Topic Introduction

Today, we're diving into the world of **Interval Scheduling**! If you've ever tried to book back-to-back meetings or watched balloons pop in a chain reaction, you've brushed up against this concept.

What is Interval Scheduling?
Interval scheduling is about working with "intervals" — ranges defined by a start and end. The core challenge is to efficiently answer questions like:

- Do any intervals overlap?
- What's the minimum number of resources (like meeting rooms) needed to handle all intervals?
- What's the best way to group or process intervals to minimize work?

How does it work?
The most common techniques are sorting intervals by their start or end times, then processing them in order. Sometimes we use a **gr eedy approach** (always making the best local choice), or a **heap/priority queue** (to efficiently track what's ending soonest).

Why is it useful in interviews?
Interval problems test your ability to:

- Recognize patterns and relationships between data.
- Use sorting and efficient data structures.
- Design greedy or heap-based solutions.

These problems are practical and appear frequently in both real-world scheduling and interviews.

Quick Example (not one of today's problems):
Suppose you have intervals: [1, 3], [2, 4], [5, 7].

- Are any overlapping?
  - Yes: [1, 3] and [2, 4] overlap.
- How many non-overlapping intervals can you select?
  - Two: [1, 3] and [5, 7], or [2, 4] and [5, 7].

Today’s lineup:

- **Meeting Rooms:** Check if any meeting times overlap.
- **Meeting Rooms II:** What's the minimum number of rooms needed so all meetings can happen?
- **Minimum Number of Arrows to Burst Balloons:** What's the least number of arrows to burst all overlapping balloons?

What connects these?
All three are **interval scheduling problems**. They challenge you to recognize overlaps, efficiently allocate resources, or minimize actions based on intervals. The core trick is always: "How can I process these intervals to make the fewest mistakes or use the fewest resources?"

Let's get cracking!

Problem 1: Meeting Rooms

Meeting Rooms - LeetCode

In your own words:

Given a list of meeting intervals (each with a start and end time), determine if a person can attend all meetings. In other words, do any meetings overlap?

Example:

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

Output: `False`

Why? The meetings [0,30] and [5,10] overlap.

Thought Process:

- If any two meetings overlap, you can't attend both.
- Overlap exists if the start of one meeting is before the end of the previous meeting.

Try this case on your own:

Input: `[[7,10],[2,4]]`

What should the output be?

Brute-Force Approach:

- Compare every pair of meetings and check if they overlap.
- Time Complexity: O(n^2) for n meetings.

Optimal Approach:

- Sort the intervals by start time.
- Then, for each consecutive pair, check if the next meeting starts before the previous one ends.
- If yes, return False. If you get through all, return True.

Pattern:

This uses the **"Sort and Sweep"** pattern — process intervals in order and look for overlaps.

Python Solution:

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

    # Check for overlaps
    for i in range(1, len(intervals)):
        prev_end = intervals[i-1][1]
        curr_start = intervals[i][0]
        # If the current meeting starts before the previous one ends, overlap!
        if curr_start < prev_end:
            return False
    return True
```

Time Complexity: O(n log n) for sorting, O(n) to check overlaps.

Space Complexity: O(1) (ignoring input sort, which might require O(n) extra space depending on language).

Explanation:

- First, we sort all meetings by their start time so they're in order.
- Next, we walk through each pair of meetings.
- If any meeting starts before the previous one ends, we return False.
- If we get through all pairs, there are no overlaps; return True.

Trace Example:

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

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

- Compare [0,30] and [5,10]: 5 < 30 → overlap! Return False.

Try this test case on your own:

Input: `[[1,4],[4,5],[5,6]]`

What should the output be?

Your Turn:

Take a moment to solve this on your own before moving to the next problem!

Problem 2: Meeting Rooms II

Meeting Rooms II - LeetCode

Here's where things get interesting. Instead of just checking for overlaps, now you want to **find the minimum number of meeting rooms required** to hold all meetings without conflicts.

How is this different?

- Problem 1 was about any overlap.
- Here, you need to handle all overlaps — sometimes several meetings overlap at once, so you need multiple rooms.

Example:

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

Output: `2`

- [0,30] and [5,10] overlap, need two rooms. [15,20] overlaps with [0,30] but not [5,10].

Brute-Force Approach:

- For every meeting, check how many other meetings overlap with it.
- Take the maximum overlap count.
- Time Complexity: O(n^2)

Optimal Approach:

We need to know, at any time, how many meetings are happening simultaneously.

Step-by-step logic:

- Separate start and end times into two lists.
- Sort both lists.
- Use two pointers: one for start times, one for end times.
- Traverse start times:
    - If the next meeting starts after the earliest one ends, reuse a room (move end pointer).
    - Otherwise, need a new room.

- Track the max number of rooms used.

Pseudocode:

```
starts = sorted(list of all start times)
ends = sorted(list of all end times)
rooms = 0
end_ptr = 0

for start in starts:
    if start < ends[end_ptr]:
        rooms += 1  # need a new room
    else:
        end_ptr += 1  # room freed up
return rooms
```

Trace Example:

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

Starts: [0,5,15]; Ends: [10,20,30]

- start=0 < end=10: rooms=1
- start=5 < end=10: rooms=2
- start=15 >= end=10: end_ptr+=1 (end_ptr=1), rooms stays at 2, continue
- start=15 < end=20: rooms=2

So, answer is 2.

Try this case yourself:

Input: `[[7,10],[2,4],[3,6]]`

How many rooms are needed?

Time Complexity: O(n log n) for sorting.

Space Complexity: O(n) for separate start/end lists.

Problem 3: Minimum Number of Arrows to Burst Balloons

Minimum Number of Arrows to Burst Balloons - LeetCode

Now, let's twist things!

Given a set of balloons in the air, each represented by a horizontal interval `[x_start, x_end]`. An arrow can burst all balloons it passes through. Find the minimum number of arrows needed to burst all balloons.

What's different here?

You're grouping **overlapping intervals** together so one action (an arrow) can "cover" them all. You want to minimize the number of groups.

This is a **greedy interval covering** problem.

Example:

Input: `[[10,16],[2,8],[1,6],[7,12]]`

Output: `2`

- One arrow at x=6 can burst [1,6], [2,8], and [7,12] overlaps with [10,16] only at 12-16, so needs another arrow.

Brute-Force Approach:

- Try every possible way to group overlapping balloons.
- Exponential complexity (not feasible).

Optimal Approach:

- Sort balloons by their end coordinate.
- Shoot an arrow at the end of the first balloon.
- For each balloon, if its start is after the position of the last arrow, need a new arrow.
- Otherwise, it's already burst.

Pseudocode:

```
sort intervals by end
arrows = 0
arrow_pos = -infinity

for balloon in intervals:
    if balloon.start > arrow_pos:
        arrows += 1
        arrow_pos = balloon.end
return arrows
```

Example Trace:

Input: `[[1,6],[2,8],[7,12],[10,16]]`

Sorted by end: [1,6], [2,8], [7,12], [10,16]

- arrow at 6 (covers first two)
- [7,12] starts after 6? Yes: new arrow at 12
- [10,16] starts after 12? No: already covered

Answer: 2

Try this case:

Input: `[[1,2],[2,3],[3,4],[4,5]]`

How many arrows are needed?

Time Complexity: O(n log n) for sorting.

Space Complexity: O(1) extra.

Summary and Next Steps

These three problems are all about **interval scheduling**:

- Recognizing and handling overlaps.

- Using sorting to bring order to chaos.
- Applying greedy or two-pointer strategies to solve efficiently.

Key Patterns:

- Always sort intervals, by start or end, for clarity.
- Use simple sweeps (for overlap checks), two pointers (for simultaneous intervals), or greedy selection (for minimum actions).
- For resource allocation, track "active intervals" with pointers or priority queues.

Common Traps:

- Not sorting correctly.
- Off-by-one errors (does [x, y] mean open or closed interval?).
- Forgetting to check the last interval.

Action List

- Solve all three problems on your own, even the one with code given.
- Try Problem 2 and 3 with a different approach (for Problem 2, try using a min-heap).
- Explore similar interval problems (like "Employee Free Time" or "Merge Intervals").
- Compare your solutions to others for edge cases and style.
- If you get stuck, take a breath and come back — practice is the best teacher!

Happy coding! Keep chasing those intervals — you'll have them all lined up in no time.