Problem 2765: Longest Alternating Subarray

Problem Information

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

Problem Description

You are given a

0-indexed

integer array

nums

. A subarray

s

of length

m

is called

alternating

if:

m

is greater than

1

.

s

1

= s

0

+ 1

.

The

0-indexed

subarray

s

looks like

[s

0

, s

1

, s

0

, s

1

,...,s

(m-1) % 2

]

. In other words,

s

1

- s

0

= 1

,

s

2

- s

1

= -1

,

s

3

- s

2

= 1

,

s

4

- s

3

= -1

, and so on up to

s[m - 1] - s[m - 2] = (-1)

m

.

Return

the maximum length of all

alternating

subarrays present in

nums

or

-1

if no such subarray exists

.

A subarray is a contiguous

non-empty

sequence of elements within an array.

Example 1:

Input:

nums = [2,3,4,3,4]

Output:

4

Explanation:

The alternating subarrays are

[2, 3]

,

[3,4]

,

[3,4,3]

, and

[3,4,3,4]

. The longest of these is

[3,4,3,4]

, which is of length 4.

Example 2:

Input:

nums = [4,5,6]

Output:

2

Explanation:

[4,5]

and

[5,6]

are the only two alternating subarrays. They are both of length 2.

Constraints:

2 <= nums.length <= 100

1 <= nums[i] <= 10

4

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

```
};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

```
}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (alternating-subarray nums)
(-> (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Longest Alternating Subarray
* Difficulty: Easy
* Tags: array
*
* 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 alternatingSubarray(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Longest Alternating Subarray
* Difficulty: Easy
* Tags: array
*
* 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 alternatingSubarray(int[] nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Longest Alternating Subarray
Difficulty: Easy
Tags: array

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 alternatingSubarray(self, nums: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Longest Alternating Subarray
* Difficulty: Easy
* Tags: array
*
* 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 alternatingSubarray = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Longest Alternating Subarray
* Difficulty: Easy
* Tags: array
*
* 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 alternatingSubarray(nums: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Longest Alternating Subarray
* Difficulty: Easy
* Tags: array
*
```

```
* 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 AlternatingSubarray(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Longest Alternating Subarray
* Difficulty: Easy
* Tags: array
*
* 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 alternatingSubarray(int* nums, int numsSize) {

}
```

Go Solution:

```
// Problem: Longest Alternating Subarray
// Difficulty: Easy
// Tags: array
//
// 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 alternatingSubarray(nums []int) int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Longest Alternating Subarray
// Difficulty: Easy
// Tags: array
//
// 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 alternating_subarray(nums: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (alternating-subarray nums)
(-> (listof exact-integer?) exact-integer?)
)
```