801. Minimum Swaps To Make Sequences Increasing

Hard Array Dynamic Programming Leetcode Link

Problem Description

In this problem, we have two integer arrays `nums1` and `nums2` of the same length. Our goal is to make both arrays strictly increasing by swapping elements at the same index in both arrays. To clarify, `nums1[i]` can be swapped with `nums2[i]` for any `i`. What we're trying to find is the minimum number of such swaps required to achieve our goal. An array is strictly increasing if each element is larger than the one before it. The challenge assures us that there is always a way to make the arrays strictly increasing through such swaps.

Intuition

To solve this problem, dynamic programming is employed to keep track of the minimum number of swaps needed. At each index `i`, we have two choices:

1. **Do not swap elements at index `i`**: We choose this if the current elements of `nums1` and `nums2` are both greater than the previous elements of their respective arrays, which keeps both arrays strictly increasing without a swap.
2. **Swap elements at index `i`**: If the current elements are not greater than the previous ones, we must swap to maintain the strictly increasing order. In some cases, even when it's not required to maintain the strictly increasing order, swapping might still minimize the total number of swaps.

We maintain two variables:

- `a`: The minimum number of swaps needed up to the current index `i` without swapping `nums1[i]` and `nums2[i]`.
- `b`: The minimum number of swaps needed if we swap `nums1[i]` and `nums2[i]`.

After considering whether a swap is required or beneficial at each index, and updating `a` and `b` accordingly, the solution returns the minimum of `a` and `b` after iterating through the arrays, which represents the minimum number of swaps necessary to make both `nums1` and `nums2` strictly increasing.

Solution Approach

The solution uses a dynamic programming approach to minimize the number of swaps required to make both arrays strictly increasing. Let's walk through the implementation details:

- We initialize two variables, `a` and `b`. Here, `a` represents the minimum number of swaps required if we do not swap at the current position `i`, and `b` is the minimum number of swaps required if we do swap at the current position.
- `a` is initially set to 0, because no swaps are performed at the start, and `b` is set to 1 since performing a swap at the first position would count as one operation.

The main loop of the algorithm starts at index `1` and goes until the end of the arrays as follows:

- For each index `i`, we first store the old values of `a` and `b` into temporary variables `x` and `y`. This is to prevent updating `a` and `b` simultaneously in the loop, which would give incorrect results, as the calculation of `b` may depend on the old value of `a`.
- There are conditions that determine if a swap is necessary or if we can move to the next index without a swap:
  - If `nums1[i-1] >= nums1[i]` or `nums2[i-1] >= nums2[i]`, then the arrays are not strictly increasing, and a swap is necessary. In this case, we set `a` to `y` (the old value of `b`) since we have to swap the previous position, and `b` to `x + 1` because we are performing a swap at the current position, which adds to the swap count.
  - If the arrays are already strictly increasing without a swap, `a` remains unchanged, and `b` becomes `y + 1` since we continue from the previous swap scenario.
- Additionally, if `nums1[i-1] < nums2[i]` and `nums2[i-1] < nums1[i]`, then swapping may yield a better result even if it's not necessary. In this case, we update `a` to the minimum of its current value and `y`, and `b` to the minimum of its current value and `x + 1`.

In the end, we return the minimum of `a` and `b`, as this would represent either performing no swap or swapping at the last index, whichever has the minimum swaps overall to make the arrays strictly increasing.

By using dynamic programming, we avoid re-computation for each index, evaluating the minimum number of swaps needed at each step. This approach ensures that the final complexity of the algorithm is O(n), where `n` is the length of the given arrays.

Example Walkthrough

Let's consider two small integer arrays `nums1` and `nums2`:

```
nums1: [1, 3, 5, 7]
nums2: [1, 2, 3, 4]
```

According to the problem, we want to make both arrays strictly increasing by swapping elements at the same index where necessary.

Initialization:

- No swaps at the start, so `a = 0`.
- If we swap at the first position, it would count as one swap, so `b = 1`.

We start our loop from index `1` and go up to the last index.

At index 1:

- Current elements are `nums1[1] = 3` and `nums2[1] = 2`.
- Previous elements are `nums1[0] = 1` and `nums2[0] = 1`.
- Arrays are already strictly increasing (`1 < 3` and `1 < 2`), so no swap needed here.
- We set `tempA = a` and `tempB = b`.
- Checking further, though, we see that swapping might be beneficial for future positions because `nums2[1] > nums1[0]` and `nums1[1] > nums2[0]`.
- We update `a` to the minimum of `a` and `tempB` which are both `0`.
- We update `b` to the minimum of `b` and `tempA + 1`, which is `1`.

