Segment Trees

Segment Trees are an important data structure in CS. They solve the problem below:

*Supose you have an array of **N** elements $a_1, a_2, \cdots, a_n$ and can perform two such operations:*

1. You are given two numbers l and r ($l \leq r$) and must return the sum of $(a_l, a_{l+1}, \cdots, a_{r-1}, a_r)$;
2. You are given two number p and v and must add v to a_p.

The naive way of solving that is doing a for loop for the first operation (it costs O(r-l) operations) and a simples attribution to a_p (it costs O(1) operations).

Instead of that, the segment tree perform both operations in O(log(N)) in the worst case. Moreover we'll see that segment trees can do operation 2. not just for a single element of the array but for a range like operation 1. (you're given three numbers l, r, v and must add v to all elements $(a_l, \cdots, a_r)$).

The first operation is called query and the second one is called update.
When we do an operation in a single element, we call this a point-operation.
When we do an operation in an interval for the array, we call this a range-operation.

So now, let's see the code for the different types os segtrees.

Range-Query Point-Update Segment Trees

These are the ones that solve the original problem. We have a binary tree where the leaf nodes have the values of the original arrays and each internal node is the sum of its children.
Then, we say that the root represents the segment $[1, n]$, its left child represents the interval $\left[1, \frac{n}{2}\right]$ and its right child represents the interval $\left[\frac{n}{2} + 1, n\right]$ and so on. For any node that represents an interval $[i, j]$ its children represent that same interval but partitioned in the middle (half for each child).

To code that, we need the original array $arr[1 \cdots N]$, the tree array $t[1 \cdots 4N]$ and three procedures `build` , `update` and `querry` .

```
int n;
// These arrays are 1-indexed (that's important)
int arr[n], t[4*n];

// To build, we use a recursive function with parameters
// the left and right points of the segment and
// the current node in the tree
// (the tree is represented by an array)
void build(int l, int r, int i)
{
    // if we're in a leaf
    if (l == r) // the value of the tree is the value of the array
        t[i] = arr[l];
    else {
        // we pick the middle point of the segment
        int m = (l+r)/2;
        // build the left and right children
        build(1, m, 2*i);
        build(m+1, r, 2*i+1);
        // and set the value of the node
        t[i] = t[2*i] + t[2*i+1];
    }
}

// To update, we use a recursive function with parameters
// the left and right points of the current segment,
// the current node in the tree,
// the point that we want to update and
// the value that we'll add in the point
//
// Notice that updating a point means that we need to
// update all the nodes in the path from the root to the
// leaf.
void update(int l, int r, int i, int p, int v)
{
    // if we're in the leaf
    if (l == r) // update it directly
        t[i] += v;
    else {
        // we pick the middle point
        int m = (l+r)/2;
        // and use binary search property to find the position p
        if (p <= m) // if p is to the left
            update(l, m, 2*i, p, v); // update the left child
        else
            update(m+1, r, 2*i+1, p, v); // update the right child
        // (2i is the left child and 2i+1 is the right one)

        // after updating the children, update the current node
        t[i] = t[2*i] + t[2*i+1];
    }
}


// To query, we use a recursive function with parameters
// the left and right points of the current segment,
// the current node in the tree and
// the left and right points of the query segment
//
// There are three possibilities:
// * the current segment is entirely inside the querry segment
// * the current segment is partially inside the querry segment
// * the current segment is entirely outside the querry segment
int query(int l, int r, int i, int ql, int qr)
{
    // if the current segment is entirely inside the querry segment
    if (ql <= l && r <= qr) {
        // return the value of the current node because this segment is
        // relevant for the entirely ql, qr segment
        return t[i];
        // as the children of i are all inside the querry segment,
        // we could go further, but that's not necessary because
        // t[i] contains all the information we need
    }

    // if the current segment is outside the querry segment,
    // just return a value that doesn't change the final value
    if (l > qr || r < ql) return 0;

    // If this line is executed, we're in a partial overlap situation
    int m = (l+r)/2; //pick the middle point
    // And find the segments that we care about in the children
    return query(l, m, 2*i, ql, qr) + query(m+1, r, 2*i+1, ql, qr);
}
```

Notice one thing which is very important in segment trees. We could do other types of segment tree with other operations instead of addition. That could multiplication, finding the min or the max or any other associative math operation. Moreover, we could set the value of a_p to be v instead of adding v. (we could do actually any operation we want). That's the powerness of this data structure.

Range-Query Range-Update Segment Trees

We can also make a range-update point-query segtree, but as point-query is a particular case of range-query, I'll no show that implementation.

