3038. Maximum Number of Operations With the Same Score I

Easy Array Simulation

Problem Description

In this problem, we have an array of integers called `nums`. We are allowed to remove pairs of elements from the start of the array. Each time we remove a pair, we calculate the sum of the two elements and consider this the score of the operation. The goal is to perform as many operations as possible with the condition that all operations must yield the same score. This means that each pair of elements we remove must have the same sum as the first pair we removed.

To find the maximum number of operations we can perform under these conditions, we need to continuously check the first two elements of the remaining array, removing them only if they match the desired score. If at any point the sum of the two elements doesn't match the score, we must stop the operations, and that's the maximum number we can perform.

Intuition

To solve this problem, the solution begins by presuming that the score we need every operation to match is the sum of the first two elements in the array, denoted as `s`. The solution then proceeds to traverse the array in steps of 2, meaning it looks at every consecutive pair of elements.

During this traversal, the solution checks if the sum of each pair matches the initial score `s`. As soon as it encounters a pair that does not have a sum equal to `s`, or if there are no more pairs to check (e.g., after removing pairs the array has only one element left), the traversal stops. This is grounded in the rule that all operations must yield the same score, and since the score is defined by the first operation, any pair that doesn't match this score would break the condition.

Solution Approach

The implemented solution follows a straightforward and direct approach. It doesn't employ complex algorithms or data structures but relies on a simple traversal of the array using a for loop.

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

1. The solution initializes a variable `s` with the sum of the first two elements in `nums`. This `s` will be the score that every operation must match.

```
s = nums[0] + nums[1]
```

2. It initializes an answer variable, `ans`, to count the number of operations performed and a variable `n` to store the length of the array.

```
ans, n = 0, len(nums)
```

3. The solution uses a for loop to iterate over the elements of the array in steps of two, looking at every pair of elements.

```
for i in range(0, n, 2):
```

4. For each pair, it checks two conditions: whether the current index `i + 1` equals the length of the array (which would mean there's no pair left to process) and whether the sum of the current pair does not match the score `s`.

```
if i + 1 == n or nums[i] + nums[i + 1] != s:
    break
```

If either condition is true, the loop breaks, effectively stopping any further operations, as either there are no more pairs to remove or the condition of maintaining the same score can no longer be met.

5. If none of the conditions for breaking the loop are met, it means that the current pair matches the score, and an operation can be performed. The `ans` variable is incremented to reflect this.

```
ans += 1
```

6. The loop continues until it has either checked all elements in the array or encountered a break. At the end of the loop, the solution returns the count `ans`, which represents the maximum number of operations that have been performed with the same score.

```
return ans
```

This solution assumes that the input array `nums` has already been set up such that every valid operation (pair with matching sum `s`) is adjacent and that there are no possible operations (pairs with the desired sum) after encountering the first invalid one. Therefore, it's essential to confirm that the input array will conform to these conditions; otherwise, the solution might not correctly calculate the maximum number of operations.

Example Walkthrough

Let's illustrate the solution approach with a simple example.

Suppose we have the following array of integers as our `nums`:

```
nums = [4, 6, 4, 6, 8, 2]
```

We will walk through the steps of the implementation using this array.

1. First, we initialize the variable `s` with the sum of the first two elements in `nums`, which is 4 + 6 = 10.

```
s = nums[0] + nums[1]  # s = 10
```

2. We initialize our answer variable, `ans`, to 0 and a variable `n` to store the length of the array, which is 6 in this case.

```
ans, n = 0, len(nums)  # ans = 0, n = 6
```

3. The for loop begins and will iterate over the pairs of elements.

```
for i in range(0, n, 2):
```

4. At first, `i = 0`, and we look at the first pair `nums[0] + nums[1]` which is 4 + 6. This equals `s` (10), so we can remove this pair. No `break` is encountered, and we increment `ans`.

```
# First iteration:
# nums[0] + nums[1] = 10, which is equal to s.
ans += 1  # ans = 1
```

5. Now `i = 2`, and we look at the next pair `nums[2] + nums[3]` which is 4 + 6. This also equals `s` (10), so we remove this pair as well. We increment `ans`.

```
# Second iteration:
# nums[2] + nums[3] = 10, which is equal to s.
ans += 1  # ans = 2
```

6. Finally, `i = 4`, and we look at the last pair `nums[4] + nums[5]` which is 8 + 2. This equals 10, the same as `s`. Even though it matches `s`, we're at the end of the array, so the loop naturally concludes.

```
# Third iteration:
# nums[4] + nums[5] = 10, which is equal to s. However, this is the last pair.
ans += 1  # ans = 3
```

7. The loop finishes because there are no more elements in `nums`. The solution will return the count `ans`, which is 3 in this case.

```
return ans  # returns 3
```

