Topic Introduction

Today, we're diving into a fundamental pattern in coding interviews: **Working with Intervals**. Intervals are simply ranges, like meeting times on a calendar or time blocks when a room is available. Coding problems involving intervals ask you to reason about overlaps, gaps, and conflicts—skills that are invaluable for scheduling systems, booking apps, and more.

What are Intervals?

An **interval** is a pair of numbers `[start, end]` representing a continuous span—say, `[2, 5]` means everything from 2 (inclusive) to 5 (inclusive or exclusive, based on the problem). In interviews, you'll often see lists of intervals and be asked to merge, intersect, or identify conflicts among them.

Why Intervals Matter

Most real-world scheduling or booking features rely on interval logic. Interviewers love these problems because they test:

- Your ability to reason about overlaps and gaps.
- Your skill at traversing two (or more) sorted lists efficiently.
- Your understanding of sorting, merging, and greedy algorithms.

A Simple Example

Suppose you have intervals `[1, 3]`, `[5, 7]`, and `[2, 4]`. After merging overlapping intervals, you'd get `[1, 4]` and `[5, 7]`.

Introducing Today's Problems

We'll explore three classic interval problems:

- **Interval List Intersections** (LeetCode 986): Find all overlaps between two lists of intervals.
- **Employee Free Time** (LeetCode 759): Given a schedule for each employee, find times when everyone is free.
- **My Calendar I** (LeetCode 729): Implement a calendar that can book new intervals only if they don't overlap with existing bookings.

Why group these? All three require you to reason about overlapping intervals and scheduling conflicts. The first asks you to find overlaps directly, the second to infer *free* time (gaps between intervals), and the third to efficiently check if a new interval can be added without conflict.

Let's get started!

Problem 1: Interval List Intersections

Problem Statement (Rephrased):

Given two lists of non-overlapping, sorted intervals, return their intersections. Each interval is a pair `[start, end]`. Your result should be a list of all overlapping intervals between the two lists.

[LeetCode 986 – Interval List Intersections]

Example:

Input:

```
A = [[0,2],[5,10],[13,23],[24,25]]
B = [[1,5],[8,12],[15,24],[25,26]]
```

Output:

```
[[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]
```

How does this work?

- `[0,2]` from A overlaps with `[1,5]` from B at `[1,2]`
- `[5,10]` from A overlaps with `[1,5]` at `[5,5]`, and with `[8,12]` at `[8,10]`
- And so on.

Thought Process:

If you try to compare every interval in `A` to every interval in `B`, it works but is inefficient (think: O(NM)).

But since both lists are sorted and non-overlapping, you can walk through both at the same time using two pointers (like merging two sorted lists), always moving forward.

Try this input by hand:

```
A = [[3,6],[8,10]]
B = [[1,2],[4,5],[9,12]]
```

What do you get?

Brute Force Approach:

- For each interval `a` in A:
    - For each interval `b` in B:
        - If `a` and `b` overlap, add their intersection to the result.

Time Complexity: O(NM)

Optimal Approach: Two Pointer Sweep

- Use two pointers, one for each list.
- At each step, check if the current intervals overlap.
    - If so, add the intersection.
- Move the pointer that points to the interval that ends first.

Why does this work?

Because intervals are sorted and non-overlapping within each list, you never miss an intersection or process one twice.

Python Solution

```python
def intervalIntersection(A, B):
    result = []
    i, j = 0, 0
    while i < len(A) and j < len(B):
        a_start, a_end = A[i]
        b_start, b_end = B[j]

        # Find overlap, if any
        start = max(a_start, b_start)
        end = min(a_end, b_end)
        if start <= end:
            result.append([start, end])

        # Move the pointer that ends first
        if a_end < b_end:
            i += 1
        else:
            j += 1
    return result
```

Time Complexity: O(N + M), N = len(A), M = len(B)

Space Complexity: O(K), K = number of intersections

Explanation

- `i`, `j`: pointers for A and B.
- At each step, find the max of the start times and the min of the end times.
- If `start <= end`, intervals overlap, so add `[start, end]`.
- Move the pointer with the smaller end—because after that, the interval can't overlap with anything further.

Trace Example

```
A = [[0,2],[5,10],[13,23],[24,25]]
B = [[1,5],[8,12],[15,24],[25,26]]
```

- Compare `[0,2]` and `[1,5]` -> overlap at `[1,2]`, advance A (ends first)
- `[5,10]` and `[1,5]` -> overlap at `[5,5]`, advance B
- `[5,10]` and `[8,12]` -> overlap at `[8,10]`, advance A

Continue this way!

Try this test case:

```
A = [[2,4],[7,9]]
B = [[1,3],[5,8]]
```

(What do you get?)

Encouragement:
Pause here and try implementing this on your own or dry-running with a test case before moving on!

Problem 2: Employee Free Time

Problem Statement (Rephrased):

Given a list of schedules, one for each employee, where each schedule is a list of non-overlapping intervals representing when that employee is busy, find the common *free* intervals when all employees are available.

[LeetCode 759 – Employee Free Time]

Example:

Input:

```
[[[1,2],[5,6]], [[1,3]], [[4,10]]]
```

Output:

```
[[3,4]]
```

Why?

