2169. Count Operations to Obtain Zero

Easy Math Simulation

Problem Description

You are provided with two non-negative integers `num1` and `num2`. Your task is to perform a series of operations to reduce either `num1` or `num2` to zero. In a single operation, you compare `num1` and `num2`. If `num1` is greater than or equal to `num2`, you subtract `num2` from `num1`. Otherwise, you subtract `num1` from `num2`. The operation is repeated until one of the numbers becomes zero. The goal is to determine the number of operations required to achieve this.

Intuition

The key to solving the problem is recognizing that in each operation, the larger number is being reduced by the smaller number. This is reminiscent of the Euclidean algorithm, which is used to find the greatest common divisor (GCD) of two numbers, although in this problem, we're not necessarily finding the GCD, but rather bringing one of the numbers down to zero.

To arrive at the solution approach, we can use a while loop that runs as long as neither `num1` nor `num2` is zero. On each iteration, the operation is performed as per the described rules: if `num1` is greater than or equal to `num2`, then `num1` is to be subtracted by `num2`, but to make the calculation easier and more efficient, we can swap `num1` and `num2` instead, and then proceed to subtract `num1` from `num2`. Each time an operation is performed, we increment a counter variable to keep track of how many operations have been carried out. We continue this process until one of the numbers is reduced to zero. At that point, we return the counter variable, which gives us the total number of operations done to reach the objective.

Solution Approach

The solution implements a straightforward iterative approach without the need for any complex algorithms, auxiliary data structures, or design patterns. Here's a step-by-step explanation of how the solution code works:

1. Initialize a counter variable `ans` to zero. This will keep track of the number of operations performed.
2. Use a `while` loop that will continue as long as both `num1` and `num2` are non-zero. This loop will break when one of the numbers becomes zero, which is our stopping condition.
3. Inside the loop, we check if `num1` is greater than or equal to `num2`. The objective is to always subtract the smaller number from the larger one. To ensure this, we swap `num1` and `num2` whenever `num1` is larger, leveraging Python's multiple assignment capability: `num1, num2 = num2, num1`. This keeps the invariant that `num1` should always be less than or equal to `num2`.
4. Perform the subtraction operation: `num2 -= num1`. This effectively reduces the larger number by the value of the smaller number, mimicking one 'operation' as described in the problem.
5. Increment the operation counter `ans` by one, signifying that an operation has been completed.
6. Continue the loop until one of `num1` or `num2` reaches zero. At that point, exit the while loop.
7. Finally, return the counter `ans`, which now contains the total number of operations performed to reach the goal.

This solution is efficient because it continuously reduces the larger number, effectively halving the problem size with many of the operations, and it does so in-place without any need for additional memory. Additionally, it takes advantage of the Python multiple assignment feature for a clean and concise implementation.

Example Walkthrough

Let's illustrate the solution approach using a small example with `num1 = 7` and `num2 = 4`.

1. We initialize our counter `ans` to `0`. This will keep track of how many operations we perform.
2. Our while condition is `while num1 != 0 and num2 != 0`, and since both `num1` and `num2` are non-zero, we enter the loop.
3. We compare `num1` and `num2`. Since `num1` (7) is greater than `num2` (4), according to our rules, we need to subtract `num2` from `num1`. However, for simplicity, we swap `num1` and `num2` instead, so now `num1` becomes `4` and `num2` becomes `7`.
4. Then we perform the subtraction: `num2 -= num1`, which means `num2` is now `7 - 4 = 3`.
5. We increment `ans` by one, so `ans = 1`.
6. We repeat this process, now `num1` (4) is still greater than `num2` (3), so we swap again. Now `num1` is `3` and `num2` is `4`.
7. We perform the operation `num2 -= num1`, so `num2` is now `4 - 3 = 1`.
8. We increment `ans` by one again, so `ans = 2`.
9. The process repeats with `num1` being `3` and `num2` being `1`, no need to swap this time. After subtraction, `num1` becomes `3 - 1 = 2` and `ans` increments to `3`.
10. Continuing, `num1` (2) is greater, so we swap. `num1` is `1`, `num2` is `2`, and after subtraction, `num2` becomes `1`. `ans` increments to `4`.
11. Finally, with `num1` being `1` and `num2` also `1`, we subtract and `num2` becomes `0`. `ans` increments to `5`.
12. Now `num2` is `0`, the while condition breaks, and we exit the loop.
13. We return our counter `ans`, which is now `5`, indicating we've performed 5 operations to reduce `num2` to zero following the given rules.

