Introduction To Segment Tree

Example

Suppose we have an array:

```
+-------+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |
+-------+-----+-----+-----+-----+-----+-----+
| Value |  -1 |  3  |  4  |  0  |  2  |  1  |
+-------+-----+-----+-----+-----+-----+-----+
```

We want to perform some query on this array. For example:

What is the minimum from index-**2** to index-**4**? -> 0
What is the maximum from index-**0** to index-**3**? -> 4
What is the summation from index-**1** to index-**5**? -> 10

How do we find it out?

Brute Force:
We could traverse the array from the starting index to the finishing index and answer our query. In this approach, each query takes `O(n)` time where **n** is the difference between the starting index and finishing index. But what if there are millions of numbers and millions of queries? For **m** queries, it would take `O(mn)` time. So for 10^5 values in our array, if we conduct 10^5 queries, for worst case, we'll need to traverse 10^{10} items.

Dynamic Programming:
We can create a 6X6 matrix to store the values for different ranges. For range minimum query(RMQ), our array would look like:

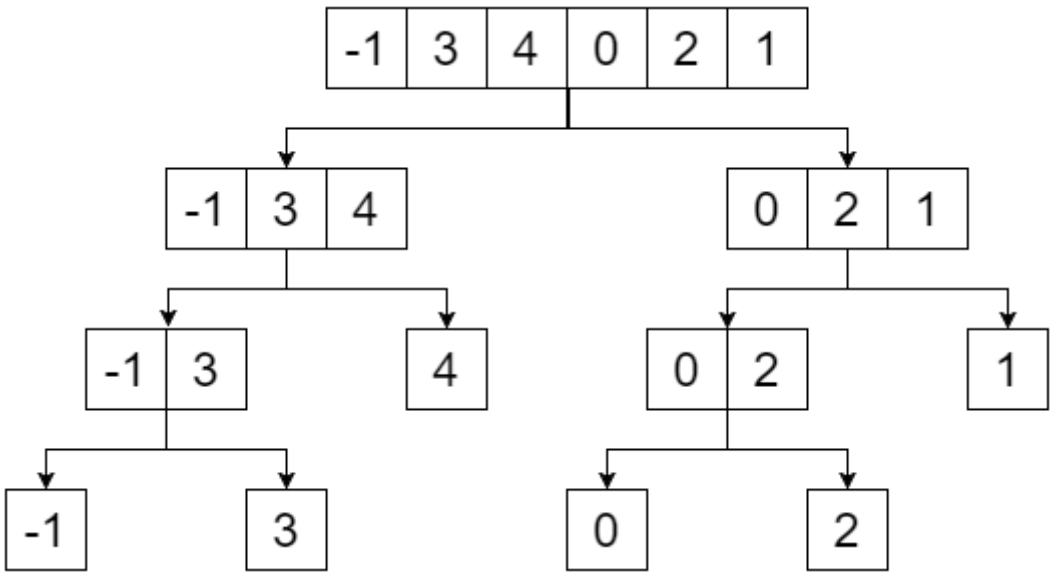
```
          0     1     2     3     4     5
   +-----+-----+-----+-----+-----+-----+-----+
   |     |  -1 |  3  |  4  |  0  |  2  |  1  |
   +-----+-----+-----+-----+-----+-----+-----+
0  |  -1 |  -1 |  -1 |  -1 |  -1 |  -1 |  -1 |
   +-----+-----+-----+-----+-----+-----+-----+
1  |  3  |     |  3  |  3  |  0  |  0  |  0  |
   +-----+-----+-----+-----+-----+-----+-----+
2  |  4  |     |     |  4  |  0  |  0  |  0  |
   +-----+-----+-----+-----+-----+-----+-----+
3  |  0  |     |     |     |  0  |  0  |  0  |
   +-----+-----+-----+-----+-----+-----+-----+
4  |  2  |     |     |     |     |  2  |  1  |
   +-----+-----+-----+-----+-----+-----+-----+
5  |  1  |     |     |     |     |     |  1  |
   +-----+-----+-----+-----+-----+-----+-----+
```

Once we have this matrix build, it would be sufficient to answer all the RMQs. Here, **Matrix[i][j]** would store the minimum value from index-**i** to index-**j**. For example: The minimum from index-**2** to index-**4** is **Matrix[2][4] = 0**. This matrix answers the query in `O(1)` time, but it takes **O(n²)** time to build this matrix and `O(n²)` space to store it. So if **n** is a really big number, this doesn't scale very well.

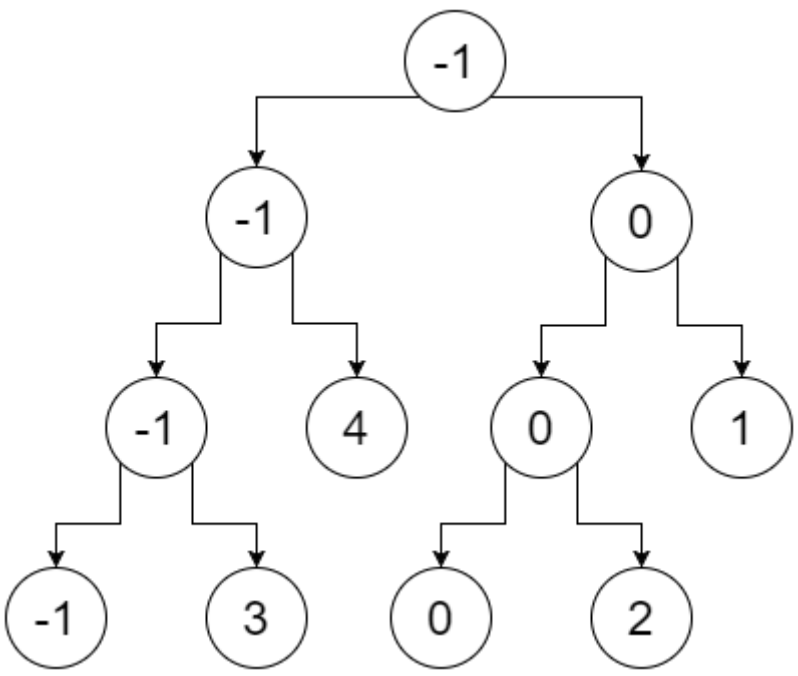
Segment Tree:

A segment tree is a tree data structure for storing intervals, or segments. It allows querying which of the stored segments contain a given point. It takes `O(n)` time to build a segment tree, it takes `O(n)` space to maintain it and it answers a query in `O(logn)` time.

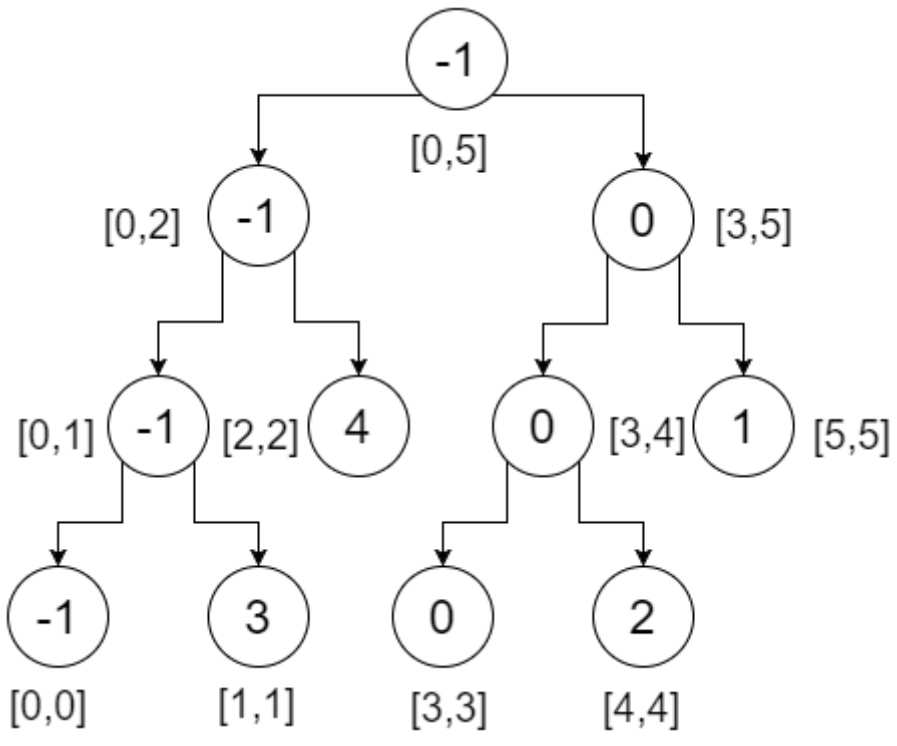
Segment tree is a binary tree and the elements of the given array will be the leaves of that tree. We'll create the segment tree by dividing the array in half till we reach a single element. So our division would provide us with:



Now if we replace the non-leaf nodes with the minimum value of their leaves, we get:



So this is our segment tree. We can notice that, the root node contains the minimum value of the whole array(range [0,5]), its left child contains the minimum value of our left array(range [0,2]), right child contains the minimum value of our right array(range [3,5]) and so on. The leaf nodes contain minimum value of each individual points. We get:



Now let's do a range query on this tree. To do a range query, we need to check three conditions:

Partial Overlap: We check both leaves.

Total Overlap: We return the value stored in the node.
No Overlap: We return a very large value or infinity.

Let's say, we want to check range **[2,4]**, that means we want to find the minimum from index-**2** to **4**. We'll start from the root and check if the range in our nodes is overlapped by our query range or not. Here,

[2,4] doesn't completely overlap **[0,5]**. So we'll check both directions.
At left subtree, **[2,4]** partially overlaps **[0,2]**. We'll check both directions.
At left subtree, **[2,4]** does not overlap **[0,1]**, so this is not going to contribute to our answer. We return **infinity**.
At right subtree, **[2,4]** totally overlaps **[2,2]**. We return **4**.
The minimum of these two returned values(4, infinity) is **4**. We get **4** from this portion.
At right subtree, **[2,4]** partially overlaps. So we'll check both directions.
At left subtree, **[2,4]** completely overlaps **[3,4]**. We return **0**.
At right subtree, **[2,4]** doesn't overlap **[5,5]**. We return **infinity**.
The minimum of these two returned values(0, infinity) is **0**. We get **0** from this portion.
The minimum of the returned values(4,0) is **0**. This is our desired minimum in range **[2,4]**.

Now we can check that, no matter what our query is, it would take maximum **4** steps to find the desired value from this segment tree.

Use:

Range Minimum Query
Lowest Common Ancestor
Lazy Propagation
Dynamic Subtree Sum
Neglect & Min
Dual Fields
Finding k-th Smallest Element
Finding Number of Unordered Pairs

‹ Previous Next ›