<https://leetcode.com/problems/minimum-swaps-to-make-sequences-increasing/description/>

You are given two integer arrays of the same length nums1 and nums2. In one operation, you are allowed to swap nums1[i] with nums2[i].

For example, if nums1 = [1,2,3,8], and nums2 = [5,6,7,4], you can swap the element at i = 3 to obtain nums1 = [1,2,3,4] and nums2 = [5,6,7,8].

Return the minimum number of needed operations to make nums1 and nums2 strictly increasing. The test cases are generated so that the given input always makes it possible.

An array arr is strictly increasing if and only if arr[0] < arr[1] < arr[2] < ... < arr[arr.length - 1].

**Example 1:**

**Input:** nums1 = [1,3,5,4], nums2 = [1,2,3,7]

**Output:** 1

**Explanation:**

Swap nums1[3] and nums2[3]. Then the sequences are:nums1 = [1, 3, 5, 7] and nums2 = [1, 2, 3, 4]which are both strictly increasing.

**Example 2:**

**Input:** nums1 = [0,3,5,8,9], nums2 = [2,1,4,6,9]

**Output:** 1

**Constraints:**

2 <= nums1.length <= 10^5

nums2.length == nums1.length

0 <= nums1[i], nums2[i] <= 2 \* 10^5

**Attempt 1: 2024-09-16**

**Solution 1: Native DFS (180 min)**

**Wrong Solution**

**java.lang.ArrayIndexOutOfBoundsException: Index -1 out of bounds for length 4**

Last Executed Input

nums1 = [1,3,5,4]

nums2 = [1,2,3,7]

class Solution {

    public int minSwap(int[] nums1, int[] nums2) {

        // Start the DFS from index 0, with no prior swaps (swapFlag = 0)

        return helper(nums1, nums2, 0, 0);

    }

    // DFS function that returns the minimum swaps starting at index `i`

    // swapFlag indicates whether the previous index was swapped

    // (0 = not swapped, 1 = swapped)

    private int helper(int[] nums1, int[] nums2, int i, int swapFlag) {

        // Base case: No more elements, no swaps needed

        if(i == nums1.length) {

            return 0;

        }

        // Only when i > 0 and decide to swap, the previous number comes

        // from the other array, if i == 0 or not decide to swap, the

        // previous number still comes from current array

        int prev\_num1 = (i > 0 && swapFlag == 1 ? nums2[i - 1] : nums1[i - 1]);

        int prev\_num2 = (i > 0 && swapFlag == 1 ? nums1[i - 1] : nums2[i - 1]);

        // No swap at index 'i' and match strictly increasing requirement:

        // arr is strictly increasing if and only if arr[0] < arr[1] < arr[2]

        // < ... < arr[arr.length - 1]

        int no\_swap = nums1.length; // At most we need swap all pairs

        if(i == 0 || (nums1[i] > prev\_num1 && nums2[i] > prev\_num2)) {

            no\_swap = helper(nums1, nums2, i + 1, 0);

        }

        // Swap at index 'i' and match strictly increasing requirement:

        // arr is strictly increasing if and only if arr[0] < arr[1] < arr[2]

        // < ... < arr[arr.length - 1]

        int swap = nums1.length; // At most we need swap all pairs

        if(i == 0 || (nums1[i] > prev\_num2 && nums2[i] > prev\_num1)) {

            swap = helper(nums1, nums2, i + 1, 1) + 1;

        }

        // Store the result in memo and return the minimum of both choices

        return Math.min(no\_swap, swap);

    }

}

**Fix the issue (TLE, 62/117)**

class Solution {

    public int minSwap(int[] nums1, int[] nums2) {

        // Start the DFS from index 0, with no prior swaps (swapFlag = 0)

        return helper(nums1, nums2, 0, 0);

    }

    // DFS function that returns the minimum swaps starting at index `i`

    // swapFlag indicates whether the previous index was swapped

    // (0 = not swapped, 1 = swapped)

    private int helper(int[] nums1, int[] nums2, int i, int swapFlag) {

        // Base case: No more elements, no swaps needed

        if(i == nums1.length) {

            return 0;

