191. Number of 1 Bits

April 15, 2016 | 121.1K views

Average Rating: 4.80 (96 votes)

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

Example 1:

```
Input: 00000000000000000000000000001011
Output: 3
Explanation: The input binary string 00000000000000000000000000001011 has a total o
f three '1' bits.
```

Example 2:

```
Input: 00000000000000000000000010000000
Output: 1
Explanation: The input binary string 00000000000000000000000010000000 has a total of
```

Example 3:

```
Input: 11111111111111111111111111111101
Output: 31
Explanation: The input binary string 11111111111111111111111111111101 has a total of
```

Note:

- Note that in some languages such as Java, there is no unsigned integer type. In this case, the input will be given as signed integer type and should not affect your implementation, as the internal binary representation of the integer is the same whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using 2's complement notation. Therefore, in **Example 3** above the input represents the signed integer `-3`.

Follow up:

If this function is called many times, how would you optimize it?

Solution

Approach #1 (Loop and Flip) [Accepted]

Algorithm

The solution is straight-forward. We check each of the 32 bits of the number. If the bit is 1, we add one to the number of 1-bits.

We can check the i^{th} bit of a number using a *bit mask*. We start with a mask $m = 1$, because the binary representation of 1 is,

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001$$

Clearly, a logical AND between any number and the mask 1 gives us the least significant bit of this number. To check the next bit, we shift the mask to the left by one.

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010$$

And so on.

Java

```java
public int hammingWeight(int n) {
    int bits = 0;
    int mask = 1;
    for (int i = 0; i < 32; i++) {
        if ((n & mask) != 0) {
            bits++;
        }
        mask <<= 1;
    }
    return bits;
}
```

Complexity Analysis

The run time depends on the number of bits in n. Because n in this piece of code is a 32-bit integer, the time complexity is $O(1)$.

The space complexity is $O(1)$, since no additional space is allocated.

Approach #2 (Bit Manipulation Trick) [Accepted]

Algorithm

We can make the previous algorithm simpler and a little faster. Instead of checking every bit of the number, we repeatedly flip the least-significant 1-bit of the number to 0, and add 1 to the sum. As soon as the number becomes 0, we know that it does not have any more 1-bits, and we return the sum.

The key idea here is to realize that for any number n, doing a bit-wise AND of n and $n - 1$ flips the least-significant 1-bit in n to 0. Why? Consider the binary representations of n and $n - 1$.

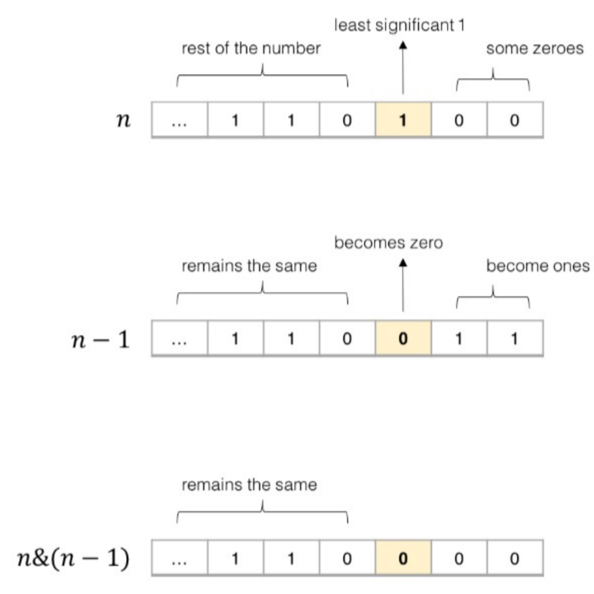

Figure 1. AND-ing n and $n - 1$ flips the least-significant 1-bit to 0.

In the binary representation, the least significant 1-bit in n always corresponds to a 0-bit in $n - 1$. Therefore, anding the two numbers n and $n - 1$ always flips the least significant 1-bit in n to 0, and keeps all other bits the same.

Using this trick, the code becomes very simple.

Java

```java
public int hammingWeight(int n) {
    int sum = 0;
    while (n != 0) {
        sum++;
        n &= (n - 1);
    }
    return sum;
}
```

Complexity Analysis

The run time depends on the number of 1-bits in n. In the worst case, all bits in n are 1-bits. In case of a 32-bit integer, the run time is $O(1)$.

The space complexity is $O(1)$, since no additional space is allocated.

Analysis written by: @noran.

Rate this article:

Previous Next

Comments: 81

Sort By

Type comment here... (Markdown is supported)

Preview Post

shamimsa ★45 November 29, 2018 1:47 PM

python
return bin(n).count('1')

44 Share Reply

SHOW 4 REPLIES

amqbxr ★57 August 15, 2018 9:00 PM

Please add Swift as a choice for submitting code in the editor for this challenge

23 Share Reply

rylexr ★78 October 15, 2018 8:52 AM

My solution in Java:

```
public class Solution {
    public int hammingWeight(int n) {
        int count = 0;
```

Read More

11 Share Reply

SHOW 3 REPLIES

TenyoChen ★13 November 9, 2017 2:32 AM

My algorithm doesn't need any bit operation.
It was inspired by "Short Division by Two with Remainder".

```
def hammingWeight(self, n):
        ans=0
```

Read More

14 Share Reply

SHOW 3 REPLIES

cannotbekilled ★13 February 27, 2019 5:26 PM

java

```
return Integer.bitCount(n);
```

13 Share Reply

SHOW 3 REPLIES

mhelvens ★685 June 27, 2019 1:39 AM

Stating the time-complexity as `O(1)` here is misleading.

Yes, we know there are never more than 32 bits, which can be considered a constant factor. But by that reasoning, every solution to a problem with explicit input constraints is `O(1)`. *Binary search* in a sorted array will typically have far fewer than 32 iterations.

Read More

14 Share Reply

SHOW 2 REPLIES

18366100262 ★3 August 1, 2018 2:53 PM

if(n>Integer.MAX_VALUE)return 1; for java

3 Share Reply

jontyc ★3 July 28, 2018 2:50 AM

Shouldn't the solution be n>>=1;

`public int hammingWeight(int n) {`
`int bits = 0;`
`int mask = 1;`

Read More

3 Share Reply

SHOW 2 REPLIES

mellone ★23 May 24, 2018 11:10 AM

um, this is an easier python one-liner: `return bin(n).count('1')`

4 Share Reply

Kaamil17 ★1 April 18, 2018 5:20 PM

well you can do like this. so easy...
'''
public int hammingWeight(int n) {
return Integer.bitCount(n);
}

Read More

1 Share Reply