<https://leetcode.com/problems/count-integers-in-intervals/description/>

Given an empty set of intervals, implement a data structure that can:

Add an interval to the set of intervals.

Count the number of integers that are present in at least one interval.

Implement the CountIntervals class:

CountIntervals() Initializes the object with an empty set of intervals.

void add(int left, int right) Adds the interval [left, right] to the set of intervals.

int count() Returns the number of integers that are present in at least one interval.

Note that an interval [left, right] denotes all the integers x where left <= x <= right.

**Example 1:**

**Input**

["CountIntervals", "add", "add", "count", "add", "count"]

[[], [2, 3], [7, 10], [], [5, 8], []]

**Output**

[null, null, null, 6, null, 8]

**Explanation**

CountIntervals countIntervals = new CountIntervals(); // initialize the object with an empty set of intervals.

countIntervals.add(2, 3); // add [2, 3] to the set of intervals.

countIntervals.add(7, 10); // add [7, 10] to the set of intervals.

countIntervals.count(); // return 6

// the integers 2 and 3 are present in the interval [2, 3].

// the integers 7, 8, 9, and 10 are present in the interval [7, 10].

countIntervals.add(5, 8); // add [5, 8] to the set of intervals.

countIntervals.count(); // return 8

// the integers 2 and 3 are present in the interval [2, 3].

// the integers 5 and 6 are present in the interval [5, 8].

// the integers 7 and 8 are present in the intervals [5, 8] and [7, 10].

// the integers 9 and 10 are present in the interval [7, 10].

**Constraints:**

1 <= left <= right <= 10^9

At most 10^5 calls in total will be made to add and count.

At least one call will be made to count.

**Attempt 1: 2024-01-07**

**Solution 1: Insert + Merge Interval (10min, TLE 69/73)**

**This solution exactly bring the idea from L56, L57, L352**

class CountIntervals {

    List<int[]> intervals;

    public CountIntervals() {

        intervals = new ArrayList<>();

    }

    public void add(int left, int right) {

        int[] newInterval = new int[]{left, right};

        int len = intervals.size();

        int i = 0;

        while(i < len && intervals.get(i)[1] < newInterval[0]) {

            i++;

        }

        int removeCount = 0;

        while(i < len && newInterval[1] >= intervals.get(i)[0]) {

            newInterval[0] = Math.min(newInterval[0], intervals.get(i)[0]);

            newInterval[1] = Math.max(newInterval[1], intervals.get(i)[1]);

            intervals.remove(i);

            removeCount++;

            i--;

            i++;

            if(i == len - removeCount) {

                break;

            }

        }

        intervals.add(i, newInterval);

    }

    public int count() {

        int result = 0;

        for(int[] interval : intervals) {

            result += interval[1] - interval[0] + 1;

        }

        return result;

    }

}

/\*\*

 \* Your CountIntervals object will be instantiated and called as such:

 \* CountIntervals obj = new CountIntervals();

 \* obj.add(left,right);

 \* int param\_2 = obj.count();

 \*/

Time Complexity: O(N), overall cost for N numbers is O(N^2)

Space Complexity: O(N)

**Solution 2: TreeMap (60min)**

**A new cognition on how to use 'TreeMap.floorKey()'**

class CountIntervals {

    // <key, value> -> <left, right>

    TreeMap<Integer, Integer> intervals;

    int count = 0;

    public CountIntervals() {

        intervals = new TreeMap<>();

    }

    // TreeMap.floorKey() -> Returns the greatest "key" less than

    // or equal to the given key, or null if there is no such key.

    public void add(int left, int right) {

        // 'intervals.get(intervals.floorKey(right))' will definitely return

        // the immediate previous interval's right end, since

        // 'intervals.floorKey(right)' only return immediate previous interval's

        // left end, that's because for current interval {left, right} pending

        // on insert not on 'intervals' yet, so 'intervals.floorKey(right)' won't

        // return current interval's left end, the while loop will guarantee

        // keep merging

        while(intervals.floorKey(right) != null && intervals.get(intervals.floorKey(right)) >= left) {

            int preLeft = intervals.floorKey(right);

            int preRight = intervals.get(preLeft);

            count -= preRight - preLeft + 1;

            left = Math.min(left, preLeft);

            right = Math.max(right, preRight);

            intervals.remove(preLeft);

