497. Random Point in Non-overlapping Rectangles

Medium Reservoir Sampling Array Math Binary Search Ordered Set Prefix Sum Randomized

Leetcode Link

Problem Description

In this LeetCode problem, we are given a list of axis-aligned rectangles defined by their coordinates. Each rectangle is represented as $[a_i, b_i, x_i, y_i]$ where (a_i, b_i) is the bottom-left corner and (x_i, y_i) is the top-right corner of the ith rectangle. The goal is to create an algorithm that can randomly pick a point within the space covered by these rectangles. A point exactly on the edge of a rectangle is still considered to be within the rectangle. The algorithm must ensure any integer point in the space is equally likely to be chosen.

Intuition

To solve this problem, the solution must be able to pick a random point with uniform probability from the space defined by the rectangles. The main challenge is that the rectangles might have different areas, so simply picking a random rectangle and then a random point within it would result in a distribution where all possible points.

Here's the intuition behind the solution:

1. First, we calculate the area of each rectangle, which is the total number of integer points that can exist within that rectangle including the edges. For example, the area can be calculated by multiplying the width `(x2 - x1 + 1)` with the height `(y2 - y1 + 1)`.
2. We use a prefix sum array to keep a running total of the areas of all rectangles up to the current index. This way, each entry `self.s[i]` in the prefix sum array contains the total number of points that can be picked from the first `i+1` rectangles.
3. To pick a random point, we first decide which rectangle the point will be in. We do this by picking a random integer `v` between `1` and the sum of all rectangle areas (i.e., `self.s[-1]`). We use a binary search (through `bisect_left`) to find the first rectangle in the prefix sum array such that the running sum is greater than or equal to `v`. This effectively selects a rectangle with a probability proportional to its area.
4. Once the rectangle is selected, we randomly pick a point within it. We use `random.randint` to select an integer `x` coordinate between `x1` and `x2`, and a `y` coordinate between `y1` and `y2`. The combination of `[x, y]` gives us our random point within the chosen rectangle.

By ensuring that larger rectangles (with more possible points) are more likely to be chosen, and then evenly picking a point within the selected rectangle, the algorithm guarantees that any point in any rectangle is equally likely to be returned.

Solution Approach

The implementation of the solution uses a combination of prefix sums, binary search, and random selection to accomplish the goal of picking a random point according to the described probability distribution.

Here's a breakdown of how the solution approach is implemented:

1. **Prefix Sum Array**: In the `__init__` method, we initialize an array `self.s` to store the prefix sums of the areas of the given rectangles. This array is critical in helping us determine which rectangle should contain the randomly picked point. The area of a rectangle is calculated by `(x2 - x1 + 1) * (y2 - y1 + 1)`, which includes all integer points inside and on the edge of each rectangle.
2. **Cumulative Area Sum**: We iterate over the `rects` array and accumulate the area of the rectangles. This cumulative sum represents the total number of points we can pick from up to the current rectangle. It gives us a way to select a rectangle proportional to its area size; rectangles with larger areas have a broader range in the prefix sum array and thus have a higher probability of being selected.
3. **Random Point Selection**: In the `pick` method, we first select a rectangle. We generate a random integer `v` between `1` and `self.s[-1]`, which is the sum of areas of all rectangles. Using the `bisect_left` function, we find the index `idx` such that `self.s[idx]` is the smallest number in the prefix sum that is greater than or equal to `v`. This index corresponds to the rectangle in which the point will be located.
4. **Random Point within the Rectangle**: Once we have the rectangle, we use `random.randint` twice to get random `x` and `y` coordinates within the selected rectangle. The function `random.randint(a, b)` selects a random integer between `a` and `b` (inclusive), which is used to find the `x` coordinate within `[x1, x2]` and the `y` coordinate within `[y1, y2]`.
5. **Return the Point**: The random `x` and `y` values are combined into a list `[x, y]` representing the random point's coordinates. This point is guaranteed to be within the rectangle found in the previous steps, and the method returns this point.

The use of prefix sums and binary search allows the algorithm to efficiently handle the selection of rectangles with different area sizes, ensuring that the likelihood of picking any given point remains uniform across the entire space defined by `rects`.

Example Walkthrough

Imagine we have three rectangles represented as follows:

- Rectangle 1: `[1, 1, 3, 3]`
- Rectangle 2: `[4, 4, 5, 5]`
- Rectangle 3: `[6, 6, 8, 8]`

Their areas are calculated as:

- Area of Rectangle 1: `(3 - 1 + 1) * (3 - 1 + 1) = 9`
- Area of Rectangle 2: `(5 - 4 + 1) * (5 - 4 + 1) = 4`
- Area of Rectangle 3: `(8 - 6 + 1) * (8 - 6 + 1) = 9`

Now, let's create the prefix sum array (assuming we've already sorted our rectangles):

- Prefix sum after Rectangle 1: `9`
- Prefix sum after Rectangle 2: `9 (previous) + 4 = 13`
- Prefix sum after Rectangle 3: `13 (previous) + 9 = 22`

The prefix sums array looks like this: `self.s = [9, 13, 22]`.

