Topic Introduction

Today, we’re diving into a classic pattern in technical interviews: **interval manipulation**. Intervals are simply pairs of numbers representing a range — for example, `[1, 4]` means "everything from 1 to 4, inclusive." In coding interviews, you're often given a list of such intervals, and asked to merge, insert, or rearrange them in some way.

What is an interval problem?
An interval problem is any problem that involves working with ranges and figuring out how they overlap, combine, or fit together. These problems are everywhere: scheduling meetings, booking rooms, merging calendar events, or finding gaps in a timeline.

How does interval manipulation work?
The key idea is to sort the intervals (usually by their start times), then process them one by one, making decisions based on whether two intervals overlap or not. This pattern lets you efficiently merge, insert, or remove intervals while maintaining order.

Why are interval problems important in interviews?
They test your ability to think about data as ranges, to spot patterns, and to use sorting and linear scanning to solve problems efficiently. They also reveal how well you can handle edge cases and overlapping data.

Quick Example (not one of our main problems):
Suppose you have intervals `[1, 2]`, `[3, 5]`, and `[4, 6]`. If you want to merge overlapping intervals, you’d first sort by start: `[1,2]`, `[3,5]`, `[4,6]`. Then, you’d see that `[3,5]` and `[4,6]` overlap, so you’d merge them into `[3,6]`. The result: `[1,2]`, `[3,6]`.

Let's look at three popular problems that use these ideas:

- Merge Intervals
- Insert Interval
- Non-overlapping Intervals

Why are these grouped together?
All three involve manipulating a list of intervals:

- Merge Intervals combines overlapping intervals into one.
- Insert Interval adds a new interval and merges if needed.
- Non-overlapping Intervals asks for the minimum number of removals needed to eliminate all overlaps.

We’ll start with the simplest, build on the pattern, and then tackle a trickier variation.

Problem 1: Merge Intervals

Problem Statement (Rephrased):
Given a list of intervals where each interval is a pair `[start, end]`, merge all overlapping intervals and return a list of the merged intervals.
Leetcode link

Example Input/Output:

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

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

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

Let’s walk through it:

- Start with `[1,3]`. Next is `[2,6]`. They overlap (since 2 <= 3), so merge into `[1,6]`.
- Next is `[8,10]`. No overlap with `[1,6]`, so keep as is.
- Finally, `[15,18]` also does not overlap.

Try this case yourself:

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

What’s the result?

Brute-force Approach:

Check every pair of intervals for overlaps, merge if needed, repeat until no more merges.

- Time complexity: O(n^2)
- Not efficient!

Optimal Approach:

- **Pattern:** Sort intervals by start time, then scan and merge as you go.
- **Step-by-step:**
    - Sort intervals by start.
    - Initialize an empty result list.
    - For each interval:
        - If the result list is empty, or current interval does *not* overlap with the last result, append it.
        - If it *does* overlap, merge by updating the last interval's end to `max(last_end, current_end)`.

Take a moment to try this on pen and paper for the example above!

Python Solution

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

    for interval in intervals:
        # If merged is empty, or no overlap, add interval
        if not merged or merged[-1][1] < interval[0]:
            merged.append(interval)
        else:
            # Overlapping: merge by updating the end of the last interval
            merged[-1][1] = max(merged[-1][1], interval[1])

    return merged
```

- **Time complexity:** O(n log n) for sorting, O(n) for the scan. So, O(n log n) overall.
- **Space complexity:** O(n) for the result list.

How does this work?

- The sort guarantees intervals are in order.
- The loop checks overlap: if the current interval starts after the previous ends, they're separate. Otherwise, merge.
- The `merged[-1][1] = max(...)` ensures we always extend the interval as far as needed.

Trace Example:

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

- Sorted: `[[1,3], [2,6], [8,10], [15,18]]`
- Start with `[1,3]` in merged.
- `[2,6]`: 2 <= 3, so merge to `[1,6]`.
- `[8,10]`: 8 > 6, so append.
- `[15,18]`: 15 > 10, so append.
- Result: `[[1,6], [8,10], [15,18]]`

Try this one yourself:

Input: `[[1,4], [2,3], [5,7], [6,8]]`

Take a moment to solve this on your own before checking your answer. Remember: pen and paper is your friend!

Did you know?

This merge pattern is the backbone for many interval problems. Try to spot it as you practice!

Problem 2: Insert Interval

Problem Statement (Rephrased):

You are given a list of non-overlapping intervals sorted by start time, and a new interval. Insert the new interval into the list, merging if necessary, so the result is still sorted and has no overlaps.

Leetcode link

Example Input/Output:

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

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

How is this similar to Merge Intervals?

- Both require combining overlapping intervals.
- But here, only one interval is being added, and intervals are already sorted and non-overlapping.

Thought Process:

- Add intervals before the new interval.
- Merge all intervals that overlap with the new interval.
- Add the rest.

Brute-force:

Insert, then call the merge-intervals solution.

- Time: O(n log n) for sort + O(n) for merge.

Optimal Approach:

- Since the intervals are already sorted and non-overlapping, we can do this in one pass:
    - Add all intervals before the new interval (no overlap).
    - Merge overlapping intervals (those where interval.start <= newInterval.end).
    - Add the merged interval.
    - Add all intervals after the new interval.

Pseudocode:

