2845. Count of Interesting Subarrays

**Prefix Sum** 

#### Problem Description

Medium Array

to count the number of subarrays considered "interesting." A subarray is defined as a contiguous non-empty sequence of elements within the array. For a subarray nums[l..r] to be interesting, it must satisfy the condition that among its elements, the number of indices i (where  $l \le i \le r$ ) such that nums[i] % modulo == k must itself be congruent to k when taken modulo modulo, i.e., cnt % modulo == k.

The problem provides an integer array called nums, indexed from 0. Additionally, two integers modulo and k are given. The task is

modulo, i.e., cnt % modulo == k.
 To understand better, consider these points:
 You go through the array and find all possible contiguous subarrays.

• If the count of such elements in the subarray also, when divided by modulo, leaves a remainder of k, that subarray is called interesting, and you

Hash Table

- need to increase the interesting subarray count by one.
   The output is the total number of interesting subarrays found this
- The output is the total number of interesting subarrays found this way.

For each subarray, you count the elements that, when divided by modulo, leave a remainder of k.

- Intuition

  To approach this problem efficiently, without checking every possible subarray individually (which would be too time-consuming),
  - we can use a technique from combinatorics that involves keeping track of the cumulative sums of certain conditions.

#### Here's the intuitive step-by-step breakdown:

**Solution Approach** 

cnt = Counter()

a prefix ending at this index.

cnt[(s - k) % modulo].

cnt[s % modulo] += 1

return ans

ans += cnt[(s - k) % modulo]

new subarray ending at this index.

subarrays based on the given criteria.

Let's work through an example to illustrate the solution approach.

to our answer and update the Counter to cnt[1] += 1.

increments to 3. We then increment cnt[2 % 2] = cnt[0] by 1.

modulated prefix sums, accumulating the number of interesting subarrays into ans.

def countInterestingSubarrays(self, nums: List[int], modulo: int, k: int) -> int:

# Initialize with 0 sum having 1 frequency as this represents empty subarray

# The current sum minus the target sum (k) mod modulo will tell us

# if there is a subarray ending at the current index which is interesting

count interesting subarrays += cumulative\_sum\_frequency[(cumulative\_sum - k) % modulo]

// Method that counts the number of subarrays where the number of elements equal to k modulo is also k.

Map<Integer, Integer> remainderCounts = new HashMap<>(); // Map to store the remainder frequencies

long interestingSubarraysCount = 0; // Variable to hold the final count of interesting subarrays

interestingSubarraysCount += remainderCounts.getOrDefault((sum - k + modulo) % modulo, 0);

// Increase the count by the number of occurrences where the adjusted sum matches the expected remainder

// Update the map with the current modulus of the sum and increase the count by 1 or set it to 1 if not present

# in the original nums array satisfies the condition (x % modulo == k).

# This array will contain 1s at the positions where the element

interesting\_elements = [int(num % modulo == k) for num in nums]

# Counter to store the frequency of cumulative sums mod modulo

# Iterate through the boolean array to count interesting subarrays

Transform the original array nums such that each element becomes 1 if it satisfies nums[i] % modulo == k or 0 otherwise. Let's call this array arr.
 Create a <u>prefix sum</u> array s such that s[i] represents the total count of '1's from the start of arr to the current index i. This helps us to quickly calculate the total number of '1's in any subarray.

occurred. The key idea is that if the difference between the prefix sums of two indices is congruent to k modulo modulo, it implies the subarray between those two indices is interesting.

4. As we iterate through the arr, we add to a running sum (s) the value of the current element. We then look up in our Counter how many times

3. Use a hash-based data structure, like Counter in Python, to keep track of how many times each possible prefix sum modulo modulo has

- we've seen prefix sums that are k less than the current sum mod modulo. These contribute to our answer.

  5. We update our Counter with the new running sum mod modulo at each index, incrementing the count since we've now encountered another subarray with that sum.
- Applying these ideas, we achieve a solution that is linear with respect to the size of nums, hence much more efficient than examining all possible subarrays individually.
- The provided solution utilizes an array transformation, prefix sums, modular arithmetic, and a hash map for efficient lookups to tackle the subarray counting challenge.

