i, \mathrm{A}[i]=0$. We will get something like this:

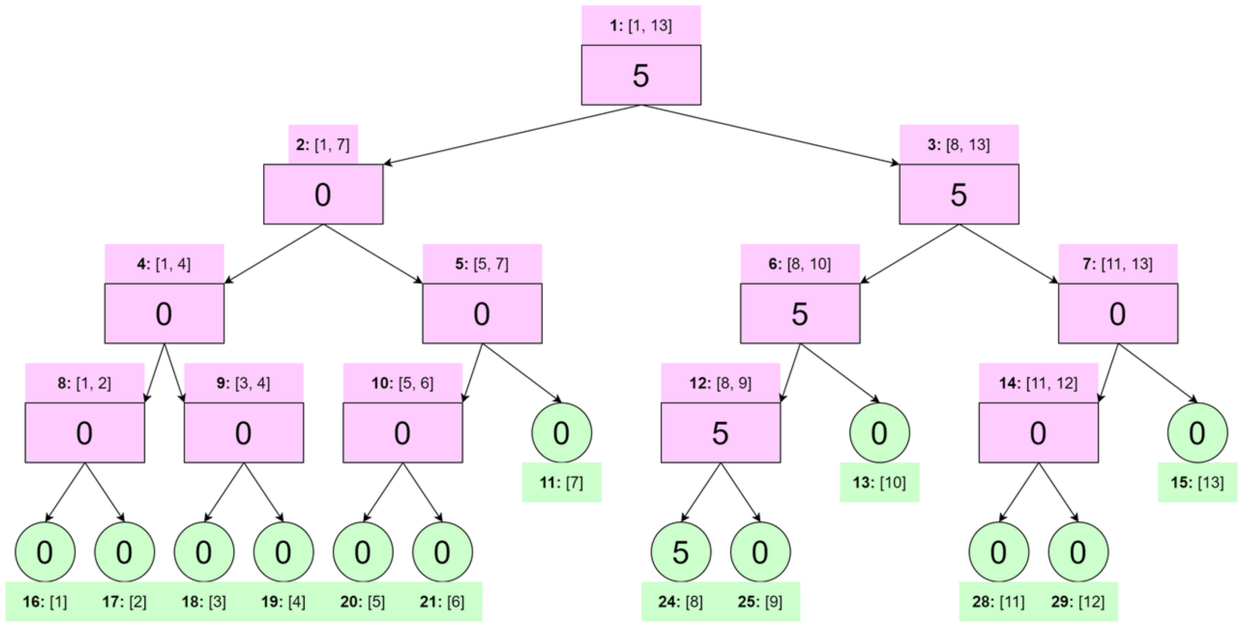


This is just like a binary tree and the numbers in bold followed by a colon represent the index of that node in a full binary tree. The numbers in square brackets that follow represent the range of indices covered by that node. The topmost node indexed 1 covers the entire range of the array $[1, 13]$. How do we build the tree above? The following simple recursive code demonstrates it:

```
// N*4 is a safe upper limit to choose for
// the number of nodes in the binary tree.

int Beg[N*4], End[N*4], Seg[N*4];

// Node i covers the range: [Beg[i], [End[i]]
// Seg[i] -> sum of elements in the range covered by the node

// Variable node covers the range [start, finish]
// Initially, to build the tree, we simply call
// buildTree(1, N, 1) if we want the segment tree
// to cover the range [1, N].
void buildTree(int start, int finish, int node=1)
{
    // Set the range of node
    Beg[node] = start;
    End[node] = finish;

    // Initially sum for the range
    Seg[node] = 0;

    // If this node covers only one element
    // we are done.
    if(start == finish)
    {
        return;
    }

    // Split the range
    int middle = (start+finish)/2;

    // Indices for the left and right nodes.
    int left = node*2;
    int right = left+1;

    // Recursively build subtrees.
    buildTree(start, middle, left);
    buildTree(middle+1, finish, right);
}
```

Building this tree takes as long as the number of nodes which is $O(N)$.

