

Intervals & Greedy

1. Insert Interval

Pattern: Intervals & Greedy

Problem Statement

You are given an array of non-overlapping intervals `intervals` where `intervals[i] = [start, end]` represent the start and the end of the `i` interval and `intervals` is sorted in ascending order by `start`. You are also given an interval `newInterval = [start, end]` that represents the start and end of another interval.

Insert `newInterval` into `intervals` such that `intervals` is still sorted in ascending order by `start` and `intervals` still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return `intervals` after the insertion.

Sample Input & Output

```
Input: intervals = [[1,3],[6,9]], newInterval = [2,5]
Output: [[1,5],[6,9]]
Explanation: [2,5] overlaps with [1,3] → merge to [1,5]
```

```
Input: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]],
       newInterval = [4,8]
Output: [[1,2],[3,10],[12,16]]
Explanation: [4,8] overlaps with [3,5], [6,7], [8,10] → merge all
```

```
Input: intervals = [], newInterval = [5,7]
Output: [[5,7]]
Explanation: Empty input → just insert new interval
```

LeetCode Editorial Solution + Inline Tests

```
from typing import List

class Solution:
    def insert(self, intervals: List[List[int]],
               newInterval: List[int]) -> List[List[int]]:
        # STEP 1: Initialize result list and unpack new interval
        # - We'll build result incrementally
        # - Extract start/end for readability
        result = []
        new_start, new_end = newInterval

        # STEP 2: Main loop - process each existing interval
        # - Maintain invariant: result has non-overlapping,
        #   sorted intervals up to current point
        for interval in intervals:
            curr_start, curr_end = interval

            # Case 1: Current interval ends before new starts
            # → No overlap; add current to result
            if curr_end < new_start:
                result.append(interval)

            # Case 2: Current interval starts after new ends
            # → No overlap; but newInterval not yet added
            elif curr_start > new_end:
                result.append([new_start, new_end])
```

```

        result.append(interval)
        new_start = float('inf') # mark as inserted

    # Case 3: Overlap exists → merge intervals
    # - Expand newInterval to cover both
    else:
        new_start = min(new_start, curr_start)
        new_end = max(new_end, curr_end)

    # STEP 3: Handle case where newInterval never inserted
    # - Happens if it extends beyond all intervals
    if new_start != float('inf'):
        result.append([new_start, new_end])

    # STEP 4: Return merged, sorted, non-overlapping list
    return result

# ----- INLINE TESTS -----
if __name__ == "__main__":
    sol = Solution()

    # Test 1: Normal case - overlap in middle
    intervals1 = [[1,3],[6,9]]
    new1 = [2,5]
    output1 = sol.insert(intervals1, new1)
    expected1 = [[1,5],[6,9]]
    assert output1 == expected1, f"Test 1 failed: {output1}"
    print(" Test 1 passed:", output1)

    # Test 2: Edge case - empty intervals
    intervals2 = []
    new2 = [5,7]
    output2 = sol.insert(intervals2, new2)
    expected2 = [[5,7]]
    assert output2 == expected2, f"Test 2 failed: {output2}"
    print(" Test 2 passed:", output2)

    # Test 3: Tricky case - newInterval spans multiple
    intervals3 = [[1,2],[3,5],[6,7],[8,10],[12,16]]
    new3 = [4,8]
    output3 = sol.insert(intervals3, new3)
    expected3 = [[1,2],[3,10],[12,16]]

```

```
assert output3 == expected3, f"Test 3 failed: {output3}"
print(" Test 3 passed:", output3)
```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 3**:

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

Goal: Insert [4,8] and merge overlapping intervals.

