<https://leetcode.com/problems/my-calendar-iii/description/>

A k-booking happens when k events have some non-empty intersection (i.e., there is some time that is common to all k events.)

You are given some events [startTime, endTime), after each given event, return an integer k representing the maximum k-booking between all the previous events.

Implement the MyCalendarThree class:

* MyCalendarThree() Initializes the object.
* int book(int startTime, int endTime) Returns an integer k representing the largest integer such that there exists a k-booking in the calendar.

**Example 1:**

Input

["MyCalendarThree", "book", "book", "book", "book", "book", "book"]

[[], [10, 20], [50, 60], [10, 40], [5, 15], [5, 10], [25, 55]]

Output

[null, 1, 1, 2, 3, 3, 3]

Explanation

MyCalendarThree myCalendarThree = new MyCalendarThree();

myCalendarThree.book(10, 20); // return 1

myCalendarThree.book(50, 60); // return 1

myCalendarThree.book(10, 40); // return 2

myCalendarThree.book(5, 15); // return 3

myCalendarThree.book(5, 10); // return 3

myCalendarThree.book(25, 55); // return 3

**Constraints:**

* 0 <= startTime < endTime <= 109
* At most 400 calls will be made to book.

**Attempt 1: 2023-12-01**

**Solution 1: Sweep Line + TreeMap (10 min)**

**注意：Sweep Line + TreeMap就是用自动排序的TreeMap结构代替了L253/P5.5.Meeting Rooms II中的开一个巨大数组包含所有可能的time slot的方式，其后的通过map.put()找delta，通过count += val找running sum都是必须的，所以time slot -> delta -> occupied(presum)三步走过程和L253一模一样，只是数据结构从单纯的array变成了TreeMap**

time 1 2 3 4 5 6 7 8 9

delta +1 +1 0 -1 -1 0 +1 0 -1

occupied 1 2 2 1 0 0 1 1 0 (summing up `delta` changes over time)

class MyCalendarThree {

TreeMap<Integer, Integer> list;

public MyCalendarThree() {

list = new TreeMap<>();

}

public int book(int startTime, int endTime) {

// Increase the counter at the start time

list.put(startTime, list.getOrDefault(startTime, 0) + 1);

// Decrease the counter at the end time

list.put(endTime, list.getOrDefault(endTime, 0) - 1);

int k = 0;

// This will track the number of ongoing events

int count = 0;

for(int val : list.values()) {

// Found the max ongoing events count during scaning each time slot

k = Math.max(k, count += val);

}

return k;

}

}

/\*\*

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

\* MyCalendarThree obj = new MyCalendarThree();

\* int param\_1 = obj.book(startTime,endTime);

\*/

**Refer to**

<https://grandyang.com/leetcode/732/>

这道题是之前那两道题[My Calendar II](http://www.cnblogs.com/grandyang/p/7968035.html)，[My Calendar I](http://www.cnblogs.com/grandyang/p/7920253.html)的拓展，论坛上有人说这题不应该算是Hard类的，但实际上如果没有之前那两道题做铺垫，直接上这道其实还是还蛮有难度的。这道题博主在做完之前那道，再做这道一下子就做出来了，因为用的就是之前那道[My Calendar II](http://www.cnblogs.com/grandyang/p/7968035.html)的解法二，具体的讲解可以参见那道题，反正博主写完那道题再来做这道题就是秒解啊，参见代码如下：

class MyCalendarThree {

public:

MyCalendarThree() {}

int book(int start, int end) {

++freq[start];

--freq[end];

int cnt = 0, mx = 0;

for (auto f : freq) {

cnt += f.second;

mx = max(mx, cnt);

}

return mx;

}

private:

map<int, int> freq;

};

**Solution 2: Segment Tree  + Lazy Propagation (720 min)**

class MyCalendarThree {

SegmentTree segmentTree;

public MyCalendarThree() {

// 1e9 based on condition {0 <= startTime < endTime <= 10^9}

segmentTree = new SegmentTree(0, 1000000000);

}

public int book(int startTime, int endTime) {

segmentTree.update(startTime, endTime, 1);

return segmentTree.getMax();

}

}

class SegmentTree {

TreeNode root;

public SegmentTree(int start, int end) {

root = new TreeNode(start, end);

}

public void update(int start, int end, int val) {

TreeNode node = new TreeNode(start, end);

update(root, node, val);

}

public void update(TreeNode root, TreeNode node, int val) {

if(root == null) {

return;

}

// If current node's range lies completely in update query range

if(root.inside(node)) {

root.booked += val;

root.maxBooked += val;