Now, let us perform some updates on this tree. First, let us add 5 to $\mathrm{A}[8]$. The tree will now look like:

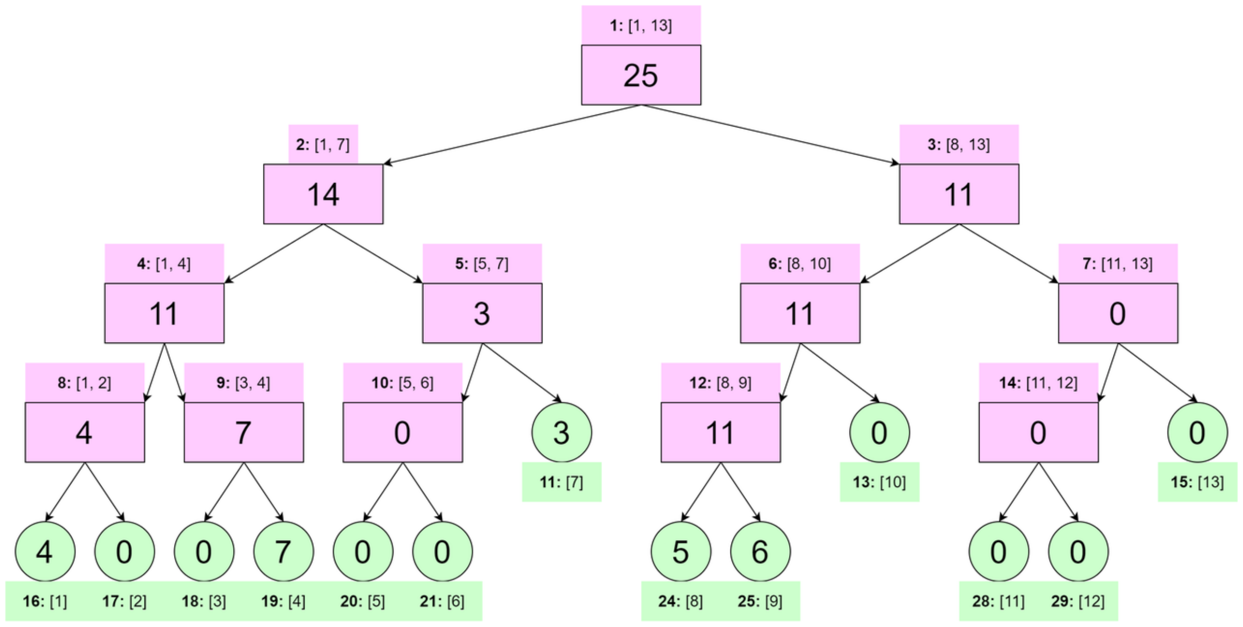


We just have to add 5 to all the nodes whose range contains 8. Clearly, the number of such nodes will not exceed the height of the tree which is $O(\log_2 N)$. We can therefore perform updates on this tree in $O(\log_2 N)$. We start at the top and make our way to the leaf node. The code for this:

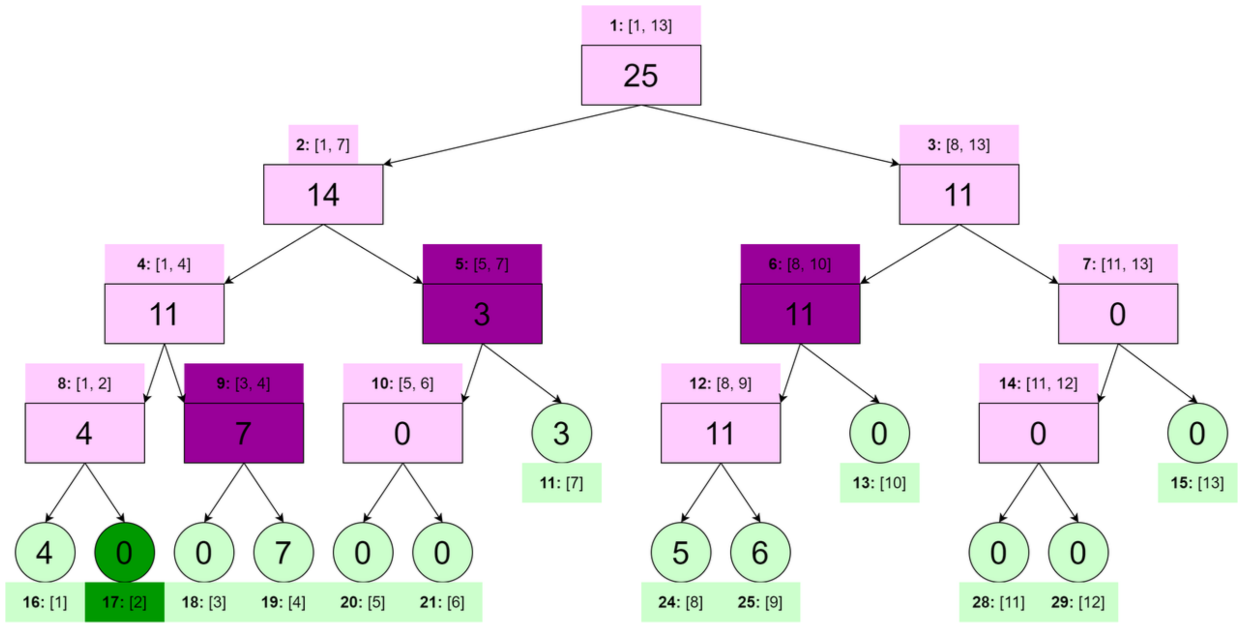
```
// This adds val to A[idx].
// Initially, we just call
// update(idx, val, 1)

// idx should always be in the range of
// the root node for this code.
// Beyond that, the code ensures
// it remains in the correct range.
void update(int idx, int val, int node=1)
{
    // Update the sum for this range,
    seg[node] += val;
```

```
        // If we are at a leaf node,
        // we are done.
        if(beg[node] == end[node])
        {
            return;
        }

        // Now, we just need to check whether
        // idx lies in the left node or the
        // right node and call the update
        // function on it.

        int left = node*2;
        int right = left+1;

        if(idx <= end[left])
        {
            update(idx, val, left);
        }
        else
        {
            update(idx, val, right);
        }
    }
```

