<https://leetcode.com/problems/meeting-rooms-iii/description/>

You are given an integer n. There are n rooms numbered from 0 to n - 1.

You are given a 2D integer array meetings where meetings[i] = [starti, endi] means that a meeting will be held during the **half-closed** time interval [starti, endi). All the values of starti are **unique**.

Meetings are allocated to rooms in the following manner:

1. Each meeting will take place in the unused room with the **lowest**
2. If there are no available rooms, the meeting will be delayed until a room becomes free. The delayed meeting should have the **same**
3. When a room becomes unused, meetings that have an earlier original **start**

Return**the number of the room that held the most meetings.**If there are multiple rooms, return**the room with the lowest number.**

A **half-closed interval** [a, b) is the interval between a and b **including** a and **not including** b.

**Example 1:**

**Input:** n = 2, meetings = [[0,10],[1,5],[2,7],[3,4]]

**Output:** 0

**Explanation:**

- At time 0, both rooms are not being used. The first meeting starts in room 0.

- At time 1, only room 1 is not being used. The second meeting starts in room 1.

- At time 2, both rooms are being used. The third meeting is delayed.

- At time 3, both rooms are being used. The fourth meeting is delayed.

- At time 5, the meeting in room 1 finishes. The third meeting starts in room 1 for the time period [5,10).

- At time 10, the meetings in both rooms finish. The fourth meeting starts in room 0 for the time period [10,11).

Both rooms 0 and 1 held 2 meetings, so we return 0.

**Example 2:**

**Input:** n = 3, meetings = [[1,20],[2,10],[3,5],[4,9],[6,8]]

**Output:** 1

**Explanation:**

- At time 1, all three rooms are not being used. The first meeting starts in room 0.

- At time 2, rooms 1 and 2 are not being used. The second meeting starts in room 1.

- At time 3, only room 2 is not being used. The third meeting starts in room 2.

- At time 4, all three rooms are being used. The fourth meeting is delayed.

- At time 5, the meeting in room 2 finishes. The fourth meeting starts in room 2 for the time period [5,10).

- At time 6, all three rooms are being used. The fifth meeting is delayed.

- At time 10, the meetings in rooms 1 and 2 finish. The fifth meeting starts in room 1 for the time period [10,12).

Room 0 held 1 meeting while rooms 1 and 2 each held 2 meetings, so we return 1.

**Constraints:**

* 1 <= n <= 100
* 1 <= meetings.length <= 10^5
* meetings[i].length == 2
* 0 <= starti < endi <= 5 \* 10^5
* All the values of starti  **unique**

**Attempt 1: 2023-12-17**

**Solution 1: Two Heaps (720 min)**

**Wrong Solution (80/82)**

class Solution {

    public int mostBooked(int n, int[][] meetings) {

        int[] count = new int[n];

        // empty\_rooms minPQ -> {room id}

        PriorityQueue<Integer> empty\_rooms = new PriorityQueue<>();

        // occupied\_rooms minPQ -> {room id, meeting end time}

        PriorityQueue<int[]> occupied\_rooms = new PriorityQueue<>((a, b) -> a[1] == b[1] ? a[0] - b[0] : a[1] - b[1]);

        for(int empty\_room\_id = 0; empty\_room\_id < n; empty\_room\_id++) {

            empty\_rooms.offer(empty\_room\_id);

        }

        // Sort meetings intervals with meeting start time

        Arrays.sort(meetings, (a, b) -> a[0] - b[0]);

        for(int i = 0; i < meetings.length; i++) {

            int start = meetings[i][0];

            int end = meetings[i][1];

            int duration = end - start;

            // Free up rooms that have finished their meeting before the start of the current meeting

            // Check current meeting start time, if no overlap with meeting stored on occupied\_rooms

            // minPQ peek position, then the meeting stored on peek position treat as ended meeting,

            // pop out ended meeting from occupied\_rooms minPQ and release the room for current meeting,

            // add the released room into empty\_rooms minPQ

            while(!occupied\_rooms.isEmpty() && start >= occupied\_rooms.peek()[1]) {

                int release\_room\_id = occupied\_rooms.poll()[0];

                empty\_rooms.offer(release\_room\_id);

