Topic Introduction

Welcome back, fellow coder! Today, we're jumping into a classic family of interview questions: **interval problems**. These are the ones with meeting times, ranges, bookings, and overlapping periods. If you've ever had to schedule meetings or merge booking slots, you've already done these in real life!

What's the Core Concept Here?

The magic tool for most interval problems is the **sweep line technique** (sometimes called "line sweeping") combined with **sorting**. Here's the idea:

- **Intervals:** Think of an interval as a pair `[start, end]`, representing a chunk of time or a range.
- **Sorting:** We usually sort intervals by their start time. This puts them in the order they appear on a timeline.
- **Sweep Line:** Imagine moving a vertical line from left to right across a number line. As it passes each interval, you track overlaps, merges, or other properties.

Why is this useful in interviews?

- Many real-world systems (like calendars, event schedulers, and memory allocators) need to manage overlapping ranges.
- Interviewers love these problems because they test your ability to reason about ordering, merging, and edge cases.

Example (not one of our target problems):

Suppose you have intervals: `[1, 3], [2, 4], [5, 7]`. If you sort by start time and walk left to right, you can easily see where overlaps happen, or where gaps occur.

Let's see this concept in action with three classic problems:

- Merge Intervals
- Insert Interval
- Meeting Rooms II

Why are these grouped together?
All three are about working with intervals on a number line. They use sorting and, often, the sweep line to find overlaps or to merge. Mastering these will make you quick and confident when any interval-based question comes your way!

Problem 1: Merge Intervals

[LeetCode: Merge Intervals](https://leetcode.com/problems/merge-intervals/)

Problem Statement (Rephrased)

You're given a list of intervals, each as `[start, end]`. Your task is to merge all overlapping intervals and return a new list of non-overlapping intervals that cover all the input intervals.

Example:

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

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

Explanation: `[1,3]` and `[2,6]` overlap and merge into `[1,6]`.

Try this one by hand:

Input: `[[1,4],[4,5]]`

What should the output be?

Brute-Force Approach

Check every pair of intervals to see if they overlap, and merge them if they do. Keep doing this until no more merges are possible.

- **Time Complexity:** O(N^2) — since for each interval, you might need to check every other interval.
- **Drawback:** Not efficient for large input.

Optimal Approach: Sort and Merge

Pattern: Sort the intervals by start time, and then sweep left to right, merging as you go.

Step-by-Step:

- **Sort** the intervals by their start time.
- **Initialize** an empty result list.
- **Iterate** through the intervals:
    - If the result list is empty, or the last interval in result does **not** overlap with the current interval, just append the current interval.
    - If they **do** overlap, merge them by updating the end of the last interval in result to be `max(last_end, current_end)`.

Let's see this in Python:

```
def merge(intervals):
    # Sort intervals by start time
    intervals.sort(key=lambda x: x[0])
    merged = []

    for interval in intervals:
        # If merged is empty, or current interval does not overlap with last, append
        if not merged or merged[-1][1] < interval[0]:
            merged.append(interval)
        else:
            # Overlap: merge by updating the end time
            merged[-1][1] = max(merged[-1][1], interval[1])
    return merged
```

Time Complexity:

- Sorting: O(N log N)
- Merging: O(N)
- **Total:** O(N log N)

Space Complexity:

- O(N) for the output list.

Explanation of the Code

- **intervals.sort(key=lambda x: x[0])** sorts the intervals by their start value.
- **merged** keeps our results.
- **for interval in intervals:** walks through each interval.
- If **merged** is empty or if the last merged interval ends before the current starts, we add the current.
- Otherwise, intervals overlap: we update the end value to be the maximum of the current and last merged interval.

Trace Example

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

- After sorting: `[[1,3],[2,6],[8,10],[15,18]]`
- Start with `[1,3]` in merged.
- Next, `[2,6]` overlaps with `[1,3]` (since 2 <= 3). Merge to `[1,6]`.
- Next, `[8,10]` does not overlap with `[1,6]`. Add as a new interval.
- Next, `[15,18]` does not overlap with `[8,10]`. Add as a new interval.

Final result: `[[1,6],[8,10],[15,18]]`

Try this one:
Input: `[[1,4],[0,4]]`
What's the output?

Take a moment to draw this out and try coding it yourself before peeking at the solution!

Problem 2: Insert Interval

[LeetCode: Insert Interval](https://leetcode.com/problems/insert-interval/)

Problem Statement (Rephrased)

Given a sorted list of non-overlapping intervals, and a single new interval to insert, insert it into the correct position so that the list remains sorted and non-overlapping. Merge if necessary.

Example:

Input: `intervals = [[1,3],[6,9]]`, `newInterval = [2,5]`

Output: `[[1,5],[6,9]]`

Try this:

Input: `intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]]`, `newInterval = [4,8]`

What's the output?

How is this Similar or Different?

- **Similar:** We're still merging intervals based on overlaps.
- **Different:** We're inserting a new interval and merging only where needed, not across the entire list.

