Number Theory

Sorting

#### **Problem Description**

**Array** 

**Math** 

Medium

continues back at the first element. The task is to apply operations to the elements of the array, where each operation either increases or decreases an element by 1. The goal is to reach a state where the sums of all subarrays of a given length k are equal.

A subarray is defined as a contiguous part of the array, but since the array is circular, this includes subarrays that wrap around

In this problem, we are dealing with a circular integer array, which means that after the last element of the array, the sequence

the end of the array to the beginning. The challenge is to find the minimum number of operations required to equalize the sum of all k-length subarrays.

Intuition

#### The key to solving this problem lies in understanding that when the array is circular and we want the sum of every subarray of length k to be equal, we are ultimately seeking to make the elements of the array equal in certain groups.

Here's why grouping matters: Since the array is circular, for n elements and subarray size k, there are certain elements that will always be together in every subarray of size k. These elements form a group and they should be equalized so that their total sum

common divisor (GCD) of n and k. Once we've determined the GCD, we get the number of groups in which the array needs to be divided. Each group consists of elements that are  $n/\gcd(n, k)$  positions apart. Sorting these elements and choosing the middle element as the target minimizes the number of operations needed to equalize all elements in the group. This is because the median minimizes the sum of absolute

remains constant in every possible k-sized subarray they are part of. The number of such groups is determined by the greatest

deviations. For each group, calculate and accumulate the number of operations needed to make all elements equal to the median. This will give us the minimum number of operations required to reach our goal.

Thus, the essence of the solution is to calculate the GCD, divide the array into groups, find the median of each group, and then perform the operations to reach the median for each element within the group. The sum of all operations across the groups is the answer we are looking for.

Solution Approach

The implementation of the solution follows a simple yet clever approach that significantly reduces the overall problem complexity.

GCD Calculation: The first step is to calculate the greatest common divisor (GCD) of the array length n and the subarray size

k. This is done using the Python gcd function. The GCD tells us into how many groups we can divide the array elements such

# that within each group, the elements will be part of the same set of k-length subarrays due to array circularity. g = gcd(n, k)

all elements in the group equal.

mid = t[len(t) >> 1]

2. **Grouping and Sorting:** Next, iterate over a range from 0 to the GCD obtained, treating each iteration index i as the starting point of a new group. The members of each group are collected by taking elements at intervals of n/g from the starting index

i.
 t = sorted(arr[i:n:g])
 Each group is then separately sorted to allow for the calculation of the median. <u>Sorting</u> is essential because the median

minimizes the sum of absolute deviations from any point, which corresponds to the minimum number of operations to make

Find Median and Calculate Operations: The median element for each sorted group is identified. Since Python lists are 0-

For each group, we then calculate the sum of the absolute differences between each element and the median, which corresponds to the count of increase/decrease operations needed to make that element equal to the median.

ans += sum(abs(x - mid) for x in t)

**Accumulate Results**: Add the count of operations from each group to ans, the accumulator for the total operations.

indexed, right-shifting the length len(t) by 1(len(t) >> 1) gives the middle index for the median value.

This approach efficiently uses the properties of a circular array and the subarray size to minimize the problem's complexity to a more manageable level. The usage of a <u>sorting</u> function and a median to calculate the minimal number of operations is both an

Finally, after processing all groups, ans is returned, giving us the desired minimum number of operations.

Let's walk through a small example to illustrate the solution approach:

Consider we have a circular integer array arr = [1, 3, 2, 6, 5] and we want to equalize the sum of all subarrays of length k = 3.

**GCD Calculation**: We first calculate the greatest common divisor (GCD) of the array length n = 5 and the subarray size k = 3.

### 2. **Grouping and Sorting**: Since the GCD is 1, it indicates that each element of arr will form its own group when considering

**Example Walkthrough** 

Following the solution approach:

g = gcd(5, 3) # g would be 1

from math import gcd

subarrays of length 3.

t = sorted(arr) # t will be [1, 2, 3, 5, 6]

mid = t[len(t) >> 1] # mid will be 3

elements equal to 3.

Solution Implementation

from typing import List

array\_length = len(array)

total\_operations = 0

for i in range(gcd\_value):

return total\_operations

gcd\_value = gcd(array\_length, k)

total\_operations += operations

return (b == 0) ? a : gcd(b, a % b);

from math import gcd

**Python** 

elegant and effective method in reaching the solution.

