<https://leetcode.com/problems/subarray-sums-divisible-by-k/description/>

Given an integer array nums and an integer k, return **the number of non-empty subarrays that have a sum divisible by**k.

A **subarray** is a **contiguous** part of an array.

**Example 1:**

**Input:** nums = [4,5,0,-2,-3,1], k = 5

**Output:** 7

**Explanation:** There are 7 subarrays with a sum divisible by k = 5:[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]

**Example 2:**

**Input:** nums = [5], k = 9

**Output:** 0

**Constraints:**

1 <= nums.length <= 3 \* 10^4

-10^4 <= nums[i] <= 10^4

2 <= k <= 10^4

**Attempt 1: 2024-01-19**

**Solution 1: Harsh Table (180 min)**

**Wrong Solution**

**Test out by**

Input: nums = [4,5,0,-2,-3,1], k = 5

Expect: 7, output 6

**Test code**

import java.util.\*;

public class Solution {

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

int n = nums.length;

int[] presum = new int[n];

presum[0] = nums[0];

for(int i = 1; i < n; i++) {

presum[i] = presum[i - 1] + nums[i];

}

int count = 0;

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

for(int j = i + 1; j < n; j++) {

if((presum[j] - presum[i]) % k == 0) {

count++;

}

}

}

return count;

}

public static void main(String[] args) {

Solution so = new Solution();

int[] nums = new int[]{4,5,0,-2,-3,1};

int result = so.subarraysDivByK(nums, 5);

System.out.println(result);

}

}

**Step by Step**

e.g nums = [4,5,0,-2,-3,1], k = 5

We suppose to get below 7 subarrays as result

=================================================================

prenum = [4, 9, 9, 7, 4, 5]

subarrays

1.[4, 5, 0, -2, -3, 1] -> NOT include in result

2.[5] -> presum[1] - presum[0] = (nums[0] + nums[1]) - nums[0] = nums[1] = 9 - 4 = 5

3.[5, 0] -> presum[2] - presum[0] = (nums[0] + nums[1] + nums[2]) - nums[0] = nums[1] + nums[2] = 9 - 4 = 5 + 0 = 5

4.[5, 0, -2, -3] -> presum[4] - presum[0] = (nums[0] + nums[1] + nums[2] + nums[3] + nums[4]) - nums[0] = nums[1] + nums[2] + nums[3] + nums[4] = 4 - 4 = 5 + 0 + (-2) + (-3) = 0

5.[0] -> presum[2] - presum[1] = (nums[0] + nums[1] + nums[2]) - (nums[0] + nums[1]) = nums[2] = 9 - 9 = 0

6.[0, -2, -3] -> presum[4] - presum[1] = (nums[0] + nums[1] + nums[2] + nums[3] + nums[4]) - (nums[0] + nums[1]) = nums[2] + nums[3] + nums[4] = 4 - 9 = 0 + (-2) + (-3) = -5

7.[-2, -3] -> presum[4] - presum[2] = (nums[0] + nums[1] + nums[2] + nums[3] + nums[4]) - (nums[0] + nums[1] + nums[2]) = nums[3] + nums[4] = 4 - 9 = (-2) + (-3) = -5

=================================================================

Why subarray = [4, 5, 0, -2, -3, 1] is missing ?

Because that's the whole array, in the wrong solution we don't setup a presum = 0 to handle this case:

(presum of subarray - (presum = 0)) % k == 0

**To fix above issue we have to add extra 0 at the beginning of presum, but this solution will TLE 71/73 since its O(N^2)**

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

int n = nums.length;

int[] presum = new int[n + 1];

for(int i = 1; i <= n; i++) {

presum[i] = presum[i - 1] + nums[i - 1];

}

int count = 0;

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

for(int j = i + 1; j <= n; j++) {

if((presum[j] - presum[i]) % k == 0) {

count++;

}

}

}

return count;

