# 978. Longest Turbulent Subarray

Medium Array Dynamic Programming Sliding Window

Leetcode Link

## Problem Description

The problem provides us with an integer array named `arr`, and our goal is to find the length of the longest turbulent subarray. A turbulent subarray is defined uniquely by its characteristic that the inequality signs between consecutive elements change or "flip" from one pair to the next. Concretely, a subarray is turbulent if for any two adjacent elements within it, one is greater than the next and the next after that is less than its predecessor, or vice versa; this pattern alternates throughout the subarray.

To put it another way, if the indices of the subarray are from `i` to `j`, then for the index `k` within `i` and `j-1`, two conditions are defined:

1. When `k` is odd, `arr[k]` should be greater than `arr[k + 1]`, and when `k` is even, `arr[k]` should be less than `arr[k + 1]`.
2. Or alternatively, when `k` is odd, `arr[k]` should be less than `arr[k + 1]`, and when `k` is even, `arr[k]` should be greater than `arr[k + 1]`.

## Intuition

When approaching this problem, we consider that a turbulent subarray can start at any index and the conditions for a sequence of elements to be turbulent depend solely on the relationship between adjacent elements. Thus, one way to solve this is by iterating through the array and tracking the length of the current turbulent subarray as we go.

The intuition behind the solution lies in a sliding window or two-pointer approach, where we track the current and maximum lengths of valid subarrays. However, we can optimize this strategy by keeping track of the last two comparisons (whether the last was '>' or '<') which can be done with counters that reset when the pattern is broken.

In the provided solution, two variables `f` and `g` are used to keep track of the length of two types of turbulent subarrays, one where the first comparison is '<' and the other where the first comparison is '>'. If the current pair of elements continue the turbulence condition, we increment the appropriate counter (`f` or `g`), otherwise, we reset it to 1 (the current pair itself can be the beginning of a new subarray). `ans` is used to keep track of the maximum size found so far.

We iterate through the array using the `pairwise` utility, which gives us adjacent pairs of elements for easy comparison. For each pair, using the comparison result, we update `f` and `g`, then update `ans` to hold the maximum of `ans`, `f`, and `g`. At the end of the iteration, `ans` contains the length of the largest turbulent subarray.

The solution leverages a dynamic and continuous update of variables to reflect the current state of the subarray being checked, thus arriving at the final answer without needing to store any intermediate subarrays or additional complex data structures.

## Solution Approach

The implementation of the solution through the provided Python code uses a straightforward approach, mainly utilizing iteration and simple condition checks without the need for additional data structures beyond simple variables. Here's how it works:

### Variables:

- `ans`: To keep track of the maximum subarray size found during iteration.
- `f`: To track the length of turbulent subarrays where the first condition (`arr[k] < arr[k + 1]` when `k` is even, and `arr[k] > arr[k + 1]` when `k` is odd) holds true.
- `g`: To track the length of turbulent subarrays where the second condition (`arr[k] > arr[k + 1]` when `k` is even, and `arr[k] < arr[k + 1]` when `k` is odd) holds true.

### Iteration:

- The code uses the `pairwise` function to create a windowed pair of array elements `(a, b)` for each adjacent pair in `arr`.
- On each iteration step, we compare the elements `a` and `b` in the tuple to determine the continuation or the end of a turbulent subarray.

### Condition checks and Updates:

- `ff` and `gg` are temporary variables used to calculate the potential new lengths of `f` and `g`.
- If `a < b` and we are considering the odd index for `k`, `f` should be incremented, since the condition `arr[k] > arr[k + 1]` must hold true for `k` being even; otherwise, `f` resets to 1 (subarray starts anew).
- Conversely, if `a > b` for an even index, `g` should be incremented, as the condition `arr[k] < arr[k + 1]` must hold true for an odd `k`; otherwise, `g` resets to 1.
- After each comparison, `f` and `g` are updated with the values from `ff` and `gg`, respectively.
- `ans` is updated with the maximum value between `ans`, `f`, and `g`.

### Return Value:

- After the loop completes, the value of `ans` denotes the longest turbulent subarray that can be found in `arr`.

