986. Interval List Intersections

Medium Array Two Pointers

Problem Description

This LeetCode problem involves finding the intersection of two lists of sorted and disjoint intervals. An interval is defined as a pair of numbers `[a, b]` indicating all real numbers x where `a <= x <= b`. The intervals within each list do not overlap and are listed sequentially.

When two intervals intersect, the result is either an empty set (if there is no common overlap) or another interval that describes the common range between the two. The objective is to calculate the set of intersecting intervals between two given lists of intervals, where each list is independently sorted and non-overlapping.

Intuition

The intuition behind solving this problem lies in two main concepts: iteration and comparison. Since the lists are sorted, we can use two pointers, one for each list, and perform a step-wise comparison to determine if there are any overlaps between intervals.

The steps are as follows:

1. We initialize two pointers, each pointing to the first interval of the respective lists.
2. At each step, we consider the intervals where the pointers are currently pointing. We find the latest starting point and the earliest ending point between these two intervals. If the starting point is less than or equal to the ending point, then this range is the overlap of those intervals, and we add it to the answer.
3. We then move the pointer of the interval that finishes earlier to the next one in its list. This is because the finished interval cannot intersect with any other intervals in the other list, given that the intervals are disjoint and sorted.
4. We keep doing this until we have exhausted at least one list.

This approach ensures that we're always moving forward, never reassessing intervals we've already considered, which allows us to find all possible intersections in an efficient manner.

The implementation of the solution leverages a two-pointer approach which is an algorithmic pattern used to traverse arrays or lists in a certain order, exploiting any intrinsic order to optimize performance, space, or complexity. Here's how it's applied:

1. We start by initializing two integer indices, `i` and `j`, to zero. These will serve as the pointers that iterate over `firstList` and `secondList` respectively.
2. We run a `while` loop that continues as long as neither `i` nor `j` has reached the end of their respective lists (`i < len(firstList)` and `j < len(secondList)`).
3. Inside the loop, we extract the start (`s1`, `s2`) and end (`e1`, `e2`) points of the current intervals from `firstList` and `secondList` using tuple unpacking: `s1, e1, s2, e2 = *firstList[i], *secondList[j]`.
4. We then determine the start of the overlapping interval as the maximum of `s1` and `s2`, and the end of the overlapping interval as the minimum of `e1` and `e2`. If the start of this potential intersection is not greater than its end (`l <= r`), it means we have a valid overlapping interval, which we append to our answer list `ans`.
5. To move our pointers forward, we compare the ending points of the current intervals and increment the index (`i` or `j`) of the list with the smaller endpoint because any further intervals in the other list can't possibly intersect with the interval that has just finished.
6. After the loop concludes (when one list is fully traversed), we have considered all possible intersections, and we return the `ans` list which contains all the overlapping intervals we've found.

Data structures used in this implementation include lists for storing intervals and the result. No additional data structures are used since the solution is designed to work with the input lists themselves and builds the result in place, efficiently using space.

The time complexity of this approach is O(N + M), where N and M are the lengths of `firstList` and `secondList`. The space complexity is O(1) if we disregard the space required for the output, as we're only using a constant amount of additional space.

Example Walkthrough

Let's consider two lists of sorted and disjoint intervals: `firstList = [[1, 3], [5, 9]]` and `secondList = [[2, 4], [6, 8]]`, and walk through the solution approach to find their intersection.

1. Initialize two pointers: `i = 0` and `j = 0`.
2. The `while` loop commences since `i < len(firstList)` and `j < len(secondList)`.
   - At the start, we are looking at intervals `firstList[i] = [1, 3]` and `secondList[j] = [2, 4]`.
3. Extract the start and end points: `s1 = 1, e1 = 3, s2 = 2, e2 = 4`.
4. Determine the overlap's start and end:
   - The start is `max(s1, s2) = max(1, 2) = 2`.
   - The end is `min(e1, e2) = min(3, 4) = 3`.
   - Since `l <= r`, `[2,3]` is a valid intersection and is appended to `ans`.
5. Move pointers:
   - Compare the ending points `e1` and `e2`.
   - `e1` is smaller, so we increment `i` and now `i = 1` and `j = 0`.
6. Continue to the next iteration:
   - Now examining `firstList[i] = [5, 9]` and `secondList[j] = [2, 4]`.
   - We find `s1 = 5, e1 = 9, s2 = 2, e2 = 4`, hence no overlap because `max(5, 2) = 5` is greater than `min(9, 4) = 4`.
   - Since `e2` is smaller, increment `j` and now `j = 1` and `j = 1`.
7. Next iteration:
   - We are now looking at `firstList[i] = [5, 9]` and `secondList[j] = [6, 8]`.
   - Extract `s1 = 5, e1 = 9, s2 = 6, e2 = 8`.
   - The start of the overlap is `max(5, 6) = 6` and the end is `min(9, 8) = 8`.
   - We have a valid intersection `[6, 8]`, append it to `ans`.
8. Adjust pointers:
   - `e2` is smaller; therefore, increment `j` but `j` has reached the end of `secondList`.
9. The `while` loop ends since `j` has reached the end of `secondList`.

After the loop, our `ans` list contains the intersections: `[[2, 3], [6, 8]]`. The algorithm exits and returns `ans` as the final list of intersecting intervals between `firstList` and `secondList`.