        }

        intervals.put(left, right);

        count += right - left + 1;

    }

    public int count() {

        return count;

    }

}

/\*\*

 \* Your CountIntervals object will be instantiated and called as such:

 \* CountIntervals obj = new CountIntervals();

 \* obj.add(left,right);

 \* int param\_2 = obj.count();

 \*/

Time Complexity: O(logN), overall cost for N numbers is O(N\*logN)

Each interval will enter the map once and leave the map once, and the cost is log(n) for treeMap. Therefore, the overall cost for (n) add is n\*log(n), amortized to O(logn) for each add.

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/count-integers-in-intervals/solutions/2039728/java-treemap-w-explanation-readability/>

Each interval will enter the map once and leave the map once, and the cost is log(n) for treeMap. Therefore, the overall cost for (n) add is n\*log(n), amortized to O(logn) for each add.

class CountIntervals {

// Interval treemap start -> finish.

TreeMap<Integer, Integer> s;

int count;

public CountIntervals() {

s = new TreeMap<Integer, Integer>();

count = 0;

}

public void add(int left, int right) {

// Add interval if there is no overlapping.

if (s.floorKey(right) == null || s.get(s.floorKey(right)) < left) {

s.put(left, right);

count += (right - left + 1);

} else {

int start = left;

int end = right;

// Remove overlapping intervals and update count.

while (true) {

int l = s.floorKey(end);

int r = s.get(l);

start = Math.min(start, l);

end = Math.max(end, r);

count -= (r - l + 1);

s.remove(l);

// Break the loop until there is no overlapping with interval (start, end).

if (s.floorKey(end) == null || s.get(s.floorKey(end)) < start) {

break;

}

}

// Add (start, end) to TreeMap and update count.

s.put(start, end);

count += (end - start + 1);

}

}

public int count() {

return count;

}

}

<https://leetcode.com/problems/count-integers-in-intervals/solutions/2039728/java-treemap-w-explanation-readability/comments/1392566>

Great idea, the hard part is: only check every intervals on the left of current right, this simplified the bi-direction merge. Reorganize the code a little bit:

class CountIntervals {

TreeMap<Integer, Integer> map = new TreeMap();

int count = 0;

public CountIntervals() {

}

public void add(int left, int right) {

while(map.floorKey(right) != null && map.get(map.floorKey(right)) >= left){

int pre\_left = map.floorKey(right);

int pre\_right = map.get(pre\_left);

left = Math.min(left, pre\_left);

right = Math.max(right, pre\_right);

count -= pre\_right - pre\_left + 1;

map.remove(pre\_left);

}

map.put(left, right);

count += right - left + 1;

}

public int count() {

return count;

}

}

**Solution 3: Segment Tree (180min)**

**The template is exactly similar to L715.Range Module (Ref.L307) how we build / update / query the Segment Tree**

**Wrong Solution**

class CountIntervals {

    SegmentTree segmentTree;

    public CountIntervals() {

        segmentTree = new SegmentTree();

    }

    public void add(int left, int right) {

        segmentTree.add(left, right);

    }

    public int count() {

        return segmentTree.query();

    }

}

class SegmentTree {

    TreeNode root;

    public SegmentTree() {

        // 1 <= left <= right <= 10^9

        root = new TreeNode(1, (int)1e9);

    }

    public void add(int left, int right) {

        update(root, left, right);

    }

    public void update(TreeNode root, int left, int right) {

        // The root fully inside update range

        if(root.lo >= left && root.hi <= right) {

            root.count = root.hi - root.lo + 1;

            root.left = null;

            root.right = null;

            return;

        }

        int mid = root.lo + (root.hi - root.lo) / 2;

        if(root.left == null) {

            root.left = new TreeNode(root.lo, mid);

        }

        if(root.right == null) {

            root.right = new TreeNode(mid + 1, root.hi);

