2918. Minimum Equal Sum of Two Arrays After Replacing Zeros

Medium

Leetcode Link

Problem Description

In this problem, we are provided with two arrays, `nums1` and `nums2`, each containing positive integers. However, both arrays may contain zeros, and we need to find a way to replace these zeros with positive integers. Our goal is to ensure that after the replacements, the sum of the integers in both arrays is the same.

The question then is: What is the minimum equal sum we can achieve after replacing all zeros with strictly positive integers? If it is not possible to make the sums of both arrays equal by replacing zeros, the function should return `-1`.

It's a problem that demands a careful examination of the sums of arrays and the zeros they contain to find a method to balance the sums with minimum changes.

Intuition

To approach the problem, we first handle the zeros in a straightforward way. We consider replacing each zero with 1, as this is the smallest strictly positive integer. This gives us a baseline sum for each array.

Let `s1` and `s2` be the sums of `nums1` and `nums2` respectively when each zero is treated as 1. Based on these sums, we can reason about the possible scenarios:

1. If `s1` equals `s2`, we already have equality and do not need to replace any zeros with larger numbers, so the baseline sum is the answer.
2. If `s1` is less than `s2`, and there are zeros in `nums1`, we can increase the sum of `nums1` by replacing zeros with numbers larger than 1 to match `s2`. In this case, since `s2` is larger, it is our target, and we return `s2` because it represents the minimum equal sum we can achieve.
3. Conversely, if `s1` is more than `s2`, we swap the roles of `nums1` and `nums2` to apply the above reasoning.
4. If `s1` is less than `s2`, but `nums1` has no zeros, we cannot increase the sum of `nums1` to match `s2`. Thus, it is impossible to make the sums equal and we return `-1`.

This way, we identify the minimum sum after replacements that makes both arrays' sums equal, or determine the impossibility of such an operation.

Solution Approach

The solution approach is to first calculate the sums of the arrays while treating all zeros as ones. This calculation is straightforward and can be done using the built-in `sum` function in Python, along with the `count` method to account for the zeros. The following code calculates these sums for `nums1` and `nums2`:

```
s1 = sum(nums1) + nums1.count(0)
s2 = sum(nums2) + nums2.count(0)
```

Once the sums are calculated, we compare them to decide the next steps:

- If `s1` is greater than `s2`, we have a helper function, `self.minSum`, that we call with the arrays swapped. This is an example of recursion that helps us reduce the problem to a single direction where `s1` is less than or equal to `s2`, ensuring a more straightforward decision process:

```
if s1 > s2:
    return self.minSum(nums2, nums1)
```

- If the sums are already equal (`s1 == s2`), we have achieved the goal of equal sums with minimum effort, merely by treating zeros as ones. The baseline sum `s1` (or `s2`, since they are equal now) is the minimum sum we can achieve, so we return it:

```
if s1 == s2:
    return s1
```

- In the case that `s1` is less than `s2`, we have two scenarios. If there are zeros in `nums1`, we can replace them with numbers greater than one to match the sum `s2`. Therefore, we return `s2` which is the target sum:

```
if nums1.count(0) != 0:
    return s2
```

- Finally, if there are no zeros in `nums1`, it implies we cannot adjust the sum of `nums1` to match `s2`, making it impossible for both array sums to be equal, and we return `-1`:

```
else:
    return -1
```

By considering these conditions and applying the recursive step, the algorithm ensures minimum modifications are made to achieve equal sums, or it correctly identifies when the task is impossible. The use of simple built-in functions and conditional logic, along with recursion, forms the core of this solution, eliminating the need for complex data structures or patterns.

Example Walkthrough

Consider the arrays `nums1 = [0, 2, 3]` and `nums2 = [1, 0, 4, 0]`.

