2748. Number of Beautiful Pairs

Number Theory

```
Problem Description
```

**Math** 

The LeetCode problem provides an array nums, indexed from 0. We need to find all the "beautiful pairs" in this array. A pair of indices (i, j) is considered beautiful if the first digit of nums[i] and the last digit of nums[j] are coprime. Two numbers are coprime if their greatest common divisor (gcd) is 1, meaning they do not have any prime factors in common besides 1.

To solve the problem, we have to count each such pair where i < j. Intuition

## To solve this problem efficiently, we recognize that there are only 10 possible digits (0 through 9), so it's possible to count

Easy

provided keeps track of the count of leading digits encountered so far in a count array cnt, as we iterate through the array. For each number x in nums, we check if the last digit of x is coprime with each digit we've seen as a first digit, using the gcd function. If they are coprime, we add the count recorded for that first digit to our answer ans, which accumulates the total number of beautiful pairs. After checking against all first digits we've seen, we then record the first digit of x in our cnt count

occurrences of leading and trailing digits without having to compare every possible pair (which would be inefficient). The solution

counts of first digits, leading to a time-efficient solution. Solution Approach

This approach ensures that we are only iterating through the array once and are maintaining a constant-size array to track the

array, incrementing the count for that digit, which will be used for subsequent numbers in nums.

The solution uses a count array cnt of size 10 (since we have 10 digits from 0 to 9) to keep a tally on the number of times a digit

appears as the first digit of a number in nums. Furthermore, it leverages the Greatest Common Divisor (gcd) to check for

# Here's a step-by-step breakdown of the algorithm:

coprimality.

1. Initialize a count array cnt with 10 zeros, corresponding to the digits from 0 to 9. 2. Initialize a variable ans to 0, which will hold the count of the beautiful pairs. For every number x in nums:

Extract the **last digit** of x by calculating x % 10. For every possible **first digit** y ranging from 0 to 9:

- Check if we have previously encountered y as a first digit (cnt[y] > 0). Calculate gcd(x % 10, y). If it is 1, x's last digit and y are coprime.
- If they are coprime, increment ans by the count of numbers (cnt[y]) that had y as their first digit. Convert x to a string and take the first character, convert it back to an integer, and increment the respective count in cnt.
- By using this calculation method, only a single pass through the nums array is needed, and we avoid comparing every pair of numbers. Complexity is reduced from potentially O(n^2) to O(n) because we're only performing a constant amount of work for

each of the n elements in nums.

2. We initialize our variable ans to 0.

Now for each number in nums:

- - **Example Walkthrough**

Let's walk through a small example to illustrate the solution approach. Suppose our input array nums is [12, 35, 46, 57, 23]. 1. We initialize our count array cnt with 10 zeros: cnt = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0].

For num 35:

For num 12:

```
    The last digit is 2 (12 % 10).

• No other numbers have been processed yet, so we add 1 to cnt [1] because 1 is the first digit.

    The cnt array becomes [0, 1, 0, 0, 0, 0, 0, 0, 0].
```

 The last digit is 5 (35 % 10). We check against all first digits we've seen so far, which is only 1. o gcd(5, 1) is 1, they are coprime, so we increment ans by the count of numbers with the first digit 1, which is 1.

- ans becomes 1. We add 1 to cnt[3] for the first digit of 35.
- For num 46: • • The last digit is 6.
- We check against first digits 1 and 3 (from 12 and 35). gcd(6, 1) is 1, so they are coprime, increment ans by cnt[1] which is 1.
- o ans becomes 2. We add 1 to cnt [4] for the first digit of 46.
- For num 57: • ◦ The last digit is 7.

∘ The cnt array becomes [0, 1, 0, 1, 1, 0, 0, 0, 0, 0].

