1288. Remove Covered Intervals

Medium Array Sorting Leetcode Link

Problem Description

The problem presents a scenario in which you're given a list of intervals, with each interval represented as a pair of integers `[start, end)`. The notation `[start, end)` signifies that the interval includes `start` and goes up to but does not include `end`. The goal is to remove intervals that are "covered" by other intervals. An interval `[a, b)` is considered covered by another interval `[c, d)` if both `c <= a` and `b <= d`. In simple terms, if one interval lies completely within the boundary of another, the inner interval is said to be covered by the outer interval. Your task is to find out how many intervals remain after removing all the covered intervals.

Intuition

The intuitive approach to solving this problem involves checking each interval against all others to see if it is covered by any. However, this would be inefficient, particularly for large lists of intervals. Thus, the key to an efficient solution is to sort the intervals in a way that makes it easier to identify and remove the covered intervals.

By sorting the intervals first by their start times and then by their end times in descending order, we line up intervals in such a way that once we find an interval not covered by the previous one, none of the following intervals will cover it either. This sorting strategy allows us to only compare each interval with the last one that wasn't removed.

With the sorted list, we iterate through the intervals, where the main idea is to compare the current interval's `end` with the previous interval's `end`. If the current `end` is greater, this interval isn't covered by the previous interval. We count this interval, and it becomes the new previous interval for subsequent comparisons. If the current `end` is not greater, it means this interval is covered by the previous interval, and we do not count it and proceed to the next interval. After checking all intervals, the count `cnt` gives the number of intervals that remain.

Solution Approach

The solution makes use of a greedy algorithm approach, where we aim to remove the minimal number of intervals by keeping the ones that aren't covered by any other. Let's walk through the solution approach along with the algorithms, data structures, or patterns used:

1. **Sorting**: We begin by sorting the `intervals` list using a custom sort key. This key sorts the intervals primarily by their `start` times in ascending order. If two intervals have the same `start` time, we sort them by their `end` times in descending order. This ensures that we have the longer intervals earlier if the `start` times are the same, making it easier to identify covered intervals.
2. **Initializing Counters**: After sorting, we initialize our count `cnt` to 1. This is because we consider the first interval as not being covered by any previous one, as there are no previous intervals yet. We also initialize `pre` to keep track of the last interval that was not covered by the one before it.
3. **Iterating Through Intervals**: We loop through the sorted intervals starting from the second one. At each iteration, we compare the current interval's `end` time (`e[1]`) with the `end` time of `pre` (`pre[1]`):
   - If `pre[1]` (previous interval's end time) is less than `e[1]` (current interval's end time), it indicates that the current interval is not completely covered by `pre`. Hence, we increment our count `cnt` and update `pre` to the current interval.
   - If `pre[1]` is greater than or equal to `e[1]`, the current interval is covered by `pre`, and we don't increment `cnt`.
4. **Return the Result**: After iterating through all intervals, `cnt` holds the count of intervals that remain after removing all that are covered by another interval.

It's important to understand that by sorting the intervals first by starting time and then by the ending time (in opposite order), we limit the comparison to only the previous interval and the current one in the loop. This greatly reduces the number of comparisons needed, resulting in a more efficient algorithm.

Below is the reference code implementation based on this approach:

```python
class Solution:
    def removeCoveredIntervals(self, intervals: List[List[int]]) -> int:
        # Sort intervals by start time, and then by end time in descending order
        intervals.sort(key=lambda x: (x[0], -x[1]))

        cnt, pre = 1, intervals[0]  # Initialize the counter and the previous interval tracker

        # Iterate through each interval in the sorted list
        for e in intervals[1:]:
            # If the current interval's end time is greater than the previous', it's not covered
            if pre[1] < e[1]:
                cnt += 1    # Increment the counter, as this interval isn't covered
                pre = e     # Update the 'pre' interval tracker to the current interval

        return cnt  # Return the final count of non-covered intervals
```

With the combination of a clever sorting strategy and a single loop, this code neatly solves the problem in an efficient manner.

Example Walkthrough

Let's go through a small example to illustrate the solution approach described.

Suppose we have a list of intervals: `[[1,4), [2,3), [3,6)]`.

Following our solution approach:

1. **Sorting**: Applying our custom sort, the list is sorted by `start` times in ascending order, and for those with the same `start` time, by `end` times in descending order. But since all our example intervals have different `start` times, we only need to sort by the first element:

Sorted list: `[[1,4), [2,3), [3,6)]`

2. **Initializing Counters**: We initialize our count `cnt` to 1, assuming the first interval `[1,4)` is not covered. We also initialize `pre` with this interval.
3. **Iterating Through Intervals**:
   - We first compare `[1,4)` with `[2,3)`. Here, `pre[1]` is 4 and `e[1]` is 3. Since `pre[1] >= e[1]`, we find that `[2,3)` is covered by `[1,4)` and thus, we do not increment `cnt`.
   - Next, we compare `[1,4)` with `[3,6)`. Here, `pre[1]` is 4 and `e[1]` is 6. Since `pre[1] < e[1]`, we find that `[3,6)` is not covered by `[1,4)`. So, we increment `cnt` to 2 and update `pre` to `[3,6)`.
4. **Return the Result**: Having finished iterating through the intervals, we find that the count `cnt` is 2, which means there are two intervals that remain after removing all that are covered: `[[1,4), [3,6)]`.

This example walk-through demonstrates the efficiency of the algorithm by effectively reducing the number of necessary comparisons and clearly exhibiting how the sorting step greatly simplifies the process of identifying covered intervals.