        }

        if(right <= mid) {

            update(root.left, left, right);

        } else if(left > mid) {

            update(root.right, left, right);

        } else {

            update(root.left, left, mid);

            update(root.right, mid + 1, right);

        }

        root.count = root.left.count + root.right.count;

    }

    public int query() {

        return root.count;

    }

}

class TreeNode {

    int lo;

    int hi;

    int count;

    TreeNode left;

    TreeNode right;

    public TreeNode(int lo, int hi) {

        this.lo = lo;

        this.hi = hi;

    }

}

/\*\*

 \* Your CountIntervals object will be instantiated and called as such:

 \* CountIntervals obj = new CountIntervals();

 \* obj.add(left,right);

 \* int param\_2 = obj.count();

 \*/

**Test out by**

Input

["CountIntervals","count","add","add","add","add","add","count","add","add"]

[[],[],[8,43],[13,16],[26,33],[28,36],[29,37],[],[34,46],[10,23]]

Output

[null,0,null,null,null,null,null,21,null,null]

Expected

[null,0,null,null,null,null,null,36,null,null]

===============================================

CountIntervals

count -> 0

add [8,43]

add [13,16]

add [26,33]

add [28,36]

add [29,37]

count -> 21 (expect 36)

add [34,46]

add [10,23]

===============================================

43 - 8 + 1 = 36

all subranges inside [8,43]

8 43

13 16

26 33

28 36

29 37

**Step by Step explain why subtree node of parent node CANNOT default count = 0, but have to inherit its parent's node count if parent's node already calculate and assign count before(in previous Round #)**

Round 1:

add(8,43)

                                                                root=[1,100]

                                                                count=0->36+0=36

                                        /                                                               \

                                    node=[1,50]                                                      node=[51,100]

                                    count=0->18+18=36                                                count=0

                                /                                          \

                        node=[1,25]                                       node=[26,50]

                        count=0->6+12=18                                  count=0->13+5=18

                    /              \                                   /                         \

              node=[1,13]          node=[14,25]                 node=[26,38]                     node=[39,50]

              count=0->0+6=6       count=0->12                  count=0->13                      count=0->5+0=5

           /           \               |                            |                           /                      \

     node=[1,7]    node=[8,13]     25-14+1=12                   38-26+1=13                 node=[39,44]               node=[45,50]

     count=0       count=0->6      count=12                     count=13                   count=0->3+2=5             count=0

                       |           its.left/right               its.left/right            /                    \

                   13-8+1=6        origianlly=null              origianlly=null     node=[39,41]             node=[42,44]

                   count=6         return                       return              count=0->3               count=0->2+0=2

                   its.left/right                                                       |                   /           \

                   origianlly=null                                                  41-39+1=3         node=[42,43]   node=[44,44]

                   return                                                           count=3           count=0->2     count=0

                                                                                    its.left/right        |

                                                                                    origianlly=null   43-42+1=2

                                                                                    return            count=2

                                                                                                      its.left/right

                                                                                                      origianlly=null

                                                                                                      return

Round 2:

Based on Round 1 add(8,43) segment tree above, we continue add(13,16), node number will not change since (13,16) fully inside(8,43) range,

and the most critical thing is we get clear how below two statement works when create left / right subtree node, we cannot default subtree

node count = 0, since it has to inherit parent node count when parent node count calculate and assign count before(in previous Round #), we

have below graphic example, compare with Round 1 basement Segement Tree, we can see node=[11,13] and node=[14,19] further split into two

subtree nodes, and since node=[11,13] and node=[14,19] already calculate and assign count before(in previous Round 1), the newly created

subtree nodes must inherit node=[11,13] and node=[14,19] count values, and follow below formula to calculate what are the split out counts

if(root.left == null) {

    root.left = new TreeNode(root.lo, mid, root.count > 0 ? mid - root.lo + 1 : 0);

}

if(root.right == null) {

    root.right = new TreeNode(mid + 1, root.hi, root.count > 0 ? root.hi - (mid + 1) + 1 : 0);

}

add(13,16)

