2243. Calculate Digit Sum of a String

Easy String Simulation

Leetcode Link

Problem Description

The problem presents a transformation process of a string `s` composed of digits according to an integer `k`. The overall goal is to repeatedly transform `s` by dividing it into groups, summing the digits in those groups, then combining those sums, until the string's length is no longer greater than `k`.

Specifically, the string `s` should go through rounds of modification with the following steps:

1. Divide `s` into consecutive groups of size `k`. This means `s` is split every `k` characters, with the last group possibly being shorter if there aren't enough characters left to form a complete group of size `k`.
2. For each group, calculate the sum of all its digits and replace the group with a single string representing that sum.
3. Merge the resulting strings from the sums to form a new string `s`.
4. If the new string's length is still greater than `k`, repeat the entire process.

The result we are looking for is the final state of the string `s` after no more rounds can be completed, meaning its length is no longer greater than `k`.

Intuition

To solve this problem, we will essentially simulate the rounds described in the problem statement. Given the iterative nature of the problem, a loop works well for our purposes. At each iteration, we check if the length of `s` exceeds `k`. If it doesn't, we are done.

If we need to process the string, we follow these steps:

1. Initialize a temporary list `t` to hold the sums of each group.
2. Loop through the string in increments of `k`. At each step, we'll take a slice of `s`, which is the next group of digits we need to sum.
3. Within each group, we convert each character to an integer and sum them up.
4. We convert the sum back to a string and add it to the list `t`.
5. After going through all groups, we join the list `t` into a new string. This is the string after one complete round.
6. The loop then repeats this process, working on the new string until its length is less or equal to `k`.

The solution relies on repeatedly applying the same transformation (dividing into groups, summing, recombining) until a condition (string length <= k) is met, which is a common approach for iterative problems like this.

Solution Approach

The solution provided uses a loop and a couple of nested loops to implement the steps outlined in the intuition. Here's an in-depth look at how the implementation corresponds to the algorithm:

```
class Solution:
    def digitSum(self, s: str, k: int) -> str:
        while len(s) > k: # Continue processing until `s` is length is <= `k`
            t = [] # List to store the sums of each group
            n = len(s) # Current length of `s`
            for i in range(0, n, k): # Loop in steps of `k`
                s = 0 # Variable to store the sum of digits in the current group
                for j in range(i, min(i + k, n)): # Iterate through the current group
                    s += int(s[j]) # Sum the digits, converting each character to an integer
                t.append(str(s)) # Add the summed digit to the list `t` as a string
            s = "".join(t) # Combine the elements of `t` into a new string `s`
        return s
```

Key points of the solution:

- **Loops**: There is an outer `while` loop controlling the number of rounds based on the length of `s`. Inside it, there's a `for` loop specifying the division of `s` into groups of size `k` and a nested loop for summing the digits.
- **String Slicing**: The innermost `for` loop performs slicing of the string `s` such that each slice corresponds to a group of size `k`, with special handling for the last group that could be shorter.
- **List `t`**: This list holds the intermediate sums as strings for each group. The list is recombined (`joined`) into a new string after each round.
- **Integer Conversion**: Each digit from the groups is converted to an integer for summation.
- **String Conversion**: The sum of the group's digits is converted back to a string before storing it in the list `t`, preserving the form required for the next round or the final result.

The elegance of this solution lies in its adherence to the problem statement's process and its use of standard Python features, such as loops for iteration, slicing for grouping, and list manipulation for combining the digits. It's a straightforward implementation that emphasizes clear representation of the problem's transformation steps in code.

Example Walkthrough

Let's illustrate the solution approach using a small example.

Suppose we have the string `s = "1234567890"` and `k = 3`. We are to repeatedly transform `s` to meet the condition that the string's length is no longer greater than `k`.

Initial String: "1234567890"

Step 1: Divide into groups of size `k = 3`

- Group 1: "123"
- Group 2: "456"
- Group 3: "789"

Step 2: Sum the digits in each group and form a new string

- Group 1 Sum: 1 + 2 + 3 = 6
- Group 2 Sum: 4 + 5 + 6 = 15
- Group 3 Sum: 7 + 8 + 9 = 24

The intermediate strings representing each sum are "6", "15", and "24".

Step 3: Merge the strings of sums to form a new string `s`

- New String: "61524"

Since the length of the new string (5) is still greater than `k` (3), we repeat the process:

New String After First Round: "61524"

Step 4: Divide into groups of size `k = 3`

- Group 1 New: "615"
- Group 2 New: "24" (Note: This group is smaller because there are not enough characters left.)

Step 5: Sum the digits in each group and form a new string

- Group 1 New Sum: 6 + 1 + 5 = 12
- Group 2 New Sum: 2 + 4 = 6

The intermediate strings representing each sum are "12" and "6".

Step 6: Merge the strings of sums to form a new string `s`

- New String: "126"

The length of the new string (3) is equal to `k` (3), so the process is complete.

Final Result: "126"

This transformed string is the final answer because its length (3) is no longer greater than `k` (3). Following this approach, we could implement it in Python using the given solution code.

