436. Find Right Interval

Medium Array Binary Search Sorting Leetcode Link

Problem Description

The goal of this problem is to find the "right interval" for a given set of intervals. Given an array `intervals`, where each element `intervals[i]` represents an interval with a `start_i` and an `end_i`, we need to identify for each interval `i` another interval `j` where the interval `j` starts at or after the end of interval `i`, and among all such possible `j`, it starts the earliest. This means that the start of interval `j` (`start_j`) must be greater than or equal to the end of interval `i` (`end_i`), and we want the smallest such `start_j`. If there is no such interval `j` that meets these criteria, the corresponding index should be `-1`.

The challenge is to write a function that returns an array of indices of the right intervals for each interval. If an interval has no right interval, as mentioned `-1` will be the placeholder for that interval.

Intuition

To approach this problem, we utilize a binary search strategy to efficiently find the right interval for each entry. Here's how we arrive at the solution:

1. To make it possible to return indices, we augment each interval with its original index in the `intervals` array.
2. We then sort the augmented intervals according to the starting points. The sort operation allows us to apply binary search later since binary search requires a sorted sequence.
3. Prepare an answer array filled with `-1` to assume initially that there is no right interval for each interval.
4. For each interval, use binary search (`bisect_left`) to efficiently find the least starting interval `j` that is greater than or equal to the end of the current interval `i`.
5. If the binary search finds such an interval, update the answer array at index `i` with the index of the found interval `j`.
6. After completing the search for all intervals, the answer array is returned.

Implementing binary search reduces the complexity of finding the right interval from a brute-force search which would be O(n^2) to O(n log n) since each binary search operation takes O(log n) and we perform it for each of the n intervals.

Solution Approach

The solution to this problem involves the following steps, which implement a binary search algorithm:

1. **Augment Intervals with Indices**: First, we update each interval to include its original index. This is achieved by iterating through the `intervals` array and appending the index to each interval.

```python
for i, v in enumerate(intervals):
    v.append(i)
```

2. **Sort Intervals by Start Times**: We then sort the augmented intervals based on their start times. This allows us to leverage binary search later on since it requires the list to be sorted.

```python
intervals.sort()
```

3. **Initialize Answer Array**: We prepare an answer array, `ans`, with the same length as the `intervals` array and initialize all its values to `-1`, which indicates that initially, we assume there is no right interval for any interval.

```python
ans = [-1] * n
```

4. **Binary Search for Right Intervals**: For each interval in `intervals`, we perform a binary search to find the minimum `start_j` value that is greater than or equal to `end_i`. To do this, we use the `bisect_left` method from the `bisect` module. It returns the index at which the `end_i` value could be inserted to maintain the sorted order.

```python
for _, e, i in intervals:
    j = bisect_left(intervals, [e])
```

5. **Updating the Answer**: If the binary search returns an index less than the number of intervals (`n`), it means we have found a right interval. We then update the `ans[i]` with the original index of the identified right interval, which is stored at `intervals[j][2]`.

```python
if j < n:
    ans[i] = intervals[j][2]
```

6. **Return the Final Array**: After the loop, the `ans` array is populated with the indices of right intervals for each interval in the given array. This array is then returned as the final answer.

```python
return ans
```

This approach efficiently uses binary search to minimize the time complexity. The key to binary search is the sorted nature of the `intervals` after they are augmented with their original indices. By maintaining a sorted list of start times and employing binary search, we're able to significantly reduce the number of comparisons needed to find the right interval from linear (checking each possibility one by one) to logarithmic, thus enhancing the performance of the solution.

Example Walkthrough

Let's walk through this approach with a small example. Consider the list of intervals `intervals = [[1,2], [3,4], [2,3], [4,5]]`.

1. First, we augment each interval with its index.

```python
augmented_intervals = [[1, 2, 0], [3, 4, 1], [2, 3, 2], [4, 5, 3]]
```

2. Next, we sort the augmented intervals by their start times.

```python
sorted_intervals = [[1, 2, 0], [2, 3, 2], [3, 4, 1], [4, 5, 3]]
```

3. We initialize the answer array with all elements set to `-1`.

```python
ans = [-1, -1, -1, -1]
```

4. Now, using a binary search, we look for the right interval for each interval in `sorted_intervals`.

   For interval `[1, 2, 0]`:

   - The end time is `2`.
   - The binary search tries to find the minimum index where `2` could be inserted to maintain the sorted order, which is index `1` (interval `[2, 3, 2]`).
   - Thus, the right interval index is `2`.

   For interval `[2, 3, 2]`:

   - The end time is `3`.
   - The binary search finds index `2` (interval `[3, 4, 1]`).
   - The right interval index is `1`.

   For interval `[3, 4, 1]`:

   - The end time is `4`.
   - The binary search finds index `3` (interval `[4, 5, 3]`).
   - The right interval index is `3`.

   For interval `[4, 5, 3]`:

   - The end time is `5`, and there's no interval starting after `5`.
   - The binary search returns an index of `4`, which is outside the array bounds.
   - There's no right interval, so the value remains `-1`.

5. After performing the binary search for all intervals, we update the answer array with the right intervals' indices.

```python
ans = [2, 1, 3, -1]
```

6. The `ans` array, which keeps track of the right interval for each interval, is returned as the final answer. In our example, this is `[2, 1, 3, -1]`, indicating that the interval `[1,2]` is followed by `[2,3]`, `[3,4]` follows `[2,3]`, and `[4,5]` follows `[3,4]`. The last interval, `[4,5]`, has no following interval that satisfies the conditions.

This approach efficiently finds the right intervals for each given interval with reduced time complexity.

