Problem 3350: Adjacent Increasing Subarrays Detection II

Problem Information

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

Problem Description

Given an array

nums

of

n

integers, your task is to find the

maximum

value of

k

for which there exist

two

adjacent

subarrays

of length

k

each, such that both subarrays are

strictly

increasing

. Specifically, check if there are

two

subarrays of length

k

starting at indices

a

and

b

(

a < b

), where:

Both subarrays

nums[a..a + k - 1]

and

nums[b..b + k - 1]

are

strictly increasing

.

The subarrays must be

adjacent

, meaning

b = a + k

.

Return the

maximum

possible

value of

k

.

A

subarray

is a contiguous

non-empty

sequence of elements within an array.

Example 1:

Input:

nums = [2,5,7,8,9,2,3,4,3,1]

Output:

3

Explanation:

The subarray starting at index 2 is

[7, 8, 9]

, which is strictly increasing.

The subarray starting at index 5 is

[2, 3, 4]

, which is also strictly increasing.

These two subarrays are adjacent, and 3 is the

maximum

possible value of

k

for which two such adjacent strictly increasing subarrays exist.

Example 2:

Input:

nums = [1,2,3,4,4,4,4,5,6,7]

Output:

2

Explanation:

The subarray starting at index 0 is

[1, 2]

, which is strictly increasing.

The subarray starting at index 2 is

[3, 4]

, which is also strictly increasing.

These two subarrays are adjacent, and 2 is the

maximum

possible value of

k

for which two such adjacent strictly increasing subarrays exist.

Constraints:

2 <= nums.length <= 2 * 10

5

-10

9

<= nums[i] <= 10

9

Code Snippets

C++:

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

}
};
```

Java:

```
class Solution {
public int maxIncreasingSubarrays(List<Integer> nums) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

```
var maxIncreasingSubarrays = function(nums) {

};
```

TypeScript:

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

};
```

C#:

```
public class Solution {
public int MaxIncreasingSubarrays(IList<int> nums) {

}
}
```

C:

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

}
```

Go:

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

}
```

Kotlin:

```
class Solution {
fun maxIncreasingSubarrays(nums: List<Int>): Int {

}
}
```

Swift:

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

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

```
object Solution {
def maxIncreasingSubarrays(nums: List[Int]): Int = {

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (max-increasing-subarrays nums)
(-> (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Adjacent Increasing Subarrays Detection II
* 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 maxIncreasingSubarrays(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Adjacent Increasing Subarrays Detection II
* 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 maxIncreasingSubarrays(List<Integer> nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Adjacent Increasing Subarrays Detection II
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 maxIncreasingSubarrays(self, nums: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Adjacent Increasing Subarrays Detection II
* 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 maxIncreasingSubarrays = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Adjacent Increasing Subarrays Detection II
* 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 maxIncreasingSubarrays(nums: number[]): number {
};
```

C# Solution:

```
/*
* Problem: Adjacent Increasing Subarrays Detection II
* 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 MaxIncreasingSubarrays(IList<int> nums) {

}
}
```

C Solution:

```
/*
* Problem: Adjacent Increasing Subarrays Detection II
* 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 maxIncreasingSubarrays(int* nums, int numsSize) {

}
```

Go Solution:

```
// Problem: Adjacent Increasing Subarrays Detection II
// 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 maxIncreasingSubarrays(nums []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun maxIncreasingSubarrays(nums: List<Int>): Int {

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Adjacent Increasing Subarrays Detection II
// 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 max_increasing_subarrays(nums: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

```
object Solution {
def maxIncreasingSubarrays(nums: List[Int]): Int = {

}
}
```

Elixir Solution:

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

```



end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (max-increasing-subarrays nums)
(-> (listof exact-integer?) exact-integer?)
)
```