2848. Points That Intersect With Cars

Math Prefix Sum

Problem Description

Hash Table

Easy

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

large. The provided solution uses a clever approach generally known as "difference array" or "prefix sum array" technique. The key

need to count every position each car occupies, but that would be inefficient, especially if the number line and the cars' range are

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).

each point on the line.

- Utilize accumulate from Python's itertools to turn the difference array into a prefix sum array. This will tell us the coverage at
- 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

Initialize the Difference Array: A difference array d is initialized with zeros and is large enough to cover all the potential end

queries. We can describe the steps in the process as follows:

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

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

Code Explanation:

class Solution:

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.

Update the Difference Array: We iterate over each car's range in nums, for each car defined by a range [a, b]. We increment

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.

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.
- # 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.

Step 3: Apply accumulate to get the [prefix sum](/problems/subarray_sum), which gives us coverage per point

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

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

Car 1 covers from point 1 to point 3.

• Car 2 covers from point 2 to point 5.

10: d = [0] * 10.

car.

Python

from typing import List

for start, end in ranges:

for each point from 0 to 109.

```
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 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

 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

Count Positive Coverages: We count all the positive values in the accumulated array, which gives us the total number of

Following the implementation described in the content, our resulting number of Points would be 8 for the given example. This

demonstrates how the difference array method efficiently computes the required coverage, even with overlapping intervals.

unique integer points covered by cars. There are 8 positive values, so 8 points on the number line are covered by at least one

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).

Solution Implementation

Increment and decrement counts in the points_count list based on the ranges.

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

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

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

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

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

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

// Accumulate the changes to find how many intervals cover this point (if any).

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

Calculate the accumulated sum to get the actual counts

Count how many points have a non-zero count after accumulation,

which signifies these points are covered by at least one range.

accumulated_counts = list(accumulate(points_count))

return sum(count > 0 for count in accumulated_counts)

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

// Iterate over the list of intervals

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

for (int increment : diffArray) {

if (currentCoverage > 0) {

pointsCount++;

return pointsCount;

TypeScript

currentCoverage += increment;

// Return the total count of covered points.

function numberOfPoints(intervals: number[][]): number {

const differences: number[] = Array(110).fill(0);

for (const [start, end] of intervals) {

for (const countDifference of differences) {

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

overlapCount += countDifference;

differences[start]++;

differences[end + 1]--;

if (overlapCount > 0) {

totalPoints++;

// Iterate through each interval pair [start, end].

// Increment at the interval's starting index.

// Decrement right after the interval's ending index.

// Add the current difference to the overlap counter.

// Array to store the differences. Initialized with zeros for a size of 110.

let totalPoints = 0; // This will hold the total number of points with overlaps.

Initialize a list to keep track of point counts within specified ranges.

The list size is 110 to accommodate the point values in the problem constraints.

points_count[start] += 1 # Increment the count for the start of the range

points_count[end + 1] -= 1 # Decrement the count after the end of the range

Increment and decrement counts in the points_count list based on the ranges.

let overlapCount = 0; // Keeps track of the current overlap count as we traverse.

// Iterate through the differences array to find the total number of overlapping points.

// Each time the overlap count is positive, it implies a point where intervals overlap.

for (List<Integer> interval : intervals) {

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

points_count[start] += 1 # Increment the count for the start of the range

points_count[end + 1] -= 1 # Decrement the count after the end of the range

covering each point. After calculating, the array looks like this: [0, 1, 2, 2, 1, 1, 2, 2, 1, 0].

Update the Difference Array: We iterate over each car's range and update d accordingly.

For range [2, 5], we increment d[2] by 1 and decrement d[6] by 1.

from itertools import accumulate class Solution: def numberOfPoints(self, ranges: List[List[int]]) -> int: # Initialize a list to keep track of point counts within specified ranges. # The list size is 110 to accommodate the point values in the problem constraints. points_count = [0] * 110

```
# The code can now be used as follows:
# solution = Solution()
# result = solution.numberOfPoints([[1,3],[5,7],[9,10]])
# print(result) # This will output the number of points that are covered by at least one range.
```

Java

class Solution {

```
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
#include <vector>
class Solution {
public:
    // Function to calculate the number of distinct points covered by the given intervals.
    int numberOfPoints(std::vector<std::vector<int>>& intervals) {
       // Initialize a difference array of sufficient size to track the changes.
        int diffArray[110] = {};
       // Building the difference array with the given intervals.
        for (const auto& interval : intervals) {
           // Increment at the start of the interval
           diffArray[interval[0]]++;
           // Decrement just after the end of the interval
            diffArray[interval[1] + 1]--;
       // This variable will hold the number of points that are covered by at least one interval.
       int pointsCount = 0;
       // This variable is used to accumulate the values to get the original array representing the interval coverage.
       int currentCoverage = 0;
       // Loop over the range of the difference array. (Here it's assumed that the range 0-109 is sufficient.)
```

// If the current coverage is greater than 0, it means this point is covered by at least one interval.

```
// Return the total number of points where at least one interval overlaps.
return totalPoints;
```

from typing import List

class Solution:

from itertools import accumulate

 $points_count = [0] * 110$

for start, end in ranges:

Time and Space Complexity

Time Complexity

on the input nums.

```
# Calculate the accumulated sum to get the actual counts
       # for each point from 0 to 109.
        accumulated_counts = list(accumulate(points_count))
       # Count how many points have a non-zero count after accumulation,
       # which signifies these points are covered by at least one range.
        return sum(count > 0 for count in accumulated counts)
# The code can now be used as follows:
# solution = Solution()
# result = solution.numberOfPoints([[1,3],[5,7],[9,10]])
# print(result) # This will output the number of points that are covered by at least one range.
```

2. Iterating over the input nums, where each element is a list [a, b]. For each element, the code performs constant-time updates to the difference array d. If the length of nums is n, this operation takes O(n) time.

The time complexity of the code is determined by several key operations:

3. Accumulating the differences in d to compute the prefix sums. The use of the accumulate function from Python's itertools module has a time complexity of O(m), where m is the size of the difference array, which is a constant 110 in this case.

Therefore, the total space complexity remains O(1).

4. Summing up the positive values in the accumulated list with a list comprehension. This operation is again O(m) because it iterates over every element in the accumulated list.

1. Initializing the difference array d with 110 elements set to 0. This is an O(1) operation since the size of the array is constant and does not depend

Since both m and the size of the difference array are constant, the dominant term in the time complexity is the iteration over nums, giving an overall time complexity of O(n), where n is the length of nums.

Space Complexity

The space complexity is determined by the additional memory used by the program, which is primarily the difference array d. The array has a constant size of 110, which gives us an O(1) space complexity. The use of the accumulate function generates a temporary list with the same size as d, but since the size of d is constant, this does not impact the asymptotic space complexity.