2607. Make K-Subarray Sums Equal

Medium Array Math Number Theory Sorting

Leetcode Link

Problem Description

In this problem, we are dealing with a circular integer array, which means that after the last element of the array, the sequence 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 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 remains constant in every possible `k`-sized subarray they are part of. The number of such groups is determined by the greatest 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 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.

1. **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.

   ```
   1 g = gcd(n, k)
   ```

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`.

   ```
   1 t = sorted(arr[i::g])
   ```

   Each group is then separately sorted to allow for the calculation of the median. Sorting is essential because the median minimizes the sum of absolute deviations from any point, which corresponds to the minimum number of operations to make all elements in the group equal.

3. **Find Median and Calculate Operations**: The median element for each sorted group is identified. Since Python lists are 0-indexed, right-shifting the length `len(t)` by 1 (`len(t) >> 1`) gives the middle index for the median value.

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

   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.

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

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

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

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 sorting function and a median to calculate the minimal number of operations is both an elegant and effective method in reaching the solution.

Example Walkthrough

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`.

Following the solution approach:

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

   ```
   1 from math import gcd
   2
   3 g = gcd(5, 3)  # g would be 1
   ```

2. **Grouping and Sorting**: Since the GCD is `1`, it indicates that each element of `arr` will form its own group when considering subarrays of length `3`.

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.

   ```
   1 t = sorted(arr)  # t will be [1, 2, 3, 5, 6]
   2 mid = t[len(t) >> 1]  # mid will be 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 elements equal to `3`.

4. **Accumulate Results**: Now, we calculate and accumulate the number of operations:

   ```
   1 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), therefore ans = 7
   ```

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

Python Solution

```
1  from typing import List
2  from math import gcd
3
4  class Solution:
5      def makeSubKSumEqual(self, arr: List[int], k: int) -> int:
6          # Find the length of the array.
7          array_length = len(arr)
8
9          # Calculate the greatest common divisor of the array length and k.
10         gcd_value = gcd(array_length, k)
11
12         # Initialize answer to 0, which will hold the total operations needed.
13         total_operations = 0
14
15         # Iterate over each subgroup which will be modified to have equal sums.
16         for i in range(gcd_value):
17             # Create a split array consisting of every gcd_value-th element starting from i.
18             # then sort this split array to find the median.
19             split_sorted_array = sorted(arr[i:array_length:gcd_value])
20
21             # Find the median value of the split_sorted_array.
22             median = split_sorted_array[len(split_sorted_array) // 2]
23
24             # Calculate the sum of absolute differences between each element and the median.
25             # This represents the number of operations to make the subarray elements equal.
26             operations = sum(abs(x - median) for x in split_sorted_array)
27
28             # Add the number of operations for this split array to the total operations.
29             total_operations += operations
30
31         # Return the total number of operations needed to make sums of subarrays of size 'k' equal.
32         return total_operations
33
```

Java Solution

```
1  class Solution {
2
3      // Function to adjust elements in the array such that all sub-arrays of length k have an equal sum
4      public long makeSubKSumEqual(int[] arr, int k) {
5          int n = arr.length; // length of the original array
6          int gcdValue = gcd(n, k); // Calculate the Greatest Common Divisor (GCD) of n and k
7          long totalOperations = 0; // To store the total number of operations needed
8
9          // Iterate over the array with increments of gcdValue
10         for (int i = 0; i < gcdValue; ++i) {
11             List<Integer> temp = new ArrayList<>(); // temporary list to store elements in the same group
12
13             // Populate the temporary list with elements that are gcdValue apart in the array
14             for (int j = i; j < n; j += gcdValue) {
15                 temp.add(arr[j]);
16             }
17
18             // Sort the temporary list
19             temp.sort((a, b) -> a - b);
20
21             // Find the median of the temporary list
22             int median = temp.get(temp.size() / 2);
23
24             // Calculate the number of operations to make all elements in the temporary list equal to the median
25             for (int element : temp) {
26                 totalOperations += Math.abs(element - median);
27             }
28         }
29
30         // Return the total number of operations
31         return totalOperations;
32     }
33
34     // Helper function to calculate the Greatest Common Divisor (GCD) using Euclidean algorithm
35     private int gcd(int a, int b) {
36         // Base case: if second number is zero, a is the GCD
37         return b == 0 ? a : gcd(b, a % b);
38     }
39 }
40
```

C++ Solution

