<https://leetcode.com/problems/continuous-subarray-sum/description/>

Given an integer array nums and an integer k, return true if nums has a **good subarray** or false otherwise.

A **good subarray** is a subarray where:

its length is **at least two**, and

the sum of the elements of the subarray is a multiple of k.

**Note** that:

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

An integer x is a multiple of k if there exists an integer n such that x = n \* k. 0 is **always** a multiple of k.

**Example 1:**

**Input:** nums = [23,2,4,6,7], k = 6

**Output:** true

**Explanation:** [2, 4] is a continuous subarray of size 2 whose elements sum up to 6.

**Example 2:**

**Input:** nums = [23,2,6,4,7], k = 6

**Output:** true

**Explanation:** [23, 2, 6, 4, 7] is an continuous subarray of size 5 whose elements sum up to 42.42 is a multiple of 6 because 42 = 7 \* 6 and 7 is an integer.

**Example 3:**

**Input:** nums = [23,2,6,4,7], k = 13

**Output:** false

**Constraints:**

1 <= nums.length <= 10^5

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

0 <= sum(nums[i]) <= 2^31 - 1

1 <= k <= 2^31 - 1

**Attempt 1: 2024-01-20**

**Solution 1: Harsh Table (60 min, refer to L974.Subarray Sums Divisible by K)**

**Wrong Solution (87/99)**

**Test out by**

Input: nums = [0], k = 1

Output: true, Expected: false

**The error reason is NOT consider:**

**A good subarray is a subarray where its length is at least two**

class Solution {

    public boolean checkSubarraySum(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];

        }

        // 0 <= sum(nums[i]) <= 2^31 - 1 -> no need consider Long

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

        for(int num : presum) {

            // 0 <= nums[i] <= 10^9, no need consider negative

            // -> no need 'int num\_mod = (num % k + k) % k;'

            int num\_mod = num % k;

            if(map.getOrDefault(num\_mod, 0) > 0) {

                return true;

            }

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

        }

        return false;

    }

}

**Correct Solution**

**Initial version**

class Solution {

    public boolean checkSubarraySum(int[] nums, int k) {

        int n = nums.length;

        if(n <= 1) {

            return false;

        }

        // We don't use extra slot presum[0] = 0 style to

        // create 'presum' array because it relate to index

        // calculation based on original 'nums' array, its

        // more complex to sync on original 'nums' array's

        // index calculation if 'presum' array has extra

        // element ahead

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

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

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

        //}

        int[] presum = new int[n];

        presum[0] = nums[0];

        for(int i = 1; i < n; i++) {

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

        }

        // 0 <= sum(nums[i]) <= 2^31 - 1 -> no need consider Long

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

        // Prepare for subarray start at index = 0 element and its sum % k == 0

        // e.g nums = {23,2,4,6,6}, k = 7, expect true, since sum {23,2,4,6} = 35

        map.put(0, -1);

        for(int i = 0; i < n; i++) {

            // 0 <= nums[i] <= 10^9, no need consider negative

            // -> no need 'int num\_mod = (num % k + k) % k;'

            int num\_mod = presum[i] % k;

            int oldest\_same\_mod\_index = map.getOrDefault(num\_mod, -1);

            if(oldest\_same\_mod\_index != -1) {

                // Not '>= 2', test out by nums = {1,0}, k = 2, expect false

                if(i - oldest\_same\_mod\_index + 1 > 2) {

                    return true;

                }

            } else {

                if(num\_mod == 0 && i - oldest\_same\_mod\_index + 1 > 2) {

                    return true;

                }

            }

            if(oldest\_same\_mod\_index != -1) {

                oldest\_same\_mod\_index = Math.min(oldest\_same\_mod\_index, i);

            } else {

                if(num\_mod != 0) {

                    oldest\_same\_mod\_index = i;

                }

            }

            map.put(num\_mod, oldest\_same\_mod\_index);

        }

        return false;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Promoted Version 1: Merge if else logic into one block**

class Solution {

    public boolean checkSubarraySum(int[] nums, int k) {

        int n = nums.length;

        if(n <= 1) {

            return false;

