398. Random Pick Index

Medium | Reservoir Sampling | Hash Table | Math | Randomized

Problem Description

The problem presents a class called `Solution` that is initialized with an array of integers `nums` that may contain duplicates. The core functionality we need to implement is the `pick` method, which should return the index of a randomly selected occurrence of a `target` number provided as input. Importantly, each possible index that matches the target should have an equal probability of being chosen. This means that if the target number appears multiple times in the array, our method should not show any bias towards any specific index.

Intuition

The intuition behind the solution approach to this problem lies in ensuring each index with the target value has an equal chance of being selected. This is a direct application of a well-known algorithm called Reservoir Sampling. Reservoir Sampling is useful when we have a large or unknown size of input and we need to select a random element (or a subset of elements) from it with equal probability.

In this case, we're iterating through the entire array, and each time we find the target number, we treat it as a potential candidate to be our random pick. As we encounter each target, we increase a count `n` which keeps track of how many times we've seen the target number so far. We then generate a random number 'x' between 1 and `n`. If 'x' is equal to `n`, which statistically will be 1 in `n` times, we set the `ans` variable to the index `i`. Through this process, every index has an equal chance (1/n) of being chosen since the choice is re-evaluated each time we see the target number. By the end of the array, this ensures that each index where `nums[i] == target` has been chosen with equal probability.

Solution Approach

The solution employs a clever implementation of Reservoir Sampling to ensure that each potential index for the target value is chosen with equal probability. Let's step through the implementation provided in the Reference Solution Approach:

- We start with the `__init__` method of the `Solution` class, which simply stores the `nums` array as an instance variable for later use.
- The `pick` method is where the logic for Reservoir Sampling comes into play. Here, we initialize two variables, `n`, which will track the number of times we've encountered the target value, and `ans`, which will hold our currently chosen index for the target.
- We then loop through each index-value pair in `nums` using the `enumerate` function.
- Inside the loop, we check if the current value `v` is equal to the target value. If it's not, we continue looping without doing anything further.
- However, if `v` is the target, we increment our count `n`. This count is crucial because it influences the probability of selecting the current index.
- Next, we generate a random number `x` between 1 and `n` using `random.randint(1, n)`. This random number decides whether we should update our current answer `ans` to the current index `i`.
- The condition `if x == n:` is the key to ensuring that each index has an equal chance of being chosen. This condition will be true exactly once in `n` occurrences on average, which aligns with the probability we want. When this condition is true, we set `ans` to `i`.
- After the loop completes, `ans` will hold the index of one of the occurrences of the target value, chosen at random. We return `ans`.

In summary, this algorithm effectively iterates through the list of numbers once (O(n) time complexity), using a constant amount of space (O(1) space complexity, not counting the input array), handling the potential of an unknown number of duplicates in a way that each target index has an equal likelihood of being selected.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose our `nums` array is `[4, 3, 3, 2, 3]`, and we want to pick an index at random where the target value `3` is located.

1. We initialize the `Solution` class with `nums`.
2. We call the `pick` method with the target value `3`.
3. The method begins to loop over `nums`. As it finds `3` at index `1`, we increase `n` to `1` and generate a random number `x`. Since `n` is `1`, `x` must also be `1`. We set `ans` to the index of our target, which is `1`.
4. The loop proceeds and finds another `3` at index `2`. It increases `n` to `2` and generates a random number `x` between `1` and `2`. If `x` turns out to be `1`, we keep `ans` as it is (`1`), but if `x` is `2`, we update `ans` to `2`.
5. Another `3` appears at index `4`. This time, `n` becomes `3`, and we repeat the process. A random number `x` is chosen between `1` and `3`. We have one-third of a chance to update `ans` to `4` if `x` is `3`.

Assuming our random numbers for each step are `1`, `2`, and `1` respectively, here is how the process unfolds:

- First occurrence (index `1`): `n` is `1`, `x` is `1`, so `ans` is set to `1`.
- Second occurrence (index `2`): `n` is `2`, `x` is `2`, so `ans` is updated to `2`.
- Third occurrence (index `4`): `n` is `3`, `x` is `1`, no change to `ans`.

Thus, the final answer returned by our `pick` method in this example could be `2`, which is one of the indexes where the target value `3` appears. The above iterations demonstrate that each index (`1`, `2`, and `4`) had an equal chance of being chosen. The probability for each was 1/3 by the end of the process.

Solution Implementation

Python

```python
import random
from typing import List

class Solution:
    def __init__(self, nums: List[int]):
        """Initialize the Solution object with a list of numbers."""
        self.nums = nums

    def pick(self, target: int) -> int:
        """
        Pick a random index from the list of numbers where the number at that index equals the target.
        This uses Reservoir Sampling to ensure that each such index has an equal probability of being chosen.
        """
        count = 0  # Counter for the occurrence of the target
        chosen_index = None  # Variable to store the randomly chosen index

        # Enumerate over the list of numbers
        for index, value in enumerate(self.nums):
            # Check if the current value matches the target
            if value == target:
                count += 1  # Increment the counter for each occurrence
                # Generate a random number between 1 and the current count (both inclusive)
                random_number = random.randint(1, count)
                # If the generated number equals the current count, update the chosen index
                if random_number == count:
                    chosen_index = index
        # Return the selected index which corresponds to the target in the list
        return chosen_index


# An example of how the Solution class might be instantiated and used:
# solution_instance = Solution(nums)
# random_index = solution_instance.pick(target)
```

