870. Advantage Shuffle

Array

Two Pointers

Sorting

Problem Description

Medium

In the given problem, we are provided with two integer arrays nums1 and nums2 of the same length. The term advantage refers to the number of positions i at which the element in nums1 is greater than the corresponding element in nums2, meaning nums1[i] > nums2[i]. The goal is to rearrange nums1 in such a way that it maximizes the advantage with respect to nums2. This means we need to reorder the elements in nums1 so that as many elements as possible are greater than the corresponding elements in nums2

when they are compared index-wise.

ntuition

The intuition behind solving this problem involves sorting and greedy strategy. First, we sort nums1 because we want to arrange its elements in an increasing order to match them with the elements from nums2 efficiently. We also sort nums2, but since we need to create a resulting array that corresponds to the original indices of nums2, we track the original indices by creating tuples

(value, index). For the solution approach, we employ a two-pointer technique. We consider the smallest element in nums1 and try to match it with the smallest element in nums2. If the current element in nums1 does not exceed the smallest remaining element of nums2, it cannot

contribute to the advantage. In such a case, we assign it to the position of the largest element of nums2 where it's less likely to

On the other hand, if the current element in nums1 can surpass the smallest element in nums2, we place it in the result array at the corresponding index and move to the next element in both nums1 and nums2. This process is repeated until all elements in nums1 are placed into the result. This greedy approach ensures that we maximize the advantage by matching each 'nums1' element with the best possible counterpart in 'nums2'.

Solution Approach The implementation of the solution approach is based on a sorted array, greedy algorithm, and two-pointer technique. Here's a

Sorting nums1: We start by sorting nums1 in non-decreasing order, which allows us to consider the smallest elements first and match them against nums2.

step-by-step breakdown:

affect the advantage negatively.

Create and Sort tuple array for nums2: We create tuples containing the value and the original index from nums2 and sort this array. This sorting helps us to consider the elements of nums2 from the smallest to the largest, while remembering their

- original positions. Initialize the result array: We initialize an empty result array ans with the same length as nums1 and nums2, which will store the
- array. These pointers will be used to traverse the elements in the tuple array. Iterating over nums1 and placing elements into ans: • We iterate over each element v in nums1. For each element v, we look at the smallest yet-to-be-assigned element in nums2 (pointed to by i).

Two-pointer approach: We set up two pointers, i to point at the start and j to point at the end of the sorted nums 2 tuple

ans at the index of the largest yet-to-be-assigned element in nums2 (pointed to by j), and decrement j. o If v is greater than t[i][0], v can contribute to the advantage. We assign v to ans at the index of the smallest yet-to-be-assigned element in nums2 (pointed to by i), and increment i.

has been greedily arranged to maximize the advantage over nums2. We return ans as the final output.

o If v is less than or equal to t[i][0] (the smallest element in nums2), the element v cannot contribute to the advantage. We then assign v to

Returning the result: After iterating through all elements in nums1, the ans array now represents a permutation of nums1 that

nums2, thus achieving the maximum advantage. Data structures used include an array of tuples for tracking nums2 elements with

The algorithm uses sorting and greedy matching to ensure each element from nums1 is used optimally against an element in

their original indices, and an additional array for constructing the result.

nums1 that maximizes the number of indices at which nums1[i] is greater than nums2[i].

Initialize the result array (ans): We set ans as an empty array of the same length: [0, 0, 0, 0].

final permutation of nums1 maximizing the advantage.

Following the solution approach: **Sorting nums1**: We sort nums1 to [8, 12, 24, 32]. Create and Sort tuple array for nums2: We create tuple pairs of nums2 with their indices: [(13, 0), (25, 1), (32, 2), (11,

Two-pointer approach: We initialize two pointers: i starts at 0 and j starts at 3 (pointing at the first and last index of the

∘ We compare 8 from nums1 with 11 (the smallest element in nums2 tuple array). Since 8 is less than 11, it can't contribute to the advantage.

• Next, compare 24 from nums1 with 13 (new smallest in nums2). 24 is greater, so it can also contribute to the advantage. Place 24 at ans [0] and

Let's consider two example arrays nums1 = [12, 24, 8, 32] and nums2 = [13, 25, 32, 11]. We aim to find a permutation of

Iterating over nums1:

increment i to 2.

Solution Implementation

Python

maximum advantage over nums2.

sorted nums2 tuple array).