To make a range-query and update segment tree with O(log(N)) in both operations, we need first to understant an important tecnique to improve the velocity of this structure: the Lazy Propagation.

Lazy Propagation

In this tecnique, we will need one more array for each node of our tree: $lz[1 \cdots N]$. Then, when we're updating a segment by a value, we'll stop whenever we find a node with segment entirely inside the query segment and update just this node, marking its children and indicating that we need to update they and everything above that nodes.

For instance, supose we have this call: `update(1, 8, 1, 3, 4, 2)`. It is, we want to update our segment tree with 8 nodes in the segment [3, 4] by adding 2 to all elements in this range.

The root node represents the segment [1, 8], its left child represents [1, 4] and its right child represents [3, 4], the range we want to update. Above this node we have the left child representing [3, 3] (the element 3 itself) and the right child representing [4, 4].

In a naive way of updating, we would first update 3 by 2 and update all the nodes in the path from the root to the node representing [3, 3] and, after that, update 4 by 2 and all the nodes in the path from the root to the node representing [4, 4].
We can see clearly that as the node representing [3, 4] is the lowest common ancestor of [3, 3] and [4, 4], we'll update [3, 4] and all the nodes in the path from the root to [3, 4] twice.
Then, instead of that, we update [3, 4] just one time by adding in it the value 2 * 2, making *lz* of its children equal to 2 and returning without exploring more down in the tree.

But now supose we have this call: `query(1, 8, 1, 3, 3)`. It is, we want to know the value of the sum in the interval [3, 3] (i.e. just the value of the element 3). We'll do the same thing we're doing and find the node that represents [3, 3]. But after finding that value, we see that its value didn't change in the last update operation becausa we had just updated the node [3, 4].

To make sure we are using the laze propagation correctly, as soon as we reach in the node, we check if `lz` of that node is different of 0. If it is, we have an late update to do and we need to do that before returning the value of that node. So first we increment 2 in that node and then return its value.

If the tree was bigger, then we would update lazily that node and propagate the update of +2 to the nodes above making their `lz` increment by 2.

Finally, we need to undestant one more thing generically: how much do we sum in the ending node. We saw that the update was telling us to update 2 in the leaf nodes (which belong to the update segment) but we've updated 2 * 2 in the ending node. Now let's understand why 2 * 2.

One of these twos is because the +2 of the update. The other one is becausa we had 2 leaf in the segment. So a generic formula of how much we need to increment is actually $v \cdot (r - l + 1)$, where v is the value of the update and l and r are the left and right end points of the segment represented by the node we're updating. Notice that $r - l + 1$ represents the number of elements that we would

update if we were in the naive solution (the number of leaves under that particular node).
If lz if not zero in the node i, we do `t[i] += lz[i] * (l-r+1)` before doing any thing in that node. Then, we make `lz[i]` become zero, indicating that there are no more lazy updates in that node. But, of course, before clearing lz, we propagate the value to i children and as their l and r are different, they will be updated by the right value.

That's the idea, let's check out the code:

Implementation

The implementation can be found in the archive `LazySegTree.cpp` . There I say that we can create an auxiliary function to check if the node has lazy value or not. Here I'll just implement this version. It's a fast way to implement.

```
ll const N = 112345;
ll arr[N], t[4 * N], lz[4 * N];

void build(ll l, ll r, ll i) {
    if (l == r) t[i] = arr[l];
    else
    {
        ll m = l + (r-l)/2;
        build(l, m, 2*i);
        build(m+1, r, 2*i+1);
        t[i] = t[2*i] + t[2*i+1];
    }
    lz[i] = 0;
}

void push(ll l, ll r, ll i) {
    if (lz[i]) {
        t[i] += lz[i] * (r-l+1);
        if (l != r) {
            lz[2*i] += lz[i];
            lz[2*i+1] += lz[i];
        }
        lz[i] = 0;
    }
}

void update(ll l, ll r, ll i, ll ql, ll qr, ll x) {
    push(l, r, i);
    if (qr < l || r < ql) return;
    if (ql <= l && r <= qr) {
        lz[i] = x;
        push(l, r, i);
    } else {
        int m = (l+r)/2;
        update(l, m, 2*i, ql, qr, x);
        update(m+1, r, 2*i+1, ql, qr, x);
        t[i] = t[2*i] + t[2*i+1];
    }
}

ll query(int l, int r, int i, int ql, int qr) {
    push(l, r, i);
    if (ql <= l && r <= qr) return t[i];
    if (r < ql || qr < l) return 0;

    int m = (l+r)/2;
    return query(l, m, 2*i, ql, qr) +
           query(m+1, r, 2*i+1, ql, qr);
}
```