253. Meeting Rooms II

Sept. 9, 2018 | 171.9K views

Average Rating: 4.77 (273 votes)

Given an array of meeting time intervals consisting of start and end times `[[s1,e1],[s2,e2],...]` (s_i < e_i), 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.

Solution

Intuition

This problem is very similar to something that employees of a company can face potentially on daily basis.

Suppose you work at a company and you belong to the IT department and one of your job responsibilities is securing rooms for meetings that are to happen throughout the day in the office.

You have multiple meeting rooms in the office and you want to make judicious use of them. You don't really want to keep people waiting and want to give a group of employees a room to hold the meeting right on time.

At the same time, you don't really want to use too many rooms unless absolutely necessary. It would make sense to hold meetings in different rooms provided that the meetings are colliding with each other, otherwise you want to make use of as less rooms as possible to hold all of the meetings. How do you go about it ?

I just represented a common scenario at an office where given the start and end times for meetings to happen throughout the day, you, as an IT guy need to setup and allocate the room numbers to different teams.

Let's approach this problem from the perspective of a group of people who want to hold a meeting and have not been allocated a room yet. What would they do?

> This group would essentially go from one room to another and check if any meeting room is free. If they find a room that is indeed free, they would start their meeting in that room. Otherwise, they would wait for a room to be free. As soon as the room frees up, they would occupy it.

This is the basic approach that we will follow in this question. So, it is a kind of simulation but not exactly. In the worst case we can assign a new room to all of the meetings but that is not really optimal right? Unless of course they all collide with each other.

> We need to be able to find out efficiently if a room is available or not for the current meeting and assign a new room only if none of the assigned rooms is currently free.

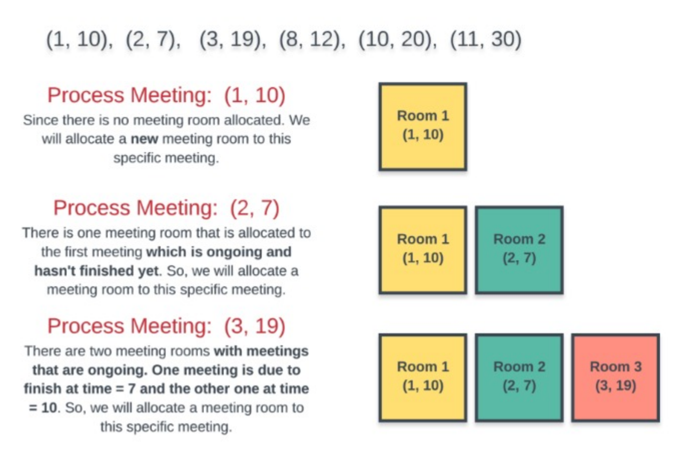
Let's look at the first approach based on the idea we just discussed.

Approach 1: Priority Queues

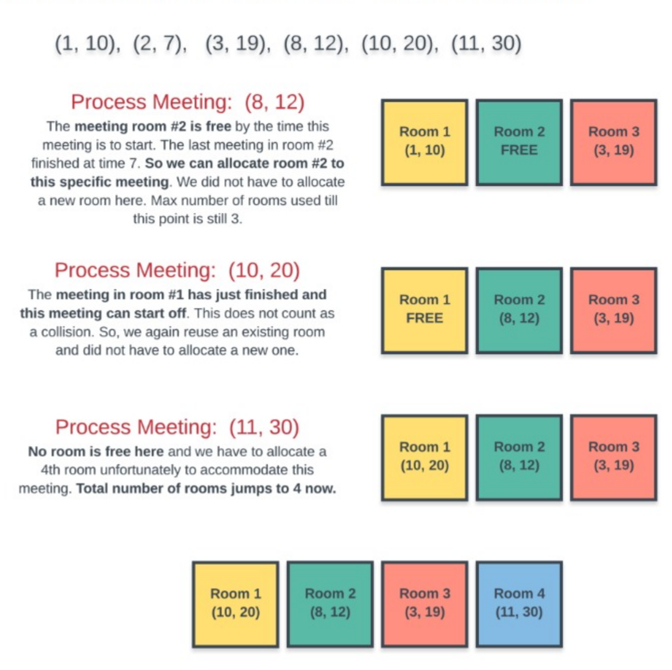
We can't really process the given meetings in any random order. The most basic way of processing the meetings is in increasing order of their `start times` and this is the order we will follow. After all if you're an IT guy, you should allocate a room to the meeting that is scheduled for 9 a.m. in the morning before you worry about the 5 p.m. meeting, right?

