2210. Count Hills and Valleys in an Array

Easy Array

Problem Description

In this problem, we are given an integer array `nums` which is 0-indexed. Our task is to determine how many hills and valleys the array contains. A hill is defined as an element which is larger than its closest distinctly different neighbors, and a valley is defined as an element which is smaller than its closest distinctly different neighbors. An important condition is that both neighbors must be different from the element being assessed and that equal elements adjacent to each other are considered as part of the same hill or valley.

Another important detail to consider is that an element needs to have distinct neighbors on both sides for it to be part of a hill or a valley. Hence, the first and the last elements of the array can never be hills or valleys since they lack a neighbor on one side.

Intuition

The intuition behind the solution is that, in order to find the hills and valleys, we should iterate through the array starting from the second element and ending at the second-to-last element. We compare each element with its neighbors to determine whether it forms a hill or a valley, taking into account that consecutive equal elements should be treated as one single entity.

The approach is to have two pointers, `i` and `j`, where `i` iterates through the array and `j` keeps track of the last element that is not equal to the current element `i`. By doing this, we skip over the equal elements because they don't help in determining a new hill or valley but are part of an existing one.

During each iteration, we compare `nums[i]` with `nums[j]` and `nums[i+1]`:

- If `nums[i] > nums[j]` and `nums[i] > nums[i+1]`, it means that `nums[i]` is a hill. We increment the answer counter.
- If `nums[i] < nums[j]` and `nums[i] < nums[i+1]`, it means that `nums[i]` is a valley. We increment the answer counter as well.
- If `nums[i]` is equal to `nums[i+1]`, we continue the iteration without changing `j` because we're still within the same hill or valley region.

Solution Approach

The implementation of the solution provided is quite straightforward and does not use any complex data structures or algorithms. It relies on a single pass iteration with a simple comparison check. Let's walk through the key steps of the provided solution:

1. Initialize two pointers `i` and `j`. `i` starts from 1 since the first element cannot be a hill or valley, and `j` starts from 0, which will keep track of the last index that has a different value from `nums[i]`.
2. Initialize a counter `ans` to 0 to keep track of the number of hills and valleys.
3. Start a loop from the second element (index 1) to the second-to-last element (index `len(nums) - 2`, since we will be checking `nums[i + 1]`).
4. Inside the loop:
   - If `nums[i] == nums[i + 1]`, then skip the rest of the loop and continue with the next iteration as we're inside a range of duplicate elements that are part of the same hill or valley.
   - If `nums[i] > nums[j]` and `nums[i] > nums[i + 1]`, we have found a hill, increment `ans` by 1.
   - If `nums[i] < nums[j]` and `nums[i] < nums[i + 1]`, we have found a valley, increment `ans` by 1.
   - Update the `j` pointer to `i` as this comparison is complete, and `i` had a different value than `nums[i + 1]`.
5. After the loop ends, return the value of `ans`, which now contains the number of hills and valleys in the array.

The algorithm's time complexity is `O(n)`, where `n` is the number of elements in `nums`, since it involves a single loop through the array. The space complexity is `O(1)` because we use a fixed amount of additional space.

This solution approach successfully counts the number of hills and valleys by using two pointers and a counter, effectively bypassing adjacent duplicates and considering them as part of the same hill/valley, thereby satisfying the problem's constraints and requirements.

Example Walkthrough

Let's illustrate the solution approach using a small example. Consider the array `nums` given by `[2, 2, 3, 4, 3, 3, 2, 2, 1, 1, 2]`. We want to find out how many hills and valleys this array contains. Looking at the array, we can visually identify one hill (comprising the element `4`) and two valleys (comprising elements `3` at index 4 and `1` at index 8).

Now, following the provided solution approach:

1. Initialize two pointers `i = 1` and `j = 0`. Initialize the answer counter `ans` to `0`.
2. Start iterating from the second element to the second-to-last element:
   - At `i = 1`, `nums[i] = 2` and `nums[j] = 2`. They are equal, so we continue without incrementing `ans`.
   - At `i = 2`, `nums[i] = 3`. Now we compare `nums[i]` with `nums[j]` (`2`) and `nums[i+1]` (`4`). Since `3` is neither a hill nor a valley, we move on, updating `j` to `i`.
   - At `i = 3`, `nums[i] = 4`, and it is greater than both `nums[j]` (`3`) and `nums[i+1]` (`3`), so we have found a hill. Increment `ans` to `1`, and update `j` to `i`.
   - At `i = 4`, `nums[i] = 3`. It is less than both `nums[j]` (`4`) and `nums[i+1]` (`3`), so we have found a valley. Increment `ans` to `2`, but do not update `j` because `nums[i]` is equal to `nums[i+1]`.
   - We skip `i = 5` since `nums[i] = nums[i+1]`.
   - At `i = 6`, `nums[i] = 2`, which is less than both `nums[j]` (`3`) and `nums[i+1]` (`2`), but since `nums[i]` is equal to `nums[i+1]`, we skip updating `ans`.
   - At `i = 7`, we skip since `nums[i] = nums[i+1]`.
   - At `i = 8`, `nums[i] = 1`, and it is less than both `nums[j]` (`2`) and `nums[i+1]` (`1`), so we have found another valley. Increment `ans` to `3`.
3. There are no more elements to check, so we have now reached the end of the array. The value of `ans` is `3`, which is the total number of hills and valleys in the array.