Initial Setup

- `result = []`
 - `new_start = 4, new_end = 8`
-

Step 1: Process [1,2]

- `curr_start = 1, curr_end = 2`
 - Check: $2 < 4 \rightarrow \text{True} \rightarrow \text{Case 1}$
 - **Action:** Append [1,2] to `result`
 - **State:**
 - `result = [[1,2]]`
 - `new_start = 4, new_end = 8`
-

Step 2: Process [3,5]

- `curr_start = 3, curr_end = 5`
 - Check: $5 < 4?$ \rightarrow No
 - Check: $3 > 8?$ \rightarrow No
 - \rightarrow **Overlap!** \rightarrow Case 3
 - **Merge:**
 - `new_start = min(4, 3) = 3`
 - `new_end = max(8, 5) = 8`
 - **State:**
 - `result = [[1,2]]`
 - `new_start = 3, new_end = 8`
-

Step 3: Process [6,7]

- `curr_start = 6, curr_end = 7`
- Check: $7 < 3?$ \rightarrow No
- Check: $6 > 8?$ \rightarrow No
- \rightarrow **Overlap!** \rightarrow Case 3
- **Merge:**
 - `new_start = min(3, 6) = 3`
 - `new_end = max(8, 7) = 8`
- **State:**

- `result = [[1,2]]`
- `new_start = 3, new_end = 8`

Step 4: Process [8,10]

- `curr_start = 8, curr_end = 10`
 - Check: $10 < 3?$ → No
 - Check: $8 > 8?$ → **No** ($8 == 8$ → not greater)
 - → **Overlap!** (touching counts as overlap) → Case 3
 - **Merge:**
 - `new_start = min(3, 8) = 3`
 - `new_end = max(8, 10) = 10`
 - **State:**
 - `result = [[1,2]]`
 - `new_start = 3, new_end = 10`
-

Step 5: Process [12,16]

- `curr_start = 12, curr_end = 16`
- Check: $16 < 3?$ → No
- Check: $12 > 10?$ → **Yes** → Case 2
- **Action:**
 - Append merged [3,10] to **result**

- Append [12,16]
- Set `new_start = inf` (mark as inserted)

- **State:**

- `result = [[1,2], [3,10], [12,16]]`
 - `new_start = inf`
-

Final Check

- `new_start == inf` → skip final append
-

Final Output

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

Key Insight:

We never insert `newInterval` immediately. Instead, we **delay insertion** until we find the first interval that starts *after* the merged interval ends — or until the end. This greedy approach ensures we merge all overlapping intervals in one pass.

Complexity Analysis

- **Time Complexity:** $O(n)$

We iterate through `intervals` exactly once. Each interval is processed in constant time (comparisons and min/max). No nested loops.

- **Space Complexity:** $O(1)$ auxiliary (not counting output)

We only use a few scalar variables (`new_start`, `new_end`). The `result` list is required for output, so it's not counted as extra space per LeetCode conventions.

2. Merge Intervals

Pattern: Intervals & Greedy

Problem Statement

Given an array of intervals where `intervals[i] = [starti, endi]`, merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input.*

You may assume that the input is not necessarily sorted.

Sample Input & Output

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

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

```
Explanation: Intervals [1,3] and [2,6] overlap → merge to [1,6].
```

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

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

```
Explanation: [1,4] and [4,5] are adjacent; LeetCode treats them as overlapping.
```

```
Input: intervals = [[1,4],[0,0]]
```

```
Output: [[0,0],[1,4]]
```

```
Explanation: Non-overlapping and unsorted input → must sort first.
```

LeetCode Editorial Solution + Inline Tests


```

from typing import List

class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        # STEP 1: Initialize structures
        # - Sort by start time to enable greedy merging
        # - Use 'merged' list to build result incrementally
        if not intervals:
            return []

        intervals.sort(key=lambda x: x[0])
        merged = [intervals[0]]

        # STEP 2: Main loop / recursion
        # - Iterate from second interval onward
        # - Compare current interval with last in 'merged'
        for current in intervals[1:]:
            last = merged[-1]

            # STEP 3: Update state / bookkeeping
            # - If current start <= last end → overlap exists
            # - Extend last interval's end if needed
            if current[0] <= last[1]:
                # Merge by updating end of last interval
                merged[-1][1] = max(last[1], current[1])
            else:
                # No overlap → add current as new interval
                merged.append(current)

        # STEP 4: Return result
        # - 'merged' already handles empty input via early return
        return merged

# ----- INLINE TESTS -----
if __name__ == "__main__":
    sol = Solution()

    # Test 1: Normal case
    assert sol.merge([[1,3],[2,6],[8,10],[15,18]]) == \
        [[1,6],[8,10],[15,18]]

    # Test 2: Edge case - adjacent intervals

```

```
assert sol.merge([[1,4],[4,5]]) == [[1,5]]

# Test 3: Tricky/negative - unsorted, disjoint
assert sol.merge([[1,4],[0,0]]) == [[0,0],[1,4]]

print(" All tests passed!")
```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1:** `intervals = [[1,3],[2,6],[8,10],[15,18]]`.

