2771. Longest Non-decreasing Subarray From Two Arrays

Medium Array Dynamic Programming

Leetcode Link

Problem Description

In this problem, you have two integer arrays `nums1` and `nums2`, both containing `n` elements, indexed from 0. Your goal is to create a third array `nums3`, also with `n` elements, where each element at index `i` is either the element from `nums1[i]` or `nums2[i]`. You want to fill `nums3` such that it contains the longest non-decreasing subarray possible; basically, you want `nums3` to have the longest sequence where elements from left to right are not in descending order. The desired output is the length of this longest non-decreasing subarray in `nums3`.

A non-decreasing subarray is a contiguous sequence of elements that go from left to right without decreasing in value. In other words, every element is greater than or equal to the element that comes before it.

Intuition

The intuition behind the solution involves dynamic programming, a method that solves problems by combining solutions to subproblems and using these solutions to construct an answer for the larger problem.

Since for each index `i`, you can choose either `nums1[i]` or `nums2[i]` for the `nums3` array, there can be many potential combinations to consider. However, we only need to keep track of the longest subarray lengths up to the current position for both potential choices. Here we define two states: `f` to represent the length of the longest non-decreasing subarray that ends with an element from `nums1`, and `g` to represent the length of the longest non-decreasing subarray that ends with an element from `nums2`.

To arrive at the solution, we start by initializing `f` and `g` to 1, as the longest subarray at the start can only be of length 1. As we iterate through each index starting from 1, we determine the possible lengths of `f` and `g` by checking if the current elements in `nums1` and `nums2` can extend the non-decreasing subarrays ending at the previous index.

For each element at index `i`, if `nums1[i]` is greater than or equal to the previous elements (`nums1[i-1]` or `nums2[i-1]`), we can extend the non-decreasing subarray ending with `nums1[i-1]` or `nums2[i-1]`. The same applies for `nums2[i]`. At each step, we update `f` and `g` to the longer of the current value of `f` or `g` and the length of the new non-decreasing subarray, which can be an extension of `f` or `g` from the previous step. Finally, the answer is the maximum value of `f` and `g` after traversing all elements.

This dynamic approach ensures that we do not need to evaluate all possible combinations of elements from `nums1` and `nums2`; instead, we can efficiently calculate the length of the longest non-decreasing subarray by considering each position's optimal choice based on the previous computations.

Solution Approach

The implementation of the solution makes use of a dynamic programming strategy. It involves tracking the potential longest non-decreasing subarray lengths dynamically as we iterate through the arrays. Here's a breakdown of how the algorithm works and the concepts it uses:

1. **Initialization of States**: We start with two variables, `f` and `g`, both set to 1 because, at the very beginning, the longest non-decreasing subarray that can be formed will just include the first element from either `nums1` or `nums2`. The variable `ans` is also set to 1, to track the length of the longest subarray found so far.
2. **Iteration through Arrays**: We loop through the arrays starting from index 1, as the 0th index is already considered in our initial state.
3. **State Transition**: For each index `i`, we compute two temporary variables `ff` and `gg` which represent the potential new values of `f` and `g` based on whether the current elements at index `i` can extend the longest non-decreasing subarray up to index `i - 1`. This involves four scenarios for updating `ff` and `gg`:
   - If `nums1[i]` is not smaller than `nums1[i - 1]`, the subarray ending with `nums1[i - 1]` can be extended, so we update `ff` to `max(ff, f + 1)`.
   - If `nums1[i]` is not smaller than `nums2[i - 1]`, the subarray ending with `nums2[i - 1]` can be extended, so we update `ff` to `max(ff, g + 1)`.
   - If `nums2[i]` is not smaller than `nums1[i - 1]`, the subarray ending with `nums1[i - 1]` can be extended, so we update `gg` to `max(gg, f + 1)`.
   - If `nums2[i]` is not smaller than `nums2[i - 1]`, the subarray ending with `nums2[i - 1]` can be extended, so we update `gg` to `max(gg, g + 1)`.
4. **State Update**: After calculating `ff` and `gg`, we set `f` and `g` to these new values, as they represent the updated longest non-decreasing subarray lengths ending at index `i` for `nums1` and `nums2` respectively.
5. **Answer Calculation**: With each iteration, the `ans` variable is updated to reflect the maximum value among `ans`, `f`, and `g`. This ensures that after traversing the entire array, `ans` holds the length of the longest non-decreasing subarray that can be formed.