The cnt array becomes [0, 1, 0, 1, 0, 0, 0, 0, 0].

gcd(6, 3) is 3, so 6 and 3 are not coprime, do nothing for cnt[3].

gcd(7, 1), gcd(7, 3), gcd(7, 4) are all 1, so 7 is coprime with 1, 3, and 4.

o Increment ans by cnt[1] + cnt[3] + cnt[4] which is 1 + 1 + 1 = 3.

gcd(3, 3) is 3, so not coprime with itself, do nothing for cnt [3].

The cnt array becomes [0, 1, 1, 1, 1, 1, 0, 0, 0, 0].

def count\_beautiful\_pairs(self, nums: List[int]) -> int:

# Iterate over each number in the input list

# Check each digit from 0 to 9

count[int(str(number)[0])] += 1

# Return the total count of beautiful pairs

for digit in range(10):

# Initialize count array with zeroes for each digit from 0 to 9

# Increment the count of the first digit of 'number'

// A helper method to calculate the greatest common divisor of two numbers.

// A beautiful pair is defined such that the greatest common divisor (gcd) of the

// Count array to keep track of the least significant digits of the numbers

// increment the beautifulPairs by the count of that digit

if (countDigits[digit] && std::gcd(number % 10, digit) == 1) {

// the gcd of the number's least significant digit and current digit is 1,

// least significant digit of one number and any digit of another number is 1

int countDigits[10] = {}; // Initializing all elements to 0

// If the count of digits is not zero and

beautifulPairs += countDigits[digit];

// Reduce the number to its least significant digit

// Increment the count of the least significant digit

int beautifulPairs = 0; // Initialize beautiful pairs count

// If b is zero, then a is the gcd. Otherwise, recursively call gcd with b and a % b.

# Initialize the answer to 0, which will hold the number of beautiful pairs

o ans becomes 5. We add 1 to cnt [5] for the first digit of 57. The cnt array becomes [0, 1, 0, 1, 1, 1, 0, 0, 0, 0].

We check against first digits 1, 3, and 4.

 We check against first digits 1, 3, 4, and 5. o gcd(3, 1) is 1, so they are coprime, increment ans by cnt[1] which is 1.

**Python** 

from math import gcd

class Solution:

For num 23:

• The last digit is 3.

o gcd(3, 4) is 1, so they are coprime, increment ans by cnt[4] which is 1. gcd(3, 5) is 1, so they are coprime, increment ans by cnt[5] which is 1.  $\circ$  ans becomes 5 + 1 + 1 + 1 = 8.

Solution Implementation

count = [0] \* 10

for number in nums:

We add 1 to cnt[2] for the first digit of 23.

from typing import List

After processing all the numbers, we have gone through the array once, and the total count of beautiful pairs ans is 8.

## # If there is a previously encountered number whose last digit has GCD=1 with current last digit of 'number' if count[digit] and gcd(number % 10, digit) == 1: # Increment 'answer' by the count of that digit since it forms a beautiful pair answer += count[digit]

return answer

private int gcd(int a, int b) {

#include <numeric> // For std::gcd

// Function to count beautiful pairs

for (int number : nums) {

while (number > 9) {

number /= 10;

++countDigits[number];

return beautifulPairs;

// Return the total count of beautiful pairs

int countBeautifulPairs(std::vector<int>& nums) {

// Loop through all numbers in the vector

return b == 0 ? a : gcd(b, a % b);

answer = 0

```
Java
class Solution {
    public int countBeautifulPairs(int[] nums) {
       // An array to count the occurrences of the last digits encountered.
       int[] countLastDigits = new int[10];
       // Initialize a variable to keep track of the number of beautiful pairs.
        int beautifulPairs = 0;
       // Iterate over each number in the input array.
        for (int number : nums) {
           // For each digit from 0 to 9, check if we have seen it before as a last digit.
            for (int y = 0; y < 10; ++y) {
               // If we have seen this last digit and the gcd of current number's last digit
               // and y is 1, increment the count of beautiful pairs by the number of times we've seen y.
               if (countLastDigits[y] > 0 && gcd(number % 10, y) == 1) {
                    beautifulPairs += countLastDigits[y];
           // Reduce the current number to its last digit.
           while (number > 9) {
               number /= 10;
           // Increment the count of the last digit of the current number.
            ++countLastDigits[number];
       // Return the total count of beautiful pairs found.
       return beautifulPairs;
```

```
// Check against all digits from 0 to 9
for (int digit = 0; digit < 10; ++digit) {</pre>
```

