Problem 2919: Minimum Increment Operations to Make Array Beautiful

Problem Information

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

Problem Description

You are given a

0-indexed

integer array

nums

having length

n

, and an integer

k

.

You can perform the following

increment

operation

any

number of times (

including zero

):

Choose an index

i

in the range

[0, n - 1]

, and increase

nums[i]

by

1

.

An array is considered

beautiful

if, for any

subarray

with a size of

3

or

more

, its

maximum

element is

greater than or equal

to

k

.

Return

an integer denoting the

minimum

number of increment operations needed to make

nums

beautiful

.

A subarray is a contiguous

non-empty

sequence of elements within an array.

Example 1:

Input:

nums = [2,3,0,0,2], k = 4

Output:

3

Explanation:

We can perform the following increment operations to make nums beautiful: Choose index i = 1 and increase nums[1] by 1 -> [2,4,0,0,2]. Choose index i = 4 and increase nums[4] by 1 -> [2,4,0,0,3]. Choose index i = 4 and increase nums[4] by 1 -> [2,4,0,0,4]. The subarrays with a size of 3 or more are: [2,4,0], [4,0,0], [0,0,4], [2,4,0,0], [4,0,0,4], [2,4,0,0,4]. In all the subarrays, the maximum element is equal to k = 4, so nums is now beautiful. It can be shown that nums cannot be made beautiful with fewer than 3 increment operations. Hence, the answer is 3.

Example 2:

Input:

nums = [0,1,3,3], k = 5

Output:

2

Explanation:

We can perform the following increment operations to make nums beautiful: Choose index i = 2 and increase nums[2] by 1 -> [0,1,4,3]. Choose index i = 2 and increase nums[2] by 1 -> [0,1,5,3]. The subarrays with a size of 3 or more are: [0,1,5], [1,5,3], [0,1,5,3]. In all the subarrays, the maximum element is equal to k = 5, so nums is now beautiful. It can be shown that nums cannot be made beautiful with fewer than 2 increment operations. Hence, the answer is 2.

Example 3:

Input:

nums = [1,1,2], k = 1

Output:

0

Explanation:

The only subarray with a size of 3 or more in this example is [1,1,2]. The maximum element, 2, is already greater than k = 1, so we don't need any increment operation. Hence, the answer is 0.

Constraints:

3 <= n == nums.length <= 10

5

0 <= nums[i] <= 10

9

0 <= k <= 10

9

Code Snippets

C++:

```
class Solution {
public:
long long minIncrementOperations(vector<int>& nums, int k) {

}
};
```

Java:

```
class Solution {
public long minIncrementOperations(int[] nums, int k) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function minIncrementOperations(nums: number[], k: number): number {

};
```

C#:

```
public class Solution {
public long MinIncrementOperations(int[] nums, int k) {
```

```
}
}
```

C:

```
long long minIncrementOperations(int* nums, int numsSize, int k) {

}
```

Go:

```
func minIncrementOperations(nums []int, k int) int64 {

}
```

Kotlin:

```
class Solution {
fun minIncrementOperations(nums: IntArray, k: Int): Long {

}
}
```

Swift:

```
class Solution {
func minIncrementOperations(_ nums: [Int], _ k: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn min_increment_operations(nums: Vec<i32>, k: i32) -> i64 {

}
}
```

Ruby:

```
# @param {Integer[]} nums
# @param {Integer} k
# @return {Integer}
def min_increment_operations(nums, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer $k
* @return Integer
*/
function minIncrementOperations($nums, $k) {

}
}
```

Dart:

```
class Solution {
int minIncrementOperations(List<int> nums, int k) {

}
}
```

Scala:

```
object Solution {
def minIncrementOperations(nums: Array[Int], k: Int): Long = {

}
}
```

Elixir:

```
defmodule Solution do
@spec min_increment_operations(nums :: [integer], k :: integer) :: integer
def min_increment_operations(nums, k) do
```

```
end
end
```

Erlang:

```
-spec min_increment_operations(Nums :: [integer()], K :: integer()) ->
integer().
min_increment_operations(Nums, K) ->
.
```

Racket:

```
(define/contract (min-increment-operations nums k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Increment Operations to Make Array Beautiful
* Difficulty: Medium
* 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:
long long minIncrementOperations(vector<int>& nums, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Minimum Increment Operations to Make Array Beautiful
* Difficulty: Medium
* 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 long minIncrementOperations(int[] nums, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Increment Operations to Make Array Beautiful
Difficulty: Medium
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 minIncrementOperations(self, nums: List[int], k: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Minimum Increment Operations to Make Array Beautiful
* Difficulty: Medium
* 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[]} nums
* @param {number} k
* @return {number}
*/
var minIncrementOperations = function(nums, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Increment Operations to Make Array Beautiful
* Difficulty: Medium
* 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 minIncrementOperations(nums: number[], k: number): number {

};
```

C# Solution:

```
/*
* Problem: Minimum Increment Operations to Make Array Beautiful
* Difficulty: Medium
```

```
* 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 long MinIncrementOperations(int[] nums, int k) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Increment Operations to Make Array Beautiful
* Difficulty: Medium
* 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
*/

long long minIncrementOperations(int* nums, int numsSize, int k) {

}
```

Go Solution:

```
// Problem: Minimum Increment Operations to Make Array Beautiful
// Difficulty: Medium
// 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 minIncrementOperations(nums []int, k int) int64 {
```

```
}
```

Kotlin Solution:

```
class Solution {
fun minIncrementOperations(nums: IntArray, k: Int): Long {

}
}
```

Swift Solution:

```
class Solution {
func minIncrementOperations(_ nums: [Int], _ k: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Minimum Increment Operations to Make Array Beautiful
// Difficulty: Medium
// 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_increment_operations(nums: Vec<i32>, k: i32) -> i64 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @param {Integer} k
# @return {Integer}
def min_increment_operations(nums, k)
```

```
end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer $k
* @return Integer
*/
function minIncrementOperations($nums, $k) {

}
}
```

Dart Solution:

```
class Solution {
int minIncrementOperations(List<int> nums, int k) {

}
}
```

Scala Solution:

```
object Solution {
def minIncrementOperations(nums: Array[Int], k: Int): Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec min_increment_operations(nums :: [integer], k :: integer) :: integer
def min_increment_operations(nums, k) do

end
end
```

Erlang Solution:

```
-spec min_increment_operations(Nums :: [integer()], K :: integer()) ->
integer().
min_increment_operations(Nums, K) ->
.
```

Racket Solution:

```
(define/contract (min-increment-operations nums k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```