Problem 822: Card Flipping Game

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You are given two

0-indexed

integer arrays

fronts

and

backs

of length

n

, where the

i

th

card has the positive integer

fronts[i]

printed on the front and

backs[i]

printed on the back. Initially, each card is placed on a table such that the front number is facing up and the other is facing down. You may flip over any number of cards (possibly zero).

After flipping the cards, an integer is considered

good

if it is facing down on some card and

not

facing up on any card.

Return

the minimum possible good integer after flipping the cards

. If there are no good integers, return

0

.

Example 1:

Input:

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

Output:

2

Explanation:

If we flip the second card, the face up numbers are [1,3,4,4,7] and the face down are [1,2,4,1,3]. 2 is the minimum good integer as it appears facing down but not facing up. It can be shown that 2 is the minimum possible good integer obtainable after flipping some cards.

Example 2:

Input:

fronts = [1], backs = [1]

Output:

0

Explanation:

There are no good integers no matter how we flip the cards, so we return 0.

Constraints:

n == fronts.length == backs.length

1 <= n <= 1000

1 <= fronts[i], backs[i] <= 2000

Code Snippets

C++:

```
class Solution {
public:
int flipgame(vector<int>& fronts, vector<int>& backs) {

}
};
```

Java:

```
class Solution {
public int flipgame(int[] fronts, int[] backs) {

}
}
```

Python3:

```
class Solution:
def flipgame(self, fronts: List[int], backs: List[int]) -> int:
```

Python:

```
class Solution(object):
def flipgame(self, fronts, backs):
"""
:type fronts: List[int]
:type backs: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} fronts
* @param {number[]} backs
* @return {number}
*/
var flipgame = function(fronts, backs) {

};
```

TypeScript:

```
function flipgame(fronts: number[], backs: number[]): number {

};
```

C#:

```
public class Solution {
public int Flipgame(int[] fronts, int[] backs) {
```

```
}
}
```

C:

```
int flipgame(int* fronts, int frontsSize, int* backs, int backsSize) {

}
```

Go:

```
func flipgame(fronts []int, backs []int) int {

}
```

Kotlin:

```
class Solution {
fun flipgame(fronts: IntArray, backs: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func flipgame(_ fronts: [Int], _ backs: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn flipgame(fronts: Vec<i32>, backs: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} fronts
# @param {Integer[]} backs
# @return {Integer}
def flipgame(fronts, backs)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $fronts
* @param Integer[] $backs
* @return Integer
*/
function flipgame($fronts, $backs) {

}
}
```

Dart:

```
class Solution {
int flipgame(List<int> fronts, List<int> backs) {

}
}
```

Scala:

```
object Solution {
def flipgame(fronts: Array[Int], backs: Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec flipgame(fronts :: [integer], backs :: [integer]) :: integer
def flipgame(fronts, backs) do
```

```
end
end
```

Erlang:

```
-spec flipgame(Fronts :: [integer()], Backs :: [integer()]) -> integer().
flipgame(Fronts, Backs) ->
.
```

Racket:

```
(define/contract (flipgame fronts backs)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Card Flipping Game
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public:
int flipgame(vector<int>& fronts, vector<int>& backs) {

}
};
```

Java Solution:

```
/**
* Problem: Card Flipping Game
```

```
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public int flipgame(int[] fronts, int[] backs) {

}
}
```

Python3 Solution:

```
"""
Problem: Card Flipping Game
Difficulty: Medium
Tags: array, hash

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class Solution:
def flipgame(self, fronts: List[int], backs: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def flipgame(self, fronts, backs):
"""
:type fronts: List[int]
:type backs: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Card Flipping Game
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {number[]} fronts
* @param {number[]} backs
* @return {number}
*/
var flipgame = function(fronts, backs) {

};
```

TypeScript Solution:

```
/**
* Problem: Card Flipping Game
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

function flipgame(fronts: number[], backs: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Card Flipping Game
* Difficulty: Medium
* Tags: array, hash
```

```
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

public class Solution {
public int Flipgame(int[] fronts, int[] backs) {

}
}
```

C Solution:

```
/*
* Problem: Card Flipping Game
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

int flipgame(int* fronts, int frontsSize, int* backs, int backsSize) {

}
```

Go Solution:

```
// Problem: Card Flipping Game
// Difficulty: Medium
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

func flipgame(fronts []int, backs []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun flipgame(fronts: IntArray, backs: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func flipgame(_ fronts: [Int], _ backs: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Card Flipping Game
// Difficulty: Medium
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

impl Solution {
pub fn flipgame(fronts: Vec<i32>, backs: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} fronts
# @param {Integer[]} backs
# @return {Integer}
def flipgame(fronts, backs)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $fronts
* @param Integer[] $backs
* @return Integer
*/
function flipgame($fronts, $backs) {

}
}
```

Dart Solution:

```
class Solution {
int flipgame(List<int> fronts, List<int> backs) {

}
}
```

Scala Solution:

```
object Solution {
def flipgame(fronts: Array[Int], backs: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec flipgame(fronts :: [integer], backs :: [integer]) :: integer
def flipgame(fronts, backs) do

end
end
```

Erlang Solution:

```
-spec flipgame(Fronts :: [integer()], Backs :: [integer()]) -> integer().
flipgame(Fronts, Backs) ->
.
```

Racket Solution:

```
(define/contract (flipgame fronts backs)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```