822. Card Flipping Game

Medium Array Hash Table

Problem Description

You have a collection of cards, each with a number on the front and back. We call these two numbers "fronts" and "backs", and they are provided in two arrays where each array is of length `n`. The arrangement is such that `fronts[i]` corresponds to the number on the front of the i-th card, and `backs[i]` corresponds to the number on the back of the i-th card. At the start, all cards are laid out in a way that only the fronts are visible.

Your task is to flip over any number of cards such that there's at least one number that appears only on the backs of the cards and not on any front. If such an arrangement is possible, the goal is to make sure the "good" number is as small as possible. If you cannot achieve this (i.e., all numbers appear on at least one front), then you return `0`.

The problem is challenging because you need to decide which cards to flip (if any) to minimize the "good" number that remains hidden on the backside, without it appearing on any visible front at the same time.

Intuition

To solve this problem, the key intuition is to identify numbers that appear on both sides of the same card. These numbers can never be "good" numbers, regardless of how we flip the cards, because flipping a card would just move the number from the back to the front.

Therefore, we begin by creating a set `s` of all numbers that are present on both the front and back of the same card. None of these numbers can be considered for our "good" number.

Once we have this set, the next step is to search for the smallest number that is not in this exclusion set `s`. We need to check both the `fronts` and `backs` arrays for potential "good" numbers because only flipping cards will reveal some back numbers and hide some front numbers.

The `min` function can be used to find the smallest number fulfilling these criteria. We use a generator expression `(x for x in chain(fronts, backs) if x not in s)` that goes through all the numbers in both `fronts` and `backs` but only yields the ones not in the exclusion set `s`.

Solution Approach

The solution approach uses the Python set as a primary data structure and the generator expression for finding the minimum "good" number.

Firstly, we create a set `s` that consists of all numbers which appear on both sides of the same card. In Python, a set is an unordered collection of distinct hashable objects, and it's specifically designed to quickly check for membership, which is exactly what's needed here.

```
s = {a for a, b in zip(fronts, backs) if a == b}
```

This line uses a set comprehension along with `zip`. The `zip` function takes two or more sequences (like our `fronts` and `backs` arrays) and creates a new iterator that yields tuples containing elements from each sequence in parallel. The set comprehension then adds the front number `a` to the set `s` only if it matches the corresponding back number `b`, indicating the number is present on both sides of the same card.

Next, we need to find the minimum number that's not in the set `s`. To achieve this without creating additional temporary lists, a generator expression coupled with `chain` from the `itertools` module is used. The `chain` function allows us to iterate over multiple sequences in a single loop, effectively treating them as one continuous sequence. This is done without the overhead of combining the lists.

```
min((x for x in chain(fronts, backs) if x not in s), default=0)
```

Here, the generator expression generates numbers from both `fronts` and `backs`, but filters out any number that is found in the set `s`. The `min` function then calculates the minimum of these numbers. The use of `default=0` is a safety net that specifies what to return if the generator yields no values – which happens if we found all numbers in both `fronts` and `backs` are also in the set `s`.

This approach efficiently utilizes memory and only calculates the required minimum, without additional processing of the two lists.

In summary:

- A set is used for its fast membership testing to exclude numbers that can't be "good".
- `zip` is used to iterate two lists in parallel.
- Generator expressions provide a memory-efficient way to handle large datasets.
- `chain` seamlessly concatenates our two lists for single-pass processing.
- `min` finds the smallest "good" number, with `default=0` to handle edge cases where no "good" number exists.

Using this combination of techniques results in a neat and efficient solution that cleverly avoids the necessity for nested loops or extra data manipulation that could complicate or slow down the algorithm.

Example Walkthrough

Let's say we have the following `fronts` and `backs` arrays for our cards:

```
fronts = [1, 2, 4, 4, 7]
backs = [1, 3, 4, 2, 8]
```

We are given 5 cards, and the numbers on the fronts and backs of these cards are listed in the arrays. We want to flip some of these cards to meet the condition stated in the problem.

