Non-overlapping Intervals Problem - Documentation

Table of Contents

1. Problem Statement

We are given an array of intervals, where each interval is represented as [start, end]. Our goal is to remove the minimum number of intervals such that the remaining intervals do not overlap.

Example 1

- Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
  Output: 1
  Explanation: Remove [1,3] to make the remaining intervals non-overlapping.

Example 2

- Input: intervals = [[1,2],[1,2],[1,2]]
  Output: 2
  Explanation: Remove two [1,2] intervals to make it non-overlapping.

Example 3

- Input: intervals = [[1,2],[2,3]]
  Output: 0
  Explanation: No need to remove any intervals since they are already non-overlapping.

Constraints:

- 1 <= intervals.length <= 10^5
- intervals[i].length == 2
- -50,000 <= start < end <= 50,000

2. Intuition

We should keep as many non-overlapping intervals as possible to minimize the number of removed intervals. The best way to do this is by selecting intervals that end the earliest, allowing more space for upcoming intervals.

3. Key Observations

- Intervals that touch at a point ([1,2] and [2,3]) are not overlapping.
- The best way to minimize removals is to always keep the interval with the smallest ending time and remove overlapping ones.
- Sorting the intervals by end time ensures that we always choose the optimal interval to keep.

4. Approach

- Sort the intervals based on their end times (intervals[i][1]).
- Use a greedy algorithm:
  - Keep track of the last selected interval (prev_end).
  - Iterate through intervals and check if they overlap with prev_end.
  - If overlapping, increment the removal count.

    - If not overlapping, update prev_end to the current interval's end.
- Return the number of removed intervals.

5. **Edge Cases**

- Single Interval ([[1,2]]) → No removals needed (Output: 0).
- All intervals are the same ([[1,2],[1,2],[1,2]]) → Remove len(intervals) - 1.
- Already non-overlapping ([[1,2],[2,3],[3,4]]) → No removals needed.
- Completely overlapping ([[1,10],[2,9],[3,8],[4,7]]) → Remove all except one.
- Large Input (10^5 intervals) → Solution must be O(n log n) to run efficiently.

6. **Complexity Analysis**

Time Complexity

- Sorting the intervals takes O(n log n).
- Iterating through intervals takes O(n).
- Overall complexity: O(n log n) (dominated by sorting).

Space Complexity

- O(1) (Only a few variables used).

7. **Alternative Approaches**

Brute Force ($O(n^2)$)

- Compare every interval with every other interval to check for overlaps.
- Remove the interval that causes the most overlap.
- Inefficient for large inputs (TLE for $n \approx 10^5$).

Dynamic Programming ($O(n^2)$)

- Find the longest subset of non-overlapping intervals (LIS-like problem).

- Inefficient for large inputs.

Greedy (O(n log n)) - Best Approach

- Sorting + greedy selection ensures optimal removals.
- Fastest and most efficient approach.

8. **Test Cases**

```
solution = Solution()

# Test Case 1
print(solution.eraseOverlapIntervals([[1,2],[2,3],[3,4],[1,3]]))  # Expected: 1

# Test Case 2
print(solution.eraseOverlapIntervals([[1,2],[1,2],[1,2]]))  # Expected: 2

# Test Case 3
print(solution.eraseOverlapIntervals([[1,2],[2,3]]))  # Expected: 0

# Test Case 4 (Already non-overlapping)
print(solution.eraseOverlapIntervals([[1,5],[6,10],[11,15]]))  # Expected: 0

# Test Case 5 (All overlapping)
print(solution.eraseOverlapIntervals([[1,10],[2,9],[3,8],[4,7]]))  # Expected: 3

# Test Case 6 (Large Input)
import random
large_intervals = [[random.randint(-50000, 50000), random.randint(-50000, 50000)] for _ in range(10**5)]
print(solution.eraseOverlapIntervals(large_intervals))  # Should run within time limits
```

9. Final Thoughts

- The greedy approach is optimal for this problem.
- Sorting + Iterating ensures a time complexity of $O(n \log n)$.
- The method is efficient for large datasets ($n \approx 10^5$).
- Edge cases, such as already non-overlapping or fully overlapping intervals are handled.
- Alternative approaches like brute force and dynamic programming are too slow.