477. Total Hamming Distance

Medium | Bit Manipulation | Array | Math

Leetcode Link

Problem Description

The problem is related to the concept of Hamming distance, which is a measure of difference between two integers represented in binary. More specifically, it is the number of bit positions in which the two bits are different. The problem asks us to calculate the sum of Hamming distances between all pairs of integers in an array of integers, `nums`.

To tackle this problem, we are given an array of integers and we want to find the pair-wise Hamming distances and then sum all those distances together. If we have N integers in our array, there are (N*(N-1))/2 pairs, where each pair's Hamming distance contributes to the sum.

Consider an example with smaller numbers and do this task manually. For `[4, 14, 2]`, represented in binary as `[0100, 1110, 0010]`, the Hamming distances between each pair of numbers would be calculated and summed up like this:

- The Hamming distance between `4 (0100)` and `14 (1110)` is 3.
- The Hamming distance between `4 (0100)` and `2 (0010)` is 1.
- The Hamming distance between `14 (1110)` and `2 (0010)` is 2.

The sum of all the Hamming distances is 3 + 1 + 2 = 6.

Intuition

The brute-force approach to calculating the sum of all Hamming distances between pairs in the array `nums` would be to compare every pair and count the bits that are different. However, calculating pairwise Hamming distances for large arrays would be an inefficient solution with a time complexity of O(n^2 * k), where n is the number of integers in `nums` and k is the number of bits.

A clever way to tackle this problem is to count, for each bit position, the number of integers in the array that have a '1' in that position and the number that have a '0'. This is based on the concept that the Hamming distance for a particular bit position across all pairs is simply the number of times '1' occurs at that position multiplied by the number of times '0' occurs at the same position, because a difference occurs only if one is 1 and the other is 0.

For instance, if at the ith bit position, we have `a` numbers with '1' and `b` numbers with '0', the total Hamming distance contributed from the ith bit position would be `a * b`. This is true because each of the `a` numbers with '1' will have a Hamming distance of 1 with each of the `b` numbers with '0'. Therefore, by doing this for all bit positions (0 to 30 for 31-bit integers, since the problem mention doesn't include negative numbers that would need the 32nd bit for the sign) and summing up the results, we get the total Hamming distance for all pairs.

This method is more efficient because it only requires us to pass through all the numbers for each bit position, making it O(n*k), which is significantly faster than the brute-force method, especially for a large number of integers in `nums`.

The given solution uses this efficient method to calculate the sum of all Hamming distances in the array `nums` using a loop that iterates over each bit position, followed by another loop to count occurrences of '1's and '0's, and then combining those counts to calculate the contribution of each bit position to the total Hamming distance.

Solution Approach

The implementation of the solution involves a bit manipulation technique and an understanding of combinatorics.

Here are the steps of the algorithm used in the provided Python code:

1. Initialize `ans` to 0. This variable will hold the sum of Hamming distances.
2. Use a loop to iterate over each bit position. In this case, we loop from 0 to 30, inclusive, because an integer is 32 bits in size and we assume we're dealing with positive numbers only (no need for the sign bit).
3. Inside this loop, initialize two counters `a` and `b` to 0. Counter `a` will keep track of the number of 1's and `b` of the number of 0's for the current bit position across all numbers in `nums`.
4. Inner loop through each number `v` in `nums`:
   - Shift `v` right by `i` bits and perform a bitwise AND with `1` (`(v >> i) & 1`) to isolate the bit at the current bit position.
   - If the result is 1, increment counter `a` (since this number contributes a 1 to the current bit position).
   - If the result is 0, increment counter `b` (since this number contributes a 0 to the current bit position).
5. Outside the inner loop but still inside the first loop, multiply `a` and `b` and add the result to `ans`. This is based on the observation that each pair of numbers contributes 1 to the Hamming distance sum for this bit position if one of them has a 1 and the other has a 0. The total contribution from this bit position is therefore the product of the numbers of 1's and 0's.
6. After completing the loops, `ans` contains the sum of the Hamming distances, and we return this value.