3. **Find Median and Calculate Operations**: There's no need to actually form groups or sort since each element is its own group, so we proceed directly to equalize the array considering all elements.

The target number to which we'll aim to equalize all of our elements can be selected as the median of the array. For a small

array like ours, we can easily find the median by sorting and then picking the middle element.

4. Accumulate Results: Now, we calculate and accumulate the number of operations: ans = sum(abs(x - mid)) for x in arr) # ans will be 2 (from 1 to 3) + 0 + 0 + 3 (from 6 to 3) + 2 (from 5 to 3), to 3), to 3)

Since the middle element (the median) in the sorted list is 3, we would need to perform operations to make the rest of the

class Solution:
 def makeSubKSumEqual(self, array: List[int], k: int) -> int:
 # Find the length of the array.

# Create a split array consisting of every gcd\_value-th element starting from i,

# Calculate the sum of absolute differences between each element and the median.

# This represents the number of operations to make the subarray elements equal.

# Return the total number of operations needed to make sums of subarrays of size 'k' equal.

# Add the number of operations for this split array to the total operations.

# Calculate the greatest common divisor of the array length and k.

# then sort this split array to find the median.

# Initialize answer to 0, which will hold the total operations needed.

# Iterate over each subgroup which will be modified to have equal sums.

split\_sorted\_array = sorted(array[i:array\_length:gcd\_value])

operations = sum(abs(x - median) for x in split\_sorted\_array)

Hence, the minimum number of operations required to equalize the sum of all 3-length subarrays in arr is 7.

## # Find the median value of the split\_sorted\_array. median = split\_sorted\_array[len(split\_sorted\_array) // 2]

```
Java
class Solution {
    // Function to adjust elements in the array such that all sub-arrays of length k have an equal sum
   public long makeSubKSumEqual(int[] arr, int k) {
        int n = arr.length; // length of the original array
        int gcdValue = gcd(n, k); // calculate the Greatest Common Divisor (GCD) of n and k
        long totalOperations = 0; // to store the total number of operations needed
       // Iterate over the array with increments of gcdValue
        for (int i = 0; i < gcdValue; ++i) {</pre>
            List<Integer> temp = new ArrayList<>(); // temporary list to store elements in the same group
            // Populate the temporary list with elements that are gcdValue apart in the array
            for (int j = i; j < n; j += gcdValue) {</pre>
                temp.add(arr[j]);
            // Sort the temporary list
            temp.sort((a, b) -> a - b);
            // Find the median of the temporary list
            int median = temp.get(temp.size() / 2);
           // Calculate the number of operations to make all elements in the temporary list equal to the median
            for (int element : temp) {
                totalOperations += Math.abs(element - median);
        // Return the total number of operations
        return totalOperations;
    // Helper function to calculate the Greatest Common Divisor (GCD) using Euclidean algorithm
    private int gcd(int a, int b) {
       // Base case for recursion: when b is zero, a is the GCD
```

```
#include <vector>
#include <algorithm> // Include algorithm for std::sort, std::gcd and std::abs
```

C++

**}**;

**TypeScript** 

let maxCost = 0;

let currentSum = 0;

for (const char of s) {

// Return the maximum cost found

# Find the length of the array.

gcd\_value = gcd(array\_length, k)

array\_length = len(array)

total\_operations = 0

for i in range(gcd\_value):

Time and Space Complexity

**Time Complexity** 

```
class Solution {
public:
   // Function to make all possible subarrays of length k have the same sum
   // by changing the values of elements.
    long long makeSubKSumEqual(vector<int>& arr, int k) {
        int n = arr.size(); // Size of the input array
       int g = std::gcd(n, k); // Calculate the greatest common divisor of n and k
        long long answer = 0; // Initialize the answer to 0
       // Iterate over each group formed by the gcd
        for (int i = 0; i < g; ++i) {
            vector<int> temp; // Temporary vector to hold elements of the current group
           // Collect all elements that belong to the same group
            for (int j = i; j < n; j += g) {
                temp.push_back(arr[j]);
           // Sort the elements within the group
            std::sort(temp.begin(), temp.end());
            // Find the median of the group
            int median = temp[temp.size() / 2];
           // Calculate the total cost required to make all elements in the group equal to the median
           for (int x : temp) {
               answer += std::abs(x - median);
       return answer; // Return the total cost
```

function maximumCostSubstring(s: string, chars: string, values: number[]): number {

const costs: number[] = Array.from({ length: 26 }, (\_, i) => i + 1);

