1229. Meeting Scheduler

Medium Array Two Pointers Sorting Leetcode Link

Problem Description

The problem is about finding a mutual meeting time slot between two people given their individual schedules and a required meeting duration. Each person's schedule is represented by a list of non-overlapping time slots where a time slot is an array `[start, end]` showing availability from `start` to `end`. The goal is to find the earliest starting time slot that is available in both schedules and lasts at least for the given `duration`. If there's no such common time slot, we return an empty array.

Intuition

To solve this problem, we can use the two-pointer technique. Since no individual time slots overlap within a single person's schedule, we can sort both schedules by the starting times of the slots. We then compare the slots from both schedules to find overlapping slots.

We use two pointers `i` and `j` to traverse `slots1` and `slots2` respectively. In each iteration, we find the latest start time by taking the maximum of `slots1[i][0]` and `slots2[j][0]` and the earliest end time by taking the minimum of `slots1[i][1]` and `slots2[j][1]`. If the overlapping time between the latest start and earliest end is greater than or equal to the required `duration`, we have found a suitable time slot and return the start and end of this meeting slot.

If the overlap is not sufficient, we move the pointer forward in the list which has the earlier ending time slot, hoping to find a longer slot that might overlap with the other person's next slot. This process continues until we either find a suitable slot or exhaust all available slots in either list.

Solution Approach

The provided Python solution follows a straightforward two-pointer approach to solve the problem:

1. **Sorting the time slots**: Both `slots1` and `slots2` are sorted based on their starting times. This ensures that we are always considering the earliest available slots first and eliminates the need for checking past time slots. Sorting is crucial as it sets up the structure for the two-pointer technique to work effectively.

2. **Two-pointer Technique**: Two pointers, `i` and `j`, are used to iterate through `slots1` and `slots2` respectively. At each step, `i` refers to the current slot in the first person's schedule, and `j` refers to the current slot in the second person's schedule.

3. **Finding Overlaps**: For the current pair of time slots pointed by `i` and `j`, we calculate the overlap by determining the maximum of the two start times and the minimum of the two end times. The variables `start` and `end` are used to record these values:

   ```
   start = max(slots1[i][0], slots2[j][0])
   end = min(slots1[i][1], slots2[j][1])
   ```

4. **Checking Duration**: We then check if the overlapping duration is greater than or equal to the required `duration` by subtracting `start` from `end`:

   ```
   if end - start >= duration:
   ```

   If the condition is met, we have found a valid time slot and can return `[start, start + duration]` as the solution.

5. **Advancing the Pointers**: If the overlapping time slot is not long enough, we need to discard the time slot that ends earlier and move forward. This decision is made by comparing the end times of the current time slots pointed by `i` and `j`. The pointer corresponding to the slot with the earlier end time is incremented:

   ```
   if slots1[i][1] < slots2[j][1]:
       i += 1
   else:
       j += 1
   ```

   This step ensures that we're always trying to find overlap with the nearest possible future slot.

6. **Continuation and Termination**: The above steps are continued in a loop until one of the lists is exhausted (i.e., `i` reaches the end of `slots1` or `j` reaches the end of `slots2`). If no common time slots with sufficient duration are found by the end of either list, we return an empty array `[]` as specified.

This algorithm makes efficient use of the sorted structure of the time slots and the two-pointer technique to minimize the number of comparisons and quickly find the first suitable time slot, achieving a time complexity that is linear in the size of the input time slots after the initial sort.

Example Walkthrough

Let's consider an example to illustrate the solution approach:

- Person A's schedule (`slots1`): `[[10, 50], [60, 120], [140, 210]]`
- Person B's schedule (`slots2`): `[[0, 15], [25, 50], [60, 70], [80, 100]]`
- Required duration for the meeting: `8 minutes`

1. **Sorting the time slots**: Both schedules are already sorted based on their starting times, eliminating the need for a sort operation in this example.

2. **Two-pointer Technique**: Initialize two pointers, `i` for `slots1` and `j` for `slots2`. Initially, `i = 0` and `j = 0`.

