<https://www.lintcode.com/problem/919/>

Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] (si < ei), find the minimum number of conference rooms required.

**Example 1:**

Input: [[0, 30],[5, 10],[15, 20]]

Output: 2

**Example 2:**

Input: [[7,10],[2,4]]

Output: 1

NOTE: input types have been changed on April 15, 2019. Please reset to default code definition to get new method signature.

**Attempt 1: 2023-03-04**

**Solution 1: Sort respectively on interval 'start' and 'end' then check if over lapping with two pointers (60 min)**

/\*\*

\* Definition of Interval:

\* public class Interval {

\* int start, end;

\* Interval(int start, int end) {

\* this.start = start;

\* this.end = end;

\* }

\* }

\*/

public class Solution {

/\*\*

\* @param intervals: an array of meeting time intervals

\* @return: the minimum number of conference rooms required

\*/

public int minMeetingRooms(List<Interval> intervals) {

int size = intervals.size();

Integer[] starts = new Integer[size];

Integer[] ends = new Integer[size];

for(int i = 0; i < size; i++) {

starts[i] = intervals.get(i).start;

ends[i] = intervals.get(i).end;

}

Arrays.sort(starts, (a, b) -> a - b);

Arrays.sort(ends, (a, b) -> a - b);

int i = 0;

int j = 0;

// Variables to keep track of maximum number of rooms used

int usedRooms = 0;

while(i < size) {

// If there is a meeting ended by the time when another meeting

// (at 'start' index) starts, we can reuse the room which holding

// the ended meeting

if(starts[i] >= ends[j]) {

usedRooms--;

j++;

}

// We do this irrespective of whether a room frees up or not.

// If a room got free, then this usedRooms += 1 wouldn't have any effect.

// usedRooms would remain the same in that case.

// If no room was free, then this would increase usedRooms

usedRooms++;

i++;

}

return usedRooms;

}

}

Time Complexity:O(nlogn), sorting take nlogn time

Space Complexity:O(n)

**Refer to**

<https://www.lintcode.com/problem/919/solution/57831>

### 方法：有序化

**思路**

提供给我们的会议时间可以确定一天中所有事件的时间顺序。我们拿到了每个会议的开始和结束时间，这有助于我们定义此顺序。

根据会议的开始时间来安排会议有助于我们了解这些会议的自然顺序。然而，仅仅知道会议的开始时间，还不足以告诉我们会议的持续时间。我们还需要按照结束时间排序会议，因为一个“会议结束”事件告诉我们必然有对应的“会议开始”事件，更重要的是，“会议结束”事件可以告诉我们，一个之前被占用的会议室现在空闲了。

一个会议由其开始和结束时间定义。然而，在本算法中，我们需要 分别 处理开始时间和结束时间。这乍一听可能不太合理，毕竟开始和结束时间都是会议的一部分，如果我们将两个属性分离并分别处理，会议自身的身份就消失了。但是，这样做其实是可取的，因为：

当我们遇到“会议结束”事件时，意味着一些较早开始的会议已经结束。我们并不关心到底是哪个会议结束。我们所需要的只是 一些 会议结束,从而提供一个空房间。

**算法**

* 分别将开始时间和结束时间存进两个数组。
* 分别对开始时间和结束时间进行排序。请注意，这将打乱开始时间和结束时间的原始对应关系。它们将被分别处理。
* 考虑两个指针：s\_ptr 和 e\_ptr ，分别代表开始指针和结束指针。开始指针遍历每个会议，结束指针帮助我们跟踪会议是否结束。
* 当考虑 s\_ptr 指向的特定会议时，检查该开始时间是否大于 e\_ptr 指向的会议。若如此，则说明 s\_ptr 开始时，已经有会议结束。于是我们可以重用房间。否则，我们就需要开新房间。
* 若有会议结束，换而言之，start[s\_ptr] >= end[e\_ptr] ，则自增 e\_ptr 。
* 重复这一过程，直到 s\_ptr 处理完所有会议。

public class Solution {

public int minMeetingRooms(List<Interval> intervals) {

// Check for the base case. If there are no intervals, return 0

if (intervals.size() == 0) {

return 0;

}

Integer[] start = new Integer[intervals.size()];

