2447. Number of Subarrays With GCD Equal to K

Medium <u>Array</u> **Math Number Theory**

Problem Description

divisor (GCD) of all the numbers in each subarray equals a specified integer k. A subarray is defined as a contiguous, non-empty sequence of elements from the array. The GCD of an array is the largest integer that divides every element of the array without leaving a remainder. The task is to explore all possible subarrays within nums and determine which of them have a GCD that matches k. The final

This problem asks us to determine the count of subarrays within a given integer array nums such that the greatest common

output should be the total number of these qualifying subarrays.

Intuition

The intuition behind the solution is based on incrementally verifying every possible subarray. The traditional brute-force approach starts with an index i and iteratively expands to include each subsequent element, recalculating the GCD each time with the

expanding the subarray one element at a time, keeping track of the GCD as it grows. If the GCD matches k at any point, that configuration is counted as a valid subarray. By starting from every possible index i in the array, we ensure that we do not miss any potential subarrays. For each fixed starting index i, we expand the subarray by adding one element at a time to the end, which we denote as x coming from the slice nums[i:] in the code. We keep updating the GCD with the help of the gcd function, which computes the GCD of two

new element included. This brute force method works by calculating the GCD of each subarray starting with a single element and

numbers. Whenever the GCD of the current subarray configuration (g) equates to k, we increment our answer (ans) indicating that we have found a subarray that satisfies the given condition. The process then repeats for all start and end index combinations. Although this method has a higher time complexity as it employs nested loops to check all subarrays, it is straightforward and makes use of the mathematical properties of GCD to identify qualifying subarrays.

Solution Approach The implemented solution approach relies on a nested loop construct to evaluate each subarray's GCD within the input nums array. The outer loop determines the starting position of the subarray, and the inner loop expands the subarray to include

We iterate over the array nums using an outer loop with the index i which defines the start of the subarray. This loop goes

For each starting index i, we initialize a variable g which will hold the GCD of the current subarray. Initially, g is set to 0,

additional elements one by one. Here's a breakdown of how the algorithm operates:

from the first element to the last in nums.

representing an "empty" subarray state.

ans since we've found a qualifying subarray.

We then enter an inner loop starting at i and going to the end of nums. Within this loop, we use the gcd function to calculate the new GCD each time an additional element (x) is included in the subarray. The gcd function is called with the current value of g and the new element x from nums.

- The result of the gcd function is then used to update g, which now holds the GCD of the subarray starting at index i and ending at the current position of the inner loop. If at any point the value of g (the GCD of the current subarray) equals the target value k, we increment a counter variable
- After all elements have been considered for the subarray starting with index i, we repeat the process for the next start index.

This approach does not use any sophisticated data structures; it simply depends on two variables—g for tracking the GCD and

ans for tracking the count of valid subarrays. The computation leans heavily on the mathematical properties of GCD, particularly

that the GCD of any two numbers remains unchanged or becomes smaller when more elements are introduced into the calculation. The complexity of the solution lies in the nested loops, making the time complexity O(n^2) in the worst case, where n is the

length of nums. This occurs because for each starting index, the inner loop could, in the worst case, iterate over the entire array

Despite the relatively simple brute-force nature of the approach, it is effective for solving this specific problem. However, for

larger arrays or constraints requiring better performance, a more optimized solution would be necessary to reduce the time

complexity of the problem. **Example Walkthrough**

Let's walk through a small example to illustrate the solution approach. Suppose we have the following input:

We want to find all subarrays whose GCD is equal to 3. Start with i = 0 and g = 0.

After checking all possible subarrays, we end up with a total count of 3 subarrays for which the GCD equals k. Hence, ans is 3,

Increment i to 2 and reset g to 0. The subarray [6] has a GCD (g) of 6, which is not equal to k (3).

and that is our solution.

Solution Implementation

answer = 0

Python

Java

class Solution {

size to form the subarray.

nums = [2, 4, 6, 3, 9]

k = 3

 Expand the subarray to [6, 3]. The GCD (g) of [6, 3] is 3, which is equal to k. We increment ans to 1. Increment i to 3 and reset g to 0.

Expand the subarray to [3, 9]. The GCD (g) of [3, 9] is 3, which is equal to k. Increment ans to 3.

○ The subarray starting at index 0 is [2]. The GCD (g) of [2] is 2, which is not equal to k (3).

○ The subarray starting at index 1 is [4]. The GCD (g) of [4] is 4, which is not equal to k (3).

• The subarray [3] has a GCD (g) of 3, which is equal to k. Increment ans to 2.

The subarray [9] has a GCD (g) of 9, which is not equal to k (3).

def subarravGCD(self. nums: List[int], k: int) -> int:

Iterate over the list starting from each index.

* @param k The GCD value that subarrays need to match.

// Outer loop to iterate over the start of the subarray

// Update gcdValue with the current num

gcdValue = gcd(gcdValue, nums[j]);

// Inner loop to iterate over the end of the subarray

* @return The count of subarrays with GCD equal to k.

Initialize qcd accumulator for this subarray to 0.

Go through the subsequent numbers to form different subarrays.

Update the current gcd with the new number included.

