477. Total Hamming Distance

Jan. 15, 2017 | 10.9K views

Average Rating: 3.78 (9 votes)

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 (just
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 =
```

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

Approach #1 Brute Force [Time Limit Exceeded]

Intuition

1. Check all the unique pairs of elements for differing bits.
2. `xor` of two bits is `1` if they are not the same, and `0` otherwise.

Algorithm

Bitwise `xor` of two numbers will give a bitwise representation of where the bits of two numbers differ. Such bit positions are represented by a `1` bit in the resultant. For example:

```
    0010 1101    ==  45
  ^ 0100 1010    ==  74
-----------------
    0110 0111
     ^^   ^^^    => 5 differing bits
```

Hence the numbers 45 and 74 have five differing bits. Thus the Hamming Distance between them is 5.

For each of the $\approx n^2/2$ pairs of elements, simply apply bitwise `xor` to them to find out a resultant representation which tells us the differing bits. Count the `1` bits to find the Hamming Distance for that pair of elements. Sum over all pairs to get the total Hamming Distance.

C++ Copy

```
int totalHammingDistance(vector<int>& nums)
{
    int ans = 0;

    if (nums.empty())
        return ans;

    for (int i = 0; i < nums.size() - 1; i++)                 // for all unique pairs of elements
        for (int j = i + 1; j < nums.size(); j++)
            ans += __builtin_popcount(nums[i] ^ nums[j]);     // count number of 1 bits in xor resultant

    return ans;
}
```

NOTE: The `__builtin_popcount()` method is an internal built-in function available in **C** (and hence by extension **C++**) which gives the count of `1` bits for an `int` type argument. Being a low level built-in method, it is understandably faster than running a hand rolled loop. As an alternative, you can iterate over all the bits of the number and count the `1` bits. Take a look at the code for Approach #2 for hints on how to achieve that.

Complexity Analysis

- Time complexity: $O(n^2 \cdot log_2 V) \simeq O(n^2)$.
  - There are exactly $\binom{n}{2} = {}^nC_2 = \frac{n \cdot (n-1)}{2}$ unique pairs of elements for an n element array.
  - Each of these pairs, when `xor`ed, result in a resultant number which is $\lceil log_2 V \rceil$ bits long. Here V is the largest value any of the elements can ever take.
  - We iterate over these many bits to count the number of `1` bits. In our case, since all elements are $\leq 10^9$, the value $\lceil log_2 V \rceil = 30$ is a small constant. Hence counting the `1` bits takes place in nearly constant (i.e. $O(1)$) time.
- Space complexity: $O(1)$ constant space.

Approach #2 Loop over the bits! [Accepted]

Intuition

Looping over all possible combinations of two element pairs, increases quadratically over the size of the input. If we could instead loop over the bits of the elements (which is constant), we could shave off an input dimension from our runtime complexity.

Algorithm

Say for any particular bit position, count the number of elements with this bit **ON** (i.e. this particular bit is `1`). Let this count be k. Hence the number of elements with this bit **OFF** (i.e. `0`) is $(n - k)$ (in an n element array).

Certainly unique pairs of elements exists where one element has this particular bit **ON** while the other element has this **OFF** (i.e. this particular bit differs for the two elements of this pair).

> We can argue that every such pair contributes one unit to the Hamming Distance for this particular bit.

We know that the count of such unique pairs is ${}^kC_1 * {}^{n-k}C_1 = k \cdot (n - k)$ for this particular bit. Hence Hamming Distance for this particular bit is $k \cdot (n - k)$.

For each of the $\lceil log_2 V \rceil$ bits that we can check, we can calculate a Hamming Distance pertaining to that bit. Taking sum over the Hamming Distances of all these bits, we get the total Hamming Distance.

C++ Copy

```
int totalHammingDistance(vector<int>& nums)
{
    if (nums.empty())
        return 0;

    int ans = 0, n = nums.size();
    vector<int> cnt(32, 0);         // count of elements with a particular bit ON

    for (auto num : nums) {         // loop over every element
        int i = 0;
        while (num > 0) {           // check every bit
            cnt[i] += (num & 0x1);
            num >>= 1;
            i++;
        }
    }

    for (auto& k : cnt) {           // loop over every bit count
        ans += k * (n - k);
    }

    return ans;
}
```

NOTE: You *can switch the order of the loops* while counting `1` bits without affecting complexity. However it might give some performance changes due to locality of reference and the resultant cache hits/misses.

Complexity Analysis

- Time complexity: $O(n \cdot log_2 V) \simeq O(n)$. Runtime performance is limited by the double loop where we are counting elements for particular bits. In our case, since all elements are $\leq 10^9$, the value $\lceil log_2 V \rceil = 30$ is a small constant. Thus the inner loop runs in nearly constant time.
- Space complexity: $O(log_2 V) \rightarrow O(1)$ extra space.
  - For each of the $\lceil log_2 V \rceil$ bits, we need to maintain a count which is later used to calculate the Hamming Distance for that bit. Since $\lceil log_2 V \rceil \approx 32$ is a small constant in our case, that takes nearly constant extra space.
  - Another thing to notice, is that if we **switch the order of the double loop**, we can do away with storing the count and calculate the Hamming Distance for that bit then and there. That results in only constant extra space being used.

Bonus!

This question is a perfect example of how vectorized operations can result in small, elegant and good performance code. Take a look at this slick **Python** solution to this problem (by @StefanPochmann):

Python Copy

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

The `*` operator turns the list of `32`-bit wide binary strings returned by `map` into individual arguments to the `zip` method. The `zip` method vectorizes the string arguments to create a list of vectors (each of which is a vector `b` of particular bits from every element in the input array; There are `32` such vectors of size `len(nums)` each, in this list). Finally we use the same technique as Approach #2 to calculate the total Hamming Distance.

Rate this article:

Comments: 4

Sort By

Type comment here... (Markdown is supported)

Preview Post

sha256pki ★ 551 July 29, 2017 11:37 AM

"We know that the count of such unique pairs is kC1∗n−kC1∗k=(n−k)()^kC_1 * ()^{n-k}C_1 ∗ k=k\cdot (n-k)kC1∗n−kC1∗k=(n−k) for this particular bit."....can you please help understand reasoning for it? The way I see it, if there're k numbers which have "ith" bit set, there're (n-k) numbers which have "ith" bit unset. Pair each of those k numbers with each of (n-k) numbers - there're total of k * (n-k) pairs. Each such pair contributes hamming distance of 1

Read More

4 Share Reply

SHOW 1 REPLY

zdxiq125 ★ 162 March 24, 2020 3:37 PM

Java version.

```
class Solution {
    public int totalHammingDistance(int[] nums) {
        // counting of bit 1 for i-th least significant bit
```

Read More

2 Share Reply

indrajohn ★ 21 June 8, 2020 12:26 PM

O(N*logV) solution and explanation is awesome. Loved it.

0 Share Reply

MaidaWu ★ 0 March 18, 2020 7:32 AM

What a perfect solution! Couldn't believe that this can be solved within linear time.

0 Share Reply