            }

            // Choose a room for the current meeting

            if(!empty\_rooms.isEmpty()) {

                // If there is an empty room available, use the one with the smallest id

                int empty\_room\_id = empty\_rooms.poll();

                // Add the meeting to the occupied\_rooms minPQ with the meeting end time and the room id

                occupied\_rooms.offer(new int[]{empty\_room\_id, end});

                count[empty\_room\_id]++;

            } else {

                // Here is the error out logic, test out by attachment input

                // Update current meeting interval to new start and end time, start time right at most

                // recent future ending meeting (stored at occupied\_rooms minPQ peek position) end time,

                // end time based on start time plus current meeting duration

                meetings[i][0] = occupied\_rooms.peek()[1];

                meetings[i][1] = meetings[i][0] + duration;

                // Backward one step for next iteration(i++ in for loop) still keep on try with same meeting

                i--;

            }

        }

        // Return the number of the room that held the most meetings. If there are multiple rooms,

        // return the room with the lowest number

        int result = -1;

        int max = 0;

        for(int i = n - 1; i >= 0; i--) {

            if(max <= count[i]) {

                max = count[i];

                result = i;

            }

        }

        return result;

    }

}

**Error test out by**



**Correct Solution**

class Solution {

    public int mostBooked(int n, int[][] meetings) {

        int[] count = new int[n];

        // empty\_rooms minPQ -> {room id}

        PriorityQueue<Integer> empty\_rooms = new PriorityQueue<>();

        // occupied\_rooms minPQ -> {room id, meeting end time}

        PriorityQueue<int[]> occupied\_rooms = new PriorityQueue<>((a, b) -> a[1] == b[1] ? a[0] - b[0] : a[1] - b[1]);

        for(int empty\_room\_id = 0; empty\_room\_id < n; empty\_room\_id++) {

            empty\_rooms.offer(empty\_room\_id);

        }

        // Sort meetings intervals with meeting start time

        Arrays.sort(meetings, (a, b) -> a[0] - b[0]);

        for(int i = 0; i < meetings.length; i++) {

            int start = meetings[i][0];

            int end = meetings[i][1];

            int duration = end - start;

            // Free up rooms that have finished their meeting before the start of the current meeting

            // Check current meeting start time, if no overlap with meeting stored on occupied\_rooms

            // minPQ peek position, then the meeting stored on peek position treat as ended meeting,

            // pop out ended meeting from occupied\_rooms minPQ and release the room for current meeting,

            // add the released room into empty\_rooms minPQ

            while(!occupied\_rooms.isEmpty() && start >= occupied\_rooms.peek()[1]) {

                int release\_room\_id = occupied\_rooms.poll()[0];

                empty\_rooms.offer(release\_room\_id);

            }

            int empty\_room\_id;

            // Choose a room for the current meeting

            if(!empty\_rooms.isEmpty()) {

                // If there is an empty room available, use the one with the smallest id

                empty\_room\_id = empty\_rooms.poll();

                // Add the meeting to the occupied\_rooms minPQ with the meeting end time and the room id

                occupied\_rooms.offer(new int[]{empty\_room\_id, end});

                //count[empty\_room\_id]++;

            } else {

                // Update current meeting interval to new start and end time, start time right at most

                // recent future ending meeting (stored at occupied\_rooms minPQ peek position) end time,

                // end time based on start time plus current meeting duration

                //meetings[i][0] = occupied\_rooms.peek()[1];

                //meetings[i][1] = meetings[i][0] + duration;

                // Backward one step for next iteration(i++ in for loop) still keep on try with same meeting

                //i--;

                // If no idle room is available, wait for the next room to be free

                // Schedule the meeting in this room, adjusting the meeting length accordingly

                // Note: no trick, no backward one step, just replace occupied\_rooms minPQ peek meeting

                // with current unsettled meeting (only care about new end time)

                int[] next\_room\_to\_be\_free = occupied\_rooms.poll();

                empty\_room\_id = next\_room\_to\_be\_free[0];

                int new\_end = next\_room\_to\_be\_free[1] + duration;

                occupied\_rooms.offer(new int[]{empty\_room\_id, new\_end});

