870. Advantage Shuffle

Medium Greedy Array Two Pointers Sorting Leetcode Link

Problem Description

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.

Intuition

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 affect the advantage negatively.

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 step-by-step breakdown:

1. **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`.

2. **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.

3. **Initialize the result array**: We initialize an empty result array `ans` with the same length as `nums1` and `nums2`, which will store the final permutation of `nums1` maximizing the advantage.

4. **Two-pointer approach**: We set up two pointers, `i` to point at the start and `j` to point at the end of the sorted `nums2` tuple array. These pointers will be used to traverse the elements in the tuple array.

5. **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`).
   - 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 `ans` at the index of the largest yet-to-be-assigned element in `nums2` (pointed to by `j`), and decrement `j`.
   - 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`.
6. **Returning the result**: After iterating through all elements in `nums1`, the `ans` array now represents a permutation of `nums1` that has been greedily arranged to maximize the advantage over `nums2`. We return `ans` as the final output.

The algorithm uses sorting and greedy matching to ensure each element from `nums1` is used optimally against an element in `nums2`, thus achieving the maximum advantage. Data structures used include an array of tuples for tracking `nums2` elements with their original indices, and an additional array for constructing the result.

Example Walkthrough

Let's consider two example arrays `nums1 = [12, 24, 8, 32]` and `nums2 = [13, 25, 32, 11]`. We aim to find a permutation of `nums1` that maximizes the number of indices at which `nums1[i]` is greater than `nums2[i]`.

Following the solution approach:

1. **Sorting `nums1`**: We sort `nums1` to `[8, 12, 24, 32]`.

2. **Create and Sort tuple array for `nums2`**: We create tuple pairs of `nums2` with their indices: `[(13, 0), (25, 1), (32, 2), (11, 3)]`. After sorting, we get `[(11, 3), (13, 0), (25, 1), (32, 2)]`.

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

4. **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 sorted `nums2` tuple array).

5. **Iterating over `nums1`**:

   - 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. Place `8` at `ans[2]` (index of largest `nums2` which is 32) and decrement `j` to `2`.
   - 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`.
   - 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 increment `i` to `2`.
   - 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`.
6. **Returning the result**: The final `ans` array is `[24, 32, 8, 12]`, representing the permutation of `nums1` that gives us the maximum advantage over `nums2`.

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.

Python Solution

```python
from typing import List

class Solution:
    def advantageCount(self, A: List[int], B: List[int]) -> List[int]:
        # Sort the first list to efficiently assign elements
        A.sort()

        # 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 the length variable for convenience and readability.
        length = len(B)

        # Initialize the answer list with placeholder zeros.
        answer = [0] * length

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

        # Iterate over the sorted list A and try to assign an advantage
        for value in A:
            # If the current value in A is greater than the smallest unassigned value in B,
            # we can assign it as an advantage over B.
            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 Solution

```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]) {
                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;
            }
        }

        return result; // Return the advantage array
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    // The function is intended to find an "advantage" permutation of nums1
    // 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
    // the same index.
    vector<int> advantageCount(vector<int>& nums1, vector<int>& nums2) {
        // Get the size of the input vectors
        int n = nums1.size();

        // Create a vector of pairs to hold elements from nums2 and their indices
        vector<pair<int, int>> nums2WithIndices;
        for (int i = 0; i < n; ++i) {
            nums2WithIndices.push_back({nums2[i], i});
        }

        // 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) {
                ans[nums2WithIndices[j--].second] = num;
            } else {
                ans[nums2WithIndices[i++].second] = num;
            }
        }

        // Return the final "advantaged" permutation
        return ans;
    }
};

```

Typescript Solution

```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++) {
        // If current element is greater than the smallest element in nums2,
        // assign it to the index where nums2 is smallest.
        if (nums1[i] > nums2[indexArray[leftPointer]]) {
            answerArray[indexArray[leftPointer]] = nums1[i];
            leftPointer++;
        } else {
            // Otherwise, assign it to the index where nums2 is the largest.
            answerArray[indexArray[rightPointer]] = nums1[i];
            rightPointer--;
        }
    }

    // Return the answer array where each element in nums1 has been placed
    // to maximise the number of elements in nums1 that are greater than
    // the element at the same index in nums2.
    return answerArray;
}

```

Time and Space Complexity

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:

Time Complexity

1. Sorting `nums1`: Sorting an array of size `n` using a comparison-based sort like Timsort (the default in Python) usually takes `O(n log n)` time.

2. 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.

3. 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 `O(n log n)`.

Space Complexity

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

2. The answer list `ans`: This also requires `O(n)` space to store the final output.

3. The temporary variables used for indexing (`i`, `j`, etc.): These are a constant number of extra space, `O(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).