Let's do a dry run of an example problem with sample meeting times and see what our algorithm should be able to do efficiently.

We will consider the following meeting times for our example `(1, 10), (2, 7), (3, 19), (8, 12), (10, 20), (11, 30)`. The first part of the tuple is the start time for the meeting and the second value represents the ending time. We are considering the meetings in a sorted order of their start times. The first diagram depicts the first three meetings where each of them requires a new room because of collisions.



The next 3 meetings start to occupy some of the existing rooms. However, the last one requires a new room altogether and overall we have to use 4 different rooms to accommodate all the meetings.



Sorting part is easy, but for every meeting how do we find out efficiently if a room is available or not? At any point in time we have multiple rooms that can be occupied and we don't really care which room is free as long as we find one when required for a new meeting.

A naive way to check if a room is available or not is to iterate on all the rooms and see if one is available when we have a new meeting at hand.

> However, we can do better than this by making use of Priority Queues or the Min-Heap data structure.

Instead of manually iterating on every room that's been allocated and checking if the room is available or not, we can keep all the rooms in a min heap where the key for the min heap would be the ending time of meeting.

So, every time we want to check if **any** room is free or not, simply check the topmost element of the min heap as that would be the room that would get free the earliest out of all the other rooms currently occupied.

If the room we extracted from the top of the min heap isn't free, then `no other room is`. So, we can save time here and simply allocate a new room.

Let us look at the algorithm before moving onto the implementation.

Algorithm

1. Sort the given meetings by their `start time`.
2. Initialize a new `min-heap` and add the first meeting's ending time to the heap. We simply need to keep track of the ending times as that tells us when a meeting room will get free.
3. For every meeting room check if the minimum element of the heap i.e. the room at the top of the heap is free or not.
4. If the room is free, then we extract the topmost element and add it back with the ending time of the current meeting we are processing.
5. If not, then we allocate a new room and add it to the heap.
6. After processing all the meetings, the size of the heap will tell us the number of rooms allocated. This will be the minimum number of rooms needed to accommodate all the meetings.

Let us not look at the implementation for this algorithm.

Java | Python | Copy

```python

        # If there is no meeting to schedule then no room needs to be allocated.
        if not intervals:
            return 0

        # The heap initialization
        free_rooms = []

        # Sort the meetings in increasing order of their start time.
        intervals.sort(key= lambda x: x[0])

        # Add the first meeting. We have to give a new room to the first meeting.
        heapq.heappush(free_rooms, intervals[0][1])

        # For all the remaining meeting rooms
        for i in intervals[1:]:

            # If the room due to free up the earliest is free, assign that room to this meeting.
            if free_rooms[0] <= i[0]:
                heapq.heappop(free_rooms)

            # 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.
            heapq.heappush(free_rooms, i[1])

        # The size of the heap tells us the minimum rooms required for all the meetings.
        return len(free_rooms)
```

Complexity Analysis

- Time Complexity: $O(N \log N)$.
  - There are two major portions that take up time here. One is `sorting` of the array that takes $O(N \log N)$ considering that the array consists of N elements.
  - Then we have the `min-heap`. In the worst case, all N meetings will collide with each other. In any case we have N add operations on the heap. In the worst case we will have N extract-min operations as well. Overall complexity being $(NlogN)$ since extract-min operation on a heap takes $O(\log N)$.
- Space Complexity: $O(N)$ because we construct the `min-heap` and that can contain N elements in the worst case as described above in the time complexity section. Hence, the space complexity is $O(N)$.

Approach 2: Chronological Ordering

Intuition

The meeting timings given to us define a chronological order of events throughout the day. We are given the start and end timings for the meetings which can help us define this ordering.