                                                                                        root=[1,100]

                                                                                        count=0->36+0=36

                                        /                                                                                                         \

                                    node=[1,50]                                                                                                node=[51,100]

                                    count=0->18+18=36(18+18=36 no change)                                                                                           count=0

                                /                                                                                    \

                        node=[1,25]                                                                               node=[26,50]

                        count=0->6+12=18(6+12=18 no change)                                                       count=0->13+5=18

                    /                                                   \                                   /                                  \

              node=[1,13]                                           node=[14,25]                        node=[26,38]                          node=[39,50]

              count=0->0+6=6                                        count=0->12                         count=0->13                           count=0->5+0=5

           /           \                                                |                                   |                           /                      \

     node=[1,7]    node=[8,13]                                      25-14+1=12                          38-26+1=13                 node=[39,44]               node=[45,50]

     count=0       count=0->6                                       count=12(6+6=12 no change)          count=13                   count=0->3+2=5             count=0

                       |                                            its.left/right                      its.left/right            /                    \

                   13-8+1=6                                         origianlly=null                     origianlly=null     node=[39,41]             node=[42,44]

                   count=6                                          return                              return              count=0->3               count=0->2+0=2

                   its.left/right                             /                   \                         |                   /           \

                   origianlly=null                       node=[14,19]             node=[20,25]          41-39+1=3         node=[42,43]   node=[44,44]

                   return                                count=6(3+3=6 no change) count=6               count=3           count=0->2     count=0

            /                   \                        root.count>0?            root.count>0?         its.left/right        |

    node=[8,10]             node=[11,13]                 mid-root.lo+1:0          root.hi-(mid+1)+1:0   origianlly=null   43-42+1=2

    count=3                 count=3(2+1=3 no change)     since root.count=12>0    since root.count=12>0 return            count=2

    root.count>0?           root.count>0?                count initialize         count initialize      its.left/right

    mid-root.lo+1:0         root.hi-(mid+1)+1:0          =19-14+1=6 inst of 0     =25-19=6 inst of 0    origianlly=null

    since root.count=6>0    since root.count=6>0                   \                                    return

    count initialize        count initialize                        \

    =10-8+1=3 inst of 0     =13-10=3 inst of 0                       \--------------------\

                            /                 \                       \                    \

                    node=[11,12]            node=[13,13]           node=[14,16]            node=[17,19]

                    count=2                 count=1                count=3                 count=3

                    root.count>0?           root.count>0?          root.count>0?           root.count>0?

                    mid-root.lo+1:0         root.hi-(mid+1)+1:0    mid-root.lo+1:0         root.hi-(mid+1)+1:0

                    since root.count=3>0    since root.count=3>0   since root.count=6>0    since root.count=6>0

                    count initialize        count initialize       count initialize        count initialize

                    =12-11+1=2 inst of 0    =13-12=1 inst of 0     =16-14+1=3 inst of 0    =19-16=3 inst of 0

                                            its.left/right         its.left/right

                                            origianlly=null        origianlly=null

                                            return                 return

**Correct Solution**

class CountIntervals {

    SegmentTree segmentTree;

    public CountIntervals() {

        segmentTree = new SegmentTree();

    }

    public void add(int left, int right) {

        segmentTree.add(left, right);

    }

    public int count() {

        return segmentTree.query();

    }

}

class SegmentTree {

    TreeNode root;

    public SegmentTree() {

        // 1 <= left <= right <= 10^9

        root = new TreeNode(1, (int)1e9, 0);

    }

    public void add(int left, int right) {

        update(root, left, right);

    }

    public void update(TreeNode root, int left, int right) {

        // The root fully inside update range

        if(root.lo >= left && root.hi <= right) {

            root.count = root.hi - root.lo + 1;

            root.left = null;

            root.right = null;

            return;

        }

        int mid = root.lo + (root.hi - root.lo) / 2;

        if(root.left == null) {

            root.left = new TreeNode(root.lo, mid, root.count > 0 ? mid - root.lo + 1 : 0);