        }

        // If i == 0, since 0 <= nums1[i], nums2[i] <= 2 \* 10^5, the dummy

        // previous number can set to -1, its elegant to server the comparison

        // between nums1[0], nums2[0] with dummy previous number to drive

        // the call into second recursion level

        // Only when i > 0 and decide to swap, the previous number comes

        // from the other array, if not decide to swap, the previous

        // number still comes from current array

        int prev\_num1 = (i == 0 ? -1 : (swapFlag == 1 ? nums2[i - 1] : nums1[i - 1]));

        int prev\_num2 = (i == 0 ? -1 : (swapFlag == 1 ? nums1[i - 1] : nums2[i - 1]));

        // No swap at index 'i' and match strictly increasing requirement:

        // arr is strictly increasing if and only if arr[0] < arr[1] < arr[2]

        // < ... < arr[arr.length - 1]

        int no\_swap = nums1.length; // At most we need swap all pairs

        if(nums1[i] > prev\_num1 && nums2[i] > prev\_num2) {

            no\_swap = helper(nums1, nums2, i + 1, 0);

        }

        // Swap at index 'i' and match strictly increasing requirement:

        // arr is strictly increasing if and only if arr[0] < arr[1] < arr[2]

        // < ... < arr[arr.length - 1]

        int swap = nums1.length; // At most we need swap all pairs

        if(nums1[i] > prev\_num2 && nums2[i] > prev\_num1) {

            swap = helper(nums1, nums2, i + 1, 1) + 1;

        }

        // Store the result in memo and return the minimum of both choices

        return Math.min(no\_swap, swap);

    }

}

**Solution 2: DFS + Memoization (10 min)**

class Solution {

    public int minSwap(int[] nums1, int[] nums2) {

// The 2nd dimension as 2 reserve for 'swapFlag' only have swap, no swap two status

        Integer[][] memo = new Integer[nums1.length][2];

        // Start the DFS from index 0, with no prior swaps (swapFlag = 0)

        return helper(nums1, nums2, 0, 0, memo);

    }

    // DFS function that returns the minimum swaps starting at index `i`

    // swapFlag indicates whether the previous index was swapped

    // (0 = not swapped, 1 = swapped)

    private int helper(int[] nums1, int[] nums2, int i, int swapFlag, Integer[][] memo) {

        // Base case: No more elements, no swaps needed

        if(i == nums1.length) {

            return 0;

        }

        if(memo[i][swapFlag] != null) {

            return memo[i][swapFlag];

        }

        // If i == 0, since 0 <= nums1[i], nums2[i] <= 2 \* 10^5, the dummy

        // previous number can set to -1, its elegant to server the comparison

        // between nums1[0], nums2[0] with dummy previous number to drive

        // the call into second recursion level

        // Only when i > 0 and decide to swap, the previous number comes

        // from the other array, if not decide to swap, the previous

        // number still comes from current array

        int prev\_num1 = (i == 0 ? -1 : (swapFlag == 1 ? nums2[i - 1] : nums1[i - 1]));

        int prev\_num2 = (i == 0 ? -1 : (swapFlag == 1 ? nums1[i - 1] : nums2[i - 1]));

        // No swap at index 'i' and match strictly increasing requirement:

        // arr is strictly increasing if and only if arr[0] < arr[1] < arr[2]

        // < ... < arr[arr.length - 1]

        int no\_swap = nums1.length; // At most we need swap all pairs

        if(nums1[i] > prev\_num1 && nums2[i] > prev\_num2) {

            no\_swap = helper(nums1, nums2, i + 1, 0, memo);

        }

        // Swap at index 'i' and match strictly increasing requirement:

        // arr is strictly increasing if and only if arr[0] < arr[1] < arr[2]

        // < ... < arr[arr.length - 1]

        int swap = nums1.length; // At most we need swap all pairs

        if(nums1[i] > prev\_num2 && nums2[i] > prev\_num1) {

            swap = helper(nums1, nums2, i + 1, 1, memo) + 1;

        }

        // Store the result in memo and return the minimum of both choices

        return memo[i][swapFlag] = Math.min(no\_swap, swap);

    }

}

Time Complexity: O(n), where n is the length of the arrays.

Each state (combination of index i and swapFlag) is computed only once, and there are 2n states.

Space Complexity: O(n), due to the memoization table and recursion stack.

**Refer to chatGPT**

The error occurs because of the way I handled the prevA and prevB values at index 0. In the current implementation, I'm accessing A[i-1] and B[i-1] even when i == 0, which results in an **ArrayIndexOutOfBoundsException**.