**}**;

**TypeScript** 

C++

public:

#include <vector>

class Solution {

```
/**
* Calculates the count of beautiful pairs in the array.
* A pair (i, j) is considered beautiful if the GCD of the last digit of nums[i] and nums[j] is 1,
* and i < j.
* @param nums - array of numbers
* @returns the count of beautiful pairs
*/
function countBeautifulPairs(nums: number[]): number {
   // Initialize an array to keep count of the last digit frequency
   const lastDigitCount: number[] = Array(10).fill(0);
    let beautifulPairsCount = 0;
   // Loop through each number in the nums array
   for (let num of nums) {
       // Check against all possible last digits
        for (let digit = 0; digit < 10; ++digit) {</pre>
           // If there's a number with this last digit and their GCD of last digits is 1, count it
```

if (lastDigitCount[digit] > 0 && gcd(num % 10, digit) === 1) {

beautifulPairsCount += lastDigitCount[digit];

// Reduce the number to its last digit

// Increment the count for this last digit

def count\_beautiful\_pairs(self, nums: List[int]) -> int:

# Iterate over each number in the input list

answer += count[digit]

# Check each digit from 0 to 9

count[int(str(number)[0])] += 1

# Return the total count of beautiful pairs

for digit in range(10):

# Initialize count array with zeroes for each digit from 0 to 9

if count[digit] and gcd(number % 10, digit) == 1:

# Increment the count of the first digit of 'number'

# Initialize the answer to 0, which will hold the number of beautiful pairs

num = Math.floor(num / 10);

// Return the total count of beautiful pairs

while (num > 9) {

++lastDigitCount[num];

return beautifulPairsCount;

count = [0] \* 10

for number in nums:

answer = 0

return answer

**Space Complexity** 

Time and Space Complexity

```
/**
   * Recursively calculates the Greatest Common Divisor (GCD) of two numbers using Euclid's algorithm.
   * @param a - first number
   * @param b - second number
   * @returns the GCD of a and b
   */
  function gcd(a: number, b: number): number {
      // Base case: if second number is 0, return the first number
      if (b === 0) {
          return a;
      // Recursive case: return the GCD of b and the remainder of a divided by b
      return gcd(b, a % b);
from typing import List
from math import gcd
class Solution:
```

```
Time Complexity
```

# If there is a previously encountered number whose last digit has GCD=1 with current last digit of 'number'

# Increment 'answer' by the count of that digit since it forms a beautiful pair

The time complexity of the given code is O(n) where n is the length of the input list nums. The analysis is as follows:

- We have a single loop that iterates over each element of nums, which contributes a factor of O(n) to the complexity. • Inside the loop, we execute a fixed number of iterations (10, for the range of y from 0 to 9), checking the greatest common divisor (gcd) of a pair of single digits. Since both the number of iterations and the gcd operation on single-digit numbers are constant-time operations, the loop inside does not depend on n and thus contributes a constant factor, 0(1), per each outer loop iteration.
- Updating the count array cnt also operates in constant time, 0(1), because it accesses a predetermined index determined by the first digit of x. Hence, combining these, the time complexity is O(n) \* O(1) = O(n).
- The space complexity of the code is 0(1) which is a constant space overhead, regardless of the input size. This is explained as follows:
- The cnt array has a fixed length of 10, which does not depend on the size of the input list nums. • The ans variable is just a single integer counter.

As neither of these two grows with the size of the input nums, the space complexity of the algorithm is constant.