Total Hamming Distance – Documentation

Table of Contents

1. Problem Statement

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

Given an integer array nums, return the sum of Hamming distances between all pairs of the integers in nums.

2. Intuition

Comparing every possible pair of integers in the array using brute force would work, but would be too slow for large arrays.

Instead, we can take advantage of bitwise properties. At each bit position (0 through 31 for 32-bit integers), we can count how many numbers have a 1 and how many have a 0, and use that to calculate the total differences at that position.

3. Key Observations

- A Hamming distance between two numbers can be computed using XOR and counting 1s.
- But computing XOR for every pair ($O(n^2)$) is inefficient for large arrays.
- For each bit position:
    - If count_ones is the number of elements with bit 1 at that position, and count_zeros is the rest:
    - Then, count_ones * count_zeros gives the total Hamming distance contributed by that bit across all pairs.

4. Approach

- Initialize a variable total to 0.
- For each of the 32 bit positions (0 to 31):
    - Count how many numbers in nums have the current bit set (1) → count_ones.
    - Total number of pairs at that bit with different bits = count_ones * (n - count_ones).
    - Add this to the overall total.
- Return total.

This bitwise method ensures we calculate the total distance in linear time.

5. Edge Cases

Case	Expected Behavior
Empty array	Not allowed by constraints ($n \geq 1$)
Array with one element	Hamming distance is 0
All elements identical	Hamming distance is 0
All elements completely different in bits	Maximum possible distances
Very large values (close to 2^{31})	Still works within 32 bits

6. Complexity Analysis

✅ Time Complexity:

$O(32 * n) = O(n)$

- We scan each bit position for all numbers in the array.

✅ Space Complexity:

$O(1)$

- Only a few integer variables used regardless of input size.

7. Alternative Approaches

Brute Force (Inefficient):

- Compare each pair (i, j) and compute XOR, then count the bits set to 1.
- Time Complexity: $O(n^2 * 32)$
- Not suitable for $n > 10^3$.

8. Test Cases

✅ Example 1:

Input: [4, 14, 2]

Binary: 0100, 1110, 0010

Output: 6

Explanation:

(4,14) = 2, (4,2) = 2, (14,2) = 2 → Total = 6

✅ Example 2:

Input: [4, 14, 4]

Output: 4

Explanation:

(4,14) = 2, (4,4) = 0, (14,4) = 2 → Total = 4

✅ Example 3:

Input: [1, 2, 3]

Output: 4

Explanation:

(1,2) = 2, (1,3) = 1, (2,3) = 1 → Total = 4

✅ Example 4:

Input: [0, 0, 0]

Output: 0

Explanation: All bits are same → No differences

9. Final Thoughts

- This problem showcases how bitwise operations can lead to elegant optimizations.
- The bit-counting approach is efficient, easy to implement, and works within constraints.
- Great example of turning a seemingly complex pairwise problem into a linear scan using bit manipulation.