Documentation for Hamming Distance Calculation

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.

Example 1

Input: x = 1, y = 4
Binary Representation:

1 -> 0001
4 -> 0100

Different bit positions: (Marked with ↑)

↑ ↑

Output: 2

Example 2

Input: x = 3, y = 1

Binary Representation:

3 -> 0011

1 -> 0001

Different bit positions: (Marked with ↑)

↑

Output: 1

Constraints

- $0 \le x, y \le 2^{31} - 1$ (Both numbers fit within a 31-bit integer.)

2. Intuition

The Hamming distance counts the number of positions where two numbers differ in their binary representation. The best way to identify differences in bits is through the XOR (^) operation.

3. Key Observations

- The XOR (^) operation results in a binary number where 1 indicates differing bit positions.
- Counting the number of 1s in this XOR result gives the Hamming distance.

Example:

For x = 1 and y = 4:

1 -> 0001

4 -> 0100

XOR -> 0101 (count of '1' = 2)

Thus, the Hamming distance is 2.

4. **Approach**

- Compute x XOR y (x ^ y), which highlights the different bits.
- Count the number of 1s in the result using bin(x ^ y).count('1').

5. **Edge Cases**

- Same numbers (x == y): The distance should be 0 because there are no different bits.
- Minimum input (x = 0, y = 0): The output should be 0.
- Maximum input (x = 2^31-1, y = 0): Ensures handling of large numbers correctly.
- One-bit difference (x = 1, y = 2): Tests if the function identifies small changes.

6. **Complexity Analysis**

Time Complexity

- O(1) → The bitwise XOR operation and counting 1s in a 31-bit integer are constant time operations.

Space Complexity

- O(1) → No extra space is used apart from a few integer variables.

7. **Alternative Approaches**

Method 1: Using Bit Manipulation (n & (n - 1))

Instead of counting 1s using bin().count('1'), we can repeatedly clear the rightmost 1 bit using n & (n - 1).

Implementation:

```
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        diff = x ^ y
        count = 0
        while diff:
            count += 1
            diff &= diff - 1  # Clears the lowest set bit
        return count
```

Time Complexity: O(k), where k is the number of 1s in the XOR result (at most 31). Space Complexity: O(1).

8. Test Cases

Test Case	Input (x, y)	Expected Output	Explanation
1	(1, 4)	2	0001 vs 0100
2	(3, 1)	1	0011 vs 0001
3	(7, 8)	4	0111 vs 1000
4	(0, 0)	0	0000 vs 0000
5	(15, 0)	4	1111 vs 0000

9. Final Thoughts

- Why XOR? → The XOR operation is ideal for detecting bit differences efficiently.
- Alternative Approach? → Bitwise manipulation offers another way to count set bits without converting to a string.
- Efficiency? → The approach runs in constant time and is optimal for 31-bit integers.

This method provides a fast, efficient, and easy-to-read solution for computing the Hamming Distance between two numbers. 🚀