Continuous Subarray Sum – Documentation

1. **Problem Statement**

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

   A good subarray is:

   - A contiguous subarray of at least length 2
   - The sum of its elements is a multiple of k (i.e., sum = n * k for some integer n)

2. **Intuition**

   The key idea is to use prefix sums and the modulo operator. If two prefix sums have the same remainder when divided by k, the difference between those sums (i.e., the subarray in between) must be divisible by k.

   We track remainders using a hash map to quickly identify when the same remainder reappears after at least 2 elements.

3. **Key Observations**

   - If (prefix_sum[i] % k) == (prefix_sum[j] % k) and i - j >= 2, then the sum of subarray between j+1 and i is a multiple of k.
   - Store the first index where each remainder was seen to ensure the subarray is of at least length 2.
   - Initialize the map with {0: -1} to handle the case where the subarray starts from index 0.

4. **Approach**

- Initialize mod_map = {0: -1} to store first index of each remainder.
- Maintain a running total sum of elements.
- For each index i in the array:
    - Add the current number to total
    - Compute remainder = total % k
    - If remainder is already in mod_map:
        - Check if i - mod_map[remainder] >= 2 → return True
    - Else, store remainder and current index in mod_map
- If loop completes without returning, return False

5. **Edge Cases**

- Subarray length must be at least 2
- k = 1: any sum is divisible by 1
- Large values of nums[i]: safe since only remainders are stored
- nums may contain zeros: check for subarrays like [0, 0]

6. **Complexity Analysis**

Time Complexity

- O(n): Each element is processed once

Space Complexity

- O(k): At most k remainders stored in the map

7. Alternative Approaches

A. Brute Force (Inefficient for Large Inputs)

- Check all subarrays of size ≥ 2
- Time: $O(n^2)$, Space: $O(1)$
- Not feasible for large arrays (TLE on Leetcode)

B. Sliding Window (Fails for general k)

- Not usable since sums need to be divisible by k and values aren't guaranteed to increase

8. Test Cases

Test Case	Input	Expected Output	Explanation
TC1	nums = [23,2,4,6,7], k = 6	True	[2, 4] → sum = 6
TC2	nums = [23,2,6,4,7], k = 6	True	[23,2,6,4,7] → sum = 42
TC3	nums = [23,2,6,4,7], k = 13	False	No subarray sum divisible by 13
TC4	nums = [0,0], k = 1	True	[0,0] → sum = 0, divisible by 1
TC5	nums = [5,0,0,0], k = 3	True	[0,0] → sum = 0

9. Final Thoughts

- This is a classic case of using prefix sums with modulo math for optimization.
- Understanding the relationship between prefix sums and modulo is essential for problems involving divisible subarrays.
- Efficient use of a hash map helps reduce time complexity from $O(n^2)$ to $O(n)$.