```
1  #include <vector>
2  #include <algorithm> // Include algorithm for std::sort, std::gcd and std::abs
3
4  class Solution {
5  public:
6      // Function to make all possible subarrays of length k have the same sum
7      // by changing the values of elements.
8      long long makeSubKSumEqual(vector<int>& arr, int k) {
9          int n = arr.size(); // Size of the input array.
10         int g = std::gcd(n, k); // Calculate the greatest common divisor of n and k
11         long long answer = 0; // Initialize the answer to 0
12
13         // Iterate over each group formed by the gcd
14         for (int i = 0; i < g; ++i) {
15             vector<int> temp; // Temporary vector to hold elements of the current group
16
17             // Collect all elements that belong to the same group
18             for (int j = i; j < n; j += g) {
19                 temp.push_back(arr[j]);
20             }
21
22             // Sort the elements within the group
23             std::sort(temp.begin(), temp.end());
24
25             // Find the median of the group
26             int median = temp[temp.size() / 2];
27
28             // Calculate the total cost required to make all elements in the group equal to the median
29             for (int x : temp) {
30                 answer += std::abs(x - median);
31             }
32         }
33
34         return answer; // Return the total cost
35     }
36 };
37
```

Typescript Solution

```
1  function maximumCostSubstring(s: string, chars: string, values: number[]): number {
2      // Define an array 'costs' to hold the cost values associated with the characters 'a' to 'z'
3      const costs: number[] = Array.from({ length: 26 }, (_, i) => i + 1);
4
5      // Update the cost values for the characters provided in 'chars' using the 'values' array
6      for (let i = 0; i < chars.length; ++i) {
7          // chars.charCodeAt(i) - 97 converts 'a' to 0, 'b' to 1, etc.,
8          // because the char code of 'a' is 97
9          costs[chars.charCodeAt(i) - 'a'.charCodeAt(0)] = values[i];
10     }
11
12     // Initialize 'maxCost' to 0. It will store the maximum cost of any substring found so far.
13     let maxCost = 0;
14
15     // Initialize 'currentSum' to 0. It will store the current sum while iterating through the string.
16     let currentSum = 0;
17
18     // Iterate over each character in the string 's'
19     for (const ch of s) {
20         // Add the cost of the current character to 'currentSum'
21         // Ensure 'currentSum' does not drop below 0 by comparing it with 0 using Math.max
22         currentSum = Math.max(currentSum, 0) + costs[ch.charCodeAt(0) - 'a'.charCodeAt(0)];
23
24         // Update 'maxCost' if 'currentSum' is greater than the current 'maxCost'
25         maxCost = Math.max(maxCost, currentSum);
26     }
27
28     // Return the maximum cost found
29     return maxCost;
30 }
31
```

Time and Space Complexity

Time Complexity

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

1. Calculate the GCD of `n` and `k`, where `n` is the length of the array. The time complexity of calculating the GCD is $O(\log(\min(n, k)))$.

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 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 $O(m \log m)$ where `m` is the number of elements to sort. Since we only sort `n/g` elements in each iteration, and we have `g` such iterations, the overall time complexity for the sorting part is $O(n/g * \log(n/g) * g)$, which simplifies to $O(n \log(n/g))$.
   - 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 function is $O(n \log(n/g) + n)$ which simplifies to $O(n \log(n/g))$ as $O(n)$ is dominated by $O(n \log(n/g))$.

Space Complexity

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

1. The space used for the GCD operation, which is $O(1)$ since it only uses a constant amount of extra space.
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 $O(n/g)$.

Hence, the space complexity of the function is $O(n/g)$. 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 $O(n)$.