        }

        // We don't use extra slot presum[0] = 0 style to

        // create 'presum' array because it relate to index

        // calculation based on original 'nums' array, its

        // more complex to sync on original 'nums' array's

        // index calculation if 'presum' array has extra

        // element ahead

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

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

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

        //}

        int[] presum = new int[n];

        presum[0] = nums[0];

        for(int i = 1; i < n; i++) {

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

        }

        // 0 <= sum(nums[i]) <= 2^31 - 1 -> no need consider Long

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

        // Prepare for subarray start at index = 0 element and its sum % k == 0

        // e.g nums = {23,2,4,6,6}, k = 7, expect true, since sum {23,2,4,6} = 35

        map.put(0, -1);

        for(int i = 0; i < n; i++) {

            // 0 <= nums[i] <= 10^9, no need consider negative

            // -> no need 'int num\_mod = (num % k + k) % k;'

            int num\_mod = presum[i] % k;

            int oldest\_same\_mod\_index = map.getOrDefault(num\_mod, -1);

            if(oldest\_same\_mod\_index != -1) {

                if(i - oldest\_same\_mod\_index >= 2) {

                    return true;

                }

                // We always want to keep the first same mode index seen so far,

                // since we want the distance between the oldest same mod index to

                // current index 'i' as large as possible

                oldest\_same\_mod\_index = Math.min(oldest\_same\_mod\_index, i);

            } else {

                // If the oldest same mod index is -1 means two scenarios:

                // 1. If current 'num\_mod' not 0, means that's the first time we

                // encounter a new mod, just update its default index from -1 to i

                // 2. If current 'num\_mod' is 0, means current subarray which

                // start from index = 0 have % k = 0, then we have to check the

                // length of current subarray against 2

                if(num\_mod != 0) {

                    oldest\_same\_mod\_index = i;

                // OR we can write as 'else if(i - 0 + 1 >= 2)' where 'i - 0 + 1'

                // means the length of current subarray which start from index = 0

                } else if(i - oldest\_same\_mod\_index >= 2) {

                    return true;

                }

            }

            map.put(num\_mod, oldest\_same\_mod\_index);

        }

        return false;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Promoted Version 2: No extra presum array needed**

class Solution {

    public boolean checkSubarraySum(int[] nums, int k) {

        int n = nums.length;

        if(n <= 1) {

            return false;

        }

        int presum = 0;

        // 0 <= sum(nums[i]) <= 2^31 - 1 -> no need consider Long

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

        // Prepare for subarray start at index = 0 element and its sum % k == 0

        // e.g nums = {23,2,4,6,6}, k = 7, expect true, since sum {23,2,4,6} = 35

        map.put(0, -1);

        for(int i = 0; i < n; i++) {

            // 0 <= nums[i] <= 10^9, no need consider negative

            // -> no need 'int num\_mod = (num % k + k) % k;'

            presum += nums[i];

            int num\_mod = presum % k;

            int oldest\_same\_mod\_index = map.getOrDefault(num\_mod, -1);

            if(oldest\_same\_mod\_index != -1) {

                if(i - oldest\_same\_mod\_index >= 2) {

                    return true;

                }

                // We always want to keep the first same mode index seen so far,

                // since we want the distance between the oldest same mod index to

                // current index 'i' as large as possible

                oldest\_same\_mod\_index = Math.min(oldest\_same\_mod\_index, i);

            } else {

                // If the oldest same mod index is -1 means two scenarios:

                // 1. If current 'num\_mod' not 0, means that's the first time we

                // encounter a new mod, just update its default index from -1 to i

                // 2. If current 'num\_mod' is 0, means current subarray which

                // start from index = 0 have % k = 0, then we have to check the

                // length of current subarray against 2

                if(num\_mod != 0) {

                    oldest\_same\_mod\_index = i;

                // OR we can write as 'else if(i - 0 + 1 >= 2)' where 'i - 0 + 1'