To fix this, we need to handle the base case for index 0 more carefully, ensuring we don't access i-1 when i == 0.

**Corrected DFS Solution:**

class Solution {

private int[][] memo; // Memoization table

public int minSwap(int[] A, int[] B) {

int n = A.length;

memo = new int[n][2]; // 2 states: swapped or not

for (int[] row : memo) {

Arrays.fill(row, -1); // Initialize with -1 to signify uncomputed states

}

// Start the DFS from index 0, with no prior swaps (swapFlag = 0)

return dfs(A, B, 0, 0);

}

// DFS function that returns the minimum swaps starting at index `i`

// swapFlag indicates whether the previous index was swapped (0 = not swapped, 1 = swapped)

private int dfs(int[] A, int[] B, int i, int swapFlag) {

if (i == A.length) {

return 0; // Base case: No more elements, no swaps needed

}

// Check if we already computed this state

if (memo[i][swapFlag] != -1) {

return memo[i][swapFlag];

}

// Initialize variables for the current A[i-1] and B[i-1]

int prevA = (i == 0) ? Integer.MIN\_VALUE : (swapFlag == 1 ? B[i - 1] : A[i - 1]);

int prevB = (i == 0) ? Integer.MIN\_VALUE : (swapFlag == 1 ? A[i - 1] : B[i - 1]);

// Try without swapping at index `i`

int noSwap = Integer.MAX\_VALUE;

if (A[i] > prevA && B[i] > prevB) {

noSwap = dfs(A, B, i + 1, 0);

}

// Try with swapping at index `i`

int swap = Integer.MAX\_VALUE;

if (A[i] > prevB && B[i] > prevA) {

swap = 1 + dfs(A, B, i + 1, 1);

}

// Store the result in memo and return the minimum of both choices

return memo[i][swapFlag] = Math.min(noSwap, swap);

}

}

**Key Fixes:**

Handling Index **0 Special Case**:

When i == 0, we cannot check A[i-1] or B[i-1] because they don't exist. Instead, we initialize prevA and prevB to Integer.MIN\_VALUE at index 0 to guarantee that the first pair A[0] and B[0] is valid without needing swaps.

Transition Based on the Current and Previous Values:

If we swap or don't swap at index i, we compare the current A[i] and B[i] with the appropriate previous values (prevA, prevB), making sure we don't violate the strictly increasing condition.

**Solution 3: DP (180 min)**

class Solution {

    public int minSwap(int[] nums1, int[] nums2) {

        int len = nums1.length;

        // dp[i][0]: Minimum swaps needed up to index i without swapping the current index i.

        // dp[i][1]: Minimum swaps needed up to index i with swapping the current index i.

        int[][] dp = new int[len + 1][2];

        // Initialize the dp table with a large value (as maximum swaps possible is len)

        for (int[] a : dp) {

            Arrays.fill(a, len);

        }

        // Base case: When there are no more elements left, no swaps are needed

        dp[len][0] = 0;

        dp[len][1] = 0;

        // Traverse from the second last element to the first

        for (int i = len - 1; i >= 0; i--) {

            // If 'i' is the last element, use dummy previous values as

            // Integer.MAX\_VALUE (actually 2 \* 10^5 + 1 is enough)

            int prev\_num1 = (i == len - 1) ? Integer.MAX\_VALUE : nums1[i + 1];

            int prev\_num2 = (i == len - 1) ? Integer.MAX\_VALUE : nums2[i + 1];

            // Case 1: No swap at index 'i'

            if (nums1[i] < prev\_num1 && nums2[i] < prev\_num2) {

                dp[i][0] = Math.min(dp[i][0], dp[i + 1][0]);

                dp[i][1] = Math.min(dp[i][1], dp[i + 1][1] + 1);

            }

            // Case 2: Swap at index 'i'

            if (nums1[i] < prev\_num2 && nums2[i] < prev\_num1) {

                dp[i][0] = Math.min(dp[i][0], dp[i + 1][1]);

                dp[i][1] = Math.min(dp[i][1], dp[i + 1][0] + 1);

            }

        }

        // Return the minimum of no swap or swap at the first index

        return Math.min(dp[0][0], dp[0][1]);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to chatGPT**

To convert your DFS solution to a DP solution, the key is to maintain the same idea of tracking two states for each position: whether the current index is swapped or not. This state information is preserved in a 2D DP table (dp[i][swapFlag]) where i is the current index, and swapFlag represents whether a swap was made at the previous index (0 means no swap, 1 means swap).

