2765. Longest Alternating Subarray

Easy Array Enumeration

Leetcode Link

Problem Description

This LeetCode problem provides you with an integer array `nums` which is indexed starting from 0. You are tasked with finding the longest subarray within this array that is deemed 'alternating'. A subarray is defined as alternating if it follows two main criteria:

1. The length of the subarray, `m`, must be greater than 1.
2. The elements of the subarray must follow an alternating increment and decrement pattern. Specifically, the first element `s0` and the second element `s1` should satisfy the condition `s1 = s0 + 1`, and for subsequent elements, the difference should alternate between +1 and -1.

This alternating pattern implies that starting with the difference of +1 for the first pair, the next pair should have a difference of -1, followed by +1 for the next, and so on, effectively creating a sequence where the sign of the difference changes with each additional element.

You are required to return the maximum length of such an alternating subarray found in `nums`. If no alternating subarray is present, you should return `-1`.

An important note is that subarrays are continuous segments of the array and cannot be empty.

Intuition

The intuition behind the proposed solution is based on iterating over the array and examining each possible starting point for an alternating subarray. To effectively find alternating subarrays, we keep track of the required difference (`+1` or `-1`) at each step using a variable `k`. For each element in the array, we look ahead to see how far the alternating pattern continues before it breaks.

We initiate the search from each index `i`, which acts as the potential start of an alternating subarray. Using a second index `j`, we move through the array while the difference between successive elements `nums[j + 1]` and `nums[j]` matches `k`, the alternating difference we are currently looking for.

We start with `k` set to 1 since `s1` must be greater by 1 than `s0`. As we advance through the array and find that the alternate difference holds true, we increment `j`, invert `k`, and continue. This allows us to track whether the next element should be higher or lower than the previous one.

Once we have found a subarray that starts at `i` and ends at `j` (where `j - i + 1` is greater than 1) that satisfies the alternating condition, we compare its length with our current maximum length. If it's longer, we update the maximum length (`ans`) to this new value. We repeat this process for every index `i` in the array to make sure all possibilities are considered.

By rigorously checking every index as a potential start and extending the length of the subarray as long as the alternating condition holds, we can ensure the longest subarray is found.

Solution Approach

The provided solution follows a straightforward brute force enumeration approach to find the longest alternating subarray in `nums`. Here is a step-by-step explanation of the approach:

- We initiate a variable `ans` to keep track of the maximum length of an alternating subarray found so far, starting with a value of `-1`.
- We also determine the length of the array `n`.

For each index `i` in the range of `0` to `n`, which represents the potential starting point of an alternating subarray:

- We set `k` to `1`, which is the initial expected difference between the first and second elements of a potential alternating subarray.
- We also set `j` to `i`, with `j` serving as an index to explore the array forward from the starting point `i`.

We then enter a loop to test the alternating pattern:

- While `j + 1 < n` (to prevent going out of bounds) and `nums[j + 1] - nums[j] == k` (the adjacent elements meet the alternating condition), we increment `j` to continue extending the subarray and multiply `k` by `-1` to switch the expected difference for the next pair of elements.
- If the aforementioned while loop breaks, it means that either the end of the array is reached or the alternating pattern is not satisfied anymore.

After exiting the loop, we check if we have a valid subarray (of length greater than 1) by confirming `j - i + 1 > 1`.

- If it's a valid subarray, we update `ans` to be the maximum of its current value and `j - i + 1`, which is the length of the current alternating subarray.

At the end of the loop over `i`, the variable `ans` will hold the length of the longest alternating subarray found, or it will remain `-1` if no such subarray exists.

This enumeration technique, combined with tracking of the alternating pattern with a variable `k`, is effective in testing every possible subarray's start point and ensuring the maximal length subarray is identified. It's an exhaustive method, not relying on any specific algorithms or data structures beyond basic control flow and variable tracking, making it a brute force solution.

The absence of advanced data structures makes the code simple, but also means the time complexity is O(n^2) in the worst case, as for each starting point `i`, we may potentially scan up to `n - i` elements to find the longest subarray.

Example Walkthrough

Let's illustrate the solution approach with a small example using the following integer array `nums`:

```
nums = [5, 6, 7, 8, 7, 6, 7, 8, 9]
```

Given the array, we need to find the maximum length of an alternating subarray following an increment-decrement pattern starting with +1.

1. Initialize `ans` as `-1`. The length `n` of `nums` is `9`.
2. For each index `i` starting from `0`, initiate our search for an alternating subarray.
   - When `i = 0`, set `k = 1` and `j = 0`.
   - While loop:
     - For `j = 0`, `nums[1] - nums[0] = 6 - 5 = 1`, which is equal to `k`.
     - Increment `j` to `1`, and `k` becomes `-1`.
     - Now `j = 1`, `nums[2] - nums[1] = 7 - 6 = 1`, which is not equal to the current `k` (-1), so the loop breaks.
   - The subarray from index `0` to `1` has a length of 2 (`j - i + 1`), so we update `ans` to `2`.
3. Repeat this process for other starting points:
   - When `i = 1`, the alternating subarray is `6, 7, 8, 7, 6`, ending at `j = 5`. Update `ans` to `5`.
   - Subsequent checks for starting points `i = 2` to `i = 8` will reveal no longer subarray than the already found length `5`.