In terms of data structures, the algorithm uses a list to store the input numbers and two integers as counters. There are no complex data structures required. The primary pattern used is bit manipulation, specifically shifting and masking to access individual bits of integers. The algorithm's time complexity is O(n*k), where n is the number of integers in the input array and k is the number of bits we are considering (31 in this case).

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the array `[4, 14, 2]`. Follow along with the steps of the provided algorithm.

Step 1: Initialize `ans` to 0.

Step 2: We will iterate over each bit position from 0 to 30. For simplicity, let's assume we only deal with 4-bit binary representations since the highest number here is 14. So, we loop from 0 to 3.

Step 3 & 4: Let's start with the least significant bit (LSB), i.e., bit position 0.

- Initialize counters `a` and `b` to 0 for this bit position.
- Inner loop over the numbers in `nums`:
  - For `4 (0100)`: `(4 >> 0) & 1` equals `0`, so we increment `b`.
  - For `14 (1110)`: `(14 >> 0) & 1` equals `0`, so we increment `b`.
  - For `2 (0010)`: `(2 >> 0) & 1` equals `0`, so we increment `b`.
- After loop, `a` is `0` and `b` is `3` for this bit position.

Step 5: Multiply `a` and `b` and add the result to `ans`.

- `a * b` is `0` because there are no `1`s in this bit position across all numbers. So, `ans` remains `0`.

Step 6: Continue this process for the next bit positions (1, 2, 3).

- Bit position 1:
  - `a` would be 2 (`4` and `2` have 1 in this bit).
  - `b` would be 1 (`14` has 0 in this bit).
  - `a * b` gives us `2 * 1 = 2`, so `ans` updates to `2`.
- Bit position 2:
  - `a` would be 1 (`14` has 1 in this bit).
  - `b` would be 2 (`4` and `2` have 0 in this bit).
  - `a * b` gives us `1 * 2 = 2`, so `ans` updates to `4`.
- Bit position 3:
  - `a` would be 1 (`14` has 1 in this bit).
  - `b` would be 2 (`4` and `2` have 0 in this bit).
  - `a * b` gives us `1 * 2 = 2`, so `ans` updates to `6`.

Step 7: After completing the loops for all bit positions, `ans` is `6`, which is the sum of the Hamming distances for all pairs in the array `[4, 14, 2]`.

Result: The final sum of Hamming distances computed is `6`. This result corresponds to adding the pairwise Hamming distances: `3` (from `4` and `14`), `1` (from `4` and `2`), and `2` (from `14` and `2`), as was calculated manually at the beginning of the problem description.

Python Solution

```python
class Solution:
    def totalHammingDistance(self, nums: List[int]) -> int:
        total_distance = 0  # Initialize total Hamming distance

        # Loop over each bit position (0 to 30, 31 bits for signed 32-bit integer)
        for bit_position in range(31):
            count_one = count_zero = 0  # Initialize counts for 1s and 0s in the current bit position

            # Iterate over each number in the input list
            for num in nums:
                # Right shift the number by the bit position and get the least significant bit
                bit = (num >> bit_position) & 1

                # Increment the count of ones or zeros based on the least significant bit
                if bit:
                    count_one += 1
                else:
                    count_zero += 1

            # Update the total distance by adding the product of one's and zero's counts
            # Each pair contributes one to the Hamming distance if one bit is 0 and the other is 1
            total_distance += count_one * count_zero

        return total_distance  # Return the computed total Hamming distance