        }

        if(root.right == null) {

            root.right = new TreeNode(mid + 1, root.hi, root.count > 0 ? root.hi - (mid + 1) + 1 : 0);

        }

        if(right <= mid) {

            update(root.left, left, right);

        } else if(left > mid) {

            update(root.right, left, right);

        } else {

            update(root.left, left, mid);

            update(root.right, mid + 1, right);

        }

        root.count = root.left.count + root.right.count;

    }

    public int query() {

        return root.count;

    }

}

class TreeNode {

    int lo;

    int hi;

    int count;

    TreeNode left;

    TreeNode right;

    public TreeNode(int lo, int hi, int count) {

        this.lo = lo;

        this.hi = hi;

        this.count = count;

    }

}

/\*\*

 \* Your CountIntervals object will be instantiated and called as such:

 \* CountIntervals obj = new CountIntervals();

 \* obj.add(left,right);

 \* int param\_2 = obj.count();

 \*/

Time Complexity: O(logN)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/count-integers-in-intervals/solutions/2039866/java-python-segment-tree/>

class Node {

int lower, upper, val;

Node left, right;

public Node(int lower, int upper, int val) {

this.lower = lower;

this.upper = upper;

this.val = val;

}

}

class CountIntervals {

Node root;

public CountIntervals() {

root = new Node(0, 1000000000, 0);

}

private void setRange(Node node, int left, int right) {

if (left <= node.lower && node.upper <= right) {

node.val = node.upper - node.lower + 1;

node.left = null;

node.right = null;

return;

}

int mid = (node.upper + node.lower) / 2;

if (node.left == null && node.right == null) {

node.left = new Node(node.lower, mid, node.val > 0 ? mid - node.lower + 1 : 0);

node.right = new Node(mid + 1, node.upper, node.val > 0 ? node.upper - (mid + 1) + 1 : 0);

}

if (left <= mid)

setRange(node.left, left, right);

if (right > mid)

setRange(node.right, left, right);

node.val = node.left.val + node.right.val;

}

public void add(int left, int right) {

setRange(root, left, right);

}

public int count() {

return root.val;

}

}

**Refer to**

<https://leetcode.com/problems/count-integers-in-intervals/solutions/2039888/c-using-segment-tree/comments/1392580>

class CountIntervals {

int count;

Node node;

public CountIntervals() {

node = new Node(1, 1\_000\_000\_000);

}

public void add(int left, int right) {

addRange(left,right, node);

}

public int count() {

return node.val;

}

private void addRange(int left, int right, Node root) {

if(root.val == root.max - root.min + 1) return;

int mid = root.min + (root.max - root.min)/2;

if(left <= root.min && root.max <= right) {

root.val = root.max - root.min + 1;

return;

}

if(left <= mid) {

if(root.left == null) root.left = new Node(root.min, mid);

addRange(left,Math.min(mid, right), root.left);

}

if(right > mid) {

if(root.right == null) root.right = new Node(mid+1, root.max);

addRange(Math.max(mid+1, left), right, root.right);

}

root.val = 0;

if(root.left != null){

root.val += root.left.val;

}

if(root.right != null) {

root.val += root.right.val;

}

return;

}

}

class Node {

int min, max, val;

Node left, right;

public Node(int min, int max) {

this.min = min;

this.max = max;

}

}

**Refer to**

<https://algo.monster/liteproblems/2276>

**Problem Description**

The LeetCode problem requires us to construct a data structure that supports adding intervals and counting the number of unique integer elements that lie within at least one of those intervals. After an interval is added, we need to keep track of all the unique numbers covered by all intervals. Specifically, the intervals are inclusive, meaning that if we add the interval [left, right], it includes all integers x where left <= x <= right.

**Intuition**

The intuition behind the solution is to use a segment tree data structure to efficiently manage the intervals and calculate the count of unique numbers. A segment tree is a tree data structure ideal for scenarios where we have to perform operations on interval or segments. It allows us to perform both updates and queries in logarithmic time.

