3028. Ant on the Boundary

Easy Array Prefix Sum Simulation

Problem Description

An ant moves along a boundary, making decisions at each step to move either left or right based on an array of non-zero integers. Each integer in the array `nums` represents a step the ant takes. If the integer is positive, the ant moves to the right; if the integer is negative, the ant moves to the left. The magnitude of the number indicates the number of units the ant moves in that direction.

The challenge is to determine how many times the ant returns to the original starting point or boundary after making each step as dictated by the `nums` array. The infinite space on both sides of the boundary suggests that the ant can move indefinitely without restrictions. The primary condition to note is the check for the ant's position at the boundary happens only after completing a full step. If the ant crosses the boundary during movement, it does not count as a return.

Intuition

The essential insight of the solution is recognizing that we do not need to track the ant's position throughout its entire journey. Instead, we only need to know when the ant's total displacement from the starting point is zero.

We arrive at the solution approach by considering that the ant's position after each step is the cumulative sum of the steps taken. If we calculate the prefix sum of the array `nums`, we can easily see where the ant's total displacement is zero. In other words, we are interested in the positions where the running sum, also known as the prefix sum, equals zero.

The use of the `accumulate` function from Python's `itertools` module allows us to generate the prefix sums efficiently. After obtaining the prefix sums, we merely count how many times the accumulated sum equals zero, which corresponds to the ant returning to the boundary.

Solution Approach

The solution takes advantage of a pattern known as "prefix sum," which is a common technique in array manipulation problems. A prefix sum is simply the sum of the elements up to a certain index in an array.

The algorithm is straightforward:

1. We use Python's `accumulate` function from the `itertools` module, which takes an iterable (in this case, our `nums` array) and returns accumulated sums.
2. When we call `accumulate(nums)`, we get an iterable of sums that represent the position of the ant at each step after moving based on the current `nums` element. If `nums` starts with `[2, -1, -1, 2]`, the prefix sums would be `[2, 1, 0, 2]`. Notice the `0` indicates that the ant came back to the boundary at the third step.
3. Since we need to count how many times the ant returns to the boundary, we simply count how many zeros there are in the array of prefix sums. This is done with a generator expression `sum(s == 0 for s in accumulate(nums))`.

By using `accumulate` for the prefix sum, we avoid manually iterating through the array and summing the elements, which could be prone to errors and is less efficient. The prefix sum approach cuts down the computational complexity since it computes all the required sums in a single pass through the array. The use of the generator expression with `sum` is a concise and efficient way to count the occurrences of zeros, exploiting Python's ability to handle iterators and generators elegantly.

Example Walkthrough

Consider an array `nums` with the following sequence of non-zero integers:

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

According to the problem, these numbers represent the ant's steps along a boundary:

1. 3 steps to the right
2. 2 steps to the left
3. 2 steps to the right
4. 3 steps to the left
5. 1 step to the right
6. 1 step to the left

We need to determine how many times the ant returns to the starting point after completing each step as dictated by these numbers. Let's follow the solution approach step-by-step:

1. We will generate the prefix sums for `nums` using `accumulate(nums)`. The prefix sum is the cumulative sum of the numbers up to the current index.

Here are the prefix sums calculated at each step:

- After the first step: `3`
- After the second step: `3 + (-2) = 1`
- After the third step: `1 + 2 = 3`
- After the fourth step: `3 + (-3) = 0` (ant returns to the boundary)
- After the fifth step: `0 + 1 = 1`
- After the sixth step: `1 + (-1) = 0` (ant returns to the boundary again)

So, the array of prefix sums is `[3, 1, 3, 0, 1, 0]`.

2. Now, we need to count the number of times this array has a `0`, which signifies that the ant has returned to the boundary.

According to our prefix sums, the `0` appears twice, meaning the ant has returned to the boundary twice after completing its movements dictated by `nums`.

