2499. Minimum Total Cost to Make Arrays Unequal

Problem Description

The problem provides two integer arrays `nums1` and `nums2`, both of the same length `n`. The goal is to make sure that for every element at any index `i`, `nums1[i]` is not equal to `nums2[i]`. To achieve this, it's possible to perform operations that involve swapping any two elements in `nums1`. The cost of a single such operation is the sum of the indices of the elements that are being swapped. The task is to find the minimum total cost to ensure that `nums1` and `nums2` have no matching elements at the same index. If this condition cannot be fulfilled, the function should return `-1`.

Intuition

The solution to this problem involves a few key observations. Firstly, since we only need to operate on `nums1` to make it different from `nums2`, we focus our attention on the indices where `nums1` and `nums2` have the same value.

The following steps are used to arrive at the solution approach:

1. Identify all the indices where `nums1[i]` equals `nums2[i]`, as we may need to swap these elements from `nums1` to make them different. Calculate the cost of these potential swaps and count how many times each number appears at the same index in both arrays.
2. Find out if there is a particular value leading the count, that is, if it has more matches than half the number of same values across `nums1` and `nums2`. If such a leading number is found, record the excess amount of times it leads (more than half).
3. Iterate through both arrays again and try to reduce the leading count by swapping non-leading values. Each swap will add to the total cost.
4. If at the end of this process, the leading count is brought down to zero (meaning we managed to make all the necessary swaps), return the sum of costs formed by the operations. If there are still excess matches and no more non-leading values can be swapped, it's not possible to separate the arrays fully, and thus, we return `-1`.

The code snippet provided uses this approach to systematically find the minimum cost required to ensure that `nums1` and `nums2` satisfy the required conditions.

Solution Approach

The given solution utilizes a Counter (a type of dictionary from the collections module in Python) to track duplicates between `nums1` and `nums2`. Here's a step-by-step breakdown of the implementation:

1. Initialize two counters: `ans` for calculating the accumulated cost and `same` for counting the number of identical elements at corresponding indices in `nums1` and `nums2`.
2. Loop through `nums1` and `nums2` using `enumerate` to get both the index `i` and the values `a` (from `nums1`) and `b` (from `nums2`):
   - Whenever `a` is equal to `b` (which is a situation that we want to avoid), increment the `same` counter, accumulate the index to the `ans` as the initial cost, and update the Counter object `cnt` with the value of `a`. The `cnt` dictionary will hold elements as keys and the number of times they need to be swapped as their values.
3. Search for a "leading" value in the `cnt` dictionary that has more than half of the number of indices where `nums1` and `nums2` are the same. This value would be the bottleneck in achieving our goal and needs special attention:
   - If such a value is found, calculate the margin `m` by which this value exceeds half of the duplicates and store it along with the leading value `lead`.
4. Go through the arrays another time and use the `m` value to determine if we have enough non-leading elements to swap:
   - For each pair `(a, b)` where `a` does not equal `b` and neither `a` nor `b` are the leading value, add the index `i` to the total cost and decrement `m`.
   - This step effectively swaps non-leading values to reduce the number of problematic indices where `nums1` and `nums2` would be equal.
5. If after the end of this process the margin `m` became zero, it would mean that we have successfully swapped all necessary elements, and `nums1[i] != nums2[i]` holds for all `i`. In this case, `ans` contains the minimum total cost and is returned.
6. If `m` is still greater than zero, it implies that we couldn't find enough non-leading elements to swap, making it impossible to make all elements at matching indices in `nums1` and `nums2` different. In that case, the function returns `-1`.

The algorithm applies a greedy approach to minimize costs, prioritizing swaps at the lowest indices. The Counter data structure is pivotal for this approach, allowing for efficient tracking and updating of the frequencies of values that need to be swapped.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider two arrays `nums1` and `nums2` of length 4 each:

```
nums1 = [1, 2, 3, 1]
nums2 = [1, 1, 2, 2]
```

We need to make sure that `nums1[i]` is not equal to `nums2[i]` for all indices `i`. Now let's walk through the solution approach step by step:

1. **Identify indexes where values match**: We find that `nums1[0]` equals `nums2[0]`, and `nums1[2]` equals `nums2[2]`. So we list the pairs (indices) that need attention: `[(0, 0), (2, 2)]`.
2. **Count duplicates and calculate cost**: For the pairs listed, we count how many times the same number appears at the same index in both arrays and calculate the initial cost. The number `1` appears twice, and the cost of swapping indices 0 and 2 would be `0 + 2 = 2`.
3. **Find the leading value**: Here, the number `1` is the leading value as it appears the maximum number of times at the same index. We calculate the margin `m` by which `1` exceeds half of the duplicates. Since we have 2 duplicates and both are `1`, it exceeds by `2 - 1 (half of 2) = 1`. Hence, `m = 1`.
4. **Search for non-leading values to swap**: We look for a pair `(a, b)` where `a` does not equal `b` and neither `a` nor `b` is the leading value `1`. We find such a pair at index `1`: `(2, 1)`. We perform the swap at the lowest index (which minimizes the cost) and swap `nums1[1]` with `nums1[0]`. This resolves one duplicate without adding to `m` since neither number is the leading value. Now `nums1` is `[1, 2, 3, 1]` after the swap.
5. **Update the counts and cost**: After the swap, we now have `nums1 = [2, 1, 3, 1]` and `nums2 = [1, 1, 2, 2]`. There's still a duplicate at index 2, but we now also have a spare `2` at index 0, which we can swap with the `3` in `nums1`. This swap costs `2` because we are swapping elements at indices 0 and 2.
6. **Check if all duplicates are fixed**: After the swap, `nums1` becomes `[3, 1, 2, 1]`. We see that `nums1[i] != nums2[i]` for all `i`. We have successfully eliminated all duplicates, and our total cost is `2` (initially calculated for the problematic pair (0, 0) + `2` for the swap of indices 0 and 2) = `4`.

Thus, the minimum total cost required to ensure that no element at index `i` in `nums1` is equal to the element at the same index in `nums2` for this example is `4`.

Python Solution

```python
from collections import Counter

class Solution:
    def minimumTotalCost(self, nums1: List[int], nums2: List[int]) -> int:
        # Initialize variables:
        # ans: Accumulated cost
        # num_same: Count of elements that are the same in both lists
        # num_same = accumulated_cost = 0
        # counter_to_store: Store frequency of elements that are the same
        # frequency_counter = Counter()
        # Calculate initial cost and count same elements
        for index, (elem1, elem2) in enumerate(zip(nums1, nums2)):
            if elem1 == elem2:
                num_same += 1
                accumulated_cost += index
                frequency_counter[elem1] += 1

        # Initialize variables to determine the leader element
        max_overlap = 0
        leader = 0
        # Find the element with more than half of the same occurrences
        for elem, freq in frequency_counter.items():
            if freq * 2 > num_same:
                max_overlap = freq * 2 - num_same
                leader = elem
                break

        # Iterate again to adjust the cost for changing elements avoiding
        # the leader element until max_overlap is reduced to zero
        for index, (elem1, elem2) in enumerate(zip(nums1, nums2)):
            if max_overlap and elem1 != elem2 and elem1 != leader and elem2 != leader:
                accumulated_cost += index
                max_overlap -= 1

        # If there is still an overlap, return -1 since it's impossible to fulfill the conditions
        # Otherwise, return the accumulated cost
        return -1 if max_overlap else accumulated_cost

```

Java Solution

```java
class Solution {
    public long minimumTotalCost(int[] nums1, int[] nums2) {
        long totalCost = 0; // Initialize the answer variable to store the total cost
        int sameValueCount = 0; // Counter for the number of same values at the same index
        int n = nums1.length; // Size of the input arrays
        int[] freqCount = new int[n + 1]; // Frequency array to count occurrences of each number

        // Loop through the arrays to find matches and calculate part of the cost
        for (int i = 0; i < n; ++i) {
            if (nums1[i] == nums2[i]) {
                totalCost += i; // Add the same at the same index, add the index to totalCost
                ++sameValueCount; // Increment the same element counter
                ++freqCount[nums1[i]]; // Increment frequency count for this number
            }
        }

        int tempCount = 0; // Temporary count for storing excess count of a number
        int leadingNumber = 0; // The number with the maximum excess count

        // Find the number with the maximum excess appearance
        for (int i = 0; i <= freqCount.length; ++i) {
            int excess = freqCount[i] * 2 - sameValueCount;
            if (excess > 0) {
                tempCount = excess; // Update tempCount if a number has excess appearances
                leadingNumber = i; // Update the leadingNumber
                break; // Break because we only need the first number with an excess appearances
            }
        }

        // Try to reduce the cost by replacing non-leading number pairs
        for (int i = 0; i < n; ++i) {
            // Check if the current index is not a matching leading number
            if (tempCount > 0 && nums1[i] != nums2[i] && nums1[i] != leadingNumber && nums2[i] != leadingNumber) {
                totalCost += i; // Increment the total cost with the index value
                --tempCount; // Decrement the count for the number of swaps left
            }
        }

        // If there are still swaps left after traversing, there's no solution
        return tempCount > 0 ? -1 : totalCost;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    long long minimumTotalCost(vector<int>& nums1, vector<int>& nums2) {
        long long totalCost = 0; // Variable to store the total cost.
        int sameElementCount = 0; // Variable to count the number of elements that are the same in nums1 and nums2.
        int n = nums1.size(); // Size of the input arrays.
        int countArray[n + 1]; // Array to count the occurrence of each number.
        memset(countArray, 0, sizeof countArray); // Initialize the countArray with zeros.

        // Loop through nums1 and nums2 to calculate the initial cost and count same elements.
        for (int i = 0; i < n; ++i) {
            if (nums1[i] == nums2[i]) { // If the elements at index i are the same,
                totalCost += i; // Add the index to the total cost.
                ++sameElementCount; // Increment the same element count.
                ++countArray[nums1[i]]; // Increment the count of this element in countArray.
            }
        }

        // Variables to store the maximum lead and the value with that lead.
        int maxLead = 0, leadValue = 0;

        // Find the value with the max lead.
        for (int i = 0; i <= n; ++i) {
            int lead = countArray[i] * 2 - sameElementCount; // Calc. lead which is count*2 - sameElementCount.
            if (lead > 0) {
                maxLead = lead;  // Update max lead.
                leadValue = i;   // Update lead value.
                break; // Exit the loop since we found the value.
            }
        }

        // Loop to potentially add more to total cost based on numbers that do not have the leading count.
        for (int i = 0; i < n; ++i) {
            if (maxLead > 0 && nums1[i] != nums2[i] && nums1[i] != leadValue && nums2[i] != leadValue) {
                // Add the index to total cost if both numbers are not equal to the lead value.
                totalCost += i;
                --maxLead; // Decrease the lead as we've handled one index.
            }
        }

        // If maxLead greater than zero, we could not match all elements, return -1. Otherwise, return total cost.
        return maxLead > 0 ? -1 : totalCost;
    }
};
```

