N meetings in one room

Problem Description

You are given:

- An integer NN representing the number of meetings.
- Two arrays, **start** and **end**, where **start[i]** is the start time and **end[i]** is the end time of the ii-th meeting.

Your goal is to find the maximum number of non-overlapping meetings that can be scheduled in one meeting room.

Key Observations

1. **Non-overlapping Meetings**:
    - Meetings must not overlap, meaning the start time of one meeting must be strictly greater than the end time of the previous meeting.
2. **Greedy Choice**:
    - To maximize the number of meetings, we should always pick the meeting that finishes the earliest. This leaves the room available for subsequent meetings as soon as possible.

Approach

To solve this problem using a greedy algorithm, follow these steps:

1. **Pair and Sort**:
    - Create pairs of (start, end) times.
    - Sort these pairs by their end times.
2. **Select Meetings**:
    - Iterate through the sorted list of meetings.
    - Keep track of the end time of the last selected meeting.
    - For each meeting, if its start time is greater than the end time of the last selected meeting, select it and update the end time.

Detailed Steps

1. **Pair and Sort**:
    - Create an array of objects where each object represents a meeting with its start and end times.
    - Sort this array by the **end** property.

2. **Iterate and Select**:
    - Initialize a counter to count the maximum number of meetings.
    - Initialize a variable to keep track of the end time of the last selected meeting.
    - For each meeting in the sorted array, if the start time is greater than the end time of the last selected meeting, select this meeting, increment the counter, and update the last end time.

Example

Example 1:

- Input:
    - N=6N=6
    - **start = [1, 3, 0, 5, 8, 5]**
    - **end = [2, 4, 6, 7, 9, 9]**
- Output: **4**
- Explanation:
    - Create pairs: **[{start: 1, end: 2}, {start: 3, end: 4}, {start: 0, end: 6}, {start: 5, end: 7}, {start: 8, end: 9}, {start: 5, end: 9}]**
    - Sort by end time: **[{start: 1, end: 2}, {start: 3, end: 4}, {start: 5, end: 7}, {start: 0, end: 6}, {start: 5, end: 9}, {start: 8, end: 9}]**
    - Select meetings based on sorted end times:
        - Select (1, 2) -> End time now 2
        - Select (3, 4) -> End time now 4
        - Select (5, 7) -> End time now 7
        - Select (8, 9) -> End time now 9
    - Total selected meetings: 4

JavaScript Implementation

Here's the JavaScript implementation of the greedy algorithm:

```javascript
function maxMeetings(start, end, N) {
    // Create an array of meetings with start and end times
    let meetings = [];
    for (let i = 0; i < N; i++) {
        meetings.push({start: start[i], end: end[i]});
    }

    // Sort the meetings by their end times
    meetings.sort((a, b) => a.end - b.end);

    // Initialize the count of meetings
    let count = 0;
    // End time of the last selected meeting
    let lastEndTime = -1;

    // Iterate through the sorted meetings
    for (let i = 0; i < N; i++) {
        if (meetings[i].start > lastEndTime) {
            // If the start time of the current meeting is greater
            // than the end time of the last selected meeting
            count++;
            lastEndTime = meetings[i].end;
        }
    }
```

```javascript
    return count;
}

// Example usage:
let N = 6;
let start = [1, 3, 0, 5, 8, 5];
let end = [2, 4, 6, 7, 9, 9];
console.log(maxMeetings(start, end, N)); // Output: 4
```

Complexity Analysis

- **Time Complexity**:
    - Sorting the meetings takes $O(N\log N)$$O(N\log N)$.
    - Iterating through the meetings takes $O(N)$$O(N)$.
    - Therefore, the overall time complexity is $O(N\log N)$$O(N\log N)$.
- **Space Complexity**:
    - Storing the list of meetings takes $O(N)$$O(N)$.
    - The space complexity is $O(N)$$O(N)$.

This approach efficiently solves the problem by using a greedy algorithm to ensure the maximum number of non-overlapping meetings are scheduled.

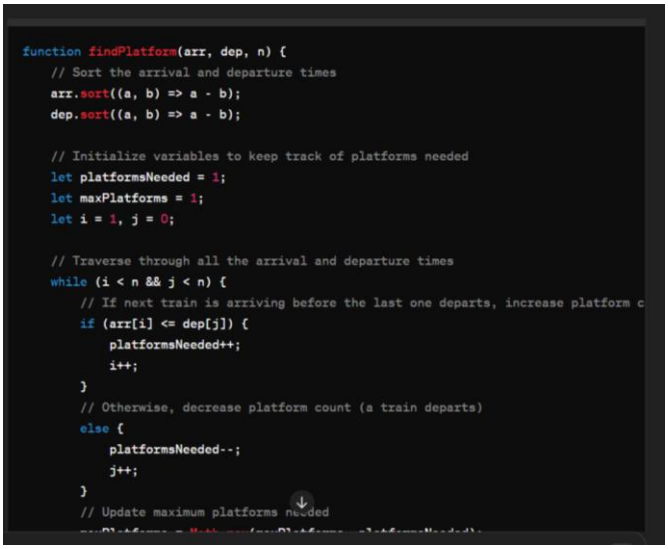
To solve the problem of finding the minimum number of platforms required at a railway station to ensure that no train is kept waiting, we need to consider the overlapping intervals of train arrivals and departures. Here's a step-by-step explanation and the corresponding JavaScript code to solve this problem:

Steps to Solve the Problem:

1. **Sort the Arrival and Departure Times**:
    - First, we need to sort both the arrival and departure times. This will help us efficiently determine when platforms are freed and when they are needed.
2. **Use Two Pointers Technique**:
    - We will use two pointers, one for the arrival array and one for the departure array.
    - Initialize two pointers **i** and **j** to 0 to track the current arrival and departure respectively.
    - Initialize variables to keep track of the current number of platforms needed and the maximum platforms needed.
3. **Traverse Through the Sorted Arrays**:
    - As we traverse through the sorted arrival and departure times, we will increase the platform count when a train arrives and decrease it when a train departs.
    - The idea is to iterate through the times and update the number of platforms needed at each step, keeping track of the maximum number of platforms required at any point in time.
4. **Compute the Result**:
    - The maximum value of platforms needed during the traversal will be our answer.

JavaScript Code:

Here's the implementation of the above logic in JavaScript:



```javascript
function findPlatform(arr, dep, n) {
    // Sort the arrival and departure times
    arr.sort((a, b) => a - b);
    dep.sort((a, b) => a - b);

    // Initialize variables to keep track of platforms needed
    let platformsNeeded = 1;
    let maxPlatforms = 1;
    let i = 1, j = 0;

    // Traverse through all the arrival and departure times
    while (i < n && j < n) {
        // If next train is arriving before the last one departs, increase platform c
        if (arr[i] <= dep[j]) {
            platformsNeeded++;
            i++;
        }
        // Otherwise, decrease platform count (a train departs)
        else {
            platformsNeeded--;
            j++;
        }
        // Update maximum platforms needed
```

```
        // Update maximum platforms needed
        maxPlatforms = Math.max(maxPlatforms, platformsNeeded);
    }

    return maxPlatforms;
}

// Example usage:
const arr1 = [900, 940, 950, 1100, 1500, 1800];
const dep1 = [910, 1200, 1120, 1130, 1900, 2000];
const n1 = arr1.length;
console.log(findPlatform(arr1, dep1, n1)); // Output: 3

const arr2 = [900, 1100, 1235];
const dep2 = [1000, 1200, 1240];
const n2 = arr2.length;
console.log(findPlatform(arr2, dep2, n2)); // Output: 1
```

Explanation:

1. **Sorting**:
    - Both **arr** and **dep** arrays are sorted to facilitate the comparison of arrivals and departures in order.
2. **Traversal with Two Pointers**:
    - By using pointers **i** and **j**, we iterate through the sorted arrays.
    - **platformsNeeded** keeps track of the current number of platforms required.
    - **maxPlatforms** keeps track of the maximum platforms required at any time.
    - If an arrival time is less than or equal to the current departure time, a new platform is needed (**platformsNeeded++**).
    - If a departure time is less than the current arrival time, a platform is freed (**platformsNeeded--**).
3. **Updating the Result**:
    - **maxPlatforms** is updated to the maximum value of **platformsNeeded** encountered during the traversal, ensuring we capture the peak platform requirement.

By following this approach, we ensure that we can efficiently compute the minimum number of platforms required with a time complexity of $O(n\log n)O(n\log n)$, which is suitable given the constraints.

Problem Breakdown

1. **Input**:
    - **N**: Number of jobs.
    - **Jobs**: List of jobs where each job is represented as **(Jobid, Deadline, Profit)**.
2. **Output**:

- The number of jobs that can be scheduled.
- The maximum profit that can be earned.

Steps to Solve

1. **Sort Jobs by Profit**:
    - We sort the jobs in descending order based on profit. This allows us to consider the highest profit jobs first, which is crucial for maximizing profit.
2. **Initialize Time Slots**:
    - We need an array to keep track of which time slots are filled. This array will have a length equal to the maximum deadline among all jobs because that's the latest time slot we might need.
3. **Schedule Jobs**:
    - For each job in the sorted list, we attempt to schedule it in the latest possible slot before its deadline. We do this by checking from the job's deadline backwards to find an empty slot.
    -

Detailed Steps and Explanation

1. **Sort Jobs by Profit**:
    - We sort the jobs in descending order based on their profit. This way, the most profitable jobs are considered first, which helps in maximizing the overall profit. The sorting is done using JavaScript's **sort** method with a custom comparator that sorts based on the third element (profit) in descending order.
2. **Initialize Time Slots**:
    - We find the maximum deadline from all the jobs. This gives us the maximum number of slots we may need.
    - We create an array **timeSlots** of length **maxDeadline** initialized with **false**, indicating that all time slots are initially empty.
3. **Schedule Jobs**:
    - We iterate over each job in the sorted list:
        - For each job, we check available slots from its deadline backwards to the start.
        - If we find an empty slot (**timeSlots[j]** is **false**), we schedule the job in that slot by setting **timeSlots[j]** to **true**.