            }

            count[empty\_room\_id]++;

        }

        // Return the number of the room that held the most meetings. If there are multiple rooms,

        // return the room with the lowest number

        int result = -1;

        int max = 0;

        for(int i = n - 1; i >= 0; i--) {

            if(max <= count[i]) {

                max = count[i];

                result = i;

            }

        }

        return result;

    }

}

Time Complexity: O(M\*logM) + O(M) + O(M\*logN) + O(M\*logN), M is number of meetings, N is number of rooms

Space Complexity: O(N)

**Step by Step**

e.g

Input: n = 3, meetings = [[1,20],[2,10],[3,5],[4,9],[6,8]]

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

Time = 1,2,3

At time 1, all three rooms are not being used. The first meeting starts in room 0.

At time 2, rooms 1 and 2 are not being used. The second meeting starts in room 1.

At time 3, only room 2 is not being used. The third meeting starts in room 2.

empty\_rooms: {empty\_room\_id}, minPQ sort by room id

-----------------------------

0

1

2

-----------------------------

occupied\_rooms: {room\_id, meeting end time}, minPQ sort by meeting end time, if same meeting end time sort by room id

-----------------------------

{2,5}

{1,10}

{0,20}

-----------------------------

count = {1,1,1} -> each room hold 1 meeting now

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

Time = 4,5

At time 4, all three rooms are being used. The fourth meeting is delayed.

empty\_rooms.isEmpty() == true, no room available, delay

occupied\_rooms: {room\_id, meeting end time}, minPQ sort by meeting end time, if same meeting end time sort by room id

-----------------------------

{2,5} -> poll out from occupied\_rooms and room id = 2 will be used for hold current unsettled meeting

{1,10}

{0,20}

-----------------------------

current meeting interval [4,9), duration = 9 - 4 = 5

new meeting interval = [occupied\_rooms.peek()[1], occupied\_rooms.peek()[1] + duration) = [5,10), but we only care about

new end time '10', which will replace the occupied\_rooms minPQ peek meeting with current unsettled meeting

occupied\_rooms: {room\_id, meeting end time}, minPQ sort by meeting end time, if same meeting end time sort by room id

-----------------------------

{1,10} -> same meeting end time ([1,10) and [5,10)) but sort by room id, so room id = 1 at empty\_rooms peek

{2,10} -> updated fourth meeting [4,9) => [5,10) to room id = 2

{0,20}

-----------------------------

count = {1,1,2} -> room id = 2 hold 2 meetings now

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

Time = 6

At time 6, all three rooms are being used. The fifth meeting is delayed.

empty\_rooms.isEmpty() == true, no room available, delay

occupied\_rooms: {room\_id, meeting end time}, minPQ sort by meeting end time, if same meeting end time sort by room id

-----------------------------

{1,10} -> poll out from occupied\_rooms and room id = 1 will be used for hold current unsettled meeting

{2,10}

{0,20}

-----------------------------

current meeting interval [6,8), duration = 8 - 6 = 2

new meeting interval = [occupied\_rooms.peek()[1], occupied\_rooms.peek()[1] + duration) = [10,12), but we only care about

new end time '12', which will replace the occupied\_rooms minPQ peek meeting with current unsettled meeting

occupied\_rooms: {room\_id, meeting end time}, minPQ sort by meeting end time, if same meeting end time sort by room id

-----------------------------

{2,10}

{1,12} -> updated fifth meeting [6,8) => [10,12) to room id = 1

{0,20}

-----------------------------

count = {1,2,2} -> room id = 1 hold 2 meetings now

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

So finally since both room id = 1 and 2 hold 2 meetings, based on:

Return the number of the room that held the most meetings. If there are multiple rooms, return the room with the lowest number.

room id = 1 is the answer

**Refer to**

<https://leetcode.com/problems/meeting-rooms-iii/solutions/2535735/detailed-explanation-simulation-using-min-heaps-c-clean-code/>

**Intuition :**

Idea here is simply to simulate the meetings according to the rules given in problem statement.

