<https://leetcode.com/problems/make-sum-divisible-by-p/description/>

Given an array of positive integers nums, remove the **smallest** subarray (possibly **empty**) such that the **sum** of the remaining elements is divisible by p. It is **not** allowed to remove the whole array.

Return **the length of the smallest subarray that you need to remove, or**-1**if it's impossible**.

A **subarray** is defined as a contiguous block of elements in the array.

**Example 1:**

**Input:** nums = [3,1,4,2], p = 6

**Output:** 1

**Explanation:** The sum of the elements in nums is 10, which is not divisible by 6. We can remove the subarray [4], and the sum of the remaining elements is 6, which is divisible by 6.

**Example 2:**

**Input:** nums = [6,3,5,2], p = 9

**Output:** 2

**Explanation:** We cannot remove a single element to get a sum divisible by 9. The best way is to remove the subarray [5,2], leaving us with [6,3] with sum 9.

**Example 3:**

**Input:** nums = [1,2,3], p = 3

**Output:** 0

**Explanation:** Here the sum is 6. which is already divisible by 3. Thus we do not need to remove anything.

**Constraints:**

1 <= nums.length <= 10^5

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

1 <= p <= 10^9

**Attempt 1: 2024-01-21**

**Solution 1: Harsh Table (180 min, refer to L974.Subarray Sums Divisible by K)**

class Solution {

    public int minSubarray(int[] nums, int p) {

        // We cannot directly sum up all nums, because it will exceed

        // integer range as in Constraints we have:

        // 1 <= nums.length <= 10^5 and 1 <= nums[i] <= 10^9

        // Use Modular Arithmetic to calculate the total sum of all

        // elements in the 'nums' array modulo 'p' without encountering

        // integer overflow issues

        // e.g distributive law (a + b) % c = (a % c + b % c) % c

        //int total\_sum = 0;

        //for(int num : nums) {

        //    total\_sum += num;

        //}

        //int total\_sum\_mod\_p = total\_sum % p;

        int total\_sum\_mod\_p = 0;

        for(int num : nums) {

            total\_sum\_mod\_p = (total\_sum\_mod\_p + num) % p;

        }

        // Already divisible by p, thus we do not need to remove anything

        if(total\_sum\_mod\_p == 0) {

            return 0;

        }

        // Find smallest subarray mod p equals total\_sum\_mod

        /\*\*

           (sum(nums) - sum(nums[j:i])) % p == 0

           sum(nums[j:i]) can be represented in terms of prefix sums:

           sum(nums[j:i]) = presum(i) - presum(j)

           -> (sum(nums) - (presum(i) - presum(j))) % p == 0

           -> sum(nums) - (presum(i) - presum(j)) = n \* p

           -> presum(i) - presum(j) = sum(nums) - n \* p

           -> (presum(i) - presum(j)) % p = (sum(nums) - n \* p) % p

           Apply distributive law of mods: (a + b) % c = (a % c + b % c) % c

           on (sum(nums) - n \* p) % p

           -> (presum(i) - presum(j)) % p = (sum(nums) % p - n \* p % p) % p

           -> n \* p % p cancels out

           -> (presum(i) - presum(j)) % p = (sum(nums) % p) % p

           Let's say r = (sum(nums) % p) % p

           -> (presum(i) - presum(j)) % p = r

           -> presum(i) - presum(j) = n \* p + r

           -> presum(j) = presum(i) - n \* p - r

           Mod both sides by p again

           -> presum(j) % p = (presum(i) - n \* p - r) % p

           In equality, the left side 'presum(j) % p' is what we assumed

           index 'j' related presum mod p will store in map when index go

           till 'j', the right side '(presum(i) - n \* p - r) % p' is the

           key that will be calculated in each iteration, now we apply

           distributive law of mods: (a + b) % c = (a % c + b % c) % c on

           right side '(presum(i) - n \* p - r) % p' again:

           -> presum(j) % p = (presum(i) - n \* p - r) % p

           -> n \* p % p cancels out

           -> presum(j) % p = (presum(i) - r) % p

           Recall we have define 'r = (sum(nums) % p) % p' earlier

           -> presum(j) % p = (presum(i) % p - (sum(nums) % p) % p) % p

           -> presum(j) % p = (presum(i) % p - sum(nums) % p) % p

           To avoid negative mod value, we plus p on right side

           -> presum(j) % p = (presum(i) % p - sum(nums) % p + p) % p

           Now '(presum(i) % p - sum(nums) % p + p) % p' is the key that will

           be calculated for each iteration and looking for in map if any

           already stored (previously store in map from each 'presum(j) % p')

