461. Hamming Distance

Easy Bit Manipulation

Leetcode Link

Problem Description

The task is to calculate the Hamming distance between two integers. The Hamming distance is simply the number of positions at which the corresponding bits of the two numbers differ. For example, if we have two numbers, say 1 (`01` in binary) and 4 (`100` in binary), their Hamming distance is 2 because they differ at two positions: the first and third bits (when looking at the binary representation from right to left with leading zeros taken into account for aligning the bit patterns).

This is a problem that deals with bit manipulation, a common topic in computer science that focuses on operations directly on binary digits or bits of the number.

Intuition

The solution uses a bitwise XOR operation to determine the different bits between the two provided integers (`x` and `y`). The XOR operation, represented by the caret symbol `^`, compares two bits and results in `1` if they are different and `0` if they are the same. For example:

- `0 ^ 0 = 0`
- `1 ^ 0 = 1`
- `0 ^ 1 = 1`
- `1 ^ 1 = 0`

Therefore, by performing `x ^ y`, we get a number whose binary representation has `1`s in all the positions where the binary representation of `x` and `y` differ.

To count the number of `1`s in the binary representation of the result (which represents the Hamming distance), we use the `bit_count()` method which efficiently counts the number of set bits in the integer.

Hence, the approach for this solution is: calculate `x ^ y` to find the binary representation of their differences, and then use `.bit_count()` to add up the bits that are set to `1`.

Solution Approach

The solution to finding the Hamming distance between two integers involves a single line of code in Python, primarily making use of a bitwise operation and a built-in method.

As a first step, we calculate the XOR (`^`) of the two numbers `x` and `y`. In Python, the XOR operation is applied using the `^` symbol between the two numbers:

```
xor_result = x ^ y
```

The `xor_result` carries a bit pattern where each bit is `1` if the corresponding bits of `x` and `y` are different, and `0` if they are the same. This binary representation is the keystone for finding the Hamming distance.

Next, we count the number of `1`s within this bit pattern to determine the Hamming distance. This can be done through iterating over each bit and adding to a count if the bit is `1`, a process that would take O(n) time, where n is the number of bits in the numbers. However, Python provides the `.bit_count()` method to do this in a more performant manner, taking advantage of the underlying hardware's ability to count set bits more efficiently. The method is directly applied to the XOR result:

```
hamming_distance = xor_result.bit_count()
```

This `hamming_distance` is the number of positions where `x` and `y` have different bits, i.e., the Hamming distance. The `.bit_count()` method returns the number of set bits (bits with value `1`) in the integer, which is what our XOR result highlights - differing bits.

In summary, the algorithm makes use of the bitwise XOR to highlight differences and then applies a bit count to quantify these differences. This approach is efficient as it utilizes a single bitwise operation followed by a specialized count operation, both of which are very fast on modern hardware.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach, considering the calculation of the Hamming distance between two integers, say 10 and 15.

Firstly, we must understand the binary representations of these integers.

- The binary representation of 10 is `1010`.
- The binary representation of 15 is `1111`.

Following the outlined solution approach, we start by calculating the XOR of 10 and 15.

```
x = 10  # `1010` in binary
y = 15  # `1111` in binary
xor_result = x ^ y
```

Performing the XOR operation, we get:

- `1010` (binary for 10)
- `1111` (binary for 15)

- `0101` (result of XOR operation)

Now, `xor_result` will hold the value `0101` in binary, which is `5` in decimal.

Next, we use the `.bit_count()` method on the XOR result to count the number of `1`s in the binary representation (the differing bits), thus finding the Hamming distance.

```
hamming_distance = xor_result.bit_count()
```

The `bit_count()` method will count the number of `1`s in the binary number `0101` which is `2`.

Therefore, the Hamming distance between the integers 10 (`1010`) and 15 (`1111`) is `2`.

In summary, through a simple XOR operation followed by using `.bit_count()` on the result, we efficiently arrived at the Hamming distance between two integers, showcasing the power of bit manipulation and built-in methods.

Python Solution

```python
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        # XOR the two numbers to find the bits that are different
        xor = x ^ y

        # Convert the result to a binary string and count the number of '1's
        # The count of '1's is equal to the Hamming distance
        return bin(xor).count('1')

```

Java Solution

```java
class Solution {
    // Method to calculate the Hamming Distance between two integers.
    public int hammingDistance(int x, int y) {
        // Use XOR to find differing bits between x and y
        int xorResult = x ^ y;

        // Integer.bitCount method counts the number of one-bits in the XOR result.
        // This count represents the number of differing bits, which is the Hamming Distance.
        int count = Integer.bitCount(xorResult);

        // Return the count of differing bits as the Hamming Distance
        return count;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the Hamming distance between two integers.
    int hammingDistance(int x, int y) {
        // Xor operation between x and y will set the bits that are different to 1.
        int xorValue = x ^ y;

        // __builtin_popcount function counts the number of set bits(1s) in the integer.
        // This gives us the number of different bits, which is the Hamming distance.
        int count = __builtin_popcount(xorValue);

        // Return the Hamming distance.
        return count;
    }
};

```

Typescript Solution

```typescript
/**
 * Calculates the Hamming distance between two numbers.
 * The Hamming distance is the number of positions at which
 * the corresponding bits are different.
 *
 * @param {number} x - The first integer to compare.
 * @param {number} y - The second integer to compare.
 * @return {number} The Hamming distance between x and y.
 */
function hammingDistance(x: number, y: number): number {
    // Perform bitwise XOR operation on x and y.
    // This will set the bits in x to 1 wherever x and y differ.
    x ^= y;

    // Initialize answer to store the total count of differing bits.
    let distanceCount = 0;

    // Count the number of bits set to 1.
    while (x) {
        // Remove the rightmost bit set to 1 from x. The expression (x & -x)
        // isolates the rightmost bit set to 1, and subtracting it from x
        // removes that bit.
        x -= x & -x;

        // Increment the count of differing bits.
        distanceCount++;
    }

    // Return the total count of differing bits, which is the Hamming distance.
    return distanceCount;
}

```

Time and Space Complexity

The time complexity of the `hammingDistance` function is primarily dependent on the number of bits in the binary representation of the numbers `x` and `y`. The XOR operation `x ^ y` itself takes `O(1)` time since integer operations are typically performed in constant time on most modern computers. However, counting the number of bits set to 1 — which is essentially what `bit_count()` does — varies depending on the number of bits set to 1 after the XOR operation. In the worst case, where all bits are different, this would be `O(n)`, where `n` is the number of bits required to represent the numbers in the system. Nevertheless, since the size of integers is fixed in Python (usually 32 or 64 bits), the overall time complexity can be considered `O(1)` as well.

The space complexity of the code is `O(1)` because the space required for the computation does not increase with the size of the inputs. The only space needed is for the temporary storage of the XOR result and its bit count, which is a constant number of memory cells regardless of the input size.