The process demonstrates how the algorithm effectively skips over equal elements, treats them as part of the same hill or valley, and correctly identifies distinct hills and valleys in compliance with the problem's definition.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def countHillValley(self, nums: List[int]) -> int:
        # Initialize the count of hills and valleys
        count = 0
        # Initialize a pointer to track the last significant number (hill or valley)
        last_significant_number_index = 0

        # Iterate over the array starting from the second element and stopping before the last
        for i in range(1, len(nums) - 1):
            # Skip if the current number is the same as the next, as it cannot be a hill or valley
            if nums[i] == nums[i + 1]:
                continue

            # Check if the current number is a hill by comparing with the last significant number and next number
            if nums[i] > nums[last_significant_number_index] and nums[i] > nums[i + 1]:
                count += 1
            # Check if the current number is a valley by comparing with the last significant number and next number
            elif nums[i] < nums[last_significant_number_index] and nums[i] < nums[i + 1]:
                count += 1

            # Update the pointer to the last significant number as the current number
            last_significant_number_index = i

        # Return the total count of hills and valleys found
        return count
```

Java

```java
class Solution {
    public int countHillValley(int[] nums) {
        // Initialize the counter for hills and valleys found.
        int count = 0;

        // Loop through the input array, checking for hills or valleys.
        // The 'previousIndex' variable will track the index of the last
        // element in the sequence that is not equal to the current element.
        for (int currentIndex = 1, previousIndex = 0; currentIndex < nums.length - 1; ++currentIndex) {

            // Skip the current element if it's equal to the next one
            // since we're looking for unique hills or valleys.
            if (nums[currentIndex] == nums[currentIndex + 1]) {
                continue;
            }

            // Check for a hill: the current number is greater than both its adjacent numbers.
            if (nums[currentIndex] > nums[previousIndex] && nums[currentIndex] > nums[currentIndex + 1]) {
                count++;
            }

            // Check for a valley: the current number is less than both its adjacent numbers.
            if (nums[currentIndex] < nums[previousIndex] && nums[currentIndex] < nums[currentIndex + 1]) {
                count++;
            }

            // Update the previousIndex to the current index.
            previousIndex = currentIndex;
        }
        // Return the total number of hills and valleys found.
        return count;
    }
}
```

C++

```cpp
class Solution {
public:
    int countHillValley(vector<int>& nums) {
        int count = 0; // Initialize counter for hills and valleys to zero

        // We use two pointers, 'previous' to keep track of the last distinct element
        // and 'current' which will iterate through the vector, starting from index 1
        for (int current = 1, previous = 0; current < nums.size() - 1; ++current) {

            // Skip over duplicate adjacent values
            if (nums[current] == nums[current + 1]) {
                continue;
            }

            // Check if nums[current] forms a hill
            if (nums[current] > nums[previous] && nums[current] > nums[current + 1]) {
                ++count; // Increment count if we found a hill
            }

            // Check if nums[current] forms a valley
            if (nums[current] < nums[previous] && nums[current] < nums[current + 1]) {
                ++count; // Increment count if we found a valley
            }

            // Update 'previous' to the last distinct element's index
            previous = current;
        }

        return count; // Return the final count of hills and valleys
    }
};
```

TypeScript

```typescript
function countHillValley(nums: number[]): number {
    // n holds the total number of elements in the input array nums.
    const numberOfElements = nums.length;
    // Initialize hillValleyCount to count the number of hills and valleys found.
    let hillValleyCount = 0;
    // Initialize previousValue to the first value of the nums array.
    let previousValue = nums[0];

    // Iterate over the array starting from the second element till the second last element.
    for (let i = 1; i < numberOfElements - 1; i++) {
        // currentValue holds the current element in the array.
        const currentValue = nums[i];
        // nextValue holds the next element in the array.
        const nextValue = nums[i + 1];
        // Continue to the next iteration if the current and next values are the same.
        if (currentValue == nextValue) {
            continue;
        }
        // Check if the currentValue is either a hill or a valley, and if so, increment hillValleyCount.
        if ((currentValue > previousValue && currentValue > nextValue) || (currentValue < previousValue && currentValue < nextVal
            hillValleyCount += 1;
        }
        // Update previousValue to be the currentValue as we move forward in the array.
        previousValue = currentValue;
    }
    // Return the total count of hills and valleys found in the array.
    return hillValleyCount;
}
```


```python
from typing import List

class Solution:
    def countHillValley(self, nums: List[int]) -> int:
        # Initialize the count of hills and valleys
        count = 0
        # Initialize a pointer to track the last significant number (hill or valley)
        last_significant_number_index = 0

        # Iterate over the array starting from the second element and stopping before the last
        for i in range(1, len(nums) - 1):
            # Skip if the current number is the same as the next, as it cannot be a hill or valley
            if nums[i] == nums[i + 1]:
                continue

            # Check if the current number is a hill by comparing with the last significant number and next number
            if nums[i] > nums[last_significant_number_index] and nums[i] > nums[i + 1]:
                count += 1
            # Check if the current number is a valley by comparing with the last significant number and next number
            elif nums[i] < nums[last_significant_number_index] and nums[i] < nums[i + 1]:
                count += 1

            # Update the pointer to the last significant number as the current number
            last_significant_number_index = i

        # Return the total count of hills and valleys found
        return count
```


Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the input list `nums`. This is because the main operation of the code, iterating through the list, is done in a single pass from the second element to the second-to-last element. Each comparison operation within the loop is done in constant time.

Space Complexity

The space complexity of the code is `O(1)` (constant space complexity). This is because the space used by the variables `ans`, `j`, and `i` does not scale with the size of the input list `nums`. No additional data structures that scale with the input size are used.