**Key Differences Between DFS and DP:**

DFS: Recursive approach, exploring all possible paths.

DP: Iterative approach, storing results of subproblems to avoid redundant calculations.

**Conversion Plan:**

State Representation:

dp[i][0]: Minimum swaps needed up to index i without swapping the current index i.

dp[i][1]: Minimum swaps needed up to index i with swapping the current index i.

Base Case:

At the end (i == len), no more elements exist, so no swaps are needed. Initialize dp[len][0] = 0 and dp[len][1] = 0.

Transition:

For each index i from len - 1 to 0, check whether the sequences are strictly increasing both with and without swaps, updating the DP table accordingly.

Previous Values:

Since we are iterating from len - 1 to 0, the "previous values" are the values from the future (i.e., from i + 1).

Here’s the completed DP solution:

class Solution {

public int minSwap(int[] nums1, int[] nums2) {

int len = nums1.length;

int[][] dp = new int[len + 1][2];

// Initialize the dp table with a large value (as maximum swaps possible is len)

for (int[] a : dp) {

Arrays.fill(a, len);

}

// Base case: When there are no more elements left, no swaps are needed

dp[len][0] = 0;

dp[len][1] = 0;

// Traverse from the second last element to the first

for (int i = len - 1; i >= 0; i--) {

// If `i` is the last element, use dummy previous values (-1)

int prev\_num1 = (i == len - 1) ? Integer.MAX\_VALUE : nums1[i + 1];

int prev\_num2 = (i == len - 1) ? Integer.MAX\_VALUE : nums2[i + 1];

// Case 1: No swap at index `i`

if (nums1[i] < prev\_num1 && nums2[i] < prev\_num2) {

dp[i][0] = Math.min(dp[i][0], dp[i + 1][0]);

}

// Case 2: Swap at index `i`

if (nums1[i] < prev\_num2 && nums2[i] < prev\_num1) {

dp[i][0] = Math.min(dp[i][0], dp[i + 1][1]);

}

// Case 3: No swap at index `i`, but swap the next one

if (nums1[i] < prev\_num1 && nums2[i] < prev\_num2) {

dp[i][1] = Math.min(dp[i][1], dp[i + 1][1] + 1);

}

// Case 4: Swap at index `i`

if (nums1[i] < prev\_num2 && nums2[i] < prev\_num1) {

dp[i][1] = Math.min(dp[i][1], dp[i + 1][0] + 1);

}

}

// Return the minimum of no swap or swap at the first index

return Math.min(dp[0][0], dp[0][1]);

}

}

**Explanation:**

Initialization: We initialize dp[len][0] and dp[len][1] to 0 because no more swaps are needed when reaching the end of the arrays.

Loop from **len - 1 to 0**:

We check whether swapping or not swapping maintains the strictly increasing property, comparing current elements with the "future" elements (nums1[i + 1], nums2[i + 1]).

Transition:

If the arrays remain strictly increasing without a swap at index i, we transfer the value from dp[i + 1][0].

If swapping at index i works, we transfer the value from dp[i + 1][1] (plus 1 swap).

Final Answer: The answer is the minimum of dp[0][0] and dp[0][1] because we want to know whether swapping or not at the first index gives the fewest total swaps.

**Time and Space Complexity:**

**Time Complexity**: O(N), since we loop over the arrays once.

**Space Complexity**: O(N), due to the DP table (dp[len][2]).

**Another scheme developed from Native DFS to Optimal DP**

**Solution 1: Native DFS (30 min)**

class Solution {

    public int minSwap(int[] nums1, int[] nums2) {

        return Math.min(helper(nums1, nums2, nums1.length - 1, 0),

            helper(nums1, nums2, nums1.length - 1, 1));

    }

    private int helper(int[] nums1, int[] nums2, int i, int swap) {

        // Base case

        if(i == 0) {

            return swap;

        }

        int result = Integer.MAX\_VALUE;

        // No swap

        if(nums1[i] > nums1[i - 1] && nums2[i] > nums2[i - 1]) {

            result = helper(nums1, nums2, i - 1, swap);

