1893. Check if All the Integers in a Range Are Covered

Problem Description

In the given problem, you are provided with an array called `ranges`. This array contains subarrays where each subarray `[start, end]` represents a range of numbers from `start` to `end`, including both `start` and `end`. In addition to this array, you are given two integers `left` and `right`, which define a range of their own. The task is to determine whether every number within the range from `left` to `right` (inclusive) is covered by at least one range specified by the subarrays in `ranges`. If every number in the range `[left, right]` is found within one or more of the given `ranges`, you should return `true`. Otherwise, return `false`.

An integer is considered covered if it lies within any of the given ranges, inclusive of the range's endpoints. In simpler terms, you need to ensure that there are no gaps in coverage from `left` to `right`. If you find even one number in this range not covered by any intervals in `ranges`, the answer should be `false`.

Intuition

The idea behind the provided solution is to leverage a technique known as the "difference array". The difference array approach is useful in handling queries of range update operations efficiently and can be used in situations like this, where we have to add or subtract values over a range and then check the sum or coverage over a continuous range.

Here, we create an auxiliary array called `diff` with extra elements (up to 52, taking into account possible values from `left` and `right`). Initially, this `diff` array is filled with zeros. The intuition is to increase the value at the start of a range by `1` and decrease the value right after the end of a range by `1`. When we traverse through the `diff` array and compute the prefix sum at each point, we can determine the total coverage at that point. The prefix sum gives us an understanding of how many ranges cover a particular number.

If we follow through with calculating the prefix sums of the `diff` array, we would find that for any interval `[l, r]` in `ranges`, the coverage would be reflected correctly. Once we've processed all the ranges, we'll iterate through the `diff` array once more. While iterating, if we encounter any place within our `[left, right]` range where the cumulative coverage drops to `0` (which means it's not covered by any interval), then we immediately return `false`. If we make it through the entire `left` to `right` range and all points are covered, we return `true`.

The strength of this approach lies in efficiently updating the range coverage and then quickly assessing whether any point within the target range `[left, right]` is uncovered.

Solution Approach

The solution to this problem makes use of a simple but powerful concept called the difference array approach, which is particularly useful in scenarios involving increment and decrement operations over a range of elements.

To understand the implementation, we follow these steps:

1. We initialize a difference array `diff` with a length sufficient to cover all possible values in the problem statement. Here, we fixed its length to 52, which is an arbitrary choice to ensure that we can accommodate all ranges given that actual range requirements are not specified. This `diff` array tracks the net change at each point.
2. We iterate over the given `ranges` array. For each range `[l, r]`, we increment `diff[l]` by `1` and decrement `diff[r + 1]` by `1`. The increment at `diff[l]` indicates that at point `l`, the coverage starts (or increases), and the decrement at `diff[r + 1]` indicates that right after point `r`, the coverage ends (or decreases).
3. We then iterate over `diff` and compute the prefix sum at each point. We use a variable `cur` to keep track of the cumulative sum as we go.
   - The prefix sum up to a certain index `i` in the `diff` array essentially represents the number of ranges covering the point corresponding to `i`.
4. During the same iteration, we check each position `i` against our target coverage range `[left, right]`. If the prefix sum at `i` (i.e., `cur`) is `0` for any `i` within this interval, it means that point `i` is not covered by any range. Hence, we return `false`.
5. If we successfully iterate over the entire `[left, right]` interval without finding any points with `0` coverage, we return `true`, since this implies that all points within the target range are covered by at least one interval in `ranges`.

The key algorithmic concepts used in the implementation are iteration, conditional checks, and the management of prefix sums, which allow us to track the cumulative effect of all the ranges on any given point.

This approach is efficient because each range update (increment and decrement at the start and end points of the range) is performed in constant time, and the final check for uncovered points is performed in a single pass through the `diff` array.

Example Walkthrough

Let's consider a small example to illustrate the solution approach:

Assume we have an array `ranges` given as `[[1, 2], [5, 6], [1, 5]]`, and we need to check if all numbers in the range `[1, 6]` are covered by at least one of the intervals in `ranges`.