Return the total count of subarrays where the gcd is equal to k.

int length = nums.length; // Use a clearer variable name for array length

int gcdValue = 0; // This will hold the GCD of the subarray starting at 'i'

// Check if the current GCD equals to k; if so, increment the count

int count = 0; // Use 'count' to represent the number of valid subarrays

Initialize the answer to 0.

for i in range(len(nums)):

for x in nums[i:]:

answer += 1

public int subarravGCD(int[] nums, int k) {

for (int i = 0; i < length; ++i) {

if (gcdValue == k) {

for (int j = i; j < length; ++j) {</pre>

current_gcd = 0

Increment i to 1 and reset g to 0.

Increment i to 4 and reset g to 0.

- this example, the process of expanding subarrays and finding their GCDs demonstrates the straightforward but computationally intensive nature of the algorithm, exemplifying how it operates in practice.
- from math import gcd from typing import List class Solution:
- current_gcd = gcd(current_gcd, x) # If the current acd equals the target k, increment the count. if current acd == k:

/** * Counts the number of subarrays that have a GCD equal to k. * @param nums The array of integers to be checked.

return answer

```
++count;
        // Return the total count of subarrays with GCD equal to k
        return count;
    /**
    * Calculates the greatest common divisor (GCD) of two numbers.
    * @param a The first number.
    * @param b The second number.
    * @return The GCD of a and b.
   private int qcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b); // Recursive implementation of the Euclidean algorithm
#include <vector>
#include <numeric> // For using gcd function
class Solution {
public:
   // Function to compute the number of subarrays whose GCD is exactly k
   int subarrayGCD(vector<int>& nums, int k) {
        int count = 0; // Variable to store the count of valid subarrays
        int size = nums.size(); // Get the number of elements in nums
        // Loop through the starting index of the subarray
        for (int start = 0; start < size; ++start) {</pre>
            int gcd_value = 0; // Initialize GCD value for this subarray
            // Expand the subarray towards the right
            for (int end = start; end < size; ++end) {</pre>
                // Update the GCD of the current subarray
                gcd_value = std::gcd(gcd_value, nums[end]);
                // If the GCD matches k, increment the count
                if (gcd value == k) {
                    ++count;
        // Return the total count of subarrays with GCD equal to k
```

return count;

* Calculates the number of subarrays with a GCD equal to k.

function subarrayGCD(nums: number[], k: number): number {

for (let start = 0; start < numsLength; ++start) {</pre>

* @param {number[]} nums - The array of numbers to be checked.

* @param $\{number\}$ k - The target GCD value for the subarrays.

// Store the length of nums to avoid recalculating it.

for (let end = start; end < numsLength; ++end) {</pre>

gcdValue = gcd(gcdValue, nums[end]);

* @return {number} - The total count of subarrays with GCD equal to k.

// Iterate over the array to start each subarray from each index.

// Initialize gcdValue to be the GCD of the current subarray being considered.

// If the current subarray's GCD is equal to k, increment the counter.

// Extend the subarray one element at a time until the end of the array.

// Update gcdValue to be the GCD of the current subarray.

// Initialize the counter for the number of subarrays found.

};

/**

TypeScript

let count = 0:

const numsLength = nums.length;

if (gcdValue === k) {

++count;

let gcdValue = 0;

```
// Return the total count of qualifying subarrays.
    return count;
/**
 * Computes the greatest common divisor (GCD) of two integers.
 * @param {number} a - The first integer.
 * @param {number} b - The second integer.
 * @return {number} - The GCD of a and b.
function gcd(a: number, b: number): number {
    // If b is 0, a is the GCD. Otherwise, call gcd recursively with b and the remainder of a divided by b.
    return b === 0 ? a : gcd(b, a % b);
from math import gcd
from typing import List
class Solution:
    def subarrayGCD(self. nums: List[int], k: int) -> int:
        # Initialize the answer to 0.
        answer = 0
        # Iterate over the list starting from each index.
        for i in range(len(nums)):
           # Initialize gcd accumulator for this subarray to 0.
            current_gcd = 0
           # Go through the subsequent numbers to form different subarrays.
            for x in nums[i:]:
                # Update the current gcd with the new number included.
                current_gcd = gcd(current_gcd, x)
                # If the current gcd equals the target k, increment the count.
                if current acd == k:
                    answer += 1
        # Return the total count of subarrays where the gcd is equal to k.
        return answer
Time and Space Complexity
Time Complexity
  The given function calculates the number of subarrays with a GCD equal to k. It uses two nested loops. The outer loop runs from
```

current index i to the end of the array. In the worst case, where i starts at 0, this is n iterations for the inner loop. As i increases, the inner loop's range decreases, but in a big-O notation sense, this still leads to an overall time complexity of about (1/2)n(n+1), which is $0(n^2)$ (quadratic time complexity), where n is the length of nums. At each step of the inner loop, the greatest common divisor (GCD) is updated and compared against k. The gcd function itself has a time complexity which is effectively O(log(min(a, b))) where a and b are the numbers for which the GCD is being

computed. However, in the worst case, since these are elements of the array which can vary widely, we often consider this part

the start of the nums array to its end. For each iteration of the outer loop, denoted by index i, the inner loop runs from the

Thus, the time complexity is dominated by the nested loops and is typically expressed as $0(n^2)$.

Space Complexity

constant because it doesn't scale with n substantially.

The space complexity of the function is 0(1) which means constant space. This is because the space used does not grow with the input size n. The only extra variables used are g and ans which are used for calculating the GCD and for keeping the running count of subarrays whose GCD equals k, respectively.