1. We start by treating all zeros in `nums1` and `nums2` as ones and calculate their sums.
   - For `nums1`: The sum is `1 (replacing 0) + 2 + 3 = 6`.
   - For `nums2`: The sum is `1 + 1 (replacing 0) + 4 + 1 (replacing 0) = 7`.
   - `s1` becomes 6 and `s2` becomes 7.
2. We compare `s1` and `s2`. Since `s1` is less than `s2`, we proceed without swapping the arrays.
3. Because `s1` (sum of `nums1`) is less than `s2` (sum of `nums2`), we check if there are zeros present in `nums1` that we can replace to match the sum `s2`. Indeed, `nums1` has one zero.
4. We can replace the zero in `nums1` with a positive integer that will raise the sum of `nums1` to match `s2`. Since `s2` is 7, and the current sum of `nums1` is 6, we can replace the zero with any number greater than 1 to meet or exceed the sum of `nums2`. For minimal increment, we can choose to replace zero with the smallest number greater than 1, which is 2. It is now `nums1 = [2, 2, 3]`.
5. After replacement, the sum of `nums1` is now `2 + 2 + 3 = 7`, which matches `s2`.
6. Since we can match the sums by replacing zeros with strictly positive integers, we return `s2`. Here, the minimum equal sum we achieve is 7.

According to this logic, if `nums1` had no zeros but still had a smaller sum than `nums2`, we would return `-1`, indicating it's not possible to match the sums.

Now, let's break down the steps according to the code snippets provided in the solution approach:

- Step 1 corresponds to the initial calculations:

```
s1 = sum(nums1) + nums1.count(0)  # s1 = 6
s2 = sum(nums2) + nums2.count(0)  # s2 = 7
```

- Since `s1 < s2`, we do not swap `nums1` and `nums2`, and the first recursive call is skipped.
- Checking `s1 == s2` would skip the second condition in this case since they are not equal.
- In the third condition, we evaluate `if nums1.count(0) != 0:`. Because `nums1` contains a zero, we return `s2` (7), which represents the minimum equal sum we can achieve after replacements.
- There is no need to execute the last condition `else: return -1` since we did not return `-1`.

```
if nums1.count(0) != 0:
    return s2  # returns 7
```

The given example demonstrates how the solution approach is applied to find the minimum equal sum after replacing all zeros with strictly positive integers or determining when it's not possible to match the sums.

Python Solution

```python
class Solution:
    def minSum(self, nums1: List[int], nums2: List[int]) -> int:
        # Calculate the sum of elements in each list. The sum includes an extra 1 for each zero found in the list
        sum_nums1 = sum(nums1) + nums1.count(0)
        sum_nums2 = sum(nums2) + nums2.count(0)

        # If the sum of the first list is greater than the second, recursively call minSum with reversed lists
        if sum_nums1 > sum_nums2:
            return self.minSum(nums2, nums1)

        # If the sums are equal, return the sum (since that is the minimum sum after replacing zeros)
        if sum_nums1 == sum_nums2:
            return sum_nums1

        # If there are no zeros in the first list, it's not possible to make the sums equal, return -1
        # Otherwise, return the sum of the second list as it is the smaller sum
        return -1 if nums1.count(0) == 0 else sum_nums2