Python Solution

```python
from typing import List

class Solution:
    def removeCoveredIntervals(self, intervals: List[List[int]]) -> int:
        # Sort intervals based on the start time; in case of a tie, sort by end time in descending order
        intervals.sort(key=lambda interval: (interval[0], -interval[1]))

        # Initialize count of non-covered intervals to 1, as the first one can never be covered by others
        non_covered_count = 1
        # The first interval is the current reference for comparison
        previous_interval = intervals[0]

        # Iterate over the sorted intervals starting from the second interval
        for current_interval in intervals[1:]:
            # If the end of the current interval is greater than the end of the previous interval,
            # it's not covered, and we should update the count and reference interval
            if previous_interval[1] < current_interval[1]:
                non_covered_count += 1
                previous_interval = current_interval

        # Return the final count of non-covered intervals
        return non_covered_count

```

Java Solution

```java
class Solution {
    public int removeCoveredIntervals(int[][] intervals) {
        // Sort the intervals. First by the start in ascending order.
        // If the starts are equal, sort by the end in descending order.
        Arrays.sort(intervals, (a, b) -> {
            if (a[0] == b[0]) {
                return b[1] - a[1];
            } else {
                return a[0] - b[0];
            }
        });

        // Initialize the previous interval as the first interval
        int[] previousInterval = intervals[0];
        // Count the first interval
        int count = 1;

        // Iterate through all intervals starting from the second one
        for (int i = 1; i < intervals.length; ++i) {
            // If the current interval's end is greater than the previous interval's end,
            // it means the current interval is not covered by the previous interval.
            if (previousInterval[1] < intervals[i][1]) {
                // Increment the count of non-covered intervals.
                ++count;
                // Update the previous interval to the current interval.
                previousInterval = intervals[i];
            }
            // If the current interval's end is not greater than the previous interval's end,
            // it's covered by the previous interval and we do nothing.
        }

        // Return the number of non-covered intervals
        return count;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include algorithm header for std::sort

// Definition for the Solution class with removeCoveredIntervals method
class Solution {
public:
    int removeCoveredIntervals(vector<vector<int>>& intervals) {
        // Sort the intervals. First by the start time, and if those are equal, by the
        // end time in descending order (to have the longer intervals come first).
        sort(intervals.begin(), intervals.end(), [](const vector<int>& intervalA, const vector<int>& intervalB) {
            if (intervalA[0] == intervalB[0]) // If the start times are the same
                return intervalB[1] < intervalA[1]; // Sort by end time in descending order
            return intervalA[0] < intervalB[0]; // Otherwise sort by start time
        });

        int countNotCovered = 1; // Initialize count of intervals not covered by others.
        vector<int> previousInterval = intervals[0]; // Store the first interval as the initial previous interval

        // Iterate through the intervals starting from the second
        for (int i = 1; i < intervals.size(); ++i) {
            // If the current interval is not covered by the previous interval
            if (previousInterval[1] < intervals[i][1]) {
                ++countNotCovered; // Increment count as this interval is not covered
                previousInterval = intervals[i]; // Update the previous interval to current interval
            }
        }

        // Return the count of intervals that are not covered by other intervals
        return countNotCovered;
    }
};

```

Typescript Solution

```typescript
// You might need to install the required type definitions for running this code in a TypeScript environment.

// Importing necessary functionalities from standard libraries
import { sort } from 'algorithm';

// Interval type definition for better type clarity
type Interval = [number, number];

// A function that sorts the intervals based on certain criteria
function sortIntervals(intervals: Interval[]): Interval[] {
    return intervals.sort((intervalA, intervalB) => {
        if (intervalA[0] === intervalB[0]) // If start times are the same
            return intervalB[1] - intervalA[1]; // Sort by end time in descending order
        return intervalA[0] - intervalB[0]; // Otherwise sort by start time
    });
}

// Function to remove covered intervals
function removeCoveredIntervals(intervals: Interval[]): number {
    // Sort the intervals using the defined sorting function
    const sortedIntervals = sortIntervals(intervals);

    let countNotCovered = 1; // Initialize count of intervals not covered by others
    let previousInterval = sortedIntervals[0]; // Store the first interval as the initial previous interval

    // Iterate through the sorted intervals starting from the second one
    for (let i = 1; i < sortedIntervals.length; i++) {
        // If the current interval is not covered by the previous interval
        if (previousInterval[1] < sortedIntervals[i][1]) {
            countNotCovered++; // Increment count as this interval is not covered
            previousInterval = sortedIntervals[i]; // Update the previous interval to current interval
        }
    }

    // Return the number of intervals that are not covered by other intervals
    return countNotCovered;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is dominated by two operations: the sorting of the intervals and the single pass through the sorted list.

1. The `sort()` function in Python uses the Timsort algorithm, which has a time complexity of `O(n log n)` where `n` is the number of intervals.
2. After sorting, the code performs a single pass through the list to count the number of non-covered intervals, which has a time complexity of `O(n)`.

Combining these two steps, the overall time complexity is `O(n log n + n)`. Simplified, it remains `O(n log n)` because the `n log n` term is dominant.

Space Complexity

The space complexity refers to the amount of extra space or temporary storage that an algorithm requires.

1. Sorting the list is done in-place, which means it doesn't require additional space proportional to the input size. Therefore, the space complexity due to sorting is constant, `O(1)`.
2. Aside from the sorted list, the algorithm only uses a fixed number of variables (`cnt`, `pre`, and `e`) which also take up constant space.

Hence, the overall space complexity is `O(1)`, because no additional space that scales with the size of the input is used.