Problem 2653: Sliding Subarray Beauty

Problem Information

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

Problem Description

Given an integer array

nums

containing

n

integers, find the

beauty

of each subarray of size

k

.

The

beauty

of a subarray is the

x

th

smallest integer

in the subarray if it is

negative

, or

0

if there are fewer than

x

negative integers.

Return

an integer array containing

n - k + 1

integers, which denote the

beauty

of the subarrays

in order

from the first index in the array.

A subarray is a contiguous

non-empty

sequence of elements within an array.

Example 1:

Input:

nums = [1,-1,-3,-2,3], k = 3, x = 2

Output:

[-1,-2,-2]

Explanation:

There are 3 subarrays with size k = 3. The first subarray is

[1, -1, -3]

and the 2

nd

smallest negative integer is -1. The second subarray is

[-1, -3, -2]

and the 2

nd

smallest negative integer is -2. The third subarray is

[-3, -2, 3]

and the 2

nd

smallest negative integer is -2.

Example 2:

Input:

nums = [-1,-2,-3,-4,-5], k = 2, x = 2

Output:

[-1,-2,-3,-4]

Explanation:

There are 4 subarrays with size k = 2. For

[-1, -2]

, the 2

nd

smallest negative integer is -1. For

[-2, -3]

, the 2

nd

smallest negative integer is -2. For

[-3, -4]

, the 2

nd

smallest negative integer is -3. For

[-4, -5]

, the 2

nd

smallest negative integer is -4.

Example 3:

Input:

nums = [-3,1,2,-3,0,-3], k = 2, x = 1

Output:

[-3,0,-3,-3,-3]

Explanation:

There are 5 subarrays with size k = 2

.

For

[-3, 1]

, the 1

st

smallest negative integer is -3. For

[1, 2]

, there is no negative integer so the beauty is 0. For

[2, -3]

, the 1

st

smallest negative integer is -3. For

[-3, 0]

, the 1

st

smallest negative integer is -3. For

[0, -3]

, the 1

st

smallest negative integer is -3.

Constraints:

n == nums.length

1 <= n <= 10

5

1 <= k <= n

1 <= x <= k

-50 <= nums[i] <= 50

Code Snippets

C++:

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

}
};
```

Java:

```
class Solution {
public int[] getSubarrayBeauty(int[] nums, int k, int x) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

```
};
```

TypeScript:

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

};
```

C#:

```
public class Solution {
public int[] GetSubarrayBeauty(int[] nums, int k, int x) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* getSubarrayBeauty(int* nums, int numsSize, int k, int x, int*
returnSize) {

}
```

Go:

```
func getSubarrayBeauty(nums []int, k int, x int) []int {

}
```

Kotlin:

```
class Solution {
fun getSubarrayBeauty(nums: IntArray, k: Int, x: Int): IntArray {

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

```
}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (get-subarray-beauty nums k x)
(-> (listof exact-integer?) exact-integer? exact-integer? (listof
exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Sliding Subarray Beauty
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public:
vector<int> getSubarrayBeauty(vector<int>& nums, int k, int x) {

}
};
```

Java Solution:

```
/**
* Problem: Sliding Subarray Beauty
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public int[] getSubarrayBeauty(int[] nums, int k, int x) {

}
}
```

Python3 Solution:

```
"""
Problem: Sliding Subarray Beauty
Difficulty: Medium
Tags: array, hash
```

```
Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Sliding Subarray Beauty
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

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

```
};
```

TypeScript Solution:

```
/**
* Problem: Sliding Subarray Beauty
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

function getSubarrayBeauty(nums: number[], k: number, x: number): number[] {

};
```

C# Solution:

```
/*
* Problem: Sliding Subarray Beauty
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

public class Solution {
public int[] GetSubarrayBeauty(int[] nums, int k, int x) {

}
}
```

C Solution:

```
/*
* Problem: Sliding Subarray Beauty
* Difficulty: Medium
```

```
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* getSubarrayBeauty(int* nums, int numsSize, int k, int x, int*
returnSize) {

}
```

Go Solution:

```
// Problem: Sliding Subarray Beauty
// Difficulty: Medium
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

func getSubarrayBeauty(nums []int, k int, x int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun getSubarrayBeauty(nums: IntArray, k: Int, x: Int): IntArray {

}
}
```

Swift Solution:

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

```
}
}
```

Rust Solution:

```
// Problem: Sliding Subarray Beauty
// Difficulty: Medium
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

impl Solution {
pub fn get_subarray_beauty(nums: Vec<i32>, k: i32, x: i32) -> Vec<i32> {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

```
}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (get-subarray-beauty nums k x)
(-> (listof exact-integer?) exact-integer? exact-integer? (listof
```

```
exact-integer?))
)
```