        }

        // Swap

        if(nums1[i] > nums2[i - 1] && nums2[i] > nums1[i - 1]) {

            result = Math.min(result, helper(nums1, nums2, i - 1, 1 - swap));

        }

        // If swap == 1, meaning a swap was done at index i,

        // we add 1 to the result (res + 1) to account for that

        // swap. Otherwise, we return res as it is.

        return swap == 0 ? result : result + 1;

    }

}

Time complexiy: O(2^N)

Space complexity: O(N)

**Refer to**

<https://leetcode.com/problems/minimum-swaps-to-make-sequences-increasing/solutions/1277478/from-recursion-to-dp-4-solutions-java/>

**Approach**

There are two states possible,

Don't swap elements at the current index

Swap elements at the current index

We just have to find out which one gives the minimum number of swaps for the rest of the array. That is, we will compute answer for both the states. The answer for the current state is dependent on the relation between the element at the current index and the previous index.

If they are already in increasing order, then the state for the current index is applied to the previous index (that is, no swap remains no swap, swap remains swap). Else, the state for the current index is reversed for the previous index. But, what if swap and no swap both achieve the increasing order? In this case, we take the minimum of both states from the previous index.

**Implementations**

N = size of nums1/nums2 array

**1. Recursion**

Time complexiy: O(2^N)

Space complexity: O(N)

class Solution {

public int minSwap(int[] nums1, int[] nums2) {

return Math.min(recurse(nums1, nums2, nums1.length - 1, 0),

recurse(nums1, nums2, nums1.length - 1, 1));

}

private int recurse(int[] nums1, int[] nums2, int i, int swap) {

// base case

if (i == 0)

return swap;

// default is set as max

int res = Integer.MAX\_VALUE;

// if array is increasing without swapping

if (nums1[i] > nums1[i - 1] && nums2[i] > nums2[i - 1])

res = recurse(nums1, nums2, i - 1, swap);

// if array is increasing with swapping

if (nums1[i] > nums2[i - 1] && nums2[i] > nums1[i - 1])

res = Math.min(res,

recurse(nums1, nums2, i - 1, 1 - swap));

return swap == 0 ? res : res + 1;

}

}

**Refer to chatGPT**

This DFS solution for Leetcode 801 (minimum swaps to make sequences strictly increasing) uses a recursive approach to evaluate two possible scenarios at each index: whether to swap the current elements or not. Here’s a detailed explanation of how the solution works:

**Key Components:**

Recursive Function:

The recursive function recurse(int[] nums1, int[] nums2, int i, int swap) evaluates whether a swap is needed at the i-th index, starting from the end of the arrays (nums1 and nums2). The parameter swap indicates whether the i-th element has been swapped (swap = 1) or not (swap = 0).

Base Case:

if (i == 0)

return swap;

When the recursion reaches the first element (i == 0), the function returns swap. This is because if we are considering swapping at index 0, the recursion must end, and we return the result of whether index 0 is swapped or not. If swap = 1, it means the first element is swapped; otherwise, it’s not.

Recursive Transitions:

At each index i, we check whether swapping or not swapping keeps the arrays strictly increasing. Based on that, we explore two scenarios:

Scenario 1: No swap at index i:

if (nums1[i] > nums1[i - 1] && nums2[i] > nums2[i - 1])

res = recurse(nums1, nums2, i - 1, swap);

Condition: If the current elements nums1[i] and nums2[i] are greater than the previous elements (nums1[i - 1] and nums2[i - 1]), then we can maintain the strictly increasing order without swapping at index i.

**Action: Call the recursive function with the previous index i - 1 and pass the current swap status, since no swap was done at index i.**

Scenario 2: Swap at index i:

if (nums1[i] > nums2[i - 1] && nums2[i] > nums1[i - 1])

res = Math.min(res, recurse(nums1, nums2, i - 1, 1 - swap));

Condition: If swapping nums1[i] and nums2[i] still maintains the strictly increasing order (nums1[i] > nums2[i - 1] and nums2[i] > nums1[i - 1]), we consider swapping at index i.

