57. Insert Interval

Medium Array Leetcode Link

Problem Description

The problem requires us to handle a collection of intervals, with each interval represented by a start and end time (inclusive). We're provided with an array of these intervals, and each interval is guaranteed not to overlap with any other. The intervals are sorted in ascending order based on their start times. Our task is to insert a new interval into this array while maintaining the order and the non-overlapping property of the intervals. If inserting the new interval causes any overlap, we must merge the overlapping intervals into a single interval that covers all the ranges.

Our goal is to return the new list of intervals after the insertion has been completed correctly.

Intuition

To solve this problem, we must first understand how to merge intervals. Merging involves combining overlapping intervals into one that spans the entire range covered by any overlapping intervals. Since the existing collection of intervals is already sorted and non-overlapping, they can be left as they are. We only need to focus on the `newInterval`.

We start by simply appending `newInterval` to our intervals array. Even though this may break the sorting, we're going to merge any intervals that need it anyway, which will handle any issues caused by the insertion.

Now, the `merge` process begins. We assume we have at least one interval in the array to start with. We'll compare each interval with the last interval in our answer list. There are two cases to consider:

1. If there is no overlap (the current interval's start time is greater than the last interval's end time in the answer list), we can safely add the current interval to the answer list.
2. If there is an overlap (the current interval's start time is less than or equal to the last interval's end time in the answer list), we merge by updating the end time of the last interval in the answer list. The end time after the merge will be the maximum end time between the overlapped intervals.

The `merge` function sorts the updated intervals array and performs the above steps to ensure that the resulting array has no overlaps. It's important to note that sorting is required only if the `newInterval` was inserted in such a way that it breaks the original order. Since we append `newInterval` directly, sorting is indeed necessary as the first step of merging. After the merge is completed, the answer list will be returned, representing the intervals after the new interval has been inserted correctly.

Solution Approach

The solution to the problem follows a fairly straightforward algorithm, making use of simple list operations and the concept of interval merging. Here is a breakdown of the approach based on the provided Python code:

1. Append `newInterval` to `intervals`: Before we can merge the intervals to eliminate any overlaps, we need to include the `newInterval` into the existing intervals list. This is straightforward as we can use the `append()` method to add `newInterval` to `intervals`. At this stage, we're not concerned with maintaining the sorted order of `intervals` because our next step is to explicitly sort the list.

   ```
   intervals.append(newInterval)
   ```

2. Define a `merge` function: The purpose of this function is to merge any overlapping intervals. We expect `merge` to handle all cases, even if we call it with an unsorted or overlapping list. The function first sorts the intervals list, which is necessary since we've just appended a `newInterval` at the end without regard to sorting.

   ```python
   def merge(intervals: List[List[int]]) -> List[List[int]]:
       intervals.sort()
       # ... rest of the merging logic goes here
   ```

3. Initialize the answer list `ans` with the first interval: Given that `merge` starts with a sorted list, we know that no interval before the first can overlap with it. Hence, we can safely initialize `ans` with just the first interval.

   ```
   ans = [intervals[0]]
   ```

4. Iterate over the rest of the intervals and merge if necessary: This step is the core of the merging logic. For each interval (after the first), we compare it with the last interval in `ans`. If the intervals do not overlap, we append the current interval to `ans`. If they do overlap, we update the end time of the last interval in `ans` to be the maximum of its own end and the current interval's end.

   ```python
   for s, e in intervals[1:]:
       if ans[-1][1] < s:
           ans.append([s, e])
       else:
           ans[-1][1] = max(ans[-1][1], e)
   ```

5. Return the answer list `ans` after all intervals have been processed: After the loop concludes, `ans` contains the merged intervals and is returned as the final result of the `insert` function.

   ```
   return ans
   ```

The overall time complexity of this approach is dominated by the sorting operation, which is `O(n log n)` where `n` is the number of intervals, including the new interval. However, if the intervals were already sorted (aside from the appended `newInterval`), this could potentially be optimized to `O(n)` by carefully inserting `newInterval` into the correct position. Since this optimization is not presented in the given solution, it remains an academic point here. The space complexity is `O(n)` as well, since we are generating a list of intervals as output.

By calling the `merge` function after appending `newInterval` to `intervals`, we handle the task in a single pass through the sorted list of intervals, making efficient use of the fact that we only need to check each interval against the last one in the `ans` list for potential merging.

Example Walkthrough

Let's walk through a simple example to illustrate the solution approach described.