}

**Correct Solution**

**关键点：**

**1.map里面存的是当前presum % k的mod值的频率，理论是presum\_i % k == a和presum\_j % k == a且i < j，那么presum = arr[i, j]可以被k整除**

**2.对于mod取余必须要考虑负数的情况，所以有(num % k + k) % k的说法**

**3.如果起点为index = 0的subarray构成的presum本身就能被k整除，我们必须前置一个presum = 0来完成覆盖，如果采用另外生成一个presum数组并计算好所有presum的话，必须用presum = new int[nums.length + 1]且presum[0] = 0来实现，如果采用map随着for loop逐步构建的话，必须用map.put(0, 1)来实现，这个频率1就是为了map.getOrDefault()的时候能返回频率+1**

**Style 1: Pre-calculate all presum and stored in another array**

class Solution {

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

        int n = nums.length;

        int[] presum = new int[n + 1];

        for(int i = 1; i <= n; i++) {

            presum[i] = presum[i - 1] + nums[i - 1];

        }

        int count = 0;

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

        for(int num : presum) {

            int num\_mod = (num % k + k) % k;

            count += map.getOrDefault(num\_mod, 0);

            map.put(num\_mod, map.getOrDefault(num\_mod, 0) + 1);

        }

        return count;

    }

}

Time Complexity: O(N)

Space Complexity: O(k), with k being the input parameter defining the divisor for subarray sums

**Style 2: No pre-calculate all presum, looply calculate, no extra space needed**

class Solution {

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

        int count = 0;

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

        // We start by initializing our counter with the

        // value {0: 1} to handle the case where a subarray

        // sum itself is directly divisible by k.

        map.put(0, 1);

        int mod\_sum = 0;

        for(int num : nums) {

            mod\_sum = (mod\_sum + num % k + k) % k;

            count += map.getOrDefault(mod\_sum, 0);

            map.put(mod\_sum, map.getOrDefault(mod\_sum, 0) + 1);

        }

        return count;

    }

}

Time Complexity: O(N)

Space Complexity: O(k), with k being the input parameter defining the divisor for subarray sums

**Refer to**

<https://grandyang.com/leetcode/974/>

这道题给了一个数组，让返回数组的数字之和可以被K整除的非空子数组的个数。博主最开始的思路是建立累加和数组，然后就可以快速的知道任意一个子数组的数字和，从而可以判断其是否可以被K整除。但是这个方法被 OJ 残忍拒绝，说是超时 TLE 了，看来需要进一步的将平方级的复杂度优化到线性复杂度才行。说到查询的线性复杂度，那么 HashMap 是当仁不让的，可是这里该如何利用它呢。这里首先要搞懂一个规律，若子数组 [0, i] 的数字之和跟子数组 [0, j] 的数字之和对K取余相同的话，假设这里 j > i，那么子数组 [i+1, j] 的数字之和一定是可以整除K的。这里就不证明了，举个例子吧，比如 [1, 2, 3, 4]，K=5，那么 [1] 之和除以5余1，[1, 2, 3] 之和除以5也余1，则 [2, 3] 之和一定可以整除5。有了这些知识，就可以建立数组和除以K的余数跟其出现次数之间的映射了，注意由于数组中可能出现负数，而我们并不希望出现负余数，所以先对K余数，然后再加个K，再对K取余数，这样一定可以得到正余数。在声明了 HashMap 后，初始化时要把 0 -> 1 先放进去，原因在后面会讲。同时新建变量 sum，用来保存当前的数组和对K的余数，遍历数组A中的数字 num，根据之前说的，num 先对K取余，然后再加上K，之和再加上 sum，再对K取余。此时将 sum 对映射值加到结果 res 中，这里就有两种情况，一种是 sum 并不存在，这样映射值默认是0，另一种是 sum 存在，则根据之前的规律，一定可以找到相同数目的子数组可以整除K，所以将映射值加入结果 res，同时要将 sum 的映射值自增1。这里解释一下为啥刚开始初始化0的映射值是1，因为若 sum 正好是0了，则当前的数组就是符合的题意的，若0的映射值不是1，则这个数组就无法被加入到结果 res 中，参见代码如下：

class Solution {

public:

int subarraysDivByK(vector<int>& A, int K) {

int res = 0, sum = 0;

unordered\_map<int, int> m{{0, 1}};

for (int num : A) {

sum = (sum + num % K + K) % K;

res += m[sum];

++m[sum];

}

return res;

}

};

**Refer to**

<https://leetcode.com/problems/subarray-sums-divisible-by-k/editorial/>

**Overview**

The problem presents an integer array nums and an integer k. Our task is to find the number of non-empty subarrays that have a sum divisible by k.

