435. Non-overlapping Intervals

Medium Greedy Array Dynamic Programming Sorting Leetcode Link

Problem Description

The problem asks us to find the minimum number of intervals that need to be removed from a given list of time intervals to ensure that no two intervals overlap with each other. Intervals overlap when one interval starts before the other one ends. For example, if one interval is `[1, 2]` and another is `[2, 3]`, they do not overlap, but if the second interval is `[1, 3]`, they do overlap because the first interval has not ended before the second one starts.

Our input is an array of intervals, where each interval is represented as a list with two elements, signifying the start and the end times. The output should be an integer that represents the minimum number of intervals that must be removed to eliminate all overlaps.

Intuition

The key intuition behind the solution lies in the greedy algorithm approach. A greedy algorithm makes the locally optimal choice at each stage with the hope of finding the global optimum. In the context of this problem, the greedy choice would be to always pick the interval that ends the earliest, because this would leave the most room for subsequent intervals.

Here is the thinking process for arriving at the solution:

1. Sort the intervals based on their end times. This way, we encounter the intervals that finish earliest first and can thus make the greedy choice.
2. Start with the first interval, considering it as non-overlapping by default, and make a note of its end time.
3. Iterate through the subsequent intervals:
   - If the start time of the current interval is not less than the end time of the last non-overlapping interval, it means this interval does not overlap with the previously considered intervals. We can then update our last known end time to be the end time of the current interval.
   - If the start time is less than the last known end time, an overlap occurs, and we must choose to remove an interval. Following the greedy approach, we keep the interval with the earlier end time and remove the other by incrementing our answer (the number of intervals to remove).

By following these steps and always choosing the interval that finishes earliest, we ensure that we take up the least possible space on the timeline for each interval, and therefore maximize the number of intervals we can include without overlapping.

Solution Approach

The provided solution follows the greedy strategy mentioned in the reference solution approach. Let's discuss how the solution is implemented in more detail:

1. **Sorting Intervals**: The input list of intervals is sorted based on the end times of the intervals. This is done using Python's `sort` function with a lambda function as the key that retrieves the end time `x[1]` from each interval `x`.

   ```
   intervals.sort(key=lambda x: x[1])
   ```

   Sorting the intervals by their end times allows us to apply the greedy algorithm effectively.

2. **Initializing Variables**: Two variables are initialized:

   - `ans`: set to `0`, it keeps count of the number of intervals we need to remove.
   - `t`: set to the end time of the first interval, it represents the latest end time of the last interval that we decided to keep.
3. **Iterating Over Intervals**: The code iterates through the rest of the intervals starting from the second interval (since the first interval's end time is stored in `t`).

   ```
   for s, e in intervals[1:]:
   ```

   Each interval `[s, e]` consists of a start time `s` and an end time `e`.

4. **Checking for Overlapping**: In each iteration, the code checks if the current interval's start time `s` is greater than or equal to the variable `t`:

   - If `s >= t`, there is no overlap with the last chosen interval, so the current interval can be kept. We then update `t` to the current interval's end time `e`.

   - If `s < t`, there is an overlap with the last chosen interval, so the current interval needs to be removed, and we increment `ans` by 1.

   ```
   if s >= t:
       t = e
   else:
       ans += 1
   ```

5. **Returning the Result**: After the loop finishes, `ans` holds the minimum number of intervals that need to be removed to make all remaining intervals non-overlapping, so it is returned as the final result.

   ```
   return ans
   ```

By following the greedy approach, the algorithm ensures that the intervals with the earliest end times are considered first, minimizing potential overlap with future intervals and thus minimizing the number of intervals that need to be removed.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we are given the following list of intervals:

```
intervals = [[1, 2], [2, 3], [3, 4], [1, 3]]
```

We want to remove the minimum number of intervals so that no two intervals overlap. Here is how we apply the solution approach:

1. **Sorting Intervals**: First, we sort the intervals by their end times:

   ```
   sorted_intervals = [[1, 2], [2, 3], [1, 3], [3, 4]]
   ```

2. **Initializing Variables**: We initialize `ans = 0` and `t = 2`, where `t` is the end time of the first interval after sorting.

3. **Iterating Over Intervals**: We iterate through the intervals starting from the second one.

4. **Checking for Overlapping**:

   - Comparing the 2nd interval `[2, 3]` with `t`: since `2 >= 2`, we update `t` to 3, and no interval is removed.
   - Comparing the 3rd interval `[1, 3]` with `t`: since `1 < 3`, this interval overlaps with the previously chosen intervals. We increment `ans` to 1.
   - Comparing the 4th interval `[3, 4]` with `t`: since `3 >= 3`, we update `t` to 4, and no interval is removed.

After checking all intervals, the number of intervals to be removed is `ans = 1`. This is because the interval `[1, 3]` was overlapping, and by removing it, we ensured that no intervals overlap.

