1590. Make Sum Divisible by P Medium Array Hash Table **Prefix Sum**

Problem Description

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

The task at hand is to find the shortest contiguous subarray that can be removed from a given array of positive integers nums,

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 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 р.

- 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
- 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.

The solution approach uses a hash map (or dictionary in Python) and a <u>prefix sum</u> 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

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.

Solution Approach

problem's requirements.

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 \mathbf{i} and the element \mathbf{x} .

- Update the current <u>prefix sum</u> 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

The data structure used here is a Hash Map (or Dictionary), which allows for an efficient lookup to find whether we have

previously encountered a specific prefix sum modulo p. The algorithm is a manifestation of a sliding window where the window is

This approach efficiently solves the problem by transforming it into a scenario to find two prefix sums with the same modulo after

return -1. Otherwise, return the ans which is the length of the smallest subarray to remove.

dynamically adjusted based on the prefix sums and the target modulo values.

sum value needs to be removed (if possible) to make the overall sum divisible by p.

removing the elements from between these two sums. By using the hash map, we are able to quickly find out if we've seen a prefix sum that allows us to create a valid sum divisible by p when the subarray between two such prefix sum occurrences is

Let's reconsider the example with the array of integers nums = [3, 1, 4, 6] and the integer p = 5. Our objective remains to identify the shortest contiguous subarray that, when removed, results in the sum of the remaining elements being divisible by p. Following the steps outlined in the solution approach: We calculate the overall sum of the array, which is 3 + 1 + 4 + 6 = 14. The modulus of this sum with p gives us 14 % 5 = 14

Since k is not 0, we understand that some elements need to be removed. If k were 0, it would imply no removal is

4, which means k = 4. This indicates that we need to remove a subarray whose sum modulo p equals 4.

We initialize a hash map last with the entry {0: -1}. This map is used to keep track of the indices where each modulo value of the prefix sum is first encountered.

necessary, and we could return 0.

Updating last to {0: -1, 3: 0, 4: 1}.

10, which is divisible by p=5.

remainder = sum(nums) % p

if remainder == 0:

min_length = len(nums)

for index, num in enumerate(nums):

Update the current mod value

mod_indices[current_mod] = index

current mod = (current mod + num) % p

target_mod = (current_mod - remainder + p) % p

return -1 if min_length == len(nums) else min_length

If the target mod value is found in the mod_indices

return 0

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

Find the remainder of the sum of nums when divided by p

If the sum of nums is already divisible by p, the subarray length is 0

Iterate through the numbers in the array to find the shortest subarray

If min length hasn't been updated, the required subarray doesn't exist

Hash map to store the most recent index where a particular mod value is found

Python

class Solution:

from typing import List

As we iterate through nums:

removed.

Example Walkthrough

 \circ At index 0 with element 3, cur = 3 % 5 = 3. We calculate target = (3 - 4 + 5) % 5 = 4. Since target is not found in last, we proceed without any removal. Updating last to {0: -1, 3: 0}.

○ At index 1 with element 1, cur = (3 + 1) % 5 = 4. We calculate target = (4 - 4 + 5) % 5 = 0. The target is found in last, indicating

a potential subarray from index -1 to 1. However, this does not provide a valid subarray for removal based on our current understanding.

○ At index 2 with element 4, cur = (4 + 4) % 5 = 3. We calculate target = (3 - 4 + 5) % 5 = 4, and last already has a 4. This again does not yield a smaller subarray than before. The hash map last remains {0: -1, 3: 0, 4: 1}. \circ At index 3 with element 6, cur = (3 + 6) % 5 = 4. We calculate target = (4 - 4 + 5) % 5 = 0 again. The target is in last,

Upon closer examination, we realize the mistake in our previous assessment. The correct approach is to identify the subarray

[4] at index 2, which has a length of 1 and a sum of 4, which is exactly k. Removing this subarray leaves us with a sum of

Therefore, the corrected answer for the input nums = [3, 1, 4, 6] and p = 5 is 1. This indicates that the shortest subarray

suggesting a potential subarray from index -1 to 3. However, this insight needs correction.