- We increase the count of scheduled jobs and add the job's profit to the total profit.

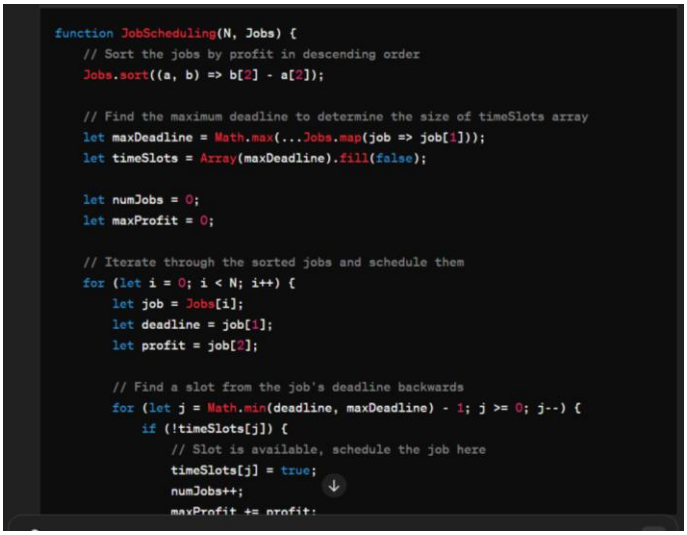
- This ensures that each job is scheduled at the latest possible time before its deadline, maximizing the number of jobs scheduled and the profit earned.

4. **Return Results**:

   - After scheduling all possible jobs, we return the total number of jobs scheduled and the maximum profit as an array.

Code Implementation

Here is the JavaScript implementation of the above logic:



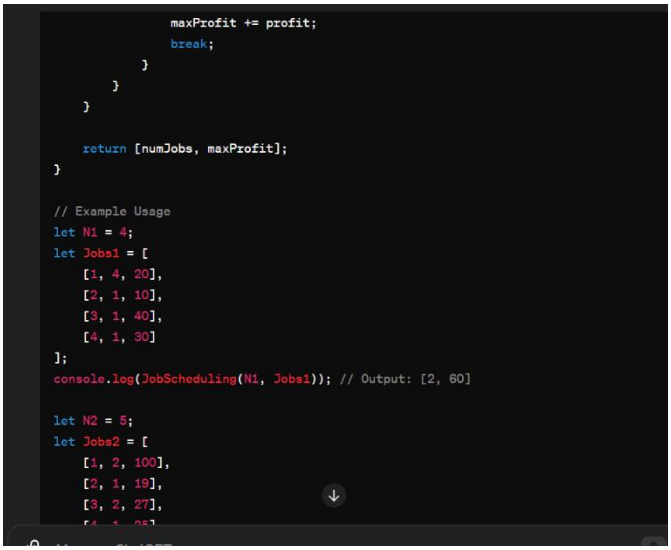
```
function JobScheduling(N, Jobs) {
    // Sort the jobs by profit in descending order
    Jobs.sort((a, b) => b[2] - a[2]);

    // Find the maximum deadline to determine the size of timeSlots array
    let maxDeadline = Math.max(...Jobs.map(job => job[1]));
    let timeSlots = Array(maxDeadline).fill(false);

    let numJobs = 0;
    let maxProfit = 0;

    // Iterate through the sorted jobs and schedule them
    for (let i = 0; i < N; i++) {
        let job = Jobs[i];
        let deadline = job[1];
        let profit = job[2];

        // Find a slot from the job's deadline backwards
        for (let j = Math.min(deadline, maxDeadline) - 1; j >= 0; j--) {
            if (!timeSlots[j]) {
                // Slot is available, schedule the job here
                timeSlots[j] = true;
                numJobs++;
                maxProfit += profit;
```



```
                maxProfit += profit;
                break;
            }
        }
    }

    return [numJobs, maxProfit];
}

// Example Usage
let N1 = 4;
let Jobs1 = [
    [1, 4, 20],
    [2, 1, 10],
    [3, 1, 40],
    [4, 1, 30]
];
console.log(JobScheduling(N1, Jobs1)); // Output: [2, 60]

let N2 = 5;
let Jobs2 = [
    [1, 2, 100],
    [2, 1, 19],
    [3, 2, 27],
```

```
    [4, 1, 25],
    [5, 1, 15]
];
console.log(JobScheduling(N2, Jobs2)); // Output: [2, 127]
```

Key Points

- **Sorting**: Sorting by profit ensures we always prioritize higher profit jobs.

- **Greedy Scheduling**: By attempting to schedule each job as late as possible (but before its deadline), we leave earlier slots open for other jobs that might have tighter deadlines.
- **Efficiency**: The solution works efficiently with a time complexity of O(N log N) due to sorting, followed by O(N * D) for scheduling (where D is the maximum deadline), which is manageable within the given constraints.

This approach ensures that we maximize the profit while adhering to the job deadlines.