**Approach: Prefix Sums and Counting**

**Intuition**

The problem is based on the concept of using prefix sums to compute the total number of subarrays that are divisible by k. A prefix sum array for nums is another array prefixSum of the same size as nums, such that the value of prefixSum[i] is the sum of all elements of the nums array from index 0 to index i, i.e., nums[0] + nums[1] + nums[2] + . . . + nums[i].

The sum of the subarray i + 1 to j (inclusive) is computed by prefixSum[j] - prefixSum[i]. Using this, we can count the number of pairs that exist for every pair (i, j) where i < j and (prefixSum[j] - prefix[i]) % k = 0. There are n \* (n - 1) / 2 pairs for an array of length n (pick any two from n). As a result, while this will provide the correct answer for every test case, it will take O(n^2) time, indicating that the time limit has been exceeded (TLE).

The character % is the modulo operator.

Let's try to use the information with respect to the remainders of every prefix sum and try to optimize the above approach.

As stated previously, our task is to determine the number of pairs (i, j) where i < j and (prefixSum[j] - prefix[i]) % k = 0. This equality can only be true if prefixSum[i] % k = prefixSum[j] % k. We will demonstrate this property.

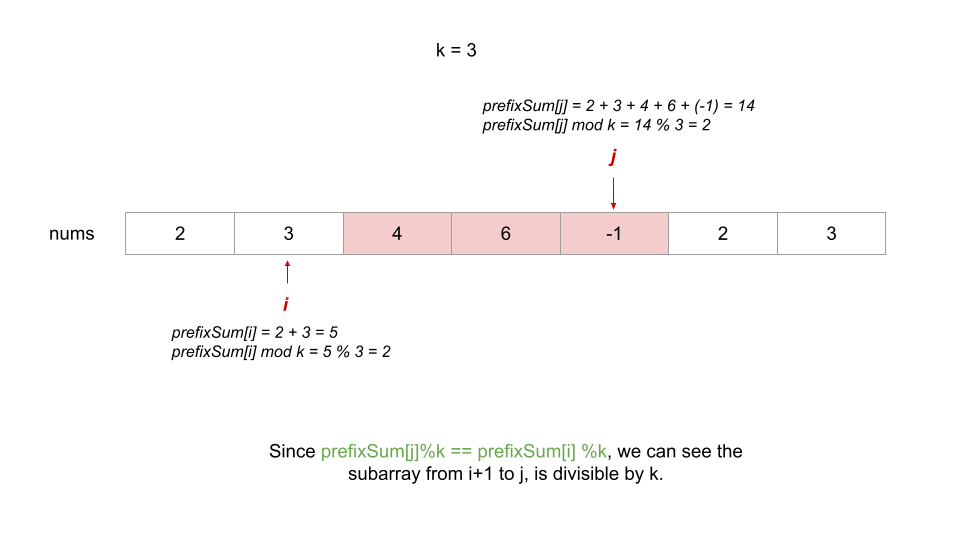
We can express any number as number = divisor × quotient + remainder. For example, 13 when divided by 3 can be written as 13 = 3 \* 4 + 1. So we can express:

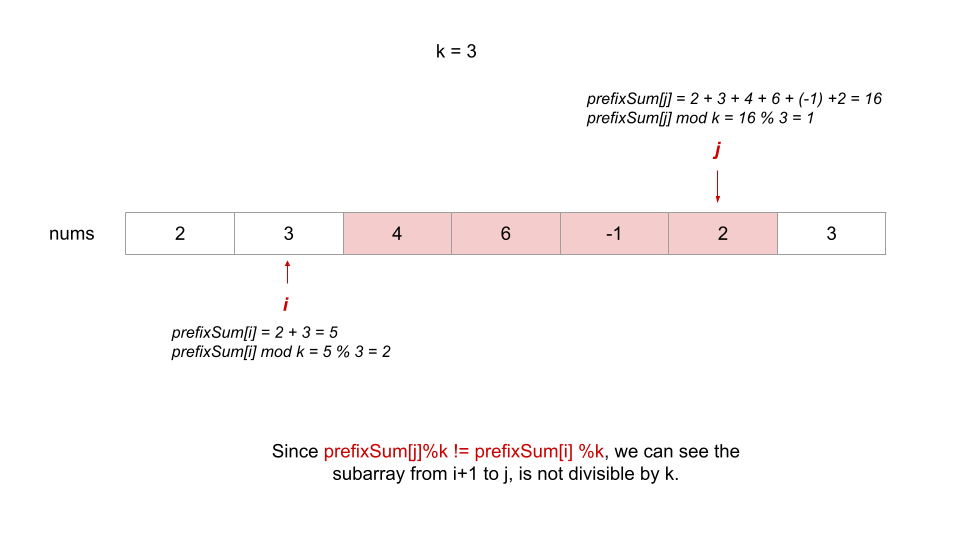
a) prefixSum[i] as prefixSum[i] = A \* k + R0 where A is the quotient and R0 is the remainder when divided by k.

b) Similarly, prefixSum[j] = B \* k + R1 where B is the quotient and R1 is the remainder when divided by k.

We can write, prefixSum[j] - prefixSum[i] = k \* (B - A) + (R1 - R0). The first term (k \* (B - A)) is divisible by k, so for the entire expression to be divisible by k, R1 - R0 must also be divisible by k. This gives us an equation R1 - R0 = C \* k, where C is some integer. Rearranging it yields R1 = C \* k + R0. Because the values of R0 and R1 will be between 0 and k - 1, R1 cannot be greater than k. So the only possible value for C is 0, leading to R0 = R1, which proves the above property. If C > 0, then the RHS would be at least k, but as stated the LHS (R1) is between 0 and k - 1.

Here are two visual examples showing the calculations:





Let's say a subarray ranging from index 0 to index j has a remainder R when the sum of its elements (prefix sum) is divided by k. Our task now becomes to figure out how many subarrays 0..i exist with i < j having the same remainder R when their prefix sum is divided by k. So, we need to maintain the count of remainders while moving in the array.

We start with an integer prefixMod = 0 to store the remainder when the sum of the elements of a subarray that start from index 0 is divided by k. We do not need the prefix sum array, since we only need to maintain the count of each remainder (0 to k - 1) so far. To maintain the count of the remainders, we initialize an array modGroups[k], where modGroups[R] stores the number of times R was the remainder so far.

We iterate over all the elements starting from index 0. We set prefixMod = (prefixMod + num[i] % k + k) % k for each element at index i to find the remainder of the sum of the subarray ranging from index 0 to index i when divided by k. The + k is needed to handle negative numbers. We can then add the number of subarrays previously seen having the same remainder prefixMod to cancel out the remainder. The total number of these arrays is in modGroups[prefixMod]. In the end, we increment the count of modGroups[R] by one to include the current subarray with the remainder R for future matches.

Till now, we chose some previous subarrays (if they exist) to delete the remainder from the existing array formed till index i when the sum of its elements is divided by k. What if the sum of the elements of the array till index i is divisible by k and we don't need another subarray to delete the remainder?

To count the complete subarray from index 0 to index i, we also initialize modGroups[0] = 1 at the start so that if a complete subarray from index 0 to the current index is divisible by k, we include the complete array in the count of modGroups[0]. It is set to start with 1 to cover the complete subarray case. For example, let's assume we are index i. Say, we have previously encountered three subarrays from index 0 to some index j where j < i that were divisible by 'k'. Now, assume the sum of elements in the array up to index i is also divisible by k. So, we will have 4 options to form a subarray ending at index i that is divisible by k. Three of these come from choosing the subarrays (resulting in subarray j + 1, .., i that is divisble by k) that were divisble by k and one comes from choosing the complete subarray starting from index 0 till index i.

**Algorithm**

Initialize an integer prefixMod = 0 to store the remainder when the sum of the elements of a array till the current index when divided by k, and the answer variable result = 0 to store the number of subarrays divisible by k.

