523. Continuous Subarray Sum

Medium Array Hash Table Math Prefix Sum

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:

1. Iterate through the array, computing the cumulative sum `s` as we go.
2. At each step, calculate the remainder of the sum `s` divided by `k` (denoted as `r = s % k`).
3. Maintain a dictionary (`mp`) that maps each remainder to the earliest index where that remainder was seen.
4. 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`.
5. If the remainder has not been seen before, store the current index in the dictionary against the remainder `r`.
6. 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:

1. **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.
2. **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).
3. **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`.
4. **Update Cumulative Sum**: With each iteration, we update the cumulative sum `s` by adding the current element value `v` to it: `s += v`.
5. **Calculate Remainder**: We calculate the remainder `r` of the current cumulative sum `s` when divided by `k`: `r = s % k`.
6. **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`.
7. **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`.
8. **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`.

1. **Initialize Cumulative Sum and Dictionary**: `s = 0`. Dictionary `mp` is initialized as `{0: -1}`.
2. **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}`.
3. **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}`.
4. **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.

Solution Implementation

Python

```python
class Solution:
    def checkSubarraySum(self, nums: List[int], k: int) -> bool:
        # Initialize the prefix sum as zero
        prefix_sum = 0

        # A dictionary to keep track of the earliest index where
        # a particular modulus (prefix sum % k) is found.
        mod_index_map = {0: -1}  # The modulus 0 is at the "imaginary" index -1

        # Iterate over the list of numbers
        for index, value in enumerate(nums):
            # Update the prefix sum with the current value
            prefix_sum += value

            # Get the modulus of the prefix sum with 'k'
            modulus = prefix_sum % k

            # If the modulus has been seen before and the distance between
            # the current index and the earlier index of the same modulus
            # is at least 2, we found a subarray sum that's multiple of k
            if modulus in mod_index_map and index - mod_index_map[modulus] >= 2:
                return True

            # Store the index of this modulus if it's not seen before
            if modulus not in mod_index_map:
                mod_index_map[modulus] = index

        # No subarray found that sums up to a multiple of k
        return False
```

Java

```java
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;
    }
}
```

C++

```cpp
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    // Function to check if the array has a contiguous subarray of size at least 2
    // that sums up to a multiple of k
    bool checkSubarraySum(vector<int>& nums, int k) {
        // Create a map to store the modulus occurrence with their index
        unordered_map<int, int> modIndexMap;
        modIndexMap[0] = -1; // Initialize with a special case to handle edge case
        int sum = 0; // Accumulated sum

        // Iterate through the numbers in the vector
        for (int i = 0; i < nums.size(); ++i) {
            sum += nums[i]; // Add current number to sum
            int mod = sum % k; // Current modulus of sum by k

            // Check if the modulus has been seen before
            if (modIndexMap.count(mod)) {
                // If the distance between two same modulus is at least 2,
                // it indicates a subarray sum that is a multiple of k
                if (i - modIndexMap[mod] >= 2) return true;
            } else {
                // If this modulus hasn't been seen before, record its index
                modIndexMap[mod] = i;
            }
        }

        // If no qualifying subarray is found, return false
        return false;
    }
};
```

TypeScript

```typescript
// Importing necessary utilities
import { HashMap } from 'hashmap';

// Function to check if the array has a contiguous subarray of size at least 2
// that sums up to a multiple of k
function checkSubarraySum(nums: number[], k: number): boolean {
    // Create a map to store the modulus occurrence with their index
    let modIndexMap: Map<number, number> = new Map();
    modIndexMap.set(0, -1); // Initialize with a special case to handle edge case
    let accumulatedSum = 0; // Accumulated sum

    // Iterate through the numbers in the array
    for (let i = 0; i < nums.length; ++i) {
        accumulatedSum += nums[i]; // Add current number to the accumulated sum
        let mod = accumulatedSum % k; // Current modulus of the accumulated sum by k

        // Check if the modulus has been seen before
        if (modIndexMap.has(mod)) {
            // If the distance between two same modulus is at least 2,
            // it indicates a subarray sum that is a multiple of k
            if (i - modIndexMap.get(mod)! >= 2) return true;
        } else {
            // If this modulus hasn't been seen before, record its index
            modIndexMap.set(mod, i);
        }
    }

    // If no qualifying subarray is found, return false
    return false;
}
```


```python
class Solution:
    def checkSubarraySum(self, nums: List[int], k: int) -> bool:
        # Initialize the prefix sum as zero
        prefix_sum = 0

        # A dictionary to keep track of the earliest index where
        # a particular modulus (prefix sum % k) is found.
        mod_index_map = {0: -1}  # The modulus 0 is at the "imaginary" index -1

        # Iterate over the list of numbers
        for index, value in enumerate(nums):
            # Update the prefix sum with the current value
            prefix_sum += value

            # Get the modulus of the prefix sum with 'k'
            modulus = prefix_sum % k

            # If the modulus has been seen before and the distance between
            # the current index and the earlier index of the same modulus
            # is at least 2, we found a subarray sum that's multiple of k
            if modulus in mod_index_map and index - mod_index_map[modulus] >= 2:
                return True

            # Store the index of this modulus if it's not seen before
            if modulus not in mod_index_map:
                mod_index_map[modulus] = index

        # No subarray found that sums up to a multiple of k
        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)`.