Integer[] end = new Integer[intervals.size()];

for (int i = 0; i < intervals.size(); i++) {

start[i] = intervals.get(i).start;

end[i] = intervals.get(i).end;

}

// Sort the intervals by end time

Arrays.sort(

end,

new Comparator<Integer>() {

public int compare(Integer a, Integer b) {

return a - b;

}

});

// Sort the intervals by start time

Arrays.sort(

start,

new Comparator<Integer>() {

public int compare(Integer a, Integer b) {

return a - b;

}

});

// The two pointers in the algorithm: e\_ptr and s\_ptr.

int startPointer = 0, endPointer = 0;

// Variables to keep track of maximum number of rooms used.

int usedRooms = 0;

// Iterate over intervals.

while (startPointer < intervals.size()) {

// If there is a meeting that has ended by the time the meeting at `start\_pointer` starts

if (start[startPointer] >= end[endPointer]) {

usedRooms -= 1;

endPointer += 1;

}

// We do this irrespective of whether a room frees up or not.

// If a room got free, then this used\_rooms += 1 wouldn't have any effect. used\_rooms would

// remain the same in that case. If no room was free, then this would increase used\_rooms

usedRooms += 1;

startPointer += 1;

}

return usedRooms;

}

}

**复杂度分析**

* 时间复杂度: *O*(*N*log*N*)。我们所做的只是将 开始时间和 结束时间两个数组分别进行排序。每个数组有*N*个元素，因为有*N*个时间间隔。
* 空间复杂度:*O*(*N*)。我们建立了两个*N*大小的数组。分别用于记录会议的开始时间和结束时间。

**Solution 2: Sort all intervals based on 'start' and push to Priority Queue based on interval 'end' (30 min)**

/\*\*

\* Definition of Interval:

\* public class Interval {

\* int start, end;

\* Interval(int start, int end) {

\* this.start = start;

\* this.end = end;

\* }

\* }

\*/

public class Solution {

/\*\*

\* @param intervals: an array of meeting time intervals

\* @return: the minimum number of conference rooms required

\*/

public int minMeetingRooms(List<Interval> intervals) {

if(intervals.size() == 0) {

return 0;

}

PriorityQueue<Integer> minPQ = new PriorityQueue<Integer>((a, b) -> a - b);

Collections.sort(intervals, (a, b) -> a.start - b.start);

minPQ.offer(intervals.get(0).end);

for(int i = 1; i < intervals.size(); i++) {

if(minPQ.peek() <= intervals.get(i).start) {

minPQ.poll();

}

minPQ.offer(intervals.get(i).end);

}

return minPQ.size();

}

}

Time Complexity:O(nlogn), sorting take nlogn time

Space Complexity:O(n)

**Refer to**

<https://www.lintcode.com/problem/919/solution/57828>

### 方法：优先队列

我们无法按任意顺序处理给定的会议。处理会议的最基本方式是按其 开始时间 顺序排序，这也是我们采取的顺序。这就是我们将遵循的顺序。毕竟，在担心下午5：00的会议之前，你肯定应该先安排上午9：00的会议，不是吗？

**算法**

* 按照 开始时间 对会议进行排序。
* 初始化一个新的 最小堆，将第一个会议的结束时间加入到堆中。我们只需要记录会议的结束时间，告诉我们什么时候房间会空。
* 对每个会议，检查堆的最小元素（即堆顶部的房间）是否空闲。
* 若房间空闲，则从堆顶拿出该元素，将其改为我们处理的会议的结束时间，加回到堆中。
* 若房间不空闲。开新房间，并加入到堆中。
* 处理完所有会议后，堆的大小即为开的房间数量。这就是容纳这些会议需要的最小房间数。

public class Solution {

public int minMeetingRooms(List<Interval> intervals) {

// Check for the base case. If there are no intervals, return 0

if (intervals.size() == 0) {

return 0;

}

// Min heap

PriorityQueue<Integer> allocator =

new PriorityQueue<Integer>(

intervals.size(),

new Comparator<Integer>() {

public int compare(Integer a, Integer b) {

return a - b;

}

});

// Sort the intervals by start time

Collections.sort(

intervals,

new Comparator<Interval>() {

public int compare(final Interval a, final Interval b) {

return a.start - b.end;