}

// If current node's range overlaps with update range, follow the

// same approach as above simple update

if(root.intersect(node)) {

// Recur for left and right children

int mid = root.start + (root.end - root.start) / 2;

if(root.left == null) {

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

}

update(root.left, node, val);

if(root.right == null) {

root.right = new TreeNode(mid, root.end);

}

update(root.right, node, val);

// Update current node using results of left and right calls as

// post-order traversal

root.maxBooked = root.booked + Math.max(root.left.maxBooked, root.right.maxBooked);

}

}

public int getMax() {

return root.maxBooked;

}

// Find maximum booked for nums[i] (start <= i <= end) is not required,

// as we only need global maximum booked between [0, 10^9], just use

// getMax() method will get

public int query(int start, int end) {

return 0;

}

}

class TreeNode {

int start;

int end;

TreeNode left;

TreeNode right;

// How much number is added to this interval(node)

int booked;

// The maximum number in this interval(node)

int maxBooked;

public TreeNode(int start, int end) {

this.start = start;

this.end = end;

}

// Check if the current node's range fully inside a given node's range

public boolean inside(TreeNode node) {

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

return true;

}

return false;

}

// Check if the current node's range intersect a given node's range

public boolean intersect(TreeNode node) {

if(inside(node) || this.start >= node.end || this.end <= node.start) {

return false;

}

return true;

}

}

/\*\*

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

\* MyCalendarThree obj = new MyCalendarThree();

\* int param\_1 = obj.book(startTime,endTime);