Array Transformation: First, the code transforms the original nums array into a binary array array array with the same length. Each

element in arr is set to 1 if nums[i] % modulo equals k, otherwise, it is set to 0. This transformation simplifies the problem

by converting it into a problem of counting the number of subarrays whose sum is congruent to k modulo modulo. arr = [int(x % modulo == k) for x in nums]

2. **Using a HashMap (Counter) for <u>Prefix Sum Lookup</u>:** The Counter is used to store the frequency of the occurrence of prefix sums modulo <u>modulo</u>. Initially, Counter is set to {0: 1} because we start with a sum of 0 and there is one way to have a sum of 0 (no elements).

cnt[0] = 1

3. Calculating Prefix Sums and Counting Interesting Subarrays: As we iterate through each element in the transformed array arr, we maintain a running sum s. For each new element x, the running sum s is incremented by x, representing the sum of

We then determine the number of interesting subarrays that end at the current index by looking up how many times we've seen prefix sums that would make the sum of the current subarray equal to k modulo modulo. This is done by checking

After checking for interesting subarrays, we update the Counter with the new running sum modulo modulo to account for the

Returning the Result: The variable ans is used to accumulate the count of interesting subarrays. After iterating over the array

arr, ans holds the final count of all interesting subarrays, which is returned as the result.

O(n) due to the additional array arr and the Counter which might store up to modulo distinct prefix sums.

Example Walkthrough

Consider the integer array nums = [1, 2, 3, 4, 5], with modulo = 2, and k = 1. The task is to count the number of interesting

Array Transformation: We transform nums into arr by setting each arr[i] to 1 if nums[i] % 2 == 1, otherwise 0. Thus,

The overall time complexity of the solution is O(n), as it requires a single pass through the array, and space complexity is also

2. Using a HashMap (Counter) for Prefix Sum Lookup: Initialize a Counter with {0: 1}, representing that the sum of zero occurs once at the beginning (no subarray).
cnt = Counter({0: 1})

Calculating Prefix Sums and Counting Interesting Subarrays: Now, we start iterating through arr and sketch out the

For arr[0], which equals 1, we increment our running sum s = 0 + 1 = 1. We then look in the Counter for cnt[(1 - 1)]

% 2] = cnt[0], which is 1, as we have seen a prefix sum (that sums to 0) exactly once before adding arr[0]. We add 1

For arr[1] with a value of 0, our running sum does not change (s = 1). We look up cnt[(1 - 1) % 2] = cnt[0], again

For arr[2] = 1, s is updated to 2. We check cnt[(2 - 1) % 2] = cnt[1] in the Counter, which is 1, so our answer

By iterating over the entire array arr, we keep a running sum s, check the Counter, and update the counts of encountered

arr becomes [1, 0, 1, 0, 1], since 1, 3, and 5 are odd numbers and give a remainder of 1 when divided by 2.

### finding a 1. We add it to our answer (now ans = 2) and leave the Counter unchanged since arr[1] is 0.

Solution Implementation

from collections import Counter

cumulative sum frequency = Counter()

# Initialize answer and cumulative sum

for element in interesting elements:

# Update cumulative frequency counter

for (int i = 0; i < totalCount; i++) {</pre>

if (remainders[i] == k) {

remainders[i] = 1;

remainders[i] = 0;

// Iterate over the remainders array

for (int remainder: remainders) {

remainders[i] = nums.qet(i) % modulo;

cumulative sum += element

cumulative\_sum\_frequency[0] = 1

count interesting subarrays = 0

cumulative\_sum = 0

returned.

**Python** 

Java

import java.util.List;

import java.util.HashMap;

} else {

import java.util.Map;

class Solution {

class Solution:

process dynamically:

- 4. **Returning the Result**: After iterating over the entire array arr, we will have the final count of all interesting subarrays stored in the variable ans. Assuming we were incrementing ans along with the iterations as described, the final result would be
- This approach simplifies the process, ensuring we only need to traverse the array once, giving an O(n) time complexity, which is efficient for large arrays.
- cumulative\_sum\_frequency[cumulative\_sum % modulo] += 1

  # Return the total count of interesting subarrays
  return count\_interesting\_subarrays

```
public long countInterestingSubarrays(List<Integer> nums, int modulo, int k) {
   int totalCount = nums.size(); // Total number of elements in nums
   int[] remainders = new int[totalCount]; // Array to store the remainders

// Populate the remainders array with 1 if nums[i] % modulo == k or with 0 otherwise
```

