Merge Intervals Documentation

Overview

The "Merge Intervals" problem involves merging overlapping intervals in a given array of intervals. Each interval is represented as a pair [start, end], where start and end are non-negative integers. The goal is to merge overlapping intervals and return a list of non-overlapping intervals that cover all the intervals in the input.

Problem Statement

Given an array of intervals `intervals`, where `intervals[i] = [starti, endi]`, merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

Example

Input

intervals1 = [[1,3],[2,6],[8,10],[15,18]]

intervals2 = [[1,4],[4,5]]

Output

Output1: [[1,6],[8,10],[15,18]]

Output2: [[1,5]]

Approach

The solution to this problem involves sorting the intervals based on their start time. After sorting, we iterate through the sorted intervals, merging overlapping intervals and appending non-overlapping intervals to a new list. To merge overlapping intervals, we compare the end time of the last merged interval with the start time of the current interval. If they overlap, we update the end time of the last merged interval to the maximum of the current interval's end time and the last merged interval's end time. If they do not overlap, we simply append the current interval to the merged list.

Example usage:

intervals1 = [[1,3],[2,6],[8,10],[15,18]]

solution = Solution()

print(solution.merge(intervals1)) # Output: [[1,6],[8,10],[15,18]]

intervals2 = [[1,4],[4,5]]

print(solution.merge(intervals2)) # Output: [[1,5]]

Complexity Analysis

- **Time Complexity:** Sorting the intervals takes O(n log n) time, where n is the number of intervals. The merging process takes O(n) time. Thus, the overall time complexity is O(n log n) + O(n), which simplifies to O(n log n) due to the dominating sorting operation.

- **Space Complexity:** We use additional space to store the merged intervals, which could be as large as the input itself. Hence, the space complexity is O(n).