253. Meeting Rooms II Heap (Priority Queue) Medium Two Pointers Prefix Sum Sorting **Array**

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: 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).

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). 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.

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.

time). In this case, 1000010 is used. 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:

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

- delta[start] += 1 delta[end] -= 1
- This creates a timeline indicating when rooms are occupied and vacated. 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

```
occupied_rooms_over_time = accumulate(delta)
  Finding the Maximum Occupancy: The peak of the occupied_rooms_over_time array represents the maximum number of
```

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.
```

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

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

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

min_rooms_required = max(occupied_rooms_over_time)

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.

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

Meeting 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:

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.

Example Walkthrough

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

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

Calculating the Prefix Sum: Using an accumulate operation (similar to a running sum), we calculate the number of rooms

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

```
1 2 3 4 5 6 7 8 9
time
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)
```

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.

that at least two conference rooms are needed to accommodate all meetings without overlap.

The minimum number of conference rooms required is 2.

Finding the Maximum Occupancy: We can see that the highest value in the occupancy timeline is 2, therefore we conclude

```
from typing import List
from itertools import accumulate
class Solution:
   def minMeetingRooms(self, intervals: List[List[int]]) -> int:
```

The range is chosen such that it covers all possible meeting times

meeting_delta[start] += 1 # Increment for a meeting starting

meeting_delta[end] -= 1 # Decrement for a meeting ending

Go through each interval in the provided list of intervals

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

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

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

// Decrement the end time to indicate a meeting ends

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

Initialize a list to keep track of the number of meetings starting or ending at any time

The `accumulate` function is used to compute the running total of active meetings at each time

`max` is then used to find the maximum number of concurrent meetings, which is the minimum number of rooms required

```
# Example usage:
# sol = Solution()
# print(sol.minMeetingRooms([[0, 30], [5, 10], [15, 20]]))  # Output: 2
```

int n = 1000010;

Java

class Solution {

Solution Implementation

meeting_delta = [0] * 1000010

for start, end in intervals:

return max(accumulate(meeting_delta))

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

// Iterate through all intervals

++delta[interval[0]];

for (int[] interval : intervals) {

Python

```
--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;
#include <vector>
#include <algorithm> // Include necessary headers
class Solution {
public:
   // Function to find the minimum number of meeting rooms required
    int minMeetingRooms(vector<vector<int>>& intervals) {
        int n = 1000010; // Define the maximum possible time point
       vector<int> countDelta(n); // Create a vector to keep track of the count changes
       // Iterate over each interval
        for (auto &interval : intervals) {
            ++countDelta[interval[0]]; // Increment count at the start time of the meeting
            --countDelta[interval[1]]; // Decrement count at the end time of the meeting
       // Prefix sum — accumulate the count changes to find the count at each time
        for (int i = 0; i < n - 1; ++i) {
            countDelta[i + 1] += countDelta[i];
       // Find and return the maximum count at any time, which is the minimum number of rooms required
       return *max_element(countDelta.begin(), countDelta.end());
```

const countChange: number[] = new Array(maximumTimePoint).fill(0); // Create an array to track the count changes, initialized

```
// Example usage:
// const intervals: number[][] = [[0, 30], [5, 10], [15, 20]];
// const numberOfRooms: number = minMeetingRooms(intervals);
```

});

};

TypeScript

import { max } from "lodash";

// Iterate over each interval

return max(countChange);

intervals.forEach(interval => {

// Import necessary module for max element search

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

const maximumTimePoint: number = 1000010; // Define the maximum possible time point

countChange[interval[0]]++; // Increment count at the start time of the meeting

// Prefix sum calculation - accumulate the count changes to find the count at each time point

// Find and return the maximum count at any time point, which is the minimum number of rooms required

countChange[interval[1]]--; // Decrement count at the end time of the meeting

function minMeetingRooms(intervals: number[][]): number {

for (let i = 0; $i < maximumTimePoint - 1; ++i) {$

countChange[i + 1] += countChange[i];

```
// console.log(numberOfRooms); // Should output the minimum number of meeting rooms required
from typing import List
from itertools import accumulate
class Solution:
   def minMeetingRooms(self, intervals: List[List[int]]) -> int:
       # Initialize a list to keep track of the number of meetings starting or ending at any time
       # The range is chosen such that it covers all possible meeting times
       meeting_delta = [0] * 1000010
       # Go through each interval in the provided list of intervals
       for start, end in intervals:
           meeting_delta[start] += 1 # Increment for a meeting starting
           meeting_delta[end] -= 1  # Decrement for a meeting ending
       # The `accumulate` function is used to compute the running total of active meetings at each time
       # `max` is then used to find the maximum number of concurrent meetings, which is the minimum number of rooms required
       return max(accumulate(meeting_delta))
# Example usage:
# sol = Solution()
# print(sol.minMeetingRooms([[0, 30], [5, 10], [15, 20]])) # Output: 2
Time and Space Complexity
  The provided Python code is meant to determine the minimum number of meeting rooms required for a set of meetings,
  represented by their start and end times. The code employs a difference array to keep track of the changes in the room
  occupancy count caused by the start and end of meetings, then uses accumulate to find the peak occupancy which gives the
  required number of rooms.
Time Complexity
```

Initializing the delta list, which has a fixed length of 1000010. This step is in 0(1) since it does not depend on the number of intervals but on a constant size.

Space Complexity

The time complexity of the code is determined by three steps:

The first for loop that populates the delta array with +1 and -1 for the start and end times of meetings. It runs 0(n) times, where n is the number of intervals (meetings).

- The use of accumulate which calculates the prefix sum of the delta array. In the worst case, this operation is O(m), where m is the size of the delta array, which is a constant of 1000010.
- The final time complexity is dominated by the larger of the two variables, which in this case is the constant time for using accumulate. Hence, the time complexity is O(m), which translates to O(1) since m is a constant.
- The space complexity of the code is determined primarily by the delta array. The delta list, which has a fixed length of 1000010. This is a constant space allocation and does not depend on the input size.

No additional data structures that grow with the size of the input are used.

Therefore, the space complexity is 0(1) due to the constant size of the delta array.