Therefore, it takes 5 operations to reduce either `num1` or `num2` to zero, given the initial values of `num1 = 7` and `num2 = 4`.

Solution Implementation

Python

```python
class Solution:
    def count_operations(self, num1: int, num2: int) -> int:
        # Initialize operation count as 0
        operation_count = 0

        # Loop until either of the numbers becomes 0
        while num1 and num2:
            # Ensure num1 is the smaller number by swapping if necessary
            if num1 >= num2:
                num1, num2 = num2, num1

            # Subtract the smaller number (num1) from the larger number (num2)
            num2 -= num1

            # Increment the operation count after each subtraction
            operation_count += 1

        # Return the total count of operations performed
        return operation_count
```

Java

```java
class Solution {
    // Counts the number of operations to make either num1 or num2 equal to 0
    // by repeatedly subtracting the smaller value from the larger one.
    public int countOperations(int num1, int num2) {
        int operationsCount = 0; // Initialize the count of operations to 0

        // Loop continues as long as neither num1 nor num2 is equal to 0
        while (num1 != 0 && num2 != 0) {
            // If num1 is greater than or equal to num2
            if (num1 >= num2) {
                num1 -= num2; // Subtract num2 from num1
            } else { // If num2 is greater than num1
                num2 -= num1; // Subtract num1 from num2
            }
            operationsCount++; // Increment the count of operations
        }
        return operationsCount; // Return the total number of operations performed
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to count the operations required to reduce either of the two numbers to zero
    // by repeatedly subtracting the smaller one from the larger one.
    int countOperations(int num1, int num2) {
        int operationsCount = 0; // Initialize counter for operations

        // Continue the loop until either num1 or num2 becomes zero
        while (num1 != 0 && num2 != 0) {
            // If num1 is greater than num2, then swap them so that num1 always has the smaller value
            if (num1 > num2) {
                std::swap(num1, num2);
            }

            // Subtract num1 from num2 (num2 is guaranteed to be the larger or equal number here)
            num2 -= num1;

            // Increment the operations counter since a valid subtraction operation was performed
            ++operationsCount;
        }

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

TypeScript

```typescript
function countOperations(num1: number, num2: number): number {
    let operationsCount = 0; // Initialize a counter to track the number of operations performed

    // Continue the process until either of the numbers becomes zero
    while (num1 !== 0 && num2 !== 0) {
        // Set num1 to the smaller of the two numbers
        // Subtract the smaller number from the laroer and set this as the new value of num2
        [num1, num2] = [Math.min(num1, num2), Math.abs(num1 - num2)];

        operationsCount++; // Increment the counter after each operation
    }

    return operationsCount; // Return the number of operations performed when the loop ends
}
```

```python
class Solution:
    def count_operations(self, num1: int, num2: int) -> int:
        # Initialize operation count as 0
        operation_count = 0

        # Loop until either of the numbers becomes 0
        while num1 and num2:
            # Ensure num1 is the smaller number by swapping if necessary
            if num1 >= num2:
                num1, num2 = num2, num1

            # Subtract the smaller number (num1) from the larger number (num2)
            num2 -= num1

            # Increment the operation count after each subtraction
            operation_count += 1

        # Return the total count of operations performed
        return operation_count
```

Time and Space Complexity

Time Complexity

The time complexity of the given algorithm is `O(num1 + num2)`. This is because in each iteration of the while loop, the algorithm subtracts the smaller number from the larger one, guaranteeing that at least one of the numbers is reduced by a proportion of its value. In the worst case, if `num1` and `num2` are consecutive Fibonacci numbers (which is the worst-case scenario for this type of subtraction loop), it will take a number of steps equal to the smaller number.

However, the actual number of operations depends on the values of `num1` and `num2`. If `num1` is much smaller than `num2`, then `num2` will be reduced very slowly, leading to a high number of operations approaching `num2 / num1`. Conversely, if `num1` is comparable to `num2`, the number of operations decreases.

Space Complexity

The space complexity of the algorithm is `O(1)`, since it uses a constant amount of space. The variables `ans`, `num1`, and `num2` are the only variables that are being modified and stored during the execution, and their space requirement does not depend on the input size. The algorithm operates directly on these variables and does not allocate any additional space that scales with the input.