Now, we follow the solution approach described:

1. Create the set `s` of numbers that are present on both sides of the same card. Here, the numbers `1` and `4` fit this condition:

```
s = {1, 4}  # Since cards 1 and 3 have the same number on front and back
```

2. It's clear that neither `1` nor `4` can be our "good" number, as they appear on both sides of certain cards.
3. We need to find the smallest number that is not in the set `s`. We will look at both `fronts` and `backs` but only consider the numbers that are not in `s`.

```
fronts = [1, 2, 4, 4, 7]
backs = [1, 3, 4, 2, 8]
s = {1, 4}

# Filtering numbers not in `s` and finding the minimum
potential_good_numbers = [x for x in chain(fronts, backs) if x not in s]
min_good_number = min(potential_good_numbers, default=0)
```

In this step, `[2, 3, 7, 2, 8]` are potential "good" numbers after filtering out `1` and `4`.

4. Calculate the minimum of these remaining numbers:

The minimum "good" number here is `2`, which is the smallest number from the potential "good" numbers list.

Therefore, the smallest "good" number that can be achieved by flipping the cards such that it appears only on the backs and not on any front is `2`. This is the final solution for the provided example arrays.

Solution Implementation

Python

```
from itertools import chain

class Solution:
    def flipGame(self, fronts: List[int], backs: List[int]) -> int:
        # Create a set of numbers that appear on both sides of the same card
        same_on_both_sides = {number for front, back in zip(fronts, backs) if front == back}

        # Go through all numbers that appear on either front or back,
        # and select the smallest one that doesn't appear on both sides of the same card.
        # If there's no such card, return 0 as the default value.
        return min(
            (number for number in chain(fronts, backs) if number not in same_on_both_sides),
            default=0
        )
```

Java

```
class Solution {

    // The flipGame function returns the minimum number that is not on both sides of any card.
    public int flipGame(int[] fronts, int[] backs) {
        // Set for storing numbers that are the same on both sides of a card.
        Set<Integer> sameNumbers = new HashSet<>();
        int n = fronts.length;

        // Store all numbers that are the same on both sides into the set.
        for (int i = 0; i < n; ++i) {
            if (fronts[i] == backs[i]) {
                sameNumbers.add(fronts[i]);
            }
        }

        // The initial answer is set high to ensure any number lower than this will replace it.
        int answer = Integer.MAX_VALUE;

        // Check numbers on the front of the cards.
        for (int front : fronts) {
            if (!sameNumbers.contains(front)) {
                answer = Math.min(answer, front);
            }
        }

        // Check numbers on the back of the cards.
        for (int back : backs) {
            if (!sameNumbers.contains(back)) {
                answer = Math.min(answer, back);
            }
        }

        // If the answer has not been updated, return 0, otherwise return the answer.
        return answer == Integer.MAX_VALUE ? 0 : answer;
    }
}
```

C++

```
#include <vector>
#include <unordered_set>
#include <algorithm>
using namespace std;

class Solution {
public:
    // Method to find the smallest number that appears on one side of a card but not both.
    int flipGame(vector<int>& fronts, vector<int>& backs) {
        unordered_set<int> sameNumberCards; // Stores numbers that appear on both sides of the same card.
        int numCards = fronts.size(); // Store the number of cards.

        // Find all the numbers that are the same on both sides of a card and add them to the set.
        for (int i = 0; i < numCards; ++i) {
            if (fronts[i] == backs[i]) {
                sameNumberCards.insert(fronts[i]);
            }
        }

        int answer = INT_MAX; // Initialize the answer with a high value.

        // Check each number on the front. If it is not in the set, it's a candidate for the answer.
        for (int frontValue : fronts) {
            if (sameNumberCards.count(frontValue) == 0) {
                answer = min(answer, frontValue);
            }
        }

        // Check each number on the back. If it is not in the set, it's a candidate for the answer.
        for (int backValue : backs) {
            if (sameNumberCards.count(backValue) == 0) {
                answer = min(answer, backValue);
            }
        }

        // If the answer remains INT_MAX, all numbers appear on both sides, so return 0.
        // Otherwise, return the answer, ensuring that if it's equal to 9999 it should return 0.
        return answer == INT_MAX ? 0 : answer % 9999;
    }
};
```