Suppose we have the following set of intervals already sorted and non-overlapping:

```
Current intervals: [[1,2], [3,5], [6,7], [8,10], [12,16]]
```

The task is to insert a new interval `[4,9]` into this set while maintaining the sorted order and non-overlapping intervals. Here's how the algorithm handles this:

1. Append `newInterval` to `intervals`: Initially, our list of intervals is:

   ```
   [[1,2], [3,5], [6,7], [8,10], [12,16]]
   ```

   After appending the new interval, it becomes:

   ```
   [[1,2], [3,5], [6,7], [8,10], [12,16], [4,9]]
   ```

2. Sort the intervals: After appending the `newInterval`, the list of intervals has to be sorted again since the `newInterval` was added at the end without considering the order. The intervals list after sorting:

   ```
   [[1,2], [3,5], [4,9], [6,7], [8,10], [12,16]]
   ```

3. Initialize the answer list `ans` with the first interval: The `ans` list starts as:

   ```
   [[1,2]]
   ```

4. Iterate over the rest of the intervals and merge if necessary: The second interval in the sorted list is `[3,5]`. There is no overlap with `[1,2]`, so it is appended to `ans`:

   ```
   [[1,2], [3,5]]
   ```

   Next, we encounter the `newInterval` which is `[4,9]`. Since it overlaps with `[3,5]`, we merge them by updating the end time of `[3,5]` to the end time of `[4,9]`:

   ```
   [[1,2], [3,9]]
   ```

   The next interval `[6,7]` is already covered by `[3,9]`, so no new interval is added, but the same interval remains. The interval `[8,10]` also gets merged into `[3,9]`, resulting in:

   ```
   [[1,2], [3,10]]
   ```

   Finally, the interval `[12,16]` does not overlap with `[3,10]`, and is appended to `ans`:

   ```
   [[1,2], [3,10], [12,16]]
   ```

5. Return the answer list `ans` after all intervals have been processed: The final `ans` returned by the algorithm is:

   ```
   [[1,2], [3,10], [12,16]]
   ```

This illustrates how the intervals are correctly merged and the final list of intervals is obtained by following the solution approach.

Python Solution

```python
from typing import List

class Solution:
    def insert(self, intervals: List[List[int]], new_interval: List[int]) -> List[List[int]]:
        # This function merges overlapping intervals.
        def merge(intervals: List[List[int]]) -> List[List[int]]:
            # First we sort the intervals based on the starting times.
            intervals.sort(key=lambda x: x[0])
            merged_intervals = [intervals[0]]  # Initialize with the first interval.

            # Iterate through the rest of the intervals to merge overlapping ones.
            for start, end in intervals[1:]:
                # If the current interval does not overlap with the last merged interval.
                if merged_intervals[-1][1] < start:
                    merged_intervals.append([start, end])  # Keep it separate.
                else:
                    # They overlap, so we merge them by updating the end time of
                    # the last interval in the merged list if needed.
                    merged_intervals[-1][1] = max(merged_intervals[-1][1], end)

            # Return the merged list of intervals.
            return merged_intervals

        # Add the new interval to the existing list of intervals.
        intervals.append(new_interval)

        # Call the merge function to merge any overlapping intervals including the new one.
        return merge(intervals)