This solution approach relies on dynamically updating counters and checking each pair once. It avoids the need for complex data structures or algorithms. Due to its simplicity and only using iteration and basic arithmetic operations, the overall time complexity is `O(n)`, where `n` is the number of elements in `arr`.

### Example Walkthrough

Let's illustrate the solution approach with a small example using the array `arr = [9, 4, 2, 10, 7, 8, 8, 1, 9]`.

1. Initialize `ans = 1`, `f = 1`, and `g = 1` because the minimum length of a turbulent subarray is 1 (a single element).
2. Begin iterating from the start of the array using the `pairwise` function:
   - First pair `(9, 4)`: since `9 > 4`, set `ff = 1` and `gg = f + 1`. Now, `g` becomes 2 (`g = gg`), because this pair meets the second condition.
   - Next pair `(4, 2)`: `4 > 2`, increment `g` to 3 (`g = g + 1`), `f` resets to 1.
   - Next pair `(2, 10)`: now `2 < 10`, so we increment `f` to `g + 1`, which is now 4, and reset `g` to 1.
   - Continue with this pattern for subsequent pairs.
3. At each step, update `ans` to the maximum of `ans`, `f`, and `g` to ensure it always holds the maximum subarray length found so far.

Following through with the example sequence:

- After processing `(10, 7)`, `f` is reset to 1 since `10 > 7`, `g = f + 1 = 2`, and `ans` is updated to `max(ans, f, g) = 4`.
- After `(7, 8)`, `f = g + 1 = 3`, `g` is reset to 1, update `ans`.
- After `(8, 8)`, both `f` and `g` reset to 1, because the elements are equal and that does not meet any turbulent condition.
- After `(8, 1)`, `g = f + 1 = 2`, `f` is reset to 1, update `ans`.
- After `(1, 9)`, `f = g + 1 = 3`, `g` is reset to 1, update `ans` which remains 4, since `f` is not greater than the current `ans`.

4. Finally, the iteration ends and we return `ans`, which is still 4, representing the longest turbulent subarray `[4, 2, 10, 7]`.

This example showcases the elegance and simplicity of the solution. Without storing any subarrays or using complex structures, the algorithm dynamically keeps track of the state and updates it with each element pair. Overall, this clever approach ensures optimal performance with linear time complexity, only passing through the array once.

## Python Solution

```python
from typing import List

class Solution:
    def maxTurbulenceSize(self, arr: List[int]) -> int:
        # Initialize the maximum length of a turbulent subarray as 1.
        max_length = increasing = decreasing = 1

        # Iterate over the array in pairs to check for turbulence.
        for i in range(len(arr) - 1):
            # Get the current and next element.
            current, next_element = arr[i], arr[i + 1]

            # Temporarily store the new length for an increasing and decreasing sequence.
            temp_inc = decreasing + 1 if current < next_element else 1
            temp_dec = increasing + 1 if current > next_element else 1

            # Update the current length of increasing and decreasing sequences.
            increasing, decreasing = temp_inc, temp_dec

            # Update the maximum length if we found a longer turbulent sequence.
            max_length = max(max_length, increasing, decreasing)

        # Return the maximum length of turbulent subarray found.
        return max_length

```

## Java Solution

```java
class Solution {
    public int maxTurbulenceSize(int[] arr) {
        int maxLength = 1; // This will store the maximum length of the turbulence.
        int incLength = 1; // This will store the current increasing turbulence length.
        int decLength = 1; // This will store the current decreasing turbulence length.

        // Iterate through the array starting from the second element.
        for (int i = 1; i < arr.length; ++i) {
            // Determine the new length of increasing turbulence if the current pair is turbulent,
            // otherwise reset the length to 1.
            int nextIncLength = arr[i - 1] < arr[i] ? decLength + 1 : 1;
            // Do the same for decreasing turbulence.
            int nextDecLength = arr[i - 1] > arr[i] ? incLength + 1 : 1;

            // Update the current turbulence lengths.
            incLength = nextIncLength;
            decLength = nextDecLength;

            // Update the maximum length found so far.
            maxLength = Math.max(maxLength, Math.max(incLength, decLength));
        }

        // Return the maximum turbulence length found.
        return maxLength;
    }
}