                // means the length of current subarray which start from index = 0

                } else if(i - oldest\_same\_mod\_index >= 2) {

                    return true;

                }

            }

            map.put(num\_mod, oldest\_same\_mod\_index);

        }

        return false;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Promoted Version 3: No oldest\_same\_mod\_index = Math.min(oldest\_same\_mod\_index, i) needed**

**Instead we have to check whether that's oldest same mod index add into map, we can use 'putIfAbsent' to guarantee that mod index only add into map when it encounter the first time**

class Solution {

    public boolean checkSubarraySum(int[] nums, int k) {

        int n = nums.length;

        if(n <= 1) {

            return false;

        }

        int presum = 0;

        // 0 <= sum(nums[i]) <= 2^31 - 1 -> no need consider Long

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

        // Prepare for subarray start at index = 0 element and its sum % k == 0

        // e.g nums = {23,2,4,6,6}, k = 7, expect true, since sum {23,2,4,6} = 35

        map.put(0, -1);

        for(int i = 0; i < n; i++) {

            // 0 <= nums[i] <= 10^9, no need consider negative

            // -> no need 'int num\_mod = (num % k + k) % k;'

            presum += nums[i];

            int num\_mod = presum % k;

            if(map.containsKey(num\_mod) && i - map.get(num\_mod) >= 2) {

                return true;

            }

            // Instead we have to check whether that's oldest same mod index

            // add into map, we can use 'putIfAbsent' to guarantee that mod

            // index only add into map when it encounter the first time

            map.putIfAbsent(num\_mod, i);

        }

        return false;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://algo.monster/liteproblems/523>

**Problem Description**

The problem provides an integer array nums and an integer k. The task is to determine whether there exists at least one subarray within nums that is both of length two or more and whose sum of elements is a multiple of k. A subarray is defined as a contiguous sequence of elements within the parent array. It's important to note that any integer is considered a multiple of k if it can be expressed as n \* k for some integer n. Zero is also considered a multiple of k by this definition (since 0 = k \* 0).

**Intuition**

To solve this problem, we can utilize the properties of modular arithmetic. The key observation here is that if the sum of a subarray nums[i:j] (where i < j) is a multiple of k, then the cumulative sums sum[0:i-1] and sum[0:j] will have the same remainder when divided by k. This stems from the fact that if (sum[0:j] - sum[0:i-1]) is a multiple of k, then (sum[0:j] % k) = (sum[0:i-1] % k).

The algorithm proceeds as follows:

Iterate through the array, computing the cumulative sum s as we go.

At each step, calculate the remainder of the sum s divided by k (denoted as r = s % k).

Maintain a dictionary (mp) that maps each remainder to the earliest index where that remainder was seen.

For each calculated remainder r, check if we have seen this remainder before. If we have and the distance between the current index and the index stored in the dictionary mp[r] is at least two, this means we've found a good subarray, and we return True.

If the remainder has not been seen before, store the current index in the dictionary against the remainder r.

If no good subarray is found throughout the iteration, return False after the loop completes.

By using this approach, we are effectively tracking the cumulative sums in such a way that we can efficiently check for subarrays whose sum is a multiple of k. The storage of the earliest index where each remainder occurs is crucial for determining the length of the subarray without having to store all possible subarrays.

**Solution Approach**

The solution approach leverages the concept of prefix sums and modular arithmetic to identify a subarray sum that is a multiple of k. Here is the step-by-step explanation of how the solution is implemented:

Initialize a Variable to Store Cumulative Sum (s): We define a variable s that will hold the cumulative sum of the elements as we iterate through the array.

Create a Dictionary (mp) to Store Remainders and Their Earliest Index: A Python dictionary mp is used to map each encountered remainder when dividing the cumulative sum by k to the lowest index where this remainder occurs. The dictionary is initialized with {0: -1} which handles the edge case wherein the cumulative sum itself is a multiple of k from the beginning of the array (i.e., the subarray starts at index 0).

