1775. Equal Sum Arrays With Minimum Number of Operations

Medium Topics Array Hash Table Counting

Problem Description

You are given two arrays of integers, `nums1` and `nums2`, which may not be of equal length. Each element in both arrays will be an integer between `1` and `6`, inclusive. Your task is to equalize the sum of both arrays using the minimum number of operations. An operation consists of changing any element's value in either array to any integer between `1` and `6` (also inclusive). You need to determine the least number of such operations needed to make the sums of both arrays equal, or return `-1` if equalizing the sums is impossible.

Intuition

To solve this problem intuitively, one should start by comparing the sums of both arrays. If they are already equal, no operations are needed, so the answer is `0`. If the sums are not equal, operations will entail either increasing some numbers in the smaller-summed array or decreasing some in the larger-summed array.

There are limit cases where it's impossible to make the sums equal. One occurs when the smaller array is so small and full of sixes, and the larger array is so large and full of ones that even maxing out the values in the smaller one won't help. Another occurs when the larger array is full of sixes, and the smaller is full of ones, and decreasing values in the larger array is equally futile.

Approaching the array with the smaller sum, we look at how much we can increase each of its elements (up to a maximum of `6`). For the larger sum array, we calculate how much we can decrease each element (down to a minimum of `1`). The resulting potential changes are combined into a single list, sorted by the magnitude of potential change in descending order.

We then iterate through each potential change from largest to smallest. With every iteration, we subtract the value of the potential change from the sum difference between the arrays. If at any point the sum difference drops to `0` or below, we know that we've made enough changes to equalize the arrays, and we return the number of operations taken thus far. If we exhaust the list of potential changes and the sum difference remains positive, we conclude that equalizing the sums is impossible and return `-1`.

Solution Approach

The implementation groups a clear algorithmic approach with efficient use of data structures, mainly lists and the `sum` function. Here's the breakdown:

1. First, calculate the sums of both input arrays, `nums1` and `nums2`, using the `sum` function. We need these sums to decide our next step and to calculate the difference between the two.
2. If the sums of both arrays are already equal (`s1 == s2`), return `0` as no operations are required.
3. Verify if the sum of `nums1` is greater than that of `nums2`. If this is true, call the function recursively with the arrays flipped because it's more practical to begin with the array that currently has the smaller sum.
4. To generate the list of potential changes, subtract each value in the smaller-summed array (`nums1`) from `6` to see the maximum increase possible for each element. For the larger-summed array (`nums2`), subtract `1` from each value to see the maximum decrease possible. Combine these two lists.
5. Sort this combined list in reverse order (from the largest to the smallest potential change) because you want the most significant changes to be considered first.
6. Initialize a variable `d` with the value of the difference between `s2` and `s1` (the sum of `nums2` and `nums1`).
7. Iterate over the sorted list of potential changes. During each iteration:
   - Subtract the current potential change from `d`.
   - Increment a counter to keep track of how many changes we've used.
   - If `d` is less or equal to `0` after the subtraction, return the number of operations (the current counter value) as you've managed to equalize the sums.
8. If you complete the iteration without `d` dropping to or below `0`, return `-1` as it confirms that equalizing the sums is not possible with the given constraints.

This algorithm effectively utilizes *greedy* strategy by sorting and consuming potential changes from greatest to least effect. It assures the minimum number of operations as each step makes the largest possible contribution to equalizing the array sums.

Example Walkthrough

Let's consider the following example to illustrate the solution approach:

- `nums1 = [1, 2, 3]` and `nums2 = [3, 1, 6, 1]`

Now, let's walk through the solution steps:

1. Calculate the sums of `nums1` and `nums2`:
   - `sum(nums1) = 1+2+3 = 6`
   - `sum(nums2) = 3+1+6+1 = 11`
2. The sums are not equal (`6 != 11`), so we proceed with the next steps.
3. Since the sum of `nums1` is greater than that of `nums1`, we recursively call the function with `nums2` and `nums1` swapped. Now our goal is to increase the sum of `nums1` or decrease the sum of `nums2`.
4. Calculate the potential changes:
   - For `nums1`, we can increase `1` to `6` (gaining `5`), `2` to `6` (gaining `4`), and we cannot increase `3` any further.
   - For `nums2`, we can decrease `3` to `1` (gaining `2`), `6` to `1` (gaining `5`), and we do not gain from `1`s.
   - The potential changes list is `[5, 4, 2, 5]`.
5. Sort the potential changes list into `[5, 5, 4, 2]`, favoring the biggest changing numbers.
6. Initialize `d` with the difference between sums: `d = 11 - 6 = 5`.
7. Iterate over the sorted list of potential changes:
   - Using `5` from the first array brings `d` to `0` (`5 - 5 = 0`).
   - Since `d` is now less than `0`, we know that we have equalized the arrays. It took us `1` operation to achieve this.