         \*/

        int n = nums.length;

        int minLen = n;

        int presum\_mod\_p = 0;

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

        // Pre store {0,-1} in map in case any subarray start with index = 0

        // have the same key = (presum % p - total\_sum\_mod + p) % p

        map.put(0, -1);

        for(int i = 0; i < n; i++) {

            presum\_mod\_p = (presum\_mod\_p + nums[i]) % p;

            int key = (presum\_mod\_p - total\_sum\_mod\_p + p) % p;

            if(map.containsKey(key)) {

                minLen = Math.min(minLen, i - map.get(key));

            }

            map.put(presum\_mod\_p, i);

        }

        return minLen == n ? -1 : minLen;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Step by Step**

e.g

nums = {3,1,4,2}, p = 6

total\_sum\_mod\_p = (3 + 1 + 4 + 2) % 6 = 10 % 6 = 4

Pre store {0,-1} in map in case any subarray start with index = 0 have

the same key = (presum % p - total\_sum\_mod\_p + p) % p

map.put(0,-1) -> {{0,-1}}

smallestLen = n = 4

Round 1:

i = 0

presum\_mod\_p += nums[0] = 3

presum\_mod\_p = 3 % 6 = 3

key = (presum\_mod\_p - total\_sum\_mod\_p + p) % p = (3 - 4 + 6) % 6 = 5

looking for 5 in map -> not exist in map

right most index for key = i = 0

map.put(presum\_mod\_p, i) = map.put(3, 0) = {{0,-1},{3,0}}

Round 2:

i = 1

presum\_mod\_p += nums[1] = 4

presum\_mod\_p = 4 % 6 = 4

key = (presum\_mod\_p - total\_sum\_mod\_p + p) % p = (4 - 4 + 6) % 6 = 0

looking for 0 in map -> found in map (we pre store {0,-1} in map have key = 0)

smallestLen = Math.min(smallestLen, i - map.get(key)) = Math.min(4, 1 - (-1)) = 2

map.put(presum\_mod\_p, i) = map.put(4, 1) = {{0,-1},{3,0},{4,1}}

Round 3:

i = 2

presum\_mod\_p += nums[2] = 8

presum\_mod\_p = 8 % 6 = 2

key = (presum\_mod\_p - total\_sum\_mod\_p + p) % p = (2 - 4 + 6) % 6 = 4

looking for 4 in map -> found in map

smallestLen = Math.min(smallestLen, i - map.get(key)) = Math.min(2, 2 - 1) = 1

map.put(presum\_mod\_p, i) = map.put(2, 2) = {{0,-1},{3,0},{4,1},{2,2}}

Round 4:

i = 3

presum\_mod\_p += nums[3] = 10

presum\_mod\_p = 10 % 6 = 4

key = (presum\_mod\_p - total\_sum\_mod\_p + p) % p = (4 - 4 + 6) % 6 = 0

looking for 0 in map -> found in map

smallestLen = Math.min(smallestLen, i - map.get(key)) = Math.min(1, 3 - (-1)) = 1

map.put(presum\_mod\_p, i) = map.put(4, 3) = {{0,-1},{3,0},{4,3},{2,2}}

Don't worry we update {4,1} to {4,3}, because to find minimum subarray length that

match required condition, we only need to track right most index for key, and since

i will auto increase during for loop, for same key, update value to current i is fine