Typescript Solution

```typescript
function minimumTotalCost(nums1: number[], nums2: number[]): number {
    let totalCost: number = 0; // Variable to store the total cost.
    let sameElementCount: number = 0; // Variable to count the number of elements that are the same in nums1 and nums2.
    const n: number = nums1.length; // Size of the input arrays.
    let countArray: number[] = new Array(n + 1).fill(0); // Array to count the occurrence of each number with initial zeros.

    // Loop through nums1 and nums2 to calculate the initial cost and count same elements.
    for (let i = 0; i < n; ++i) {
        if (nums1[i] === nums2[i]) { // If the elements at index i are the same,
            totalCost += i; // Add the index to the total cost.
            ++sameElementCount; // Increment the same element count.
            ++countArray[nums1[i]]; // Increment the count of this element in countArray.
        }
    }

    // Variables to store the maximum lead and the value with that lead.
    let maxLead: number = 0;
    let leadValue: number = 0;

    // Find the value with the maximum lead.
    for (let i = 0; i <= n; ++i) {
        let lead: number = countArray[i] * 2 - sameElementCount; // Calculate lead which is count*2 - sameElementCount.
        if (lead > 0) {
            maxLead = lead;  // Update max lead.
            leadValue = i;   // Update lead value.
            break; // Exit the loop since we found the value.
        }
    }

    // Loop to potentially add more to total cost based on numbers that do not have the leading count.
    for (let i = 0; i < n; ++i) {
        if (maxLead > 0 && nums1[i] !== nums2[i] && nums1[i] !== leadValue && nums2[i] !== leadValue) {
            // Add the index to total cost if both numbers are not equal to the lead value.
            totalCost += i;
            --maxLead; // Decrease the lead as we've handled one index.
        }
    }

    // If maxLead is greater than zero, we could not match all elements, return -1. Otherwise, return the total cost.
    return maxLead > 0 ? -1 : totalCost;
}
```

Time and Space Complexity

The given code block consists of two separate for-loops that iterate over the provided inputs `nums1` and `nums2`, as well as a loop over the `Counter` dictionary `cnt`. None of these loops are nested.

Time Complexity:

- The first for-loop runs through all elements in `nums1` and `nums2`, contributing to a time complexity of `O(N)`, where `N` is the length of the lists.
- The second for-loop iterates over the `Counter` dictionary `cnt`. The maximum size of `cnt` is bounded by the number of unique elements in `nums1` since only elements that are equal in both `nums1` and `nums2` are counted. In the worst case, all elements are the same, so this is also `O(N)` but happens only once. Therefore, it doesn't change the overall complexity.
- The third for-loop again processes each element, adding to another `O(N)` time complexity.

Combining all the loops, the time complexity remains linear, therefore the total time complexity is `O(N)` where `N` is the size of the input arrays.

Space Complexity:

- The `Counter` object `cnt` stores the counts of the numbers that are the same in `nums1` and `nums2`. It will at most have `N` entries (since, in the worst case, all numbers in `nums1` and `nums2` will be the same). This gives us `O(N)` space complexity.
- There are constant space usages for the `ans`, `same`, `n`, and `lead` variables.

Thus, the final space complexity is `O(N)` accounting for the `Counter` object. All other variables occupy constant space, contributing `O(1)`.

Therefore, the overall space complexity of the code is `O(N)`.