Initialize an array, modGroups[k] where modGroup[R] stores the number of subarrays encountered with the sum of elements having a remainder R when divided by k. Set modGroups[0] = 1.

Iterate over all the elements of num.

For each index i, compute the prefix modulo as prefixMod = (prefixMod + num[i] % k + k) % k. We take modulo twice in (prefixMod + num[i] % k + k) % k to remove negative numbers since num[i] can be a negative number and the sum prefixMod + nums[i] % k can turn out to be negative. To remove the negative number we add k to make it positive and then takes its modulo again with k.

Add the number of subarrays encountered till now that have the same remainder to the result: result = result + modGroups[prefixMod].

In the end, we include the remainder of the subarray in the modGroups, i.e., modGroups[prefixMod] = modGroups[prefixMod] + 1 for future matches.

Return result.

**Implementation**

class Solution {

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

int n = nums.length;

int prefixMod = 0, result = 0;

// There are k mod groups 0...k-1.

int[] modGroups = new int[k];

modGroups[0] = 1;

for (int num: nums) {

// Take modulo twice to avoid negative remainders.

prefixMod = (prefixMod + num % k + k) % k;

// Add the count of subarrays that have the same remainder as the current

// one to cancel out the remainders.

result += modGroups[prefixMod];

modGroups[prefixMod]++;

}

return result;

}

}

**Complexity Analysis**

Here, n is the length of nums and kkk is the given integer.

**Time complexity: O(n+k)**

We require O(k) time to initialize the modGroups array.

We also require O(n) time to iterate over all the elements of the nums array. The computation of the prefixSum and the calculation of the subarrays divisible by k take O(1) time for each index of the array.

**Space complexity: O(k)**

We require O(k) space for the modGroups array.

**Refer to**

<https://algo.monster/liteproblems/974>

**Problem Description**

The given problem presents us with a challenge that involves finding particular subsets of an array nums that comply with a certain mathematical property. Specifically, we need to find and count all the contiguous subarrays from nums such that the sum of the elements in each subarray is divisible by an integer k. A subarray, as mentioned, is a contiguous section of the array, which means the elements are consecutive without any gaps.

For example, if we have the array nums = [1, 2, 3, 4] and k = 5, a valid subarray whose sum is divisible by k would be [2, 3], since 2 + 3 = 5, and 5 is divisible by 5. The objective here is to count the total number of such instances in the given array.

**Intuition**

To solve this problem, we utilize a mathematical concept known as the "Pigeonhole Principle" and properties of remainder. The key idea relies on the observation that if the cumulative sum from array elements nums[0] through nums[i] is sum\_i, and sum\_i % k equals sum\_j % k for any j < i, then the subarray nums[j+1] ... nums[i] is divisible by k. This is because the cumulative sum of that subarray yields a remainder of 0 when divided by k.

To implement this concept in our solution, we use a hash table or counter to store the frequency of cumulative sums modulo k that we've encountered so far. **We start by initializing our counter with the value {0: 1} to handle the case where a subarray sum itself is directly divisible by k.**

As we iterate through the array, we update the cumulative sum s, take its modulo with k to compute the current remainder, and check if this remainder has been seen before. If it has, it means there are already subarrays that we’ve processed which, when extended with the current element, would result in a sum divisible by k. Therefore, we increment our answer by the frequency of this remainder in our counter. After checking, we then update the counter to reflect the presence of this new sum modulo k.

The solution uses these steps to continuously update both the counter and the total number of valid subarrays throughout the iteration of nums, ultimately returning the total count of subarrays that satisfy the divisibility condition.

**Solution Approach**

To implement the solution to our problem, we employ the use of a hash table data structure, which in Python can be conveniently represented with a Counter object. The hash table is used to efficiently track and update the count of subarrays whose sum modulo k yields identical remainders.

Let's walk through the implementation step-by-step, referring to the key parts of the provided solution code:

Initialization of the Counter: cnt = Counter({0: 1})

We initialize our counter with a dictionary having a key 0 with a value 1. This represents that we have one subarray sum (an empty prefix) that is divisible by k. This is critical as it allows for the correct computation of subarrays whose cumulative sum is exactly divisible by k from the very beginning of the array.

