461. Hamming Distance

Jan. 3, 2020 | 17.3K views

Previous Next

Average Rating: 4.89 (37 votes)

The Hamming distance between two integers is the number of positions at which the corresponding bits are different.

Given two integers `x` and `y`, calculate the Hamming distance.

Note:
$0 \leq x, y < 2^{31}$.

Example:

```
Input: x = 1, y = 4

Output: 2

Explanation:
1   (0 0 0 1)
4   (0 1 0 0)
      ↑   ↑

The above arrows point to positions where the corresponding bits are different.
```

Solution

Intuition

Hamming distance is an interesting metric that is widely applied in several domains, *e.g.* in coding theory for error detection, in information theory for quantifying the difference between strings.

> The Hamming distance between two **integer** numbers is the number of positions at which the corresponding bits are different.

Given the above definition, it might remind one of the bit operation called XOR which outputs `1` *if and only if* the input bits are different.

> As a result, in order to measure the hamming distance between `x` and `y`, we can first do `x XOR y` operation, then we simply count the number of bit `1` in the result of XOR operation.

We now convert the original problem into a bit-counting problem. There are several ways to count the bits though, as we will discuss in the following sections.

Approach 1: Built-in BitCounting Functions

Intuition

First of all, let us talk of the elephant in the room. As one can imagine, we have various built-in functions that could count the bit `1` for us, in all (or at least most of) programming languages. So if this is the task that one is asked in a project, then one should probably just go for it, rather than reinventing the wheel. We given two examples in the following.

Now, since this is a LeetCode problem, some of you would argue that using the built-in function is like *"implementing a LinkedList with LinkedList"*, which we fully second as well. So no worry, we will see later some fun hand-crafted algorithms for bit counting.

Algorithm

Java | Python

```
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        return bin(x ^ y).count('1')
```

Complexity Analysis

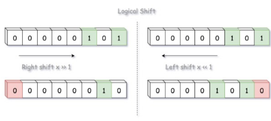
- Time Complexity: $\mathcal{O}(1)$.
  - There are two operations in the algorithm. First, we do the XOR operation which takes a constant time.
  - Then, we call the built-in bitCount function. In the worst scenario, the function would take $\mathcal{O}(k)$ time where k is the number of bits for an integer number. Since the integer type is of fixed size in both Python and Java, the overall time complexity of the algorithm becomes constant, regardless the input numbers.
- Space Complexity: $\mathcal{O}(1)$, a temporary memory of constant size is consumed, to hold the result of XOR operation.
  - We assume that the built-in function also takes a constant space.

Approach 2: Bit Shift

Intuition

In order to count the number of bit `1`, we could *shift* each of the bit to either the leftmost or the rightmost position and then check if the bit is one or not.

More precisely, we should do the logical shift where zeros are shifted in to replace the discarded bits.



Here we adopt the right shift operation, where each bit would has its turn to be shifted to the rightmost position. Once shifted, we then check if the rightmost bit is one, which we can use either the `modulo` operation (i.e. `1 % 2`) or the bit AND operation (i.e. `1 & 1`). Both operations would ***mask out*** the rest of the bits other than the rightmost bit.

Algorithm

Java | Python

```
class Solution(object):
    def hammingDistance(self, x, y):
        """
        :type x: int
        :type y: int
        :rtype: int
        """
        xor = x ^ y
        distance = 0
        while xor:
            # mask out the rest bits
            if xor & 1:
                distance += 1
            xor = xor >> 1
        return distance
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(1)$, since the Integer is of fixed size in Python and Java, the algorithm takes a constant time. For an Integer of 32 bit, the algorithm would take *at most* 32 iterations.
- Space Complexity: $\mathcal{O}(1)$, a constant size of memory is used, regardless the input.

Approach 3: Brian Kernighan's Algorithm

Intuition

In the above approach, one might wonder that *"rather than shifting the bits one by one, is there a faster way to count the bits of one ?"*. And the answer is yes.

> If one is asked to count the bits of one, as humans, rather than mechanically examining each bit, we could ***skip*** the bits of zero in between the bits of one, *e.g.* `10001000`.

In the above example, after encountering the first bit of one at the rightmost position, it would be more efficient if we just jump at the next bit of one, skipping all the zeros in between.

This is the basic idea of the Brian Kernighan's bit counting algorithm, which applies some smart bit and arithmetic operations to ***clear*** the rightmost bit of one. Here is the secret recipe.

> When we do AND bit operation between `number` and `number-1`, the rightmost bit of one in the original `number` would be cleared.

Based on the above idea, we then can count the bits of one for the input of `10001000` in 2 iterations, rather than 8.

Algorithm

Java | Python

```
class Solution:
    def hammingDistance(self, x, y):
        xor = x ^ y
        distance = 0
        while xor:
            distance += 1
            # remove the rightmost bit of '1'
            xor = xor & (xor - 1)
        return distance
```

Note, according to the online book of Bit Twiddling Hacks, the algorithm was published as an exercise in 1988, in the book of *the C Programming Language 2nd Ed.* (by **Brian W. Kernighan** and Dennis M. Ritchie), though on April 19, 2006 Donald Knuth pointed out that this method *"was first published by Peter Wegner in CACM 3 (1960), 322. (Also discovered independently by Derrick Lehmer and published in 1964 in a book edited by Beckenbach.)"*. By the way, one can find many other tricks about bit operations in the aforementioned book.

Complexity Analysis

- Time Complexity: $\mathcal{O}(1)$.
  - Similar as the approach of bit shift, since the size (i.e. bit number) of integer number is fixed, we have a constant time complexity.
  - However, this algorithm would require less iterations than the bit shift approach, as we have discussed in the intuition.
- Space Complexity: $\mathcal{O}(1)$, a constant size of memory is used, regardless the input.

Rate this article:

Previous Next

Comments: 7

Sort By

Type comment here... (Markdown is supported)

Preview Post

2020c ★ 0 June 28, 2020 5:17 AM

I tried this approach

```
public int hammingDistance(int x, int y) {
        int sum = 0;
        for(int i = 0 ; i < 32; i++)
```

Read More

3 Share Reply

CYLik ★ 0 May 6, 2020 5:18 PM

Why is the Time Complexity constant? I still can't wrap my head around this. Is it because the length of the bits of an integer constant?

3 Share Reply

SHOW 2 REPLIES

ivgrishchuck ★ 1 July 5, 2020 3:55 AM

```
int hammingDistance(int x, int y) {
    return __builtin_popcount(x ^ y);
}
```

1 Share Reply

willye ★ 89 April 23, 2020 12:23 PM

For the Python solution of Approach 2, you can use `xor >>= 1` instead of `xor = xor >> 1`

1 Share Reply

fnaf ★ 0 January 4, 2020 11:44 AM

Hi @andvary, @liaison great article thanks. I have a question related to bit manipulation. What is the best way to get the left-most bit from a binary representation i.e Given a number e.g 21 the representation is 1 0 1 0 1 (16 + 4 + 1) the output should be 16. Right now I do this in my code:

```
highest_power_for_hammingDistance(x)
```

Read More

1 Share Reply

SHOW 1 REPLIES

133c7 ★ 17 July 14, 2020 4:00 AM

The Java solution in the second approach seems to be using an arithmetic shift (`>>`) instead of logical shift (`>>>`). Why does it work despite that? What am I missing?

0 Share Reply

pmane6422 ★ 255 July 8, 2020 12:00 AM

Right shift and check if one number is odd and other is even

```
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        count = 0
```

Read More

0 Share Reply