Total Hamming Distance

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

Now your job is to find the total Hamming distance between all pairs of the given numbers.

Example:

```
Input: 4, 14, 2

Output: 6

Explanation: In binary representation, the 4 is 0100, 14 is 1110, and 2 is 0010 (ju
st
showing the four bits relevant in this case). So the answer will be:
HammingDistance(4, 14) + HammingDistance(4, 2) + HammingDistance(14, 2) = 2 + 2 + 2
 = 6.
```

Note:

1. Elements of the given array are in the range of `0` to `10^9`
2. Length of the array will not exceed `10^4`.

Solution 1

For each bit position 1-32 in a 32-bit integer, we count the number of integers in the array which have that bit set. Then, if there are n integers in the array and k of them have a particular bit set and (n-k) do not, then that bit contributes k*(n-k) hamming distance to the total.

```
public int totalHammingDistance(int[] nums) {
    int total = 0, n = nums.length;
    for (int j=0;j<32;j++) {
        int bitCount = 0;
        for (int i=0;i<n;i++)
            bitCount += (nums[i] >> j) & 1;
        total += bitCount*(n - bitCount);
    }
    return total;
}
```

written by compton_scatter original link here

Solution 2

```
def totalHammingDistance(self, nums):
    return sum(b.count('0') * b.count('1') for b in zip(*map('{:032b}'.format, nu
ms)))
```

written by StefanPochmann original link here

Solution 3

1. Problem

The problem is to find the total Hamming distance between all pairs of the given numbers.

2. Thinking process

2.1 For one pair

When you calculate Hamming distance between x and y, you just

1. calculate p = x ^ y;
2. count the number of 1's in p

The distance from x to y is as same as y to x.

2.2 Trival approach

For a series of number: a1, a2, a3,..., an

Use the approach in 2.1
(suppose distance(x, y) is the Hamming distance between x and y):

For a1, calculate S(1) = distance(a1, a2)+distance(a1, a3)+ ... +distance(a1, an)
For a2, calculate S(2) = distance(a2, a3)+distance(a2, a4)+ ... +distance(a2, an)
.....
For a(n - 1), calculate S(n - 1) = distance(a(n - 1), a(n))

Finally , **S = S(1) + S(2) + ... + S(n - 1)** .

The function distance is called **1 + 2 + ... + (n - 1) = n(n - 1)/2** times! That's too much!

2.3 New idea

The total Hamming distance is constructed **bit by bit** in this approach.

Let's take a series of number: **a1, a2, a3,..., an**

Just think about all the **Least Significant Bit (LSB)** of a(k) (1 ≤ k ≤ n).

How many Hamming distance will they bring to the total?

1. If a pair of number has same LSB, the total distance will get 0.
2. If a pair of number has different LSB, the total distance will get 1.

For all number **a1, a2, a3,..., a(n)**, if there are **p** numbers have **0 as LSB (put in set M)**, and **q** numbers have **1 for LSB (put in set N)** .

There are **2 situations**:

Situation 1. If the **2 number in a pair both comes from M (or N)** , the total will get **0**.

Situation 2. If the **1 number in a pair comes from M, the other comes from N**, the total will get **1**.

Since **Situation 1** will add **NOTHING** to the total, we only need to think about **Situation 2**

How many pairs are there in Situation 2?

We choose **1 number from M (p possibilities)** , and **1 number from N (q possibilities)**.

The total possibilities is **p × q = pq** , which means

> **The total Hamming distance will get pq from LSB.**

If we **remove the LSB of all numbers (right logical shift)** , the same idea can be used **again and again until all numbers becomes zero**

2.4 Time Complexity

In each loop, we need to **visit all numbers in nums** to **calculate how many numbers has 0 (or 1) as LSB**.

If the biggest number in nums[] is K, **the total number of loop is [logM] + 1** .

So, **the total time complexity of this approach is O(n)** .

---

3. Code

```
class Solution {
public:
    int totalHammingDistance(vector<int>& nums) {
        int size = nums.size();
        if(size < 2) return 0;
        int ans = 0;
        int *zeroOne = new int[2];
        while(true)
        {
            int zeroCount = 0;
            zeroOne[0] = 0;
            zeroOne[1] = 0;
            for(int i = 0; i < nums.size(); i++)
            {
                if(nums[i] == 0) zeroCount++;
                zeroOne[nums[i] % 2]++;
                nums[i] = nums[i] >> 1;
            }
            ans += zeroOne[0] * zeroOne[1];
            if(zeroCount == nums.size()) return ans;
        }
    }
};
```

written by KJer original link here

From Leetcoder.