When we call the `pick` method, here's what happens:

1. We select a random integer `v` between `1` and `22`. Let's say `v = 10`.
2. Using binary search (`bisect_left`), we find the first index in `self.s` that is not less than `10`. In our prefix sums array, this is index `1`.
3. This index corresponds to Rectangle 2 because `self.s[0] < v ≤ self.s[1]`.
4. We then pick a random point within Rectangle 2. The `x` coordinate is picked between `4` and `5`, and the `y` coordinate is also picked between `4` and `5`. Assuming `random.randint` gives us `x = 4` and `y = 4`, our chosen point is `[4, 4]`.

This example illustrates how each rectangle gets a chance proportional to its area to be selected for picking a point, ensuring a uniform distribution over all integer points within the space of the rectangles provided.

Python Solution

```python
import random
from bisect import bisect_left
from typing import List

class Solution:
    def __init__(self, rects: List[List[int]]):
        # Initialize the Solution object with a list of rectangles defined by bottom-left and top-right coordinates
        self.rectangles = rects
        # An array to store the accumulated count of points in all rectangles
        self.points_counts = [0] * len(rects)
        for i, (x1, y1, x2, y2) in enumerate(rects):
            # Calculate the count of points in the current rectangle and add it to the previous accumulated count
            # The count for a rectangle is the number of integer points inside it, which is (width + height)
            self.accumulated_counts[i] = self.accumulated_counts[i - 1] + (x2 - x1 + 1) * (y2 - y1 + 1)

    def pick(self) -> List[int]:
        # Pick a random integer point from the total number of points in all rectangles
        point_choice = random.randint(1, self.accumulated_counts[-1])
        # Find the rectangle that the point_choice falls into using binary search
        rectangle_index = bisect_left(self.accumulated_counts, point_choice)
        # Retrieve the coordinates of the chosen rectangle
        x1, y1, x2, y2 = self.rectangles[rectangle_index]
        # Transform the point_choice to fit within the range of the chosen rectangle
        # The point_choice is adjusted to start from 0 for the selected rectangle by subtracting the accumulated count of the previous rectangle
        adjusted_count = point_choice - (self.accumulated_counts[rectangle_index - 1] if rectangle_index > 0 else 0)
        # Calculate the width and height of the chosen rectangle
        width, height = x2 - x1 + 1, y2 - y1 + 1
        # Find the coordinates within the rectangle by using adjusted count, width, and height
        # The adjusted x coordinate is obtained by taking adjusted_count modulo width
        # The adjusted y coordinate is obtained by dividing the adjusted_count by width
        x = x1 + (adjusted_count - 1) % width
        y = y1 + (adjusted_count - 1) // width
        # Return the random point coordinates
        return [x, y]

# Your Solution object will be instantiated and called as such:
# obj = Solution(rects)
# param_1 = obj.pick()
```