Python Solution

```python
from bisect import bisect_left
from typing import List

class Solution:
    def findRightInterval(self, intervals: List[List[int]]) -> List[int]:
        # Append the original index to each interval
        for index, interval in enumerate(intervals):
            interval.append(index)

        # Sort intervals based on their start times
        intervals.sort()
        n = len(intervals)
        answer = [-1] * n  # Initialize the answer list with -1

        # Iterate through the sorted intervals
        for _, end, original_index in intervals:
            # Find the leftmost interval starting after the current interval's end
            right_index = bisect_left(intervals, [end])
            # If such an interval exists, update the corresponding position in answer
            if right_index < n:
                answer[original_index] = intervals[right_index][2]

        return answer

```

Java Solution

```java
class Solution {
    public int[] findRightInterval(int[][] intervals) {
        int numIntervals = intervals.length;
        // This list will hold the start points and their corresponding indices
        List<int[]> startIndexPairs = new ArrayList<>();

        // Populate the list with the start points and their indices
        for (int i = 0; i < numIntervals; ++i) {
            startIndexPairs.add(new int[] {intervals[i][0], i});
        }

        // Sort the startIndexPairs based on the start points in ascending order
        startIndexPairs.sort(Comparator.comparingInt(a -> a[0]));

        // Prepare an array to store the result
        int[] result = new int[numIntervals];

        // Initialize an index for placing interval results
        int resultIndex = 0;

        // Loop through each interval to find the right interval
        for (int[] interval : intervals) {
            int left = 0, right = numIntervals - 1;
            int intervalEnd = interval[1];

            // Binary search to find the minimum start point >= interval's end point
            while (left < right) {
                int mid = (left + right) / 2;
                if (startIndexPairs.get(mid)[0] >= intervalEnd) {
                    right = mid;
                } else {
                    left = mid + 1;
                }
            }

            // Check if the found start point is valid and set the result accordingly
            result[resultIndex++] = startIndexPairs.get(left)[0] < intervalEnd ? -1 : startIndexPairs.get(left)[1];
        }

        // Return the populated result array
        return result;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<int> findRightInterval(vector<vector<int>>& intervals) {
        int n = intervals.size(); // The total number of intervals

        // Vector of pairs to hold the start of each interval and its index
        vector<pair<int, int>> startWithIndexPairs;
        for (int i = 0; i < n; ++i) {
            // Emplace back will construct the pair in-place
            startWithIndexPairs.emplace_back(intervals[i][0], i);
        }
        // Sort the startWithIndexPairs array based on the interval starts
        sort(startWithIndexPairs.begin(), startWithIndexPairs.end());

        // This will hold the result; for each interval the index of the right interval
        vector<int> result;

        // Iterate over each interval to find the right interval
        for (const auto& interval : intervals) {
            int left = 0, right = n - 1; // Binary search bounds
            int end = interval[1]; // The end of the current interval

            // Perform a binary search to find the least start >= end
            while (left < right) {
                int mid = (left + right) / 2; // This avoids the overflow that (l+r)>>1 might cause
                if (startWithIndexPairs[mid].first >= end)
                    right = mid; // We have found a candidate, try to find an earlier one
                else
                    left = mid + 1; // Not a valid candidate, look further to the right
            }

            // Check if the found interval starts at or after the end of the current interval
            // If not, append -1 to indicate there is no right interval
            int index = (startWithIndexPairs[left].first >= end) ? startWithIndexPairs[left].second : -1;
            result.push_back(index);
        }

        return result; // Return the populated result vector
    }
};

```

Typescript Solution

```typescript
function findRightInterval(intervals: number[][]): number[] {
    // Get the total number of intervals
    const intervalCount = intervals.length;

    // Create an array to store the start points and their original indices
    const startPoints = Array.from({ length: intervalCount }, () => new Array<number>(2));

    // Fill the startPoints array with start points and their original indices
    for (let i = 0; i < intervalCount; i++) {
        startPoints[i][0] = intervals[i][0]; // Start point of interval
        startPoints[i][1] = i;               // Original index
    }

    // Sort the array of start points in ascending order
    startPoints.sort((a, b) => a[0] - b[0]);

    // Map each interval to the index of the interval with the closest start point that is greater than or equal to the end point of
    return intervals.map(([_, end]) => {
        let left = 0;
        let right = intervalCount;

        // Binary search to find the right interval
        while (left < right) {
            const mid = (left + right) >>> 1; // Equivalent to Math.floor((left + right) / 2)
            if (startPoints[mid][0] < end) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }

        // If left is out of bounds, return -1 to indicate no such interval was found
        if (left >= intervalCount) {
            return -1;
        }

        // Return the original index of the found interval
        return startPoints[left][1];
    });
}

```

Time and Space Complexity

The time complexity of the provided code consists of two major operations: sorting the `intervals` list and performing binary search using `bisect_left` for each interval.

1. Sorting the `intervals` list using the `sort` method has a time complexity of `O(n log n)`, where `n` is the number of intervals.
2. Iterating over each interval and performing a binary search using `bisect_left` has a time complexity of `O(n log n)` since the binary search operation is `O(log n)` and it is executed `n` times, once for each interval.

Thus, the combined time complexity of these operations would be `O(n log n + n log n)`. However, since both terms have the same order of growth, the time complexity simplifies to `O(n log n)`.

The space complexity of the code is `O(n)` for the following reasons:

1. The `intervals` list is expanded to include the original index position of each interval, but the overall space required is still linearly proportional to the number of intervals `n`.
2. The `ans` list is created to store the result for each interval, which requires `O(n)` space.
3. Apart from the two lists mentioned above, no additional space that scales with the size of the input is used.

Therefore, the total space complexity is `O(n)`.