Now our arrays look like this (no change because no swap was done):

```
nums1: [1, 3, 5, 7]
nums2: [1, 2, 3, 4]
```

At index 2:

- Current elements are `nums1[2] = 5` and `nums2[2] = 3`.
- Previous elements are `nums1[1] = 3` and `nums2[1] = 2`.
- We cannot move to the next index without a swap since `nums2[2]` is not greater than `nums2[1]`.
- Therefore, a swap is necessary for `nums2`.
- We again set `tempA = a` and `tempB = b` (here `tempA = 0` and `tempB = 1`).
- We set `a` to `tempB` since we are continuing from the previous decision to swap.
- We set `b` to `tempA + 1` which equals `1`.
- After swapping, our arrays looks like this:

```
nums1: [1, 3, 3, 7]
nums2: [1, 2, 5, 4]
```

- But now we have a problem as `nums1` is no longer strictly increasing. We must also swap at index 1 to correct this.

Result after also swapping at index 1:

```
nums1: [1, 2, 3, 7]
nums2: [1, 3, 5, 4]
```

At index 3:

- Arrays are `nums1: [1, 2, 3, 7]` and `nums2: [1, 3, 5, 4]`.
- Current elements are fine as `3 < 7` and `5 < 4` requires a swap.
- We set `tempA = a` and `tempB = b`.
- Since the arrays without the current elements being swapped are increasing, we do not update `a`.
- To minimize total swaps, we choose not to swap here, so `b = tempB + 1`.

Finally, our arrays look like this, which are both strictly increasing:

```
nums1: [1, 2, 3, 7]
nums2: [1, 3, 5, 4]
```

The minimum number of swaps required is `min(a, b)`. Given the values of `a = 0` and `b = 2`, the minimum number of swaps is `0`.

This small example illustrated the solution approach using dynamic programming, where we made both `nums1` and `nums2` strictly increasing while minimizing the total number of swaps necessary.

Python Solution

```python
from typing import List

class Solution:
    def minSwap(self, nums1: List[int], nums2: List[int]) -> int:
        # Initialize the number of swaps when not swapping or swapping the i-th element
        no_swap, swap = 0, 1

        # Iterate over arrays starting from the second element (index 1)
        for i in range(1, len(nums1)):
            no_swap_cost, swap_cost = no_swap, swap
            if nums1[i - 1] >= nums1[i] or nums2[i - 1] >= nums2[i]:
                # If the current pair or the previous pair is not strictly increasing,
                # we must swap this pair to make the sequence increasing
                no_swap, swap = swap_cost, no_swap_cost + 1
            else:
                # If it is strictly increasing, no need to swap the previous pair
                swap = swap_cost + 1  # We can choose to swap the current pair
                # If swapping makes both arrays increasing,
                # consider minimum swaps by either swapping or not swapping previous pairs
                if nums1[i - 1] < nums2[i] and nums2[i - 1] < nums1[i]:
                    no_swap, swap = min(no_swap, swap_cost), min(swap, no_swap_cost + 1)

        # Return the minimum number of swaps to make both arrays strictly increasings
        return min(no_swap, swap)
```