Example Walkthrough

• Now, compare 12 from nums1 with 11 from nums2 (current smallest). 12 is greater, so it can contribute to the advantage. Place 12 at ans [3] (index of current smallest in nums2) and increment i to 1.

def advantageCount(self, A: List[int], B: List[int]) -> List[int]:

Initialize the length variable for convenience and readability.

Iterate over the sorted list A and try to assign an advantage

If the current value in A is greater than the smallest unassigned value in B,

Sort the first list to efficiently assign elements

Initialize the answer list with placeholder zeros.

we can assign it as an advantage over B.

return result; // Return the advantage array

// Get the size of the input vectors

for (int i = 0; i < n; ++i) {

vector<pair<int, int>> nums2WithIndices;

nums2WithIndices.push_back({nums2[i], i});

// The function is intended to find an "advantage" permutation of nums1

vector<int> advantageCount(vector<int>& nums1, vector<int>& nums2) {

// such that for each element in nums2, there is a corresponding element in

// nums1 that is greater. The output is a permutation of nums1 that maximizes

// the number of elements in nums1 that are greater than elements in nums2 at

// Create a vector of pairs to hold elements from nums2 and their indices

// If current element is greater than the smallest element in nums2,

// Otherwise, assign it to the index where nums2 is the largest.

// assign it to the index where nums2 is smallest.

answerArray[indexArray[leftPointer]] = nums1[i];

answerArray[indexArray[rightPointer]] = nums1[i];

// Return the answer array where each element in nums1 has been placed

Create tuples of value and index from list B, then sort these tuples.

Initialize the length variable for convenience and readability.

Iterate over the sorted list A and try to assign an advantage

answer[sorted_B_with_indices[left_pointer][1]] = value

Otherwise, there is no advantage, so assign the value to the largest

remaining element in B to discard it as efficiently as possible.

if value > sorted_B_with_indices[left_pointer][0]:

This will allow us to compare elements in A with the sorted elements of B.

sorted_B_with_indices = sorted((value, index) for index, value in enumerate(B))

If the current value in A is greater than the smallest unassigned value in B,

// to maximise the number of elements in nums1 that are greater than

def advantageCount(self, A: List[int], B: List[int]) -> List[int]:

Sort the first list to efficiently assign elements

Initialize the answer list with placeholder zeros.

Initialize two pointers for the sorted B list.

we can assign it as an advantage over B.

print(solution.advantageCount([12,24,8,32], [13,25,32,11]))

left_pointer, right_pointer = 0, length - 1

left_pointer += 1

Example of using the class and method above:

if (nums1[i] > nums2[indexArray[leftPointer]]) {

leftPointer++;

rightPointer--;

// the element at the same index in nums2.

} else {

return answerArray;

from typing import List

A.sort()

length = len(B)

for value in A:

answer = [0] * length

class Solution:

Place 8 at ans [2] (index of largest nums2 which is 32) and decrement j to 2.

3)]. After sorting, we get [(11, 3), (13, 0), (25, 1), (32, 2)].

o Lastly, 32 from nums1 is compared with 25 (new smallest in nums2). 32 is greater and contributes to the advantage. Place 32 at ans [1] and increment i to 3. Returning the result: The final ans array is [24, 32, 8, 12], representing the permutation of nums1 that gives us the

By applying this solution approach to the given arrays, we successfully arranged nums1 ([24, 32, 8, 12]) to maximize the

advantage against nums2 ([13, 25, 32, 11]), resulting in an advantage at 3 positions: at indices 0, 1, and 3.

- from typing import List class Solution:
- # Create tuples of value and index from list B, then sort these tuples. # This will allow us to compare elements in A with the sorted elements of B. sorted_B_with_indices = sorted((value, index) for index, value in enumerate(B))

Initialize two pointers for the sorted B list. left_pointer, right_pointer = 0, length - 1

length = len(B)

for value in A:

answer = [0] * length

A.sort()

```
if value > sorted_B_with_indices[left_pointer][0]:
                answer[sorted_B_with_indices[left_pointer][1]] = value
                left_pointer += 1
            # Otherwise, there is no advantage, so assign the value to the largest
            # remaining element in B to discard it as efficiently as possible.
            else:
                answer[sorted_B_with_indices[right_pointer][1]] = value
                right_pointer -= 1
        return answer
# Example of using the class and method above:
# solution = Solution()
# print(solution.advantageCount([12,24,8,32], [13,25,32,11]))
Java
import java.util.Arrays;
class Solution {
    public int[] advantageCount(int[] A, int[] B) {
        int n = A.length; // Length of input arrays
        int[][] sortedBWithIndex = new int[n][2]; // Array to keep track of B's elements and their indices
       // Fill the array with pairs {value, index} for B
        for (int i = 0; i < n; ++i) {
            sortedBWithIndex[i] = new int[] {B[i], i};
       // Sort the array of pairs by their values (the values from B)
       Arrays.sort(sortedBWithIndex, (a, b) -> a[0] - b[0]);
       // Sort array A to efficiently find the advantage count
       Arrays.sort(A);
        int[] result = new int[n]; // Result array to store the advantage count
        int left = 0; // Pointer for the smallest value in A
        int right = n - 1; // Pointer for the largest value in A
       // Iterate through A to determine the advantage
        for (int value : A) {
           // If the current value is less than or equal to the smallest in sortedBWithIndex,
           // put the value at the end of result (because it doesn't have an advantage)
           // and decrease the right pointer
            if (value <= sortedBWithIndex[left][0]) {</pre>
                result[sortedBWithIndex[right--][1]] = value;
            } else {
                // If the current value has an advantage (is larger),
                // assign it to the corresponding index in the result array
                // and increase the left pointer
                result[sortedBWithIndex[left++][1]] = value;
```

C++

public:

#include <vector>

class Solution {

#include <algorithm>

using namespace std;

// the same index.

int n = nums1.size();

```
// Sort the nums2WithIndices based on the values of nums2
        sort(nums2WithIndices.begin(), nums2WithIndices.end());
       // Sort nums1 in ascending order
        sort(nums1.begin(), nums1.end());
       // Use a two-pointer approach to assign elements from nums1 to nums2
       // Start i from the beginning and j from the end
        int i = 0, j = n - 1;
        // The ans vector will store the "advantaged" permutation of nums1
       vector<int> ans(n);
        for (int num : nums1) {
            // If the current num in nums1 is less than or equal to the smallest
            // unprocessed num in nums2, then this num in nums1 cannot have advantage
           // over any unprocessed nums in nums2. So, assign it to the largest remaining
           // num in nums2 (by decreasing index "j").
           // If num is greater, assign it to the current smallest unprocessed num in
           // nums2 (by increasing index "i") for the advantage.
            if (num <= nums2WithIndices[i].first) {</pre>
                ans[nums2WithIndices[j--].second] = num;
            } else {
                ans[nums2WithIndices[i++].second] = num;
       // Return the final "advantaged" permutation
        return ans;
};
TypeScript
function advantageCount(nums1: number[], nums2: number[]): number[] {
    // Determine the length of the arrays.
    const length = nums1.length;
    // Create an index array for sorting the indices based on nums2 values.
    const indexArray = Array.from({ length }, (_, i) => i);
    indexArray.sort((i, j) => nums2[i] - nums2[j]);
    // Sort nums1 in ascending order to easily find the next greater element.
    nums1.sort((a, b) => a - b);
    // Initialize an answer array with zeroes to store results.
    const answerArray = new Array(length).fill(0);
    let leftPointer = 0;
    let rightPointer = length - 1;
    for (let i = 0; i < length; i++) {</pre>
```

```
else:
       answer[sorted_B_with_indices[right_pointer][1]] = value
       right_pointer -= 1
return answer
```

Time and Space Complexity

solution = Solution()

Time Complexity Sorting nums1: Sorting an array of size n using a comparison-based sort like Timsort (the default in Python) usually takes 0 (n

The given Python code sorts both arrays and then iterates through them to match elements in nums1 with elements in nums2 in a

way that optimizes the advantage condition. Here's the computational complexity analysis of the code provided:

log n) time.

Creating and sorting t: The list comprehension first iterates over nums2 to create a list of tuples where each tuple contains an element and its index. This operation takes O(n). The list t is then sorted, which again takes O(n log n) time.

Iterating and building the ans array: The main loop iterates over every element in nums1, which is n operations. Within each

- operation, it performs constant time checks and assignments, so this step is O(n). When combining these steps, the sorting operations dominate the complexity, so the total time complexity is 0(n log n).
- **Space Complexity**

The sorted list of tuples t: This requires O(n) space to store the elements of nums2 along with their indices.

- The answer list ans: This also requires O(n) space to store the final output.
- The temporary variables used for indexing (i, j, etc.): These are a constant number of extra space, 0(1). Adding these up, the total space complexity of the algorithm is O(n) (since the O(n) space for the ans array and for the list t is the
- significant factor, and the constant space is negligible).