Problem 565: Array Nesting

Problem Information

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

Problem Description

You are given an integer array

nums

of length

n

where

nums

is a permutation of the numbers in the range

[0, n - 1]

.

You should build a set

s[k] = {nums[k], nums[nums[k]], nums[nums[nums[k]]], ... }

subjected to the following rule:

The first element in

s[k]

starts with the selection of the element

nums[k]

of

index = k

.

The next element in

s[k]

should be

nums[nums[k]]

, and then

nums[nums[nums[k]]]

, and so on.

We stop adding right before a duplicate element occurs in

s[k]

.

Return

the longest length of a set

s[k]

.

Example 1:

Input:

nums = [5,4,0,3,1,6,2]

Output:

4

Explanation:

nums[0] = 5, nums[1] = 4, nums[2] = 0, nums[3] = 3, nums[4] = 1, nums[5] = 6, nums[6] = 2.
One of the longest sets s[k]: s[0] = {nums[0], nums[5], nums[6], nums[2]} = {5, 6, 2, 0}

Example 2:

Input:

nums = [0,1,2]

Output:

1

Constraints:

1 <= nums.length <= 10

5

0 <= nums[i] < nums.length

All the values of

nums

are

unique

.

Code Snippets

C++:

```
class Solution {
public:
int arrayNesting(vector<int>& nums) {

}
};
```

Java:

```
class Solution {
public int arrayNesting(int[] nums) {

}
}
```

Python3:

```
class Solution:
def arrayNesting(self, nums: List[int]) -> int:
```

Python:

```
class Solution(object):
def arrayNesting(self, nums):
"""
:type nums: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} nums
```

```
 * @return {number}
 */
var arrayNesting = function(nums) {

};
```

TypeScript:

```
function arrayNesting(nums: number[]): number {

};
```

C#:

```
public class Solution {
public int ArrayNesting(int[] nums) {

}
}
```

C:

```
int arrayNesting(int* nums, int numsSize) {

}
```

Go:

```
func arrayNesting(nums []int) int {

}
```

Kotlin:

```
class Solution {
fun arrayNesting(nums: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func arrayNesting(_ nums: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn array_nesting(nums: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} nums
# @return {Integer}
def array_nesting(nums)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @return Integer
*/
function arrayNesting($nums) {

}
}
```

Dart:

```
class Solution {
int arrayNesting(List<int> nums) {

}
}
```

Scala:

```
object Solution {
def arrayNesting(nums: Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec array_nesting(nums :: [integer]) :: integer
def array_nesting(nums) do

end
end
```

Erlang:

```
-spec array_nesting(Nums :: [integer()]) -> integer().
array_nesting(Nums) ->
.
```

Racket:

```
(define/contract (array-nesting nums)
(-> (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Array Nesting
* Difficulty: Medium
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
class Solution {
public:
int arrayNesting(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Array Nesting
* Difficulty: Medium
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public int arrayNesting(int[] nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Array Nesting
Difficulty: Medium
Tags: array, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def arrayNesting(self, nums: List[int]) -> int:
# TODO: Implement optimized solution
```

```
pass
```

Python Solution:

```
class Solution(object):
def arrayNesting(self, nums):
"""
:type nums: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Array Nesting
* Difficulty: Medium
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[]} nums
* @return {number}
*/
var arrayNesting = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Array Nesting
* Difficulty: Medium
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
```

```
*/

function arrayNesting(nums: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Array Nesting
* Difficulty: Medium
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int ArrayNesting(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Array Nesting
* Difficulty: Medium
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

int arrayNesting(int* nums, int numsSize) {

}
```

Go Solution:

```
// Problem: Array Nesting
// Difficulty: Medium
// Tags: array, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func arrayNesting(nums []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun arrayNesting(nums: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func arrayNesting(_ nums: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Array Nesting
// Difficulty: Medium
// Tags: array, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn array_nesting(nums: Vec<i32>) -> i32 {

}
```

```
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @return {Integer}
def array_nesting(nums)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @return Integer
*/
function arrayNesting($nums) {

}
}
```

Dart Solution:

```
class Solution {
int arrayNesting(List<int> nums) {

}
}
```

Scala Solution:

```
object Solution {
def arrayNesting(nums: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec array_nesting(nums :: [integer]) :: integer
def array_nesting(nums) do

end
end
```

Erlang Solution:

```
-spec array_nesting(Nums :: [integer()]) -> integer().
array_nesting(Nums) ->
.
```

Racket Solution:

```
(define/contract (array-nesting nums)
(-> (listof exact-integer?) exact-integer?)
)
```