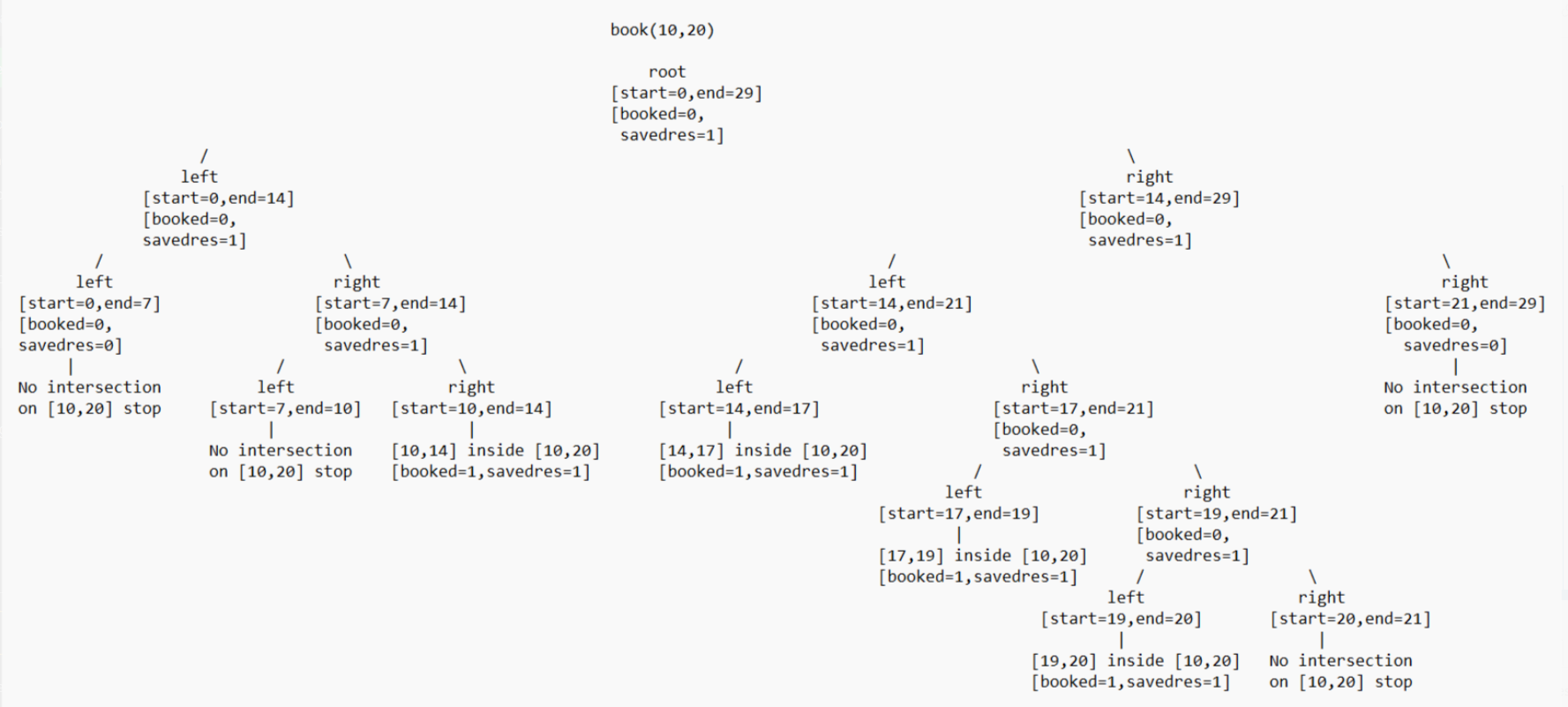
\*/

Time Complexity:

To query a range maximum, we process at most two nodes at every level and number of levels is O(logN). Thus the overall complexity is O(N\*log(len)), where len is the length of this segment (ie. 10^9 in this problem).

**与L307.Range Sum Query - Mutable提供的以int[] array = new int[4 \* N]的数组为segment tree载体的思路不同，这里的Segment Tree构建方式采用了Tree Node树节点的模式，通过每个节点存储的额外信息来完成query时反馈和update时更新的效果**

**Step by Step simulate book(10,20)**



**Text version**

book(10,20)

root

[start=0,end=29]

[booked=0,

savedres=1]

/ \

left right

[start=0,end=14] [start=14,end=29]

[booked=0, [booked=0,

savedres=1] savedres=1]

/ \ / \

left right left right

[start=0,end=7] [start=7,end=14] [start=14,end=21] [start=21,end=29]

[booked=0, [booked=0, [booked=0, [booked=0,

savedres=0] savedres=1] savedres=1] savedres=0]

| / \ / \ |

No intersection left right left right No intersection

on [10,20] stop [start=7,end=10] [start=10,end=14] [start=14,end=17] [start=17,end=21] on [10,20] stop

| | | [booked=0,

No intersection [10,14] inside [10,20] [14,17] inside [10,20] savedres=1]

on [10,20] stop [booked=1,savedres=1] [booked=1,savedres=1] / \

left right

[start=17,end=19] [start=19,end=21]

| [booked=0,

[17,19] inside [10,20] savedres=1]

[booked=1,savedres=1] / \

left right

[start=19,end=20] [start=20,end=21]

| |

[19,20] inside [10,20] No intersection

[booked=1,savedres=1] on [10,20] stop

**Refer to**

<https://leetcode.com/problems/my-calendar-iii/solutions/109568/java-solution-o-n-log-len-beats-100-segment-tree/>

This solution is basically a **segment tree** solution.

The qurey [MyCalendarThree.book(start](http://MyCalendarThree.book(start), end) can be treated as for(i = start; i < end; i++) nums[i] += 1. And the request becomes "find the maximum nums[i]".

class MyCalendarThree {

SegmentTree segmentTree;

public MyCalendarThree() {

segmentTree = new SegmentTree(0, 1000000000);

}

public int book(int start, int end) {

segmentTree.add(start, end, 1);

return segmentTree.getMax();

}

}

class SegmentTree {

TreeNode root;

public SegmentTree(int left, int right) {

root = new TreeNode(left, right);

}

public void add(int start, int end, int val) {

TreeNode event = new TreeNode(start, end);

add(root, event, val);

}

private void add(TreeNode root, TreeNode event, int val) {

if(root == null) {

return ;

}

/\*\*

\* If current node's range lies completely in update query range.

\*/

if(root.inside(event)) {

root.booked += val;

root.savedres += val;

}

/\*\*

\* If current node's range overlaps with update range, follow the same approach as above simple update.

\*/

if(root.intersect(event)) {

// Recur for left and right children.

int mid = (root.start + root.end) / 2;

if(root.left == null) {

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

}

add(root.left, event, val);

if(root.right == null) {

root.right = new TreeNode(mid, root.end);

}

add(root.right, event, val);

// Update current node using results of left and right calls.

root.savedres = Math.max(root.left.savedres, root.right.savedres) + root.booked;

}

}

public int getMax() {

return root.savedres;

}

/\*\*

\* find maximum for nums[i] (start <= i <= end) is not required.

\* so i did not implement it.

\*/

public int get(int start, int right) {return 0;}

class TreeNode {

int start, end;

TreeNode left = null, right = null;

/\*\*

\* How much number is added to this interval(node)

\*/

int booked = 0;

/\*\*

\* The maximum number in this interval(node).

