2541. Minimum Operations to Make Array Equal II

Medium Greedy Array Math Leetcode Link

Problem Description

In this problem, we are working with two integer arrays, `nums1` and `nums2`, both of the same length `n`. We're given an integer `k` which we can use in a specific operation that we can perform on `nums1` as many times as needed. An operation consists of picking two indices, `i` and `j`, in `nums1` and adding `k` to the value at index `i` while subtracting `k` from the value at index `j`. Our goal is to make the elements of `nums1` match the corresponding elements in `nums2` using the least number of these operations. If it is impossible to make the arrays equal, we must return `-1`.

To understand whether it's possible to make `nums1` equal to `nums2`, we should consider that each operation effectively transfers `k` units of value from one element of `nums1` to another. Therefore, if the difference between any pair of corresponding elements in `nums1` and `nums2` is not a multiple of `k`, we cannot make the elements equal through the allowed operation, and we should return `-1`. If all differences are multiples of `k`, we can then calculate the minimal number of operations required by summing the absolute values of the quotients of differences by `k`.

Intuition

The intuition behind the solution is to first iterate over each corresponding pair of elements between `nums1` and `nums2`. We calculate the difference between the elements of `nums1` and `nums2`. If `k` is zero, we immediately know that unless all elements are already equal, it's impossible to make them equal as no operation can modify `nums1`. If `k` is nonzero, any difference that isn't a multiple of `k` would make it impossible for the two corresponding elements to ever match, so we return `-1`.

If a difference is a multiple of `k`, this multiple represents how many operations of magnitude `k` are needed to equalize this pair of elements. We accumulate this count in an `ans` variable.

Additionally, we keep track of the net effect of all operations in a variable `x`. This is because each operation has two parts: incrementing one element and decrementing another. So, if `x` is nonzero after considering all pairs, it means we've either incremented or decremented too much and can't balance this out with further operations. If `x` equals zero, it means that for every increment there is a corresponding decrement, and `nums1` can be made equal to `nums2`.

However, since every operation affects two elements (increments one and decrements another), the total number of operations required is half the sum of absolute values of all quoted differences, as each operation is counted twice in `ans`. Hence, we return `ans // 2`, but only if `x` equals zero, which ensures that all increments and decrements are paired.

Solution Approach

The solution follows a straightforward approach where it iterates through the two lists simultaneously using the built-in `zip` function. For each pair of elements `(a, b)` from `nums1` and `nums2` respectively, it performs several checks and calculations.

Here's a step-by-step breakdown of the implementation:

- Initialize `ans` and `x` to `0`. `ans` will hold the total number of operations required, while `x` will track the net effect of all those operations.
- Iterate over `nums1` and `nums2` in tandem using `zip(nums1, nums2)`.
- For each pair `(a, b)`:
  - Check if `k` is `0`. If it is and `a != b`, return `-1` immediately because no operations can be performed to change the values.
  - Calculate the difference `a - b` and check if it is a multiple of `k`. This is done by checking if `(a - b) % k` equals `0`. If it does not, return `-1` because it's impossible to make `a` equal to `b` through the allowed operation.
  - Otherwise, compute the quotient `y = (a - b) // k`, which represents how many operations of magnitude `k` are needed to equalize `a` and `b`. We need to consider the absolute value of `y` to add to `ans` because operations can be incrementing or decrementing, and we're counting total operations needed regardless of direction.
  - Add `y` to the net effect tracker `x`. An increment in one element will be a positive `y`, and a decrement will be a negative `y`, so the net effect after all pairs are considered should balance to zero.
- After the loop, check if `x` is nonzero. If it is, return `-1` because the net effect hasn't balanced out, implying that not all increments have a corresponding decrement.
- If `x` is zero, return `ans // 2`. Each operation affects two elements of `nums1`, so every actual operation is counted twice in `ans`.

This solution approach uses simple mathematical operations and control structures to determine the minimum number of operations. It leverages the fact that each allowed operation can be represented mathematically as an equation and that the collective effect of these operations must balance for equality to be possible.

Example Walkthrough

Let's go through a small example to illustrate the solution approach.

Suppose we have the following input:

- `nums1 = [1, 3, 5]`
- `nums2 = [5, 1, 3]`
- `k = 2`

We want to make each element in `nums1` match the corresponding element in `nums2` using the smallest number of operations.

