982. Triples with Bitwise AND Equal To Zero

```
Bit Manipulation
                                Hash Table
                       Array
Hard
```

Given an array of integers nums, we are tasked with finding the count of all unique "AND triples." An "AND triple" is defined as a combination of three indices (i, j, k) within the bounds of the array nums such that the following conditions are met:

```
• 0 <= i < nums.length
• 0 <= j < nums.length
```

**Problem Description** 

• 0 <= k < nums.length

The key condition for an "AND triple" is that the bitwise AND operation (&) applied to nums[i], nums[j], and nums[k] results in

zero, i.e., nums[i] & nums[j] & nums[k] == 0. The bitwise AND operation takes two numbers as operands and performs the AND operation on every pair of corresponding bits.

The operation results in a 1 in each bit position for which the corresponding bits of both operands are 1s. The goal is to write a function that will return the total number of such "AND triples."

Intuition

#### To solve the problem efficiently, we focus on the definition of the bitwise AND operator. When performing an AND operation where the result is 0, any bit in the resulting number must have been 0 in at least one of the operands. This means that as long as

there is a zero at the same position in either i, j, or k index of nums, their AND operation will yield a zero. An intuitive approach might be trying all possible triples (i, j, k) and checking their AND operation one by one, but this would result in a time complexity of O(n^3), which is not efficient for large arrays.

Instead, we can take advantage of the commutative property of the AND operation (i.e., a & b = b & a) and count the number of occurrences for every possible result of nums[i] & nums[j]. We can precompute and store these counts in a Counter dictionary,

which is a type of dictionary provided by Python's collections module that counts the occurrences of each element. After counting all the possible AND results of pairs (i, j), we iterate through each value z in nums and for each pair (xy, v) in our Counter dictionary (where xy is a result of nums[i] & nums[j] and v is the count of how many times this result occurred). If xy

 $\delta z == 0$ , it means this z paired with all v occurrences of xy will contribute v to the total count of "AND triples".

Solution Approach The solution employs an efficient approach that involves a combination of precomputation and hash mapping through the

Counter class, which is part of Python's collections module. The idea is to leverage the properties of the bitwise AND operation

to reduce the computational complexity. Here's a breakdown of the implementation steps using the provided solution code:

### Precomputation of Pairwise AND results: We create a Counter to precompute and store all possible bitwise AND results x &

elements in nums.

cnt = Counter(x & y for x in nums for y in nums) cnt now holds the frequency of each result that can be obtained by performing the bitwise AND operation on any two

y for each pair in the array nums. This precomputation is done in a nested loop where x and y iterate over all elements in nums.

Iterate Through the Counter and the Array: We iterate through both the Counter dictionary cnt and the array nums. For every combination of xy (a precomputed AND result from cnt) and z (an element in nums), we check if the bitwise AND of xy & z equals 0.

In this line of code, xy represents a possible result of nums[i] & nums[j] and v is the number of times this result occurs (the count). We are interested in the cases where xy & z is zero because that means we've found a valid "AND triple".

In the above implementation, the complex part of the problem, which is finding pairs whose AND is 0, is handled by the Counter,

which amortizes that work over all element pairs with a single pass through nums. Then, the final loop provides a multiplication

```
factor (v) for each zero-resulting AND pair when checked against all elements of nums. This clever use of hash mapping paired
with the bitwise operation properties results in an elegant solution that is more computationally efficient than brute force.
```

• Hash mapping: To store and access the count of occurrences of pairwise AND results efficiently.

return sum(v for xy, v in cnt.items() for z in nums if xy & z == 0)

• Bitwise operations: Core part of the logic, determining the property of the "AND triple". Complexity Analysis:

• Time Complexity: O(n^2), since we iterate over all pairs once to compute the AND results and store them in the Counter, and then iterate over

• Space Complexity: O(n^2), primarily due to the space required to store the Counter dictionary, which in the worst case could hold a distinct

• Counter (hash map): A specialized dictionary for counting hashable objects, quite handy for counting occurrences of results (precomputed AND

**Example Walkthrough** 

this Counter and nums in a nested fashion.

count for every pair combination.