The code uses a simple `for` loop for iteration and `max` function calls for comparison to execute this algorithm effectively. By only maintaining the necessary states, we optimize space complexity while ensuring that the program runs in linear time relative to the size of the given arrays.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Assume our input arrays are `nums1 = [3, 4, 2, 6]` and `nums2 = [1, 2, 3, 4]`, and we wish to construct `nums3` with the longest non-decreasing subarray.

Initially, we set `f = 1`, `g = 1`, and `ans = 1`, since the longest non-decreasing subarray at the beginning can only include one of the first elements.

Now we proceed to iterate through each index, starting from 1.

For `i = 1`:

- Our temporary variables `ff` and `gg` at the start of every iteration, they are set to zero.
- Since `nums1[1]` (4) is not smaller than `nums1[1 - 1]` (3), we can extend the subarray ending with `nums1[i - 1]`. Thus, `ff = max(ff, f + 1) => ff = max(0, 1 + 1) => ff = 2`.
- Since `nums1[1]` (4) is also not smaller than `nums2[1 - 1]` (1), we can extend the subarray ending with `nums2[i - 1]`. Thus, `ff = max(ff, g + 1) => ff = max(2, 1 + 1) => ff = 2`.
- `nums2[1]` (2) is not smaller than `nums1[1 - 1]` (3), so we cannot extend from `nums1[i - 1]` this time; `gg` remains the same.
- Since `nums2[1]` (2) is not smaller than `nums2[1 - 1]` (1), we can extend the subarray ending with `nums2[i - 1]`. Thus, `gg = max(gg, g + 1) => gg = max(0, 1 + 1) => gg = 2`.
- We now perform the state update, `f = ff` and `g = gg`, setting both `f` and `g` to 2.

Now, `ans` is updated to be the max of `ans`, `f`, and `g`, which is `max(1, 2, 2)`, so `ans` becomes 2.

Repeating this process for the rest of the indices:

For `i = 2`:

- `nums1[2]` (2) is smaller than `nums1[1 - 1]` (4), so `ff` does not update based on `f`.
- But `nums1[2]` (2) is not smaller than `nums2[1 - 1]` (2), so `ff = max(ff, g + 1) => ff = max(0, 2 + 1) => ff = 3`.
- `nums2[2]` (3) is not smaller than `nums1[1 - 1]` (4), so `gg` does not update based on `f`.
- `nums2[2]` (3) is larger than `nums2[1 - 1]` (2), so `gg = max(gg, g + 1) => gg = max(0, 2 + 1) => gg = 3`.
- Reset `f` and `g` to their new values `ff` and `gg`, respectively, both are now 3.
- `ans` updates to `max(ans, f, g) => ans = max(3, 3, 4) => ans = 4`.

For `i = 3`:

- `nums1[3]` (6) is larger than both `nums1[i - 1]` (2) and `nums2[i - 1]` (3), so `ff` becomes `max(0, f + 1)` and `max(0, g + 1)`, resulting in `ff = 4`.
- `nums2[3]` (4) is larger than `nums1[i - 1]` (2) but not `nums2[i - 1]` (3), so `gg` only updates from `f`, giving `gg = max(0, f + 1) => gg = 4`.
- State update puts `f = 4` and `g = 4`.
- `ans` updates to `max(ans, f, g) => ans = max(3, 4, 4) => ans = 4`.

In the end, the longest non-decreasing subarray that can be created (`nums3`) is of length 4, which is our answer. Thus, `nums3` can be `[3, 4, 3, 4]` or `[3, 4, 2, 4]`, both of which have the longest non-decreasing subarray of length 4.