Java Solution

```java
class Solution {
    public int minSwap(int[] nums1, int[] nums2) {
        // Initialize the two states: 'swap' and 'notSwap'.
        // swap: min swaps needed when the last element is swapped.
        // notSwap: min swaps needed when the last element is not swapped.
        int notSwap = 0, swap = 1;

        // Iterate through each pair of elements starting from the second pair.
        for (int i = 1; i < nums1.length; ++i) {
            // 'prevNotSwap' and 'prevSwap' hold the previous state's values.
            int prevNotSwap = notSwap, prevSwap = swap;

            // Reset the values for the current state.
            notSwap = swap = nums1.length;

            // Check if the current and previous elements of both arrays are strictly increasing.
            if (nums1[i - 1] < nums1[i] && nums2[i - 1] < nums2[i]) {
                // If no swap is needed, carry forward the previous notSwap value.
                notSwap = prevNotSwap;
                // If we choose to swap the current elements, add 1 to the previous swap value.
                swap = prevSwap + 1;
            }

            // Check if swapping the previous elements makes the current pair strictly increasing.
            if (nums1[i - 1] < nums2[i] && nums2[i - 1] < nums1[i]) {
                // If so, notSwap can also be the minimum of itself and the previous swap value.
                notSwap = Math.min(notSwap, prevSwap);
                // Similarly, swap can also be the minimum of itself and prevNotSwap+1.
                swap = Math.min(swap, prevNotSwap + 1);
            }
        }

        // Return the minimum of the two final states.
        return Math.min(notSwap, swap);
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minSwap(vector<int>& nums1, vector<int>& nums2) {
        int noSwap = 0;     // Initialize noSwap, representing the minimum swaps needed without swapping the current element.
        int swap = 1;       // Initialize swap, representing the minimum swaps needed with swapping the current element.
        int numElements = nums1.size(); // Get the number of elements in the arrays.

        // Loop through the arrays starting from the second element
        for (int i = 1; i < numElements; ++i) {
            int prevNoSwap = noSwap; // Store the previous noSwap value
            int prevSwap = swap;     // Store the previous swap value

            // Reset noSwap and swap for the current iteration
            noSwap = numElements;
            swap = numElements;

            // Check if it is not possible to proceed without swapping (i.e. strictly increasing not maintained)
            if (nums1[i - 1] >= nums1[i] || nums2[i - 1] >= nums2[i]) {
                // If an element at index i or index i-1 must be swapped to remain strictly increasing:
                // Current noSwap takes the value of previous swap
                // Current swap takes the value of one plus the previous noSwap
                noSwap = prevSwap;
                swap = prevNoSwap + 1;
            } else {
                // If current configuration already strictly increasing without swapping
                // swap value should consider at least one more swap (previous swap + 1)
                swap = prevSwap + 1;
                // Check if arrays are both strictly increasing if we swapped the previous pair
                if (nums1[i - 1] < nums2[i] && nums2[i - 1] < nums1[i]) {
                    // Minimal swaps can be either previous noSwap or swap, take the minimum.
                    noSwap = min(noSwap, prevSwap);
                    // Similarly, current swap can also be minimum of it or noSwap plus one (accounting for a swap at the previous st
                    swap = min(swap, prevNoSwap + 1);
                }
            }
        }

        // Return the minimum of noSwap and swap for the last element,
        // it gives the minimum number of swaps to make both sequences strictly increasing.
        return min(noSwap, swap);
    }
};
```

Typescript Solution

```typescript
// Function to calculate the minimum number of swaps required
// to make two arrays strictly increasing
function minSwap(nums1: number[], nums2: number[]): number {
    let noSwap = 0; // Minimum swaps needed without swapping the current element
    let swap = 1; // Minimum swaps needed with swapping the current element
    const numElements = nums1.length; // Get the number of elements in the arrays

    // Loop through both arrays, starting from the second element
    for (let i = 1; i < numElements; ++i) {
        const prevNoSwap = noSwap; // Store the previous noSwap value
        const prevSwap = swap; // Store the previous swap value

        // Reset noSwap and swap for the current iteration to a large value
        noSwap = numElements;
        swap = numElements;

        // No change required if both arrays are already strictly increasing
        if (nums1[i - 1] < nums1[i] && nums2[i - 1] < nums2[i]) {
            // Current configuration is already strictly increasing without swapping
            noSwap = prevNoSwap;
            // Increment swap because we consider it's possible
            // that we may have swapped the previous pair
            swap = prevSwap + 1;
        }

        // If either pair can be swapped to maintain a strictly increasing order
        if (nums1[i - 1] < nums2[i] && nums2[i - 1] < nums1[i]) {
            // The new noSwap can potentially be the previous swap (if the previous was swapped)
            noSwap = Math.min(noSwap, prevSwap);
            // The new swap can potentially be the previous noSwap plus one
            swap = Math.min(swap, prevNoSwap + 1);
        }
    }

    // Return the smaller of noSwap and swap for the final element,
    // this will yield the minimum number of swaps to make both sequences strictly increasing.
    return Math.min(noSwap, swap);
}

// Example usage:
const nums1 = [1, 3, 5];
const nums2 = [1, 2, 3];
console.log(minSwap(nums1, nums2)); // Output will be minimum swaps required
```

Time and Space Complexity

The code provided is a dynamic programming solution designed to determine the minimum number of swaps needed to make two lists strictly increasing.

Time Complexity

To analyze the time complexity of this approach, we need to consider the operations performed with each iteration of the loop that runs `len(nums1)` times.

- Within each iteration, the algorithm performs constant-time comparisons and assignments, regardless of the size of the lists.
- The loop runs for every element of the two lists starting from index 1, which means it runs `n-1` times, where `n` is the number of elements in each list.

As the time taken per iteration does not depend on the size of the dataset, the overall time complexity of the algorithm is `O(n)`, where `n` is the length of the lists.

Space Complexity

For the space complexity:

- Two variables, `a` and `b`, are used to keep track of the minimum number of swaps when the ith element is not swapped and when it is swapped, respectively.
- Additionally, temporary variables `x` and `y` are used within the loop.

Since the space used does not increase with the size of the input lists, this algorithm has a constant space complexity, thus the space complexity is `O(1)`.