<https://leetcode.com/problems/points-that-intersect-with-cars/description/>

You are given a **0-indexed** 2D integer array nums representing the coordinates of the cars parking on a number line. For any index i, nums[i] = [starti, endi] where starti is the starting point of the ith car and endi is the ending point of the ith car.

Return *the number of integer points on the line that are covered with* ***any part*** *of a car.*

**Example 1:**

Input: nums = [[3,6],[1,5],[4,7]]

Output: 7

Explanation: All the points from 1 to 7 intersect at least one car, therefore the answer would be 7.

**Example 2:**

Input: nums = [[1,3],[5,8]]

Output: 7

Explanation: Points intersecting at least one car are 1, 2, 3, 5, 6, 7, 8. There are a total of 7 points, therefore the answer would be 7.

**Constraints:**

* 1 <= nums.length <= 100
* nums[i].length == 2
* 1 <= starti <= endi <= 100

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

**Wrong Solution**

**错误的地方在于在数组操作上的end time比给定的[start, end]区间的end要+1，比如给定[3,6]，在时区覆盖这个问题上，6并不是真实的end，因为[3,6]表示从3到6都覆盖了，是闭区间，真正的在sweep line delta数组构建的时候针对end减1的状态发生在6+1=7的时候，而不是6的时候**

[[3,6],[1,5],[4,7]] => [[3,7),[1,6),[4,8)]

**Wrong Code**

class Solution {

public int numberOfPoints(List<List<Integer>> nums) {

// index{0, 99} + 1 => mapping to actual {start, end}

// time 1 to 100 defined by 1 <= starti <= endi <= 100

int[] timeline = new int[100];

for(List<Integer> num : nums) {

timeline[num.get(0) - 1]++;

timeline[num.get(1) - 1]--;

}

int result = 0;

int count = 0;

for(int i = 0; i < 100; i++) {

count += timeline[i];

if(count > 0) {

result++;

}

}

return result;

}

}

**Correct step by step**

Example:

==================================================

[[3,6],[1,5],[4,7]]

time 1 2 3 4 5 6 7

\* \* \* \* \* \* \*

+ - -> [3,6]

+ - -> [1,5]

+ - -> [4,7]

index 0 1 2 3 4 5 6

delta 1 0 1 1 -1 -1 -1

presum 1 1 2 3 2 1 0 -> count > 0 => 6

==================================================

The actual decrease should happen at 'end + 1'

[[3,6],[1,5],[4,7]] => [[3,7),[1,6),[4,8)]

time 1 2 3 4 5 6 7 8

\* \* \* \* \* \* \* \*

+ - -> [3,7)

+ - -> [1,6)

+ - -> [4,8)

index 0 1 2 3 4 5 6 7

delta 1 0 1 1 0 -1 -1 -1

presum 1 1 2 3 3 2 1 0 -> count > 0 => 7

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

class Solution {

public int numberOfPoints(List<List<Integer>> nums) {

// index{0, 99} + 1 => mapping to actual {start, end}

// time 1 to 100 defined by 1 <= starti <= endi <= 100

//int[] timeline = new int[100];

// But we need 1 more slot to cover the actual potential

// max end time as 101, which require timeline array max

// index as 100

int[] timeline = new int[101];

for(List<Integer> num : nums) {

timeline[num.get(0) - 1]++;

//timeline[num.get(1) - 1]--;

// the '+1' means each parking actual end

// e.g [[3,6],[1,5],[4,7]] => [[3,7),[1,6),[4,8)]

// the '-1' for mapping 1-based time to 0-based index

timeline[num.get(1) + 1 - 1]--;

}

int result = 0;

int count = 0;

//for(int i = 0; i < 100; i++) {

for(int i = 0; i < 101; i++) {

count += timeline[i];

if(count > 0) {

result++;

}

}

return result;

}

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://algo.monster/liteproblems/2848>

## Problem Description

In this problem, we have an array representing the range of positions occupied by cars on a number line. Each car's position is defined by a pair [start\_i, end\_i], indicating that the car covers every integer coordinate from start\_i to end\_i inclusively. We need to determine the total number of unique integer points covered by any part of a car on this number line.

To clarify with an example, if we have a car occupying positions from 2 to 4, it covers three points on the number line: 2, 3, and 4.

## Intuition

We can approach this problem by visualizing the number line and cars as intervals which can overlap. It may initially seem that we need to count every position each car occupies, but that would be inefficient, especially if the number line and the cars' range are large.

The provided solution uses a clever approach generally known as "difference array" or "prefix sum array" technique. The key insight here is to track changes at the start and just past the end positions of the cars' ranges. We increment at the start position, indicating the beginning of a car, and decrement right after the end position, indicating the end of a car's coverage.

This way, when we traverse the modified array and accumulate these changes, we can determine the coverage at each point on the number line. A positive number indicates that we are within a car's range at that point.

Finally, we sum up all the positions on the number line where the accumulated sum is greater than zero, which signifies a point being covered by at least one car.