Step 0: Initial Setup

- Input: `[[1,3],[2,6],[8,10],[15,18]]`
- Check if `not intervals` → `false` (list not empty).
- **Sort intervals by start:** already sorted → no change.
- Initialize `merged = [[1,3]]`

State: `merged = [[1,3]]`

Step 1: Process [2,6]

- `current = [2,6]`
- `last = merged[-1] = [1,3]`
- Check: `current[0] = 2 <= last[1] = 3` → **True** (overlap!)
- Update `merged[-1][1] = max(3, 6) = 6`
- Now `merged = [[1,6]]`

State: `merged = [[1,6]]`

Step 2: Process [8,10]

- `current = [8,10]`
- `last = [1,6]`
- Check: $8 \leq 6 \rightarrow \text{False}$ (no overlap)
- Append `[8,10]` to `merged`
- Now `merged = [[1,6], [8,10]]`

State: `merged = [[1,6], [8,10]]`

Step 3: Process [15,18]

- `current = [15,18]`
- `last = [8,10]`
- Check: $15 \leq 10 \rightarrow \text{False}$
- Append `[15,18]`
- Now `merged = [[1,6], [8,10], [15,18]]`

State: `merged = [[1,6], [8,10], [15,18]]`

Step 4: Return Result

- Return `[[1,6], [8,10], [15,18]]` \rightarrow matches expected output.

Key Insight:

By **sorting first**, we guarantee that any overlap must involve the **most recent merged interval**. This enables a **greedy** one-pass merge — no need to check all previous intervals!

Complexity Analysis

- **Time Complexity:** $O(n \log n)$

Dominated by sorting ($O(n \log n)$). The single loop is $O(n)$, and each merge operation is $O(1)$. Total: $O(n \log n + n) = O(n \log n)$.

- **Space Complexity:** $O(n)$

The merged list stores up to n intervals in the worst case (no overlaps). Sorting may use $O(\log n)$ stack space (Timsort), but output space is $O(n)$, which dominates.

3. Non-overlapping Intervals

Pattern: Intervals & Greedy

Problem Statement

Given an array of intervals `intervals` where `intervals[i] = [starti, endi]`, return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping*.

Note:

- Intervals `[1,2]` and `[2,3]` are considered non-overlapping.
- The problem is equivalent to: *Find the maximum number of non-overlapping intervals*, then subtract from total.

Sample Input & Output

```
Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
Output: 1
Explanation: [1,3] overlaps with [1,2] and [2,3].
Removing it leaves 3 non-overlapping intervals.
```

Input: intervals = [[1,2],[1,2],[1,2]]
Output: 2
Explanation: All intervals overlap. Keep only one → remove 2.

Input: intervals = [[1,2]]
Output: 0
Explanation: Single interval → no overlaps possible.

LeetCode Editorial Solution + Inline Tests

```
from typing import List

class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        # STEP 1: Initialize structures
        # - Sort by end time to enable greedy selection
        # - Greedy choice: pick interval that ends earliest
        if not intervals:
            return 0

        intervals.sort(key=lambda x: x[1])

        # STEP 2: Main loop / recursion
        # - Maintain last_end: end of last kept interval
        # - Count kept intervals (non-overlapping)
        last_end = intervals[0][1]
        kept = 1

        # STEP 3: Update state / bookkeeping
        # - If current start >= last_end → no overlap
        # - Update last_end and increment kept
        for i in range(1, len(intervals)):
            start, end = intervals[i]
            if start >= last_end:
                kept += 1
                last_end = end
```

```

        # STEP 4: Return result
        #   - Total intervals minus max non-overlapping = removals
        return len(intervals) - kept

# ----- INLINE TESTS -----
if __name__ == "__main__":
    sol = Solution()

    # Test 1: Normal case
    assert sol.eraseOverlapIntervals([[1,2],[2,3],[3,4],[1,3]]) == 1

    # Test 2: Edge case
    assert sol.eraseOverlapIntervals([[1,2]]) == 0

    # Test 3: Tricky/negative
    assert sol.eraseOverlapIntervals([[1,2],[1,2],[1,2]]) == 2

    print(" All tests passed!")

