376. Wiggle Subsequence

Medium Greedy Array Dynamic Programming Leetcode Link

Problem Description

The aim of this problem is to find the length of the longest *wiggle subsequence* in a given integer array, `nums`. A wiggle sequence is defined as one where consecutive numbers alternate between increasing and decreasing. That means the difference between successive numbers should switch between positive and negative. Single-element sequences and two-element sequences with distinct values are also considered wiggle sequences.

For instance, `[1, 7, 4, 9, 2, 5]` is a wiggle sequence because the differences between consecutive numbers `(6, -3, 5, -7, 3)` switch signs from positive to negative and vice versa. On the other hand, `[1, 4, 7, 2, 5]` is not a wiggle sequence because it starts with two positive differences. Similarly, `[1, 7, 4, 5, 5]` isn't either because it ends with a difference of zero.

A subsequence is derived by removing some elements from the original sequence while maintaining the relative order of the remaining elements. The problem asks to return the maximum length of such a longest wiggle subsequence.

Intuition

The intuition behind solving this problem lies in dynamic programming. The solution leverages two counters, `up` and `down`, to keep track of the two scenarios while traversing the array:

- `up`: This counter is incremented when the current element is greater than the preceding one, indicating an "up" wiggle. If this occurs, the length of the longest wiggle sequence considering the current number as an "up" number is the current value of `down` plus one. In other words, it's extending a sequence that was last going "down" with an "up" number.

- `down`: Conversely, this counter is incremented when the current element is less than the preceding one, signifying a "down" wiggle. In this situation, the length of the longest wiggle sequence considering the current number as a "down" number is derived from the current length of `up` plus one. That is, we're adding a "down" number to a subsequence that was previously "up".

An important point to note is that for both `up` and `down`, we use the `max()` function to ensure that we are always recording the maximum length of wiggle subsequence found so far – effectively making this a dynamic programming solution where the optimal solution of the current step is built upon the previous steps. At each step, we make a decision based on the current and previous elements to update these counters.

At the end of the traversal, the length of the longest wiggle subsequence will be the maximum value between the `up` and `down` counters because we are interested in the longest possible sequence, regardless of whether it ends on an "up" wiggle or a "down" wiggle.

Solution Approach

The implementation of the solution is grounded in the principles of dynamic programming, which is an optimization technique that solves complex problems by breaking them down into simpler subproblems.

In this specific problem, the dynamic programming pattern used involves two one-dimensional arrays, `up` and `down`, where each entry `up[i]` or `down[i]` would normally represent the length of the longest wiggle subsequence up to the `i-th` index of `nums` that ends with an increasing or decreasing wiggle, respectively. The reference solution compresses these arrays into two simple integer variables because it cleverly leverages the fact the current state only depends on the previous state.

Here's how the algorithm progresses:

1. Initialize two variables `up` and `down` to 1, since a single element is trivially a wiggle subsequence.
2. Loop through the elements of the array starting from the second element because we're comparing each element with its previous one.
3. For each element at index `i`, compare it with the element at index `i - 1`.
   - If `nums[i]` is greater than `nums[i-1]`, it means we have a potential "up" wiggle. We then update the `up` counter as follows:
     ```
     up = max(up, down + 1)
     ```
     This is under the notion that the current "up" wiggle can extend a sequence that was previously "down", hence `down + 1`.
   - If `nums[i]` is less than `nums[i - 1]`, it signals a potential "down" wiggle and `down` is updated similarly:
     ```
     down = max(down, up + 1)
     ```
     Here, the current "down" wiggle can extend a sequence that was previously "up".
4. As the `for` loop continues, `up` and `down` are updated dynamically with respect to the sign alternation between consecutive elements.
5. After completing the traversal, the largest of `up` or `down` is returned as the maximum length of the longest wiggle subsequence found.

The reason behind this simple and elegant implementation is that for every position in the array `nums[i]`, the `up` and `down` states captures the length of the longest wiggle subsequence that ends with a "down" or "up" wiggle up to that position, respectively. The two possible cases of a wiggle (up or down) are covered by these two variables. There's no need to maintain an additional array because the maximum lengths of "up" and "down" wiggle subsequences at any position only depend on the immediate previous one.

Therefore, this algorithm is efficient in terms of both time complexity, which is `O(n)` since it only requires a single pass through the array, and space complexity, which is `O(1)` because it maintains only a constant number of variables.