Java

```java
import java.util.Random;

class Solution {
    private int[] nums; // This array holds the original array of numbers.
    private Random random = new Random(); // Random object to generate random numbers.

    // Constructor that receives an array of numbers.
    public Solution(int[] nums) {
        this.nums = nums; // Initialize the nums array with the given input array.
    }

    // Method to pick a random index where the target value is found in the nums array.
    public int pick(int target) {
        int count = 0; // Counter to track how many times we've seen the target so far.
        int result = 0; // Variable to keep the result index.

        // Iterating over the array to find target.
        for (int i = 0; i < nums.length; ++i) {
            if (nums[i] == target) { // Check if current element is the target.
                count++; // Increment the count since we have found the target.
                // Generate a random number between 1 and the number of times target has been seen inclusively.
                int randomNumber = 1 + random.nextInt(count);

                // If the random number equals to the count (probability 1/n),
                // set the result to current index i.
                if (randomNumber == count) {
                    result = i;
                }
            }
        }
        return result; // Return the index of target chosen uniformly at random.
    }
}

/**
 * The following is how to use the Solution class:
 * Solution solution = new Solution(nums);
 * int index = solution.pick(target);
 */
```

C++

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

class Solution {
private:
    vector<int> elements; // Renamed 'nums' to 'elements' for clarity

public:
    // Constructor which initializes the 'elements' vector
    Solution(vector<int>& nums) : elements(nums) {}

    // Function to pick a random index for a given target
    // This implements Reservoir Sampling
    int pick(int target) {
        int count = 0;        // Store the number of occurrences of 'target'
        int chosenIndex = -1; // Store the randomly chosen index of 'target'

        // Iterate through the elements to find 'target'
        for (int i = 0; i < elements.size(); ++i) {
            if (elements[i] == target) {
                count++; // Increment count for each occurrence
                // With probability 1/count, choose this index
                if ((rand() % count) == 0) {
                    chosenIndex = i;
                }
            }
        }
        // Return the randomly chosen index of 'target'
        return chosenIndex;
    }
};

/**
 * Usage:
 * vector<int> numbers = {1, 2, 3, 3, 3};
 * Solution* solution = new Solution(numbers);
 * int randomIndexForTarget = solution->pick(3);
 * // randomIndexForTarget will have the index of '3' chosen uniformly at random
 * delete solution; // Don't forget to free the allocated memory!
 */
```

TypeScript

```typescript
// Import the required module for generating random numbers
import { randomInt } from "crypto";

// Array to store the elements
let elements: number[] = [];

// Function to initialize the 'elements' array
function initialize(nums: number[]): void {
    elements = nums;
}

// Function to pick a random index for a given target using Reservoir Sampling
function pick(target: number): number {
    let count: number = 0;        // Store the number of occurrences of 'target'
    let chosenIndex: number = -1; // Store the randomly chosen index of 'target'

    // Iterate through the elements to find occurrences of 'target'
    for (let i = 0; i < elements.length; i++) {
        if (elements[i] === target) {
            count++; // Increment count for each occurrence
            // With probability 1/count, choose this index
            if (randomInt(count) === 0) {
                chosenIndex = i;
            }
        }
    }

    // Return the randomly chosen index of 'target'
    return chosenIndex;
}

// Example usage:
// initialize([1, 2, 3, 3, 3]);
// let randomIndexForTarget: number = pick(3);
// randomIndexForTarget will be an index of '3' chosen uniformly at random
```


```python
import random
from typing import List

class Solution:
    def __init__(self, nums: List[int]):
        """Initialize the Solution object with a list of numbers."""
        self.nums = nums

    def pick(self, target: int) -> int:
        """
        Pick a random index from the list of numbers where the number at that index equals the target.
        This uses Reservoir Sampling to ensure that each such index has an equal probability of being chosen.
        """
        count = 0  # Counter for the occurrence of the target
        chosen_index = None  # Variable to store the randomly chosen index

        # Enumerate over the list of numbers
        for index, value in enumerate(self.nums):
            # Check if the current value matches the target
            if value == target:
                count += 1  # Increment the counter for each occurrence
                # Generate a random number between 1 and the current count (both inclusive)
                random_number = random.randint(1, count)
                # If the generated number equals the current count, update the chosen index
                if random_number == count:
                    chosen_index = index
        # Return the selected index which corresponds to the target in the list
        return chosen_index


# An example of how the Solution class might be instantiated and used:
# solution_instance = Solution(nums)
# random_index = solution_instance.pick(target)
```


Time and Space Complexity

Time Complexity

The time complexity of the `pick` method is `O(N)`, where `N` is the total number of elements in `nums`. This is because, in the worst case, we have to iterate over all the elements in `nums` to find all occurrences of `target` and decide whether to pick each occurrence or not.

During each iteration, we perform the following operations:

1. Compare the current value `v` with `target`.
2. If they match, we increment `n`.
3. Generate a random number `x` with `random.randint(1, n)`, which has constant time complexity `O(1)`.
4. Compare `x` to `n` and possibly update `ans`.

These operations are within the single pass loop through `nums`, hence maintaining the overall time complexity of `O(N)`.

Space Complexity

The space complexity of the `pick` method is `O(1)`. The additional space required for the method execution does not depend on the size of the input array but only on a fixed set of variables (`n`, `ans`, `i`, and `v`), which use a constant amount of space.

The class `Solution` itself has space complexity `O(N)`, where `N` is the number of elements in `nums`, since it stores the entire list of numbers.

However, when analyzing the space complexity of the `pick` method, we consider only the extra space used by the method excluding the space used to store the input, which in this case remains constant.