The output of our algorithm for this example would be `1`, indicating we need to remove a single interval to eliminate all overlaps.

Python Solution

```python
from typing import List

class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        # Sort the intervals based on the end time of each interval
        intervals.sort(key=lambda interval: interval[1])

        # Initialize the count of removed intervals and set the time to compare against
        removed_intervals_count = 0
        end_time = intervals[0][1]

        # Iterate through the intervals starting from the second one
        for start, end in intervals[1:]:
            # If the current interval does not overlap, update the `end_time`
            if start >= end_time:
                end_time = end
            # If the interval overlaps, increment the count of intervals to remove
            else:
                removed_intervals_count += 1

        # Return the total count of removed intervals to avoid overlap
        return removed_intervals_count

```

Java Solution

```java
import java.util.Arrays; // Required for Arrays.sort
import java.util.Comparator; // Required for Comparator

class Solution {
    public int eraseOverlapIntervals(int[][] intervals) {
        // Sort the intervals array based on the end time of each interval
        Arrays.sort(intervals, Comparator.comparingInt(a -> a[1]));

        // Set 'end' as the end time of the first interval
        int end = intervals[0][1];

        // Initialize 'overlaps' to count the number of overlapping intervals
        int overlaps = 0;

        // Iterate through each interval starting from the second one
        for (int i = 1; i < intervals.length; i++) {
            // If the current interval does not overlap with the previous, update 'end'
            if (intervals[i][0] >= end) {
                end = intervals[i][1];
            } else {
                // If the current interval overlaps, increment 'overlaps'
                overlaps++;
            }
        }

        // Return the total number of overlapping intervals to be removed
        return overlaps;
    }
}

```

C++ Solution

```cpp
#include <vector> // Required for using the vector container
#include <algorithm> // Required for using the sort algorithm

class Solution {
public:
    // Function to find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        // Sort the intervals based on their end time.
        sort(intervals.begin(), intervals.end(), [](const auto& a, const auto& b) {
            return a[1] < b[1];
        });

        int nonOverlappingCount = 0;   // To keep track of the number of non-overlapping intervals
        int currentEndTime = intervals[0][1]; // Track the end time of the last added interval

        // Iterate over all intervals starting from the second one
        for (int i = 1; i < intervals.size(); ++i) {
            // If the current interval starts after the end of the last added interval, it does not overlap
            if (currentEndTime <= intervals[i][0]) {
                currentEndTime = intervals[i][1]; // Update the end time to the current interval's end time
            } else {
                // If the current interval overlaps, increment the nonOverlappingCount
                ++nonOverlappingCount;
            }
        }

        // Return the total number of overlapping intervals to remove
        return nonOverlappingCount;
    }
};

```

Typescript Solution

```typescript
function eraseOverlapIntervals(intervals: number[][]): number {
    // Sort the intervals based on the ending time of each interval
    intervals.sort((a, b) => a[1] - b[1]);

    // Initialize the end variable to the end of the first interval
    let lastIntervalEnd = intervals[0][1];

    // Initialize counter for the number of intervals to remove
    let intervalsToRemove = 0;

    // Iterate through the intervals starting from the second one
    for (let i = 1; i < intervals.length; i++) {
        // Current interval being considered
        let currentInterval = intervals[i];

        // Check if the current interval overlaps with the last non-overlapping interval
        if (lastIntervalEnd > currentInterval[0]) {
            // If it overlaps, increment the removal counter
            intervalsToRemove++;
        } else {
            // If it doesn't overlap, update the end to be the end of the current interval
            lastIntervalEnd = currentInterval[1];
        }
    }

    // Return the number of intervals that need to be removed to eliminate all overlaps
    return intervalsToRemove;
}

```

Time and Space Complexity

The given Python code aims to find the minimum number of intervals that can be removed from a set of intervals so that none of them overlap. Here's an analysis of its time complexity and space complexity:

Time Complexity

The time complexity of the code is primarily determined by the sorting operation and the single pass through the sorted interval list:

1. `intervals.sort(key=lambda x: x[1])` sorts the intervals based on the end time. Sorting in Python uses the Timsort algorithm, which has a time complexity of $O(n \log n)$, where n is the number of intervals.
2. The `for` loop iterates through the list of intervals once. The loop runs in $O(n)$ as each interval is visited exactly once.

Combining these steps, the overall time complexity is dominated by the sorting step, leading to a total time complexity of $O(n \log n)$.

Space Complexity

The space complexity of the code is mainly the space required to hold the input and the variables used for the function:

1. In-place sorting of the intervals doesn't require additional space proportional to the input size, so the space complexity of the sorting operation is $O(1)$.
2. The only extra variables that are used include `ans`, and `t`. These are constant-size variables and do not scale with the input size.

Hence, the overall space complexity of the code is $O(1)$, indicating it requires constant additional space.