Documentation in Find Right Interval

Table of Contents

1. Problem Statement

Given an array of intervals, where intervals[i] = [start_i, end_i] and each start_i is unique, we need to find the right interval for each interval.

A right interval for interval i is an interval j such that:

- start_j >= end_i
- start_j is minimized

If no right interval exists for interval i, return -1.

Examples

Example 1:

- Input: intervals = [[1,2]]
- Output: [-1]
- Explanation: Only one interval exists, so no right interval is found.

Example 2:

- Input: intervals = [[3,4], [2,3], [1,2]]
- Output:[-1, 0, 1]
- Explanation:
    - [3,4] has no right interval (-1).
    - [2,3] finds [3,4] (index 0).
    - [1,2] finds [2,3] (index 1).

Example 3:

- Input: intervals = [[1,4], [2,3], [3,4]]
- Output:[-1, 2, -1]
- Explanation:
    - [1,4] has no right interval (-1).
    - [2,3] finds [3,4] (index 2).
    - [3,4] has no right interval (-1).

2. Intuition

To efficiently find the smallest start_j >= end_i for each interval, we can:

- Sort the intervals by start
- Use binary search to quickly locate the smallest valid interval

Sorting allows us to leverage binary search for efficiency instead of scanning all intervals.

3. Key Observations

- Each interval's start is unique, so sorting won't cause duplicates.
- Sorting helps efficiently locate the right interval instead of a brute-force approach.
- Binary search (bisect_left) finds the smallest valid start_j efficiently.

4. Approach

Step 1: Sort Intervals by Start

- Store (start, index) pairs and sort by start value.
- Keep track of the original indices for later reference.

Step 2: Perform Binary Search for Each Interval

- For each interval [start, end], use bisect_left to find the smallest start value that is >= end.
- If such a value exists, store the corresponding index; otherwise, store -1.

Step 3: Return the Result

- Construct and return an array with the found indices.

5. Edge Cases

✅ Single interval (should return [-1])

✅ No valid right intervals exist (should return [-1] for those cases)

✅ All intervals are overlapping (should correctly identify minimal valid start)

✅ Start and end values at edge limits (-10^6 to 10^6, handled by sorting)

6. Complexity Analysis

Time Complexity

- Sorting intervals: O(nlogn)
- Binary search for each interval: O(log n) per interval, totaling O(nlogn)
- Overall Complexity: O(nlogn)

Space Complexity

- Storing the sorted list: O(n)
- Result array: O(n)
- Overall Complexity: O(n)

7. Alternative Approaches

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

- For each interval, iterate over all intervals to find the minimal start_j >= end_i.
- Drawback: Inefficient for large inputs.

Using a Heap (O(n log n))

- Use a min-heap to track available start values.
- Drawback: More complex implementation compared to binary search.

8. Test Cases

Basic Test Cases

```
solution = Solution()

assert solution.findRightInterval([[1,2]]) == [-1]
assert solution.findRightInterval([[3,4],[2,3],[1,2]]) == [-1, 0, 1]
assert solution.findRightInterval([[1,4],[2,3],[3,4]]) == [-1, 2, -1]
```

Edge Cases

```
assert solution.findRightInterval([[10,20],[30,40],[50,60]]) == [-1, -1, -1]  # No right intervals
assert solution.findRightInterval([[1,5],[2,6],[3,7]]) == [-1, -1, -1]  # Completely overlapping
assert solution.findRightInterval([[-1000000, -500000], [500000, 1000000]]) == [1, -1]  # Large negative and positive values
```

9. Final Thoughts

✅ Why This Works Well

- Sorting + Binary Search keeps it efficient.
- Uses minimal extra space beyond input storage.
- Easily handles edge cases like overlapping intervals.

📌 Key Takeaway: Sorting + Binary Search is optimal for finding minimal valid start_j >= end_i efficiently.