Here are the steps in detail:

* Initialize a difference array d with zeros. Its size should be large enough to cover the maximum possible end point of any car.
* Iterate through each car's range in nums. For each range [a, b], increment d[a] by 1 (indicating a car starts) and decrement d[b + 1] by 1 (indicating the point after a car ends).
* Utilize accumulate from Python's itertools to turn the difference array into a prefix sum array. This will tell us the coverage at each point on the line.
* Count and return how many points on the number line have a positive coverage.

## Solution Approach

The solution adopts the difference array technique, a classical computational algorithm for efficiently implementing range update queries. We can describe the steps in the process as follows:

* **Initialize the Difference Array**: A difference array d is initialized with zeros and is large enough to cover all the potential end points of the cars. This array will help us record the change at the start and just past the end of each car's interval.
* **Update the Difference Array**: We iterate over each car's range in nums, for each car defined by a range [a, b]. We increment d[a] by 1 to denote the start of the car's range and decrement d[b + 1] by 1 to denote the point just after the end of the car's range. Thus, these increments and decrements represent the change at those specific points.
* **Cumulative Sum (Accumulate)**: After updating the difference array, we use accumulate from Python's itertools to calculate the cumulative sum. This results in an array where each index now represents how many cars are covering that specific point on the number line.
* **Count Positive Coverages**: We then count all the points with a positive value in the accumulated array. A positive value indicates that the point is covered by at least one car. So, the final count gives us the total number of unique integer points covered by cars.

### Code Explanation:

class Solution:

def numberOfPoints(self, nums: List[List[int]]) -> int:

# Step 1: Initialize the difference array with zeros.

d = [0] \* 110

# Step 2: Update the difference array based on car ranges.

for a, b in nums:

d[a] += 1 # Increment at the start of the car's range.

d[b + 1] -= 1 # Decrement right after the end of the car's range.

# Step 3: Apply accumulate to get the prefix sum, which gives us coverage per point.

# Step 4: Sum up the count of points where the accumulated value is greater than 0.

return sum(s > 0 for s in accumulate(d))

The choice of 110 as the array size is arbitrary and should cover the problem's given constraints. This number would need to be adjusted according to the specific constraints of the problem, such as the maximum value of the end of the cars' ranges.

This solution takes O(N+R) time, where N is the number of cars and R is the range of the number line we are interested in. The space complexity is O(R), which is required for the difference array.

### Example Walkthrough

To help illustrate the solution approach, let's walk through a small example. Suppose our input is the list of car ranges nums = [[1, 3], [2, 5], [6, 8]]. This means we have three cars covering the following ranges on the number line:

* Car 1 covers from point 1 to point 3.
* Car 2 covers from point 2 to point 5.
* Car 3 covers from point 6 to point 8.

Let's follow the steps of the solution:

* **Initialize the Difference Array**: We initialize a difference array, d, with zeros. For this example, we can choose a length that covers the maximum end point in our input, which is 8. But to be certain we cover the end points plus one, we'll extend it to 10: d = [0] \* 10.
* **Update the Difference Array**: We iterate over each car's range and update d accordingly.

1. For range [1, 3], we increment d[1] by 1 and decrement d[4] by 1 (since we increment one past the end of the interval).
2. For range [2, 5], we increment d[2] by 1 and decrement d[6] by 1.
3. For range [6, 8], we increment d[6] by 1 and decrement d[9] by 1.

After these updates, d looks like this: [0, 1, 1, 0, -1, 0, 1, 0, -1, -1].

* **Cumulative Sum (Accumulate)**: Now, we apply accumulate to d to get the prefix sum array. This reflects the number of cars covering each point. After calculating, the array looks like this: [0, 1, 2, 2, 1, 1, 2, 2, 1, 0].
* **Count Positive Coverages**: We count all the positive values in the accumulated array, which gives us the total number of unique integer points covered by cars. There are 8 positive values, so 8 points on the number line are covered by at least one car.

Following the implementation described in the content, our resulting numberOfPoints would be 8 for the given example. This demonstrates how the difference array method efficiently computes the required coverage, even with overlapping intervals.

class Solution {

// This method calculates the number of discrete points covered by a list of intervals

public int numberOfPoints(List<List<Integer>> intervals) {

int[] deltaArray = new int[110]; // Array to store the changes in the number of intervals covering a point

// Iterate over the list of intervals

for (List<Integer> interval : intervals) {

// For the start point of the interval, increment the count in the array

deltaArray[interval.get(0)]++;

// For the point after the end of the interval, decrement the count in the array

deltaArray[interval.get(1) + 1]--;

}

int totalPoints = 0; // Variable to hold the total number of points covered

int currentCoverage = 0; // Variable to hold the current cumulative coverage

// Iterate over the delta array to count the points that are covered by at least one interval

for (int countChange : deltaArray) {

// Update the current coverage by adding the change at this point

currentCoverage += countChange;

// If the current coverage is greater than 0, the point is covered

if (currentCoverage > 0) {

totalPoints++; // Increment the total points covered

}

}

return totalPoints; // Return the total number of points covered by the intervals

}

}