```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**: `intervals = [[1,2],[2,3],[3,4],[1,3]]`.

Step 0: Initial Setup

- Input: `[[1,2],[2,3],[3,4],[1,3]]`
- Check if not intervals → false → skip return.

Step 1: Sort by end time

- Original: `[[1,2],[2,3],[3,4],[1,3]]`
- End times: 2, 3, 4, 3

- After sorting by $x[1]$:
 $[[1,2], [2,3], [1,3], [3,4]]$
 \rightarrow Why? Because ends are $[2, 3, 3, 4] \rightarrow$ stable sort keeps $[2,3]$ before $[1,3]$.

Step 2: Initialize tracking variables

- $\text{last_end} = \text{intervals}[0][1] = 2$
- $\text{kept} = 1$ (we keep the first interval $[1,2]$)

Step 3: Loop through remaining intervals

Iteration 1 ($i=1$):

- $\text{intervals}[1] = [2,3] \rightarrow \text{start}=2, \text{end}=3$
- Check: $\text{start} (2) \geq \text{last_end} (2) \rightarrow \mathbf{True}$
- Action:
- $\text{kept} += 1 \rightarrow \text{kept} = 2$
- $\text{last_end} = \text{end} = 3$
- Now: kept intervals = $[[1,2], [2,3]]$

Iteration 2 ($i=2$):

- $\text{intervals}[2] = [1,3] \rightarrow \text{start}=1, \text{end}=3$
- Check: $1 \geq 3 \rightarrow \mathbf{False} \rightarrow \text{skip}$
- No change to kept or last_end

Iteration 3 ($i=3$):

- $\text{intervals}[3] = [3,4] \rightarrow \text{start}=3, \text{end}=4$
- Check: $3 \geq 3 \rightarrow \mathbf{True}$
- Action:
- $\text{kept} += 1 \rightarrow \text{kept} = 3$
- $\text{last_end} = 4$
- Now: kept intervals = $[[1,2], [2,3], [3,4]]$

Step 4: Compute result

- Total intervals = 4
- Kept = 3
- Return $4 - 3 = 1$

Final output: 1 \rightarrow matches expected.

Key Insight:

By sorting by **end time**, we always pick the interval that *finishes earliest*, leaving maximum room for future intervals — classic **greedy interval scheduling**.

Complexity Analysis

- **Time Complexity:** $O(n \log n)$

Dominated by sorting ($O(n \log n)$). The loop is $O(n)$, so total is $O(n \log n)$.

- **Space Complexity:** $O(1)$

Sorting is in-place (Python's Timsort uses $O(n)$ worst-case, but we consider auxiliary space).

We only use a few extra variables (`last_end`, `kept`) \rightarrow constant extra space.

4. Meeting Rooms

Pattern: Intervals & Greedy

Problem Statement

Given an array of meeting time intervals where `intervals[i] = [start_i, end_i]`, determine if a person could attend all meetings.

Constraints: - $0 \leq \text{intervals.length} \leq 10^4$ - $\text{intervals}[i].\text{length} == 2$ - $0 \leq \text{start}_i < \text{end}_i \leq 10^6$

Clarification: A person **cannot** attend two meetings that overlap, even if one ends exactly when another starts (e.g., `[0,5]` and `[5,10]` are **non-overlapping** and allowed).

Sample Input & Output

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

Output: `False`

Explanation: Meetings `[0,30]` and `[5,10]` overlap → conflict.