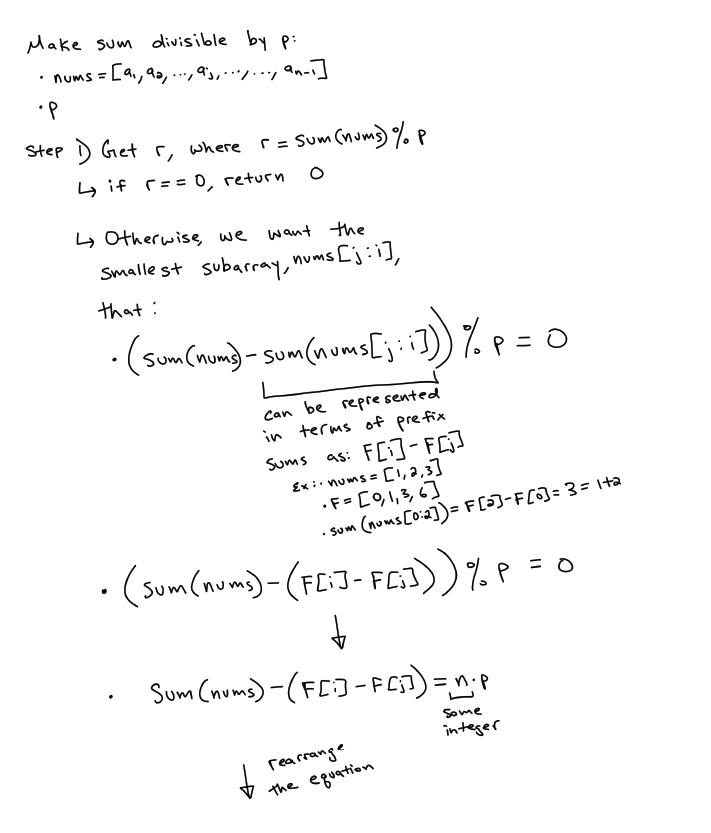
**Refer to**

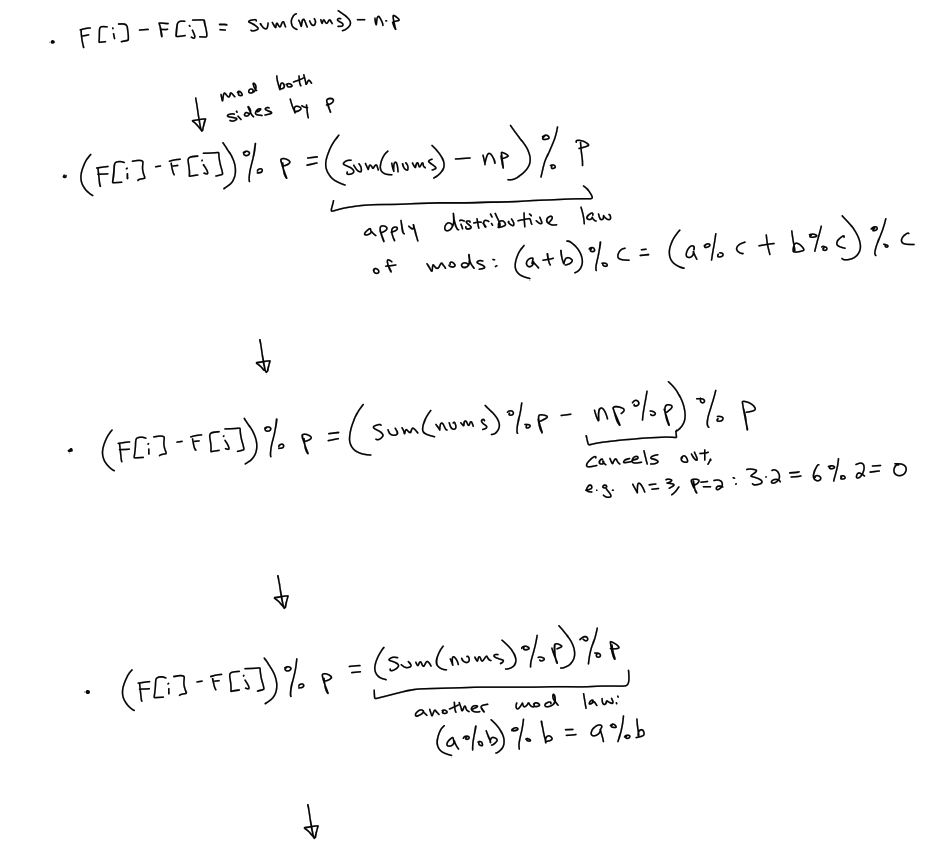
<https://leetcode.com/problems/make-sum-divisible-by-p/solutions/859173/detailed-explanation-of-math-behind-o-n-solution/>

