974. Subarray Sums Divisible by K

Medium Array Hash Table Prefix Sum Leetcode Link

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:

1. 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.
2. 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.
3. 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`.
4. 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`.
5. 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`.

1. 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})`.

2. We also initialize `ans` and `s` to 0.

3. 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})`.

4. 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.

Python Solution

```python
from collections import Counter  # Import Counter class from collections module.

class Solution:
    def subarraysDivByK(self, A: List[int], K: int) -> int:
        prefix_sum_counter = Counter({0: 1})  # Initialize prefix_sum counter with 0 having a count of 1.
        total_subarrays = 0  # Initialize the count of subarrays divisible by K.
        current_sum = 0  # Initialize current prefix sum as 0.

        # Iterate through each number in the input array.
        for num in A:
            current_sum = (current_sum + num) % K  # Update prefix sum and mod by K.
            total_subarrays += prefix_sum_counter[current_sum]  # Add the count of this sum from the counter to total.
            prefix_sum_counter[current_sum] += 1  # Increment the count of this sum in the counter.

        return total_subarrays  # Return the total number of subarrays.

```

Java Solution

```java
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;
    }
}

```

C++ Solution

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

class Solution {
public:
    int subarraysDivByK(vector<int>& nums, int k) {
        // Create a hash map to store the frequency of each prefix sum mod k.
        unordered_map<int, int> prefixSumCount {{0, 1}};
        int countSubarrays = 0; // Initialize the number of subarrays divisible by k.
        int cumulativeSum = 0; // This will keep track of the cumulative sum of elements.

        // Iterate over each number in the array.
        for (int num : nums) {
            // Add current number to cumulative sum and do mod by k.
            // The double mod ensures that cumulativeSum is positive.
            cumulativeSum = ((cumulativeSum + num) % k + k) % k;

            // If a subarray has cumulativeSum mod k equals to some previous subarray's cumulativeSum mod k,
            // then the subarray in between is divisible by k.
            countSubarrays += prefixSumCount[cumulativeSum]++;

            // The prefixSumCount[cumulativeSum]++ increases the count of the
            // current prefix sum mod k, which will be used for future subarray checks.
        }

        // Return the total count of subarrays with sum divisible by k.
        return countSubarrays;
    }
};

```

Typescript Solution

```typescript
// Function to find the number of subarrays that are divisible by k
function subarraysDivByK(nums: number[], k: number): number {
    // Map to store the frequency of cumulative sums mod k
    const countMap = new Map<number, number>();
    // Base case: a cumulative sum of 0 has 1 occurrence
    countMap.set(0, 1);

    let cumulativeSum = 0; // Initialize the cumulative sum variable
    let answer = 0;         // Initialize the count of subarrays that meet the condition

    // Iterate over the numbers in the array
    for (const num of nums) {
        // Add the current number to cumulative sum
        cumulativeSum += num;
        // Compute the mod of cumulativeSum by k, adjusting for negative results
        const modValue = ((cumulativeSum % k) + k) % k;
        // If the modValue exists in the map, increase the answer by the frequency count
        // This step counts the number of subarrays ending at the current index with modValue
        answer += countMap.get(modValue) || 0;
        // Increment the count of modValue in the map by 1, or set it to 1 if it doesn't exist
        countMap.set(modValue, (countMap.get(modValue) || 0) + 1);
    }
    // Return the total number of subarrays 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`.