// chars.charCodeAt(i) - 97 converts 'a' to 0, 'b' to 1, etc.,

costs[chars.charCodeAt(i) - 'a'.charCodeAt(0)] = values[i];

// Add the cost of the current character to 'currentSum'

for (let i = 0;  $i < chars.length; ++i) {$ 

// because the char code of 'a' is 97

// Iterate over each character in the string 's'

maxCost = Math.max(maxCost, currentSum);

def makeSubKSumEqual(self, array: List[int], k: int) -> int:

# then sort this split array to find the median.

# Find the median value of the split\_sorted\_array.

# Calculate the greatest common divisor of the array length and k.

# Initialize answer to 0, which will hold the total operations needed.

# Iterate over each subgroup which will be modified to have equal sums.

split\_sorted\_array = sorted(array[i:array\_length:gcd\_value])

median = split sorted array[len(split sorted array) // 2]

// Define an array 'costs' to hold the cost values associated with the characters 'a' to 'z'

// Update the cost values for the characters provided in 'chars' using the 'values' array

// Initialize 'maxCost' to 0. It will store the maximum cost of any substring found so far.

// Ensure 'currentSum' does not drop below 0 by comparing it with 0 using Math.max

// Update 'maxCost' if 'currentSum' is greater than the current 'maxCost'

currentSum = Math.max(currentSum, 0) + costs[char.charCodeAt(0) - 'a'.charCodeAt(0)];

// Initialize 'currentSum' to 0. It will store the current sum while iterating through the string.

#### from typing import List from math import gcd

class Solution:

return maxCost;

```
# Calculate the sum of absolute differences between each element and the median.
# This represents the number of operations to make the subarray elements equal.
operations = sum(abs(x - median) for x in split_sorted_array)

# Add the number of operations for this split array to the total operations.
total_operations += operations

# Return the total number of operations needed to make sums of subarrays of size 'k' equal.
return total_operations
```

# Create a split array consisting of every gcd\_value-th element starting from i,

k))).

2. Iterate over the array arr with a step size equal to the GCD g. This results in g iterations. The time complexity of this loop is

The time complexity of the function makeSubKSumEqual consists of the following parts:

- 2. Iterate over the array arr with a step size equal to the GCD g. This results in g itera affected by the sort operation and the summation operation within each iteration.
- Sorting the subarray: The sorting algorithm, typically quicksort in Python's sorted() function, has a time complexity of 0(m log m) where m is the number of elements to sort. Since we only sort n/g elements in each iteration, and we have g such
- Summation operation: The summation operation within the loop has a time complexity of O(n/g) for each iteration since it sums up n/g elements, leading to a total time complexity of O(n/g \* g) which simplifies to O(n) across all g iterations.
  Combining these complexities and considering that n/g dominates n for large values of n, the overall time complexity of the

iterations, the overall time complexity for the sorting part is 0(n/g \* log(n/g) \* g), which simplifies to 0(n log(n/g)).

Calculate the GCD of n and k, where n is the length of the array. The time complexity of calculating the GCD is 0(log(min(n,

function is  $0(n \log(n/g) + n)$  which simplifies to  $0(n \log(n/g))$  as 0(n) is dominated by  $0(n \log(n/g))$ .

Space Complexity

The space used for the GCD operation, which is 0(1) since it only uses a constant amount of extra space.

The space complexity of the function makeSubKSumEqual consists of the following parts:

- 2. The space used for sorting the subarray. The sorted subarray t creates a new list at each iteration that can contain up to n/g elements. However, since these lists are not stored simultaneously and each is generated in its own iteration, the space
- complexity is 0(n/g).

  Hence, the space complexity of the function is 0(n/g). However, since g cannot be greater than n, we can say that the space

complexity is at worst O(n).

Overall, the function makeSubKSumEqual has a time complexity of  $O(n \log(n/g))$  and a space complexity of O(n).