TypeScript

```
function flipGame(fronts: number[], backs: number[]): number {
    // Create a set to store numbers that are the same on both the front and back of cards.
    const identicalNumbers: Set<number> = new Set();
    const totalCards = fronts.length;

    // Identify cards with the same number on both sides and add to the set.
    for (let i = 0; i < totalCards; ++i) {
        if (fronts[i] === backs[i]) {
            identicalNumbers.add(fronts[i]);
        }
    }

    let minNumber = Number.MAX_SAFE_INTEGER; // Initialize the minimum number to a very high value.

    // Iterate through the front of cards to find the minimum number that isn't the same on both sides.
    for (const number of fronts) {
        if (!identicalNumbers.has(number)) {
            minNumber = Math.min(minNumber, number);
        }
    }

    // Iterate through the back of cards to find the minimum number that isn't the same on both sides.
    for (const number of backs) {
        if (!identicalNumbers.has(number)) {
            minNumber = Math.min(minNumber, number);
        }
    }

    // If the smallest number is less than Number.MAX_SAFE_INTEGER, return it. Otherwise, return 0.
    // This assumes there's no valid flip game number to be returned if the minimum hasn't changed.
    return minNumber < Number.MAX_SAFE_INTEGER ? minNumber : 0;
}
```

```
from itertools import chain

class Solution:
    def flipGame(self, fronts: List[int], backs: List[int]) -> int:
        # Create a set of numbers that appear on both sides of the same card
        same_on_both_sides = {number for front, back in zip(fronts, backs) if front == back}

        # Go through all numbers that appear on either front or back,
        # and select the smallest one that doesn't appear on both sides of the same card.
        # If there's no such card, return 0 as the default value.
        return min(
            (number for number in chain(fronts, backs) if number not in same_on_both_sides),
            default=0
        )
```

Time and Space Complexity

The time complexity of the provided code can be analyzed by examining each operation:

1. `zip(fronts, backs)`: This operation goes through the lists `fronts` and `backs` once, creating an iterable comprised of tuples containing corresponding elements from both lists. The time complexity of this operation is `O(N)`, where `N` is the number of cards.
2. `if a == b`: This is a comparison operation inside the set comprehension that is applied to each of the `N` elements, and hence does not add more than a constant factor to the overall time complexity.
3. `{a for a, b in zip(fronts, backs) if a == b}`: Creating a set of elements where cards show the same number on both sides requires iterating over all `N` pairs and inserting into a set, which is `O(1)` average time complexity for each insertion. Hence, the complexity for this part is `O(N)`.
4. `chain(fronts, backs)`: The `chain` function from the `itertools` module is used to iterate over both `fronts` and `backs` without creating a new list in memory. This operation itself is `O(1)`, but iterating over it will be `O(2N)` since the `fronts` and `backs` are both of size `N`.
5. `min((x for x in chain(fronts, backs) if x not in s), default=0)`: Here, we generate an iterator that filters out all numbers present in set `s` from the chained list of `fronts` and `backs`. The `in` operation in a set is `O(1)`, so this filter operation is `O(2N)`. Finding the minimum value is also `O(2N)` because, in the worst case, it has to inspect each element of the combined `fronts` and `backs`.

Therefore, the overall time complexity is `O(2N)`, where `N` is the number of cards. However, we generally consider `O(2N)` as `O(N)` because constant factors are dropped in Big O notation.

The space complexity can be analyzed similarly:

1. Set `s`: In the worst case, if all cards have the same number on both sides, the set will store every card number, yielding a space complexity of `O(N)`.
2. The `chain` operation doesn't consume additional space since it's only combining two iterators, so it does not contribute to space complexity.

Consequently, the space complexity of the algorithm is `O(N)`.

In summary, the time complexity is `O(N)` and the space complexity is `O(N)`.