2934. Minimum Operations to Maximize Last Elements in Arrays

```
Medium
           <u>Array</u>
                     Enumeration
```

Problem Description

ability to perform operations on these arrays, where a single operation consists of swapping the values at the same index i in both arrays. The goal is to make the last element of each array the maximum value within that respective array. The problem asks to determine the minimum number of such operations required to achieve this goal for both arrays—or to establish that it is impossible to do so, in which case the result should be -1. To clarify:

You are given two integer arrays nums1 and nums2, both of the same length n and indexed starting from zero. You have the

nums2[n - 1] should be equal to the maximum value in nums2.

Intuition

You have to return the smallest number of operations needed to satisfy both conditions or -1 if it's not possible to satisfy them.

nums1[n - 1] should be equal to the maximum value in nums1.

First, the problem can be approached by considering two cases: swapping or not swapping the values of nums1[n - 1] and nums2[n - 1]. For each case, two scenarios can occur for any index i: either no swap is needed because both elements are

already in their correct place according to the maximum value constraint, or a swap is needed to move a larger value to the last

index of its respective array.

We can create a helper function f(x, y) that will count the number of operations (swaps) needed when x and y represent the last values of nums1 and nums2 respectively. As we loop through the arrays, if we encounter a pair where neither array can have their last value as the maximum without swapping with the other, we immediately know it is impossible to satisfy the condition, and we can return -1.

• If both values at the current index i are less than or equal to x and y respectively, no action is needed. • If one value is greater than x and can be swapped to be the new last value of nums1, and the other is greater than y and can be swapped to be the new last value of nums2, we perform a swap. • If neither of these conditions is satisfied, fulfilling the goal is not possible.

4. If a swap is needed, the count cnt is incremented.

The implementation makes use of:

nums1 = [1,3,5,4]

nums2 = [1,2,3,7]

0.

Python

class Solution:

The solution employs a greedy strategy:

- The implementation consists of a helper function f(x, y) that is responsible for determining the number of swaps required to
- make x the last value of nums1 and y the last value of nums2. This function iterates through the arrays, excluding their last elements, to check if the current condition meeting criteria is satisfied without swapping or if a swap is needed.
- Here are the steps in the algorithm: 1. Check each element pair from nums1 and nums2 (up to n-1) to see if they already satisfy the condition where nums1[i] <= x and nums2[i] <=

2. If the condition is not satisfied, check if making a swap would satisfy it, which means nums1[i] <= y and nums2[i] <= x after the swap. 3. If neither the current state nor a swap can satisfy the condition, it means it's impossible to achieve the goal, and f(x, y) will return -1.

у.

Solution Approach

swap for last elements). It stores the results in variables a and b. The final step in the algorithm is to evaluate these two outcomes: • If both a and b are -1, it means it's impossible to satisfy the conditions using either case, and the method returns -1. • If at least one of the cases has a non--1 result, the method returns the minimum number of operations needed to satisfy the conditions. Since

swapping the last elements counts as an operation, in case 2 (if b is not -1), 1 is added to b before comparing it to a.

• A greedy approach that incrementally solves the problem by deciding the best action (swap or no swap) for each element pair.

Iterative traversal of both arrays which efficiently enables us to check conditions and count swaps at the same time.

Following this, the minOperations() method calls f(x, y) twice with different parameters: once where x and y are the initial

last values of nums1 and nums2 respectively (case 1: no swap for last elements), and then with x and y swapped (case 2: with

Example Walkthrough Let's walk through a small example to illustrate the solution approach. Consider the following arrays:

be the maximum of nums2.

5 the last element of nums1 and keep 7 as the last element of nums2.

Index 0: Both 1 and 1 are less than 4 and 7. No swap needed.

Index 1: 3 is less than 4, and 2 is less than 7. No swap needed.

condition for nums1. After this swap, nums1 becomes [1,3,4,5].

def min operations(self, nums1: List[int], nums2: List[int]) -> int:

Otherwise, an operation (swap) is needed.

operations a = calculate operations(nums1[-1], nums2[-1])

operations_b = calculate_operations(nums2[-1], nums1[-1])

at the end between the last elements of nums1 and nums2.

print(result) # Output would depend on the logic provided by the algorithm

private int length; // Naming 'n' as 'length' for better readability.

int count = 0; // Initialize operation counter.

if (nums1[i] <= limit1 && nums2[i] <= limit2) {</pre>

if (!(nums1[i] <= limit2 && nums2[i] <= limit1)) {</pre>

return count; // Return the total operations counted.

int operationsA = countMinOperations(nums1.back(), nums2.back());

int operationsB = countMinOperations(nums2.back(), nums1.back());

// nums1 and nums2 are arrays we are trying to make compatible through operations.

// Helper function to count operations needed to make each pair compatible.