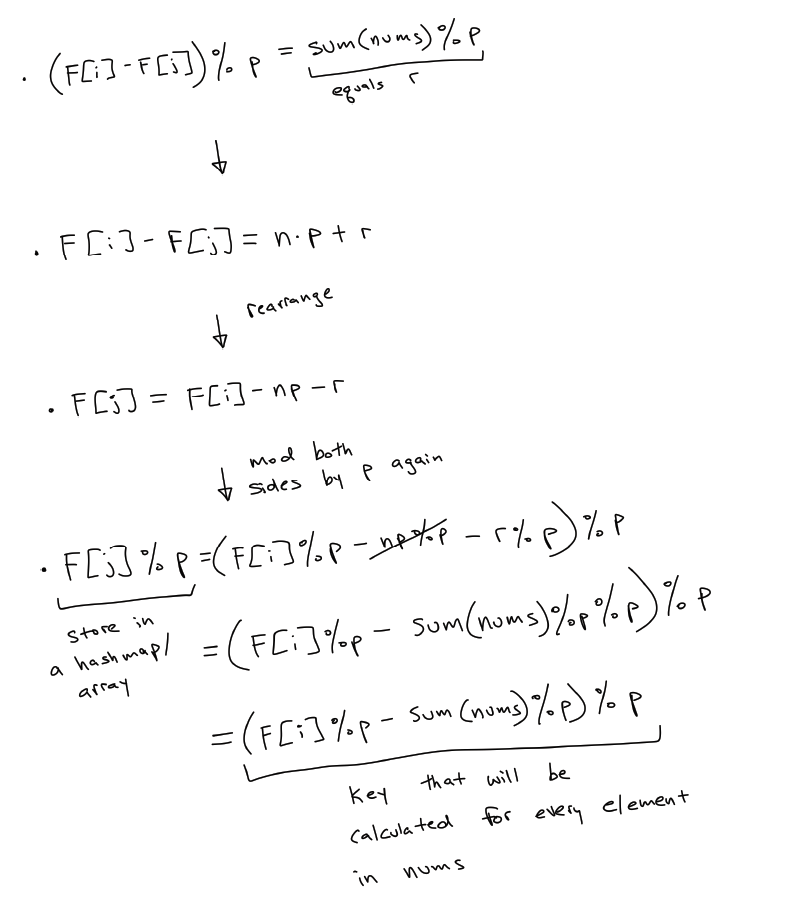
If you don't know what the prefix-sum technique is, please first go through:

**subarray-sum-equals-k** - can be thought of as an intro to prefix sum problems, and I have provided the same explanation for this problem [L560.Subarray Sum Equals K](note://5CE29DE019904808986311AFAC74C85C)

**subarray-sums-divisible-by-k**, which can be thought of as a precursor to this problem, and I have provided a detailed explanation for this problem [L974.Subarray Sums Divisible by K (Ref.L560,L523)](note://WEBde138f801994c4eb8fff2da131b5856f)







(sum(nums) - sum(nums[j:i])) % p == 0

sum(nums[j:i]) can be represented in terms of prefix sums:

sum(nums[j:i]) = presum(i) - presum(j)

-> (sum(nums) - (presum(i) - presum(j))) % p == 0

-> sum(nums) - (presum(i) - presum(j)) = n \* p

-> presum(i) - presum(j) = sum(nums) - n \* p

-> (presum(i) - presum(j)) % p = (sum(nums) - n \* p) % p

Apply distributive law of mods: (a + b) % c = (a % c + b % c) % c

on (sum(nums) - n \* p) % p

-> (presum(i) - presum(j)) % p = (sum(nums) % p - n \* p % p) % p

-> n \* p % p cancels out

-> (presum(i) - presum(j)) % p = (sum(nums) % p) % p

Let's say r = (sum(nums) % p) % p

-> (presum(i) - presum(j)) % p = r

-> presum(i) - presum(j) = n \* p + r

-> presum(j) = presum(i) - n \* p - r

Mod both sides by p again

-> presum(j) % p = (presum(i) - n \* p - r) % p

In equality, the left side 'presum(j) % p' is what we assumed

index 'j' related presum mod p will store in map when index go

till 'j', the right side '(presum(i) - n \* p - r) % p' is the

key that will be calculated in each iteration, now we apply

distributive law of mods: (a + b) % c = (a % c + b % c) % c on

right side '(presum(i) - n \* p - r) % p' again:

-> presum(j) % p = (presum(i) - n \* p - r) % p

-> n \* p % p cancels out

-> presum(j) % p = (presum(i) - r) % p

Recall we have define 'r = (sum(nums) % p) % p' earlier

-> presum(j) % p = (presum(i) % p - (sum(nums) % p) % p) % p

-> presum(j) % p = (presum(i) % p - sum(nums) % p) % p

To avoid negative mod value, we plus p on right side

-> presum(j) % p = (presum(i) % p - sum(nums) % p + p) % p

Now '(presum(i) % p - sum(nums) % p + p) % p' is the key that will

be calculated for each iteration and looking for in map if any

already stored (previously store in map from each 'presum(j) % p')

**Code**

class Solution:

def minSubarray(self, nums: List[int], p: int) -> int:

n = len(nums)

target = sum(nums)%p

if not target:

return 0

answer = n

prefix\_sum = 0

hashmap = {0: -1}

for i, num in enumerate(nums):

prefix\_sum += num

key = (prefix\_sum%p - target)%p

if key in hashmap:

answer = min(answer, i-hashmap[key])

hashmap[prefix\_sum%p] = i

return answer if answer < n else -1

**Refer to**

<https://algo.monster/liteproblems/1590>

**Problem Description**

The task at hand is to find the shortest contiguous subarray that can be removed from a given array of positive integers nums, such that the sum of the remaining elements is divisible by a given integer p. The subarray to be removed can range in size from zero (meaning no elements need to be removed) to one less than the size of the array (since removing the entire array isn't allowed). If it's impossible to find such a subarray, the function should return -1.

This is a modulo-based problem dealing with the concept of remainders. When we talk about the sum of the remaining elements being divisible by p, the sum modulo p should be 0 (that is sum % p == 0).

**Intuition**

The keyword in this problem is "divisibility by p", which involves understanding how the modulo operation works. To arrive at the solution, we need to find a subarray such that when it's removed, the sum of the remaining elements of the array is a multiple of p.

The intuition behind the solution lies in two key observations:

Prefix Sum and Modulo: Compute the cumulative sum of elements as you traverse through the array, taking the modulo with p at each step. This helps us detect if by removing a previous part of the sequence, we can achieve a sum that's a multiple of p.

Using a Hash Map to Remember Modulo Indices: By keeping track of the indices where each modulo value is first seen in a hash map, we can quickly find out where to cut the subarray. If the current modulo value minus the target modulo value has been seen before, the segment between that index and the current index could potentially be removed to satisfy the problem's requirements.

If the sum of the entire array modulo p is 0, no removal is needed (the result is zero subarray length). If the sum modulo p equals k, we need to remove a segment of the array with a sum that is equivalent to k modulo p. The solution uses this approach to find the minimum length subarray that satisfies the condition.

**Solution Approach**

The solution approach uses a hash map (or dictionary in Python) and a prefix sum concept combined with the modulo operation. Here's how the implementation works, broken down step by step:

Calculation of the overall sum modulo p: The variable k holds the result of total sum modulo p which helps us identify what sum value needs to be removed (if possible) to make the overall sum divisible by p.

If k is 0, nothing needs to be removed since the total sum is already divisible by p. The solution will return 0 in this case.

Initialization of a hash map last with a key-value pair {0: -1} which tracks the modulus of the prefix sum and its index.

Loop through the array using enumerate, which gives both the index i and the element x.

Update the current prefix sum modulo p, store it in cur.

Compute target, which is the prefix sum that we need to find in the last hash map. This is calculated as (cur - k + p) % p.

If the target is found in the last map, this means there exists a subarray whose sum modulo p is exactly k, and we could remove it to satisfy the condition. Update the ans with the minimum length found so far.

Update the hash map last with the current prefix sum modulo p and its index.

After finishing the loop, check if ans is still equal to the length of the array (which means no valid subarray was found) and return -1. Otherwise, return the ans which is the length of the smallest subarray to remove.

**Example Walkthrough**

Let's consider an example to illustrate the solution approach. Suppose we have an array of integers nums = [3, 1, 4, 6] and an integer p = 5. Our goal is to find the shortest contiguous subarray that can be removed so that the sum of the remaining elements in the array is divisible by p.

Here's how we would apply the steps of the given solution approach:

We calculate the overall sum of the array, which is 3 + 1 + 4 + 6 = 14. Since we are concerned with the modulus, we compute 14 % 5 which gives us k = 4. This means we need to remove a subarray with a sum that is 4 modulo p.

