Problem 3028: Ant on the Boundary

Problem Information

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

Problem Description

An ant is on a boundary. It sometimes goes

left

and sometimes

right

.

You are given an array of

non-zero

integers

nums

. The ant starts reading

nums

from the first element of it to its end. At each step, it moves according to the value of the current element:

If

nums[i] < 0

, it moves

left

by

-nums[i]

units.

If

nums[i] > 0

, it moves

right

by

nums[i]

units.

Return

the number of times the ant

returns

to the boundary.

Notes:

There is an infinite space on both sides of the boundary.

We check whether the ant is on the boundary only after it has moved

|nums[i]|

units. In other words, if the ant crosses the boundary during its movement, it does not count.

Example 1:

Input:

nums = [2,3,-5]

Output:

1

Explanation:

After the first step, the ant is 2 steps to the right of the boundary

. After the second step, the ant is 5 steps to the right of the boundary

. After the third step, the ant is on the boundary. So the answer is 1.

Example 2:

Input:

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

Output:

0

Explanation:

After the first step, the ant is 3 steps to the right of the boundary

. After the second step, the ant is 5 steps to the right of the boundary

. After the third step, the ant is 2 steps to the right of the boundary

. After the fourth step, the ant is 2 steps to the left of the boundary

. The ant never returned to the boundary, so the answer is 0.

Constraints:

1 <= nums.length <= 100

-10 <= nums[i] <= 10

nums[i] != 0

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
```

```
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (return-to-boundary-count nums)
(-> (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Ant on the Boundary
* 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 returnToBoundaryCount(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Ant on the Boundary
* 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 returnToBoundaryCount(int[] nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Ant on the Boundary
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 returnToBoundaryCount(self, nums: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Ant on the Boundary
* 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 returnToBoundaryCount = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Ant on the Boundary
* 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 returnToBoundaryCount(nums: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Ant on the Boundary
* 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 ReturnToBoundaryCount(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Ant on the Boundary
* 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 returnToBoundaryCount(int* nums, int numsSize) {

}
```

Go Solution:

```
// Problem: Ant on the Boundary
// 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 returnToBoundaryCount(nums []int) int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Ant on the Boundary
// 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 return_to_boundary_count(nums: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (return-to-boundary-count nums)
(-> (listof exact-integer?) exact-integer?)
)
```