2857. Count Pairs of Points With Distance k Bit Manipulation Array Medium **Hash Table**

Problem Description In this problem, we are given a list of points on a 2D plane, where each point is represented as a pair of integers (x, y). These

j) have a specific distance between them, where the distance is defined as (x1 XOR x2) + (y1 XOR y2) and XOR stands for the bitwise XOR operation. A key point to note is that a valid pair (i, j) must have the indices i less than j to ensure pairs are not counted multiple times and to maintain uniqueness. Intuition

pairs are the coordinates of the points. We are also given an integer k. The task is to count how many unique pairs of points (i,

The intuition behind the solution emerges from understanding how XOR operation works and then optimizing the brute force approach of checking each possible pair of points to see if they meet our distance criteria.

1. Understanding XOR: The XOR operation has a crucial property: if a XOR b = c, then a = b XOR c and b = a XOR c. This

property will help us quickly find potential pairs that can have the distance k between them.

Iterating Efficiently: A naive approach would involve checking every possible pair of points, which would be inefficient with

form a valid pair, so we add the count from the counter to our answer.

the number of valid point pairs that are at a distance k from each other.

that stores elements as keys and their counts as values.

Here are the steps to understand and arrive at the solution:

- large datasets. We can optimize this by smartly iterating over possible values of one coordinate, reducing the search space. Counter Dictionary: We use a Counter (dictionary subclass in Python that helps keep track of element counts) to keep track of how many points with certain coordinates we already have seen. The key of the counter will be the coordinate pair, and the
- value is the number of times we have seen that specific pair.
- Calculating Potential Matches: For each new point (x2, y2) we're inspecting, we iterate through possible values of a (from 0 to k) and calculate a corresponding b such that a + b = k. With these values of a and b, we can determine what the

Counting Valid Pairs: For each of these potential matching coordinates (x1, y1), we use our counter to see if we have

previously come across such a point. If we have, we know that the current point (x2, y2) and the counted (x1, y1) would

Updating the Counter: After running through all possible a and corresponding b, we add the current point (x2, y2) to our

coordinate (x1, y1) of a potential matching point would need to be using the XOR property mentioned earlier.

counter, indicating that we have now seen this particular point. **Return the Answer:** Once we have iterated over all points and calculated the valid pairs, we return the total count.

The solution uses a blend of counting methodology with smart iteration based on the properties of the XOR operation to count

• A Counter object cnt is initialized to keep track of how many times a particular point (x, y) is seen. In Python, a Counter is a dictionary

Here is a step-by-step explanation of the algorithm: **Initialization**:

An integer ans is initialized to 0, which will accumulate the number of valid pairs that meet the distance criteria.

• We iterate over each point (x2, y2) in the coordinates array using a for-loop. This loop is responsible for looking at each point and determining how many points we've seen so far can form a valid pair with it at distance k.

Iterate Over Points:

Solution Approach

Determining Potential Pairs Based on Distance k: o Inside the first loop, another for-loop runs through integer values from 0 to k inclusive. With each iteration, we take a as the current value

• We then use the XOR property to find what x1 and y1 would be if x2 XOR x1 = a and y2 XOR y1 = b. This gives us the coordinates of a point that could be at the required distance k from (x2, y2).

Counting and Summing Valid Pairs:

current point to create a distance of k.

of the loop, and we compute b such that b = k - a.

Let's take an example with the following points and k = 5:

Now, let's walk through the solution approach step by step for k = 5:

ans = 0 is initialized to accumulate the number of valid pairs.

points = [(0, 0), (1, 4), (4, 1), (5, 0)]

Updating the Counter: • After inspecting all possible matching points for (x2, y2), we update the cnt dictionary by incrementing the count of the current point (x2, y2). This means that now (x2, y2) can be considered as a potential match for future points.

Using the calculated (x1, y1), we check the cnt dictionary to see if we already came across this point in our previous iterations. If so, the

value from the dictionary for the key (x1, y1) is added to ans. This step counts how many times we have a point that can pair with our

pairs with the specified distance, is returned. The algorithm efficiently uses a single pass through the points while leveraging the constant-time lookup feature of dictionaries in

Python to keep the overall performance manageable. Moreover, the approach takes full advantage of the XOR operation's

properties, avoiding an exhaustive enumeration of all possible point pairs, which could otherwise become computationally

o Once all points have been accounted for, and all valid pairs have been counted, the final value of lans, which represents the number of valid

Example Walkthrough

Iterate Over Points:

so we add 1 to ans.