```
result = []
i = 0
n = number of intervals

// 1. Add all intervals before newInterval
while i < n and intervals[i].end < newInterval.start:
    result.append(intervals[i])
    i += 1

// 2. Merge all overlapping intervals with newInterval
while i < n and intervals[i].start <= newInterval.end:
    newInterval.start = min(newInterval.start, intervals[i].start)
    newInterval.end = max(newInterval.end, intervals[i].end)
    i += 1

// 3. Add the merged interval
result.append(newInterval)

// 4. Add the remaining intervals
while i < n:
    result.append(intervals[i])
    i += 1

return result
```

Example Walkthrough:

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

- Step 1: `[1,3]` overlaps with `[2,5]` (since 3 >= 2). So nothing added yet.
- Step 2: Merge `[1,3]` and `[2,5]` to `[1,5]`.
- Step 3: `[6,9]` does not overlap, so just add it.
- Result: `[[1,5], [6,9]]`

Try this case yourself:

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

What should the output be?

Complexity:

- Time: O(n), one pass through intervals.
- Space: O(n), for the result.

Trace Example:

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

- Add `[1,2]` (before new interval).
- `[3,5]` overlaps with `[4,8]` (since 3 <= 8), so merge: min(3,4)=3, max(5,8)=8. Now newInterval is `[3,8]`.
- `[6,7]` also overlaps, merge: min(3,6)=3, max(8,7)=8.
- `[8,10]` overlaps, merge: min(3,8)=3, max(10,8)=10. Now `[3,10]`.
- `[12,16]` does not overlap, so add.
- Result: `[[1,2],[3,10],[12,16]]`

Try another: intervals = `[[1,5]]`, newInterval = `[2,3]`

Problem 3: Non-overlapping Intervals

Problem Statement (Rephrased):

Given a list of intervals, find the minimum number of intervals you need to remove so that the rest are non-overlapping.

Leetcode link

How is this different?

- Instead of merging or inserting, you want to remove the fewest intervals to eliminate all overlaps.
- This requires making choices about which intervals to keep.

Brute-force:

Try all combinations of intervals to remove.

- Time: Exponential, not practical.

Optimal Approach:

- This is a classic greedy problem.
- **Pattern:**
    - Sort intervals by end time (not start!).
    - Always pick the interval with the earliest end, and remove ones that overlap with it.
- **Why by end time?**
    - By always picking the interval that ends first, you leave more room for the following intervals, minimizing removals.

Pseudocode:

```
sort intervals by end time
count = 0
prev_end = intervals[0].end

for i from 1 to n-1:
    if intervals[i].start < prev_end:
        // Overlap, must remove this one
        count += 1
    else:
        // No overlap, move window
        prev_end = intervals[i].end

return count
```

Example:

Input: `[[1,2],[2,3],[3,4],[1,3]]`

- Sorted by end: `[[1,2],[1,3],[2,3],[3,4]]`
- Keep `[1,2]` (prev_end = 2)
- `[1,3]` starts at 1 < 2, so overlaps. Remove (count = 1)
- `[2,3]` starts at 2 == 2, no overlap. Keep (prev_end = 3)
- `[3,4]` starts at 3 == 3, no overlap. Keep (prev_end = 4)
- Total removed: 1

Try this case:

Input: `[[1,2],[1,2],[1,2]]`

Complexity:

- Time: O(n log n) for sort, O(n) for scan.
- Space: O(1) (just variables).

Implement and test this for:

Input: `[[1,100],[11,22],[1,11],[2,12]]`

Reflect:

Notice how switching from start-time to end-time sorting completely changes what the greedy choice looks like. Can you think of other problems where this "earliest end time" trick applies?

Summary and Next Steps

Today, you worked through three core interval manipulation problems:

- Merging overlapping intervals.
- Inserting and merging a new interval.
- Removing the minimum number of intervals to eliminate overlaps.

Key patterns and insights:

- **Sorting is often step one.** Getting intervals in order (start or end) makes processing easier.
- **Greedy scanning:** For merging/inserting, focus on start times. For minimal removals, focus on end times.
- **Edge cases:** Intervals that just touch (`[a, b]` and `[b, c]`) may or may not be considered overlapping based on the problem. Always check problem definitions!

Common traps:

- Forgetting to sort, or sorting by the wrong value (start vs end).
- Off-by-one errors: intervals that just touch.
- Not handling the last interval in the list.

Action List

- Solve all three interval problems on your own, including the one we coded together.
- Try Problem 2 and 3 using different approaches, for instance:
    - For Problem 2, what if you used the regular merge intervals method after inserting?
    - For Problem 3, what happens if you sort by start time instead?
- Seek out other interval problems (like "Meeting Rooms" or "Minimum Number of Arrows to Burst Balloons").
- Compare your answers with other people's solutions — pay special attention to how they handle edge cases.
- Get stuck? That’s okay! The point is to keep practicing and learning. Each attempt builds your skills.

Intervals are everywhere, and mastering these tricks pays off in many interviews. See you in the next PrepLetter!