```

## C++ Solution

```cpp
#include <vector>
#include <algorithm> // for max function

class Solution {
public:
    // Returns the length of the maximum size turbulent subarray of `arr`.
    int maxTurbulenceSize(vector<int>& arr) {
        // Initialize the answer to 1, since the minimum turbulent subarray has a length of 1
        int max_length = 1;

        // Initialize counters for increasing and decreasing sequences
        // `current_inc` = length of the current subarray where arr[i - 1] < arr[i]
        // `current_dec` = length of the current subarray where arr[i - 1] > arr[i]
        int current_inc = 1, current_dec = 1;

        // Loop from the second element to the end of the array
        for (int i = 1; i < arr.size(); ++i) {
            // If the current element is greater than the previous, increment the increasing counter,
            // otherwise reset it to 1 (a subarray of length 1 starts with just the current element)
            int next_inc = arr[i - 1] < arr[i] ? current_dec + 1 : 1;

            // If the current element is less than the previous, increment the decreasing counter,
            // otherwise reset it to 1 (a subarray of length 1 starts with just the current element)
            int next_dec = arr[i - 1] > arr[i] ? current_inc + 1 : 1;

            // Update the current counters with computed next values
            current_inc = next_inc;
            current_dec = next_dec;

            // Update max_length with the longer of the current increasing and decreasing sequences
            max_length = max({max_length, current_inc, current_dec});
        }

        // Return the maximum length of a turbulent subarray in `arr`
        return max_length;
    }
};

```

## Typescript Solution

```typescript
function maxTurbulenceSize(arr: number[]): number {
    // Initializing variables to track the lengths of turbulent sequences
    let increasingSequenceLength = 1; // Tracks length of increasing sequence
    let decreasingSequenceLength = 1; // Tracks length of decreasing sequence
    let maxSequenceLength = 1; // Tracks the maximum length found

    // Iterate through the array starting from the second element
    for (let i = 1; i < arr.length; ++i) {
        // If the current element is greater than the previous,
        // then the current sequence of 'increasingSequenceLength' can extend
        // the 'decreasingSequenceLength' sequence by 1.
        // Otherwise, reset 'increasingSequenceLength' as a new sequence starts.
        let tempIncreasingLength = arr[i - 1] < arr[i] ? decreasingSequenceLength + 1 : 1;

        // Similarly, if the current element is less than the previous,
        // then the current sequence of 'decreasingSequenceLength' can extend
        // the 'increasingSequenceLength' sequence by 1.
        // Otherwise, reset 'decreasingSequenceLength' as a new sequence starts.
        let tempDecreasingLength = arr[i - 1] > arr[i] ? increasingSequenceLength + 1 : 1;

        // Update the 'increasingSequenceLength' and 'decreasingSequenceLength' with new lengths
        increasingSequenceLength = tempIncreasingLength;
        decreasingSequenceLength = tempDecreasingLength;

        // Update 'maxSequenceLength' to hold the max
        // between itself and the two sequence lengths
        maxSequenceLength = Math.max(maxSequenceLength, increasingSequenceLength, decreasingSequenceLength);
    }

    // Return the maximum turbulence size found in the array
    return maxSequenceLength;
}

```

## Time and Space Complexity

### Time Complexity

The time complexity of the code is primarily governed by the single for loop that iterates through the pairwise elements of the array `arr`. In Python, the `pairwise` utility creates an iterator that aggregates elements from each iterable sequence. Considering an input array of size `n`, there will be `n - 1` iterations (since `pairwise` generates tuples of two consecutive elements).

Within the for loop, there are constant time operations like comparisons, assignments, and a call to the `max` function with three arguments. These operations contribute `O(1)` time complexity for each iteration.

Therefore, the total time complexity of this algorithm is `O(n)`.

### Space Complexity

The space complexity of the algorithm is determined by the extra space used besides the input array. In this case, only a constant number of integer variables (`ans`, `f`, `g`, `ff`, and `gg`) are used. No auxiliary data structures that scale with the input size are utilized.

Thus, the space complexity is `O(1)`, which means it uses a constant amount of space regardless of the input array size.