

Problem 2104: Sum of Subarray Ranges

Problem Information

Difficulty: Medium

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

You are given an integer array

nums

. The

range

of a subarray of

nums

is the difference between the largest and smallest element in the subarray.

Return

the

sum of all

subarray ranges of

nums

.

A subarray is a contiguous non-empty sequence of elements within an array.

Example 1:

Input:

nums = [1,2,3]

Output:

4

Explanation:

The 6 subarrays of nums are the following: [1], range = largest - smallest = 1 - 1 = 0 [2], range = 2 - 2 = 0 [3], range = 3 - 3 = 0 [1,2], range = 2 - 1 = 1 [2,3], range = 3 - 2 = 1 [1,2,3], range = 3 - 1 = 2 So the sum of all ranges is $0 + 0 + 0 + 1 + 1 + 2 = 4$.

Example 2:

Input:

nums = [1,3,3]

Output:

4

Explanation:

The 6 subarrays of nums are the following: [1], range = largest - smallest = 1 - 1 = 0 [3], range = 3 - 3 = 0 [3], range = 3 - 3 = 0 [1,3], range = 3 - 1 = 2 [3,3], range = 3 - 3 = 0 [1,3,3], range = 3 - 1 = 2 So the sum of all ranges is $0 + 0 + 0 + 2 + 0 + 2 = 4$.

Example 3:

Input:

```
nums = [4,-2,-3,4,1]
```

Output:

```
59
```

Explanation:

The sum of all subarray ranges of nums is 59.

Constraints:

```
1 <= nums.length <= 1000
```

```
-10
```

```
9
```

```
<= nums[i] <= 10
```

```
9
```

Follow-up:

Could you find a solution with

$O(n)$

time complexity?

Code Snippets

C++:

```
class Solution {  
public:
```

```
long long subArrayRanges(vector<int>& nums) {  
}  
};
```

Java:

```
class Solution {  
    public long subArrayRanges(int[] nums) {  
        }  
    }
```

Python3:

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

Python:

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

JavaScript:

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

TypeScript:

```
function subArrayRanges(nums: number[]): number {  
};
```

C#:

```
public class Solution {  
    public long SubArrayRanges(int[] nums) {  
  
    }  
}
```

C:

```
long long subArrayRanges(int* nums, int numsSize) {  
  
}
```

Go:

```
func subArrayRanges(nums []int) int64 {  
  
}
```

Kotlin:

```
class Solution {  
    fun subArrayRanges(nums: IntArray): Long {  
  
    }  
}
```

Swift:

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

Rust:

```
impl Solution {  
    pub fn sub_array_ranges(nums: Vec<i32>) -> i64 {  
  
    }  
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

    }
}
```

Dart:

```
class Solution {
  int subArrayRanges(List<int> nums) {
}
```

Scala:

```
object Solution {
  def subArrayRanges(nums: Array[Int]): Long = {
}
```

Elixir:

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

```
end  
end
```

Erlang:

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

Racket:

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

Solutions

C++ Solution:

```
/*  
 * Problem: Sum of Subarray Ranges  
 * Difficulty: Medium  
 * Tags: array, stack  
 *  
 * 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:  
    long long subArrayRanges(vector<int>& nums) {  
  
    }  
};
```

Java Solution:

```
/**  
 * Problem: Sum of Subarray Ranges
```

```

* Difficulty: Medium
* Tags: array, stack
*
* 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 long subArrayRanges(int[] nums) {
}
}

```

Python3 Solution:

```

"""
Problem: Sum of Subarray Ranges
Difficulty: Medium
Tags: array, stack

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

```

Python Solution:

```

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

```

JavaScript Solution:

```

/**
 * Problem: Sum of Subarray Ranges
 * Difficulty: Medium
 * Tags: array, stack
 *
 * 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 subArrayRanges = function(nums) {
};


```

TypeScript Solution:

```

/**
 * Problem: Sum of Subarray Ranges
 * Difficulty: Medium
 * Tags: array, stack
 *
 * 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 subArrayRanges(nums: number[]): number {

};


```

C# Solution:

```

/*
 * Problem: Sum of Subarray Ranges
 * Difficulty: Medium
 * Tags: array, stack
 *
 * 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 long SubArrayRanges(int[] nums) {

    }
}

```

C Solution:

```

/*
 * Problem: Sum of Subarray Ranges
 * Difficulty: Medium
 * Tags: array, stack
 *
 * 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
 */

long long subArrayRanges(int* nums, int numsSize) {

}

```

Go Solution:

```

// Problem: Sum of Subarray Ranges
// Difficulty: Medium
// Tags: array, stack
//
// 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 subArrayRanges(nums []int) int64 {
}

```

Kotlin Solution:

```
class Solution {  
    fun subArrayRanges(nums: IntArray): Long {  
        }  
        }  
}
```

Swift Solution:

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

Rust Solution:

```
// Problem: Sum of Subarray Ranges  
// Difficulty: Medium  
// Tags: array, stack  
//  
// 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 sub_array_ranges(nums: Vec<i32>) -> i64 {  
        }  
        }  
}
```

Ruby Solution:

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

PHP Solution:

```
class Solution {
```

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

}
```

Dart Solution:

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

}
```

Scala Solution:

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

}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

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