**Action: Recursively call the function with i - 1 and the opposite swap value (1 - swap), as we are swapping the current index.**

Handling the Result:

return swap == 0 ? res : res + 1;

**After evaluating both scenarios (swapping or not swapping), the function returns the minimum number of swaps required. If swap == 1, meaning a swap was done at index i, we add 1 to the result (res + 1) to account for that swap. Otherwise, we return res as it is.**

**How It Works:**

The function begins by comparing the current nums1[i] and nums2[i] values with their previous values (nums1[i-1], nums2[i-1]) to determine whether the arrays remain strictly increasing without swapping.

If no swap is needed, the function explores the case where no swap is done. However, if a swap at the current index makes the arrays valid, it explores the option of swapping.

By recursing from the end of the arrays (i = len - 1) down to the first element (i = 0), the function ensures that all possible combinations of swaps and non-swaps are explored.

The goal is to minimize the total number of swaps required to keep both sequences strictly increasing.

**Why Two Initial Calls?**

return Math.min(recurse(nums1, nums2, nums1.length - 1, 0),

recurse(nums1, nums2, nums1.length - 1, 1));

The initial call to recurse is made twice:

Once with swap = 0, assuming the last element is not swapped.

Once with swap = 1, assuming the last element is swapped.

These two calls ensure that the function considers both scenarios (swapped or not swapped) for the last element, and the minimum number of swaps is taken from the two options.

**Summary:**

The solution uses recursion (DFS) to explore whether swapping at each index helps maintain a strictly increasing sequence in nums1 and nums2.

For each index, it checks both the "no swap" and "swap" cases, and the result is the minimum number of swaps required.

It starts from the last index and works backwards to index 0, ensuring that each possible swap combination is explored.

**Not very understand about below two statements:**

**1.Scenario 1: No swap at index i**

**Action: Call the recursive function with the previous index i - 1 and pass the current swap status, since no swap was done at index i.**

**2.Scenario 2: Swap at index i**

**Action: Recursively call the function with i - 1 and the opposite swap value (1 - swap), as we are swapping the current index.**

Let me clarify these two statements in more detail:

**Scenario 1: No swap at index i**

Condition: The condition checks if the elements in both arrays, nums1 and nums2, are in strictly increasing order without any swaps at index i:

if (nums1[i] > nums1[i - 1] && nums2[i] > nums2[i - 1])

This means that nums1[i] should be greater than nums1[i-1] and nums2[i] should be greater than nums2[i-1]. If this condition is true, then no swap is required at index i because the sequences are already strictly increasing.

Action: Since we are not swapping at index i, we call the recursive function for the previous index i - 1, while maintaining the same swap status.

If swap == 0 (indicating no swap happened at index i), we continue to the previous index (i - 1) without changing the swap status.

Here's how it works in the code:

res = recurse(nums1, nums2, i - 1, swap);

**This call essentially means: "Let’s now check the previous index i - 1 and continue with the current swap status (whether or not we swapped previously)."**

**Scenario 2: Swap at index i**

Condition: The condition checks if swapping the elements at index i can still maintain the strictly increasing order:

if (nums1[i] > nums2[i - 1] && nums2[i] > nums1[i - 1])

This condition is true if nums1[i] is greater than nums2[i-1] and nums2[i] is greater than nums1[i-1]. In other words, if swapping nums1[i] and nums2[i] at index i keeps the sequences in strictly increasing order, then this condition is valid.

Action: If the condition holds, we explore the scenario where we swap at index i. This means we now switch the swap flag for the recursive call. If swap == 0 (no swap at index i), the swap at index i would result in swap == 1 for index i-1. Conversely, if swap == 1, the next call will continue with swap == 0 for index i-1.

This is why the recursive call changes the swap status:

res = Math.min(res, recurse(nums1, nums2, i - 1, 1 - swap));

**Here, 1 - swap flips the swap status. If the current call has swap = 0 (no swap at i), the recursive call will pass swap = 1, indicating that a swap has been made at index i.**

**Summary of Actions:**

Scenario 1 (No Swap at **i)**: If the arrays are already strictly increasing, we move to index i-1 with the same swap status (swap is unchanged).

Scenario 2 (Swap at **i)**: If swapping at index i maintains the increasing order, we recurse for index i-1, but we **flip the swap status** (swap becomes 1 - swap) to indicate that a swap was performed at the current index i.

