# Segment Tree I

By: Andrew Qi Tang

# Segment Tree

A data structure that can deal with lots of range queries.

Range query is finding the answer for a certain range from [l, r].

Divide And Conquer Structure

Binary Tree

Represents many intervals.

Allows querying and updating of many operations in logarithmic time.

# Example Array

| idx | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| arr[idx] | 2 | -3 | 7 | 8 | -7 | 8 | 4 | 0 |

# Node Structure

The Segment Tree is built on nodes.

A node represents an interval of the original array.

Uses “l” to represent the left endpoint and “r” to represent the right endpoint.

You can also add other elements inside, that represents the range. (eg. sum)

# Node Structure Code

```
struct node{
 int l, r; //represents the range [l, r]
 //add other stuff as well
}
```

# Segment Tree Structure

Uses nodes to represents intervals [l, r].

The root of the tree represents the whole array [1, N] where N is the size of array.

Every node except for the leaves, will be split into two other children, left and right.

Let mid = (l+r)/2, (rounded down)

Left child’s range: [1, mid], Right child’s range: [mid+1, r]

Leaves are when they represent one element: [l, r] where l == r.

# Segment Tree Labelling

The root will have a label of 1.

Assume that the current node’s labelling is idx.

Left child’s label is 2*idx.

Right child’s label is 2*idx+1.

# Segment Tree Structure With Labelling Block Visual

<table>
<tr><td colspan="8">1: [1, 8]</td></tr>
<tr><td colspan="4">2: [1, 4]</td><td colspan="4">3: [5, 8]</td></tr>
<tr><td colspan="2">4: [1, 2]</td><td colspan="2">5: [3, 4]</td><td colspan="2">6: [5, 6]</td><td colspan="2">7: [7, 8]</td></tr>
<tr><td>8: [1, 1]</td><td>9[2, 2]</td><td>10: [3, 3]</td><td>11: [4, 4]</td><td>12: [5, 5]</td><td>13: [6, 6]</td><td>14: [7, 7]</td><td>15: [8, 8]</td></tr>
</table>

# Push Up

Visually pushing up information from the adjacent children to the current node.

Whatever you set. (Eg. sum)

# Building The Segment Tree

Going to be built recursively.

The largest index possible will be less than or equal to 4*N.

Starting from the root label it 1 with interval [1, N].

Then recursively go to the left.

After that’s done recursively go to the right.

Finally, push up information from its adjacent children to the current node.

O(N) time to build.

# Building Function Code

```
node seg[4*size+1];

void build(int l, int r, int idx){
 seg[idx].l = l; //setting left endpoint
 seg[idx].r = r; //setting right endpoint
 if(l == r){
   //storing information in the leaves
   return;
 }
 int mid = (l + r) / 2; //getting mid
 build(l, mid, 2 * idx); //recursively going left
 build(mid + 1, r, 2 * idx + 1); //recursively going right
 //push up information from the children
}
```

# Range Query In Segment Tree

[ql, qr] represents the interval that we are trying to find.

Assume we are at a node with range [l, r].

In a segment tree, it is guaranteed that l <= ql and r >= qr.

Divided into 4 cases and done recursively.

# The 4 Cases

[ql, qr] is the query range, [l, r] is the node’s interval and mid = ⌊(l+r)/2⌋

1. Covers the whole interval completely
2. Only part of [l, mid]
3. Only part of [mid+1, r]
4. Part of both but not completely

# Case #1

Covers the whole interval completely.

Simply return the value up the subtree.

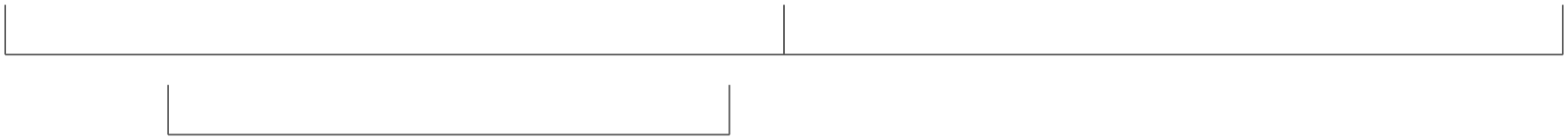
Check if ql == l and qr == r.

# Case #2

Only part of [l, mid].

Move to the left subtree with the same interval.

Check if qr <= mid.

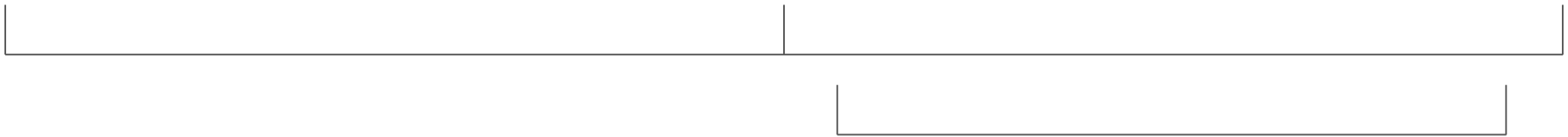


# Case #3

Only part of [mid+1, r].

Move to the left subtree with the same interval.

Check if l > mid.



# Case #4

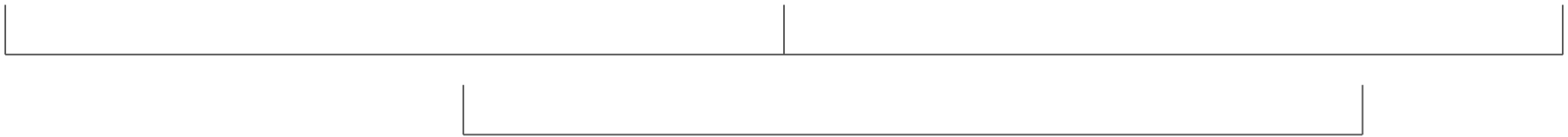
Part of both but not completely.

Split the query interval into [ql, mid] and [mid+1, qr]

Query the left subtree [ql, mid] and query the right subtree [mid+1, qr]

Combine results afterward.

This is the else or when ql <= mid and qr > mid.



# Range Query Function Code

```
int query(int l, int r, int idx){
   if(seg[idx].l == l && seg[idx].r == r){ //first case
       return seg[idx].v; //returned value
   }
   int mid = seg[idx].l + seg[idx].r >> 1; //getting mid point of the node
   if(r <= mid){ //second case
       return query(l, r, 2*idx);
   }
   else if(l > mid){ //third case
       return query(l, r, 2*idx+1);
   }
   else{ //fourth case
       return max(query(l, mid, 2*idx), query(mid+1, r, 2*idx+1)); //how to combine in this case: maximum
   }
}
```

# Point Update In Segment Tree

The value for one index can be changed in a Segment Tree.

Until we get to a leaf node, we recursively go down the respected node.

Assume that the position that we are trying to update is p.

If p <= mid, then it’s part of the left subtree.

Otherwise, it’s part of the right subtree.

When reaching the leaf just set the value to be that value.

# Point Update Function Code

```
void upd(int p, int v, int idx){
   if(seg[idx].l == seg[idx].r){ //if reaches a leaf node
       seg[idx].v = v; //updating the value
       return; //stop here
   }
   int mid = seg[idx].l + seg[idx].r >> 1; //get middle of node interval
   if(p <= mid){ //part of left subtree
       upd(p, v, 2*idx);
   }
   else{ //part of right subtree
       upd(p, v, 2*idx+1);
   }
   seg[idx].v = max(seg[2*idx].v, seg[2*idx+1].v); //push up the information in this case: maximum
}
```