- we can remove to make the sum of the remaining elements divisible by 5 has a length of 1. Solution Implementation
 - mod indices = $\{0: -1\}$ # The current prefix sum mod p current mod = 0 # Initialize minimum length to the length of nums array

Calculate the target mod value which would balance the current mod to make a divisible sum

if target mod in mod indices: # Update the min length if a shorter subarray is found min length = min(min length, index - mod indices[target_mod]) # Update the mod indices with the current index

Java

```
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.containsKev(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;
C++
```

```
if (remainder === 0) {
    return 0;
// Create a map to store the last index where a particular remainder was found.
const lastIndexOfRemainder = new Map<number, number>();
```

TypeScript

let remainder = 0:

for (const num of nums) {

lastIndexOfRemainder.set(0, -1);

const n = nums.length;

};

class Solution {

int minSubarray(vector<int>& nums, int p) {

// Calculate the sum of nums mod p.

unordered map<int, int> modIndexMap;

// Iterate through the nums array.

if (modIndexMap.count(target)) {

modIndexMap[currentSum] = i;

return minLength == n ? -1 : minLength;

function minSubarray(nums: number[], p: number): number {

// Calculate the sum of the array elements modulo p.

// Get the total number of elements in the array.

// Initialize answer as the length of the array.

remainder = (remainder + num) % p;

for (int i = 0; i < n; ++i) {

int n = nums.size(); // The length of the nums array.

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

minLength = min(minLength, i - modIndexMap[target]);

// Initialize a variable to store the remainder of the array sum modulo p.

// Map the remainder 0 to the index before the start of the array.

// If the remainder is 0, the entire array is already divisible by p, so return 0.

int currentSum = 0; // Running sum of the elements.

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

remainder = (remainder + num) % p;

for (int& num : nums) {

if (remainder == 0) {

modIndexMap[0] = -1;

return 0;

int remainder = 0; // Use 'remainder' to store the mod value of the sum of array.

// Use a hashmap to store the most recent index where a certain mod value was seen.

// Update the map with the current cumulative mod value and current index.

// If minLength is not changed, return -1 for no such subarray, otherwise return the minLength.

// Calculate the target mod value that could potentially reduce the running sum to a multiple of p.

// If the target is found in the map, update the minLength with the shortest length found so far.

int minLength = n; // Initialize minLength with the maximum possible length.

// If the remainder is 0, the whole array satisfies the condition.

public:

```
let answer = n;
    // Initialize a variable to store the current prefix sum modulo p.
    let currentPrefixSum = 0;
    // Iterate through the array to find the minimum length of subarray.
    for (let i = 0; i < n; ++i) {
       // Update the current prefix sum.
       currentPrefixSum = (currentPrefixSum + nums[i]) % p;
       // Calculate the target remainder we want to find in the map.
        const targetRemainder = (currentPrefixSum - remainder + p) % p;
       // Check if we have previously seen this target remainder.
        if (lastIndexOfRemainder.has(targetRemainder)) {
            // Get the last index where this remainder was seen.
            const lastIndex = lastIndexOfRemainder.get(targetRemainder)!;
            // Update answer with the minimum length found so far.
            answer = Math.min(answer, i - lastIndex);
       // Update the map with the current prefix sum and its corresponding index.
        lastIndexOfRemainder.set(currentPrefixSum, i);
   // If answer is still equal to n, a valid subarray of length less than n was not found.
   // Therefore, return -1. Otherwise, return answer.
   return answer === n ? -1 : answer;
from typing import List
class Solution:
   def minSubarray(self, nums: List[int], p: int) -> int:
       # Find the remainder of the sum of nums when divided by p
       remainder = sum(nums) % p
       # If the sum of nums is already divisible by p, the subarray length is 0
       if remainder == 0:
            return 0
       # Hash map to store the most recent index where a particular mod value is found
       mod indices = \{0: -1\}
       # The current prefix sum mod p
       current mod = 0
       # Initialize minimum length to the length of nums array
       min_length = len(nums)
       # Iterate through the numbers in the array to find the shortest subarray
        for index, num in enumerate(nums):
           # Update the current mod value
           current mod = (current mod + num) % p
           # Calculate the target mod value which would balance the current mod to make a divisible sum
            target_mod = (current_mod - remainder + p) % p
           # If the target mod value is found in the mod_indices
           if target mod in mod indices:
                # Update the min length if a shorter subarray is found
               min length = min(min length, index - mod indices[target_mod])
           # Update the mod indices with the current index
           mod_indices[current_mod] = index
       # If min length hasn't been updated, the required subarray doesn't exist
```

Time Complexity

could grow up to n.

Time and Space Complexity

return -1 if min_length == len(nums) else min_length

• 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

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

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).

• 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

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:

• There are only a few other integer variables which don't depend on the size of the input, so their space usage is 0(1). Therefore, because the predominant factor is the size of the last dictionary, the space complexity is last dictionary.