October 8

Easy Greedy Algorithms Problems

Problem 1: Non-overlapping Intervals

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 that intervals that only touch at a point are considered non-overlapping. For example, [1, 2] and [2, 3] are non-overlapping.

Link to problem:

https://leetcode.com/problems/non-overlapping-intervals/

Example 1:

- **Input:** `intervals = [[1,2],[2,3],[3,4],[1,3]]`
- **Output:**`1`
  Explanation: Removing `[1,3]` results in non-overlapping intervals.

Example 2:

- **Input:** `intervals = [[1,2],[1,2],[1,2]]`
- **Output:**`2`
  Explanation: You need to remove two `[1,2]` to make the rest non-overlapping.

Example 3:

- **Input:** `intervals = [[1,2],[2,3]]`
- **Output:**`0`
  Explanation: No removal is needed as the intervals are already non-overlapping..

Solution:

```
class Solution {
    public int eraseOverlapIntervals(int[][] intervals) {
        // Sort the intervals by their end time
        Arrays.sort(intervals, (a, b) -> Integer.compare(a[1], b[1]));
        int count = 0; // Count of intervals to remove
        int end = intervals[0][1]; // End time of the last non-overlapping interval

        // Loop through intervals starting from the second one
        for (int i = 1; i < intervals.length; i++) {
```

```
            // If the current interval starts before the last one ends, it overlaps
            if (intervals[i][0] < end) {
                count++; // Increment the count for removal
            } else {
                end = intervals[i][1]; // Update the end time
            }
        }
        return count; // Return the total count of intervals to remove
    }
}
```

Explanation:

- The intervals are sorted based on their end times. This is crucial as it allows us to consider the earliest ending interval first, maximizing the chances of accommodating more intervals.
- A variable count keeps track of how many intervals need to be removed.
- We loop through the sorted intervals:
  - If the current interval starts before the last non-overlapping interval ends, it means there is an overlap, and we increment the count.
  - If there is no overlap, we update the end time to the current interval's end time.
- Finally, we return the count of intervals that need to be removed.

Time Complexity:

- O(n log n), where n is the number of intervals, due to the sorting step.

Space Complexity:

- O(1), as we are using a constant amount of extra space for the counters.
