56. Merge Intervals

Medium Array Sorting Leetcode Link

Problem Description

The given problem requires us to merge all the overlapping intervals in a list. An interval is represented as a list with two elements where the first element is the start and the second is the end of the interval (`[start, end]`). An "overlap" occurs when one interval's start is less than or equal to the end of another interval. The goal is to simplify the list of intervals to a list where no intervals overlap, ensuring that the new list collectively spans the same range as the original intervals.

Here's an example for clarification:

Original list of intervals: `[[1,3],[2,6],[8,10],[15,18]]`

After merging all the overlapping intervals, we get: `[[1,6],[8,10],[15,18]]`

In the merged intervals, there is no pair of intervals such that one overlaps with another.

Intuition

The intuition behind the solution comes from two realizations:

1. If we sort the intervals based on their start times, any overlapping intervals will be placed next to each other in the list.
2. To merge intervals, we only need to track the end time since the sorted order ensures that the next interval's start time is always greater than or equal to the current interval's start time.

The approach is as follows:

1. Sort the list of intervals based on their start times.
2. Initialize a new list to hold the merged intervals and add the first interval to it.
3. Iterate through the rest of the intervals, and for each one, compare its start time with the end time of the last interval in the merged list.
4. If the start time is greater than the end time of the last interval in the merged list, then there is no overlap, and we can add the current interval as a new entry to the merged list.
5. If there is overlap (the start time is less than or equal to the end time), we update the end time of the last interval in the merged list to be the maximum of the end times of the last interval and the current interval.
6. The process continues until we have gone through all the intervals.
7. We return the merged list of intervals as the answer.

This solution guarantees that we merge all overlapping intervals and result in a list of intervals with no overlaps.

Solution Approach

The solution to the problem involves sorting the intervals and then iterating through the sorted list to merge any overlapping intervals.

Here's a step-by-step breakdown of the implementation:

1. First, we sort the given list of intervals. This is done in-place using the native sort function provided by Python, which sorts the intervals based on their first element (the start times).

   ```
   intervals.sort()
   ```

   By sorting the intervals, we are able to take advantage of the fact that any intervals that might need merging will appear next to each other in the list.

2. We then initialize a new list called `ans`, which will store our merged intervals, and we start by adding the first interval to it.

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

   This acts as a comparison base for merging subsequent intervals.

3. We then proceed to iterate over the rest of the intervals, starting from the second interval onward, to check for overlapping with the currently last interval in our `ans` list.

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

   Here, `(s, e)` represents the start and end times of the current interval we are looking at.

4. If the start time `s` of the current interval is greater than the end time of the last interval in `ans`, it means there is no overlap and we can simply add this interval to `ans`.

   ```
   if ans[-1][1] < s:
       ans.append([s, e])
   ```

5. However, if an overlap exists, we need to merge the current interval with the last interval in `ans`. To do this, we update the end time of the last interval in `ans` with the maximum end time between the two overlapping intervals.

   ```
   else:
       ans[-1][1] = max(ans[-1][1], e)
   ```

   This ensures that the intervals are merged, covering the overlapping time spans without duplicating intervals.

6. Once we finish iterating through all intervals, the `ans` list contains the merged intervals. We return `ans` as the final set of non-overlapping intervals.

The algorithm uses the sort-merge pattern, which is common for interval problems. By sorting and then merging, we bring the overall run-time complexity down to O(N log N) where N is the number of intervals, with the sort contributing to the log N factor and the merge process being linear in nature. Regarding data structures, the solution leverages lists and the use of tuple unpacking for readability.

Example Walkthrough

Let's take a small example to illustrate the solution approach with the provided intervals: `[[5,7],[1,3],[3,4],[2,6]]`.

Sorting the Intervals

First, we need to sort the intervals by their starting points to align any intervals that might overlap:

```
Before sort: [[5,7],[1,3],[3,4],[2,6]]
After sort:  [[1,3],[2,6],[3,4],[5,7]]
```

We used `intervals.sort()` to achieve this.

Initializing and Iterating for Merging

We then initialize the `ans` list with the first sorted interval, treating it as the base for our merged intervals:

```
ans = [[1,3]]
```

Next, we start iterating from the second element of the sorted intervals:

- We look at `[2,6]` and compare it to the last element of `ans`, which is `[1,3]`. Since the start `2` is within `[1,3]` (as `3` is greater than `2`), they overlap. We merge them by updating the end of the last interval in `ans` to the max end of both intervals, now `ans` becomes `[[1,6]]`.

- Proceeding to `[3,4]`, we compare it to the last element `[1,6]`. Again, it overlaps because `4` is not greater than `6`. No need to change the end time since `6` is already the maximum end.

- Finally, we look at `[5,7]`. This does not overlap with `[1,6]` as `5` is not greater than `6`. Since `7` is greater than `6`, we add `[5,7]` as a new interval to `ans`. After the addition, `ans` becomes `[[1,6],[5,7]]`.

However, our merging logic must have the current start to be greater than the last end to avoid overlap. Therefore, we should adjust the last step:

- `[5,7]` is checked again and it actually overlaps with `[1,6]` (since `5` is less than or equal to `6`). We merge them by updating the end of the last interval in `ans` to `7`, and `ans` now becomes `[[1,7]]`.