The Python code for the above procedure using `itertools.accumulate` could look something like this:

```
from itertools import accumulate

nums = [3, -2, 2, -3, 1, -1]
# Calculate prefix sums
prefix_sums = list(accumulate(nums))
# Count the zeros in the prefix sums, indicating returns to the boundary
returns_to_boundary = sum(s == 0 for s in prefix_sums)

print(f"The ant returns to the starting point {returns_to_boundary} times.")
```

By running the code above, we would get the output indicating that the ant returns to the starting point 2 times.

Solution Implementation

Python

```
from itertools import accumulate
from typing import List

class Solution:
    def returnToBoundaryCount(self, nums: List[int]) -> int:
        # This function counts the number of times a running total
        # of the numbers in the list 'nums' returns to zero.

        # Use the accumulate function to create a running total of the numbers in 'nums'
        running_totals = accumulate(nums)

        # Use a generator expression to count how many times the running total is 0
        zero_count = sum(total == 0 for total in running_totals)

        return zero_count

# Example usage:
# sol = Solution()
# result = sol.returnToBoundaryCount([1, -1, 2, -2, 3, -3])
# print(result)  # This would print 3, since the running total would be 0 at three points
```

Java

```
class Solution {
    public int returnToBoundaryCount(int[] nums) {
        int count = 0; // This will hold the number of times the sum returns to 0 (boundary)
        int sum = 0; // This is used to compute the cumulative sum of the array elements

        // Iterate through each element in the array
        for (int num : nums) {
            sum += num; // Add the current element to the sum

            // If the sum is 0, increment the count
            // This condition checks if we've returned to the boundary (sum of 0)
            if (sum == 0) {
                count++;
            }
        }

        // Return the number of times the sum has returned to the boundary
        return count;
    }
}
```

C++

```
class Solution {
public:
    // This method counts how many times the cumulative sum returns to zero.
    int returnToBoundaryCount(vector<int>& nums) {
        int count = 0; // Initializes the count of times we return to boundary (cumulative sum equals zero)
        int sum = 0;  // Initializes the cumulative sum

        // Iterate over each element in the vector 'nums'
        for (int num : nums) {
            sum += num;             // Update the cumulative sum with the current element
            count += (sum == 0);   // If the cumulative sum is zero, increment the count
        }

        // Return the total count of times the cumulative sum returned to zero
        return count;
    }
};
```

TypeScript

```
// Counts the number of times a running sum returns to zero from a list of numbers
function returnToBoundaryCount(nums: number[]): number {
    // Initialize count of return-to-zero occurrences (ans) and the running sum (runningSum)
    let [zeroReturnCount, runningSum] = [0, 0];

    // Iterate over each number in the given array
    for (const num of nums) {
        // Add the current number to the running sum
        runningSum += num;
        // If the running sum is zero, increment the return-to-zero count
        zeroReturnCount += runningSum === 0 ? 1 : 0;
    }

    // Return the total count of return-to-zero occurrences
    return zeroReturnCount;
}
```

```
from itertools import accumulate
from typing import List

class Solution:
    def returnToBoundaryCount(self, nums: List[int]) -> int:
        # This function counts the number of times a running total
        # of the numbers in the list 'nums' returns to zero.

        # Use the accumulate function to create a running total of the numbers in 'nums'
        running_totals = accumulate(nums)

        # Use a generator expression to count how many times the running total is 0
        zero_count = sum(total == 0 for total in running_totals)

        return zero_count

# Example usage:
# sol = Solution()
# result = sol.returnToBoundaryCount([1, -1, 2, -2, 3, -3])
# print(result)  # This would print 3, since the running total would be 0 at three points
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `nums` list. This is because the `accumulate` function generates a cumulative sum of elements in `nums`, which requires one pass through all elements.

As for space complexity, contrary to the reference answer, it is not `O(1)`. Instead, it should be considered `O(n)` since the `accumulate` function produces an intermediate iterable with the cumulative sum that has the same number of elements as `nums`. However, if the accumulator is considered to be generated lazily and only its current value is stored at each step when iterating, then the space complexity can indeed be `O(1)`. This depends on the implementation of the `accumulate` function.