- All are busy at `[1,2]`, `[1,3]`, `[4,10]`, `[5,6]`.
- The only gap when everyone is free is `[3,4]`.

How is this similar to Problem 1?

Both require reasoning about overlapping intervals. Previously we found intersections; now we want *gaps*—the periods between combined busy intervals.

Brute Force:

- Generate the union of all busy intervals.
- At every time, check if all employees are busy or not (expensive).

Optimal Approach: Merge & Find Gaps

- Collect all busy intervals from all employees.
- Sort them by start time.
- Merge overlapping intervals (classic merge intervals).
- The gaps between merged intervals are the free times.

Step-by-Step:

- Flatten all schedules into a single list.
- Sort by start time.

- Merge overlapping intervals:
    - If the next interval starts after the current ends, that's a free slot.
    - Else, extend the current interval.

Pseudocode:

```
intervals = flatten all schedules
sort intervals by start time

merged = []
for interval in intervals:
    if merged is empty or merged[-1].end < interval.start:
        merged.append(interval)
    else:
        merged[-1].end = max(merged[-1].end, interval.end)

free_times = []
for i in 1 to len(merged) - 1:
    free_times.append([merged[i-1].end, merged[i].start])
return free_times
```

Example Trace:

Input:

```
[[[1,3],[6,7]], [[2,4]], [[2,5],[9,12]]]
```

Flattened: `[1,3]`, `[6,7]`, `[2,4]`, `[2,5]`, `[9,12]`

Sorted: `[1,3]`, `[2,4]`, `[2,5]`, `[6,7]`, `[9,12]`

Merge:

- Merge `[1,3]`, `[2,4]`, `[2,5]` -> `[1,5]`
- Next, `[6,7]` is after `[1,5]` (gap `[5,6]`)
- Merge `[6,7]` -> `[6,7]`
- `[9,12]` after `[6,7]` (gap `[7,9]`)

Free times: `[5,6]`, `[7,9]`

Test Case:

Input:

```
[[[1,3],[5,6]], [[2,4]], [[6,8]]]
```

What are the free times?

Time Complexity: O(N log N) (N = total intervals) for sorting and merging.

Space Complexity: O(N) for storing intervals.

Problem 3: My Calendar I

Problem Statement (Rephrased):

Design a calendar where you can book a new interval `[start, end)`, but only if it doesn't overlap with any existing booked interval. Return `True` if booking is successful, `False` otherwise.

[LeetCode 729 – My Calendar I]

How is this different?

- Now, instead of finding overlaps, you must *prevent* them.
- You must efficiently check for conflicts on every booking.

Brute Force:

- For each new booking, check every existing booking for overlap.
- For small numbers of bookings, this is fine.

Optimal Approach: Binary Search Insert

If bookings are stored in a sorted list, you can use binary search to find where the new interval should go, and check only neighboring intervals for conflicts.

Pseudocode:

```
class MyCalendar:
    intervals = []

    book(start, end):
        for each (s, e) in intervals:
            if start < e and end > s:  # overlap condition
                return False
        intervals.append([start, end])
        sort intervals by start
        return True
```

Optionally, use `bisect` module for binary search insertion for better efficiency.

Example Trace:

- Book `[10, 20)` → intervals = `[[10, 20]]` → returns True
- Book `[15, 25)` → overlaps with `[10, 20]` (since 15 < 20 and 25 > 10) → returns False
- Book `[20, 30)` → no overlap → add and return True

Test Case:

- Book `[5,10)`, `[10,15)`, `[10,12)`. What happens?

Time Complexity: O(N) per booking (since we may check all intervals). With binary search, O(log N) for search, but still O(N) for insertion.

Space Complexity: O(N) for booked intervals.

Encouragement:
Try implementing the above pseudocode, and see what happens for booking `[10, 20)`, `[20, 30)`, `[15, 25)` in different orders. Can you spot any edge cases?

Summary and Next Steps

Today, you saw three classic interval problems—finding intersections, merging busy times to find free slots, and preventing overlaps when booking. You should now have a solid understanding of:

- How to walk through two sorted interval lists using two pointers.
- How to merge intervals and find gaps.
- How to check for and prevent scheduling conflicts.

Key Patterns & Insights:

- For sorted, non-overlapping interval lists, two-pointer sweeps are powerful.
- Merging intervals helps expose gaps (free time).
- Checking for conflicts is all about overlap logic: `start < existing_end and end > existing_start`.

Common Mistakes:

- Forgetting to advance the correct pointer when intervals end at the same time.
- Overlooking edge cases at interval boundaries (e.g., end exclusive vs. inclusive).
- Not sorting intervals before merging or checking.

Action List

- **Solve all three problems by hand and in code.** Even if code is provided, try to reimplement from scratch.
- **Experiment with different approaches**, like using heaps or binary search trees for calendar bookings.
- **Come up with your own edge-case test cases**, such as intervals that touch but do not overlap.
- **Compare your solutions** with the LeetCode discussions or other reference solutions.
- **Reflect on the patterns**—where else might you see interval merging or intersection in interviews?

Remember: Every tough interval problem you master makes you a much stronger coder. Keep practicing, and happy coding!