<https://leetcode.com/problems/count-of-interesting-subarrays/description/>

You are given a **0-indexed** integer array nums, an integer modulo, and an integer k.

Your task is to find the count of subarrays that are **interesting**.

A **subarray** nums[l..r] is **interesting** if the following condition holds:

Let cnt be the number of indices i in the range [l, r] such that nums[i] % modulo == k. Then, cnt % modulo == k.

Return an integer denoting the count of interesting subarrays.

**Note**: A subarray is a contiguous non-empty sequence of elements within an array.

**Example 1:**

**Input:** nums = [3,2,4], modulo = 2, k = 1

**Output:** 3

**Explanation:** In this example the interesting subarrays are:

The subarray nums[0..0] which is [3].

- There is only one index, i = 0, in the range [0, 0] that satisfies nums[i] % modulo == k.

- Hence, cnt = 1 and cnt % modulo == k.

The subarray nums[0..1] which is [3,2].

- There is only one index, i = 0, in the range [0, 1] that satisfies nums[i] % modulo == k.

- Hence, cnt = 1 and cnt % modulo == k.

The subarray nums[0..2] which is [3,2,4].

- There is only one index, i = 0, in the range [0, 2] that satisfies nums[i] % modulo == k.

- Hence, cnt = 1 and cnt % modulo == k.

It can be shown that there are no other interesting subarrays. So, the answer is 3.

**Example 2:**

**Input:** nums = [3,1,9,6], modulo = 3, k = 0

**Output:** 2

**Explanation:** In this example the interesting subarrays are:

The subarray nums[0..3] which is [3,1,9,6].

- There are three indices, i = 0, 2, 3, in the range [0, 3] that satisfy nums[i] % modulo == k.

- Hence, cnt = 3 and cnt % modulo == k.

The subarray nums[1..1] which is [1].

- There is no index, i, in the range [1, 1] that satisfies nums[i] % modulo == k.

- Hence, cnt = 0 and cnt % modulo == k.

It can be shown that there are no other interesting subarrays. So, the answer is 2.

**Constraints:**

1 <= nums.length <= 10^5

1 <= nums[i] <= 10^9

1 <= modulo <= 10^9

0 <= k < modulo

**Attempt 1: 2024-01-22**

**Solution 1: Hash Table (60 min)**

**The idea refer to L974.Subarray Sums Divisible by K**

class Solution {

    public long countInterestingSubarrays(List<Integer> nums, int modulo, int k) {

        // We don't care the value of nums[i], we care if nums[i] % mod == k.

        // So if nums[i] % mod == k, we take nums[i] as 1, otherwise 0

        int n = nums.size();

        int[] arr = new int[n];

        for(int i = 0; i < n; i++) {

            if(nums.get(i) % modulo == k) {

                arr[i] = 1;

            }

        }

        long count = 0;

        // Variable to accumulate the sum of remainders

        int sum = 0;

        Map<Integer, Integer> map = new HashMap<>();

        // Initialize a Counter with {0: 1}, representing that the sum of zero

        // occurs once at the beginning (no subarray).

        map.put(0, 1);

        for(int i = 0; i < n; i++) {

            sum += arr[i];

            count += map.getOrDefault((sum - k + modulo) % modulo, 0);

            map.put(sum % modulo, map.getOrDefault(sum % modulo, 0) + 1);

        }

        return count;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/count-of-interesting-subarrays/solutions/3994985/java-c-python-prefix-o-n/>

**Intuition**

We don't care the value of A[i],

we care if A[i] % mod == k.

So if A[i] % mod == k,

we take A[i] as 1,

otherwise 0.

**Explanation**

we calculate the prefix sum acc of A,

then acc mean the number of A[i] % mod == k in i + 1 first elements.

count is a hashmap,

where count[v] means the number of prefix array that have acc % mod == k.

and we initial count[0] = 1 for empty prefix subarray.

Then we iterate a in A,

and we update prefix sum acc,

and update increment res by count[(acc - k) % mod].

Finally return res

**Complexity**

Time O(n)

Space O(mod)

public long countInterestingSubarrays(List<Integer> A, int mod, int k) {

long res = 0;

int acc = 0;

HashMap<Integer, Integer> map = new HashMap<>(Map.of(0, 1));

for (int a : A) {

acc = (acc + (a % mod == k ? 1 : 0)) % mod;

res += map.getOrDefault((acc - k + mod) % mod, 0);

map.put(acc, map.getOrDefault(acc, 0) + 1);

}

return res;

}

**Refer to**

<https://algo.monster/liteproblems/2845>

**Problem Description**

The problem provides an integer array called nums, indexed from 0. Additionally, two integers modulo and k are given. The task is 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 ≤ i ≤ r) such that nums[i] % modulo == k must itself be congruent to k when taken modulo modulo, i.e., cnt % modulo == k.

To understand better, consider these points:

You go through the array and find all possible contiguous subarrays.

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

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 need to increase the interesting subarray count by one.

The output is the total number of interesting subarrays found this way.

**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:

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 prefix sum 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.

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

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

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.

**Solution Approach**

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 arr 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]

Using a HashMap (Counter) for Prefix Sum Lookup: The Counter is used to store the frequency of the occurrence of prefix sums modulo modulo. 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 = Counter()

cnt[0] = 1

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 a prefix ending at this index.

s += x

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 cnt[(s - k) % modulo].

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

After checking for interesting subarrays, we update the Counter with the new running sum modulo modulo to account for the new subarray ending at this index.

cnt[s % modulo] += 1

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.

return ans

The overall time complexity of the solution is O(n), as it requires a single pass through the array, and space complexity is also O(n) due to the additional array arr and the Counter which might store up to modulo distinct prefix sums.

**Example Walkthrough**

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

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 subarrays based on the given criteria.

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

**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 process dynamically:

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

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

By iterating over the entire array arr, we keep a running sum s, check the Counter, and update the counts of encountered modulated prefix sums, accumulating the number of interesting subarrays into ans.

**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 returned.

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.

**Java Solution**

class Solution {

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

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

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

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

if (remainders[i] == k) {

remainders[i] = 1;

} else {

remainders[i] = 0;

}

}

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

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

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

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

// Iterate over the remainders array

for (int remainder : remainders) {

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

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

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

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

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

}

return interestingSubarraysCount; // Return the count of interesting subarrays

}

}

**Time and Space Complexity**

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.

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.