Following the steps outlined in the solution approach:

Step 0 (Initial Setup):

- We define `left = 1`, `right = 6`, and we initialize our `diff` array of size 52 (to cover the possible range) with all `0`s.

Step 1 (Updating the Difference Array):

- For the range `[1, 2]`, we increment `diff[1]` by `1` and decrement `diff[3]` by `1`.
- For the range `[5, 6]`, we increment `diff[5]` by `1` and decrement `diff[7]` by `1`.
- For the range `[1, 5]`, we increment `diff[1]` again by `1` and decrement `diff[6]` by `1`.

After step 1, the starting segment of our `diff` array looks like this: `[0, 2, 0, -1, 0, 1, -2, 1, ... (remaining all zeros)]`.

Step 2 (Computing Prefix Sums and Checking for Coverage):

- We initialize `cur` to `0` and start iterating from `1` to `6` (the range `[left, right]`). We will compute the prefix sum and check it against our coverage requirement:
  - At `i = 1`: `cur += diff[1]` (which is `2`), so `cur` becomes `2`.
  - At `i = 2`: `cur += diff[2]` (which is `0`), so `cur` remains `2`.
  - At `i = 3`: `cur += diff[3]` (which is `-1`), so `cur` becomes `1`.
  - At `i = 4`: `cur += diff[4]` (which is `0`), so `cur` remains `1`.
  - At `i = 5`: `cur += diff[5]` (which is `1`), so `cur` becomes `2`.
  - At `i = 6`: `cur += diff[6]` (which is `-2`), so `cur` becomes `0`.

During this iteration, we check whether `cur` becomes `0` before reaching the end of our range. Here, `cur` does become `0` at `i = 6`, indicating that the point `6` is not covered by any interval since the cumulative sum drops to zero at this point.

Thus according to our algorithm, we would return `false`, as there is at least one number (`6`) in the range `[left, right]` that isn't covered by any range in `ranges`.

Through this small example, we've followed the difference array approach to determine whether every number within the target range is covered by the given ranges. By performing constant time updates to our `diff` array and a single pass check, we efficiently arrive at our answer.

Python Solution

```python
class Solution:
    def isCovered(self, ranges: List[List[int]], left: int, right: int) -> bool:
        # Initialize a difference array with 52 elements, one extra to accommodate the 0-indexing and one more to handle the 'r+1' wi
        difference_array = [0] * 52

        # Iterate over each range in the ranges list.
        for range_start, range_end in ranges:
            # Increment the count at the start index of the range.
            difference_array[range_start] += 1
            # Decrement the count immediately after the end index of the range.
            difference_array[range_end + 1] -= 1

        # Accumulate the frequency count while traversing from start to end.
        current_coverage = 0
        for index, freq_change in enumerate(difference_array):
            # Update the coverage by adding the current frequency change.
            current_coverage += freq_change

            # If the index is in the query range [left, right] and the current coverage is 0,
            # it means this number is not covered by any range, thus return False.
            if left <= index <= right and current_coverage == 0:
                return False

        # If the loop completed without returning False, all numbers in [left, right] are covered.
        return True
```

