710. Random Pick with Blacklist

Hard Array Hash Table Math Binary Search Sorting Randomized Leetcode Link

Problem Description

The problem states that an integer `n` and an array `blacklist` are given. The task is to design an algorithm that can randomly choose an integer from 0 to `n - 1` that is not included in the `blacklist`. Every valid integer within the range should have an equal chance of being picked. The algorithm should also minimize the number of calls to the random function provided by the programming language you are using.

This means that we need to find an efficient way to track which numbers have been blacklisted and ensure that only non-blacklisted numbers are considered when we want to pick a random number. All non-blacklisted numbers should be equally likely to be chosen.

Intuition

The primary challenge lies in designing an algorithm that maintains the equal probability of picking a non-blacklisted number while keeping the process efficient.

One intuitive way is to map all the blacklisted numbers within the range `[0, k)` to some non-blacklisted numbers in the range `[k, n)`, where `k` is the count of non-blacklisted numbers. This is done because the random number selection will only happen within the `[0, k)` range, thereby avoiding the blacklisted numbers. If a number within this range is not blacklisted, it maps to itself. If it is blacklisted, it maps to a non-blacklisted number outside the range.

The intuition for the solution comes from the observation that picking a random number in the `[0, n - len(blacklist))` range will ensure that each pick is equally likely. Then we can 'remap' these picks if they are supposed to be a blacklisted value to some non-blacklisted value. This remapping is done using a dictionary that keeps track of the blacklisted numbers and their corresponding non-blacklisted 'replacement'. When picking a number, we first check if it needs to be remapped and get the corresponding value if needed.

The `Solution` class is initialized by creating the mapping of blacklisted numbers if they are within the `[0, k)` range to the first non-blacklisted numbers outside of this range. As for the `pick` method, it randomly selects a number within `[0, k)` and returns the corresponding value in the mapping if it exists, or itself if it is not blacklisted.

Solution Approach

The solution to this problem employs a smart mapping strategy and a clever use of data structures to efficiently enable the random selection of non-blacklisted numbers. By understanding that only those blacklisted numbers that fall within the range of `[0, k)` truly interfere with our random pick, the solution avoids the need to deal with ones that are naturally excluded from the random range.

Here's a walk-through of the implementation details:

1. **Initialization (`__init__` method):**
    - Compute `k` as `n` minus the size of the blacklist, which effectively represents the count of valid integers after excluding the blacklisted ones.
    - Initialize a dictionary `self.d` that will hold the mapping from the blacklisted within `[0, k)` to the unblacklisted numbers in `[k, n)`.
    - Loop through each number in the blacklist:
        - For each blacklisted number `b` that is less than `k` (and hence within the range of numbers we can randomly pick from), find a number `i` (starting from `k`) which is not in the blacklist. This determines the non-blacklisted number that `b` will map to.
        - Update `self.d[b]` to `i` to keep the track of the mapping.
        - Move to the next integer for mapping (increment `i`).
2. **Random Pick (`pick` method):**
    - Randomly select an integer `x` within the range `[0, k)`.
    - If `x` is a key in our mapping dictionary (which signifies that `x` is blacklisted), retrieve the mapped value (the substitute non-blacklisted number). Otherwise, if `x` is not blacklisted, it can be returned as is.

The use of the dictionary data structure allows the algorithm to quickly access the remapped value, thereby minimizing the runtime complexity for each pick operation to constant time (`O(1)`). Together with only initializing the mappings once in the constructor, this makes the random pick operation very efficient.

By limiting the calls to the random function to the interval `[0, k)`, the algorithm ensures that it minimizes the number of calls to this potentially expensive operation.

Overall, this solution is both time and space-efficient, where space complexity is dictated by the number of blacklisted numbers within `[0, k)`, and the time complexity for the `pick` method is constant, irrespective of the size of the blacklist.

Example Walkthrough

To illustrate the solution approach, consider the following example:

- Let `n` be 10, so the valid range of numbers is `[0, 9]`.
- Let the `blacklist` be `[3, 5, 8]`.

In the initialization step of our solution, we perform the following actions:

1. Compute `k` as `n - len(blacklist)`, which is `10 - 3 = 7`. Therefore, `k` is 7, and our random pick range becomes `[0, 6]`, because we want to avoid picking a blacklisted number directly.

2. Initialize an empty dictionary `self.d` to hold the mappings. Since `3` and `5` are the only blacklisted numbers within `[0, k)`, they will need mapping to non-blacklisted numbers in `[k, n)`.