Data Structures:

Algorithm Patterns:

operations).

- **Precomputation of Pairwise AND results:**
- nums, the pairs and their AND results are: **2** & 2 = 2 ■ 2 & 1 = 0 (since the binary representation of 2 is 10 and 1 is 01, the AND operation results in 00)

• We will calculate the bitwise AND for every pair of numbers in the array and count the occurrences of each result. For our example array

#### ■ 1 & 4 = 0 (binary 01 AND 100 results in 000) **4 & 4 = 4**

**1** & 1 = 1

With the Counter ready, we now iterate through each unique AND result stored in cnt and each element z in nums, and count if the AND of

Let's consider a small array nums = [2, 1, 4] to illustrate the solution approach:

■ For xy = 2: The AND operation with any of nums does not yield 0, so no count is added. ■ For xy = 0: The AND operation with all elements in nums (2, 1, 4) will yield 0, and since cnt [0] = 3, we have 3 valid "AND triples" for

each element in nums, totaling 9.

def countTriplets(self, nums: List[int]) -> int:

if bitwise\_result & z == 0:

triplet\_count += frequency

the stored result with z equals 0:

Solution Implementation

from collections import Counter

triplet\_count = 0

for z in nums:

return triplet\_count

public int countTriplets(int[] nums) {

maxVal = Math.max(maxVal, num);

int[] count = new int[maxVal + 1];

for (int y : nums) {

class Solution:

2 & 4 = 0 (binary 10 AND 100 results in 000)

Iterate Through the Counter and the Array:

- $\circ$  The count for this iteration will be 0 + 9 + 0 + 0 = 9. Therefore, the total count of all unique "AND triples" for the array nums = [2, 1, 4] is 9.
- **Python**

# Create a counter (dictionary) to store the frequency of each bitwise AND result.

# Iterate through the items in the frequency counter. Each item is a tuple (bitwise result, frequency).

# We calculate bitwise AND for all possible pairs of numbers in the nums list.

■ For xy = 4: The AND operation with any of nums does not yield 0, so no count is added.

• The frequency of AND results will then be stored in a Counter: cnt becomes {2: 1, 0: 3, 1: 1, 4: 1}.

■ For xy = 1: The AND operation with all elements in nums (2, 1, 4) does not yield 0, so no count is added.

for bitwise\_result, frequency in frequency\_counter.items(): # For each bitwise AND result, we check if there's a number z in nums such that # bitwise\_result & z is equal to 0. If yes, we add the frequency of the bitwise result # to the triplet\_count since each occurrence contributes to a valid triplet.

# After iterating through all items and nums, we return the count of triplets.

// Counts the triplets (x, y, z) such that (x & y & z) == 0 from the given nums array.

// Initialize the count array that will hold the frequency of occurrences of x & y.

// Find the maximum value in nums to determine the size of the count array.

// Function to count the number of triplets (x, y, z) from the nums array such that

// Populate the count array with the frequency of all possible (x & y) results.

// Iterate over all possible values of (x & y) and every element z in nums.

// Check if the current combination satisfies x & y & z == 0.

# Create a counter (dictionary) to store the frequency of each bitwise AND result.

# For each bitwise AND result, we check if there's a number z in nums such that

# to the triplet\_count since each occurrence contributes to a valid triplet.

# After iterating through all items and nums, we return the count of triplets.

# bitwise\_result & z is equal to 0. If yes, we add the frequency of the bitwise result

# Iterate through the items in the frequency counter. Each item is a tuple (bitwise result, frequency).

# We calculate bitwise AND for all possible pairs of numbers in the nums list.

frequency\_counter = Counter(x & y for x in nums for y in nums)

# Initialize a result variable to store the count of triplets.

for bitwise\_result, frequency in frequency\_counter.items():

// If so, add the frequency of (x & y) to the answer.

// Find the maximum element to determine the size of the count array.

// Create and initialize the count array to store the frequency

// Initialize the answer to count the number of valid triplets.

