Problem 3269: Constructing Two Increasing Arrays

Problem Information

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

Problem Description

Given 2 integer arrays

nums1

and

nums2

consisting only of 0 and 1, your task is to calculate the

minimum

possible

largest

number in arrays

nums1

and

nums2

, after doing the following.

Replace every 0 with an

even positive integer

and every 1 with an

odd positive integer

. After replacement, both arrays should be

increasing

and each integer should be used

at most

once.

Return the

minimum possible largest number

after applying the changes.

Example 1:

Input:

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

Output:

5

Explanation:

After replacing,

nums1 = []

, and

nums2 = [1, 2, 3, 5]

.

Example 2:

Input:

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

Output:

9

Explanation:

One way to replace, having 9 as the largest element is

nums1 = [2, 3, 8, 9]

, and

nums2 = [1, 4, 6, 7]

.

Example 3:

Input:

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

Output:

13

Explanation:

One way to replace, having 13 as the largest element is

nums1 = [2, 3, 4, 6, 7]

, and

nums2 = [8, 10, 12, 13]

.

Constraints:

0 <= nums1.length <= 1000

1 <= nums2.length <= 1000

nums1

and

nums2

consist only of 0 and 1.

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

```

end
end
```

Erlang:

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

Racket:

```
(define/contract (min-largest nums1 nums2)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Constructing Two Increasing Arrays
* 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 minLargest(vector<int>& nums1, vector<int>& nums2) {

}
};
```

Java Solution:

```
/**
* Problem: Constructing Two Increasing Arrays
* 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 minLargest(int[] nums1, int[] nums2) {

}
}
```

Python3 Solution:

```
"""
Problem: Constructing Two Increasing Arrays
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 minLargest(self, nums1: List[int], nums2: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Constructing Two Increasing Arrays
* 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 minLargest = function(nums1, nums2) {

};
```

TypeScript Solution:

```
/**
* Problem: Constructing Two Increasing Arrays
* 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 minLargest(nums1: number[], nums2: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Constructing Two Increasing Arrays
* 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 MinLargest(int[] nums1, int[] nums2) {

}
}
```

C Solution:

```
/*
* Problem: Constructing Two Increasing Arrays
* 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 minLargest(int* nums1, int nums1Size, int* nums2, int nums2Size) {

}
```

Go Solution:

```
// Problem: Constructing Two Increasing Arrays
// 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 minLargest(nums1 []int, nums2 []int) int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Constructing Two Increasing Arrays
// 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 min_largest(nums1: Vec<i32>, nums2: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (min-largest nums1 nums2)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```