Problem 2143: Choose Numbers From Two Arrays in Range

Problem Information

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

Problem Description

You are given two

0-indexed

integer arrays

nums1

and

nums2

of length

n

.

A range

[l, r]

(

inclusive

) where

0 <= l <= r < n

is

balanced

if:

For every

i

in the range

[l, r]

, you pick either

nums1[i]

or

nums2[i]

.

The sum of the numbers you pick from

nums1

equals to the sum of the numbers you pick from

nums2

(the sum is considered to be

0

if you pick no numbers from an array).

Two

balanced

ranges from

[l

1

, r

1

]

and

[l

2

, r

2

]

are considered to be

different

if at least one of the following is true:

l

1

!= l

2

r

1

!= r

2

nums1[i]

is picked in the first range, and

nums2[i]

is picked in the second range or

vice versa

for at least one

i

.

Return

the number of

different

ranges that are balanced.

Since the answer may be very large, return it

modulo

10

9

+ 7

.

Example 1:

Input:

nums1 = [1,2,5], nums2 = [2,6,3]

Output:

3

Explanation:

The balanced ranges are: - [0, 1] where we choose nums2[0], and nums1[1]. The sum of the numbers chosen from nums1 equals the sum of the numbers chosen from nums2: 2 = 2. - [0, 2] where we choose nums1[0], nums2[1], and nums1[2]. The sum of the numbers chosen from nums1 equals the sum of the numbers chosen from nums2: 1 + 5 = 6. - [0, 2] where we choose nums1[0], nums1[1], and nums2[2]. The sum of the numbers chosen from nums1 equals the sum of the numbers chosen from nums2: 1 + 2 = 3. Note that the second and third balanced ranges are different. In the second balanced range, we choose nums2[1] and in the third balanced range, we choose nums1[1].

Example 2:

Input:

nums1 = [0,1], nums2 = [1,0]

Output:

4

Explanation:

The balanced ranges are: - [0, 0] where we choose nums1[0]. The sum of the numbers chosen from nums1 equals the sum of the numbers chosen from nums2: 0 = 0. - [1, 1] where we choose nums2[1]. The sum of the numbers chosen from nums1 equals the sum of the numbers chosen from nums2: 0 = 0. - [0, 1] where we choose nums1[0] and nums2[1]. The sum of the numbers chosen from nums1 equals the sum of the numbers chosen from nums2: 0 = 0. - [0, 1] where we choose nums2[0] and nums1[1]. The sum of the numbers chosen from nums1 equals the sum of the numbers chosen from nums2: 1 = 1.

Constraints:

n == nums1.length == nums2.length

1 <= n <= 100

0 <= nums1[i], nums2[i] <= 100

Code Snippets

C++:

```
class Solution {
public:
int countSubranges(vector<int>& nums1, vector<int>& nums2) {

}
};
```

Java:

```
class Solution {
public int countSubranges(int[] nums1, int[] nums2) {

}
}
```

Python3:

```
class Solution:
def countSubranges(self, nums1: List[int], nums2: List[int]) -> int:
```

Python:

```
class Solution(object):
def countSubranges(self, nums1, nums2):
"""
:type nums1: List[int]
:type nums2: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} nums1
* @param {number[]} nums2
* @return {number}
*/
var countSubranges = function(nums1, nums2) {

};
```

TypeScript:

```
function countSubranges(nums1: number[], nums2: number[]): number {

};
```

C#:

```
public class Solution {
public int CountSubranges(int[] nums1, int[] nums2) {

}
}
```

C:

```
int countSubranges(int* nums1, int nums1Size, int* nums2, int nums2Size) {

}
```

Go:

```
func countSubranges(nums1 []int, nums2 []int) int {

}
```

Kotlin:

```
class Solution {
fun countSubranges(nums1: IntArray, nums2: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func countSubranges(_ nums1: [Int], _ nums2: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn count_subranges(nums1: Vec<i32>, nums2: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} nums1
# @param {Integer[]} nums2
# @return {Integer}
def count_subranges(nums1, nums2)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums1
* @param Integer[] $nums2
* @return Integer
*/
function countSubranges($nums1, $nums2) {

}
}
```

Dart:

```
class Solution {
int countSubranges(List<int> nums1, List<int> nums2) {

}
}
```

Scala:

```
object Solution {
def countSubranges(nums1: Array[Int], nums2: Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec count_subranges(nums1 :: [integer], nums2 :: [integer]) :: integer
def count_subranges(nums1, nums2) do

end
end
```

Erlang:

```
-spec count_subranges(Nums1 :: [integer()], Nums2 :: [integer()]) ->
integer().
```

```
count_subranges(Nums1, Nums2) ->
.
```

Racket:

```
(define/contract (count-subranges nums1 nums2)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Choose Numbers From Two Arrays in Range
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
int countSubranges(vector<int>& nums1, vector<int>& nums2) {

}
};
```

Java Solution:

```
/**
* Problem: Choose Numbers From Two Arrays in Range
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
```

```
*/

class Solution {
public int countSubranges(int[] nums1, int[] nums2) {

}
}
```

Python3 Solution:

```
"""
Problem: Choose Numbers From Two Arrays in Range
Difficulty: Hard
Tags: array, dp

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def countSubranges(self, nums1: List[int], nums2: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def countSubranges(self, nums1, nums2):
"""
:type nums1: List[int]
:type nums2: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Choose Numbers From Two Arrays in Range
* Difficulty: Hard
* Tags: array, dp
```

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

/**
* @param {number[]} nums1
* @param {number[]} nums2
* @return {number}
*/
var countSubranges = function(nums1, nums2) {

};
```

TypeScript Solution:

```
/**
* Problem: Choose Numbers From Two Arrays in Range
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function countSubranges(nums1: number[], nums2: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Choose Numbers From Two Arrays in Range
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
```

```
*/

public class Solution {
public int CountSubranges(int[] nums1, int[] nums2) {

}
}
```

C Solution:

```
/*
* Problem: Choose Numbers From Two Arrays in Range
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

int countSubranges(int* nums1, int nums1Size, int* nums2, int nums2Size) {

}
```

Go Solution:

```
// Problem: Choose Numbers From Two Arrays in Range
// Difficulty: Hard
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func countSubranges(nums1 []int, nums2 []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun countSubranges(nums1: IntArray, nums2: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func countSubranges(_ nums1: [Int], _ nums2: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Choose Numbers From Two Arrays in Range
// Difficulty: Hard
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn count_subranges(nums1: Vec<i32>, nums2: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums1
# @param {Integer[]} nums2
# @return {Integer}
def count_subranges(nums1, nums2)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums1
* @param Integer[] $nums2
* @return Integer
*/
function countSubranges($nums1, $nums2) {

}
}
```

Dart Solution:

```
class Solution {
int countSubranges(List<int> nums1, List<int> nums2) {

}
}
```

Scala Solution:

```
object Solution {
def countSubranges(nums1: Array[Int], nums2: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec count_subranges(nums1 :: [integer], nums2 :: [integer]) :: integer
def count_subranges(nums1, nums2) do

end
end
```

Erlang Solution:

```
-spec count_subranges(Nums1 :: [integer()], Nums2 :: [integer()]) ->
integer().
count_subranges(Nums1, Nums2) ->
```

```
.
```

Racket Solution:

```
(define/contract (count-subranges nums1 nums2)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```