Documentation on Number of 1 Bits

Problem Overview

- The problem at hand is to determine the number of "set bits" (1s) in the binary representation of a given positive integer. The count of these set bits is known as the Hamming weight of the number. The task is to develop an efficient solution to compute the Hamming weight for a given integer, particularly when the function may be called multiple times.

Problem Constraints

- The input will be a positive integer n.
- The integer n is guaranteed to be in the range: $1 <= n <= 2^{31}$ 1.

Definitions

1. **Set Bit:** A bit in a binary representation of a number that is set to 1.
2. **Hamming Weight:** The number of set bits (1s) in the binary representation of a number.
3. **Bitwise AND Operation:** This operation takes two binary numbers and performs an AND between corresponding bits. The result is 1 only if both bits are 1.
4. **Right Shift Operation:** This operation shifts the bits of a number to the right, discarding the rightmost bit, effectively dividing the number by 2.

Example 1:

- **Input:** n = 11
- **Binary Representation:** 1011
- **Explanation:** There are three 1s in the binary representation of 11. Therefore, the result is 3.

Example 2:

- **Input:** n = 128
- **Binary Representation:** 10000000
- **Explanation:** There is one 1 in the binary representation of 128. Therefore, the result is 1.

Example 3:

- **Input:** n = 2147483645
- **Binary Representation:** 11111111111111111111111111111101
- **Explanation:** There are thirty 1s in the binary representation of 2147483645. Therefore, the result is 30.

Approach

The solution can be approached using bit manipulation techniques, which leverage bitwise operations to efficiently determine the Hamming weight of a number. There are two main approaches:

1. **Basic Approach Using Bitwise Operations:**

- ***Procedure:***
    1. Initialize a counter to keep track of the number of set bits.
    2. Check if the last bit of the number is 1 by performing a bitwise AND operation between the number and 1 (n & 1). If the result is 1, it means the least significant bit is a 1, and we increment the counter.
    3. Right-shift the number by 1 bit to discard the least significant bit and proceed to the next bit.
    4. Repeat steps 2 and 3 until the number becomes 0.
    5. Return the count as the result.
- ***Time Complexity:*** O(log n), where n is the input number. This is because we are iterating over every bit in the binary representation of the number.
- ***Space Complexity:*** O(1) since we are only using a constant amount of extra space.

2. Optimized Approach Using Bit Manipulation:

- ***Procedure:***
    1. Instead of checking each bit one by one, we can optimize the process using the observation that n = n & (n 1) removes the lowest set bit in n. This operation reduces the number of set bits directly, making it more efficient.
    2. Initialize a counter to track the number of set bits.
    3. While the number is not 0, apply the operation n = n & (n 1) and increment the counter each time.
    4. The operation continues until the number becomes 0.
    5. Return the counter as the result.
- ***Time Complexity:*** O(k), where k is the number of set bits in the binary representation of n. This approach can be more efficient when the number of set bits is significantly smaller than the total number of bits in n.
- ***Space Complexity:*** O(1) since no extra space is required apart from the counter.

Edge Cases

1. **Smallest Input:** For n = 1, the binary representation is 1, so the output should be 1.
2. **Power of Two:** For numbers like 128, which are powers of two, there is exactly one set bit in the binary representation.
3. **Large Input:** For large values close to the maximum value of n, such as n = 2^{31} 1, the binary representation will have many set bits (31 ones in total), so the algorithm should handle this case efficiently.

Optimization Considerations

- If this function is called multiple times with different values of n, the optimized approach using n = n & (n 1) is recommended. This optimization reduces the number of iterations to the number of set bits, rather than iterating over all bits in the number. This makes the solution faster for numbers with fewer set bits.

Follow-up Discussion

- To further optimize the solution, we could precompute the Hamming weight for a smaller range of integers and store the results in a lookup table (bitwise manipulation cache). This would allow us to quickly retrieve the number of set bits for common values without recomputing each time. However, this would increase the space complexity and may not be necessary unless the function is called an extremely large number of times.

Conclusion

- The problem of finding the Hamming weight of an integer can be solved efficiently using bit manipulation techniques. The choice of approach depends on the context of the problem—whether the function is called once or many times, and whether the input number has many or few set bits. Both approaches offer a time-efficient solution, but the bit manipulation trick using n = n & (n 1) is generally faster for numbers with fewer set bits.