



LeetCode 56 — Merge Intervals

1. Problem Title & Link

- **Title:** LeetCode 56 — Merge Intervals
- **Link:** <https://leetcode.com/problems/merge-intervals/>

2. Problem Statement (Short Summary)

You are given an array of intervals $[[\text{start}, \text{end}]]$.

Your task is to **merge all overlapping intervals** and return the result.

Overlapping example:

$[1,3]$ overlaps with $[2,6]$

Merged $\rightarrow [1,6]$

3. Examples (Input → Output)

Example 1

Input: intervals = $[[1,3],[2,6],[8,10],[15,18]]$

Output: $[[1,6],[8,10],[15,18]]$

Example 2

Input: intervals = $[[1,4],[4,5]]$

Output: $[[1,5]]$

Example 3

Input: intervals = $[[1,4],[0,2],[3,5]]$

Output: $[[0,5]]$

4. Constraints

- $1 \leq \text{intervals.length} \leq 10^4$
- $\text{intervals}[i].length == 2$
- Intervals may be in any order
- Must return sorted, merged intervals

5. Core Concept (Pattern / Topic)

Greedy + Sorting

Steps:

1. Sort intervals by **start time**
2. Iterate through sorted intervals
3. Merge if overlapping
4. Else push as a new interval



Because sorting gives global order, merging becomes linear afterwards.

6. Thought Process (Step-by-Step Explanation)

1. Sort intervals based on start
2. Create a result list; push the first interval
3. For each next interval:
 - o Compare with last interval in result
 - o If overlapping → merge
 - o Else → push new interval
4. Return result

Overlap Condition

Two intervals $[s_1, e_1]$ and $[s_2, e_2]$ overlap **iff**:

$$s_2 \leq e_1$$

Merge Operation

$$\text{merged_end} = \max(e_1, e_2)$$

7. Visual / Intuition Diagram

```
Input:
[1,3], [2,6], [8,10], [15,18]
```

Sort:

```
[1,3], [2,6], [8,10], [15,18]
```

Merge process:

```
[1,3] + [2,6] → [1,6]
```

```
[1,6], [8,10] → no overlap → add
```

```
[8,10], [15,18] → add
```

Final:

```
[1,6], [8,10], [15,18]
```

8. Pseudocode

```
sort intervals by start

res = [intervals[0]]

for each interval in intervals[1:]:
    last = res[-1]
    if interval.start <= last.end:
        last.end = max(last.end, interval.end)
    else:
        res.append(interval)
```



```
return res
```

9. Code Implementation

✓ Python

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort() # sort by start

        merged = [intervals[0]]

        for curr in intervals[1:]:
            last = merged[-1]
            if curr[0] <= last[1]: # overlap
                last[1] = max(last[1], curr[1])
            else:
                merged.append(curr)

        return merged
```

✓ Java

```
class Solution {
    public int[][] merge(int[][] intervals) {
        Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

        List<int[]> result = new ArrayList<>();
        int[] current = intervals[0];
        result.add(current);

        for (int[] interval : intervals) {
            int currStart = interval[0];
            int currEnd = interval[1];
            int lastEnd = current[1];

            if (currStart <= lastEnd) { // overlap
                current[1] = Math.max(lastEnd, currEnd);
            } else {
                current = interval;
                result.add(current);
            }
        }
    }
}
```



```

        return result.toArray(new int[result.size()][]);
    }
}

```

10. Time & Space Complexity

Metric	Complexity
Time	$O(n \log n)$ — sorting dominates
Space	$O(n)$ — output list

11. Common Mistakes / Edge Cases

- ✗ Not sorting first
- ✗ Wrong overlap condition
- ✗ Modifying original intervals incorrectly
- ✗ Not updating last interval properly

Edge cases:

- Single interval
- All intervals disjoint
- All intervals overlapping
- Intervals inside other intervals

Example: [1,10], [2,3], [4,9] → [1,10]

12. Detailed Dry Run (Step-by-Step Table)

Input:

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

Step	current	merged[-1]	Overlap?	Action
Start	[1,3]	—	—	merged = [[1,3]]
1	[2,6]	[1,3]	$2 \leq 3 \rightarrow \text{YES}$	merge → [1,6]
2	[8,10]	[1,6]	$8 \leq 6 \rightarrow \text{NO}$	push → [8,10]
3	[15,18]	[8,10]	$15 \leq 10 \rightarrow \text{NO}$	push → [15,18]

Final:

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



13. Common Use Cases

- Calendar event merging
- Reservation systems
- Combining booking time slots
- Merging time ranges
- Data compression of ranges

14. Common Traps

- Forgetting to update last interval
- Using wrong condition: curr.start < last.end (should be \leq)
- Not handling nested intervals
- Sorting incorrectly (descending instead of ascending)

15. Builds To (Related Problems)

- **LC 57** — Insert Interval
- **LC 435** — Non-overlapping intervals
- **LC 452** — Minimum number of arrows to burst balloons
- **LC 253** — Meeting Rooms II

16. Alternate Approaches + Comparison

Approach	Time	Space	Notes
Sorting + Merge	$O(n \log n)$	$O(n)$	Standard, best
Sweep Line	$O(n \log n)$	$O(n)$	More complex
Brute force	$O(n^2)$	$O(1)$	Too slow

17. Why This Solution Works (Short Intuition)

Sorting ensures intervals are processed in increasing order of start time, so merging becomes a simple linear process.

18. Variations / Follow-Up Questions

- Insert a new interval (LC 57)
- Count non-overlapping intervals (LC 435)
- Find intersections of two interval lists
- Merge intervals on a circular timeline