**Solution 2: DFS + Memoization (10 min)**

class Solution {

    public int minSwap(int[] nums1, int[] nums2) {

        Integer[][] memo = new Integer[nums1.length][2];

        return Math.min(helper(nums1, nums2, nums1.length - 1, 0, memo),

            helper(nums1, nums2, nums1.length - 1, 1, memo));

    }

    private int helper(int[] nums1, int[] nums2, int i, int swap, Integer[][] memo) {

        // Base case

        if(i == 0) {

            return swap;

        }

        if(memo[i][swap] != null) {

            return memo[i][swap];

        }

        int result = Integer.MAX\_VALUE;

        // No swap

        if(nums1[i] > nums1[i - 1] && nums2[i] > nums2[i - 1]) {

            result = helper(nums1, nums2, i - 1, swap, memo);

        }

        // Swap

        if(nums1[i] > nums2[i - 1] && nums2[i] > nums1[i - 1]) {

            result = Math.min(result, helper(nums1, nums2, i - 1, 1 - swap, memo));

        }

        // If swap == 1, meaning a swap was done at index i,

        // we add 1 to the result (res + 1) to account for that

        // swap. Otherwise, we return res as it is.

        return memo[i][swap] = (swap == 0 ? result : result + 1);

    }

}

Time complexiy: O(N)

Space complexity: O(N)

**Refer to**

<https://leetcode.com/problems/minimum-swaps-to-make-sequences-increasing/solutions/1277478/from-recursion-to-dp-4-solutions-java/>

**2. Memoization**

Time complexiy: O(N)

Space complexity: O(N)

class Solution {

public int minSwap(int[] nums1, int[] nums2) {

int n = nums1.length;

// initialize dp table

int[][] memo = new int[2][n];

Arrays.fill(memo[0], -1);

Arrays.fill(memo[1], -1);

memo[0][0] = 0;

memo[1][0] = 1;

return Math.min(recurse(nums1, nums2, n - 1, 0, memo),

recurse(nums1, nums2, n - 1, 1, memo));

}

private int recurse(int[] nums1, int[] nums2, int i, int swap, int[][] memo) {

//check dp table

if (memo[swap][i] != -1)

return memo[swap][i];

// initial value is set as max

int res = Integer.MAX\_VALUE;

// if array is increasing without swapping

if (nums1[i] > nums1[i - 1] && nums2[i] > nums2[i - 1])

res = recurse(nums1, nums2, i - 1, swap, memo);

// if array is increasing with swapping

if (nums1[i] > nums2[i - 1] && nums2[i] > nums1[i - 1])

res = Math.min(res,

recurse(nums1, nums2, i - 1, 1 - swap, memo));

memo[swap][i] = swap == 0 ? res : res + 1;

return memo[swap][i];

}

}

**Solution 3: DP (180 min, hard to figure out dp relation formula agaisnt Native DFS root logic)**

class Solution {

    public int minSwap(int[] nums1, int[] nums2) {

        int len = nums1.length;

        // dp[i][0]: Minimum swaps needed up to index i without swapping the current index i.

        // dp[i][1]: Minimum swaps needed up to index i with swapping the current index i.

        int[][] dp = new int[len][2];

        for(int[] a : dp) {

            Arrays.fill(a, Integer.MAX\_VALUE);

        }

        dp[0][0] = 0;

        dp[0][1] = 1;

        for(int i = 1; i < len; i++) {

            // No swap at index 'i'

            if(nums1[i] > nums1[i - 1] && nums2[i] > nums2[i - 1]) {

                dp[i][0] = dp[i - 1][0]; // (a)

                // Although no swap at index 'i' but swap happen

                // at index 'i - 1' cause '+ 1'

                dp[i][1] = dp[i - 1][1] + 1; // (b)

            }

            // Swap at index 'i'

            if(nums1[i] > nums2[i - 1] && nums2[i] > nums1[i - 1]) {

                dp[i][0] = Math.min(dp[i][0], dp[i - 1][1]); // (c)

                dp[i][1] = Math.min(dp[i][1], dp[i - 1][0] + 1); // (d)

            }

        }

        return Math.min(dp[len - 1][0], dp[len - 1][1]);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**It has below mapping relation between Native DFS, focus on 'swap' value change is the best way to demonstrate how the convertion happened**

**(a) comes from (1) -> (3)**

e.g helper(nums1, nums2, nums1.length - 1, 0) // (1)

^ = 0

-> helper(nums1, nums2, i - 1, swap); // (3)

^ = 0

it mapping to dp[i][0] = dp[i - 1][0]; // (a)

