

# 523. Continuous Subarray Sum

Medium

Array

Hash Table

Math

Prefix Sum

Leetcode Link

## 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.

## Python Solution

```
1 class Solution:
2     def checkSubarraySum(self, nums: List[int], k: int) -> bool:
3         # Initialize the prefix sum as zero
4         prefix_sum = 0
5
6         # A dictionary to keep track of the earliest index where
7         # a particular modulus (prefix_sum % k) is found.
8         mod_index_map = {0: -1} # The modulus 0 is at the "imaginary" index -1
9
10        # Iterate over the list of numbers
11        for index, value in enumerate(nums):
12            # Update the prefix sum with the current value
13            prefix_sum += value
14
15            # Get the modulus of the prefix sum with 'k'
16            modulus = prefix_sum % k
17
18            # If the modulus has been seen before and the distance between
19            # the current index and the earlier index of the same modulus
20            # is at least 2, we found a subarray sum that's multiple of k
21            if modulus in mod_index_map and index - mod_index_map[modulus] >= 2:
22                return True
23
24            # Store the index of this modulus if it's not seen before
25            if modulus not in mod_index_map:
26                mod_index_map[modulus] = index
27
28            # No subarray found that sums up to a multiple of k
29            return False
30
```

## Java Solution

```
1 class Solution {
2     public boolean checkSubarraySum(int[] nums, int k) {
3         // HashMap to store the remainder of the sum encountered so far and its index
4         Map<Integer, Integer> remainderIndexMap = new HashMap<>();
5         // To handle the case when subarray starts from index 0
6         remainderIndexMap.put(0, -1);
7         // Initialize the sum to 0
8         int sum = 0;
9
10        // Iterate through the array
11        for (int i = 0; i < nums.length; ++i) {
12            // Add current number to the sum
13            sum += nums[i];
14            // Calculate the remainder of the sum w.r.t k
15            int remainder = sum % k;
16            // If the remainder is already in the map and the subarray is of size at least 2
17            if (remainderIndexMap.containsKey(remainder) && i - remainderIndexMap.get(remainder) >= 2) {
18                // We found a subarray with a sum that is a multiple of k
19                return true;
20            }
21            // Put the remainder and index in the map if not already present
22            remainderIndexMap.putIfAbsent(remainder, i);
23        }
24        // If we reach here, no valid subarray was found
25        return false;
26    }
27 }
28
```

## C++ Solution

```
1 #include <vector>
2 #include <unordered_map>
3 using namespace std;
4
5 class Solution {
6 public:
7     // Function to check if the array has a contiguous subarray of size at least 2
8     // that sums up to a multiple of k
9     bool checkSubarraySum(vector<int>& nums, int k) {
10        // Create a map to store the modulus occurrence with their index
11        unordered_map<int, int> modIndexMap;
12        modIndexMap[0] = -1; // Initialize with a special case to handle edge case
13        int sum = 0; // Accumulated sum
14
15        // Iterate through the numbers in the vector
16        for (int i = 0; i < nums.size(); ++i) {
17            sum += nums[i]; // Add current number to sum
18            int mod = sum % k; // Current modulus of sum by k
19
20            // Check if the modulus has been seen before
21            if (modIndexMap.count(mod)) {
22                // If the distance between two same modulus is at least 2,
23                // it indicates a subarray sum that is a multiple of k
24                if (i - modIndexMap[mod] >= 2) return true;
25            } else {
26                // If this modulus hasn't been seen before, record its index
27                modIndexMap[mod] = i;
28            }
29        }
30
31        // If no qualifying subarray is found, return false
32        return false;
33    }
34 };
35
```

## Typescript Solution

```
1 // Importing necessary utilities
2 import { HashMap } from 'hashmap';
3
4 // Function to check if the array has a contiguous subarray of size at least 2
5 // that sums up to a multiple of k
6 function checkSubarraySum(nums: number[], k: number): boolean {
7     // Create a map to store the modulus occurrence with their index
8     let modIndexMap: Map<number, number> = new Map();
9     modIndexMap.set(0, -1); // Initialize with a special case to handle edge case
10    let accumulatedSum = 0; // Accumulated sum
11
12    // Iterate through the numbers in the array
13    for (let i = 0; i < nums.length; ++i) {
14        accumulatedSum += nums[i]; // Add current number to the accumulated sum
15        let mod = accumulatedSum % k; // Current modulus of the accumulated sum by k
16
17        // Check if the modulus has been seen before
18        if (modIndexMap.has(mod)) {
19            // If the distance between two same modulus is at least 2,
20            // it indicates a subarray sum that is a multiple of k
21            if (i - modIndexMap.get(mod)! >= 2) return true;
22        } else {
23            // If this modulus hasn't been seen before, record its index
24            modIndexMap.set(mod, i);
25        }
26    }
27
28    // If no qualifying subarray is found, return false
29    return false;
30 }
31
```

## 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  $O(n)$ .