Let us also add 4 to $\mathrm{A}[1]$, 7 to $\mathrm{A}[4]$, 3 to $\mathrm{A}[7]$ and 6 to $\mathrm{A}[9]$. Our updated segment tree now looks like:



Now, how do we go about answering sum queries on this tree? Let us consider the query $[2, 10]$. How do we get the sum of these elements?



The nodes that have been darkened cover the ranges we are interested in:

- Node 17: $[2, 2]$
- Node 7: $[3, 4]$
- Node 3: $[5, 7]$
- Node 6: $[8, 10]$

The union of these non-intersecting ranges is the range covered by the query. We just need the sum of all these nodes which is $0 + 7 + 3 + 11 = 21$.

The key optimization is that when we make our way downwards, if anytime we reach a node whose range is entirely covered by the range of the query, then we can just stop here as we have the sum of this entire range precomputed. A recursive implementation of this:

```
// This returns the sum of elements in the range:
// [start, finish] <intersection with> [beg[node], [end[node]].
// Initially, we call query(start, finish, 1) at the root node.

// Three cases:
//
// Complete Intersection: when the range of the node
// is completely contained in the range of the query.
//
// No Intersection: when the range of the node and
// the range of the query are completely disjoint.
//
// Partial Intersection: otherwise

int query(int start, int finish, int node=1)
{
    // For Complete Intersection,
    // we just return the sum of the node's range.
    if(start <= Beg[node] and End[node] <= finish)
```

```
        {
            return Seg[node];
        }

        // For No Intersection,
        // we return 0.
        if(finish < Beg[node] or End[node] < start)
        {
            return 0;
        }

        // For Partial Intersection,
        // we query both the left and right nodes.

        int left = node*2;
        int right = left+1;

        return (query(start, finish, left) + query(start, finish, right));
    }
```

Let's analyze the complexity of this code. Let the height of the node at the bottommost level be 1. The height of the nodes at the next upper level is 2. And so on. The height of the root node will be $\operatorname{ceil}(\log_2 N) + 1$.

We define two functions $f(node, h)$ and $g(node, h)$ which denote the number of nodes that our query function will visit in the subtree whose topmost element is $node$ and is at height h under two different conditions.

$f(node, h)$ is for when $\text{Beg}[node] < start \leq finish < \text{End}(node)$.

$g(node, h)$ for when $start \leq \text{Beg}[node] \leq finish < \text{End}(node)$ or $\text{Beg}[node] < start \leq \text{End}(node) \leq finish$.

$f()$ is the case when the range is entirely contained in this node. $g()$ is the case where the range intersects with one of the endpoints of the node and does not completely contain the node's range itself.

Note that for all other conditions, the answer will be 1 as there is no need to query any child nodes.

In both these cases, our function will certainly recursively call both the child nodes. The trivial cases are the ones where for the child nodes neither of the two above conditions hold and the answer is immediately 1. What are the non-trivial cases for $f()$?

