Number of 1 Bits

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

For example, the 32-bit integer '11' has binary representation `00000000000000000000000000001011`, so the function should return 3.

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 1

```
public static int hammingWeight(int n) {
        int ones = 0;
        while(n!=0) {
            ones = ones + (n & 1);
            n = n>>>1;
        }
        return ones;
    }
```

- An Integer in Java has 32 bits, e.g. 00101000011110010100001000011010.
- To count the 1s in the Integer representation with put the input int n in bit AND with 1 (that is represented as 00000000000000000000000000000001, and if this operation result is 1, that means that the last bit of the input integer is 1. Thus we add it to the 1s count.

> ones = ones + (n & 1);

- Then we shift the input Integer by one on the right, to check for the next bit.

> n = n>>>1;

We need to use bit shifting unsigned operation >>> (while >> depends on sign extension)

- We keep doing this until the input Integer is 0.

In Java we need to put attention on the fact that the maximum integer is 2147483647. Integer type in Java is signed and there is no unsigned int. So the input 2147483648 is represented in Java as -2147483648 (in java int type has a cyclic representation, that means **Integer.MAX*VALUE+1==Integer.MIN*VALUE**). This force us to use

> n!=0

in the while condition and we cannot use

> n>0

because the input 2147483648 would correspond to -2147483648 in java and the code would not enter the while if the condition is n>0 for n=2147483648.

written by fabrizio3 original link here

Solution 2

Each time of "n &= n - 1", we delete one '1' from n.

```
int hammingWeight(uint32_t n)
{
    int res = 0;
    while(n)
    {
        n &= n - 1;
        ++ res;
    }
    return res;
}
```

Another several method of O(1) time.

Add 1 by Tree:

```
// This is a naive implementation, shown for comparison, and to help in understand
ing the better functions.
// It uses 24 arithmetic operations (shift, add, and).
int hammingWeight(uint32_t n)
{
    n = (n & 0x55555555) + (n >>  1 & 0x55555555); // put count of each  2 bits i
nto those  2 bits
    n = (n & 0x33333333) + (n >>  2 & 0x33333333); // put count of each  4 bits i
nto those  4 bits
    n = (n & 0x0F0F0F0F) + (n >>  4 & 0x0F0F0F0F); // put count of each  8 bits i
nto those  8 bits
    n = (n & 0x00FF00FF) + (n >>  8 & 0x00FF00FF); // put count of each 16 bits i
nto those 16 bits
    n = (n & 0x0000FFFF) + (n >> 16 & 0x0000FFFF); // put count of each 32 bits i
nto those 32 bits
    return n;
}

// This uses fewer arithmetic operations than any other known implementation on ma
chines with slow multiplication.
// It uses 17 arithmetic operations.
int hammingWeight(uint32_t n)
{
    n -= (n >> 1) & 0x55555555; //put count of each 2 bits into those 2 bits
    n = (n & 0x33333333) + (n >> 2 & 0x33333333); //put count of each 4 bits into
those 4 bits
    n = (n + (n >> 4)) & 0x0F0F0F0F; //put count of each 8 bits into those 8 bits
    n += n >> 8; // put count of each 16 bits into those 8 bits
    n += n >> 16; // put count of each 32 bits into those 8 bits
    return n & 0xFF;
}

// This uses fewer arithmetic operations than any other known implementation on ma
chines with fast multiplication.
// It uses 12 arithmetic operations, one of which is a multiply.
int hammingWeight(uint32_t n)
{
    n -= (n >> 1) & 0x55555555; // put count of each 2 bits into those 2 bits
    n = (n & 0x33333333) + (n >> 2 & 0x33333333); // put count of each 4 bits int
o those 4 bits
    n = (n + (n >> 4)) & 0x0F0F0F0F; // put count of each 8 bits into those 8 bit
s
    return n * 0x01010101 >> 24; // returns left 8 bits of x + (x<<8) + (x<<16) +
(x<<24)
}
```

——From Wikipedia.

written by makuiyu original link here

Solution 3

```
int hammingWeight(uint32_t n) {
    int count = 0;

    while (n) {
        n &= (n - 1);
        count++;
    }

    return count;
}
```

n & (n - 1) drops the lowest set bit. It's a neat little bit trick.

Let's use n = 00101100 as an example. This binary representation has three 1s.

If n = 00101100, then n - 1 = 00101011, so n & (n - 1) = 00101100 & 00101011 = 00101000. Count = 1.

If n = 00101000, then n - 1 = 00100111, so n & (n - 1) = 00101000 & 00100111 = 00100000. Count = 2.

If n = 00100000, then n - 1 = 00011111, so n & (n - 1) = 00100000 & 00011111 = 00000000. Count = 3.

n is now zero, so the while loop ends, and the final count (the numbers of set bits) is returned.

written by housed original link here

From Leetcoder.