remainderCounts.put(0, 1); // Initialize with 0 remainder seen once

sum += remainder: // Increase the sum with the current remainder

int sum = 0; // Variable to accumulate the sum of remainders

remainderCounts.merge(sum % modulo, 1, Integer::sum);

```
return interestingSubarraysCount; // Return the count of interesting subarrays
C++
#include <vector>
#include <unordered_map>
class Solution {
public:
    // Function that counts the number of "interesting" subarrays
    // An "interesting" subarray is one where the sum of its elements, modulo 'modulo', equals 'k'
    long long countInterestingSubarrays(std::vector<int>& nums, int modulo, int k) {
        int n = nums.size(); // Get the size of the 'nums' vector
        std::vector<int> modArray(n); // Array to store modulo transformations
        // Preprocessing: Populate 'modArray' with 1 if nums[i] % modulo == k, otherwise 0
        for (int i = 0; i < n; ++i) {
            modArray[i] = (nums[i] % modulo == k) ? 1 : 0;
        // Hash map to keep track of the count of prefix sums modulo 'modulo'
        std::unordered map<int, int> prefixCount;
        prefixCount[0] = 1; // Initialize for the case where subarray begins at index 0
        long long interestingSubarraysCount = 0; // Variable to store the count of interesting subarrays
        int currentSumModulo = 0; // Variable to store current prefix sum modulo 'modulo'
        // Iterate over the modified array to count "interesting" subarrays
        for (int element : modArray) {
            currentSumModulo += element: // Update current prefix sum
            // Calculate adjusted sum for negative cases and find count in 'prefixCount'
            interestingSubarraysCount += prefixCount[(currentSumModulo - k + modulo) % modulo];
            // Increase the count for this sum modulo 'modulo'
            prefixCount[currentSumModulo % modulo]++;
        // Return the final count of "interesting" subarrays
        return interestingSubarraysCount;
```

// Initialize an array to store binary values, 1 for integers that have a remainder equal to targetRemainder when divided by modu

function countInterestingSubarrays(nums: number[], modulo: number, targetRemainder: number): number {

cumulativeSumCounts.set(0, 1); // Initialize with a zero sum to account for subarrays that start from index 0.

// Accumulate the binary values to keep track of the number of elements equal to targetRemainder modulo 'modulo'

cumulativeSumCounts.set(cumulativeSum % modulo, (cumulativeSumCounts.get(cumulativeSum % modulo) || 0) + 1);

binaryRemainderArray.push(num % modulo === targetRemainder ? 1 : 0);

// Create a map to count the occurrences of cumulative sums modulo 'modulo'.

let cumulativeSum = 0; // Initialize the cumulative sum of binary values.

// Calculate the adjusted cumulative sum for the interesting subarray.

const adjustedSum = (cumulativeSum - targetRemainder + modulo) % modulo;

interestingSubarraysCount += cumulativeSumCounts.get(adjustedSum) || 0;

// Add the number of occurrences where the adjusted cumulative sum has been seen before.

let interestingSubarraysCount = 0: // Initialize the count of interesting subarrays.

const cumulativeSumCounts: Map<number, number> = new Map();

// Increment the count for the current cumulative sum.

for (const binaryValue of binaryRemainderArray) {

cumulativeSum += binaryValue;

## return interestingSubarraysCount; // Return the total count of interesting subarrays found. } from collections import Counter

**}**;

**TypeScript** 

const binaryRemainderArray: number[] = [];

for (const num of nums) {

```
from collections import Counter
class Solution:
   def countInterestingSubarrays(self, nums: List[int], modulo: int, k: int) -> int:
       # This array will contain 1s at the positions where the element
       # in the original nums array satisfies the condition (x \% modulo == k).
        interesting_elements = [int(num % modulo == k) for num in nums]
       # Counter to store the frequency of cumulative sums mod modulo
       cumulative sum frequency = Counter()
       # Initialize with 0 sum having 1 frequency as this represents empty subarray
       cumulative_sum_frequency[0] = 1
       # Initialize answer and cumulative sum
       count interesting subarrays = 0
       cumulative_sum = 0
       # Iterate through the boolean array to count interesting subarrays
       for element in interesting elements:
            cumulative sum += element
           # The current sum minus the target sum (k) mod modulo will tell us
           # if there is a subarray ending at the current index which is interesting
            count interesting subarrays += cumulative_sum_frequency[(cumulative_sum - k) % modulo]
           # Update cumulative frequency counter
           cumulative_sum_frequency[cumulative_sum % modulo] += 1
       # Return the total count of interesting subarrays
       return count_interesting_subarrays
```

# The time complexity of the code is O(n), where n is the length of the input list nums. This is because the code iterates through the nums list once, performing a constant amount of work for each element by computing the modulo, updating the sum s, looking up and updating the count in the cnt dictionary, and incrementing the answer ans.

Time and Space Complexity

looking up and updating the count in the cnt dictionary, and incrementing the answer ans.

The space complexity of the code is also O(n) due to the use of the cnt dictionary, which stores up to n unique sums modulo the value of modulo, and the list arr which stores n elements.