Since k is not 0, we need to find a subarray to remove. Otherwise, if it were 0, we would return 0 right away because no removal is necessary.

We initialize a hash map last with {0: -1} to track the prefix sums' modulo values and their indices.

Now we begin to loop through the array nums.

At index 0, with element 3, cur = 3 % 5 = 3. We compute target = (3 - 4 + 5) % 5 = 4. Since target is not in last, nothing happens.

The hash map last is now updated to {0: -1, 3: 0}.

At index 1, with element 1, cur = (3 + 1) % 5 = 4 % 5 = 4. The target = (4 - 4 + 5) % 5 = 0. The target is in last, so we find a potential subarray from index -1 to 1 (length 2).

The hash map last is updated to {0: -1, 3: 0, 4: 1}.

At index 2, with element 4, cur = (4 + 4) % 5 = 3. The target = (3 - 4 + 5) % 5 = 4, and last already has a 4. However, this does not give us a smaller subarray than before.

The hash map last is updated to {0: -1, 3: 0, 4: 1}.

At index 3, with element 6, cur = (3 + 6) % 5 = 4. The target = (4 - 4 + 5) % 5 = 0 again. We see that target is in last, so we find a potential subarray from index -1 to 3 (length 4), which is not smaller than the previous.

The hash map last is now {0: -1, 3: 0, 4: 1}.

The shortest subarray we can remove is from indices -1 to 1 which gives us a length of 2.

Since we were able to find such a subarray, we do not return -1. Instead, we return the length of the shortest subarray we found, which is 2.

So the answer for the input nums = [3, 1, 4, 6] and p = 5 is 2, meaning the shortest subarray that we can remove to make the remaining elements' sum divisible by 5 is of length 2.

**Java Solution**

class Solution {

    public int minSubarray(int[] nums, int p) {

        // Initialize remainder to accumulate the sum of the array elements modulo p

        int remainder = 0;

        for (int num : nums) {

            remainder = (remainder + num) % p;

        }

        // If the total sum is a multiple of p, no subarray needs to be removed

        if (remainder == 0) {

            return 0;

        }

        // Create a hashmap to store the most recent index where a certain modulo value was seen

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

        lastIndex.put(0, -1); // Initialize with the value 0 at index -1

        int n = nums.length;

        // Set the initial smallest subarray length to the array's length

        int smallestLength = n;

        int currentSumModP = 0; // This will keep the running sum modulo p

        for (int i = 0; i < n; ++i) {

            currentSumModP = (currentSumModP + nums[i]) % p;

            // Calculate the target modulo value that would achieve our remainder if removed

            int target = (currentSumModP - remainder + p) % p;

            // If the target already exists in the hashmap, calculate the length of the subarray that could be removed

            if (lastIndex.containsKey(target)) {

                smallestLength = Math.min(smallestLength, i - lastIndex.get(target));

            }

            // Update the hashmap with the current modulo value and its index

            lastIndex.put(currentSumModP, i);

        }

        // If the smallestLength was not updated, return -1 to signify no valid subarray exists

        return smallestLength == n ? -1 : smallestLength;

    }

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the given code is O(n), where n is the length of the input list nums. Here's why:

There is a single loop that iterates over all the elements in nums. Inside the loop, the operations are a constant time: updating cur, calculating target, and checking if target in last.

The in operation for the last dictionary, which is checking if the target is present in the keys of last, is an O(1) operation on average because dictionary lookups in Python are assumed to be constant time under average conditions.

So, combining these together, we see that the time complexity is proportional to the length of nums, hence O(n).

**Space Complexity**

The space complexity of the given code is also O(n), where n is the length of the input list nums. Here's why:

A dictionary last is maintained to store indices of the prefix sums. In the worst case, if all the prefix sums are unique, the size of the dictionary could grow up to n.

There are only a few other integer variables which don't depend on the size of the input, so their space usage is O(1).

Therefore, because the predominant factor is the size of the last dictionary, the space complexity is O(n).

**Refer to**

[L974.Subarray Sums Divisible by K (Ref.L560,L523)](note://WEBde138f801994c4eb8fff2da131b5856f)

[L560.Subarray Sum Equals K](note://5CE29DE019904808986311AFAC74C85C)

[L523.Continuous Subarray Sum (Ref.L974)](note://WEBc2ea24cf3f984fc0adf073f4552f8db8)