2749. Minimum Operations to Make the Integer Zero

Medium Bit Manipulation Brainteaser Leetcode Link

Problem Description

The problem presents a mathematical challenge where you have two integers, `num1` and `num2`. You are tasked with finding the minimum number of operations required to reduce `num1` to 0. In each operation, you can choose an integer `i` in the range [0, 60], and then subtract `2^i + num2` from `num1`. The operation is a two-step process: first, you choose a power of 2 (2 raised to the power i), and then you add `num2` to this value before subtracting this sum from `num1`.

If, after a series of such operations, it is not possible to make `num1` equal to 0, the function should return -1. Otherwise, it should return the minimum number of these operations required to achieve the goal.

Intuition

The intuition behind the solution is to iteratively attempt to perform the operation described, starting with the smallest non-negative multiples of `num2` (expressed as `k * num2`) and subtracting this from `num1`. This is done in a loop, each time incrementing `k` to check the next multiple. This is feasible because larger values of `i` in the `2^i` term allow for larger subtractions, potentially reaching 0 in fewer steps if carefully picked.

The stopping condition for the loop is when `x`, which is `num1 - k * num2`, becomes negative, which means we've subtracted too much and it's impossible to reach exactly 0 with the current `k`.

We also have a condition that `x.bit_count()` must be less than or equal to `k`. The `bit_count()` function returns the number of 1-bits in `x`. This condition ensures that we have enough operations left to eliminate all 1-bits of `x` (because each operation can potentially remove one 1-bit by choosing the correct power of 2) and that `k` is large enough to handle its binary representation in terms of operations.

Moreover, `k` should be less than or equal to `x`, since we want to ensure we can make subtraction at least `k` more times; otherwise, we won't have enough operations to make `num1` zero with the current multiple of `num2`.

If we do not find such a `k` at the end of the loop, the function returns -1, indicating it's impossible to reduce `num1` to 0 given the conditions.

Solution Approach

The solution approach uses a simple brute force method that takes advantage of the properties of powers of two and bit manipulation:

1. **Importing the necessary module**: The solution begins by importing the `count` function from the itertools module, which provides a simple iterator that generates consecutive integers starting with the parameter supplied to it.

2. **The use of a bit count**: Since every number can be represented in binary form, the bit count (`x.bit_count()`) is crucial for determining the number of 1-bits that we can potentially eliminate through the operations. This is because each operation has the potential to eliminate a 1-bit if the correct power of 2 is chosen.

3. **Looping over multiples of `num2`**: The `for` loop runs indefinitely, increasing `k` with each iteration. `k` represents the multiple of `num2` that is going to be subtracted along with the power of 2 from `num1`. The loop stops if subtracting the multiple makes `num1` negative, which would indicate that it is impossible to reach 0 with the current value of `k`.

4. **Calculating `x`**: Within the loop, `x` is calculated by subtracting `k * num2` from `num1`. `x` therefore represents the resulting number after one or more operations have been performed.

5. **Checking conditions for a valid operation**: Two conditions are checked to see if the current `k` would result in a viable number of operations:

   - `x.bit_count() <= k`: Ensure that there are enough operations to flip each 1-bit to 0 in `x`.
   - `k <= x`: This ensures that `k` is not greater than `x`, which would mean there aren't enough remaining numbers to subtract from and thus it would be impossible to reach 0 with the current `k`.
6. **Returning the result**: If a valid `k` is found that satisfies both conditions, that `k` is returned as it represents the minimum number of operations needed to make `num1` equal to 0. If the loop finishes without finding such a `k`, the function returns -1 to indicate it's impossible to make `num1` zero with the given `num2`.

This method does not require complex data structures or algorithms beyond the bitwise operations and basic control structures, leveraging the inherent mathematical relationships within the problem's constraints.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have num1 = 42 and num2 = 3. We want to find the minimum number of operations to reduce num1 to 0 by subtracting `2^i + num2` from num1.

- Start by importing `count` from the itertools module if it's in Python, or if it's just a conceptual step, we set `k` initially to 0 and iterate over increasing values of `k`.
- The operation we perform in each step is essentially choosing an `i` such that `2^i + num2` is a valid term to subtract from `num1` to get closer to 0.

Now let's run through the approach:

1. We start with `k = 0`. We calculate `x = num1 - k * num2`. At this moment, `x = 42 - 0 * 3 = 42`.

   - Conditions to check:
     - `x.bit_count() <= k`? (Does 42 have less than or equal to 0 bits set to 1? No, it has more.)
     - `k <= x`? (Is 0 less than or equal to 42? Yes, it is.)
     - We don't proceed with this `k` since the first condition fails.
2. Increment `k` to 1. Now, `x = 42 - 1 * 3 = 39`.

   - Conditions to check:
     - `x.bit_count() <= k`? (Does 39 have less than or equal to 1 bits set to 1? No, it has more).
     - `k <= x`? (Is 1 less than or equal to 39? Yes, it is.)
     - We don't proceed with this `k` since the first condition fails.
3. Increment `k` to 2. Now `x = 42 - 2 * 3 = 36`.

   - Conditions to check:
     - `x.bit_count() <= k`? (Does 36 have less than or equal to 2 bits set to 1? No, it has 2).
     - `k <= x`? (Is 2 less than or equal to 36? Yes, it is.)
     - Both conditions pass, so it is possible to reduce 36 to 0 in 2 operations by choosing the right powers of 2.

