252. Meeting Rooms

Easy Array Sorting

Problem Description

The problem presents a scenario in which an individual has multiple meetings scheduled, represented by time intervals. Each interval is an array of two values `[start_i, end_i]`, where `start_i` is the time a particular meeting begins, and `end_i` is the time the meeting ends. The `intervals` array contains all such meeting time intervals.

The task is to determine whether the person can attend all the scheduled meetings without any overlaps. More specifically, no two meetings can occur at the same time. The person can only attend all meetings if, for any two meetings, one finishes before the other starts.

Intuition

To solve this problem, it is essential first to understand that if the person can attend all meetings, then there cannot be any overlap between the time intervals. In other words, the end time of a meeting must be earlier than or equal to the start time of the subsequent meeting.

The intuition behind the solution is to first sort the meetings based on their start times. Sorting the meetings ensures that they are arranged sequentially, which makes it easier to check for any overlaps.

Once the meetings are sorted, we need to compare the end time of each meeting with the start time of the next one in the list. This is where the `pairwise` function is beneficial. It creates pairs of consecutive elements (in this case, meeting time intervals). We simply iterate through these pairs and check if the end time of the first meeting (`a[1]`) is less than or equal to the start time of the next meeting (`b[0]`).

By using the `all` function, which verifies that all elements in an iterable satisfy a particular condition, it is possible to ensure that for all pairs of consecutive meetings, the first one ends before the second one begins. If this condition is true for all pairs, the function returns `True`, and hence, the person can attend all meetings. Otherwise, it returns `False`, indicating a conflict in the meeting times.

Solution Approach

The solution's algorithm follows these steps:

1. **Sorting the intervals**: The input list named `intervals` is sorted in-place. Since `intervals` consists of sub-lists where each sub-list represents a meeting time interval `[start, end]`, the `sort()` method by default sorts the sub-lists based on the first elements, which are the start times of the meetings. This is crucial because it orders the meetings in the way they will occur through the day.
2. **Checking intervals in pairs**: After sorting, the `pairwise` function from Python's `itertools` is used to create an iterator that will return tuples containing pairs of consecutive elements (or intervals in this case). This means if the sorted `intervals` list has elements `[a, b, c, d, ...]`, `pairwise(intervals)` would give us an iterator over `(a, b), (b, c), (c, d), ...`.
3. **Using a generator expression with `all`**: The code includes a generator expression `all(a[1] <= b[0] for a, b in pairwise(intervals))` which iterates through every pair of consecutive meetings from the iterator and checks if the end time of the first meeting (`a[1]`) is less than or equal to the start time of the next meeting (`b[0]`). The function `all` will return `True` only if every evaluation within the expression yields `True`, meaning that there are no overlapping meetings. If any meeting overlaps, meaning the end time of one meeting is later than the start time of the following meeting, the expression will yield `False`, and thus the `all` function will short-circuit and return `False`.
4. **Returning the result**: The boolean result from the `all` function is returned directly. If the result is `True`, it means all the meetings do not overlap, and the individual can attend them all. If the result is `False`, it means there is at least one overlapping pair of meetings, which makes it impossible for the person to attend all of them.

In summary, this solution effectively leverages sorting and simple comparisons in pairs to determine whether meeting intervals overlap. The use of the `all` function along with a generator expression makes the implementation concise and efficient.

Example Walkthrough

Let's consider the following small example to illustrate the solution approach:

Assume we have the following meeting time intervals:

```
intervals = [[0, 30], [5, 10], [15, 20]]
```

Here, we have three meetings scheduled:

1. Meeting 1 from time 0 to time 30.
2. Meeting 2 from time 5 to time 10.
3. Meeting 3 from time 15 to time 20.

Now, let's go through the solution step by step:

1. **Sorting the intervals**: We sort the `intervals` based on their start times:

```
intervals.sort()  # Result: [[0, 30], [5, 10], [15, 20]]
```

After sorting, the order of `intervals` does not change because they were already sorted by their start times by default.

2. **Checking intervals in pairs**: We use the `pairwise` function from Python's `itertools` to consider the intervals in pairs for comparison:

```
from itertools import pairwise
# This pseudo operation would create iterator pairs: ([0, 30], [5, 10]), ([5, 10], [15, 20])
```

3. **Using a generator expression with `all`**: We then construct a generator expression with the `all` function to check for overlaps:

```
all(a[1] <= b[0] for a, b in pairwise(intervals))
# This checks: 30 <= 5 (for [0, 30], [5, 10]) and 10 <= 15 (for [5, 10], [15, 20])
```

During iteration:

- The first pair `([0, 30], [5, 10])` is evaluated. Since the end time of the first meeting `30` is not less than the start time of the second meeting `5`, the condition `a[1] <= b[0]` is `False`. Therefore, the `all` function would return `False`.
- Because `all` short-circuits (stops evaluating) as soon as it encounters a `False`, the result is determined without needing to evaluate the second pair `([5, 10], [15, 20])`.

