528. Random Pick with Weight

Medium Array Math Binary Search Prefix Sum Randomized

Leetcode Link

Problem Description

In this problem, we are given an array `w` of positive integers where each integer represents the weight of the corresponding index. The objective is to implement a function called `pickIndex()` that randomly selects an index from 0 to `w.length - 1`. However, this isn't just any random selection; the index must be chosen such that the probability of selecting any index `i` is proportional to its weight `w[i]` relative to the sum of all weights in array `w`. The probability of picking index `i` is calculated by dividing `w[i]` by the sum of all elements in the array `w` (`sum(w)`). The task is to select an index randomly, in a weighted manner, according to these probabilities.

Intuition

The intuition behind the solution is to use a prefix sum array to convert the weights into a range of cumulative sums. A prefix sum array is an array where each element at index `i` stores the sum of all elements of the original array from 0 to `i`. This way, we can represent the weight of each index as a range in the cumulative sum.

Once we have the prefix sum array `self.s`, the idea is to generate a random number `x` between 1 and the sum of the weights (`self.s[-1]`). This random number effectively chooses a "position" within the total weight. Our goal now is to find the index in the original weights array `w` that corresponds to this position if weights were laid out on a number line according to their weight sizes.

We do this by performing a binary search on the prefix sum array to find the smallest prefix sum that is equal to or greater than the randomly picked number `x`. The binary search narrows down the range of possible positions until it finds the correct index whose weight range contains `x`.

This method ensures the index is chosen randomly, and with a probability proportional to its weight, fulfilling the requirement of the problem.

Solution Approach

The solution is implemented in two parts: the constructor `__init__(self, w: List[int])` and the method `pickIndex(self) -> int`. Here's how each part contributes to the overall solution:

1. **Constructor (`__init__`)**: Initialize the Solution class with the given weights array `w`. We calculate a prefix sums array which is stored in `self.s`. This array is built by starting with a 0 and then cumulatively adding the weights from the `w` array. The `self.s` array is one element longer than `w`, where `self.s[i]` represents the sum of weights from `w[0]` through `w[i-1]`.
   - If we have `w = [1, 3, 2]`, the resulting prefix sums array will be `self.s = [0, 1, 4, 6]`. Note how each element in `self.s` represents the cumulative weight up to but not including the current index in `w`.
2. **`pickIndex` method**: This method is where the random selection takes place, using the prefix sums array `self.s`.
   - First, we pick a random number `x` in the range from 1 to the cumulative weight of all elements (`self.s[-1]`).
   - Then, we perform a binary search to find the first element in the prefix sums array that is greater than or equal to this randomly chosen number `x`. The purpose is to find the segment where this random weight `x` would fall. We do this by maintaining two pointers `left` and `right` and repeatedly narrowing down the search space by adjusting these pointers based on the current middle element (`mid`), until `left` is just less than `right`.
   - The binary search condition `if self.s[mid] >= x` checks if the cumulative weight at `mid` is at least `x`. If so, we search to the left (adjust `right` to `mid`) as we may still find a smaller prefix sum that is still greater than or equal to `x`. Otherwise, we search to the right (set `left` to `mid + 1`) as we need a larger prefix sum to be greater than or equal to `x`.
   - Once the binary search completes, `left` will point to the first prefix sum that is greater than or equal to `x`, and hence `left - 1` will be the index of the weight in array `w` that corresponds to the random number `x`.

This solution efficiently simulates picking an index according to the weights' distribution. It uses the prefix sum to map a uniform distribution to the desired weighted distribution and employs binary search for fast index retrieval.

