1590. Make Sum Divisible by P

Medium Array Hash Table Prefix Sum Leetcode Link

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:

1. **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`.

2. **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:

1. 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`.

2. 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.

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

4. 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`.

5. 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.

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

7. 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.

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

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

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:

1. 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`.

2. 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.

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

4. 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}`.

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

6. 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`.

Python Solution

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

```

Java Solution

```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.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;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minSubarray(vector<int>& nums, int p) {
        int remainder = 0; // Use 'remainder' to store the mod value of the sum of array.

        // Calculate the sum of nums mod p.
        for (int& num : nums) {
            remainder = (remainder + num) % p;
        }

        // If the remainder is 0, the whole array satisfies the condition.
        if (remainder == 0) {
            return 0;
        }

        // Use a hashmap to store the most recent index where a certain mod value was seen.
        unordered_map<int, int> modIndexMap;
        modIndexMap[0] = -1;
        int n = nums.size(); // The length of the nums array.
        int minLength = n; // Initialize minLength with the maximum possible length.
        int currentSum = 0; // Running sum of the elements.

        // Iterate through the nums array.
        for (int i = 0; i < n; ++i) {
            currentSum = (currentSum + nums[i]) % p;

            // Calculate the target mod value that could potentially reduce the running sum to a multiple of p.
            int target = (currentSum - remainder + p) % p;

            // If the target is found in the map, update the minLength with the shortest length found so far.
            if (modIndexMap.count(target)) {
                minLength = min(minLength, i - modIndexMap[target]);
            }

            // Update the map with the current cumulative mod value and current index.
            modIndexMap[currentSum] = i;
        }

        // If minLength is not changed, return -1 for no such subarray, otherwise return the minLength.
        return minLength == n ? -1 : minLength;
    }
};

```

Typescript Solution

```typescript
function minSubarray(nums: number[], p: number): number {
    // Initialize a variable to store the remainder of the array sum modulo p.
    let remainder = 0;
    // Calculate the sum of the array elements modulo p.
    for (const num of nums) {
        remainder = (remainder + num) % p;
    }
    // If the remainder is 0, the entire array is already divisible by p, so return 0.
    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>();
    // Map the remainder 0 to the index before the start of the array.
    lastIndexOfRemainder.set(0, -1);
    // Get the total number of elements in the array.
    const n = nums.length;
    // Initialize answer as the length of the array.
    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;
}

```

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)`.