3. We map blacklisted numbers within `[0, k)` to the first available non-blacklisted numbers starting from `k`. Starting at `k = 7`, we map:
    - Blacklisted `3` maps to `7`, as `7` is the first non-blacklisted number available.
    - Blacklisted `5` maps to `9`, as `8` is blacklisted and `9` is the next available non-blacklisted number.

The dictionary now looks like this: `self.d = {3: 7, 5: 9}`.

Let's proceed with the random pick method:

1. We invoke the `pick` method to randomly select a number. Let's say the random function returns `5`.

2. We check if `5` is in our mapping dictionary `self.d`. Since `5` is a key in `self.d`, we return `self.d[5]`, which is `9`.

3. If the random function returned `4`, since `4` is not in `self.d`, `4` is not blacklisted, and we would return `4` directly.

This process ensures that numbers are only picked from the valid set `[0, 6]`, and if a blacklisted number is picked, it is properly mapped to a non-blacklisted number hence maintaining equal probability of picking any non-blacklisted number. The pick operation remains efficient as it involves a constant-time dictionary lookup and a random choice within the reduced range.

Python Solution

```python
from random import randrange
from typing import List

class Solution:
    def __init__(self, n: int, blacklist: List[int]):
        self.mapping_range_limit = n - len(blacklist) # The upper limit for the mapping
        self.mapping_dict = {} # Dictionary to hold the mapping
        blacklist_set = set(blacklist) # Convert the blacklist to a set for efficient lookup

        # Initialize an index to start mapping from blacklist numbers less than the mapping range limit
        mapping_start_index = self.mapping_range_limit

        # Iterate through each blacklisted number
        for black_number in blacklist:
            # If the blacklisted number is within the range of mappable values,
            # find a non-blacklisted number to map it to
            if black_number < self.mapping_range_limit:
                # Skip all numbers that are in the blacklist until a valid one is found
                while mapping_start_index in blacklist_set:
                    mapping_start_index += 1
                # Map the blacklisted number within range to a non-blacklisted number
                self.mapping_dict[black_number] = mapping_start_index
                # Move on to the next possible number for mapping
                mapping_start_index += 1

    def pick(self) -> int:
        # Pick a random number from 0 to the upper exclusive limit (mapping_range_limit)
        random_pick = randrange(self.mapping_range_limit)
        # Return the mapped value if it exists, otherwise return the random_pick itself
        return self.mapping_dict.get(random_pick, random_pick)

# Example usage:
# Instantiate the Solution object with a length n and a blacklist
# obj = Solution(n, blacklist)
# Get a random number that's not on the blacklist
# param_1 = obj.pick()

```

Java Solution

```java
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Random;
import java.util.Set;

class Solution {
    // A map to keep the mapping of blacklisted numbers to the safe numbers.
    private Map<Integer, Integer> mapping = new HashMap<>();
    // Instance of Random to generate random numbers.
    private Random random = new Random();
    // The threshold for picking a safe number.
    private int threshold;

    // Constructor that takes the total number of elements (n) and the list of blacklisted elements (blacklist).
    public Solution(int n, int[] blacklist) {
        threshold = n - blacklist.length;
        // Convert the blacklist into a set for faster lookup.
        Set<Integer> blacklistSet = new HashSet<>();
        for (int b : blacklist) {
            blacklistSet.add(b);
        }
        // Initialize a variable to the start of the possible non-blacklisted numbers above the threshold.
        int nextSafeNumber = threshold;
        for (int b : blacklist) {
            // Only remap blacklisted numbers below the threshold.
            if (b < threshold) {
                // Find the next non-blacklisted number to map to.
                while (blacklistSet.contains(nextSafeNumber)) {
                    ++nextSafeNumber;
                }
                // Add the mapping from the blacklisted number to the safe number.
                mapping.put(b, nextSafeNumber++);
            }
        }
    }

    // Function to pick a random non-blacklisted number.
    public int pick() {
        // Generate a random number within the range [0, threshold).
        int randomNumber = random.nextInt(threshold);
        // If the number is remapped, return the remapped number, otherwise return it as is.
        return mapping.getOrDefault(randomNumber, randomNumber);
    }
}

/**
 * Your Solution object will be instantiated and called as such:
 * Solution obj = new Solution(n, blacklist);
 * int param_1 = obj.pick();
 */