```

Java Solution

```java
class Solution {

    // Function to insert a new interval into an existing list of intervals
    public int[][] insert(int[][] intervals, int[] newInterval) {
        // Initialize an expanded array to hold the existing intervals and the new interval
        int[][] expandedIntervals = new int[intervals.length + 1][2];

        // Copy existing intervals into the expanded array
        for (int i = 0; i < intervals.length; ++i) {
            expandedIntervals[i] = intervals[i];
        }

        // Add the new interval to the end of the expanded intervals array
        expandedIntervals[intervals.length] = newInterval;

        // Merge overlapping intervals and return the result
        return merge(expandedIntervals);
    }

    // Helper function to merge overlapping intervals
    private int[][] merge(int[][] intervals) {
        // Sort the intervals based on the starting times
        Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

        // List to hold the merged intervals
        List<int[]> mergedIntervals = new ArrayList<>();

        // Add the first interval to the list as initialization
        mergedIntervals.add(intervals[0]);

        // Iterate through each interval and merge if necessary
        for (int i = 1; i < intervals.length; ++i) {
            // Get the start and end times of the current interval
            int start = intervals[i][0];
            int end = intervals[i][1];

            // Get end time of the last interval in the list.
            int lastEnd = mergedIntervals.get(mergedIntervals.size() - 1)[1];

            // If the current interval does not overlap with the previous, simply add it
            if (lastEnd < start) {
                mergedIntervals.add(intervals[i]);
            } else {
                // Otherwise, merge the current interval with the previous one by updating the end time
                mergedIntervals.get(mergedIntervals.size() - 1)[1] = Math.max(lastEnd, end);
            }
        }

        // Convert the list back into an array and return
        return mergedIntervals.toArray(new int[mergedIntervals.size()][]);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Method to insert a new interval into the list of existing intervals and then merge them
    vector<vector<int>> insert(vector<vector<int>>& intervals, vector<int>& newInterval) {
        // Add the new interval to the end of the intervals vector
        intervals.emplace_back(newInterval);
        // Merge the updated list of intervals
        return merge(intervals);
    }

    // Method to merge overlapping intervals in a list
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        // Sort the intervals in ascending order based on the start times
        sort(intervals.begin(), intervals.end());

        // This will hold the merged intervals
        vector<vector<int>> mergedIntervals;

        // Add the first interval to the merged list as a starting point
        mergedIntervals.emplace_back(intervals[0]);

        // Iterate through the intervals starting with the second interval
        for (int i = 1; i < intervals.size(); ++i) {
            // If the current interval does not overlap with the last interval in the merged list,
            // it means we can add it as a new entry to the merged list
            if (mergedIntervals.back()[1] < intervals[i][0]) {
                mergedIntervals.emplace_back(intervals[i]);
            } else {
                // If they overlap, merge the current interval with the last interval of the merged list
                // by updating the ending time of the last interval in mergedList
                mergedIntervals.back()[1] = max(mergedIntervals.back()[1], intervals[i][1]);
            }
        }

        // Return the list of merged intervals
        return mergedIntervals;
    }
};

```

Typescript Solution

```typescript
function insert(intervals: number[][], newInterval: number[]): number[][] {
    let [start, end] = newInterval;  // Destructure the start and end of the new interval
    const result: number[][] = [];   // Initialize an array to hold the merged intervals
    let inserted = false;            // Flag to check if new interval has been added

    // Loop through each interval in the sorted list
    for (const [intervalStart, intervalEnd] of intervals) {
        if (end < intervalStart) { // If the end of newInterval is before the current interval
            if (!inserted) {       // And if newInterval hasn't been inserted yet
                result.push([start, end]); // Insert the newInterval
                inserted = true;           // Set the flag as inserted
            }
            result.push([intervalStart, intervalEnd]); // Add the current interval
        } else if (intervalEnd < start) { // If the current interval ends before newInterval starts
            result.push([intervalStart, intervalEnd]); // Add the current interval
        } else {                           // If intervals overlap
            start = Math.min(start, intervalStart); // Merge intervals by taking the min start
            end = Math.max(end, intervalEnd);       // And max end
        }
    }
    // If newInterval was not inserted, add it to the end
    if (!inserted) {
        result.push([start, end]);
    }

    return result; // Return the merged list of intervals
}

```

Time and Space Complexity

Time Complexity

The given code consists of two main operations: sorting the list of intervals, and then merging these intervals. Here's how each operation contributes to the total time complexity:

1. **Sorting**: 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. Since we are appending a new interval before sorting, the sorting step will operate on `n + 1` intervals, but this does not change the overall complexity class, so it remains `O(n log n)`.

2. **Merging**: The `merge` function iterates through the sorted list of intervals once to combine overlapping intervals. This is a linear pass, which means it runs in `O(n)` time, considering `n` as the number of intervals including the new one we added.

Combining both steps, the time complexity of the algorithm is dominated by the sorting step, so the overall time complexity is `O(n log n)`.

Space Complexity

The space complexity is determined by the extra space used by the algorithm. In this case, we have:

1. The additional list `ans` that is initially a copy of the first interval, and worst-case, could be extended to include all intervals if none overlap. This results in a worst-case space complexity of `O(n)`.

2. The in-place `sort()` generally has a space complexity of `O(1)` for the actual sorting since Timsort is a hybrid stable sorting algorithm that takes advantage of the existing order in the list. Yet, it might require a temporary space of up to `O(n)` in the worst case when merging runs. But since we are considering the space for the output as separate, we do not count this towards additional space.

Thus, the overall space complexity of the algorithm is `O(n)`.