```

Java Solution

```java
class Solution {
    public int totalHammingDistance(int[] nums) {
        int totalDistance = 0; // Initialize a variable to hold the total Hamming distance.

        // Iterate over each bit position (0 to 30 since the problem statement implies 32-bit integers and excludes the sign bit).
        for (int i = 0; i < 31; ++i) {
            int countOnes = 0; // Counter for the number of 1s at the ith bit position across all numbers.
            int countZeros = 0; // Counter for the number of 0s at the ith bit position across all numbers.

            // Check each number in the array for the ith bit.
            for (int num : nums) {
                // Shift the number i bits to the right and check if the least significant bit is 1.
                int bit = (num >> i) & 1;

                // Increment the respective counter based on the bit value (1 or 0).
                if (bit == 1) {
                    countOnes++;
                } else {
                    countZeros++;
                }
            }

            // The Hamming distance for the ith bit is the product of the number of 1s and 0s.
            // Each pair of different bits (one 1 and one 0) at the ith position contributes to one Hamming distance.
            totalDistance += countOnes * countZeros;
        }

        return totalDistance; // Return the sum of the Hamming distances across all bit positions.
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    int totalHammingDistance(std::vector<int>& nums) {
        int totalDistance = 0; // This will accumulate the total Hamming distance.

        // Loop over each bit position (integer in C++ is typically 32 bits, but the problem can be assuming 31-bit integers).
        for (int bitPosition = 0; bitPosition < 31; ++bitPosition) {
            int countOnes = 0;   // Number of elements with the current bit set to 1.
            int countZeros = 0;  // Number of elements with the current bit set to 0.

            // Iterate over all the numbers in the array.
            for (int num : nums) {
                // Isolate the bit at the current position 'bitPosition'.
                int bitValue = (num >> bitPosition) & 1;

                // Increment the count of ones or zeros based on the bit value.
                if (bitValue == 1) {
                    countOnes++;
                } else {
                    countZeros++;
                }
            }

            // The Hamming distance contributed by the current bit position is
            // the product of the count of ones and the count of zeros,
            // since each 1 can form a pair with each 0 resulting in a difference.
            totalDistance += countOnes * countZeros;
        }
        return totalDistance; // Return the computed total Hamming distance.
    }
};

```

Typescript Solution

```typescript
// This function calculates the total Hamming distance between all pairs of numbers in an array.
function totalHammingDistance(nums: number[]): number {
    let totalDistance = 0; // This will accumulate the total Hamming distance.

    // Loop over each bit position (integer in TypeScript is typically 32 bits).
    for (let bitPosition = 0; bitPosition < 31; ++bitPosition) {
        let countOnes = 0;  // Number of elements with the current bit set to 1.
        let countZeros = 0; // Number of elements with the current bit set to 0.

        // Iterate over all the numbers in the array.
        for (const num of nums) {
            // Isolate the bit at the current position 'bitPosition'.
            const bitValue = (num >> bitPosition) & 1;

            // Increment the count of ones or zeros based on the bit value.
            if (bitValue === 1) {
                countOnes++;
            } else {
                countZeros++;
            }
        }

        // The Hamming distance contributed by the current bit position is
        // the product of the count of ones and the count of zeros,
        // since each 1 can form a pair with each 0 resulting in a difference.
        totalDistance += countOnes * countZeros;
    }

    // Return the computed total Hamming distance.
    return totalDistance;
}

```

Time and Space Complexity

The provided Python code calculates the total Hamming distance between all pairs of integers in an input array. Each integer is represented in 32-bit binary form (assuming a maximum of 31 usable bits due to the problem constraints). Here's the analysis of the complexities:

Time Complexity:

To analyze the time complexity, we observe that the code consists of two nested loops:

- The outer loop runs a fixed 31 times, corresponding to the number of bits in a 32-bit integer (minus the sign bit).
- The inner loop iterates through every element in the input list `nums`.

Since the outer loop is constant, we are mostly concerned with the inner loop, which has a runtime proportional to `n`, where `n` is the length of `nums`. Iterating through `nums` happens once for each bit, so the overall time complexity is O(n * 31), which simplifies to `O(n)`.

Thus, the time complexity of the code is `O(n)`.

Space Complexity:

The space complexity is determined by the amount of extra space used by the program, which does not grow with the input. The variables `ans`, `a`, `b`, and `t` use a fixed amount of space regardless of the input size. No additional data structures that scale with the input size are being used.

Hence, the space complexity of the code is `O(1)`.