}

});

// Add the first meeting

allocator.add(intervals.get(0).end);

// Iterate over remaining intervals

for (int i = 1; i < intervals.size(); i++) {

// If the room due to free up the earliest is free, assign that room to this meeting.

if (intervals.get(i).start >= allocator.peek()) {

allocator.poll();

}

// If a new room is to be assigned, then also we add to the heap,

// If an old room is allocated, then also we have to add to the heap with updated end time.

allocator.add(intervals.get(i).end);

}

// The size of the heap tells us the minimum rooms required for all the meetings.

return allocator.size();

}

}

**复杂度分析**

* 时间复杂度：*O*(*N*log*N*)
* 时间开销主要有两部分。第一部分是数组的 排序过程，消耗*O*(*N*log*N*)的时间。数组中有*N*个元素。
* 接下来是 最小堆占用的时间。在最坏的情况下，全部*N*个会议都会互相冲突。在任何情况下，我们都要向堆执行 *N*次插入操作。在最坏的情况下，我们要对堆进行*N*次查找并删除最小值操作。总的时间复杂度为(*NlogN*)，因为查找并删除最小值操作只消耗*O*(log*N*)的时间。
* 空间复杂度：*O*(*N*)。额外空间用于建立 最小堆。在最坏的情况下，堆需要容纳全部*N*个元素。因此空间复杂度为 *O*(*N*)

**Solution 3: Sweep Line (30 min)**

class Solution {

public int minMeetingRooms(int[][] intervals) {

// Define the size for time slots (with assumed maximum time as 10^6+10)

int[] timeline = new int[100010];

for(int[] interval : intervals) {

// Increment the start time to indicate a new meeting starts

timeline[interval[0]]++;

// Decrement the end time to indicate a meeting ends

timeline[interval[1]]--;

}

// Cumulate the changes to find active meetings at time i

int[] presum = new int[timeline.length + 1];

for(int i = 1; i < presum.length; i++) {

presum[i] = presum[i - 1] + timeline[i - 1];

}

// Traverse over the delta array to find maximum number of ongoing meetings at any time

int result = 0;

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

result = Math.max(result, presum[i]);

}

return result;

}

}

Time Complexity:O(n)

Space Complexity:O(n)

**Refer to**

<https://algo.monster/liteproblems/253>

## Problem Description

The problem presents a scenario where we have an array of meeting time intervals, each represented by a pair of numbers [start\_i, end\_i]. These pairs indicate when a meeting starts and ends. The goal is to find the minimum number of conference rooms required to accommodate all these meetings without any overlap. In other words, we want to allocate space such that no two meetings occur in the same room simultaneously.

## Intuition

The core idea behind the solution is to track the changes in room occupancy over time, which is akin to tracking the number of trains at a station at any given time. We can visualize the timeline from the start of the first meeting to the end of the last meeting, and keep a counter that increments when a meeting starts and decrements when a meeting ends. This approach is similar to the sweep line algorithm, often used in computational geometry to keep track of changes over time or another dimension.

By iterating through all the meetings, we apply these increments/decrements at the respective start and end times. The maximum value reached by this counter at any point in time represents the peak occupancy, thus indicating the minimum number of conference rooms needed. To implement this:

1. We initialize an array delta that is large enough to span all potential meeting times. We use a fixed size in this solution, which assumes the meeting times fall within a predefined range (0 to 1000009 in this case).
2. Iterate through the intervals list, and for each meeting interval [start, end], increment the value at index start in the delta array, and decrement the value at index end. This effectively marks the start of a meeting with +1 (indicating a room is now occupied) and the end of a meeting with -1 (a room has been vacated).
3. Accumulate the changes in the delta array using the accumulate function, which applies a running sum over the array elements. The maximum number reached in this accumulated array is our answer, as it represents the highest number of simultaneous meetings, i.e., the minimum number of conference rooms required.

This solution is efficient because it avoids the need to sort the meetings by their start or end times, and it provides a direct way to calculate the running sum of room occupancy over the entire timeline.

## Solution Approach

The solution uses a simple array and the concept of the prefix sum (running sum) to keep track of room occupancy over time—an approach that is both space-efficient and does not require complex data structures.