Example Walkthrough

Let's consider the following input array: `nums = [1, 17, 5, 10, 13, 15, 10, 5, 16, 8]`. We want to find the length of the longest wiggle subsequence.

Following the solution approach:

1. Initialize two variables `up` and `down` to 1. Without loss of generality, we can think of every single number as a trivial wiggle subsequence.

2. Start loop from index 1, since we can only start determining a wiggle by comparing at least two numbers:

   - At `i = 1` (`nums[1] = 17`, `nums[0] = 1`):
     - As `nums[1] > nums[0]`, we have an "up" wiggle. Update `up`:
       ```
       up = max(up, down + 1) = max(1, 1 + 1) = 2
       ```
   - At `i = 2` (`nums[2] = 5`, `nums[1] = 17`):
     - As `nums[2] < nums[1]`, we have a "down" wiggle. Update `down`:
       ```
       down = max(down, up + 1) = max(1, 2 + 1) = 3
       ```
   - At `i = 3` (`nums[3] = 10`, `nums[2] = 5`):
     - As `nums[3] > nums[2]`, we have an "up" wiggle. Update `up`:
       ```
       up = max(up, down + 1) = max(2, 3 + 1) = 4
       ```

3. Continue this process for each element in the array. The updates will proceed as follows:

   - At `i = 4`:
     - `nums[4] = 13`, `nums[3] = 10`, so `up` remains 4 (`13` continues the "up" sequence from `10`).
   - At `i = 5`:
     - `nums[5] = 15`, `nums[4] = 13`, so `up` remains 4 (`15` continues the "up" sequence from `13`).
   - At `i = 6`:
     - `nums[6] = 10`, `nums[5] = 15`, we update `down` to 5 (as `10` is down from `15`).
   - At `i = 7`:
     - `nums[7] = 5`, `nums[6] = 10`, `down` remains 5 (`5` continues the "down" sequence from `10`).
   - At `i = 8`:
     - `nums[8] = 16`, `nums[7] = 5`, we update `up` to 6 (as `16` is up from `5`).
   - At `i = 9`:
     - `nums[9] = 8`, `nums[8] = 16`, we update `down` to 7 (as `8` is down from `16`).
4. By the end of the loop, the `up` and `down` variables have been updated accordingly. The maximum length of the longest wiggle subsequence is the larger one of `up` and `down`, which in this case is `down = 7`.

This example illustrates how the approach keeps updating the two counters that represent the length of the longest possible wiggle subsequences based on whether the current element goes "up" or "down" relative to the previous element in the subsequence. With this method, we can effectively solve the problem in linear time with constant extra space.

Python Solution

```python
class Solution:
    def wiggleMaxLength(self, nums: List[int]) -> int:
        if not nums:
            return 0

        # Initialize the two counters that represent the lengths of the longest wiggle sequences
        # ending with a rising edge (up) and with a falling edge (down),
        # starting with the first element both being 1.
        up_sequence_length = 1
        down_sequence_length = 1

        # Iterate through the array starting from the second element
        for i in range(1, len(nums)):
            # If the current element is greater than the previous one,
            # update the rising edge length (up_sequence_length)
            if nums[i] > nums[i - 1]:
                up_sequence_length = max(up_sequence_length, down_sequence_length + 1)
            # If the current element is smaller than the previous one,
            # update the falling edge length (down_sequence_length)
            elif nums[i] < nums[i - 1]:
                down_sequence_length = max(down_sequence_length, up_sequence_length + 1)

        # Return the maximum length between the two wiggle subsequences
        return max(up_sequence_length, down_sequence_length)

```

Java Solution

```java
class Solution {
    public int wiggleMaxLength(int[] nums) {
        int increasingLength = 1, decreasingLength = 1; // Initialize lengths of the wiggle subsequence

        // Loop through the array starting from the second element
        for (int i = 1; i < nums.length; ++i) {
            if (nums[i] > nums[i - 1]) {
                // If the current element is greater than the previous one,
                // then a 'wiggle' peak has been found.
                // Update the increasingLength to be the larger of itself or
                // one more than decreasingLength (indicating a new peak).
                increasingLength = Math.max(increasingLength, decreasingLength + 1);
            } else if (nums[i] < nums[i - 1]) {
                // If the current element is less than the previous one,
                // then a 'wiggle' trough has been found.
                // Update the decreasingLength to be the larger of itself or
                // one more than increasingLength (indicating a new trough).
                decreasingLength = Math.max(decreasingLength, increasingLength + 1);
            }
            // If nums[i] == nums[i - 1], do nothing, as equal values do not contribute to wiggle length
        }

        // The maximum length of wiggle subsequence would be the max of both
        // increasing and decreasing lengths
        return Math.max(increasingLength, decreasingLength);
    }
}