// x & y & z == 0, where & is the bitwise AND operator.

int maxElement = \*max\_element(nums.begin(), nums.end());

// of bitwise AND results of all possible pairs (x, y).

int countTriplets(vector<int>& nums) {

int count[maxElement + 1];

for (int y : nums) {

for (int z : nums) {

count[x & y]++;

for (int xy = 0; xy <= maxElement; ++xy) {</pre>

answer += count[xy];

if ((xy & z) == 0) {

for (int x : nums) {

int answer = 0;

memset(count, 0, sizeof(count));

frequency\_counter = Counter(x & y for x in nums for y in nums)

# Initialize a result variable to store the count of triplets.

## Java

int maxVal = 0;

for (int num : nums) {

for (int x : nums) {

class Solution {

```
// Increment the count of x & y.
                count[x & y]++;
        // Initialize the answer to count the number of valid triplets.
        int answer = 0;
        // Iterate over all possible combinations of x \& y and check with each z in nums.
        for (int xy = 0; xy <= maxVal; ++xy) {</pre>
            for (int z : nums) {
                // If x \& y \& z == 0, add the count of x \& y to answer.
                if ((xy \& z) == 0) {
                    answer += count[xy];
        // Return the total count of valid triplets.
        return answer;
C++
#include <vector>
#include <algorithm>
#include <cstring>
class Solution {
public:
```

```
// Return the total count of valid triplets.
          return answer;
  };
  TypeScript
  function countTriplets(nums: number[]): number {
      // Find the maximum number in the 'nums' array.
      const maxNum = Math.max(...nums);
      // Create an array to store the count of all possible 'x & y' results.
      const count: number[] = Array(maxNum + 1).fill(0);
      // Calculate the frequency of each 'x & y' result.
      for (const x of nums) {
          for (const y of nums) {
              count[x & y]++;
      // Initialize a variable to store the total count of triplets.
      let tripletCount = 0;
      // Iterate through all possible 'x & y' results.
      for (let xy = 0; xy <= maxNum; ++xy) {</pre>
          // For each element 'z' in 'nums',
          // if 'xy & z' is zero, increment the total count of triplets by the pre-calculated frequency of 'xy'.
          for (const z of nums) {
              if ((xy \& z) === 0) {
                  tripletCount += count[xy];
      // Return the total count of triplets.
      return tripletCount;
from collections import Counter
class Solution:
```

# Time and Space Complexity

**Time Complexity** 

results in n^2 iterations.

return triplet\_count

for z in nums:

triplet count = 0

def countTriplets(self, nums: List[int]) -> int:

if bitwise\_result & z == 0:

triplet\_count += frequency

### • For each iteration, the bitwise AND operation is performed once, which takes constant time, so the nested loops contribute 0(n^2) to the time complexity.

The time complexity of the provided code can be analyzed by breaking down the nested loops and operations:

• The count (cnt) is updated during these iterations, which, in the worst case, will store n^2 unique values if all possible AND operations result in unique values. Each update to the Counter (a form of dictionary) takes an average of 0(1) time.

• There are two nested loops that iterate over the array nums to compute bitwise AND (&) for every pair of elements. If n is the length of nums, this

- The final loop iterates v times over the elements z in nums for each xy key-value pair in cnt. If v represents the count of occurrences of xy, and there are m such unique xy pairs (with m being at most  $n^2$ ), the final loop could iterate up to m \* n times. The bitwise AND operation and comparison inside the final loop also take constant time.
- Based on the information above, the total time complexity of this code is  $0(n^2) + 0(m * n)$ , which simplifies to  $0(n^2)$  because m is at most  $n^2$ , and therefore m \* n is at most  $n^3$ , which is dominated by the  $n^2$  term when n is large.
- **Space Complexity**

For space complexity, we consider the additional space used apart from the input:

- The Counter object (cnt) holds the result of bitwise AND operations for each pair of elements in nums, which can grow up to n^2 unique keyvalue pairs in the worst case. Hence, space complexity due to cnt is 0(n^2). No other significant additional space is used that grows with the input size.
- Thus, the overall space complexity of the code is  $0(n^2)$ .