Brute-Force Approach

Add the new interval to the list, sort, and then use the merge intervals algorithm from Problem 1.

- **Time Complexity:** O(N log N) for sorting, O(N) for merging.

Optimal Approach: One-Pass Merge

Since input intervals are already sorted and non-overlapping, we can do this in a single pass:

Step-by-Step:

- Initialize an empty result list.
- Iterate through the intervals:
    - If the current interval ends before the new interval starts, add it to result.
    - If the current interval starts after the new interval ends, add the new interval to result (if not already added), then add the current interval.
    - If the current interval overlaps with the new interval, merge them by updating the new interval's start to `min(current.start, new.start)` and end to `max(current.end, new.end)`.
- After the loop, if the new interval hasn't been added, add it.

Pseudocode:

```
result = []
for interval in intervals:
    if interval.end < newInterval.start:
        add interval to result
    else if interval.start > newInterval.end:
        if newInterval not added:
            add newInterval to result
            mark as added
        add interval to result
    else:
        newInterval.start = min(interval.start, newInterval.start)
```

```
        newInterval.end = max(interval.end, newInterval.end)
if newInterval not added:
    add newInterval to result
return result
```

Example Trace:

Input: intervals = `[[1,2],[3,5],[6,7],[8,10],[12,16]]`, newInterval = `[4,8]`

- `[1,2]` is before `[4,8]`, add `[1,2]`.
- `[3,5]` overlaps with `[4,8]` — merge to `[3,8]`.
- `[6,7]` overlaps with `[3,8]` — merge to `[3,8]`.
- `[8,10]` overlaps with `[3,8]` — merge to `[3,10]`.
- `[12,16]` is after `[3,10]`, so add merged `[3,10]`, then `[12,16]`.

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

Try this one:
Input: intervals = `[[1,5]]`, newInterval = `[2,7]`
What's the output?

Time Complexity: O(N) — single pass through intervals
Space Complexity: O(N) — for the result list

Problem 3: Meeting Rooms II

[LeetCode: Meeting Rooms II](https://leetcode.com/problems/meeting-rooms-ii/)

Problem Statement (Rephrased)

Given a list of meeting time intervals consisting of start and end times, find the minimum number of meeting rooms required so that no meetings overlap.

Example:
Input: `[[0, 30], [5, 10], [15, 20]]`
Output: `2`
Explanation: You need at least 2 rooms since `[0,30]` and `[5,10]` overlap, and `[15,20]` overlaps with one of the ongoing meetings.

Try this one:
Input: `[[7,10],[2,4]]`
What's the output?

What's Different or More Challenging?

Here, instead of merging intervals, we care about the **maximum number of overlapping intervals** at any time.

Brute-Force Approach

For each interval, count how many other intervals overlap with it, and keep track of the maximum.

- **Time Complexity:** O(N^2) — not efficient!

Optimal Approach: Sweep Line with Two Pointers

Pattern:

- Separate all start and end times.
- Sort both arrays.
- Walk through the starts, using a pointer for ends.
    - If a meeting starts before the earliest ending meeting ends, you need a new room.
    - If a meeting starts after or at the same time as an ending, a room is freed up.

Pseudocode:

```
starts = sorted([interval[0] for interval in intervals])
ends = sorted([interval[1] for interval in intervals])
rooms = 0
end_ptr = 0
for i in range(len(intervals)):
    if starts[i] < ends[end_ptr]:
        rooms += 1
    else:
        end_ptr += 1
return rooms
```

Example Trace:

Input: `[[0, 30], [5, 10], [15, 20]]`

- starts: [0, 5, 15]
- ends: [10, 20, 30]
- 0 < 10: room 1
- 5 < 10: room 2
- 15 >= 10: end_ptr += 1 (free a room), 15 < 20: room 2

Result: 2

Try this one:

Input: `[[2, 7], [3, 5], [6, 9], [8, 10]]`

How many rooms do you need?

Time Complexity: O(N log N) — for sorting

Space Complexity: O(N) — for the start and end arrays

Summary and Next Steps

What did we learn?

• These three problems — Merge Intervals, Insert Interval, Meeting Rooms II — all rely on understanding how to process and combine intervals.

• **Key patterns:** Sort, scan, and merge; or, for overlaps, use separate start/end arrays and sweep line logic.

• **Common mistakes:** Forgetting to sort, missing edge cases (like touching intervals), or adding/removing intervals at the wrong time.

Tips for interviews:

• Always clarify if intervals are closed or open.

• Draw a diagram when stuck — it helps clarify overlaps and merges.

• Watch out for off-by-one errors with interval endpoints.

Action Steps:

• Solve all three problems on your own — even the one with code provided.

• Try solving Insert Interval and Meeting Rooms II using a different technique (like using a heap for Meeting Rooms II).

• Explore other interval problems (e.g., Employee Free Time, Interval List Intersections).

• Compare your solutions with others for edge-case handling and style.

• Don’t worry if you get stuck — keep practicing and you’ll master these patterns!

Happy coding — and may your intervals never overlap (unless you want them to)!