3.Hard\07.Merge_overlapping_subinterval.cpp

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

Example 1:
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

APPROACH:
To merge overlapping intervals, we can follow these steps:
1. Sort the intervals based on the start time.
2. Initialize a vector `ans` to store the merged intervals.
3. Add the first interval from the sorted intervals to the `ans` vector.
4. Iterate through the remaining intervals:
   - If the start time of the current interval is less than or equal to the end time of the last interval in the `ans` vector, it means they overlap. Update the end time of the last interval in the `ans` vector if necessary.
   - If the start time of the current interval is greater than the end time of the last interval in the `ans` vector, it means they don't overlap. Add the current interval to the `ans` vector.
5. Return the `ans` vector as the merged non-overlapping intervals.

CODE:
*/

vector<vector<int>> merge(vector<vector<int>>& intervals) {
    sort(intervals.begin(), intervals.end());
    vector<vector<int>> ans;
    ans.push_back(intervals[0]);

    for(int i = 1; i < intervals.size(); i++){
        if(ans.back()[1] >= intervals[i][0]){
            ans.back()[1] = max(ans.back()[1], intervals[i][1]);
        }
        else{
            ans.push_back(intervals[i]);
        }
    }

    return ans;
}

/*
TIME COMPLEXITY: O(nlogn), where n is the number of intervals in the input.
The sorting step takes O(nlogn) time, and the merging step takes O(n) time.
Overall, the time complexity is dominated by the sorting step.
SPACE COMPLEXITY: O(n), where n is the number of intervals in the input.
We are using additional space to store the merged intervals in the `ans` vector.
*/
```

```
47
48
```