For this particular problem, the segment tree nodes need to keep track of whether the entire segment is covered by any intervals (indicated by a tag) and the total count of unique numbers in that segment (tot). Each node represents a segment; if it is entirely covered by at least one interval (i.e., no part of the segment is outside any interval), we mark tag as 1, and tot becomes the size of that segment.

By utilizing lazy propagation in the segment tree, we avoid unnecessary updates to nodes. Lazy propagation means that unless necessary, we do not instantly update or query each child of a node. Instead, we mark the node as updated (tag = 1) and postpone updating its children until we specifically need to descend into them for further updates or queries. This technique significantly optimizes performance because not all nodes need to be looked at if a large chunk of them is entirely covered by the new interval.

When adding a new interval [left, right], we update the relevant segments of the tree. If an interval completely overlaps a segment, we set its tag to 1, and tot to the size of that segment. If the interval only partially covers the segment, we recursively update the corresponding child segments.

The count method simply returns the total count of unique numbers in all intervals, which is stored at the root of the segment tree.

This solution efficiently handles a series of interval additions and keeps an up-to-date count of the number of unique elements in all intervals, which can be returned in constant time.

**Solution Approach**

To implement the solution, we use a Segment Tree data structure, which is a binary tree structure that efficiently supports both update and query operations on intervals. Here's how we use it to approach the problem:

**Node Structure** - We define a class Node that represents a segment in the segment tree. Each node has four properties:

tag: A lazy propagation flag to indicate if the whole segment that the node represents is fully covered by intervals.

tot: The total count of unique numbers within the intervals covered by this segment.

left: Reference to the left child node in the segment tree.

right: Reference to the right child node in the segment tree.

**Initiation** - The CountIntervals class constructor initializes the segment tree by creating a root node. The root node's range is selected to cover the entire range of possible values (0 to 1000000010 in this implementation).

**Adding Intervals (add method)** - When a new interval [left, right] is added, we need to update our segment tree; this is done by calling the update method of the root node.

Here’s the update algorithm in detail:

The update operation is applied recursively. If the current segment represented by the node is fully covered by the interval being added, we mark the tag as 1 and set tot to the size of the segment (b - a + 1).

The recursion stops when the interval fully covers the segment corresponding to the current node, or the node is already fully marked (self.tag == 1).

If the interval only partially covers the segment, we split the interval into two subintervals at the midpoint and recursively update the left and right child nodes, if they exist. If a child does not exist, we create it at this point. After updating children, we update the current node's tot to be the sum of the children's tots.

The use of lazy propagation means we don't immediately update all descendants of a node; instead, we propagate updates as necessary to maintain correct counts when querying.

**Counting (count method)** - This method is straightforward; it simply returns the total count of unique numbers represented by the root of the segment tree (self.tree.tot). Since the tree is kept up-to-date by the add method, this value is readily available.

This approach allows for efficient operations on intervals and maintains an accurate count of the numbers contained in at least one interval with the Segment Tree's inherent logarithmic time complexity for updates and queries.

**Example Walkthrough**

To illustrate the solution approach described, let's consider a scenario where we're initially working with an empty set of intervals and we want to add intervals [1,3] and [2,5] to our data structure. We then want to count the number of unique integer elements covered by these intervals.

**Initiation:** Initialize the segment tree. The root node will cover the full range of possible values.

**Adding interval [1,3]:**

Call the update method to add interval [1,3].

Initially, no nodes are fully covered, so we will end up creating nodes to represent the required ranges and mark them accordingly.

After the update, we would have a tree where the segments [1,3] are marked with tag = 1 and tot = 3 since there are three unique numbers (1, 2, 3).

**Adding interval [2,5]:**

Call the update method to add interval [2,5].

We check if the current node's segment is covered by [2,5].

Nodes representing segments [2,3] are already marked, so we don't update them.

Segments [4,5] need to be updated. If nodes for these segments don't exist, we create them and mark them as covered (setting tag = 1 and tot to their respective segment sizes).

**Counting unique elements:**

We call the count method to find the total count of unique numbers in all intervals.

This is a straightforward tree query; the segment tree's root holds the total count in its tot variable.