By following this algorithm, the final value of `ans` is `5`, which is the length of the longest alternating subarray found in `nums`, with the subarray being `[6, 7, 8, 7, 6]`. If no such subarray existed, `ans` would've remained `-1`.

Python Solution

```python
class Solution:
    def alternatingSubarray(self, nums: List[int]) -> int:
        # Initialize the answer to -1 to handle cases where no alternating subarray is found.
        max_length = -1
        # Get the length of the input list of numbers.
        n = len(nums)

        # Iterate over the list to find all possible starting points of alternating subarrays.
        for i in range(n):
            # Initialize the sign change indicator to 1 for alternating checks.
            sign = 1
            # Initialize the pointer to track subarray length from current position.
            end = i

            # Go through the list while the following conditions is true:
            # 1. The next position is within the array bounds.
            # 2. Consecutive numbers have an alternating difference.
            while end + 1 < n and nums[end + 1] - nums[end] == sign:
                # Move to the next number in the subarray.
                end += 1
                # Change the sign to alternate.
                sign *= -1

            # Update the length of the longest alternating subarray found.
            # Only consider subarrays with more than one element.
            if end - i + 1 > 1:
                max_length = max(max_length, end - i + 1)

        # Return the length of the longest alternating subarray, or -1 if none are found.
        return max_length

```

Java Solution

```java
class Solution {

    // Method to find the length of the longest alternating subarray
    public int alternatingSubarray(int[] nums) {
        // Initialize the answer to -1 since we want to find the length
        // and the minimum length of a valid alternating subarray is 2
        int maxLengthOfAllSubarray = -1;
        int arrayLength = nums.length;

        // Iterate over the array to find all possible subarrays
        for (int startIndex = 0; startIndex < arrayLength; ++startIndex) {
            // Initialize the alternating difference factor for the subarray
            int alternatingFactor = 1;
            // `startIndex` is the beginning of the potential alternating subarray
            int currentIndex = startIndex;

            // Compare adjacent elements to check if they form an alternating subarray
            for (; currentIndex + 1 < arrayLength && nums[currentIndex + 1] - nums[currentIndex] == alternatingFactor; ++currentIndex
                // After each comparison, flip the alternating difference factor
                alternatingFactor *= -1;
            }

            // Check if we found a valid alternating subarray of more than 1 element
            if (currentIndex - startIndex + 1 > 1) {
                // Update the answer if the current subarray is longer than the previous subarrays found
                maxLengthOfAllSubarray = Math.max(maxLengthOfAllSubarray, currentIndex - startIndex + 1);
            }
        }
        // Return the length of the longest alternating subarray found
        return maxLengthOfAllSubarray;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the length of the longest subarray
    // where adjacent elements are alternately increasing and decreasing
    int alternatingSubarray(vector<int>& nums) {
        int maxLen = -1; // Variable to store the length of longest subarray, initialized to -1
        int n = nums.size(); // Size of the input array
        for (int i = 0; i < n; ++i) { // Loop over the array
            int sequenceDiff = 1; // Initialize the difference that we are looking for (either 1 or -1)
            int j = i; // Start of the current subarray
            // Increase the length of the subarray while the difference between
            // consecutive elements matches the expected difference
            for (; j + 1 < n && nums[j + 1] - nums[j] == sequenceDiff; ++j) {
                sequenceDiff *= -1; // Flip the expected difference for the next pair
            }
            // If we found a subarray of length greater than 1, update the answer
            if (j - i + 1 > 1) {
                maxLen = max(maxLen, j - i + 1);
            }
        }
        // Return the length of the longest alternating subarray
        return maxLen;
    }
};

```

Typescript Solution

```typescript
function alternatingSubarray(nums: number[]): number {
    let maxSubarrayLength = -1; // Initialize the maximum subarray length to -1
    const arrayLength = nums.length; // Get the length of the input array

    // Iterate over the input array
    for (let startIndex = 0; startIndex < arrayLength; ++startIndex) {
        let alternatingDifference = 1; // This alternating difference changes from 1 to -1 and vice versa
        let endIndex = startIndex; // Initialize the end index for the subarray to the start index

        // Check if the current subarray alternates by checking the difference between consecutive elements
        while (endIndex + 1 < arrayLength && nums[endIndex + 1] - nums[endIndex] === alternatingDifference) {
            alternatingDifference *= -1; // Alternate the expected difference for the next pair
            endIndex++; // Increment the end index of the current subarray
        }

        // If the length of the current subarray is greater than 1, update the maxSubarrayLength
        if (endIndex - startIndex + 1 > 1) {
            maxSubarrayLength = Math.max(maxSubarrayLength, endIndex - startIndex + 1);
        }
    }

    return maxSubarrayLength; // Return the length of the longest alternating subarray
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n^2)`, where `n` is the length of the input array `nums`. This is because the code includes a loop that iterates over each possible starting index `i` for a subarray. For each starting index, the inner loop potentially iterates over the remaining part of the array to find the longest alternating subarray starting at that point. In the worst-case scenario, this takes `O(n)` time for each starting index, and since there are `n` possible starting indices, the overall complexity is `n * O(n)`, which simplifies to `O(n^2)`.

The space complexity of the algorithm is `O(1)`, which means it is constant. This is because the memory usage does not depend on the size of the input array; the code only uses a fixed number of integer variables (`ans`, `n`, `i`, `k`, and `j`) to compute the result.