Initializing the answer and sum variables: ans = s = 0

We set the initial result variable ans and the cumulative sum variable s to 0. ans will hold our final count of valid subarrays, and s will be used to store the running sum as we iterate through the array.

Iterating through nums array:

We loop over each element x in the nums array to calculate the running sum and its modulo with k:

for x in nums:

s = (s + x) % k

At each iteration, we add the current element x to s and take modulo k to keep track of the remainder of the cumulative sum. Taking the modulo ensures that we are only interested in the remainder which helps us in finding the sum of contiguous subarrays divisible by k.

Counting subarrays with the same sum modulo k: ans += cnt[s]

Here, we add to our answer the count of previously seen subarrays that have the same cumulative sum modulo k. This is where the heart of our algorithm lies, following the principle that if two cumulative sums have the same remainder when divided by k, the sum of the elements between these two indices is divisible by k.

Updating the Counter: cnt[s] += 1

After accounting for the current remainder s in our answer, we need to update our counter to reflect that we've encountered this remainder one more time. This means that if we see the same remainder again in the future, there'll be more subarrays with cumulative sums that are divisible by k.

Finally, after iterating through all elements in nums, we return the result stored in ans, which is the count of non-empty subarrays with sums divisible by k.

By utilizing the Counter to efficiently handle the frequencies and the cumulative sum technique, we achieve a time complexity of O(N), where N is the size of the input array, since we only traverse the array once.

**Example Walkthrough**

Let's illustrate the solution approach with a small example with the array nums = [4, 5, 0, -2, -3, 1] and k = 5.

We start by initializing a Counter with {0: 1} because a sum of zero is always divisible by any k. So, cnt = Counter({0: 1}).

We also initialize ans and s to 0.

Now we begin the iteration through nums. We will update s with each element's value and keep track of s % k.

For the first element 4, s = (0 + 4) % 5 = 4. cnt does not have 4 as a key, so we add it: cnt = Counter({0: 1, 4: 1}). ans remains 0 as there are no previous sums with a remainder of 4.

Next, 5 is added to the sum, s = (4 + 5) % 5 = 4. Now cnt[4] exists, so we add its value to ans: ans += cnt[4]. Now ans = 1, because the subarray [5] can be formed whose sum is divisible by 5. We also increment cnt[4] since we have seen another sum with the same remainder: cnt = Counter({0: 1, 4: 2}).

For 0, s = (4 + 0) % 5 = 4. Similarly, ans += cnt[4] (which is 2), so ans is now 3. Then, increment cnt[4]: cnt = Counter({0: 1, 4: 3}).

For -2, s = (4 - 2) % 5 = 2. There are no previous sums with a remainder of 2, so we add it to cnt: cnt = Counter({0: 1, 4: 3, 2: 1}). ans remains 3.

For -3, s = (2 - 3) % 5 = 4, since modulo operation needs to be positive. We add cnt[4] (which is 3) to ans: ans = 6. Then, increment cnt[4]: cnt = Counter({0: 1, 4: 4, 2: 1}).

Finally, for the last element 1, s = (4 + 1) % 5 = 0. We add cnt[0] (which is 1) to ans: ans = 7. cnt[0] is then incremented: cnt = Counter({0: 2, 4: 4, 2: 1}).

After the loop, we're done iterating and our answer ans is 7. We have found 7 non-empty subarrays where the sum is divisible by k which are [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3], and the entire array [4, 5, 0, -2, -3, 1] since its sum 5 is also divisible by 5.

Therefore, by following the solution approach, we were able to efficiently count the subarrays with 7 as the final answer using a single pass through the array and auxiliary space for the Counter.

**Java Solution**

class Solution {

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

        // Create a hashmap to store the frequencies of prefix sum remainders.