Therefore, in this example, the smallest number of operations needed to equalize the sums of both arrays is `1`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def min_operations(self, nums1: List[int], nums2: List[int]) -> int:
        # Calculate the sum of both lists
        sum1, sum2 = sum(nums1), sum(nums2)

        # If sums are equal, no operations are needed
        if sum1 == sum2:
            return 0

        # If sum1 is greater than sum2, swap the lists and re-run the function
        if sum1 > sum2:
            return self.min_operations(nums2, nums1)

        # Calculate the difference between the sums
        difference = sum2 - sum1

        # Create a list of possible changes (increments for nums1, decrements for nums2)
        possible_changes = [6 - num for num in nums1] + [num - 1 for num in nums2]

        # Sort the changes in descending order to optimize the number of operations
        sorted_changes = sorted(possible_changes, reverse=True)

        # Apply the changes one by one and count the number of operations
        for i, value in enumerate(sorted_changes, start=1):
            difference -= value  # Apply the change to the difference
            if difference <= 0:
                return i  # Return the number of operations if the new sum is equal or higher

        # If it's not possible to make sums equal, return -1
        return -1
```

Java

```java
class Solution {
    public int minOperations(int[] nums1, int[] nums2) {
        // Calculate the sum of elements in nums1 and nums2
        int sum1 = Arrays.stream(nums1).sum();
        int sum2 = Arrays.stream(nums2).sum();

        // If the sums are equal, no operations are required
        if (sum1 == sum2) {
            return 0;
        }

        // If sum1 is greater than sum2, we will need to perform operations on nums2 to make
        // both arrays' sums equal. So we recursively call the function with reversed parameters.
        if (sum1 > sum2) {
            return minOperations(nums2, nums1);
        }

        // Calculate the difference that needs to be bridged
        int difference = sum2 - sum1;

        // Array to count the number of operations needed to increment or decrement
        // Each index i represents the number of operations to increase sum1 or decrease sum2 by (i+1)
        // Index 0 represents a single increment/decrement up to index 5 representing six increments/decrements
        int[] counts = new int[5];

        // Count the potential operations in nums1 (increment operations)
        for (int value : nums1) {
            ++counts[6 - value];
        }
        // Count the potential operations in nums2 (decrement operations)
        for (int value : nums2) {
            ++counts[value - 1];
        }

        // Variable to store the number of operations performed
        int operations = 0;

        // Iterate through the potential operations from the highest (i=5, six-crement)
        // to the lowest (i=0, double-crement) to minimize the number of operations needed
        for (int i = 5; i > 0; --i) {
            // As long as there are operations left that can be performed and the difference is positive,
            // keep reducing the difference and incrementing the operations count
            while (counts[i] > 0 && difference > 0) {
                difference -= i;
                --counts[i];
                ++operations;
            }
        }

        // If the difference is now zero or less, we were successful in equalling the sums
        // using a minimum number of operations; otherwise, it's not possible (-1)
        return difference <= 0 ? operations : -1;
    }
}
```

C++

```cpp
#include <vector>
#include <numeric>
#include <algorithm>

class Solution {
public:
    // Calculates the minimum number of operations to make the sum of nums1 equal to the sum of nums2
    int minOperations(std::vector<int>& nums1, std::vector<int>& nums2) {
        // Calculate the sum of elements in nums1 and nums2
        int sum1 = std::accumulate(nums1.begin(), nums1.end(), 0);
        int sum2 = std::accumulate(nums2.begin(), nums2.end(), 0);

        // If the sums are already equal, no operations are needed
        if (sum1 == sum2) return 0;

        // Ensure sum1 is the smaller sum, for consistent processing
        if (sum1 > sum2) return minOperations(nums2, nums1);

        // Calculate the difference that needs to be overcome
        int difference = sum2 - sum1;

        // Create an array to store the maximum possible increments from nums1
        // and the maximum possible decrements from nums2
        std::vector<int> changes;
        changes.reserve(nums1.size() + nums2.size());

        // Calculate potential changes if we increase nums1's values (each number can go as high as 6)
        for (int value : nums1) changes.push_back(6 - value);
        // Calculate potential changes if we decrease nums2's values (each number can go as low as 1)
        for (int value : nums2) changes.push_back(value - 1);