3. **Finding Overlaps**:

   - First Comparison:

     - `slots1[i] = [10, 50]` and `slots2[j] = [0, 15]`
     - The overlap is from `10` (max of 10 and 0) to `15` (min of 50 and 15), which is `5` minutes long. This is not enough for the `8-minute` duration.
     - Move the pointer `j` forward because `slots2[j][1]` is less than `slots1[i][1]`.

   - Second Comparison:

     - `slots1[i] = [10, 50]` and `slots2[j] = [25, 50]`
     - The overlap is from `25` to `50`, which is `25` minutes long and suffices for the `8-minute` duration.
     - We have found a suitable slot, so we can return the result `[25, 25 + 8]`, which is `[25, 33]`.

This result indicates that Person A and Person B can successfully schedule a meeting starting at `25` minutes past the hour and lasting until `33` minutes past the hour.

The method described above is efficient because it continuously seeks the earliest possible meeting time by looking for overlaps in the schedules and moves forward based on the end times of the current slots. As soon as a fitting time slot is found, it returns the result without unnecessary comparisons of later time slots, saving time and computation.

Python Solution

```python
from typing import List

class Solution:
    def minAvailableDuration(self, slots1: List[List[int]], slots2: List[List[int]], duration: int) -> List[int]:
        # Sort the time slots for both people to allow for easy comparison
        slots1.sort()
        slots2.sort()

        # Initialize variables to track current index in slots1 and slots2
        index1 = index2 = 0

        # Get the total number of slots for both people
        total_slots1 = len(slots1)
        total_slots2 = len(slots2)

        # Iterate over the slots until at least one person's slots are fully checked
        while index1 < total_slots1 and index2 < total_slots2:
            # Find the overlapping start time of the current slots
            overlap_start = max(slots1[index1][0], slots2[index2][0])
            # Find the overlapping end time of the current slots
            overlap_end = min(slots1[index1][1], slots2[index2][1])

            # If the overlapping period is greater than or equal to the required duration
            if overlap_end - overlap_start >= duration:
                # Return the start time and start time plus the duration
                return [overlap_start, overlap_start + duration]

            # If the end time in slots1 is before the end time in slots2,
            # increase the index for slots1 to check the next slot
            if slots1[index1][1] < slots2[index2][1]:
                index1 += 1
            else:
                # Otherwise, increase the index for slots2 to check the next slot
                index2 += 1

        # If no overlapping period long enough for the meeting is found, return an empty list
        return []

# Example usage:
# sol = Solution()
# result = sol.minAvailableDuration([[10, 50], [60, 120], [140, 210]], [[0, 15], [60, 70]], 8)
# print(result) # This will output: [60, 68]

```

Java Solution

```java
class Solution {
    public List<Integer> minAvailableDuration(int[][] slots1, int[][] slots2, int duration) {
        // Sort the time slots for both people based on the start times
        Arrays.sort(slots1, (a, b) -> a[0] - b[0]);
        Arrays.sort(slots2, (a, b) -> a[0] - b[0]);

        int index1 = 0; // Index for navigating person 1's time slots
        int index2 = 0; // Index for navigating person 2's time slots
        int len1 = slots1.length; // Total number of slots for person 1
        int len2 = slots2.length; // Total number of slots for person 2

        // Iterate through both sets of slots
        while (index1 < len1 && index2 < len2) {
            // Calculate the overlap start time
            int overlapStart = Math.max(slots1[index1][0], slots2[index2][0]);
            // Calculate the overlap end time
            int overlapEnd = Math.min(slots1[index1][1], slots2[index2][1]);

            // Check if the overlapping duration is at least the required duration
            if (overlapEnd - overlapStart >= duration) {
                // If so, return the start time of the meeting and start time plus duration
                return Arrays.asList(overlapStart, overlapStart + duration);
            }

            // Move to the next slot in the list that has an earlier end time
            if (slots1[index1][1] < slots2[index2][1]) {
                index1++;
            } else {
                index2++;
            }
        }

        // If no common slot is found that fits the duration, return an empty list
        return Collections.emptyList();
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    std::vector<int> minAvailableDuration(std::vector<std::vector<int>>& slots1, std::vector<std::vector<int>>& slots2, int duration)
        // Sort both sets of time slots in ascending order based on the start times
        std::sort(slots1.begin(), slots1.end());
        std::sort(slots2.begin(), slots2.end());

        // Use indices to track the current slot in each set
        int index1 = 0;
        int index2 = 0;

        // Loop through both sets of time slots to find a common free duration
        while (index1 < slots1.size() && index2 < slots2.size()) {
            // Find the latest start time and earliest end time between two slots
            int latestStart = std::max(slots1[index1][0], slots2[index2][0]);
            int earliestEnd = std::min(slots1[index1][1], slots2[index2][1]);

            // Check if the overlap duration between two slots is greater than or equal to the desired duration
            if (earliestEnd - latestStart >= duration) {
                // If so, return the start time and start time plus duration as the available duration
                return {latestStart, latestStart + duration};
            }

            // Move to the next slot in the set with the earlier ending time
            if (slots1[index1][1] < slots2[index2][1]) {
                ++index1;
            } else {
                ++index2;
            }
        }

        // If no common free duration is found, return an empty vector
        return {};
    }
};