Python Solution

```python
from typing import List

class Solution:
    def maxNonDecreasingLength(self, nums1: List[int], nums2: List[int]) -> int:
        # Get the length of the input lists
        n = len(nums1)

        # Initialize the lengths of non-decreasing sequences ending with the last element of nums1 and nums2
        max_length_end_nums1 = max_length_end_nums2 = 1

        # Initialize the answer with 1, since the smallest non-decreasing sequence has at least one element
        max_length = 1

        # Iterate through the lists starting from the second element
        for i in range(1, n):
            # Initialize new max lengths for the current iteration
            new_max_length_end_nums1 = new_max_length_end_nums2 = 1

            # Update the new_max_length_end_nums1 if the current nums1 element is not decreasing compared to the previous one
            if nums1[i] >= nums1[i - 1]:
                new_max_length_end_nums1 = max(new_max_length_end_nums1, max_length_end_nums1 + 1)

            # Update the new_max_length_end_nums1 if the current nums1 element is not decreasing compared to the previous nums2 elem
            if nums1[i] >= nums2[i - 1]:
                new_max_length_end_nums1 = max(new_max_length_end_nums1, max_length_end_nums2 + 1)

            # Update the new_max_length_end_nums2 if the current nums2 element is not decreasing compared to the previous nums2 elem
            if nums2[i] >= nums2[i - 1]:
                new_max_length_end_nums2 = max(new_max_length_end_nums2, max_length_end_nums2 + 1)

            # Update the new_max_length_end_nums2 if the current nums2 element is not decreasing compared to the previous nums1 elem
            if nums2[i] >= nums1[i - 1]:
                new_max_length_end_nums2 = max(new_max_length_end_nums2, max_length_end_nums1 + 1)

            # Update the lengths of non-decreasing sequences ending with the last element of nums1 and nums2
            max_length_end_nums1, max_length_end_nums2 = new_max_length_end_nums1, new_max_length_end_nums2

            # Update the global max_length if necessary
            max_length = max(max_length, new_max_length_end_nums1, max_length_end_nums2)

        # Return the length of the maximum non-decreasing sequence that can be made
        return max_length
```

Java Solution

```java
class Solution {

    /**
     * Computes the maximum length of a non-decreasing sequence from two arrays.
     * The non-decreasing sequence can switch between arrays at any point.
     *
     * @param nums1 The first input array.
     * @param nums2 The second input array.
     * @return The maximum length of non-decreasing sequence.
     */
    public int maxNonDecreasingLength(int[] nums1, int[] nums2) {
        int n = nums1.length; // Length of the input arrays.
        int maxLengthNums1 = 1; // Tracks max non-dec length ending with an element from nums1.
        int maxLengthNums2 = 1; // Tracks max non-dec length ending with an element from nums2.
        int ans = 1; // Stores the overall maximum non-decreasing length.

        // Loop through both arrays starting from the second element.
        for (int i = 1; i < n; ++i) {
            int tempMaxLengthNums1 = 1; // Temp variable for non-dec length ending with nums1[i].
            int tempMaxLengthNums2 = 1; // Temp variable for non-dec length ending with nums2[i].

            // If the current element in nums1 is not smaller than the previous one in nums1,
            // update the temporary maximum for nums1.
            if (nums1[i] >= nums1[i - 1]) {
                tempMaxLengthNums1 = Math.max(tempMaxLengthNums1, maxLengthNums1 + 1);
            }

            // If the current element in nums1 is not smaller than the previous one in nums2,
            // update the temporary maximum for nums1.
            if (nums1[i] >= nums2[i - 1]) {
                tempMaxLengthNums1 = Math.max(tempMaxLengthNums1, maxLengthNums2 + 1);
            }

            // If the current element in nums2 is not smaller than the previous one in nums2,
            // update the temporary maximum for nums2.
            if (nums2[i] >= nums2[i - 1]) {
                tempMaxLengthNums2 = Math.max(tempMaxLengthNums2, maxLengthNums2 + 1);
            }

            // If the current element in nums2 is not smaller than the previous one in nums1,
            // update the temporary maximum for nums2.
            if (nums2[i] >= nums1[i - 1]) {
                tempMaxLengthNums2 = Math.max(tempMaxLengthNums2, maxLengthNums1 + 1);
            }

            // Update the lengths with the new computed values
            maxLengthNums1 = tempMaxLengthNums1;
            maxLengthNums2 = tempMaxLengthNums2;

            // Calculate the maximum length considering both nums1 and nums2.
            ans = Math.max(ans, Math.max(maxLengthNums1, maxLengthNums2));
        }

        // Return the overall maximum non-decreasing length found.
        return ans;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Method to find the length of the longest non-decreasing subsequence that can be obtained
    // by choosing elements either from nums1 or nums2 at each step, under the constraint
    // that the subsequence remains non-decreasing.
    int maxNonDecreasingLength(vector<int>& nums1, vector<int>& nums2) {
        int n = nums1.size(); // Get the size of the input vectors

        // Initialize lengths for non-decreasing subsequences ending with nums1[i] and nums2[i]
        int lengthNums1 = 1, lengthNums2 = 1;

        // Initialize the answer with a subsequence of length 1
        int maxSequenceLength = 1;

        // Loop through the array to compute the maximum length of non-decreasing subsequence
        for (int i = 1; i < n; ++i) {
            // Temporary variables to hold possible new lengths for the current index
            int newLengthNums1 = 1, newLengthNums2 = 1;

            // If the current element of nums1 is not smaller than the previous element of nums1,
            // update the new length for nums1.
            if (nums1[i] >= nums1[i - 1]) {
                newLengthNums1 = max(newLengthNums1, lengthNums1 + 1);
            }

            // If the current element of nums1 is not smaller than the previous element of nums2,
            // update the new length for nums1. This indicates switching from nums2 to nums1.
            if (nums1[i] >= nums2[i - 1]) {
                newLengthNums1 = max(newLengthNums1, lengthNums2 + 1);
            }

            // If the current element of nums2 is not smaller than the previous element of nums2,
            // update the new length for nums2.
            if (nums2[i] >= nums2[i - 1]) {
                newLengthNums2 = max(newLengthNums2, lengthNums2 + 1);
            }

            // If the current element of nums2 is not smaller than the previous element of nums1,
            // update the new length for nums2. This indicates switching from nums1 to nums2.
            if (nums2[i] >= nums1[i - 1]) {
                newLengthNums2 = max(newLengthNums2, lengthNums1 + 1);
            }

            // Update the lengths with the new computed values
            lengthNums1 = newLengthNums1;
            lengthNums2 = newLengthNums2;

            // Update the maxSequenceLength considering the lengths of both nums1 and nums2
            maxSequenceLength = max(maxSequenceLength, max(lengthNums1, lengthNums2));
        }

        // Return the computed maximum length of the non-decreasing subsequence
        return maxSequenceLength;
    }
};
```