When the

$$f(node, h) = \begin{cases} 1 + f(left, h-1) + 1, & \text{if the range is contained entirely in the left node} \\ 1 + 1 + f(right, h-1), & \text{if the range is contained entirely in the right node} \\ 1 + g(left, h-1) + g(right, h-1) & \text{if the range is partially in both the child nodes} \end{cases}$$

Similarly, let's list out the non-trivial cases for $g()$.

Without loss of generality, assume: $start \leq \text{Beg}[node] \leq finish < \text{End}(node)$.

$$g(node, h) = \begin{cases} 1 + g(left, h-1) + 1, & \text{if the range does not intersect with the right node} \\ 1 + 1 + g(right, h-1) & \text{if the range intersects with the right node} \end{cases}$$

For $g()$, as we descend each level, we add 2. So, in the worst case, $g(node, h) = 2h - 1$.

What about $f()$? If the range were to split at $node$, then in the worst case, $f(node, h) = 4h - 1$. This is the actual worst case for this as if the range is entirely contained in either of the child nodes, that would require us to visit fewer nodes.

So, the worst case for this function when called on the root node is $4h - 1$. And $h = \text{ceil}(log_2 N)$ so, the time complexity of this code is $O(log_2 N)$.

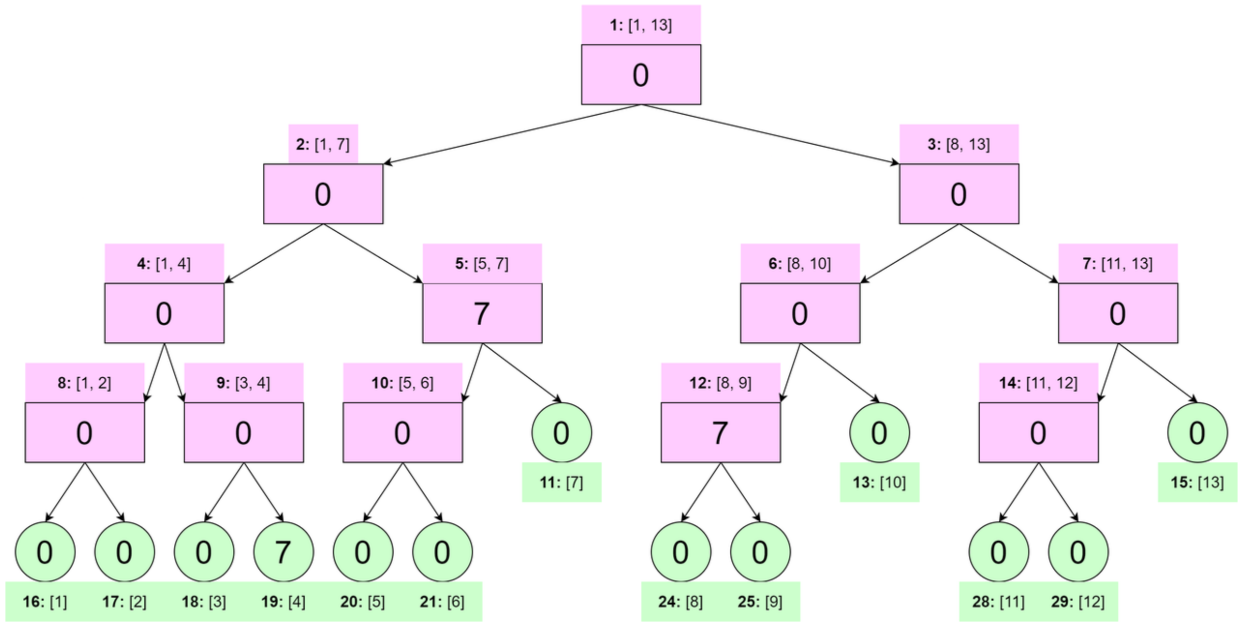
What if updates didn't add but replace?

What if the updates don't ask us to add a value to $\mathrm{A}[i]$ but rather to change it to some new value v? For this, we simply maintain the array A and when we have to update $\mathrm{A}[i]$ to v, we just call the update function with $val = v - \mathrm{A}[i]$.

Range Updates and Point Queries

Segment trees can also easily handle a similar question. Now, our updates specify a range $[l, r]$ and a value v which is to be added to every $\mathrm{A}[i]$ where $l \le i \le r$. Our queries specify an index i and we have to return $\mathrm{A}[i]$.

This time, the value of an element is the sum of values stored at all nodes whose range includes it. Let's look at how we would add 7 to all elements in the range $[4, 9]$.