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

Output: `True`

Explanation: Meetings `[2,4]` and `[7,10]` do not overlap → can attend both.

Input: `[]`

Output: `True`

Explanation: No meetings → trivially can attend all (zero) meetings.

LeetCode Editorial Solution + Inline Tests

```
from typing import List

class Solution:
    def canAttendMeetings(
        self, intervals: List[List[int]]
    ) -> bool:
        # STEP 1: Initialize structures
        #   - Sort by start time to process in chronological order.
        #   - Greedy choice: earliest-ending meetings first minimizes
        #     future conflicts (classic interval scheduling).
        if not intervals:
            return True

        intervals.sort(key=lambda x: x[0])

        # STEP 2: Main loop / recursion
        #   - Compare current meeting's start with previous end.
        #   - Invariant: all prior meetings are non-overlapping.
        for i in range(1, len(intervals)):
            prev_end = intervals[i - 1][1]
```

```

        curr_start = intervals[i][0]

        # STEP 3: Update state / bookkeeping
        # - If current starts before previous ends → overlap.
        # - Immediate return avoids unnecessary checks.
        if curr_start < prev_end:
            return False

    # STEP 4: Return result
    # - Default: no overlaps found → can attend all.
    return True

# ----- INLINE TESTS -----
if __name__ == "__main__":
    sol = Solution()

    # Test 1: Normal case
    assert sol.canAttendMeetings([[0,30],[5,10],[15,20]]) \
        == False

    # Test 2: Edge case
    assert sol.canAttendMeetings([]) == True

    # Test 3: Tricky/negative
    assert sol.canAttendMeetings([[9,10],[4,9],[4,17]]) \
        == False

```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 3**: `[[9,10],[4,9],[4,17]]`

Step 1: Check if intervals is empty

- `intervals = [[9,10],[4,9],[4,17]]` → not empty → skip return.

Step 2: Sort intervals by start time

- Before sort: `[[9,10],[4,9],[4,17]]`

- After sort: `[[4,9], [4,17], [9,10]]`
→ Now processed in chronological order.

Step 3: Enter loop ($i = 1$)

- $i = 1$
- `prev_end = intervals[0][1] = 9`
- `curr_start = intervals[1][0] = 4`
- Check: $4 < 9 \rightarrow \text{True} \rightarrow \text{return False immediately}$

Why?

- Meeting `[4,17]` starts at 4, but previous meeting `[4,9]` ends at 9.
- Since $4 < 9$, they **overlap** (both occupy time 4–9).
- Person cannot attend both → answer is **False**.

Final Output: False

The algorithm stops early on first conflict — efficient and correct.

Complexity Analysis

- **Time Complexity:** $O(n \log n)$

Dominated by sorting ($O(n \log n)$). The loop is $O(n)$, but sorting is costlier.

- **Space Complexity:** $O(1)$ (or $O(n)$ in Python due to sort)

Sorting in Python uses Timsort, which may use $O(n)$ auxiliary space.
No additional data structures scale with input — only a few variables.

5. Meeting Rooms II

Pattern: Intervals & Greedy

Problem Statement

Given an array of meeting time intervals `intervals` where `intervals[i] = [start_i, end_i]`, return the minimum number of conference rooms required.

Sample Input & Output

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

Output: 2

Explanation: Meetings `[0,30]` overlaps with both `[5,10]` and `[15,20]`, so we need 2 rooms.

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

Output: 1

Explanation: No overlapping meetings → only 1 room needed.

Input: `[[9,10],[4,9],[4,17]]`

Output: 2

Explanation: `[4,9]` and `[4,17]` overlap at time 4-9, but `[9,10]` starts exactly when `[4,9]` ends → no conflict with `[4,9]`, but still overlaps with `[4,17]`.