Java Solution

```java
import java.util.Random;

class Solution {
    private final int[] prefixSums;
    private final int[][] rectangles;
    private final Random random = new Random();

    public Solution(int[][] rects) {
        int n = rects.length;
        this.rectangles = rects;
        prefixSums = new int[n + 1]; // Prefix sums of areas to help with random selection

        // Pre-compute the prefix sum array where each entry represents the total number
        // of points from the start up to and including the current rectangle.
        for (int i = 0; i < n; ++i) {
            // Calculate the area of the current rectangle using (width + height)
            // and add it to the previous sum.
            prefixSums[i + 1] = prefixSums[i] +
                (rectangles[i][2] - rectangles[i][0] + 1) *
                (rectangles[i][3] - rectangles[i][1] + 1);
        }
    }

    public int[] pick() {
        int v = rectangles.length;
        // Pick a random value that falls within the range of total number of points.
        int randomValue = 1 + random.nextInt(prefixSums[v]);

        // Binary search to find the rectangle that the point falls into based
        // on the randomValue.
        int left = 0, right = v;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (prefixSums[mid] >= randomValue) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        // Get the rectangle where the point falls into.
        int[] selectedRect = rectangles[left - 1];

        // Randomly pick a point within the selected rectangle.
        return new int[] {
            // X coordinate
            selectedRect[0] + random.nextInt(selectedRect[2] - selectedRect[0] + 1),
            // Y coordinate
            selectedRect[1] + random.nextInt(selectedRect[3] - selectedRect[1] + 1)
        };
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ctime>

using std::vector;
using std::lower_bound;
using std::srand;
using std::rand;
using std::time;

class Solution {
private:
    vector<int> prefixSums;          // Stores the cumulative area sums of the rectangles.
    vector<vector<int>> rectangles;  // Stores the list of input rectangles.

public:
    Solution(vector<vector<int>>& rects) {
        rectangles = rects; // Store a copy of the rectangles array.
        int numRectangles = rects.size();
        prefixSums.resize(numRectangles + 1, 0);

        // Calculate the cumulative area of rectangles to use for weighted random selection.
        for (int i = 0; i < numRectangles; ++i) {
            // Calculate the area of the rectangle and add it to the cumulative sum.
            prefixSums[i + 1] = prefixSums[i] +
                                (rects[i][2] - rects[i][0] + 1) *
                                (rects[i][3] - rects[i][1] + 1);
        }

        // Seed the random number generator with the current time.
        srand(static_cast<unsigned int>(time(nullptr)));
    }

    // Picks a random point uniformly from the total area covered by rectangles.
    vector<int> pick() {
        // Choose a random value from 1 to the total area inclusive.
        int target = 1 + rand() % prefixSums.back();

        // Find the rectangle that will contain the point.
        int idx = static_cast<int>(lower_bound(prefixSums.begin(), prefixSums.end(), target) - prefixSums.begin()) - 1;

        // Get the rectangle information.
        auto& rect = rectangles[idx];

        // Pick a random point within the chosen rectangle.
        int x = rect[0] + rand() % (rect[2] - rect[0] + 1);
        int y = rect[1] + rand() % (rect[3] - rect[1] + 1);

        return {x, y}; // Return the random point as a 2D vector.
    }
};

// The following lines are provided for context and to illustrate usage.
// This code would typically reside in a separate function and not within the class itself.
// Solution obj = Solution(rects);
// vector<int> param_1 = obj->pick();
```

Typescript Solution

```typescript
// The equivalent of the C++ vector in TypeScript is Array.
let prefixSums: number[] = []; // Stores the cumulative area sums of the rectangles.
let rectangles: number[][]; // Stores the list of input rectangles.

// Function that initializes the prefix sums. Equivalent to the constructor in the C++ version.
function initialize(rects: number[][]): void {
    rectangles = rects; // Store a copy of the rectangles array.
    let numRectangles = rectangles.length;
    prefixSums = new Array(numRectangles + 1).fill(0);

    // Calculate the cumulative area of rectangles to use for weighted random selection.
    for (let i = 0; i < numRectangles; ++i) {
        // Calculate the area of the rectangle and add it to the cumulative sum.
        prefixSums[i + 1] = prefixSums[i] +
                            (rects[i][2] - rects[i][0] + 1) *
                            (rects[i][3] - rects[i][1] + 1);
    }

    // Seed the random number generator.
    // Note: unlike C++, TypeScript's random number generator does not need to be seeded.
}

// Picks a random point uniformly from the total area covered by rectangles.
function pick(): number[] {
    // Choose a random value from 1 to the total area inclusive.
    let target = 1 + Math.floor(Math.random() * prefixSums[prefixSums.length - 1]);

    // Find the rectangle that will contain the point.
    let idx = prefixSums.findIndex(sum => sum >= target) - 1;

    // Get the rectangle information.
    let rect = rectangles[idx];

    // Pick a random point within the chosen rectangle.
    let x = rect[0] + Math.floor(Math.random() * (rect[2] - rect[0] + 1));
    let y = rect[1] + Math.floor(Math.random() * (rect[3] - rect[1] + 1));

    return [x, y]; // Return the random point as a 2D array.
}

// Example usage:
// initialize(rects);
// let point = pick();
// console.log(point);
```

Time and Space Complexity

Time Complexity:

- `__init__` method:
  - The method goes through each and calculates its area, which is done in $O(1)$ time for each rectangle.
  - This results in an overall time complexity of $O(n)$ for the `__init__` method, where n is the number of rectangles.
- `pick` method:
  - The `pick` method generates a random integer with `random.randint`, which is $O(1)$ in time complexity.
  - It then uses `bisect_left` to find the appropriate rectangle which takes $O(\log n)$ time, where n is the number of rectangles.
  - Finally, generating a random point within the rectangle is again $O(1)$.
  - Therefore, the time complexity of the `pick` method is $O(\log n)$.

Space Complexity

- The additional space used by an instance of the Solution class is for storing the cumulative sum of the areas in the `self.s` list and the list of rectangles `self.rects`.
  - Since `self.s` has one entry per rectangle, its space complexity is $O(n)$ where n is the number of rectangles.
  - The `self.rects` stores n rectangles, and each rectangle has 4 integers, so the space taken by `self.rects` is also $O(n)$.
  - Therefore, the total space complexity is $O(n)$, dominated by the storage of the rectangles and the cumulative sum.