For (4, 1), repeating the process for values of a:

4) . It's not in cnt, so no match is added to ans .

expensive.

Returning the Result:

Initialization: cnt = Counter() is initialized to keep track of how many times a particular point is seen.

• First we consider the point (0, 0). There are no previous points to compare it with, so we simply move on after adding it to the cnt.

Determining Potential Pairs Based on Distance k: Examining the next point (1, 4). We iterate for a from 0 to 5 (our k value). ■ When a = 0: Then b = 5, we look for a point with coordinates such that 0 XOR x1 = 0 and 4 XOR y1 = 5. This gives us x1 = 0 and y1 = 1. But we have not seen (0, 1) before, so there is no match.

■ When a = 1: Then b = 4, we seek a point where 1 XOR x1 = 1 and 4 XOR y1 = 4. We get x1 = 0 and y1 = 0 which we have seen,

■ When a = 0: Then b = 5, and we look for 4 XOR x1 = 0 & 1 XOR y1 = 5 which yields x1 = 4, y1 = 4. No such point exists yet.

■ When a = 1: Then b = 4, and $4 \times XOR \times 1 = 1 & 1 \times XOR \times 1 = 4$ which yields x1 = 5, y1 = 5. No match.

■ And so on, until a match is found for a = 4: Then b = 1. Here, x1 = 0 & y1 = 0 is a match, so we add 1 to ans.

o cnt[(1,4)] += 1 **Next Points:**

o cnt[(4, 1)] += 1

o cnt[(5, 0)] += 1

0) & (4, 1).

Returning the Result:

Solution Implementation

from collections import Counter

count = Counter()

ans = 0

return ans

import java.util.*;

class Solution {

Java

Initialize the answer to zero

Iterate over all coordinates

for a in range(k + 1):

 $x1, y1 = a ^ x2, b ^ y2$

Return the total count of pairs found

for x2, y2 in coordinates:

b = k - a

Python

class Solution:

Last Point: ∘ For (5, 0), following the same method for different values of a, we find a match for a = 1 & b = 4, which corresponds to the point (4,

def count pairs(self, coordinates: List[List[int]], k: int) -> int:

Check all possible combinations of (a, b) where a + b = k

Create a counter to keep track of the number of occurrences of each coordinate

Calculate the original coordinates (x1, y1) before being XOR'ed

// Method to count the number of pairs whose bitwise XOR meets specified conditions

// Create a hashmap to store the frequency of occurrences of each coordinate pair

// Increment count for this pair if it's already in the hashmap

answer += frequencyCount.getOrDefault(List.of(x1, y1), 0);

// Update the frequencyMap with the current coordinate.

// incrementing its count or adding it if doesn't exist

// Define a helper function to create a unique hash for a pair of coordinates

// Find the counterpart for the current coordinate that will form a valid pair

// Increase the count by the number of times the counterpart has been seen

Create a counter to keep track of the number of occurrences of each coordinate

Calculate the original coordinates (x1, y1) before being XOR'ed

by using the current coordinate (x2, y2) and the calculated values of a and b

Add the number of times the original coordinate (x1, y1) has been seen so far

const hashCoordinates = (x: number, y: number): number => x * 1000000 + y;

// Increment the count for the current coordinate pair in the map

const currentHash = hashCoordinates(coordinateX2, coordinateY2);

countMap.set(currentHash, (countMap.get(currentHash) ?? 0) + 1);

def count pairs(self, coordinates: List[List[int]], k: int) -> int:

Check all possible combinations of (a, b) where a + b = k

// Initialize the count of valid pairs

for (let a = 0; a <= k; ++a) {

// Return the final count of valid pairs

Initialize the answer to zero

Iterate over all coordinates

for a in range(k + 1):

x1, $y1 = a ^ x2$, $b ^ y2$

ans += count[(x1, y1)]

for x2, y2 in coordinates:

b = k - a

const b = k - a;