LeetCode Editorial Solution + Inline Tests

```
from typing import List
import heapq

class Solution:
    def minMeetingRooms(self, intervals: List[List[int]]) -> int:
        # STEP 1: Initialize structures
        #   - Sort by start time to process meetings chronologically
        #   - Use min-heap to track end times of ongoing meetings
        if not intervals:
            return 0

        intervals.sort(key=lambda x: x[0])
        min_heap = [] # stores end times of active meetings

        # STEP 2: Main loop / recursion
        #   - For each meeting, check if earliest ending meeting
        #     has finished (end <= current start)
        for start, end in intervals:
```

```

        # If room is free (earliest end <= current start),
        # reuse it by popping from heap
        if min_heap and min_heap[0] <= start:
            heapq.heappop(min_heap)

        # STEP 3: Update state / bookkeeping
        #   - Always add current meeting's end time
        #   - Heap size = number of rooms in use
        heapq.heappush(min_heap, end)

    # STEP 4: Return result
    #   - Final heap size = max concurrent meetings = min rooms
    return len(min_heap)

# ----- INLINE TESTS -----
if __name__ == "__main__":
    sol = Solution()

    # Test 1: Normal case
    assert sol.minMeetingRooms([[0,30],[5,10],[15,20]]) == 2

    # Test 2: Edge case - no meetings
    assert sol.minMeetingRooms([]) == 0

    # Test 3: Tricky/negative - back-to-back & overlap
    assert sol.minMeetingRooms([[9,10],[4,9],[4,17]]) == 2

    print(" All tests passed!")

```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 3**: `intervals = [[9,10],[4,9],[4,17]]`

Step 0: Initial Setup

- Input: `[[9,10],[4,9],[4,17]]`
- After sorting by start time: `[[4,9],[4,17],[9,10]]`
- `min_heap = []` (empty at start)

Step 1: Process meeting [4, 9]

- start = 4, end = 9
- min_heap is empty → skip pop
- Push 9 into heap → min_heap = [9]
- **State:** 1 room in use

Step 2: Process meeting [4, 17]

- start = 4, end = 17
- Check: min_heap[0] = 9 → is 9 ≤ 4? **No** → cannot reuse room
- Push 17 into heap → min_heap = [9, 17] (heapified: [9, 17])
- **State:** 2 rooms in use

Step 3: Process meeting [9, 10]

- start = 9, end = 10
- Check: min_heap[0] = 9 → is 9 ≤ 9? **Yes!** → free that room
- Pop 9 from heap → min_heap = [17]
- Push 10 → min_heap = [10, 17] → heapified to [10, 17]
- **State:** still 2 rooms (one freed, one reused)

Final Step: Return Result

- len(min_heap) = 2 → return 2
- Matches expected output!

Key Insight:

The heap always tracks **currently occupied rooms** by their **end times**.

By reusing a room whenever the earliest-ending meeting is done, we **minimize total rooms** — a classic **greedy choice**.

Complexity Analysis

- **Time Complexity:** $O(n \log n)$

Sorting takes $O(n \log n)$. Each heap operation (push/pop) is $O(\log n)$, and we do up to n of them \rightarrow total $O(n \log n)$.

- **Space Complexity:** $O(n)$

In worst case (all meetings overlap), heap stores all n end times. Sorting may use $O(n)$ space in Python (Timsort).

6. Employee Free Time

Pattern: Intervals & Greedy

Problem Statement

We are given a list of schedules for employees, where each schedule is a list of non-overlapping intervals sorted in increasing order. Each interval has a **start** and **end** time.

Return a list of **finite** intervals representing common, positive-length free time for **all** employees, also in sorted order.

(Free time is when no employee is working.)

Sample Input & Output

```
Input: schedule = [[[1,2],[5,6]], [[1,3]], [[4,10]]]
Output: [[3,4]]
Explanation: Only time slot when all are free is [3,4].
```

```
Input: schedule = [[[1,3],[6,7]], [[2,4]], [[2,5],[9,12]]]
Output: [[5,6],[7,9]]
Explanation: Two common free intervals exist.
```

Input: schedule = [[[1,10]]]
Output: []
Explanation: Only one employee → no "common" free time.