```

Typescript Solution

```typescript
function minAvailableDuration(slots1: number[][], slots2: number[][], duration: number): number[] {
    // Sort both sets of time slots by start time in ascending order
    slots1.sort((a, b) => a[0] - b[0]);
    slots2.sort((a, b) => a[0] - b[0]);

    // Initialize pointers to traverse the slots
    let index1 = 0;
    let index2 = 0;

    while (index1 < slots1.length && index2 < slots2.length) {
        // Calculate the maximum start time and the minimum end time between two slots
        const latestStart = Math.max(slots1[index1][0], slots2[index2][0]);
        const earliestEnd = Math.min(slots1[index1][1], slots2[index2][1]);

        // Check if there is a slot sufficient for the required duration
        if (earliestEnd - latestStart >= duration) {
            // If found, return the time range starting from latestStart and continuing for the duration
            return [latestStart, latestStart + duration];
        }

        // Move to the next slot in the list with the earlier end time
        if (slots1[index1][1] < slots2[index2][1]) {
            index1++; // Move the pointer in the first list forward
        } else {
            index2++; // Move the pointer in the second list forward
        }
    }

    // If no matching slots were found, return an empty array
    return [];
}

// Example usage:
// const result = minAvailableDuration([[10, 50], [60, 120], [140, 210]], [[0, 15], [60, 70]], 8);
// console.log(result); // Should log a valid time slot with at least 8 minutes of duration, or an empty array if there's no slot.

```

Time and Space Complexity

Time Complexity

The given algorithm primarily consists of two parts: sorting the time slots and then iterating through the sorted lists to find a common available duration.

First, we sort both `slots1` and `slots2`. Assuming that `slots1` has `m` intervals and `slots2` has `n` intervals, the time taken to sort these lists using a comparison-based sorting algorithm, like Timsort (Python's default sorting algorithm), is `O(m log m)` for `slots1` and `O(n log n)` for `slots2`.

After sorting, we have a while loop that runs until we reach the end of one of the slot lists. In the worst case, we'll compare each pair of slots once, which leads to a complexity of `O(m + n)` since each list is traversed at most once.

The overall time complexity is the sum of the complexities of sorting and iterating through the slot lists, which is `O(m log m + n log n + m + n)`. However, the `log` factor dominates the linear factor in computational complexity analysis for large values. Hence, the time complexity simplifies to `O(m log m + n log n)`.

Space Complexity

The space complexity of the algorithm is determined by the space we use to sort `slots1` and `slots2`. Since the Timsort algorithm can require a certain amount of space for its operation, the space complexity is `O(m+n)` due to the auxiliary space used in sorting. All other operations use a constant amount of space, and no additional space is used that depends on the input size, so the space complexity remains `O(m+n)`.