Meeting Rooms II

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

For example,
Given `[[0, 30],[5, 10],[15, 20]]`,
return `2`.

Solution 1

Just want to share another idea that uses min heap, average time complexity is O(nlogn).

```java
public int minMeetingRooms(Interval[] intervals) {
    if (intervals == null || intervals.length == 0)
        return 0;

    // Sort the intervals by start time
    Arrays.sort(intervals, new Comparator<Interval>() {
        public int compare(Interval a, Interval b) { return a.start - b.start; }
    });

    // Use a min heap to track the minimum end time of merged intervals
    PriorityQueue<Interval> heap = new PriorityQueue<Interval>(intervals.length,
new Comparator<Interval>() {
        public int compare(Interval a, Interval b) { return a.end - b.end; }
    });

    // start with the first meeting, put it to a meeting room
    heap.offer(intervals[0]);

    for (int i = 1; i < intervals.length; i++) {
        // get the meeting room that finishes earliest
        Interval interval = heap.poll();

        if (intervals[i].start >= interval.end) {
            // if the current meeting starts right after
            // there's no need for a new room, merge the interval
            interval.end = intervals[i].end;
        } else {
            // otherwise, this meeting needs a new room
            heap.offer(intervals[i]);
        }

        // don't forget to put the meeting room back
        heap.offer(interval);
    }

    return heap.size();
}
```

written by jeantimex original link here

Solution 2

First collect the changes: at what times the number of meetings goes up or down and by how much. Then go through those changes in ascending order and keep track of the current and maximum number of rooms needed.

Solution 1: Using `map` ... 600 ms

```cpp
int minMeetingRooms(vector<Interval>& intervals) {
    map<int, int> changes;
    for (auto i : intervals) {
        changes[i.start] += 1;
        changes[i.end] -= 1;
    }
    int rooms = 0, maxrooms = 0;
    for (auto change : changes)
        maxrooms = max(maxrooms, rooms += change.second);
    return maxrooms;
}
```

Solution 2: Using `vector` ... 588 ms

```cpp
int minMeetingRooms(vector<Interval>& intervals) {
    vector<pair<int, int>> changes;
    for (auto i : intervals) {
        changes.push_back({i.start, 1});
        changes.push_back({i.end, -1});
    };
    sort(begin(changes), end(changes));
    int rooms = 0, maxrooms = 0;
    for (auto change : changes)
        maxrooms = max(maxrooms, rooms += change.second);
    return maxrooms;
}
```

Solution 3: Using two `vector` s ... 584 ms

Based on yinfeng.zhang.9's Python solution. Uses separate vectors for start and end times, which ends up consistently being fastest. I'm guessing it's mostly because working with ints is simpler than working with pairs of ints. The initial sorting also needs fewer steps, 2(nlogn) instead of (2n)log(2n), but I think the added merging in the later loop cancels that advantage out.

```
int minMeetingRooms(vector<Interval>& intervals) {
    vector<int> starts, ends;
    for (auto i : intervals) {
        starts.push_back(i.start);
        ends.push_back(i.end);
    }
    sort(begin(starts), end(starts));
    sort(begin(ends), end(ends));
    int e = 0, rooms = 0, available = 0;
    for (int start : starts) {
        while (ends[e] <= start) {
            ++e;
            ++available;
        }
        available ? --available : ++rooms;
    }
    return rooms;
}
```

written by StefanPochmann original link here

Solution 3

Simulate event queue procession. Create event for each `start` and `end` of intervals. Then for `start` event, open one more room; for `end` event, close one meeting room. At the same time, update the most rooms that is required.

Be careful of events like `[(end at 11), (start at 11)]`. Put `end` before `start` event when they share the same happening time, so that two events can share one meeting room.

```java
public class Solution {

    private static final int START = 1;

    private static final int END = 0;

    private class Event {
        int time;
        int type; // end event is 0; start event is 1

        public Event(int time, int type) {
            this.time = time;
            this.type = type;
        }
    }

    public int minMeetingRooms(Interval[] intervals) {
        int rooms = 0; // occupied meeting rooms
        int res = 0;

        // initialize an event queue based on event's happening time
        Queue<Event> events = new PriorityQueue<>(new Comparator<Event>() {
            @Override
            public int compare(Event e1, Event e2) {
                // for same time, let END event happens first to save rooms
                return e1.time != e2.time ?
                       e1.time - e2.time : e1.type - e2.type;
            }
        });

        // create event and push into event queue
        for (Interval interval : intervals) {
            events.offer(new Event(interval.start, START));
            events.offer(new Event(interval.end, END));
        }

        // process events
        while (!events.isEmpty()) {
            Event event = events.poll();
            if (event.type == START) {
                rooms++;
                res = Math.max(res, rooms);
            } else {
                rooms--;
            }
        }

        return res;
    }

}
```

written by StevenCooks original link here