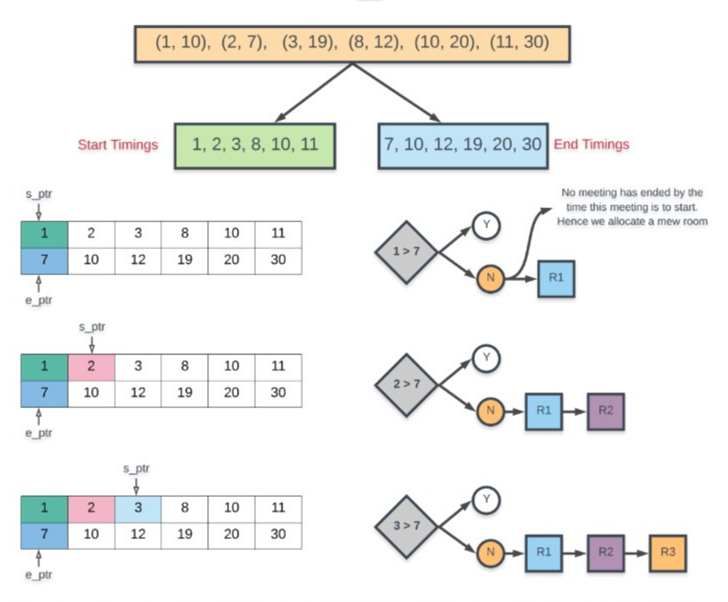
Arranging the meetings according to their start times helps us know the natural order of meetings throughout the day. However, simply knowing when a meeting starts doesn't tell us much about its duration.

We also need the meetings sorted by their ending times because an ending event essentially tells us that there must have been a corresponding starting event and more importantly, an ending event tell us that a previously occupied room has now become free.

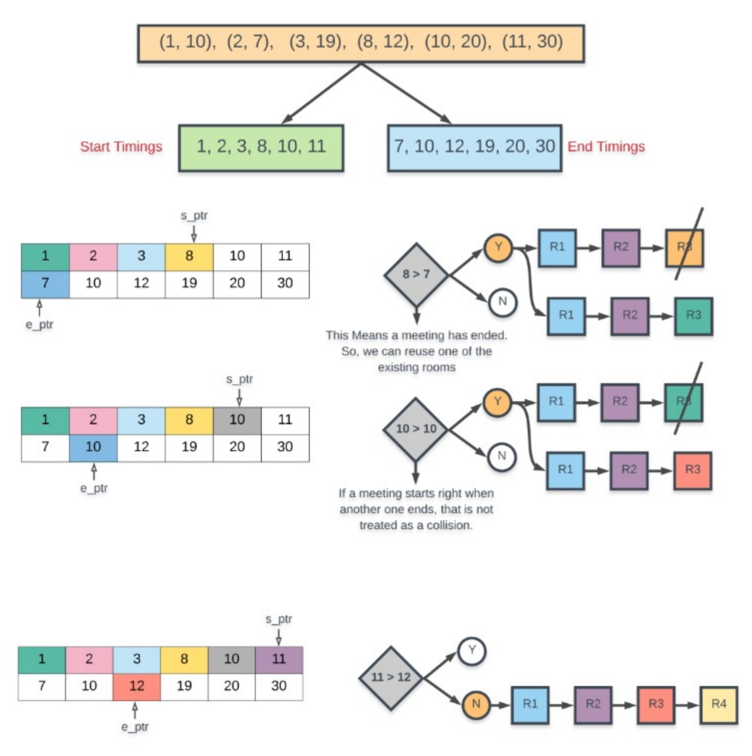
A meeting is defined by its start and end times. However, for this specific algorithm, we need to treat the start and end times `individually`. This might not make sense right away because a meeting is defined by its start and end times. If we separate the two and treat them individually, then the identity of a meeting goes away. This is fine because:

> When we encounter an ending event, that means that some meeting that started earlier has ended now. We are not really concerned with which meeting has ended. All we need is that **some** meeting ended thus making a room available.

Let us consider the same example as we did in the last approach. We have the following meetings to be scheduled: `(1, 10), (2, 7), (3, 19), (8, 12), (10, 20), (11, 30)`. As before, the first diagram show us that the first three meetings are colliding with each other and they have to be allocated separate rooms.



The next two diagrams process the remaining meetings and we see that we can now reuse some of the existing meeting rooms. The final result is the same, we need 4 different meeting rooms to process all the meetings. That's the best we can do here.