Final Merged List

Having completed the iteration over the sorted intervals, we have a list of merged intervals where no two intervals overlap. The final `ans` is:

```
[[1,7]]
```

This means we successfully merged all intervals to cover the same range without having any overlapping intervals. The solution approach, by sorting and then merging, streamlined the process and ensures an efficient way to obtain the merged intervals.

Python Solution

```python
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        # Sort the interval list based on the start times of intervals
        intervals.sort()

        # Initialize the merged_intervals list with the first interval
        merged_intervals = [intervals[0]]

        # Iterate over the intervals, starting from the second interval
        for start, end in intervals[1:]:
            # Check if the current interval does not overlap with the last interval in merged_intervals
            if merged_intervals[-1][1] < start:
                # If it does not overlap, add the current interval to merged_intervals
                merged_intervals.append([start, end])
            else:
                # If it does overlap, merge the current interval with the last one by
                # updating the end time of the last interval to the maximum end time seen so far
                merged_intervals[-1][1] = max(merged_intervals[-1][1], end)

        # Return the merged intervals
        return merged_intervals

```

Java Solution

```java
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;

class Solution {

    // Method to merge overlapping intervals.
    public int[][] merge(int[][] intervals) {
        // Sort the intervals by their starting times.
        Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

        // List that holds the merged intervals.
        List<int[]> mergedIntervals = new ArrayList<>();
        // Add the first interval to the list as starting interval for merging.
        mergedIntervals.add(intervals[0]);

        // Loop through all the intervals starting from the second one.
        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 the last interval in the merged list.
            int[] lastMergedInterval = mergedIntervals.get(mergedIntervals.size() - 1);

            // Check if there is an overlap with the last interval in the merged list.
            if (lastMergedInterval[1] < start) {
                // No overlap, so we can add the current interval as it is.
                mergedIntervals.add(intervals[i]);
            } else {
                // Overlap exists, so we extend the last interval's end time.
                lastMergedInterval[1] = Math.max(lastMergedInterval[1], end);
            }
        }

        // Convert the merged intervals list to a 2D array and return it.
        return mergedIntervals.toArray(new int[mergedIntervals.size()][]);
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to merge overlapping intervals
    std::vector<std::vector<int>> merge(std::vector<std::vector<int>>& intervals) {
        // First, sort the intervals based on the starting times
        std::sort(intervals.begin(), intervals.end());

        // This will be the result vector for merged intervals
        std::vector<std::vector<int>> mergedIntervals;

        // Initialize the result vector with the first interval
        mergedIntervals.push_back(intervals[0]);

        // Iterate through all the intervals starting from the second one
        for (int i = 1; i < intervals.size(); ++i) {
            // If the current interval does not overlap with the last interval in the result,
            // then simply add the current interval to the result
            if (mergedIntervals.back()[1] < intervals[i][0]) {
                mergedIntervals.push_back(intervals[i]);
            } else {
                // If there is an overlap, merge the current interval with the last interval
                // in the result by updating the end time to the maximum end time seen
                mergedIntervals.back()[1] = std::max(mergedIntervals.back()[1], intervals[i][1]);
            }
        }
        // Return the merged intervals
        return mergedIntervals;
    }
};

```

Typescript Solution

```typescript
function merge(intervals: number[][]): number[][] {
    // First, we sort the intervals array based on the start times
    intervals.sort((a, b) => a[0] - b[0]);

    // Initialize the merged intervals array with the first interval
    const mergedIntervals: number[][] = [intervals[0]];

    // Iterate through the intervals starting from the second element
    for (let i = 1; i < intervals.length; ++i) {
        // Get the last interval in the mergedIntervals array
        const lastInterval = mergedIntervals[mergedIntervals.length - 1];

        // If the current interval does not overlap with the last interval in mergedIntervals
        if (lastInterval[1] < intervals[i][0]) {
            // Add the current interval to the mergedIntervals array as it cannot be merged
            mergedIntervals.push(intervals[i]);
        } else {
            // If there is an overlap, merge the current interval with the last interval
            // by updating the end time of the last interval to the maximum end time
            lastInterval[1] = Math.max(lastInterval[1], intervals[i][1]);
        }
    }

    // Return the array containing all the merged intervals
    return mergedIntervals;
}

```

Time and Space Complexity

Time Complexity

The given code has two main operations:

1. Sorting the `intervals` list.
2. Iterating through the sorted list and merging overlapping intervals.

For a list of `n` intervals:

- The sort operation typically has a complexity of `O(n log n)`, since Python uses TimSort (a hybrid sorting algorithm derived from merge sort and insertion sort) for sorting lists.
- The iteration over the list has a complexity of `O(n)`, because we go through the intervals only once.

Hence, the total time complexity is the sum of these two operations, which is `O(n log n) + O(n)`. Since `O(n log n)` is the higher order term, it dominates the total time complexity, which simplifies to `O(n log n)`.

Space Complexity

The space complexity consists of the additional space used by the algorithm apart from the input:

- The `ans` list which contains the merged intervals is the main additional data structure used, and in the worst case, if no intervals overlap, it will contain `n` intervals.
- Since the `ans` list reuses the intervals from the original input list, and the input list size itself is not included in the additional space used for computing space complexity, the space used to store `ans` can be considered `O(1)` (constant space) in this context.

Thus, the space complexity is `O(1)`.