Python Solution

```
class Solution:
    def digitSum(self, s: str, k: int) -> str:
        # Continue the process until the length of the string `s` is less than or equal to `k`
        while len(s) > k:
            # Initialize an empty list to store the sum of each group
            groups = []
            # Calculate the length of the string `s`
            length_of_s = len(s)
            # Split the string into groups of size `k` and sum each group
            for i in range(0, length_of_s, k):
                # Initialize the sum for the current group to 0
                group_sum = 0
                # Sum all digits in the current group
                for j in range(i, min(i + k, length_of_s)):
                    group_sum += int(s[j])
                # Append the sum of this group to the list as a string
                groups.append(str(group_sum))
            # Join all the group sums to form the new string `s`
            s = "".join(group_sums)
        # Return the final string `s` after the while loop ends
        return s


```

Java Solution

```
class Solution {

    // Function to calculate the digit sum of a string according to the given rules
    public String digitSumOfString(String s, int k) {

        // Continue the loop until the string 's' length is greater than 'k'
        while (s.length() > k) {

            // Get the current length of the string 's'
            int stringLength = s.length();

            // StringBuilder to build the new string after calculating the digit sum
            StringBuilder temporaryString = new StringBuilder();

            // Loop through the string in chunks of size 'k' or smaller if at the end of the string
            for (int i = 0; i < stringLength; i += k) {

                // Initialize the sum for the current chunk to 0
                int chunkSum = 0;

                // Calculate the sum for the current chunk
                for (int j = i; j < Math.min(i + k, stringLength); ++j) {
                    chunkSum += s.charAt(j) - '0'; // Convert the character to an integer and add to chunkSum
                }

                // Append the sum of the current chunk to 'temporaryString'
                temporaryString.append(chunkSum);
            }

            // Assign the string representation of 'temporaryString' to 's' for the next iteration
            s = temporaryString.toString();
        }
        // Return the processed string when the length of 's' is less than or equal to 'k'
        return s;
    }
}
```

C++ Solution

```
class Solution {
public:
    // Function to calculate the digit sum of the string `s` with window size `k`
    string digitSum(string s, int k) {
        // As long as the length of the string `s` is greater than `k`
        while (s.size() > k) {
            string temp; // Temporary string to hold the new computed values
            int strSize = s.size(); // Size of the current string `s`

            // Iterate over the string in chunks of size `k`
            for (int i = 0; i < strSize; i += k) {
                int partSum = 0; // Sum for the current part

                // Sum digits within the current window of size `k` or the remaining part
                for (int j = i; j < min(i + k, strSize); ++j) {
                    partSum += s[j] - '0'; // Convert char to int and add to sum
                }

                // Append the calculated part sum to the temporary string
                temp += to_string(partSum);
            }

            // Replace the original string `s` with the newly computed one
            s = temp;
        }
        // Return the possibly transformed string `s`
        return s;
    }
};
```

Typescript Solution

```
/**
 * Calculates the digital sum of a string `s` grouping by `k` digits,
 * until the resulting string is shorter than or equal to `k`.
 *
 * @param {string} s - The initial numeric strings to process.
 * @param {number} k - The group size to sum up.
 * @returns {string} - The final digital sum string.
 */
function digitSum(s: string, k: number): string {
    // This variable will hold intermediate results
    let intermediateResults: number[] = [];

    // Continue processing the string until its length is less than or equal to k
    while (s.length > k) {
        // Iterate over the string in steps of k
        for (let i = 0; i < s.length; i += k) {
            // Extract the current group of characters to process
            let currentGroup = s.slice(i, i + k);

            // Calculate the sum of the digits in the current group and add to the results
            let groupSum = currentGroup.split('').reduce((accumulator, currentChar) => accumulator + parseInt(currentChar), 0);
            intermediateResults.push(groupSum);
        }

        // Join all calculated sums to form a new string
        s = intermediateResults.join('');

        // Reset the intermediate results for the next iteration
        intermediateResults = [];
    }

    // Return the processed string
    return s;
}

// Example:
console.log(digitSum("111111222223", 3)); // Output should be "135"
```

Time and Space Complexity

Time Complexity

The main part of the given code consists of a `while` loop that runs as long as the length of the string `s` is greater than `k`. Within this loop, there is a `for` loop that iterates over the string in chunks of size `k`, which is done at most `ceil(n/k)` times, where `n` is the length of `s`. For each chunk, the numeric conversion and summation of at most `k` digits is performed. The time complexity contribution of this part is linear with respect to the chunk size `k`. Thus, for each iteration, the time complexity is $O(k)$.

However, because `s` is reassigned a new value after each iteration and its length generally decreases at the rate proportional to `k`, the total number of iterations of the while loop would be the number of times `k` divides `n`, in the worst case, which could be $\log_k(n)$ approximately.

Hence, the overall time complexity of this code is $O(n + \log_k(n))$.

Space Complexity

Space complexity is concerned with the additional space the algorithm uses excluding the input. The temporary list `t` and the string `s` updates within the loop are the main contributing factors. In the worst case, the list `t` may contain a number of elements equal to `ceil(n/k)`. Each element in `t` is a string representation of a number that is at most `9k`. Since the size of these numbers is at most $\log10(9k) + 1$, the space to store each chunk is $O(k)$ since $\log10(9k) + 1$ is $O(1)$ for this purpose.

Since `s` is recreated in each iteration with potentially smaller size due to the nature of the problem and not reused outside of the loop, the space complexity is $O(k)$.

Therefore, the final space complexity of the provided code is $O(k)$.