Following the solution approach:

1. Initialize `ans` and `x` to `0`. These will track the number of operations and the net effect, respectively.
2. Iterate over the pairs `(1, 5)`, `(3, 1)`, and `(5, 3)` from `nums1` and `nums2`.
3. Starting with the first pair `(1, 5)`:
   - The difference `1 - 5` is `-4`, which is a multiple of `k` (2).
   - The quotient `y` is `-4 // 2 = -2`. We take the absolute value `|y| = 2` and add it to `ans`, so now `ans = 2`.
   - We add `y` (which is `-2`) to `x` to track the net effect, now `x = -2`.
4. Next, the pair `(3, 1)`:
   - The difference `3 - 1` is `2`, which is a multiple of `k`.
   - The quotient `y` is `2 // 2 = 1`. We add the absolute value `|y| = 1` to `ans`, so now `ans = 3`.
   - We add `y` (which is `1`) to `x`, making the net effect now `x = -1`.
5. Finally, the pair `(5, 3)`:
   - The difference `5 - 3` is `2`, which is also a multiple of `k`.
   - The quotient `y` is `2 // 2 = 1`. We add the absolute value `|y| = 1` to `ans`, now `ans = 4`.
   - We add `y` to `x`, and as `x` was previously `-1` and `y` is `1`, the net effect balances out, `x = 0`.
6. Since `x` is `0`, all increments have matching decrements. We can therefore equalize `nums1` to `nums2`.
7. Our answer is `ans // 2 = 4 // 2 = 2`. Hence, the minimum number of operations required is `2`.

This example clearly demonstrated that we need to perform two operations of magnitude `k` (`2` in this case) to make `nums1` equal to `nums2`. The steps as outlined in the solution approach guide us through a process of calculating the necessary number of operations or determining the impossibility of equalizing the two arrays.

Python Solution

```python
from typing import List

class Solution:
    def minOperations(self, nums1: List[int], nums2: List[int], k: int) -> int:
        # Initialize the number of operations required and the difference accumulator
        num_operations = total_difference = 0

        # Iterate over pairs of elements from nums1 and nums2
        for num1, num2 in zip(nums1, nums2):
            # If k is 0, no operations can make a difference, we need to check if arrays are same
            if k == 0:
                if num1 != num2:
                    return -1  # Arrays differ and we cannot perform operations
                continue  # Arrays are same up to now, continue checking

            # If the difference between num1 and num2 is not divisible by k, return -1
            if (num1 - num2) % k:
                return -1

            # Calculate how many times we need to add or subtract k
            difference = (num1 - num2) // k
            # Increment the number of operations by the absolute value of difference
            num_operations += abs(difference)
            # Update the total difference
            total_difference += difference

        # If the total difference is not zero, return -1 since it is not possible to equalize
        # Using the // 2 because each pair's operations partially cancel each other out
        return -1 if total_difference else num_operations // 2