        // Sort the array of changes in decreasing order to maximize the effect of each operation
        std::sort(changes.begin(), changes.end(), std::greater<int>());

        // Apply the changes in order, and count the operations, until the difference is eliminated
        for (size_t i = 0; i < changes.size(); ++i) {
            difference -= changes[i];
            if (difference <= 0) return i + 1;  // Return the number of operations taken so far
        }

        // If we applied all changes and the difference still wasn't eliminated, return -1
        return -1;
    }
};
```

TypeScript

```typescript
function minOperations(nums1: number[], nums2: number[]): number {
    // Sum the elements of nums1 and nums2
    let sum1: number = nums1.reduce((a, b) => a + b, 0);
    let sum2: number = nums2.reduce((a, b) => a + b, 0);

    // If the sums are already equal, no operations are needed
    if (sum1 === sum2) return 0;

    // Ensure sum1 is the smaller sum, for consistent processing
    if (sum1 > sum2) return minOperations(nums2, nums1);

    // Calculate the difference that needs to be overcome
    let difference: number = sum2 - sum1;

    // Create an array to store the maximum possible increments from nums1
    // and the maximum possible decrements from nums2
    let changes: number[] = [];

    // Calculate potential changes if we increase nums1's values (each number can go as high as 6)
    nums1.forEach(value => changes.push(6 - value));
    // Calculate potential changes if we decrease nums2's values (each number can go as low as 1)
    nums2.forEach(value => changes.push(value - 1));

    // Sort the array of changes in decreasing order to maximize the effect of each operation
    changes.sort((a, b) => b - a);

    // Apply the changes in order, and count the operations, until the difference is eliminated
    for (let i = 0; i < changes.length; ++i) {
        difference -= changes[i];
        if (difference <= 0) {
            // Return the number of operations taken so far
            return i + 1;
        }
    }

    // If we applied all changes and the difference still wasn't eliminated, return -1
    return -1;
}

// Example usage
// const ans = minOperations([1, 2, 3], [4, 5, 6]);
// console.log(ans); // Should log the number of operations needed
```

```python
from typing import List

class Solution:
    def min_operations(self, nums1: List[int], nums2: List[int]) -> int:
        # Calculate the sum of both lists
        sum1, sum2 = sum(nums1), sum(nums2)

        # If sums are equal, no operations are needed
        if sum1 == sum2:
            return 0

        # If sum1 is greater than sum2, swap the lists and re-run the function
        if sum1 > sum2:
            return self.min_operations(nums2, nums1)

        # Calculate the difference between the sums
        difference = sum2 - sum1

        # Create a list of possible changes (increments for nums1, decrements for nums2)
        possible_changes = [6 - num for num in nums1] + [num - 1 for num in nums2]

        # Sort the changes in descending order to optimize the number of operations
        sorted_changes = sorted(possible_changes, reverse=True)

        # Apply the changes one by one and count the number of operations
        for i, value in enumerate(sorted_changes, start=1):
            difference -= value  # Apply the change to the difference
            if difference <= 0:
                return i  # Return the number of operations if the new sum is equal or higher

        # If it's not possible to make sums equal, return -1
        return -1
```

Time and Space Complexity

Time Complexity

The time complexity of the code can be broken down into the following parts:

- Summation of both arrays `nums1` and `nums2`: This has a time complexity of `O(N + M)`, where `N` is the length of `nums1`, and `M` is the length of `nums2`, since each array is iterated over once.
- The recursive call `self.minOperations(nums2, nums1)` doesn't change the time complexity since it happens at most once and just swaps the role of `nums1` and `nums2`.
- Creation of the combined array `arr`: This involves iterating over both `nums1` and `nums2` and thus has a complexity of `O(N + M)`.
- Sorting the `arr`: The sorting operation has a complexity of `O((N + M) * log(N + M))`, because it sorts a list that has a length equal to the sum of lengths of `nums1` and `nums2`.
- Iterating over the sorted array to find the minimum number of operations: In the worst case, this is `O(N + M)` when we iterate over the entire combined list.

Therefore, the dominant factor for time complexity here is the sorting operation. The overall time complexity of the function is `O((N + M) * log(N + M))`.

Space Complexity

The space complexity of the code can be analyzed as follows:

- Extra space for the sum operations is `O(1)` since we just accumulate values.
- Extra space for the combined array `arr`: This takes up `O(N + M)` space.
- Space required for the sorting of `arr`: Sorting typically requires `O(N + M)` space.

Given that the combined array `arr` and its sorting are the steps that require the most space, the overall space complexity is `O(N + M)`.