\*/

int savedres = 0;

public TreeNode(int s, int t) {

this.start = s;

this.end = t;

}

public boolean inside(TreeNode b) {

if(b.start <= start && end <= b.end) {

return true;

}

return false;

}

public boolean intersect(TreeNode b) {

if(inside(b) || end <= b.start || b.end <= start) {

return false;

}

return true;

}

}

}

**Time Complexity:**

To query a range maximum, we process at most two nodes at every level and number of levels is O(logn)[1](http://www.geeksforgeeks.org/segment-tree-set-1-range-minimum-query/). Thus the overall complexity is O(n log(len)), where len is the length of this segment (ie. 10^9 in this problem).

**Segment Tree + Lazy Propagation**

**Refer to**

<https://algo.monster/liteproblems/732>

## Problem Description

In this problem, we are asked to implement a class MyCalendarThree which simulates a calendar that can track events and determine the maximum number of concurrent events at any given time. A k-booking means that k events overlap at some point. The book method will be called multiple times, each time with a startTime and an endTime of a new event (startTime, endTime), and it should return the maximum number of concurrent or overlapping events (k) that have been booked so far.

The core challenge comes from the fact that the number of book method calls can be quite large and that we must keep track of the maximum number of overlapping intervals efficiently after each call.

## Intuition

To address the problem efficiently, the solution leverages a data structure known as a segment tree. Segment trees are ideal for scenarios where we frequently perform range-based queries and modifications to an array or, in this case, ranges of time.

The intuition behind using a segment tree for this problem comes from the fact that we can treat each booking as an increment operation over a range of time. When we book a new event, we increase the overlapping count for that time interval.

The Segment Tree is built with methods for modifying (adding a booking) and querying (checking the maximum number of bookings) over a range. It is initialized to cover the entire range of possible bookings (from 1 to 10^9, as given in the Node initial range). Each node in the tree represents a range of time with the range's left boundary (l), right boundary (r), and the maximum bookings (v) in that range.

When a new booking is made:

1. We use the modify function to increment the overlap count for every relevant segment in the tree that the event passes through.
2. While making updates, we apply a "lazy propagation" strategy (pushdown method) to avoid unnecessary updates until required, which helps to keep the operation efficient.
3. After modifying the tree, we use the query function to find the maximum overlap (k) by examining the relevant segments of the tree.

By the end of the book method call, we have the updated segment tree reflecting all the bookings and can return the overall maximum overlap. This method allows us to maintain and query the bookings in logarithmic time complexity in relation to the depth of the segment tree, which is much faster than simple linear checking of all the intervals.

## Solution Approach

The solution approach entails a few essential components: a dynamic segment tree structure, lazy propagation for efficient updates, and methods to modify and query the tree. Below is a step-by-step breakdown of each part of the implementation:

### Node Class

The Node class represents each node in the segment tree. Each node has potential children (left, right), boundaries for the segment it represents (l, r), a midpoint (mid) for splitting the segment, a value (v) to store the number of bookings in the segment, and an additional value (add) that helps in lazy updates.

### SegmentTree Class

Here, we define the segment tree itself with a root node covering the entire range of possible values. The class includes functions that operate on the tree:

* **modify(l, r, v, node)**: This recursive function is called whenever a booking is made. It increments the booking count in each relevant segment by v. If the entire segment lies within the booking interval, it updates the node's value and the lazy value (used for propagation). If only parts of the node's segment are affected, it calls pushdown to ensure children nodes are created and/or updated, and then recurses into them.
* **query(l, r, node)**: This function finds the maximum booking count in the range [l, r]. It uses the same segment-matching logic as modify to drill down into the tree. When a node is fully within the range, its value is used directly. Otherwise, the function recurses into relevant children nodes and takes the maximum value found.
* **pushup(node)**: This is a helper function to update a node's value based on its children's values.
* **pushdown(node)**: Before further recursions or queries into child nodes, pushdown propagates the add value of a node to its children nodes (performing actual updates on them), which is key to the lazy update mechanism. This ensures updates are made only when they are necessary.

### MyCalendarThree Class

This class uses an instance of SegmentTree.

* **init()**: Initializes a new SegmentTree.
* **book(start, end)**: Public method called to book an event. It modifies the tree by incrementing the booking count in the range [start + 1, end] because we are using 1-based indices. Then it queries the entire range of possible bookings to find and return the maximum k (booking count).

By utilizing these methods and the dynamic, lazy segment tree, the MyCalendarThree class can efficiently track the maximum booking overlap after each event is booked.

### Example Walkthrough

Let's illustrate the solution approach with a simple example:

Suppose we make three booking calls with the following intervals: (10, 20), (15, 25), and (20, 30). We will walk through how the algorithm handles these bookings.