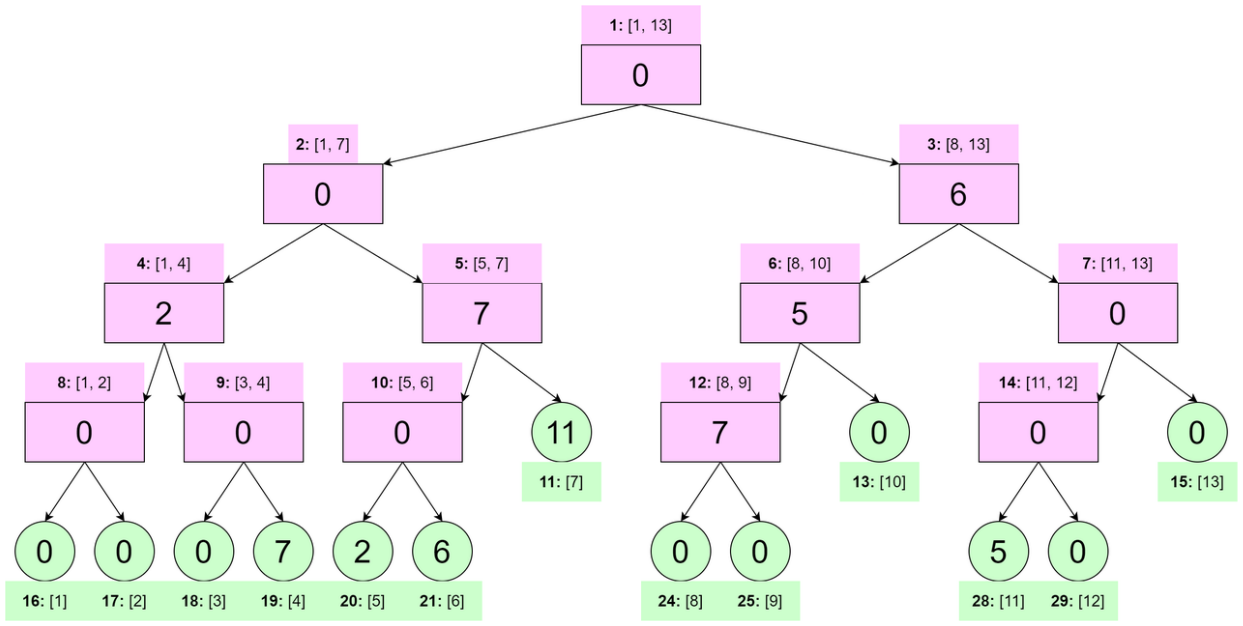


To find the value of any index, we just need to travel from the root to the appropriate leaf node adding up all the values along the path. Let us perform a few more updates:

- Add 5 to the range $[7, 11]$.

- Add 2 to the range $[1, 5]$.
- Add 6 to the range $[6, 13]$.

Now our array is: $[2, 2, 2, 9, 9, 13, 18, 18, 18, 11, 11, 6, 6]$. Following is the corresponding segment tree:



The code for constructing the tree remains the same. There are slight modifications to the other two functions. The update function now looks a lot like the query function from the previous question and vice versa.

```
// This adds val to the elements in the range:
// [start, finish] <intersection with> [beg[node], [end[node]].
// Initially, we call update(start, finish, val, 1) at the root node.

// Three cases:
//
// Complete Intersection: when the range of the node
// is completely contained in the range of the query.
//
// No Intersection: when the range of the node and
// the range of the query are completely disjoint.
//
// Partial Intersection: otherwise

void update(int start, int finish, int val, int node=1)
{
    // For Complete Intersection,
    // we just update the sum of the node's range.
    if(start <= Beg[node] and finish >= End[node])
    {
        Seg[node] += val;
        return;
    }

    // For No Intersection,
    // we do nothing.
    if(finish < Beg[node] or End[node] < start)
    {
        return;
    }

    // For Partial Intersection,
    // we update both the left and right nodes.

    int left = node*2;
    int right = left+1;

    update(start, finish, val, left);
    update(start, finish, val, right);
}
```

```
// This finds the value of A[idx].
// Initially, we just call
// query(idx, val, 1)

// idx should always be in the range of
// the root node for this code.
// Beyond that, the code ensures
// it remains in the correct range.
int query(int idx, int node=1)
{
    // If we are at a leaf node,
    // we are just return the value
    // at this node.
    if(beg[node] == end[node])
    {
        return seg[node];
    }

    int left = node*2;
    int right = left+1;

    // Now, we just need to check whether
    // idx lies in the left node or the
    // right node and add the value
    // returned on querying that node
    // to the value of the current node
    // and return it.

    if(idx <= end[left])
    {
        return seg[node]+query(idx, left);
    }
    else
    {
        return seg[node]+query(idx, right);
    }
}
```

What if updates didn’t add but replace?