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

        // Initialize with remainder 0 having frequency 1.

        countMap.put(0, 1);

        // 'answer' will keep the total count of subarrays divisible by k.

        int answer = 0;

        // 'sum' will store the cumulative sum.

        int sum = 0;

        // Loop through all numbers in the array.

        for (int num : nums) {

            // Update the cumulative sum and adjust it to always be positive and within the range of [0, k-1]

            sum = ((sum + num) % k + k) % k;

            // If this remainder has been seen before, add the number of times it has been seen to the answer.

            answer += countMap.getOrDefault(sum, 0);

            // Increase the frequency of this remainder by 1.

            countMap.merge(sum, 1, Integer::sum);

        }

        // Return the total count of subarrays that are divisible by 'k'.

        return answer;

    }

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the code is O(N), where N is the length of the nums array. This is because the code iterates through the nums array once, performing a constant amount of work for each element by adding the element to the cumulative sum s, computing the modulo k of the sum, and updating the cnt dictionary. The operations of updating the cnt dictionary and reading from it take O(1) time on average, due to hashing.

**Space Complexity**

The space complexity is O(K), with K being the input parameter defining the divisor for subarray sums. The cnt dictionary can have at most K unique keys since each key is the result of the modulo k operation, and there are only K different results possible (from 0 to K-1). Therefore, even in the worst-case scenario, the space used to store counts in the dictionary cannot exceed the number of possible remainders, which is determined by K.

**Refer to chatGPT**

**Approach:**

We can solve this problem using the **prefix sum** and **hash map** technique:

Calculate the prefix sum as we iterate through the array.

Track the remainder of the prefix sum divided by k:

If two prefix sums have the **same remainder**, the subarray between them is divisible by k.

Use a hash map to count how many times a remainder has appeared so far.

**Key Observations:**

For a subarray sum sum(i...j):
sum(i...j) = prefix[j] − prefix[i − 1]
If sum(i...j) is divisible by k, then:
(prefix[j] % k) = (prefix[i − 1] % k)

Use a hash map to store the frequency of each remainder (prefix % k).

Handle negative remainders:

In Java, the % operator can return a negative remainder. To normalize it:
remainder=(remainder+k)%k

class Solution {

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

// Map to store the frequency of remainders

HashMap<Integer, Integer> remainderCount = new HashMap<>();

remainderCount.put(0, 1); // Initialize with remainder 0 appearing once

int prefixSum = 0; // Cumulative prefix sum

int count = 0; // Count of subarrays divisible by k

for (int num : nums) {

// Calculate prefix sum

prefixSum += num;

// Calculate remainder and normalize it to be non-negative

int remainder = ((prefixSum % k) + k) % k;

// If remainder has been seen before, increment count

if (remainderCount.containsKey(remainder)) {

count += remainderCount.get(remainder);

}

// Update the frequency of the current remainder

remainderCount.put(remainder, remainderCount.getOrDefault(remainder, 0) + 1);

}

return count;

}

}

**Explanation:**

Initialization:

The hash map remainderCount starts with {0: 1} because a subarray that itself has a sum divisible by k will have a remainder of 0.

Iterating Through **nums**:

Update the cumulative prefix sum.

Calculate the remainder when divided by k and normalize it to ensure it’s positive.

Check if the remainder has been seen before in the map:

If yes, add its frequency to the count.

Update the hash map with the new remainder.

Return Result:

The final count gives the total number of subarrays whose sum is divisible by k.

**Example:**

**Input:**

nums = [4, 5, 0, -2, -3, 1], k = 5

**Execution:**

Initialize remainderCount = {0: 1}, prefixSum = 0, count = 0.

Iterate through nums:

num = 4: prefixSum = 4, remainder = 4 % 5 = 4, update {4: 1}.

num = 5: prefixSum = 9, remainder = 9 % 5 = 4, count += 1 → {4: 2}.

num = 0: prefixSum = 9, remainder = 9 % 5 = 4, count += 2 → {4: 3}.

num = -2: prefixSum = 7, remainder = (7 % 5 + 5) % 5 = 2, update {2: 1}.

num = -3: prefixSum = 4, remainder = 4 % 5 = 4, count += 3 → {4: 4}.

num = 1: prefixSum = 5, remainder = 5 % 5 = 0, count += 1 → {0: 2}.

Final count = 7.

**Refer to**

[L560.Subarray Sum Equals K](note://5CE29DE019904808986311AFAC74C85C)

[L523.Continuous Subarray Sum (Ref.L974)](note://WEBc2ea24cf3f984fc0adf073f4552f8db8)