Here's a step-by-step breakdown of the implementation:

**1.Initialization**: A large array delta is created with all elements initialized to 0. The size of the array is chosen to be large enough to handle all potential meeting times (1 more than the largest possible time to account for the last meeting's end time). In this case, 1000010 is used.

**2.Updating the delta Array**: For each meeting interval, say [start, end], we treat the start time as the point where a new room is needed (increment counter) and the end time as the point where a room is freed (decrement counter).

for start, end in intervals:

delta[start] += 1

delta[end] -= 1

This creates a timeline indicating when rooms are occupied and vacated.

**3.Calculating the Prefix Sum**: We use the accumulate function from the itertools module of Python to create a running sum (also known as a prefix sum) over the delta array. The result is a new array indicating the number of rooms occupied at each time.

occupied\_rooms\_over\_time = accumulate(delta)

**4.Finding the Maximum Occupancy**: The peak of the occupied\_rooms\_over\_time array represents the maximum number of rooms simultaneously occupied, hence the minimum number of rooms we need.

The max function is used to find this peak value, which completes our solution.

min\_rooms\_required = max(occupied\_rooms\_over\_time)

The beauty of this approach is in its simplicity and efficiency. Instead of worrying about sorting meetings by starts or ends or using complex data structures like priority queues, we leverage the fact that when we are only interested in the max count, the order of increments and decrements on the timeline does not matter. As long as we correctly increment at the start times and decrement at the end times, the accumulate function ensures we get a correct count at each time point.

In conclusion, this method provides an elegant solution to the problem using basic array manipulation and the concept of prefix sums.

### Example Walkthrough

Let's consider a small set of meeting intervals to illustrate the solution approach:

1Meeting intervals: [[1, 4], [2, 5], [7, 9]]

Here we have three meetings. The first meeting starts at time 1 and ends at time 4, the second meeting starts at time 2 and ends at time 5, and the third meeting starts at time 7 and ends at time 9.

Following the solution steps:

**1.Initialization:** We create an array delta of size 1000010, which is a bit overkill for this small example, but let's go with the provided approach. Initially, all elements in delta are set to 0.

**2.Updating the delta Array:** We iterate through the meeting intervals and update the delta array accordingly.

After the updates, the delta array will reflect changes in room occupancy at the start and end times of the meetings.

delta[1] += 1 # Meeting 1 starts, need a room

delta[4] -= 1 # Meeting 1 ends, free a room

delta[2] += 1 # Meeting 2 starts, need a room

delta[5] -= 1 # Meeting 2 ends, free a room

delta[7] += 1 # Meeting 3 starts, need a room

delta[9] -= 1 # Meeting 3 ends, free a room

**3.Calculating the Prefix Sum:** Using an accumulate operation (similar to a running sum), we calculate the number of rooms occupied at each point in time. For simplicity, we will perform the cumulation manually:

The maximum number during this running sum is 2, which occurs at times 2 and 3.

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)

**4.Finding the Maximum Occupancy:** We can see that the highest value in the occupancy timeline is 2, therefore we conclude that at least two conference rooms are needed to accommodate all meetings without overlap.

The minimum number of conference rooms required is 2.

## **Java Solution**

class Solution {

// Function to find the minimum number of meeting rooms required

public int minMeetingRooms(int[][] intervals) {

// Define the size for time slots (with assumed maximum time as 10^6+10)

int n = 1000010;

int[] delta = new int[n]; // Array to hold the changes in ongoing meetings

// Iterate through all intervals

for (int[] interval : intervals) {

// Increment the start time to indicate a new meeting starts

++delta[interval[0]];

// Decrement the end time to indicate a meeting ends

--delta[interval[1]];

}

// Initialize res to the first time slot to handle the case if only one meeting

int res = delta[0];

// Traverse over the delta array to find maximum number of ongoing meetings at any time

for (int i = 1; i < n; ++i) {

// Cumulate the changes to find active meetings at time i

delta[i] += delta[i - 1];

// Update res if the current time slot has more meetings than previously recorded

res = Math.max(res, delta[i]);

}

// Return the maximum value found in delta, which is the minimum number of rooms required

return res;

}

}