In this case, it should return tot = 5 as there are five unique numbers (1 through 5) covered by the added intervals [1,3] and [2,5].

Throughout our operations, we use lazy propagation to update nodes only as necessary. This ensures our updates are efficient, and when we call count, the segment tree has already accounted for all the unique elements in the added intervals.

**Java Solution**

class Node {

    Node left;

    Node right;

    int start;

    int end;

    int mid;

    int value;

    int add;

    // Constructor for Node which sets up the segment range and calculates the mid point

    public Node(int start, int end) {

        this.start = start;

        this.end = end;

        this.mid = (start + end) >> 1; // Equivalent to (start + end) / 2

    }

}

// A class representing a segment tree data structure

class SegmentTree {

    private Node root = new Node(1, (int) 1e9 + 1); // Constructs the root node covering the entire range

    // Constructor is empty since the root is already initialized

    public SegmentTree() {

    }

    // Public method to modify a range of values in the segment tree

    public void modify(int left, int right, int value) {

        modify(left, right, value, root);

    }

    // Helper method to recursively modify the range [left, right] with the given value

    private void modify(int left, int right, int value, Node node) {

        if (left > right) {

            return;

        }

        if (node.start >= left && node.end <= right) {

            node.value = node.end - node.start + 1;

            node.add = value;

            return;

        }

        pushDown(node);

        if (left <= node.mid) {

            modify(left, right, value, node.left);

        }

        if (right > node.mid) {

            modify(left, right, value, node.right);

        }

        pushUp(node);

    }

    // Public method to query the value of a range in the segment tree

    public int query(int left, int right) {

        return query(left, right, root);

    }

    // Helper method to recursively query the range [left, right]

    private int query(int left, int right, Node node) {

        if (left > right) {

            return 0;

        }

        if (node.start >= left && node.end <= right) {

            return node.value;

        }

        pushDown(node);

        int totalValue = 0;

        if (left <= node.mid) {

            totalValue += query(left, right, node.left);

        }

        if (right > node.mid) {

            totalValue += query(left, right, node.right);

        }

        return totalValue;

    }

    // Method to update the node's value based on values of its children

    private void pushUp(Node node) {

        node.value = node.left.value + node.right.value;

    }

    // Method to propagate the lazy update values down to a node's children

    private void pushDown(Node node) {

        if (node.left == null) {

            node.left = new Node(node.start, node.mid);

        }

        if (node.right == null) {

            node.right = new Node(node.mid + 1, node.end);

        }

        if (node.add != 0) {

            Node left = node.left, right = node.right;

            left.add = node.add;

            right.add = node.add;

            left.value = left.end - left.start + 1;

            right.value = right.end - right.start + 1;

            node.add = 0;

        }

    }

}

// Class to count intervals using SegmentTree

class CountIntervals {

    private SegmentTree tree = new SegmentTree();

    // Constructor is empty since the SegmentTree is already initialized

    public CountIntervals() {

    }

    // Method to add an interval in the SegmentTree

    public void add(int left, int right) {

        tree.modify(left, right, 1);

    }

    // Method to count the covered intervals in the SegmentTree

    public int count() {

        return tree.query(1, (int) 1e9);

    }

}

/\*\*

 \* Usage:

 \*

 \*  CountIntervals obj = new CountIntervals();

 \*  obj.add(left, right);

 \*  int count = obj.count();

 \*/

**Time and Space Complexity**

**Time Complexity**

The time complexity of the add method is O(log N) where N is the range of the intervals, because update function divides the interval into two halves each time, similar to a binary search. However, it also depends on the distribution of the intervals added which could worsen the time complexity if the intervals are distributed in such a way that they lead to the creation of many nodes in the segments.

For count, the time complexity is O(1) as it merely returns the total number of intervals (tot) stored at the root of the segment tree.

**Space Complexity**

The space complexity can be in the worst case O(N) where N is the range of the intervals. This would be the case if the update function is called in such a way that every possible interval becomes a node, effectively storing the whole segment tree in memory. However, the space complexity in typical use-cases would be better than this, since many intervals may overlap, reducing the number of nodes needed to represent the intervals.