191. Number of 1 Bits

Easy Bit Manipulation Divide and Conquer

Leetcode Link

Problem Description

The task is to write a function that receives an unsigned integer's binary representation and calculates the number of '1' bits (also called the Hamming weight) in it. A binary representation means the number is expressed in base 2, consisting of only of '0's and '1's. For example, the binary representation of the decimal number 5 is '101', which has two '1' bits.

It's important to note that in some programming languages, such as Java, there isn't an unsigned integer type, which means that you might have to deal with negative numbers as well. However, for the purpose of counting '1' bits, you can consider the binary representation regardless of whether the integer is signed or unsigned. In 2's complement notation, which Java uses to represent signed integers, negative numbers also have a binary representation that can be used to count the number of '1' bits.

Intuition

The solution approach uses a bit manipulation technique that exploits a nice property of binary numbers: subtracting 1 from a number flips all the bits after the rightmost '1' bit (including the '1' bit). So, if we perform a bitwise AND between the number `n` and `n-1`, we effectively remove the rightmost '1' bit from `n`.

For instance, if `n` is 101100, `n-1` is 101011. The bitwise AND of `n` and `n-1` would be 101000, which removes the rightmost '1' bit from the original number. We repeatedly do this operation and count how many times we perform it until `n` becomes 0. The count gives us the number of '1' bits in the original number since each operation removes exactly one '1' bit.

By following this approach, we can calculate the Hamming weight efficiently. Let's break down the steps involved in the provided solution:

1. Initialize a variable `ans` to zero. This will be used to track the number of '1' bits.
2. Use a while loop to iterate as long as `n` is not zero.
3. Inside the loop, perform the operation `n &= n - 1`, which removes the rightmost '1' bit from `n`.
4. Increment `ans` by one after each bit-removal operation.
5. Once `n` becomes zero, which means all '1' bits have been removed, return `ans`, the count of '1' bits.

This technique is efficient because it directly targets the '1' bits and minimizes the number of operations relative to the number of '1' bits in the binary representation.

Solution Approach

The provided Python solution implements a commonly known algorithm for counting the number of '1' bits in the binary representation of an unsigned integer using bit manipulation.

Here are the key components of the implementation:

- **Bitwise AND Operator (&):** The bitwise AND operator is used to perform the AND operation on each bit of the binary representations of two numbers. It is a fundamental operation used in the given solution to remove the rightmost '1' bit.
- **While Loop:** The solution utilizes a while loop to iterate until the given number `n` is reduced to zero. The condition of the while loop `while n:` takes advantage of the fact that in Python, zero is considered `False` and all other integers are considered `True`. Thus, the loop continues as long as `n` has at least one '1' bit remaining.
- **Bit Manipulation Trick (`n &= n - 1`):** This specific operation is used to clear the least significant '1' bit of the number `n`. In each iteration of the while loop, `n` is updated to `n & (n - 1)`, which removes the rightmost '1' bit from `n`. This is the core step that reduces the value of `n` while also counting the number of '1' bits.
- **Counter Variable (`ans`):** A counter variable `ans` is used to track the number of '1' bits. It is incremented by 1 for each iteration of the while loop, which corresponds to the removal of each '1' bit.

The combination of these elements leads to an elegant and efficient approach to solving the problem. The solution does not require any additional data structures, since the count is maintained in a single integer variable, and the input number `n` is manipulated in place to count the number of '1' bits.

The pattern used here is quite effective for dealing with common bit manipulation problems and is a good example of using bitwise operators to simplify complex operations. Since each operation potentially reduces `n` by eliminating a '1' bit, the number of iterations is equal to the number of '1' bits, making the algorithm run in O(k) time complexity, where k is the number of '1' bits in the binary representation of `n`.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Consider the unsigned integer 11, which has the binary representation `1011`. Let's apply the algorithm step-by-step to calculate the Hamming weight.