LeetCode Editorial Solution + Inline Tests

```
from typing import List

# Definition for an Interval (as provided by LeetCode)
class Interval:
    def __init__(self, start: int = 0, end: int = 0):
        self.start = start
        self.end = end

class Solution:
    def employeeFreeTime(
        self, schedule: List[List[Interval]]
    ) -> List[Interval]:
        # STEP 1: Flatten all intervals into one list
        # - We lose employee identity but keep all busy times
        all_intervals = []
        for employee in schedule:
            for interval in employee:
                all_intervals.append(interval)

        # STEP 2: Sort intervals by start time (greedy choice)
        # - Ensures we can merge in one pass
        all_intervals.sort(key=lambda x: x.start)

        # STEP 3: Merge overlapping intervals
        # - Maintain merged list of *busy* time
        merged = []
        for interval in all_intervals:
            # If no overlap with last merged interval
            if not merged or merged[-1].end < interval.start:
                merged.append(interval)
            else:
                # Extend last interval to cover overlap
```



```

        merged[-1].end = max(merged[-1].end, interval.end)

# STEP 4: Find gaps between merged busy intervals
# - These gaps = common free time
free_time = []
for i in range(1, len(merged)):
    start_free = merged[i - 1].end
    end_free = merged[i].start
    if start_free < end_free: # positive length
        free_time.append(Interval(start_free, end_free))

return free_time

# ----- INLINE TESTS -----
if __name__ == "__main__":
    sol = Solution()

    # Test 1: Normal case
    emp1 = [Interval(1,2), Interval(5,6)]
    emp2 = [Interval(1,3)]
    emp3 = [Interval(4,10)]
    result1 = sol.employeeFreeTime([emp1, emp2, emp3])
    print("Test 1:", [(iv.start, iv.end) for iv in result1])
    # Expected: [(3, 4)]

    # Test 2: Edge case - only one employee
    emp = [Interval(1,10)]
    result2 = sol.employeeFreeTime([emp])
    print("Test 2:", [(iv.start, iv.end) for iv in result2])
    # Expected: []

    # Test 3: Tricky case - multiple free slots
    e1 = [Interval(1,3), Interval(6,7)]
    e2 = [Interval(2,4)]
    e3 = [Interval(2,5), Interval(9,12)]
    result3 = sol.employeeFreeTime([e1, e2, e3])
    print("Test 3:", [(iv.start, iv.end) for iv in result3])
    # Expected: [(5,6), (7,9)]

```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**:

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

Step 1: Flatten all intervals

- Loop through each employee and their intervals.
- `all_intervals` becomes:
[[1,2], [5,6], [1,3], [4,10]]
(as `Interval` objects)

Step 2: Sort by start time

- After sorting:
[[1,2], [1,3], [4,10], [5,6]]
→ Actually, [1,3] comes before [4,10], and [5,6] is last.

Step 3: Merge overlapping intervals

- Start with empty `merged = []`
- Process [1,2]: `merged` is empty → append → `merged = [[1,2]]`
- Process [1,3]:
 - Last in `merged` is [1,2], and $2 \geq 1$ → overlap!
 - Update end to $\max(2, 3) = 3$ → `merged = [[1,3]]`
- Process [4,10]:
 - $3 < 4$ → no overlap → append → `merged = [[1,3], [4,10]]`
- Process [5,6]:
 - Last is [4,10], and $10 \geq 5$ → overlap

– Update end to $\max(10, 6) = 10 \rightarrow \text{merged} = [[1,3], [4,10]]$

Final merged busy time: [[1,3], [4,10]]

Step 4: Find gaps between busy intervals

- Compare `merged[0].end = 3` and `merged[1].start = 4`
- Since $3 < 4$, free interval = `[3,4]` \rightarrow add to `free_time`
- No more intervals \rightarrow done.

Output: `[Interval(3,4)]` \rightarrow printed as `[(3, 4)]`

Complexity Analysis

- **Time Complexity:** $O(N \log N)$

Where N = total number of intervals across all employees.

Sorting dominates ($O(N \log N)$). Merging and gap-finding are $O(N)$.

- **Space Complexity:** $O(N)$

We store all intervals in `all_intervals` and `merged`.

Output list `free_time` also scales with gaps (N).