Let us look at each rule and try to build logic using it:

**Rule 1 :** Allocate room to a meet only when it is empty and if multiple rooms are empty, then pick one with smallest number.

So here we can use a min heap empty to keep track of all empty rooms.

And to get the room with smallest room number.

**Rule 2 :** If all rooms are occupied, then wait till some meeting ends, and then allocate current meeting to emptied room.

Here, we can use another min heap occupied that keeps track of finishing/ending time of a meeting along with room number.

Using this, we can get a room as early as possible and with smallest room number (in case multiple rooms are emptied at same time).

**Rule 3 :** Allocated room to meetings based on their original start time.

This means allocate room to meetings on 1 basis.

i.e if one room is empty and multiple meets are to be allocated, then give priority to the meeting that has least original start time.

This can be done easily by sorting meetings list, and then start allocating rooms from i=0.

Now that we have all rules ready along with implementation, we can start simulation.

* First we will sort meetings list and also add all rooms to empty min heap.
* Then, we will start allocating rooms to each meet one by one.
* So, first of all, if some meetings are finished till now, then move rooms from occupied list to empty list.
* Next if we have a room empty, then allocate meet to it, add ending time and room number to occupied list.
* But, if at current start time of meet, no room is empty, then get the room which is ending earliest from now.
* Then update finish/end time for current meet and room number, and add this to occupied list
* Also, for each meeting, increment count of meets allocated to that room accordingly.
* Continue this process till all meetings are allocated to some room.
* In the end, get the smallest room with most meets allocated.

Dry run over a sample test case :

  meetings = [[1,20],[2,10],[3,5],[4,9],[6,8]], n = 3  -> meetings is sorted here

  Initially, empty = {0, 1, 2} and occupied = { }, meetsCount = [0, 0, 0]

  meet 0 : start = 1, end = 20, duration = 20 - 1 = 19

      -> No meet is finishing in occupied rooms at current start time

      -> roomNum = 0, meetCount[0] = 1, allocate to room 0

      -> empty = {1, 2} , occupied = { (20, 0) } , meetsCount = [1, 0, 0]

  meet 1 : start = 2, end = 10, duration = 10 - 2 = 8

      -> No meet is finishing in occupied rooms at current start time

      -> roomNum = 1, meetCount[1] = 1, allocate to room 1

      -> empty = {2} , occupied = { (10, 1), (20, 0) } , meetsCount = [1, 1, 0]

  meet 2 : start = 3, end = 5, duration = 5 - 3 = 2

      -> No meet is finishing in occupied rooms at current start time

      -> roomNum = 2, meetCount[2] = 1, allocate to room 2

      -> empty = { } , occupied = { (5, 2), (10, 1), (20, 0) } , meetsCount = [1, 1, 1]

  meet 3 : start = 4, end = 9, duration = 9 - 4 = 5

      -> No meet is finishing in occupied rooms at current start time, also no room is empty

      -> Wait till time=5, when room 2 gets empty. Then we can use that room.

      -> occupied = { (10, 1), (20, 0) }, since we have popped top element (5, 2) from heap [Intermediate step]

      -> endTime = time + duration = 5 + 5 = 10

      -> roomNum = 2, meetCount[2] = 2, allocate to room 2

      -> empty = { } , occupied = { (10, 1), (10, 2), (20, 0) } , meetsCount = [1, 1, 2]

  meet 4 : start = 6, end = 8, duration = 8 - 6 = 2

      -> No meet is finishing in occupied rooms at current start time, also no room is empty

      -> Wait till time=10, when room 1 gets empty. Then we can use that room.

      -> occupied = { (10, 2), (20, 0) }, since we have popped top element (10, 1) from heap [Intermediate step]

      -> endTime = time + duration = 10 + 2 = 12

      -> roomNum = 1, meetCount[1] = 2, allocate to room 1

      -> empty = { } , occupied = { (10, 2), (12, 1), (20, 0) } , meetsCount = [1, 2, 2]

  - So we have allocated rooms to all meetings.