```

Java Solution

```java
class Solution {

    /**
     * Computes the minimum sum possible between two given arrays after making sure
     * that none of the elements in the arrays are less than one.
     * It returns -1 if it is impossible to make both sums equal without the use of zeros.
     *
     * @param nums1 the first array of integers
     * @param nums2 the second array of integers
     * @return the minimum sum possible or -1 if it can't be done
     */
    public long minSum(int[] nums1, int[] nums2) {
        long sum1 = 0; // Initialize sum for the first array
        long sum2 = 0; // Initialize sum for the second array
        boolean hasZero = false; // Flag to check for presence of zero in nums1

        // Iterate over the first array
        for (int value : nums1) {
            hasZero |= value == 0; // Set the flag if zero is found
            sum1 += Math.max(value, 1); // Ensure that each value contributes at least 1 to the sum
        }

        // Iterate over the second array
        for (int value : nums2) {
            sum2 += Math.max(value, 1); // Ensure that each value contributes at least 1 to the sum
        }

        // If the sum of the first array is greater, call the function again with reversed parameters
        if (sum1 > sum2) {
            return minSum(nums2, nums1);
        }

        // If the sums are equal, return the sum of the first array
        if (sum1 == sum2) {
            return sum1;
        }

        // If there is a zero in the first array and the sums are not equal, returning the sum of the
        // second array is valid; otherwise, return -1 as it is impossible to make sums equal.
        return hasZero ? sum2 : -1;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the minimum sum of modified arrays such that their sums are equal
    long long minSum(vector<int>& nums1, vector<int>& nums2) {
        long long sumNums1 = 0; // Sum of elements in nums1
        long long sumNums2 = 0; // Sum of elements in nums2
        bool hasZero = false;   // Flag to check if nums1 contains a zero

        // Calculate the sum for nums1, replacing zeros with ones (as min(x, 1) ensures)
        for (int num : nums1) {
            hasZero |= num == 0;  // Update the flag if there's a zero in the array
            sumNums1 += std::max(num, 1); // Add the maximum of the number and 1 to the sum
        }

        // Calculate the sum for nums2 in a similar way, replacing zeros with ones
        for (int num : nums2) {
            sumNums2 += std::max(num, 1);
        }

        // If sum of nums1 is greater than sum of nums2, we should calculate minSum in the opposite order
        if (sumNums1 > sumNums2) {
            return minSum(nums2, nums1);
        }

        // If sums are equal, return the sum of nums1 (which is now confirmed to be the smaller sum)
        if (sumNums1 == sumNums2) {
            return sumNums1;
        }

        // If there's a zero in nums1 and the sums are not equal, it's impossible to equalize the sums
        // Since we can't lower the sum of the other array, hence we return -1.
        // If there is no zero, we can return the sum of the second array since it's larger.
        return hasZero ? sumNums2 : -1;
    }
};

```

Typescript Solution

```typescript
/**
 * Calculates the minimal sum of two arrays after ensuring that all elements are at least 1.
 * If an array contains a zero, it is possible to swap arrays to minimize the sum.
 *
 * @param {number[]} nums1 First array of numbers.
 * @param {number[]} nums2 Second array of numbers.
 * @returns {number} The minimal sum or -1 if it's not possible to equalize sums with the given conditions.
 */
function minSum(nums1: number[], nums2: number[]): number {
    let sum1 = 0;
    let sum2 = 0;
    let containsZero = false;

    // Calculate sum for nums1 and flag if zero is found
    for (const number of nums1) {
        if (number === 0) {
            containsZero = true;
        }
        sum1 += Math.max(number, 1);
    }

    // Calculate sum for nums2
    for (const number of nums2) {
        sum2 += Math.max(number, 1);
    }

    // Recursive call to swap nums1 and nums2 if sum1 is greater than sum2
    if (sum1 > sum2) {
        return minSum(nums2, nums1);
    }

    // If sums are equal, return the calculated sum
    if (sum1 === sum2) {
        return sum1;
    }

    // If contains a zero, we can swap to possibly minimize sum, otherwise return -1 as it's not possible
    return containsZero ? sum2 : -1;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be broken down as follows:

- Calculating sum of `nums1` and `nums2` requires traversing both arrays, which costs `O(n)` and `O(m)` respectively, where `n` is the length of `nums1` and `m` is the length of `nums2`.
- Counting the zeroes in `nums1` and `nums2` with `nums1.count(0)` and `nums2.count(0)` also takes `O(n)` and `O(m)` time respectively.
- The recursive call `self.minSum(nums2, nums1)` only happens if `s1 > s2`. If multiple invocations occur, the process will still not exceed `O(n + m)` because the sum and count operations are repeated once for each list.
- The `if` comparisons are constant time operations, `O(1)`.

Aggregating all these costs, the total time complexity is `O(n + m)` because we are summing separate linear terms associated with each list.

Space Complexity

- The space complexity is `O(1)` as no additional space that grows with the input size is used. Variables `s1` and `s2` use constant space, and the recursive call does not exceed a single level of stack space since it's a direct swap with no further recursion.