How would we accommodate this modification for range updates? The hint for that lies in the following section.

Range Updates and Range Queries

This time we have updates which want us to add a value to all elements of the array in the specified range, and we have queries which want the sum of values of all elements of the array in the specified range.

We will work with the segment tree for point updates and range queries. That is, $\text{Seg}[node] = \sum_{i=\text{beg}[node]}^{\text{end}[node]} \text{A}[i]$. Each node stores the sum of array elements covered by its range. We can do queries on this but updating this could require $O(N)$ time. Particularly, if we were to add v to all the elements in the array, we would have to add v to all the nodes.

To deal with this, we use a technique called lazy propagation. Let's say we have to add v to all elements covered by the range of $node$. We update $\text{Seg}[node] = \text{Seg}[node] + v(\text{End[node]-Beg[node]+1})$. Now, instead of updating all the other nodes in its subtree, we just "make a note", reminding us that we have to do this. We do this by maintaining a lazy value $\text{Lazy}[node]$ for each node. Here, we will just update $\text{Lazy}[node] = \text{Lazy}[node] + v$.

Since we put this off for later, our update only takes $O(\log_2 N)$ time. When are these values actually propagated downwards? Only when we have to visit them! We just call the following code on $node$ before visiting its children:

```
void propagate(int node)
{
    int left = node*2;
    int right = left+1;

    Seg[left] += Lazy[node]*(End[left]-Beg[left]+1);
    Lazy[left] += Lazy[node];

    Seg[right] += Lazy[node]*(End[right]-Beg[right]+1);
    Lazy[right] += Lazy[node];

    // Now that we've propagated these values,
    // we reset the value for lazy propagation.
    Lazy[node] = 0;
}
```

Now, our new update code:

```
// This adds val to the elements in the range:
// [start, finish] <intersection with> [beg[node], [end[node]].
// Initially, we call update(start, finish, val, 1) at the root node.

// Three cases:
//
// Complete Intersection: when the range of the node
// is completely contained in the range of the query.
//
// No Intersection: when the range of the node and
// the range of the query are completely disjoint.
//
```

```
// Partial Intersection: otherwise

void update(int start, int finish, int val, int node=1)
{
    // For Complete Intersection,
    // we just update the sum of the node's range
    // and set the value for lazy propagation.
    if(start <= Beg[node] and finish >= End[node])
    {
        Seg[node] += val*(End[node]-Beg[node]+1);
        Lazy[node] += val
        return;
    }

    // For No Intersection,
    // we do nothing.
    if(finish < Beg[node] or End[node] < start)
    {
        return;
    }

    // For Partial Intersection,
    // we push the lazy values downwards,
    // and update both the left and right nodes.

    propagate(node);

    int left = node*2;
    int right = left+1;

    update(start, finish, val, left);
    update(start, finish, val, right);

    // Finally, we update the sum of
    // elements in the range of this node.
    Seg[node] = Seg[left] + Seg[right];
}
```

The new query code:

```
// This returns the sum of elements in the range:
// [start, finish] <intersection with> [beg[node], [end[node]].
// Initially, we call query(start, finish, 1) at the root node.

// Three cases:
//
// Complete Intersection: when the range of the node
// is completely contained in the range of the query.
//
// No Intersection: when the range of the node and
// the range of the query are completely disjoint.
//
// Partial Intersection: otherwise
```

```
int query(int start, int finish, int node=1)
{
    // For Complete Intersection,
    // we just return the sum of the node's range.
    if(start <= Beg[node] and End[node] <= finish)
    {
        return Seg[node];
    }

    // For No Intersection,
    // we return 0.
    if(finish < Beg[node] or End[node] < start)
    {
        return 0;
    }

    // For Partial Intersection,
    // we push the lazy values downwards,
    // and query both the left and right nodes.

    propagate(node);

    int left = node*2;
    int right = left+1;

    return (query(start, finish, left) + query(start, finish, right));
}
```

What if updates didn't add but replace?

Try if you can figure out this variation.