* The MyCalendarThree class is initialized, and an instance of SegmentTree is created. The segment tree's root node covers the entire possible booking range.
* The first booking is made with the book call: book(10, 20).

1. modify is called on the segment tree with the range [10, 20).
2. Since this is the first booking, the segment tree nodes that represent this range are incremented by 1, and the lazy propagation mechanism ensures that this update is done efficiently.
3. The query function is then called to find the maximum booking so far, which, in this case, is only 1.

* The second booking is made with the call: book(15, 25).

1. The modify function is called for the range [15, 25).
2. The segment tree is updated to reflect this booking. Since the range [15, 20) overlaps with the first booking, the relevant nodes will have their count incremented accordingly.
3. Lazy propagation ensures that these updates are made only where necessary and propagate accurately to child nodes.
4. A query returns that the maximum booking is now 2, representing the overlap between the first and second bookings.

* Lastly, we call book(20, 30).

1. modify updates the segment tree for the range [20, 30).
2. This time, there is no overlap with the previous bookings, so it only affects the counts for the relevant time segment from [20, 30).
3. A query still returns 2 since there is no change to the maximum booking overlap—it remains at the intersection of the first two bookings.

At the end of these bookings, MyCalendarThree can answer that the maximum number of concurrent events (k) is 2, meaning that at some point, there were two events happening at the same time. The implementation of the segment tree, particularly the lazy propagation, allowed us to efficiently manage and query the overlaps even as more bookings were added.

class Node {

Node left; // left child in the segment tree

Node right; // right child in the segment tree

int start; // start of the range

int end; // end of the range

int mid; // mid-point of the range

int value; // value of the node, carrying additional information (e.g., count of events)

int lazy; // lazy propagation value to apply to children

// Constructor for creating a new Node with a given range

public Node(int start, int end) {

this.start = start;

this.end = end;

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

}

}

class SegmentTree {

// Class representing a segment tree with lazy propagation

private Node root = new Node(1, (int) 1e9 + 1); // create a root node for the segment tree

// Empty constructor for SegmentTree

public SegmentTree() {

}

// Public method to add a value over a specified range

public void modify(int start, int end, int value) {

modify(start, end, value, root);

}

// Helper method to recursively add a value over a specified range

private void modify(int start, int end, int value, Node node) {

if (start > end) {

return; // invalid range, return early

}

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

// Current segment is fully covered by [start, end]

node.value += value;

node.lazy += value; // Set lazy value for further propagation

return;

}

pushdown(node); // Propagate lazy values

if (start <= node.mid) {

modify(start, end, value, node.left); // Modify left child

}

if (end > node.mid) {

modify(start, end, value, node.right); // Modify right child

}

pushup(node); // Update current node based on children

}

// Public method to query the maximum value over a specified range

public int query(int start, int end) {

return query(start, end, root);

}

// Helper method to get the maximum value over a specified range

private int query(int start, int end, Node node) {

if (start > end) {

return 0; // invalid range, return early

}

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

// Current segment is fully covered by [start, end]

return node.value;

}

pushdown(node); // Propagate lazy values

int maxVal = 0;

if (start <= node.mid) {

maxVal = Math.max(maxVal, query(start, end, node.left)); // Query left child

}

if (end > node.mid) {

maxVal = Math.max(maxVal, query(start, end, node.right)); // Query right child

}

return maxVal;

}

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

private void pushup(Node node) {

node.value = Math.max(node.left.value, node.right.value);

}

// Method to propagate lazy values to the node's children

private void pushdown(Node node) {

// Initialize children if necessary

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);

}

// Propagate lazy values to children if any

if (node.lazy != 0) {

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

left.lazy += node.lazy;

right.lazy += node.lazy;

left.value += node.lazy;

right.value += node.lazy;

node.lazy = 0; // Reset lazy value after propagation

}

}

}

class MyCalendarThree {

// Class representing a calendar that tracks the maximum number of simultaneous events

private SegmentTree tree = new SegmentTree();

// Empty constructor for MyCalendarThree

public MyCalendarThree() {

}

// Public method to book an event and get the maximum number of simultaneous events

public int book(int start, int end) {

tree.modify(start + 1, end, 1); // Add event to the segment tree

return tree.query(1, (int) 1e9 + 1); // Query the entire range for the maximum

}

}

// Example of how to use the MyCalendarThree class:

// MyCalendarThree obj = new MyCalendarThree();

// int param\_1 = obj.book(start, end);