At this point, our example concludes that the minimum number of operations required to reduce num1 to 0 is 2, considering the choice of `i` we make in each step can actually reduce the number down. In reality, this process repeats, incrementing `k` and checking both conditions until we either find a valid `k` or determine that it's impossible to reach zero. If impossible, the function would return -1. However, in our example, we succeeded with `k = 2`.

Python Solution

```python
from itertools import count

class Solution:
    def makeTheIntegerZero(self, num1: int, num2: int) -> int:
        # Start an infinite loop incrementing k starting from 1
        for k in count(1):
            # Calculate the difference between num1 and k times num2
            difference = num1 - k * num2

            # If the difference becomes negative, we break out of the loop
            if difference < 0:
                break

            # Check if the number of set bits (1-bits) in 'difference' is less than or equal to 'k'
            # and if 'k' is less than or equal to 'difference'
            if difference.bit_count() <= k <= difference:
                # If the condition is satisfied, we return 'k' as the result
                return k

        # If no valid 'k' is found, return -1 indicating the operation is not possible
        return -1

```

Java Solution

```java
class Solution {

    /**
     * Returns the smallest positive k such that the number of 1-bits in num1 - k * num2 is less than or equal to k,
     * and k is less than or equal to num1 - k * num2; otherwise, returns -1 if no such k exists.
     *
     * @param num1 the initial number we're trying to make zero
     * @param num2 the number we subtract from num1, multiplied by k
     * @return the smallest k that satisfies the condition or -1 if no such k exists
     */
    public int makeTheIntegerZero(int num1, int num2) {
        // We start with k = 1 and check for every increasing k value
        for (long k = 1; ; ++k) {
            // Calculate the new number after subtracting k times num2 from num1
            long result = num1 - k * num2;

            // If the result is negative, no further positive k can satisfy the problem's condition
            if (result < 0) {
                break;
            }

            // Check if the number of 1-bits in result is less than or equal to k AND if k is less than or equal to result
            if (Long.bitCount(result) <= k && k <= result) {
                // If the condition is true, return the current value of k
                return (int) k;
            }
        }
        // If the loop completes without returning, then no valid k was found; return -1
        return -1;
    }
}

```

C++ Solution

```cpp
#include <bitset>

class Solution {
public:
    // Method to find the smallest positive integer k such that:
    // 1. num1 - k * num2 is non-negative.
    // 2. The number of set bits (1-bits) in the binary representation
    //    of num1 - k * num2 is less than or equal to k.
    // 3. k is less than or equal to num1 - k * num2.
    int makeTheIntegerZero(int num1, int num2) {

        // Using 'long long' for larger range, ensuring the variables can handle
        // the case when num1 and num2 are large values.
        // 'll' is an alias to 'long long' for convenience.
        using ll = long long;

        // Start with k = 1 and increase it until a valid k is found or
        // the break condition is reached when num1 - k * num2 < 0.
        for (ll k = 1;; ++k) {

            // Calculate the difference between num1 and k * num2.
            ll difference = num1 - k * num2;

            // If the difference is negative, break the loop as the
            // desired conditions cannot be met.
            if (difference < 0) {
                break;
            }

            // Check if the number of set bits (1s) in the binary representation
            // of the difference is <= k, and k is less than or equal to the difference.
            if (__builtin_popcountll(difference) <= k && k <= difference) {
                // If the condition is met, return k as the answer.
                return k;
            }
        }

        // If the loop ends without returning, no valid k was found; return -1.
        return -1;
    }
};

```

Typescript Solution

```typescript
// Method to count the number of set bits (1-bits) in the binary representation of a number.
function countSetBits(num: number): number {
    let count = 0;
    while (num > 0) {
        count += num & 1; // Increment count if the least significant bit is 1.
        num >>>= 1; // Right shift num by 1 bit, using zero-fill right shift.
    }
    return count;
}

// Method to find the smallest positive integer k such that:
// 1. num1 - k * num2 is non-negative.
// 2. The number of set bits (1-bits) in the binary representation of num1 - k * num2 is less than or equal to k.
// 3. k is less than or equal to num1 - k * num2.
function makeTheIntegerZero(num1: number, num2: number): number {
    // Iterate over possible values of k starting from 1 to find the valid k.
    for (let k = 1; ; ++k) {
        // Calculate the difference between num1 and k times num2.
        let difference = num1 - k * num2;

        // If the difference becomes negative, break the loop since we are not going to find a valid k this way.
        if (difference < 0) {
            break;
        }

        // Check if the number of set bits in the binary representation of the difference is <= k and k is <= difference.
        if (countSetBits(difference) <= k && k <= difference) {
            // If the condition is met, return k as the valid result.
            return k;
        }
    }
    // If no valid k is found, return -1 indicating failure to meet the criteria.
    return -1;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided function is `O(num1 / num2)`.

Here's why:

- The function loops over `k`, incrementing it by one each time.
- The loop runs until `x = num1 - k * num2` becomes negative, which happens after approximately `num1 / num2` iterations.
- Each iteration includes calculation of `x`, checking if `x` is smaller than zero, calculation of `x.bit_count()`, and comparison operations. None of these operations have a complexity greater than O(1).
- Since these constant time operations are inside a loop that runs `num1 / num2` times, the overall time complexity is `O(num1 / num2)`.

Space Complexity

The space complexity of the function is `O(1)`.

This is because:

- There are a finite, small number of variables being used (`k`, `x`), and their size does not scale with the input size `num1` or `num2`.
- No additional data structures are being used to store values that scale with the input size.
- Since the space used does not scale with the input size, the space complexity is constant.