Iterate Through the Array: Using a for-loop, we iterate through the array while keeping track of the current index i and the element value v.

Update Cumulative Sum: With each iteration, we update the cumulative sum s by adding the current element value v to it: s += v.

Calculate Remainder: We calculate the remainder r of the current cumulative sum s when divided by k: r = s % k.

Check for a Previously Encountered Remainder: If the remainder r has been seen before, and the index difference i - mp[r] is greater than or equal to 2, we have found a "good subarray." This is because the equal remainders signify that the sum of elements in between these two indices is a multiple of k. If such a condition is met, the function returns True.

Store the Remainder and Index If Not Already Present: If the remainder r has not been previously encountered, we store this remainder with its corresponding index i into the dictionary: mp[r] = i.

Return False If No Good Subarray Is Found: If the for-loop completes without returning True, it implies that no "good subarray" has been found. In this case, the function returns False.

By using a hashmap to keep track of the remainders, the algorithm ensures a single-pass solution with O(n) time complexity and O(min(n, k)) space complexity, since the number of possible remainders is bounded by k.

**Example Walkthrough**

Let's go through an example to illustrate the solution approach. Suppose we have an array nums = [23, 2, 4, 6, 7] and an integer k = 6. We want to find out if there exists at least one subarray with a sum that is a multiple of k.

**Initialize Cumulative Sum and Dictionary: s = 0. Dictionary mp is initialized as {0: -1}.**

**Iteration 1:**

Index i = 0, Element v = 23.

Update s: s = 0 + 23 = 23.

Calculate remainder r: r = 23 % 6 = 5.

Remainder 5 is not in mp, so we add it: mp = {0: -1, 5: 0}.

**Iteration 2:**

Index i = 1, Element v = 2.

Update s: s = 23 + 2 = 25.

Calculate remainder r: r = 25 % 6 = 1.

Remainder 1 is not in mp, so we add it: mp = {0: -1, 5: 0, 1: 1}.

**Iteration 3:**

Index i = 2, Element v = 4.

Update s: s = 25 + 4 = 29.

Calculate remainder r: r = 29 % 6 = 5.

Remainder 5 is already in mp, and i - mp[5] = 2 - 0 = 2 which is equal to or greater than 2, hence we have found a "good subarray" [23, 2, 4] with sum 29 which is a multiple of k (since 29 - 23 = 6 which is 6\*1).

Return True.

In this example walkthrough, we found a "good subarray" in the third iteration and therefore returned True. This means at least one subarray meets the criteria, thus the function would terminate early with a positive result.

**Java Solution**

class Solution {

    public boolean checkSubarraySum(int[] nums, int k) {

        // HashMap to store the remainder of the sum encountered so far and its index

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

        // To handle the case when subarray starts from index 0

        remainderIndexMap.put(0, -1);

        // Initialize the sum to 0

        int sum = 0;

        // Iterate through the array

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

            // Add current number to the sum

            sum += nums[i];

            // Calculate the remainder of the sum w.r.t k

            int remainder = sum % k;

            // If the remainder is already in the map and the subarray is of size at least 2

            if (remainderIndexMap.containsKey(remainder) && i - remainderIndexMap.get(remainder) >= 2) {

                // We found a subarray with a sum that is a multiple of k

                return true;

            }

            // Put the remainder and index in the map if not already present

            remainderIndexMap.putIfAbsent(remainder, i);

        }

        // If we reach here, no valid subarray was found

        return false;

    }

}

**Time and Space Complexity**

**Time Complexity**

The provided code consists of a single loop that iterates over the list nums once. For each element of nums, it performs constant-time operations involving addition, modulus, and dictionary access (both lookup and insert). Therefore, the time complexity is determined by the loop and is O(n), where n is the number of elements in nums.

**Space Complexity**

The space complexity of the code is primarily dependent on the dictionary mp that is used to store the remainders and their respective indices. In the worst case, each element could result in a unique remainder when taken modulo k. Therefore, the maximum size of mp could be n (where n is the number of elements in nums). Thus, the space complexity is also O(n).