Algorithm

1. Separate out the start times and the end times in their separate arrays.
2. Sort the start times and the end times separately. Note that this will mess up the original correspondence of start times and end times. They will be treated individually now.
3. We consider two pointers: `s_ptr` and `e_ptr` which refer to start pointer and end pointer. The start pointer simply iterates over all the meetings and the end pointer helps us track if a meeting has ended and if we can reuse a room.
4. When considering a specific meeting pointed to by `s_ptr`, we check if this start timing is greater than the meeting pointed to by `e_ptr`. If this is the case then that would mean some meeting has ended by the time the meeting at `s_ptr` had to start. So we can reuse one of the rooms. Otherwise, we have to allocate a new room.
5. If a meeting has indeed ended i.e. if `start[s_ptr] >= end[e_ptr]`, then we increment `e_ptr`.
6. Repeat this process until `s_ptr` processes all of the meetings.

Let us not look at the implementation for this algorithm.

Java | Python | Copy

```python
class Solution:
    def minMeetingRooms(self, intervals: List[List[int]]) -> int:

        # If there are no meetings, we don't need any rooms.
        if not intervals:
            return 0

        used_rooms = 0

        # Separate out the start and the end timings and sort them individually.
        start_timings = sorted([i[0] for i in intervals])
        end_timings = sorted(i[1] for i in intervals)
        L = len(intervals)

        # The two pointers in the algorithm: e_ptr and s_ptr.
        end_pointer = 0
        start_pointer = 0

        # Until all the meetings have been processed
        while start_pointer < L:
            # If there is a meeting that has ended by the time the meeting at `start_pointer` starts
            if start_timings[start_pointer] >= end_timings[end_pointer]:
                # Free up a room and increment the end_pointer.
                used_rooms -= 1
                end_pointer += 1

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

Complexity Analysis

- Time Complexity: $O(N \log N)$ because all we are doing is sorting the two arrays for `start timings` and `end timings` individually and each of them would contain N elements considering there are N intervals.
- Space Complexity: $O(N)$ because we create two separate arrays of size N, one for recording the start times and one for the end times.

Rate this article: ★★★★★

Previous Next

Comments: 80 Sort By

Type comment here... (Markdown is supported)

Preview Post

shuo21 ★ 203 October 10, 2018 12:33 AM

Let us NOT look at the implementation for this algorithm.
I like it .

194 Share Reply

SHOW 5 REPLIES

michalmichal ★ 201 September 25, 2018 10:45 PM Report

Great explanation, I wish every question was explained that well!

85 Share Reply

SHOW 2 REPLIES

mehranangelo ★ 109 December 3, 2018 8:01 AM

This is my short interview friendly solution

```
    public int minMeetingRooms(Interval[] intervals) {
        Arrays.sort(intervals,(a,b)->(a.start-b.start));
```

Read More

73 Share Reply

SHOW 6 REPLIES

SuM_007 ★ 98 November 26, 2018 10:27 AM

```
class Solution {
    public int minMeetingRooms(Interval[] intervals) {
        int[] starts = new int[intervals.length];
```

Read More

44 Share Reply

enjoymrsun ★ 92 November 14, 2018 2:20 AM Report

absolutely 5 star solution

34 Share Reply

SHOW 1 REPLY

sahiti4 ★ 30 September 12, 2018 7:19 AM

Those are some freakish ideas.......

22 Share Reply

motime ★ 22 March 4, 2019 2:01 AM

Why do we need to define a comparator for PriorityQueue? Isn't it a min integer heap by default?

21 Share Reply

SHOW 4 REPLIES

a-b-c ★ 690 November 24, 2018 11:07 PM

one of the best articles on leetcode, thank you

17 Share Reply

SHOW 1 REPLY

slow_danger ★ 13 March 31, 2019 9:34 PM Report

One small improvement to make this more idiomatic Java would be to use anonymous functions for the comparators:

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

Read More

13 Share Reply

SHOW 2 REPLIES

christopherwu0529 ★ 1063 February 8, 2019 10:05 AM

Solution 1 is so smart

13 Share Reply

SHOW 2 REPLIES

‹ 1 2 3 4 5 6 7 8 ›