// Loop through each coordinate in the array

const partnerX = a ^ coordinateX2;

const partnerY = b ^ coordinateY2;

for (const [coordinateX2, coordinateY2] of coordinates) {

// Check all possible pairs with Manhattan distance = k

let pairCount = 0;

return pairCount;

ans = 0

class Solution:

from collections import Counter

count = Counter()

frequencyCount.merge(coordinate, 1, Integer::sum);

public int countPairs(List<List<Integer>> coordinates, int k) {

int answer = 0; // Initialize count of valid pairs to 0

// Extract x2 and y2 from the current coordinate

// Iterating through each coordinate pair in the list

for (List<Integer> coordinate : coordinates) {

int x2 = coordinate.get(0);

int y2 = coordinate.get(1);

int $x1 = a ^ x2$;

int $y1 = b ^ y2;$

// Return the final count of valid pairs

Map<List<Integer>, Integer> frequencyCount = new HashMap<>();

by using the current coordinate (x2, y2) and the calculated values of a and b

■ We continue this process for other values of a until 5 but find no more matches.

The algorithm allowed us to figure out the number of valid point pairs with the desired property by efficiently iterating through the list of points and using the Counter to avoid unnecessary pair checks.

○ Having processed all points, we find that ans = 2. There are two unique pairs of points with XOR-based distance 5: (0, 0) & (1, 4), (0,

Add the number of times the original coordinate (x1, y1) has been seen so far ans += count[(x1, y1)] # Increment the count of the current coordinate (x2, y2)count [(x2, y2)] += 1

// Calculate all possible pairs (x1, y1) within the range 0 to k for (int a = 0; a <= k; ++a) { int b = k - a; // Since a + b should be equal to k// Compute x1 and y1 using XOR operation on a, b with x2, y2 respectively

return answer;

```
#include <vector>
#include <unordered_map>
class Solution {
public:
    // Method to count the number of distinct pairs of points from the input coordinates that have a Manhattan distance of exactly k.
    int countPairs(vector<vector<int>>& coordinates. int k) {
        // Create a hash map to count occurrences of points
        unordered_map<long long, int> pointCount;
        // Helper function to convert a 2D point into a unique long long value
        auto getUniqueKev = [](int x, int v) -> long long {
            return static_cast<long long>(x) * 1000000L + y;
        };
        int pairCount = 0; // Initialize the count of pairs to zero
        // Iterate through each point in coordinates
        for (auto& point : coordinates) {
            int x2 = point[0], y2 = point[1];
            // Check all possible points (x1, y1) that could form a pair with (x2, y2) having Manhattan distance k
            for (int a = 0; a <= k; ++a) {
                int b = k - a:
                // Use the XOR operation to find the corresponding x1 and y1
                int x1 = a ^ x2;
                int y1 = b ^ y2;
                // Increase the count of pairs by the occurrences of the (x1, y1) point
                pairCount += pointCount[getUniqueKey(x1, y1)];
            // Register the occurrence of the current point (x2, y2)
            ++pointCount[getUniqueKey(x2, y2)];
        // Return the total count of pairs found
        return pairCount;
};
TypeScript
function countPairs(coordinates: number[][], k: number): number {
    // Initialize a map to keep track of the count of each coordinate pair
    const countMap: Map<number, number> = new Map();
```

pairCount += countMap.get(hashCoordinates(partnerX, partnerY)) ?? 0; // Nullish coalescing to handle undefined values

Increment the count of the current coordinate (x2, y2)count [(x2, y2)] += 1# Return the total count of pairs found

Time and Space Complexity

return ans

Time Complexity

The given Python code has a nested loop where the outer loop goes through each coordinate in the list coordinates, and for each of these coordinates, the inner loop iterates k + 1 times. If n is the number of coordinates, the total number of iterations of the inner loop across all executions of the outer loop is n * (k + 1). Since the other operations inside the inner loop, including dictionary access and updates, take constant time, the time complexity is 0(n * k). **Space Complexity**

The space complexity of the code is determined by the additional space required for the Counter object cnt. In the worst-case scenario, if all coordinates are unique, the counter will have an entry for each coordinate in the list. Therefore, if there are n coordinates, the space complexity will be O(n) for storing all the unique coordinates in cnt.