^ = 0 ^ = 0

**(b) comes from (2) -> (3)**

**(c) comes from (1) -> (4)**

e.g helper(nums1, nums2, nums1.length - 1, 0) // (1)

^ = 0

-> helper(nums1, nums2, i - 1, 1 - swap)); // (4)

^ = 1 - 0 = 1

it mapping to dp[i][0] = Math.min(dp[i][0], dp[i - 1][1]); // (c)

^ = 0 ^ = 1

**(d) comes from (2) -> (4)**

**Below is the reference Native DFS code**

class Solution {

    public int minSwap(int[] nums1, int[] nums2) {

        return Math.min(helper(nums1, nums2, nums1.length - 1, 0), // (1)

            helper(nums1, nums2, nums1.length - 1, 1)); // (2)

    }

    private int helper(int[] nums1, int[] nums2, int i, int swap) {

        // Base case

        if(i == 0) {

            return swap;

        }

        int result = Integer.MAX\_VALUE;

        // No swap

        if(nums1[i] > nums1[i - 1] && nums2[i] > nums2[i - 1]) {

            result = helper(nums1, nums2, i - 1, swap); // (3)

        }

        // Swap

        if(nums1[i] > nums2[i - 1] && nums2[i] > nums1[i - 1]) {

            result = Math.min(result, helper(nums1, nums2, i - 1, 1 - swap)); // (4)

        }

        // If swap == 1, meaning a swap was done at index i,

        // we add 1 to the result (res + 1) to account for that

        // swap. Otherwise, we return res as it is.

        return swap == 0 ? result : result + 1;

    }

}

**Refer to**

<https://leetcode.com/problems/minimum-swaps-to-make-sequences-increasing/solutions/1277478/from-recursion-to-dp-4-solutions-java/>

**3.Tabulation**

Time complexiy: O(N)

Space complexity: O(N)

class Solution {

public int minSwap(int[] nums1, int[] nums2) {

int n = nums1.length;

// initialize dp table

int[][] table = new int[2][n];

table[0][0] = 0;

table[1][0] = 1;

for (int i = 1; i < n; i++) {

// initial value

table[0][i] = Integer.MAX\_VALUE;

table[1][i] = Integer.MAX\_VALUE;

// if array is increasing without swapping

if (nums1[i] > nums1[i - 1] && nums2[i] > nums2[i - 1]) {

table[0][i] = table[0][i - 1];

table[1][i] = 1 + table[1][i - 1];

}

// if array is increasing with swapping

if (nums1[i] > nums2[i - 1] && nums2[i] > nums1[i - 1]) {

table[0][i] = Math.min(table[0][i], table[1][i - 1]);

table[1][i] = Math.min(table[1][i], 1 + table[0][i - 1]);

}

}

return Math.min(table[0][n - 1], table[1][n - 1]);

}

}

**Solution 4: Optimal DP (180 min)**

**Refer to**

<https://leetcode.com/problems/minimum-swaps-to-make-sequences-increasing/solutions/1277478/from-recursion-to-dp-4-solutions-java/>

**4.Tabulation with Constant Space**

Time complexiy: O(N)

Space complexity: O(1)

class Solution {

public int minSwap(int[] nums1, int[] nums2) {

int n = nums1.length;

// initialize current res

int curNoSwap = 0, curSwap = 1;

for (int i = 1; i < n; i++) {

// update previous values

int prevNoSwap = curNoSwap, prevSwap = curSwap;

// reset cur values

curNoSwap = Integer.MAX\_VALUE;

curSwap = Integer.MAX\_VALUE;

// if array is increasing without swapping

if (nums1[i] > nums1[i - 1] && nums2[i] > nums2[i - 1]) {

curNoSwap = prevNoSwap;

curSwap = 1 + prevSwap;

}

// if array is increasing with swapping

if (nums1[i] > nums2[i - 1] && nums2[i] > nums1[i - 1]) {

curNoSwap = Math.min(curNoSwap, prevSwap);

curSwap = Math.min(curSwap, 1 + prevNoSwap);

}

}

return Math.min(curNoSwap, curSwap);

}

}

**Refer to**

[L2111.Minimum Operations to Make the Array K-Increasing (Ref.L300,L1964)](note://WEBfdbb832ca68996dc86b298085010f587)