```

Java Solution

```java
class Solution {

    // This method calculates the minimum number of operations needed to make elements in two arrays equal
    // where each operation consists of adding or subtracting a number 'k' from an element in nums1 array.
    public long minOperations(int[] nums1, int[] nums2, int k) {

        // 'totalOperations' will store the total number of operations performed
        long totalOperations = 0;

        // 'netChanges' will accumulate the net changes made to the nums1 array elements
        long netChanges = 0;

        // Iterate over elements of both arrays
        for (int i = 0; i < nums1.length; ++i) {
            int num1 = nums1[i]; // Element from the nums1 array
            int num2 = nums2[i]; // Corresponding element from the nums2 array

            // If 'k' is zero, it's impossible to perform operations, so we check if elements are already equal
            if (k == 0) {
                if (num1 != num2) {
                    return -1; // If any pair of elements is not equal, return -1 to indicate no solution
                }
                continue; // Skip to the next pair of elements since no operation is needed
            }

            // Check if the difference between num1 and num2 is divisible by 'k'
            if ((num1 - num2) % k != 0) {
                return -1; // If it's not, the operation cannot be performed so return -1
            }

            // Calculate the number of operations needed for the current pair of elements
            int operationsNeeded = (num1 - num2) / k;

            // Increment 'totalOperations' by the absolute number of operations needed
            totalOperations += Math.abs(operationsNeeded);

            // Update 'netChanges' by adding operations needed (this could be negative or positive)
            netChanges += operationsNeeded;
        }

        // If the net effect of all operations is zero, we can pair operations to cancel each other out
        // Therefore, we return half of 'totalOperations', since each operation can be paired with its inverse
        return netChanges == 0 ? totalOperations / 2 : -1; // If 'netChanges' is not zero, a solution is not possible
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cstdlib> // For abs()

using std::vector;

class Solution {
public:
    // Function to calculate the minimum operations to make each corresponding
    // pair of elements in nums1 and nums2 equal using increments or decrements by k
    long long minOperations(vector<int>& nums1, vector<int>& nums2, int k) {
        // Initialize the answer and the total adjustments needed
        long long totalOperations = 0, totalAdjustments = 0;

        // Iterate over both arrays
        for (int i = 0; i < nums1.size(); ++i) {
            int num1 = nums1[i], num2 = nums2[i];

            // If k is 0, we cannot perform any operation, and if num1 is not equal to num2
            // it means it's impossible to make them equal, thus return -1
            if (k == 0) {
                if (num1 != num2) {
                    return -1;
                }
                continue; // Otherwise, no operation needed for this pair, continue to next pair
            }

            // Check if (num1 - num2) is not a multiple of k, return -1 as it's impossible
            // to equalize the pair with increments/decrements of k
            if ((num1 - num2) % k != 0) {
                return -1;
            }

            // Calculate the number of operations needed to make the two numbers equal
            int operationsNeeded = (num1 - num2) / k;

            // Accumulate the absolute value of the operations needed
            totalOperations += abs(operationsNeeded);

            // Update the total adjustments which might be positive, negative or zero
            totalAdjustments += operationsNeeded;
        }

        // If total adjustments sum to zero, the result is totalOperations / 2
        // since each operation can be counted twice (once for each array element)
        return totalAdjustments == 0 ? totalOperations / 2 : -1;
    }
};

```

Typescript Solution

```typescript
// Function to find the minimum number of operations required to make both arrays equal
// by only incrementing or decrementing elements by a value of 'k'.
// Returns the minimum number of operations needed, or -1 if it's not possible.
function minOperations(nums1: number[], nums2: number[], incrementStep: number): number {

    // Number of elements in the first array
    const arrayLength = nums1.length;

    // If incrementStep is zero, check if arrays are already equal
    if (incrementStep === 0) {
        return nums1.every((value, index) => value === nums2[index]) ? 0 : -1;
    }

    // Initial sum of differences and the total adjustments needed
    let sumDifferences = 0;
    let totalAdjustments = 0;

    // Loop through each element to sum the differences and total adjustments needed
    for (let i = 0; i < arrayLength; i++) {
        // Calculate the difference between the corresponding elements
        const difference = nums1[i] - nums2[i];

        // Accumulate the sum of differences
        sumDifferences += difference;

        // If the difference is not divisible by incrementStep, it's not possible to make arrays equal
        if (difference % incrementStep !== 0) {
            return -1;
        }

        // Sum the absolute value of the difference to calculate total adjustment steps needed
        totalAdjustments += Math.abs(difference);
    }

    // If the sum of differences is not zero, arrays cannot be made equal
    if (sumDifferences !== 0) {
        return -1;
    }

    // Return the number of operations calculated by the total adjustments
    // divided by the step size multiplied by two (since each operation changes
    // the difference by incrementStep * 2)
    return totalAdjustments / (incrementStep * 2);
}

```

Time and Space Complexity

The time complexity of the given code is `O(N)` where `N` is the length of the shorter of the two input lists, `nums1` and `nums2`. This is because the code iterates over each pair of elements from `nums1` and `nums2` once by using `zip`, and the operations within the loop (such as modulo, division, comparison, and addition) are all `O(1)` operations. No nested loops or recursive calls that would increase the complexity are present.

The space complexity of the code is `O(1)` since the space used does not increase with the size of the input; it only uses a fixed number of variables (`ans`, `x`, `a`, `b`, `y`) to process the input and produce the output.