```

C++ Solution

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

class Solution {
public:
    int wiggleMaxLength(std::vector<int>& nums) {
        int lengthUp = 1;   // Initialize 'lengthUp' to represent the length of a wiggle sequence ending with an 'up' difference
        int lengthDown = 1; // Initialize 'lengthDown' to represent the length of a wiggle sequence ending with a 'down' difference

        // Iterate through the array starting from the second element
        for (int i = 1; i < nums.size(); ++i) {
            // Check for an 'up' wiggle
            if (nums[i] > nums[i - 1]) {
                // If the current number is greater than the previous, update 'lengthUp'
                // It's the max of itself and 'lengthDown + 1' (to include the current 'up' wiggle)
                lengthUp = std::max(lengthUp, lengthDown + 1);
            }
            // Check for a 'down' wiggle
            else if (nums[i] < nums[i - 1]) {
                // If the current number is less than the previous, update 'lengthDown'
                // It's the max of itself and 'lengthUp + 1' (to include the current 'down' wiggle)
                lengthDown = std::max(lengthDown, lengthUp + 1);
            }
            // If nums[i] == nums[i - 1], do nothing as equal elements do not contribute to wiggle sequence
        }
        // Return the maximum length of the two possible wiggle sequences
        return std::max(lengthUp, lengthDown);
    }
};

```

Typescript Solution

```typescript
/**
 * Calculates the length of the longest wiggle sequence from the given array.
 * A wiggle sequence is made up of elements where the differences between
 * successive numbers strictly alternate between positive and negative.
 *
 * @param {number[]} nums - The input array of numbers
 * @return {number} - The length of the longest wiggle sequence
 */
function wiggleMaxLength(nums: number[]): number {
    // Initialize 'up' and 'down' to count the length of the last wiggle subsequences
    // that are respectively increasing and decreasing.
    let upSequenceLength = 1;
    let downSequenceLength = 1;

    // Iterate through the array starting from the second element
    for (let i = 1; i < nums.length; i++) {
        const previousNum = nums[i - 1];
        const currentNum = nums[i];

        // If the current number is greater than the previous, the sequence is going up.
        if (currentNum > previousNum) {
            // Update upSequenceLength to be the downSequenceLength plus 1
            // which represents adding the current number to the sequence after a down.
            upSequenceLength = Math.max(upSequenceLength, downSequenceLength + 1);
        }
        // If the current number is less than the previous, the sequence is going down.
        else if (currentNum < previousNum) {
            // Update downSequenceLength to be the upSequenceLength plus 1
            // which represents adding the current number to the sequence after an up.
            downSequenceLength = Math.max(downSequenceLength, upSequenceLength + 1);
        }
        // If the numbers are equal, neither sequence length is updated, as it doesn't contribute
        // to a wiggle pattern (since there's no change).
    }

    // Return the maximum length between the two subsequences since they track the last wiggle
    // and the longest sequence can end on either an 'up' or a 'down'.
    return Math.max(upSequenceLength, downSequenceLength);
}

```

Time and Space Complexity

The given Python code snippet implements the solution for finding the length of the longest wiggle subsequence in an array.

Time Complexity

To analyze the time complexity, let's look at the operations performed by the code. The code iterates through the `nums` list once using a single loop that starts from `1` up to `len(nums)` (non-inclusive). Inside this loop, there are only constant-time operations: comparisons, arithmetic operations, and `max` function calls on integers.

As the loop runs for `n-1` iterations (where `n` is the length of `nums`) and each operation within the loop takes constant time, the overall time complexity of the code is `O(n)`.

Space Complexity

Regarding space complexity, the code uses a fixed number of integer variables (`up` and `down`). No matter the size of the input, the space used by these variables does not increase. There are no additional data structures like lists or sets that grow with the size of the input.

Therefore, the space complexity of the code is `O(1)`, which means it's constant space complexity.