  - Smallest room number which held maximum meet i.e max( meetsCount[i] ) = 1

class Solution {

    public:

    int mostBooked(int n, vector<vector<int>>& meetings) {

        // Sort the meetings list by [startTime\_i]

        sort(begin(meetings), end(meetings));

        // Count of meets that are held in total inside each room from 0 to n-1

        vector<int> meetsCount(n, 0);

        // Min Heap to store ending time of meet held in some room.

        // This will give room number with smallest ending time

        priority\_queue<pair<long long, int>, vector<pair<long long, int>>, greater<pair<long long, int>>> occupied;

        // Min Heap to store empty/available room number

        // This will give smallest room number that is empty

        priority\_queue<int, vector<int>, greater<int>> empty;

        // Initially all n rooms are empty

        for(int i=0; i<n; i++) empty.push(i);

        // Iterate over all sorted meetings, and try to allocate a room that is empty

        // And if not empty then wait till it gets available

        for(auto& meet : meetings) {

            // Clear all room in which meeting is ended

            // i.e ending time of meet is less than or eq to start time of current meet

            long long currTime = meet[0];

            while(occupied.size()) {

                if(currTime < occupied.top().first) break;

                auto [time, room] = occupied.top(); occupied.pop();

                empty.push(room);

            }

            // If room available then allocate that room to current meeting

            int roomNum = -1;

            if(empty.size()) {

                roomNum = empty.top(); empty.pop();

                occupied.push({ meet[1], roomNum });

            }

            // If no room is available, then wait till a meet ends and allocate that room

            else if(occupied.size()) {

                auto [time, room] = occupied.top(); occupied.pop();

                roomNum = room;

                long long endTime = time + meet[1] - meet[0];

                occupied.push({endTime, roomNum});

            }

            // Once room is allocated for meet, increment count of meets held in that room

            if(roomNum != -1) meetsCount[roomNum]++;

        }

        // Get the index i.e room number with maximum meetings held

        int idx = -1, maxVal = 0;

        for(int i=0; i<n; i++) {

            if(meetsCount[i] > maxVal) {

                idx = i;

                maxVal = meetsCount[i];

            }

        }

        return idx;

    }

};

**Complexity :**

* Time : O(NlogN + M (logN + logM))
* Space : O(N + M)
  + M is size of meetings array
  + N is number of rooms available

**Refer to**

<https://algo.monster/liteproblems/2402>

**Problem Description**

In the given problem, we have n rooms where meetings are scheduled to take place. Our goal is to identify which room will have hosted the most number of meetings by the end of all meetings. Each meeting is represented as a pair of integers indicating its start and end times. A key point to note is that these meetings' start times are unique, meaning no two meetings will start at exactly the same time. The meetings follow certain allocation rules to rooms:

1. Each meeting is scheduled in the lowest-numbered room available.
2. If all rooms are busy, the meeting will be delayed until a room is free. The meeting will retain its original duration even when delayed.
3. Rooms are allocated to meetings based on their original start times, with earlier meetings getting priority.

Meetings are represented by half-closed intervals [start, end), which include the start time but not the end time. The objective is to find the room number that has the most meetings by the end. And if there's a tie, we are interested in the room with the lowest number.

**Intuition**

The solution is based on sorting and using heaps (priority queues). Here's the intuition step by step:

**Sorting the Meetings:** We first sort the meetings by their start time. This ensures that we always consider meetings in the order they are meant to start, which is important for fairness and aligns with the problem constraints.

**Using Two Heaps:** We maintain two heaps (priority queues):

* idle: A min-heap of available rooms.
* busy: A min-heap of rooms currently occupied by meetings, with their expected free time as the key.

By default, all n rooms are added to idle heap to indicate they are available.

**Processing Meetings:** We iterate through each meeting after sorting. For each meeting:

* We first free up rooms in the busy heap where meetings have ended—i.e., rooms with free time less than or equal to the current meeting's start time.
* We then check if there's an available room in the idle heap. If there is, we take the one at the top (lowest number) and schedule the current meeting there.
* If there isn't an available room, we look in the busy heap for the room that will become available first, delay the current meeting until this room is free, and reschedule with the same duration.