Example Walkthrough

Let's walkthrough a small example to illustrate the solution approach using the following array `w = [2, 5, 3]`:

Step 1: Initialize the Solution class

- Pass the array `w` to the constructor `__init__()`.
- Create a prefix sums array `self.s`. Starting with `[0]` and then adding each element from `w` cumulatively. For `w = [2, 5, 3]`, the prefix sums array will become `self.s = [0, 2, 7, 10]`.
  - `0` is the starting point.
  - `2` is the sum of weights up to but not including index `1` (`w[0]`).
  - `7` is the sum of weights up to but not including index `2` (`w[0] + w[1]`).
  - `10` is the sum of weights for all indexes (`w[0] + w[1] + w[2]`).

Step 2: Pick a random index with `pickIndex()` method

- Generate a random number `x` between `1` and `10` (the total weight).
- Suppose our random number `x` is `6`.
- Perform binary search to find the smallest element in `self.s` that is greater than or equal to `6`.
  - Initially, `left = 0` and `right = len(self.s) - 1`, which is `3`.
  - Our mid value in the first iteration will be `(0 + 3) // 2 = 1`.
  - Since `self.s[1]` is `2` and it's less than `6`, we make `left = mid + 1`, which is `2`.
  - On the next iteration, `left = 2`, `right = 3`, and so mid will be `(2 + 3) // 2 = 2`.
  - Now `self.s[2]` is `7` which is greater than `6`, set `right` to `mid`, now `right` becomes `2`.
  - The loop terminates when `left` is no longer less than `right`, so `left` will now point to `2`.

Since `left` is now `2`, index `1` (`left - 1`) in the original `w` array will be returned from `pickIndex()`. This is because `6` falls into the cumulative range `[2, 7)` (exclusive of 2 and inclusive of 7), corresponding to the second element (`5`) in the original weights array `w`. The choice of index `1` reflects the higher likelihood due to the weight of `5` in `w`.

This simple example demonstrates the weighted random selection using the prefix sums array and binary search.

Python Solution

```python
import random
from typing import List

class Solution:
    def __init__(self, weights: List[int]):
        # Initialize an empty list to store cumulative weights
        self.cumulative_weights = [0]
        # Build up the cumulative weight list for later binary search
        for weight in weights:
            self.cumulative_weights.append(self.cumulative_weights[-1] + weight)

    def pickIndex(self) -> int:
        # Generate a random number between 1 and the total sum of weights
        target = random.randint(1, self.cumulative_weights[-1])
        # Perform a binary search to find the target within the cumulative weights
        left, right = 1, len(self.cumulative_weights) - 1
        while left < right:
            # Calculate the middle index
            mid = (left + right) // 2
            # Since we want to find the first element that is not less than the target,
            # move the right pointer to mid if the middle cumulative weight is >= target
            if self.cumulative_weights[mid] >= target:
                right = mid
            # Otherwise, move the left pointer to one after the current middle
            else:
                left = mid + 1
        # The final index will be right - 1, since the cumulative_weights includes
        # an extra 0 at the beginning that we added during initialization
        return right - 1

# How to use this class:
# Create a Solution object with a given list of weights
# obj = Solution(weights)
# Pick an index based on the weight distribution
# index = obj.pickIndex()
```