```

C++ Solution

```cpp
#include <unordered_map>
#include <unordered_set>
#include <vector>

class Solution {
private:
    std::unordered_map<int, int> whitelistMapping;
    int whitelistSize;

public:
    // Constructor takes the size of the array (n) and a list of blacklisted indices (blacklist).
    Solution(int n, std::vector<int>& blacklist) {
        // Calculate the effective size of the whitelist (array size minus the size of the blacklist).
        whitelistSize = n - blacklist.size();

        // Set an index pointing to the start of the upper range which is outside of the whitelist.
        int upperIndex = whitelistSize;
        std::unordered_set<int> blacklistSet(blacklist.begin(), blacklist.end());

        // Iterate over each number in the blacklist.
        for (int& blackNumber : blacklist) {
            // Only process blacklisted numbers that would have been chosen otherwise.
            if (blackNumber < whitelistSize) {
                // Find the next number not in the blacklist past the initial whitelist range.
                while (blacklistSet.count(upperIndex)) {
                    ++upperIndex;
                }
                // Establish a mapping from the blacklisted number to a whitelisted number from the upper range.
                whitelistMapping[blackNumber] = upperIndex++;
            }
        }
    }

    // Function to randomly pick an index from the available whitelisted indices
    int pick() {
        // Choose a random index from the initial range of whitelisted indices.
        int randomIndex = rand() % whitelistSize;
        // If the index has been remapped due to being blacklisted, fetch the remapped index.
        // Otherwise, return it as is because it's not blacklisted.
        return whitelistMapping.count(randomIndex) ? whitelistMapping[randomIndex] : randomIndex;
    }
};

/**
 * Your Solution object will be instantiated and called as such:
 * Solution* obj = new Solution(n, blacklist);
 * int result = obj->pick();
 */

```

Typescript Solution

```typescript
// TypeScript lacks `unordered_map` and `unordered_set` but has the `Map` and `Set` classes.

let whitelistMapping: Map<number, number>;
let whitelistSize: number;

// Initialize necessary structures and perform calculations as would be performed in the constructor.
function initialize(n: number, blacklist: number[]): void {
    whitelistMapping = new Map<number, number>();
    whitelistSize = n - blacklist.length;

    // Convert the blacklist array into a Set for O(1) lookups.
    let blacklistSet: Set<number> = new Set(blacklist);
    let upperIndex: number = whitelistSize;

    // Iterate over each item in the blacklist array.
    blacklist.forEach(blackNumber => {
        // Focus on blacklisted numbers within the initial whitelist range.
        if (blackNumber < whitelistSize) {
            // Locate a non-blacklisted number beyond the initial whitelist range.
            while (blacklistSet.has(upperIndex)) {
                upperIndex++;
            }
            // Create a mapping for the blacklisted number to the identified whitelisted number.
            whitelistMapping.set(blackNumber, upperIndex++);
        }
    });
}

// Function to select a random whitelisted index.
function pick(): number {
    // Generate a random index within the initial whitelist range.
    let randomIndex: number = Math.floor(Math.random() * whitelistSize);

    // Check and return the mapped index if originally blacklisted, else return the original.
    return whitelistMapping.get(randomIndex) ?? randomIndex;
}

// Usage example:
// initialize(100, [1, 2, 3]);
// let randomPick: number = pick();

```

Time and Space Complexity

Time Complexity

- `__init__` method: The initialization method initializes the mapping of a blacklist to new values. For each value in the blacklist, there is a possibility of a while loop running if the value is less than `self.k`. Considering that `i` is incremented each time to find a non-blacklisted value, and assuming the worst-case scenario where all the blacklisted values are less than `k`, the while loop has to iterate over all values from `self.k` to `n - 1` in the worst case. However, since each value from the blacklist requires only one mapping operation, and we do not revisit already mapped values, the time complexity would be O(B), where B is the length of the blacklist, assuming set membership test operations are O(1) which is the average case for Python sets.

- `pick` method: This method generates a random number and looks up the value in the dictionary (if it is a blacklisted value that has been remapped). Generating a random number is O(1), and dictionary lookup is on average O(1). In the worst case (not likely in average scenarios) due to hash collisions, it could be O(k) but we generally do not consider this for average-case analysis, particularly because Python dictionaries are well optimized. Hence, the time complexity of `pick` is O(1).

Space Complexity

- The space complexity of the `__init__` method is O(B). The dictionary `self.d` used to store the remapped values will at most have B entries, where B is the length of the blacklist. The set `black` also takes O(B) space.
- The `pick` method does not use any additional space that scales with the input size, so its space complexity is O(1).