4. **Returning the result**: The generator expression would return `False`, indicating that the person cannot attend all meetings because at least one pair of meetings overlaps.

In this example, the meetings at intervals `[0, 30]` and `[5, 10]` are overlapping, which means the person cannot attend both meetings. So the result for whether the individual can attend all meetings is `False`.

Solution Implementation

Python

```python
from typing import List
from itertools import tee

class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        # Function to generate consecutive pairs in a list
        def pairwise(iterable):
            "s -> (s0,s1), (s1,s2), (s2, s3), ..."
            a, b = tee(iterable)
            next(b, None)
            return zip(a, b)

        # Sort the intervals based on their start times.
        intervals.sort()

        # Check if there is any overlap between consecutive intervals.
        # If the end time of the first interval is greater than the start
        # time of the second interval, they overlap, which means
        # one cannot attend all meetings.
        # The `all` function returns True if all comparisons are True.
        return all(end_time <= start_time for (start_time, end_time), (next_start_time, _) in pairwise(intervals))
```

Java

```java
class Solution {

    // Function to check if a person can attend all meetings
    public boolean canAttendMeetings(int[][] intervals) {

        // Sort the intervals based on the start times
        Arrays.sort(intervals, (interval1, interval2) -> interval1[0] - interval2[0]);

        // Iterate through the sorted intervals to check for any overlaps
        for (int i = 1; i < intervals.length; ++i) {
            // Get the current and previous intervals
            int[] currentInterval = intervals[i];
            int[] previousInterval = intervals[i - 1];

            // If the end time of the previous interval is greater than the start time of the current one,
            // they overlap, so it's not possible to attend all meetings
            if (previousInterval[1] > currentInterval[0]) {
                return false;
            }
        }

        // If there are no overlaps, the person can attend all meetings
        return true;
    }
}
```

C++

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

class Solution {
public:
    // Function to determine if a person can attend all meetings
    bool canAttendMeetings(std::vector<std::vector<int>>& intervals) {
        // Sort the intervals based on their start times
        std::sort(intervals.begin(), intervals.end(), [](const std::vector<int>& a, const std::vector<int>& b) {
            return a[0] < b[0];
        });

        // Loop through the intervals to check for any overlaps
        for (int i = 1; i < intervals.size(); ++i) {
            // If the current interval's start time is less than the previous interval's end time
            if (intervals[i][0] < intervals[i - 1][1]) {
                // Overlap detected, cannot attend all meetings
                return false;
            }
        }

        // No overlaps found, can attend all meetings
        return true;
    }
};
```

TypeScript

```typescript
/**
 * Determines if an individual can attend all the given meetings.
 * Meetings are represented by intervals where each interval is an array of two integers,
 * with the start time as the first element and the end time as the second element.
 *
 * @param {number[][]} intervals - A list of intervals representing meetings.
 * @returns {boolean} - Returns true if an individual can attend all meetings without
 *                      any overlaps; otherwise, returns false.
 */
function canAttendMeetings(intervals: number[][]): boolean {
    // Sort the intervals based on their start time.
    intervals.sort((a, b) => a[0] - b[0]);

    // Iterate over the sorted intervals starting from the second interval.
    for (let i = 1; i < intervals.length; i++) {
        // Check if the current interval starts before the previous interval ends.
        // If so, there's an overlap, and the person cannot attend all meetings.
        if (intervals[i][0] < intervals[i - 1][1]) {
            return false;
        }
    }

    // If there were no overlaps found, return true.
    return true;
}
```

```python
from typing import List
from itertools import tee

class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        # Function to generate consecutive pairs in a list
        def pairwise(iterable):
            "s -> (s0,s1), (s1,s2), (s2, s3), ..."
            a, b = tee(iterable)
            next(b, None)
            return zip(a, b)

        # Sort the intervals based on their start times.
        intervals.sort()

        # Check if there is any overlap between consecutive intervals.
        # If the end time of the first interval is greater than the start
        # time of the second interval, they overlap, which means
        # one cannot attend all meetings.
        # The `all` function returns True if all comparisons are True.
        return all(end_time <= start_time for (start_time, end_time), (next_start_time, _) in pairwise(intervals))
```

Time and Space Complexity

Time Complexity

The time complexity of sorting the intervals is `O(n log n)`, where `n` is the number of intervals in the input list.

After sorting, the code iterates over the sorted list just once to compare the start time of an interval with the end time of the previous interval using the `pairwise` function. This iteration is `O(n)`.

Therefore, the overall time complexity is `O(n log n + n)` which simplifies to `O(n log n)` since the sorting part dominates the overall time complexity.

Space Complexity

The space complexity is `O(n)` if we take into account the space required for sorting the intervals, which usually requires temporary space proportional to the size of the list being sorted.

However, the use of `pairwise` in Python does not require additional space proportional to the number of intervals, as it generates pairs of consecutive items lazily. Thus, the additional space complexity due to `pairwise` is `O(1)`.

So, the overall space complexity is `O(n)`.