1. We initialize the counter `ans` to zero. This will keep track of the number of '1' bits.
2. The binary representation of the integer 11 is `1011`, and it's not zero, so we enter the while loop.
3. We calculate `n-1`. For `n = 1011` (11 in decimal), `n-1` would be `1010` (10 in decimal).
4. Then, we perform `n &= n - 1`, which is `1011 & 1010` resulting in `1010` (10 in decimal), thereby removing the last '1' bit.
5. We increment `ans` by one. `ans` is now 1.
6. `n` is now `1010` (10 in decimal), which is still not zero, so the while loop continues.
7. Again, `n-1` is calculated to be `1001` (9 in decimal).
8. Performing `n &= n - 1` now, which is `1010 & 1001`, results in `1000` (8 in decimal), removing another '1'.
9. Increment `ans` by one again. `ans` is now 2.
10. `n` is now `1000` (8 in decimal), which is still not zero.
11. `n-1` is `0111` (7 in decimal).
12. Performing `n &= n - 1` again results in `1000 & 0111`, which results in `0000` (0 in decimal).
13. We increment `ans` by one last time. `ans` is now 3.
14. Now, `n` is zero, so we exit the while loop.

After exiting the loop, `ans` is 3, which is the number of '1' bits in the binary representation of the unsigned integer 11 (`1011`). Thus, the Hamming weight of 11 is 3.

This technique minimizes the number of operations by ensuring that each iteration removes one '1' bit, leading to a time complexity that is linear with respect to the number of '1' bits in the binary representation.

Python Solution

```python
class Solution:
    def hammingWeight(self, n: int) -> int:
        # Initialize count of set bits to 0
        count_of_set_bits = 0

        # Iterate until all bits are traversed
        while n:

            # Perform bitwise AND operation between n and (n-1)
            # This operation removes the rightmost set bit from n
            n &= n - 1

            # Increment count of set bits
            count_of_set_bits += 1

        # Return the total count of set bits in the integer
        return count_of_set_bits

```

Java Solution

```java
public class Solution {
    /**
     * This method calculates the number of 1-bits in the binary representation of a number.
     * Treats the input number as an unsigned value.
     *
     * @param n - The input integer (considered as unsigned) to count the 1-bits in.
     * @return The number of 1s in the binary representation of n.
     */
    public int hammingWeight(int n) {
        int onesCount = 0; // Store the count of 1-bits encountered

        // Use '!=0' in the condition to ensure we process all bits of n.
        // Since Java does not support unsigned int natively, we interpret n as unsigned by comparing directly to 0.
        while (n != 0) {
            // Apply the bit manipulation trick n & (n - 1) which clears the least significant 1-bit in n.
            n &= n - 1;

            // Increment the count of 1-bits for every 1-bit cleared by the operation above.
            ++onesCount;
        }

        return onesCount; // Return the total count of 1-bits found
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function returns the number of '1' bits in the binary representation of the given unsigned integer.
    int hammingWeight(uint32_t n) {
        int count = 0;  // Initialize a counter for the '1' bits
        while (n) {  // Continue until all bits are traversed
            n &= n - 1;  // Clear the least significant '1' bit
            ++count;  // Increment the counter by one
        }
        return count;  // Return the total count of '1' bits
    }
};

```

Typescript Solution

```typescript
/**
 * Function to count the number of 1 bits in the binary representation of a positive integer.
 * @param n - a positive integer
 * @returns The number of 1's in the binary representation of n.
 */
function hammingWeight(n: number): number {
    // Initialize a count for the number of 1 bits
    let count: number = 0;

    // Continue looping as long as n is not 0
    while (n !== 0) {
        // Apply bitwise AND between n and n-1, which flips the least significant 1 bit of n to 0
        n &= n - 1;

        // Increment the count of 1 bits
        count++;
    }

    // Return the final count of 1 bits in n
    return count;
}

```

Time and Space Complexity

The given Python code defines a function `hammingWeight` that calculates the number of 1s in the binary representation of a given integer `n`. The algorithm works by turning off the rightmost 1-bit of `n` at each step until `n` becomes 0.

Time Complexity

The time complexity of the algorithm is `O(k)`, where `k` is the number of 1-bits in `n`. In the worst case, when `n` is a power of 2, `k` will be logarithmic in the value of `n` because there will be only one 1-bit. In the average case, it will be less than logarithmic since numbers typically have fewer than the maximum possible number of 1-bits. Therefore, in terms of `n`, the time complexity can be considered `O(log n)` because `k` will not exceed the number of bits in `n`, which is the logarithm of `n`.

Space Complexity

The space complexity of the algorithm is `O(1)` because it uses a fixed number of variables (`ans` and `n`) whose size does not depend on the input size.