<https://leetcode.com/problems/check-if-all-the-integers-in-a-range-are-covered/description/>

You are given a 2D integer array ranges and two integers left and right. Each ranges[i] = [starti, endi] represents an **inclusive** interval between starti and endi.

Return true *if each integer in the inclusive range* [left, right] *is covered by* ***at least one*** *interval in* ranges. Return false *otherwise*.

An integer x is covered by an interval ranges[i] = [starti, endi] if starti <= x <= endi.

**Example 1:**

Input: ranges = [[1,2],[3,4],[5,6]], left = 2, right = 5

Output: true

Explanation: Every integer between 2 and 5 is covered:

- 2 is covered by the first range.

- 3 and 4 are covered by the second range.

- 5 is covered by the third range.

**Example 2:**

Input: ranges = [[1,10],[10,20]], left = 21, right = 21

Output: false

Explanation: 21 is not covered by any range.

**Constraints:**

* 1 <= ranges.length <= 50
* 1 <= starti <= endi <= 50
* 1 <= left <= right <= 50

**Attempt 1: 2023-12-03**

**Solution 1: Sweep Line (10 min)**

**Style 1: Create presum first**

class Solution {

public boolean isCovered(int[][] ranges, int left, int right) {

// 1 <= starti <= endi <= 50

int[] slots = new int[51];

for(int[] range : ranges) {

slots[range[0] - 1]++;

// '+1' for 'end' in range [start, end] inclusive

// same as L2848.Points That Intersect With Cars

// '-1' for 1-based range mapping to 0-based index

slots[range[1] + 1 - 1]--;

}

int[] presum = new int[slots.length + 1];

for(int i = 1; i < presum.length; i++) {

presum[i] = presum[i - 1] + slots[i - 1];

}

for(int i = left; i <= right; i++) {

if(presum[i] <= 0) {

return false;

}

}

return true;

}

}

Time Complexity: O(N)

Space Compleixty: O(N)

**Style 2: No need presum create**

class Solution {

public boolean isCovered(int[][] ranges, int left, int right) {

// 1 <= starti <= endi <= 50

int[] slots = new int[51];

for(int[] range : ranges) {

slots[range[0] - 1]++;

// '+1' for 'end' in range [start, end] inclusive

// same as L2848.Points That Intersect With Cars

// '-1' for 1-based range mapping to 0-based index

slots[range[1] + 1 - 1]--;

}

int count = 0;

for(int i = 0; i < slots.length; i++) {

count += slots[i];

// Don't forget '-1' for 1-based range mapping to 0-based index

if(count <= 0 && i >= left - 1 && i <= right - 1) {

return false;

}

}

return true;

}

}

Time Complexity: O(N)

Space Compleixty: O(N)

**Refer to**

<https://algo.monster/liteproblems/1893>

## 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:

* 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.
* 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).
* 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.

1. 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.

* 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.
* 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 0s.

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:

1. At i = 1: cur += diff[1] (which is 2), so cur becomes 2.
2. At i = 2: cur += diff[2] (which is 0), so cur remains 2.
3. At i = 3: cur += diff[3] (which is -1), so cur becomes 1.
4. At i = 4: cur += diff[4] (which is 0), so cur remains 1.
5. At i = 5: cur += diff[5] (which is 1), so cur becomes 2.
6. 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.

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 range 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;

}

}