Solution Implementation

Python

```python
class Solution:
    def intervalIntersection(self, firstList: List[List[int]], secondList: List[List[int]]) -> List[List[int]]:
        # Initialize indexes for firstList and secondList
        index_first = 0
        index_second = 0

        # This is where we will store the result intervals
        intersections = []

        # Iterate through both lists as long as neither is exhausted
        while index_first < len(firstList) and index_second < len(secondList):
            # Extract the start and end points of the current intervals for better readability
            start_first, end_first = firstList[index_first]
            start_second, end_second = secondList[index_second]

            # Determine the start and end of the overlapping interval, if any
            start_overlap = max(start_first, start_second)
            end_overlap = min(end_first, end_second)

            # If there's an overlap, the start of the overlap will be less than or equal to the end
            if start_overlap <= end_overlap:
                # Append the overlapping interval to the result list
                intersections.append([start_overlap, end_overlap])

            # Move to the next interval in either the first or second list,
            # selecting the one that ends earlier, as it cannot overlap with any further intervals
            if end_first < end_second:
                index_first += 1
            else:
                index_second += 1

        # Return the list of intersecting intervals
        return intersections
```

Java

```java
class Solution {
    public int[][] intervalIntersection(int[][] firstList, int[][] secondList) {
        List<int[]> intersections = new ArrayList<>();
        int firstLen = firstList.length, secondLen = secondList.length;

        // Use two-pointers technique to iterate through both lists
        int i = 0, j = 0; // i for firstList, j for secondList
        while (i < firstLen && j < secondLen) {
            // Find the start and end of the intersection, if it exists
            int startMax = Math.max(firstList[i][0], secondList[j][0]);
            int endMin = Math.min(firstList[i][1], secondList[j][1]);

            // Check if the intervals intersect
            if (startMax <= endMin) {
                // Store the intersection
                intersections.add(new int[] {startMax, endMin});
            }

            // Move to the next interval in the list that finishes earlier
            if (firstList[i][1] < secondList[j][1]) {
                i++; // Increment the pointer for the firstList
            } else {
                j++; // Increment the pointer for the secondList
            }
        }

        // Convert the list of intersections to an array before returning
        return intersections.toArray(new int[intersections.size()][]);
    }
}
```

C++

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    vector<vector<int>> intervalIntersection(vector<vector<int>>& firstList, vector<vector<int>>& secondList) {
        // Initialize the answer vector to store the intervals of intersection.
        vector<vector<int>> intersections;

        // Get the size of both input lists
        int firstListSize = firstList.size();
        int secondListSize = secondList.size();

        // Initialize pointers for firstList and secondList
        int i = 0, j = 0;

        // Iterate through both lists as long as there are elements in both
        while (i < firstListSize && j < secondListSize) {
            // Find the maximum of the start points
            int startMax = max(firstList[i][0], secondList[j][0]);

            // Find the minimum of the end points
            int endMin = min(firstList[i][1], secondList[j][1]);

            // Check if intervals overlap: if the start is less or equal to the end
            if (startMax <= endMin) {
                // Add the intersected interval to the answer list
                intersections.push_back({startMax, endMin});
            }

            // Move to the next interval in the list, based on end points comparison
            if (firstList[i][1] < secondList[j][1])
                i++; // Move forward in the first list
            else
                j++; // Move forward in the second list
        }

        // Return the list of intersected intervals
        return intersections;
    }
};
```

TypeScript

```typescript
function intervalIntersection(firstList: number[][], secondList: number[][]): number[][] {
    const firstLength = firstList.length; // Length of the first list
    const secondLength = secondList.length; // Length of the second list
    const intersections: number[][] = []; // Holds the intersections of intervals

    // Initialize pointers for both lists
    let firstIndex = 0;
    let secondIndex = 0;

    // Iterate through both lists until one is exhausted
    while (firstIndex < firstLength && secondIndex < secondLength) {
        // Calculate the start and end points of intersection
        const start = Math.max(firstList[firstIndex][0], secondList[secondIndex][0]);
        const end = Math.min(firstList[firstIndex][1], secondList[secondIndex][1]);

        // If there's an overlap, add the interval to the result list
        if (start <= end) {
            intersections.push([start, end]);
        }

        // Move the pointer for the list with the smaller endpoint forward
        if (firstList[firstIndex][1] < secondList[secondIndex][1]) {
            firstIndex++;
        } else {
            secondIndex++;
        }
    }

    // Return the list of intersecting intervals
    return intersections;
}
```

```python
class Solution:
    def intervalIntersection(self, firstList: List[List[int]], secondList: List[List[int]]) -> List[List[int]]:
        # Initialize indexes for firstList and secondList
        index_first = 0
        index_second = 0

        # This is where we will store the result intervals
        intersections = []

        # Iterate through both lists as long as neither is exhausted
        while index_first < len(firstList) and index_second < len(secondList):
            # Extract the start and end points of the current intervals for better readability
            start_first, end_first = firstList[index_first]
            start_second, end_second = secondList[index_second]

            # Determine the start and end of the overlapping interval, if any
            start_overlap = max(start_first, start_second)
            end_overlap = min(end_first, end_second)

            # If there's an overlap, the start of the overlap will be less than or equal to the end
            if start_overlap <= end_overlap:
                # Append the overlapping interval to the result list
                intersections.append([start_overlap, end_overlap])

            # Move to the next interval in either the first or second list,
            # selecting the one that ends earlier, as it cannot overlap with any further intervals
            if end_first < end_second:
                index_first += 1
            else:
                index_second += 1

        # Return the list of intersecting intervals
        return intersections
```

Time and Space Complexity

The time complexity of the given code is `O(N + M)`, where `N` is the length of `firstList` and `M` is the length of `secondList`. This is because the code iterates through both lists at most once. The two pointers `i` and `j` advance through their respective lists without ever backtracking.

The space complexity of the code is `O(K)`, where `K` is the number of intersecting intervals between `firstList` and `secondList`. In the worst case, every interval in `firstList` intersects with every interval in `secondList`, leading to `min(N, M)` intersections. The reason why it is not `O(N + M)` is that we only store the intersections, not the individual intervals from the input lists.