// x and v are the target values for nums1 and nums2 to be compatible with.

if (nums1[i] <= targetNums1 && nums2[i] <= targetNums2) {</pre>

if (!(nums1[i] <= targetNums2 && nums2[i] <= targetNums1)) {</pre>

// Call the helper function with the last elements of nums1 and nums2 as targets.

const countOperations = (targetNums1: number, targetNums2: number): number => {

const arrayLength = nums1.length; // Store the length of the arrays.

function minOperations(nums1: number[], nums2: number[]): number {

let operationCount = 0; // Initialize operation count.

for (let i = 0; i < arrayLength - 1; ++i) {</pre>

continue;

return -1;

for (int i = 0; i < size0fNums - 1; ++i) {

// A swap is counted as an operation.

continue;

return -1;

++count;

};

};

TypeScript

// Iterate over the elements of the vectors, excluding the last element.

return operationsA + operationsB == -2 ? -1 : min(operationsA, operationsB + 1);

// Function to find the minimum number of operations to make all elements pair—wise compatible.

// No operation needed if current elements are already less than or equal to targets.

// If swapping doesn't make the numbers compatible with the targets, return -1.

// If swapping cannot help to be within limits, operation is not possible.

// If both elements at index i are within the specified limits, no operation is needed.

// Use the lambda function to count the operations for the limits determined by the last elements of both vectors.

// If both results are -1, return -1 (operation isn't possible). Otherwise, return the minimum of the two, adjusting for the

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

// Method to calculate the minimum operations to make nums1 and nums2 equal.

operations count += 1

return operations_count

result = sol.min operations([1.2.3]. [4.5.6])

Define a helper function to calculate the number of operations needed.

and the last element of nums2 as the target for nums2, and vice-versa.

smallest number of operations needed is 0.

Solution Implementation

from typing import List

• A helper function to encapsulate the logic for checking and counting swaps needed.

1. We start with the helper function f(x, y), with x = 4 and y = 7, the last elements of nums1 and nums2 respectively. 2. Loop through the arrays up to n-1 (i.e., the third element), checking if a swap is necessary:

Both arrays have a length of n = 4. According to the problem, we want nums1[3] to be the maximum of nums1 and nums2[3] to

Initially, nums1[3] = 4, and nums2[3] = 7. The maximum values of nums1 and nums2 are 5 and 7, respectively. We need to make

4. Now, we call the helper function f(x, y) again with x = 7 and y = 4 to see if we should have swapped the last elements initially. • This time, the elements of nums1 and nums2 at index 3 already satisfy the condition. There is no need for any swaps. So in this case, our count cnt is 0.

Finally, we compare results a and b from both cases. Since the result from the second case b (without any initial swap) is the

minimum and is not -1, we add 1 to b because of the initial swap we considered. Therefore, b + 1 = 0 + 1 = 1. However, since

we did not need that initial swap, the smallest number of operations needed to satisfy the conditions is actually just b, which is

o Index 2: 5 is greater than 4 and needs to be the last element of nums1, and 3 is less than 7, so we can swap 5 and 4 to satisfy the

In conclusion, o swaps are required to make the last element of each array the maximum value within that respective array. We found that by not performing the swap on the last elements initially, our arrays were already in the desired state. Hence, the

3. Since we were able to make a swap that progresses towards our goal, we increment our count cnt by 1.

def calculate operations(target1: int, target2: int) -> int: operations count = 0 # Iterate through both lists except the last element. for elem1, elem2 in zip(nums1[:-1], nums2[:-1]): # If both elements are less than or equal to their respective targets, no operation is needed. if elem1 <= target1 and elem2 <= target2:</pre> continue # If it's impossible to make one less than or equal to the other's target, return -1. if not (elem1 <= target2 and elem2 <= target1):</pre> return -1

return -1 if operations_a == operations_b == -1 else min(operations_a, float('inf') if operations_b == -1 else operations_b +

Calculate the operations needed if we consider the last element of nums1 as the target for nums1,

If both calculations returned -1, there's no solution. Otherwise, take the minimum of the two

In the case where operations b is not -1, we need to add 1 to it because it symbolizes an extra swap

// Initialize length to be the length of nums1 array (assuming nums1 and nums2 are of same length).