**Counting Meetings per Room:** We use an array cnt to keep track of the number of meetings in each room. Every time a meeting is scheduled in a room, we increment the count for that room.

**Finding the Room with Most Meetings:** After scheduling all meetings, we find the room that has the highest count in cnt. In case of a tie, the iteration ensures that the room with the lowest number prevails due to its earlier index in array.

The function ends by returning the index (room number) that had the most meetings.

The Python code implements this intuition using the heapq module, which provides an implementation for min-heaps in Python.

**Solution Approach**

The solution implements an efficient approach using sorting and heap data structures to manage the rooms and their availability. Here's a detailed walkthrough:

**Sort the Meetings:** The meetings are sorted by their start time to ensure we process them in the correct order.

meetings.sort()

**Initialization:**

* Create an idle heap, populated with room numbers (0 through n-1). This represents all available rooms.

idle = list(range(n))

heapify(idle)

* Create a busy heap to keep track of rooms that are currently in use, along with when they will become free.
* Have a counter cnt to count the number of meetings for each room.

busy = []

cnt = [0] \* n

**Iterating Over Meetings:** Loop through each meeting and make decisions based on room availability.

for s, e in meetings:

**Freeing Up Rooms:** Before scheduling a meeting, check if any meetings have ended, and if so, move those rooms from the busy heap back to the idle heap.

while busy and busy[0][0] <= s:

heappush(idle, heappop(busy)[1])

**Scheduling Meetings:**

* If there are available rooms (idle is not empty), pop the lowest-numbered room and schedule the current meeting in that room. Then push the room along with the new end time onto the busy heap and increment the corresponding room count.

if idle:

i = heappop(idle)

cnt[i] += 1

heappush(busy, (e, i))

* If no rooms are available, pull the room from the busy heap that will become available first. Delay the current meeting until this room is free, maintain its duration, and push it back onto the busy heap with the updated end time. Also, increment the corresponding room count.

else:

a, i = heappop(busy)

cnt[i] += 1

heappush(busy, (a + e - s, i))

**Finding the Room with Most Meetings:** After processing all meetings, iterate over the cnt list to find the room number that has the highest count (most meetings). In the case of a tie, the loop ensures that the earliest room number is chosen.

ans = 0

for i, v in enumerate(cnt):

if cnt[ans] < v:

ans = i

**Return the Result:** The variable ans holds the room number that hosted the most meetings, and we return it from the function.

return ans

This approach cleverly uses sorting to handle meetings in chronological order and heaps to manage room availability efficiently. The heap data structure allows us to access the room that will become free soonest in O(log n) time, which is much faster than linear search. The result is a well-considered solution that efficiently processes meetings and room allocations, satisfying all given constraints.

**Example Walkthrough**

Let's consider an example to illustrate the solution approach step by step.

Assuming we have 3 rooms (n = 3) and four meetings with the following start and end times meetings = [(1, 4), (2, 3), (3, 5), (5, 6)].

1. **Sort the Meetings:**

After sorting based on start times, meetings remains the same as the input is already sorted.

1. **Initialization:**

The idle heap is initialized with [0, 1, 2], representing three available rooms.

The busy heap is empty initially, as no meetings are taking place.

The cnt list is initialized to [0, 0, 0] to track the number of meetings in each room.

1. **First Meeting** (1, 4):

The idle heap has [0, 1, 2], we take out room 0 for our meeting and update the busy heap to [(4, 0)].

The cnt array is updated to [1, 0, 0].

1. **Second Meeting** (2, 3):

Still, before scheduling, we check for free rooms. None are free.

The idle heap has [1, 2]. We use room 1 and now the busy heap is [(3, 1), (4, 0)].

The cnt array is updated to [1, 1, 0].

1. **Third Meeting** (3, 5):

Before this meeting, room 1 becomes free. So the busy heap becomes [(4, 0)] and idle heap is [1, 2].

Room 1 is reused, and now busy heap is [(4, 0), (5, 1)].

The cnt array is updated to [1, 2, 0].