Typescript Solution

```typescript
/**
 * Calculates the maximum length of a non-decreasing sequence
 * that can be formed by elements taken from either nums1 or nums2
 * by making decisions while iterating in order.
 * @param nums1 - The first array of numbers.
 * @param nums2 - The second array of numbers.
 * @returns The maximum length of the non-decreasing sequence from nums1 or nums2.
 */
function maxNonDecreasingLength(nums1: number[], nums2: number[]): number {
    const n: number = nums1.length;
    // Initialize lengths of non-decreasing subsequences for nums1 and nums2
    let numsLength1: number = 1;
    let numsLength2: number = 1;
    // Initialize the global maximum length
    let maxLength: number = 1;

    // Iterate through the arrays to determine the non-decreasing subsequences
    for (let i = 1; i < n; ++i) {
        // Initialize next lengths to store the subsequence lengths at the current iteration
        let nextNums1Length: number = 1;
        let nextNums2Length: number = 1;

        // Check and update the non-decreasing subsequence for nums1
        if (nums1[i] >= nums1[i - 1]) {
            nextNums1Length = Math.max(nextNums1Length, numsLength1 + 1);
        }
        if (nums1[i] >= nums2[i - 1]) {
            nextNums1Length = Math.max(nextNums1Length, numsLength2 + 1);
        }

        // Check and update the non-decreasing subsequence length for nums2
        if (nums2[i] >= nums2[i - 1]) {
            nextNums2Length = Math.max(nextNums2Length, numsLength2 + 1);
        }
        if (nums2[i] >= nums1[i - 1]) {
            nextNums2Length = Math.max(nextNums2Length, numsLength1 + 1);
        }

        // Update the current lengths of the subsequences for nums1 and nums2
        numsLength1 = nextNums1Length;
        numsLength2 = nextNums2Length;

        // Update the global maximum length
        maxLength = Math.max(maxLength, numsLength1, numsLength2);
    }

    return maxLength;
}
```

Time and Space Complexity

The time complexity of this code is $O(n)$, where n is the length of the input arrays `nums1` and `nums2`. The code consists of a single loop that iterates from `1` to `n - 1`, performing a constant amount of work for each element with no nested loops, resulting in a linear time complexity.

The space complexity of the code is $O(1)$, as it uses a fixed number of integer variables (`f`, `g`, `ff`, `gg`, `ans`, and `n`). No additional space that grows with the input size is utilized, resulting in constant space complexity.