Java Solution

```java
import java.util.Random;

class Solution {
    private int[] prefixSums; // stores the prefix sums of the weights
    private Random random = new Random(); // random number generator

    public Solution(int[] weights) {
        int n = weights.length;
        prefixSums = new int[n + 1];
        // Generate prefix sums array where each element represents the sum of weights up to that index.
        for (int i = 0; i < n; ++i) {
            prefixSums[i + 1] = prefixSums[i] + weights[i];
        }
    }

    public int pickIndex() {
        // Generate a random number between 1 and the total sum of weights.
        int n = 1 + random.nextInt(prefixSums[prefixSums.length - 1]);
        int left = 1, right = prefixSums.length - 1;

        // Perform binary search to find the index for which prefixSums[index] is greater than or equal to x.
        while (left < right) {
            int mid = (left + right) >>> 1; // Use unsigned right shift to avoid potential overflow
            if (prefixSums[mid] >= n) {
                // If the mid-index satisfies the condition, we search the left subarray.
                right = mid;
            } else {
                // Otherwise, we search the right subarray.
                left = mid + 1;
            }
        }
        // Since we have shifted our prefixSums array by one, we subtract one to get the original index.
        return left - 1;
    }
}

/**
 * The main class where instances of the Solution class can be created and the pickIndex() method can be called.
 */
public class Main {
    public static void main(String[] args) {
        int[] weights = {1, 3, 4, 6}; // for example
        Solution solution = new Solution(weights);
        int index = solution.pickIndex();
        System.out.println(index); // The picked index based on the weight
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <cstdlib> // For rand()

class Solution {
public:
    // Prefix sums array where each element at index i contains the sum of weights up to index i-1
    std::vector<int> prefixSums;

    // Constructor that initializes the Solution with a vector of weights
    Solution(std::vector<int>& weights) {
        int numWeights = weights.size();
        prefixSums.resize(numWeights + 1);
        // Build the prefix sums array.
        for (int i = 0; i < numWeights; ++i) {
            prefixSums[i + 1] = prefixSums[i] + weights[i];
        }
    }

    // Function to pick an index based on the weights (the weight at each index indicates the probability of picking that index)
    int pickIndex() {
        int numElements = prefixSums.size();
        // Generate a random number between 1 and the sum of all weights
        int randomNumber = 1 + rand() % prefixSums[numElements - 1];

        // Perform binary search to find the right index
        int left = 1, right = numElements - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            // If the prefix sum at mid is at least as large as the random number, search to the left
            if (prefixSums[mid] >= randomNumber)
                right = mid;
            else
                // Else, search to the right
                left = mid + 1;
        }
        // The index in the original array is left-1 because of the extra element at the beginning of prefixSums
        return left - 1;
    }
};

/**
 * Your Solution object will be instantiated and called as such:
 * Solution* obj = new Solution(weights);
 * int index = obj->pickIndex();
 */
```

Typescript Solution

```typescript
// Define the prefix sum array as a global variable.
let prefixSums: number[] = [];

/**
 * Initializes the prefix sums array using the input weights.
 * @param {number[]} weights - The list of weights, which corresponds to probabilities indirectly.
 */
function initialize(weights: number[]): void {
    const n = weights.length;
    prefixSums = new Array(n + 1).fill(0);
    for (let i = 0; i < n; ++i) {
        prefixSums[i + 1] = prefixSums[i] + weights[i];
    }
}

/**
 * Picks an index randomly based on the weights initialized.
 * The random pick is done using a binary search to find the interval
 * that the random number falls into (exclusive of the prefix sum) in intervals.
 * @return {number} The picked index corresponding to the original weights' distribution.
 */
function pickIndex(): number {
    const n = prefixSums.length;
    const randomNum = 1 + Math.floor(Math.random() * prefixSums[n - 1]);
    let left = 1;
    let right = n - 1;

    // Binary search to find the smallest index such that prefixSums[index] >= randomNum
    while (left < right) {
        const mid = Math.floor((left + right) / 2);
        if (prefixSums[mid] >= randomNum) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    // Left - 1 because the prefixSums array starts from 1 to n and we need to return 0 to n-1
    return left - 1;
}

// Example usage:
// initialize([10, 20, 15]); // Initializes the weights
// console.log(pickIndex()); // Logs an index, where the probability correlates with weight.
```

Time and Space Complexity

Time Complexity

For the `__init__` method:

- The time complexity is $O(n)$, where n is the length of the input list `w`. This is because we iterate through the list `w` once to compute the prefix sum array `self.s`.

For the `pickIndex` method:

- The time complexity is $O(\log n)$ because we use binary search to find the index in the prefix sum array. The binary search divides the search space in half during each iteration, which leads to a logarithmic time complexity.

Space Complexity

For both methods:

- The space complexity is $O(n)$ due to the storage required for the prefix sum array `self.s`, which has one more element than the original input list `w`.