1. **Fourth Meeting** (5, 6):

Before this meeting, room 0 becomes free. busy now is [(5, 1)] and idle is [0, 2].

Meeting takes place in room 0, and busy heap is [(6, 0), (5, 1)].

The cnt array is updated to [2, 2, 0].

1. **Finding the Room with Most Meetings:**

After processing all meetings, we observe that the cnt array is [2, 2, 0].

Rooms 0 and 1 both hosted two meetings. Due to the iteration, the result will be the lower-numbered room in case of a tie; thus, room 0 is the answer.

1. **Return the Result:**

The function would return 0 as the room that hosted the most meetings in this example.

class Solution {

    // Function to find the room that gets booked the most

    public int mostBooked(int n, int[][] meetings) {

        // Sort all the meetings by their start time

        Arrays.sort(meetings, (a, b) -> a[0] - b[0]);

        // Priority queue to keep track of busy rooms

        // Sort by end time; if end times are same, sort by room id

        PriorityQueue<int[]> busyRooms = new PriorityQueue<>(

                (a, b) -> a[0] == b[0] ? a[1] - b[1] : a[0] - b[0]);

        // Priority queue for idle rooms, sorted by room id

        PriorityQueue<Integer> idleRooms = new PriorityQueue<>();

        // Initialize all rooms as idle

        for (int i = 0; i < n; i++) {

            idleRooms.offer(i);

        }

        // Counter for the number of meetings each room has

        int[] count = new int[n];

        // Iterate over all meetings

        for (int[] meeting : meetings) {

            int start = meeting[0], end = meeting[1];

            // Free up rooms that have finished their meeting before the start of the current meeting

            while (!busyRooms.isEmpty() && busyRooms.peek()[0] <= start) {

                idleRooms.offer(busyRooms.poll()[1]);

            }

            // Choose a room for the current meeting

            int roomId;

            if (!idleRooms.isEmpty()) {

                // If there is an idle room available, use the one with the smallest id

                roomId = idleRooms.poll();

                // Add the meeting to the busy queue with the new end time and the room id

                busyRooms.offer(new int[] {end, roomId});

            } else {

                // If no idle room is available, wait for the next room to be free

                int[] busyRoom = busyRooms.poll();

                roomId = busyRoom[1];

                // Schedule the meeting in this room, adjusting the meeting length accordingly

                busyRooms.offer(new int[] {busyRoom[0] + end - start, roomId});

            }

            // Increment the count of meetings for the chosen room

            count[roomId]++;

        }

        // Find the room with the maximum number of meetings booked

        int mostBookedRoom = 0;

        for (int i = 0; i < n; i++) {

            if (count[mostBookedRoom] < count[i]) {

                mostBookedRoom = i;

            }

        }

        // Return the room id with the highest booking count

        return mostBookedRoom;

    }

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the solution is determined by several factors:

1. Sorting the

 meetings list: This step has a complexity of O(M log M), where M is the number of meetings.

1. Iterating through

 meetings: Each meeting is processed once, adding an O(M) factor.

1. Maintaining the

 busy heap: Each meeting can lead to a push and pop operation. The heap operations take O(log N), where N is the number of rooms. In the worst case, all meetings require an operation, adding an O(M log N) factor.

1. Maintaining the

 idle heap: Similar to the busy heap, each operation can take O(log N). Since operations are conducted whenever a room becomes free or is used, this operation also contributes O(M log N) to the complexity.

The mentioned operations are all that are required, so the total time complexity is O(M log M) + O(M) + O(M log N) + O(M log N). Given that O(M log M) is the dominating factor, we can approximate the total time complexity to O(M log M).

**Space Complexity**

The space complexity of the solution is determined by the storage used:

1. Arrays

 cnt and idle: Both of these use O(N) space where N is the number of rooms.

1. Heap

 busy: In the worst case, the busy heap may store details of all N rooms, so it uses O(N) space.

1. Sorted

 meetings list: This list is sorted in-place, so it does not use extra space beyond the initial input list.

The total space complexity is therefore O(N), derived from the largest storage component used in the algorithm.