Java Solution

```java
class Solution {
    // Method to check if all integers in the range [left, right] are covered by any of the ranges
    public boolean isCovered(int[][] ranges, int left, int right) {
        // Array for the difference between the count of start and end points
        int[] diff = new int[52]; // A size of 52 to cover ranges from 0 to 50 and to account for the end offset.

        // Loop through each range in the input array and update the diff array.
        for (int[] range : ranges) {
            int start = range[0]; // Start of the current range
            int end = range[1];   // End of the current range

            ++diff[start]; // Increment the start index to indicate the range starts here
            --diff[end + 1]; // Decrement the index after the end point to indicate the range ends before this index
        }

        // Variable to keep track of the current coverage status
        int coverage = 0;
        // Loop over the diff array and check if all numbers are covered
        for (int i = 0; i < diff.length; ++i) {
            coverage += diff[i]; // Add the difference to get the current number of ranges covering i

            // If the current number falls within the query range and is not covered by any range, return false.
            if (i >= left && i <= right && coverage == 0) {
                return false;
            }
        }

        // If we pass through the loop without returning false, all numbers in [left, right] are covered.
        return true;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    bool isCovered(vector<vector<int>>& ranges, int left, int right) {
        // Creating a difference array with an extra space to avoid index out-of-bounds,
        // since we will operate in the range of [1, 50] according to the problem's constraints
        int diff[52] = {};

        // Iterate through each range and maintain a difference array
        for (auto& range : ranges) {
            int rangeStart = range[0];
            int rangeEnd = range[1];
            // Increment at the start index, indicating the beginning of a range covering
            ++diff[rangeStart];
            // Decrement just after the end index, indicating the end of coverage
            --diff[rangeEnd + 1];
        }

        // Initialize a variable to track the coverage at each point
        int coverage = 0;

        // Iterate through the hypothetical line where the points need to be covered
        for (int i = 1; i <= 52; ++i) {
            // Apply the effect of the current index on the coverage
            coverage += diff[i];
            // Check if the current index falls within the range to be checked for coverage
            if (i >= left && i <= right) {
                // If the coverage drops to zero or below, it means this point is not covered
                if (coverage <= 0) {
                    return false;
                }
            }
        }

        // If the function hasn't returned false, all points are covered
        return true;
    }
};
```

Typescript Solution

```typescript
function isCovered(ranges: number[][], left: number, right: number): boolean {
    // Create a difference array with an initially filled with zeros to track coverages.
    const coverageDiff = new Array(52).fill(0);

    // Populate the difference array using the range updates (using the prefix sum technique).
    for (const [start, end] of ranges) {
        ++coverageDiff[start];
        --coverageDiff[end + 1];
    }

    // 'currentCoverage' will track the coverage of the current position by summing up values.
    let currentCoverage = 0;

    // Iterate through each position up to 51 (given the array starts at 0).
    for (let position = 0; position < 52; ++position) {
        // Add the coverage difference at the current position to the running total 'currentCoverage'.
        currentCoverage += coverageDiff[position];

        // If the current position is within the specified range and is not covered, return false.
        if (position >= left && position <= right && currentCoverage <= 0) {
            return false;
        }
    }

    // If all positions in the specified range are covered, return true.
    return true;
}
```

Time and Space Complexity

The given Python code implements a difference array to determine if all the numbers in the interval `[left, right]` are covered by at least one of the ranges in `ranges`.

Time complexity:

The time complexity of this algorithm is determined by several steps in the code:

1. **Initialize the difference array:** The difference array `diff` has a fixed size of `52`, so this step is `O(1)`.
2. **Populate the difference array:** For each range `[l, r]` in `ranges`, we perform a constant time operation to increment and decrement at position `l` and `r + 1`, respectively. If there are `n` ranges in `ranges`, this step has a complexity of `O(n)`.
3. **Accumulate the difference array and check coverage:** We then accumulate the difference array values to get the coverage count up to each index. Since the range of the `diff` array is from `0` to `51`, this step is `O(52)` which is `O(1)` as it has a fixed size.
4. **Checking the interval `[left, right]`:** We iterate through the `diff` array, which is a fixed size, and checking if the coverage is `0` for any point between `[left, right]`. This is also `O(1)` since the interval `[left, right]` is within a fixed-size range.

   The final time complexity is the sum of the complexities of these steps: `O(n) + O(1) + O(1) + O(1)`, which simplifies to `O(n)` where `n` is the number of ranges provided.

Space complexity:

The space complexity is determined by the storage used which is mainly for the difference array:

1. **Difference array `diff`:** A fixed-size array of length `52` is used, so space complexity is `O(1)`, as it doesn't depend on the input size.

Therefore, the overall space complexity of the algorithm is `O(1)`.