This example shows that the array `nums` allowed for 3 pairs to be removed with the sum matching `s = 10`, hence 3 operations were performed. The example follows the method of sequential checking and early termination if the condition is not met. In this instance, the conditions were satisfied by all element pairs, so the maximum number of operations is equal to the number of pairs in the array.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def maxOperations(self, nums: List[int]) -> int:
        # Check if there are enough numbers to form pairs.
        if len(nums) < 2:
            return 0

        # Initialize the sum of the first pair and the answer counter.
        sum_of_pair = nums[0] + nums[1]
        operations_count = 0
        total_numbers = len(nums)

        # Iterate through the list in steps of 2 to form pairs.
        for i in range(0, total_numbers, 2):
            # Check whether we reached the end or if the sum of the current pair doesn't match sum_of_pair.
            if i + 1 == total_numbers or nums[i] + nums[i + 1] != sum_of_pair:
                break
            # If we found a matching pair, increment the counter.
            operations_count += 1

        # Return the total pairs formed.
        return operations_count
```

Java

```java
public class Solution {
    public int maxOperations(int[] nums) {
        // Calculate the sum of the first pair of elements in the array
        int targetSum = nums[0] + nums[1];

        // Initialize a counter for the number of valid operations
        int operationsCount = 0;

        // Length of the nums array
        int n = nums.length;

        // Loop through the array in pairs, up to the second-to-last element (i + 1 < n)
        for (int i = 0; i + 1 < n; i += 2) {
            // Check if the current pair sums up to the target sum
            if (nums[i] + nums[i + 1] == targetSum) {
                // If it does, increment the operations counter
                ++operationsCount;
            } else {
                // If a pair doesn't sum up to the target sum, break out of the loop
                // No need to continue as subsequent operations will not be valid
                break;
            }
        }

        // Return the total number of valid operations
        return operationsCount;
    }
}
```

C++

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

class Solution {
public:
    // Function to determine the maximum number of operations where each operation consists of
    // finding a pair of elements from the array that add up to the same value
    int maxOperations(vector<int>& nums) {
        // Initialize the sum "s" with the sum of the first two elements
        int targetSum = nums[0] + nums[1];
        // Initialize the answer, which stores the number of valid operations
        int operations_count = 0;
        // Get the size of the nums array
        int n = nums.size();

        // Iterate over the elements of the vector in pairs
        for (int i = 0; i + 1 < n && nums[i] + nums[i + 1] == target_sum; i += 2) {
            // If the sum of the current pair of elements equals the target sum, increment the operations count
            ++operations_count;
        }
        // Return the total number of valid operations
        return operations_count;
    }
};
```

TypeScript

```typescript
function maxOperations(nums: number[]): number {
    // Initialize a sum 'requiredSum' with the sum of the first two elements.
    const requiredSum = nums[0] + nums[1];

    // 'n' represents the total number of elements in the 'nums' array.
    const n = nums.length;

    // 'operationsCount' will hold the number of valid operations performed.
    let operationsCount = 0;

    // Iterate over the 'nums' array with a step of 2.
    for (let i = 0; i + 1 < n; i += 2) {
        // Check if the current and next element sum up to the 'requiredSum'.
        if (nums[i] + nums[i + 1] === requiredSum) {
            // Increment the count of valid operations if the condition is met.
            operationsCount++;
        } else {
            // If the condition is not met, break the loop as it's assumed that nums were initially arranged in pairs with the s
            break;
        }
    }

    // Return the total number of operations performed.
    return operationsCount;
}
```

```python
from typing import List

class Solution:
    def maxOperations(self, nums: List[int]) -> int:
        # Check if there are enough numbers to form pairs.
        if len(nums) < 2:
            return 0

        # Initialize the sum of the first pair and the answer counter.
        sum_of_pair = nums[0] + nums[1]
        operations_count = 0
        total_numbers = len(nums)

        # Iterate through the list in steps of 2 to form pairs.
        for i in range(0, total_numbers, 2):
            # Check whether we reached the end or if the sum of the current pair doesn't match sum_of_pair.
            if i + 1 == total_numbers or nums[i] + nums[i + 1] != sum_of_pair:
                break
            # If we found a matching pair, increment the counter.
            operations_count += 1

        # Return the total pairs formed.
        return operations_count
```

Time and Space Complexity

The time complexity of the code is $O(n/2)$ since the loop increments by 2 each time, resulting in looping through half of the array size in the worst case. However, this simplifies to $O(n)$ because constant factors are dropped in Big O notation.

The space complexity of the code is $O(1)$ because it uses a fixed amount of additional space (variables `s`, `ans`, and `n`) that doesn't scale with the size of the input array `nums`.