// Calculate the minimum operations required by comparing last elements of both arrays in two ways.

int operationsFromNums1 = calculateOperations(nums1, nums2, nums1[length - 1], nums2[length - 1]);

int operationsFromNums2 = calculateOperations(nums1, nums2, nums2[length - 1], nums1[length - 1]);

Java class Solution {

length = nums1.length;

Example usage:

sol = Solution()

```
// If both calculations resulted in -1, return -1, else return the minimum of two calculations.
       // When comparing operationsFromNums2, we add 1 since we are swapping numbers from the other array.
        return (operationsFromNums1 == -1 && operationsFromNums2 == -1) ? -1
            : Math.min(operationsFromNums1, operationsFromNums2 + 1);
   // Helper method to calculate the operations needed to make each pair of elements in nums1 and nums2 ordered with respect to x ar
   private int calculateOperations(int[] nums1, int[] nums2, int x, int v) {
        int count = 0; // Counter to keep track of the number of operations.
       // Go through each pair of elements up to the second-to-last pair.
        for (int i = 0; i < length - 1; ++i) {
           // Continue if both elements are already in the correct order with respect to x and y.
            if (nums1[i] <= x && nums2[i] <= y) {</pre>
               continue;
           // If it's not possible to make elements in nums1 and nums2 in order by swapping,
           // return -1 indicating it's not possible to equalize the arrays.
           if (!(nums1[i] <= y && nums2[i] <= x)) {</pre>
                return -1;
            // Otherwise, a swap is required, increment the count of operations.
           ++count;
        return count; // Return the total number of operations needed.
class Solution {
public:
   int minOperations(vector<int>& nums1, vector<int>& nums2) {
        int sizeOfNums = nums1.size(); // Variable name changed from 'n' to 'sizeOfNums' for better readability.
       // Lambda function to calculate the minimum operations required when comparing the elements of nums1 and nums2.
       auto countMinOperations = [&](int limit1, int limit2) {
```

```
operationCount++; // Increment the operation count if we need to swap.
    return operationCount; // Return the total number of operations needed.
};
```

```
const operationsA = countOperations(nums1.at(-1), nums2.at(-1));
    const operationsB = countOperations(nums2.at(-1), nums1.at(-1));
    // If both cases result in -1, we return -1, meaning it's not possible to make arrays compatible.
    if (operationsA === -1 && operationsB === -1) {
       return -1;
    // If one of the operations counts is -1, return the other one.
    // Otherwise, return the minimum count of the two operations plus one
    // (which accounts for the last swap not counted by the helper function).
    Math.max(operationsA, operationsB) :
          Math.min(operationsA, operationsB + 1);
from typing import List
class Solution:
    def min operations(self, nums1: List[int], nums2: List[int]) -> int:
       # Define a helper function to calculate the number of operations needed.
       def calculate operations(target1: int, target2: int) -> int:
           operations count = 0
           # Iterate through both lists except the last element.
            for elem1, elem2 in zip(nums1[:-1], nums2[:-1]):
               # If both elements are less than or equal to their respective targets, no operation is needed.
               if elem1 <= target1 and elem2 <= target2:</pre>
                   continue
               # If it's impossible to make one less than or equal to the other's target, return -1.
               if not (elem1 <= target2 and elem2 <= target1):</pre>
                   return -1
               # Otherwise, an operation (swap) is needed.
               operations count += 1
           return operations_count
       # Calculate the operations needed if we consider the last element of nums1 as the target for nums1,
       # and the last element of nums2 as the target for nums2, and vice-versa.
       operations a = calculate operations(nums1[-1], nums2[-1])
       operations_b = calculate_operations(nums2[-1], nums1[-1])
       # If both calculations returned -1, there's no solution. Otherwise, take the minimum of the two
       # In the case where operations b is not -1, we need to add 1 to it because it symbolizes an extra swap
       # at the end between the last elements of nums1 and nums2.
       return -1 if operations_a == operations_b == -1 else min(operations_a, float('inf') if operations_b == -1 else operations_b +
 Example usage:
# sol = Solution()
# result = sol.min operations([1.2.3]. [4.5.6])
 print(result) # Output would depend on the logic provided by the algorithm
Time and Space Complexity
```

Time Complexity

function within minOperations.

Space Complexity

Function f: This function includes a for-loop that iterates over two arrays (nums1 and nums2). The loop runs for the length of the arrays minus one (n-1) times) because the [:-1] slice is used, which excludes the last element of each array. Inside the loop, there is a constant amount of work being done: a series of comparisons and a conditional increment of cnt. Therefore,

The time complexity of the code can be determined by looking at the two main components: the f function and the calls to this

the complexity of the f function is 0(n-1) which simplifies to 0(n). Calls within minOperations: The function f is called twice inside minOperations. Since each call has a time complexity of O(n), and they are not nested, this does not change the overall time complexity of the code, which remains O(n). Combining the parts, we maintain an overall time complexity of O(n) for the minOperations function.

Analyzing the space complexity, there are no additional data structures that grow with the size of the input being used. The

variables cnt, a, and b use a constant amount of space. The slices used in the zip function in f do not create new arrays; they

just create views of the original arrays, so they don't add to the space complexity. Therefore, the space complexity is 0(1), as it does not depend on the size of the input arrays.