1945. Sum of Digits of String After Convert

Easy String Simulation

Leetcode Link

Problem Description

In this problem, you have a string `s` that is composed of lowercase English letters, and an integer `k`. You are required to perform a series of transformations on the string to convert it into an integer using a specific set of rules, and then reduce it further by a process of digit summation. This reduction process is to be repeated `k` times.

Here is what you need to do step by step:

1. **Convert** the string `s` into an integer by substituting each letter with its corresponding position in the English alphabet, where 'a' is replaced by 1, 'b' by 2, and so on until 'z' which is replaced by 26.
2. **Transform** the resulting integer by summing its digits to create a new, smaller integer.
3. Repeat the **transform** operation a total of `k` times.

Your goal is to return the final integer after all the transformations are complete.

For example, given the string `s = "zbax"` and `k = 2`, the process would look like this:

- **Convert**: "zbax" becomes "262124" because 'z'=26, 'b'=2, 'a'=1, and 'x'=24.
- **Transform #1**: 262124 becomes 17 since 2+6+2+1+2+4 = 17.
- **Transform #2**: 17 becomes 8 since 1+7 = 8.

At the end of these operations, you would return the integer 8.

Intuition

The solution approach breaks down as follows:

1. **String to Integer Conversion**:
   - First, understand that each letter corresponds to a numeric value based on its position in the alphabet.
   - Iterate over each character in the string and convert it to the respective numeric value by using the built-in `ord()` function to get the ASCII value and then subtracting the ASCII value of the character 'a'. This will give a number between 0 and 25, to which we then add 1 to get the correct position in the alphabet (1-26).
   - As we convert the characters, concatenate their corresponding numeric strings to form a new string which represents the large integer.
2. **Sum of Digits Transformation**:
   - Now, we need to transform the integer by summing its digits. We do this by iterating over the string representation of the integer, converting each character back to an integer, and summing them.
   - This sum is then converted back into a string so that we can repeat the process.
3. **Repeat Transformation**:
   - The summation transformation is repeated `k` times as specified by the problem. Each time, the result of the previous transformation is used as the input for the next.

Implementing this step-by-step approach, the string is first fully converted, and then progressively reduced through digit summation until we've applied the transformation `k` times, resulting in the final integer.

Solution Approach

The solution is implemented in Python with a straightforward approach. Let's dive deep into the algorithm, step by step:

1. **String to Integer Conversion**:
   - The solution starts with an iteration over the input string `s` using a generator expression. It performs the conversion of each letter in the string to its corresponding position in the alphabet.
   - To achieve this, the `ord()` function is used, which takes a character and returns its ASCII encoding. Since `ord('a')` returns 97 (the ASCII value of 'a'), to map 'a' to 1, 'b' to 2, ..., 'z' to 26, we subtract `ord('a') - 1` from `ord(c)`, where `c` is the current character in the string.
   - Each mapped integer is converted to a string and joined together without delimiters to form the large integer as a string, which is assigned back to `s`.
2. **Sum of Digits Transformation**:
   - A loop is utilized to perform the transformation `k` times. Inside the loop, a temporary variable `t` is employed to accumulate the sum of digits of `s`.
   - The summation uses a list comprehension that iterates through each character in the string `s`, converts it to an integer, and sums all these integers.
   - After the summation, `s` is updated to the string representation of `t`, preparing it for the next iteration (if any).
3. **Repeat Transformation**:
   - The loop ensures that the sum of digits transformation is done `k` times as per the problem requirements.
   - After the loop completes `k` iterations, `s` will have been transformed into a string representation of the final integer, at which point we can return it as an integer.
4. **Return the Result**:
   - The loop naturally ends after the `k` iterations, and then the final string `s`, which is now representative of the last digit sum, is converted into an integer using `int()` and returned as the result.

The entire implementation doesn't require any complex data structures or algorithms as it is a straightforward computational process. The use of Python's built-in functions like `sum()`, `ord()`, and comprehensions makes the code concise and efficient. The algorithm has a time complexity of O(nk), where n is the length of the input string, because it iterates over the string to convert it to a numeric form, and then over the digits `k` times to compute the sum. The space complexity is O(n) due to holding the numeric representation of the string as well as the intermediate sums during the transformation process.

Example Walkthrough

Let's go through an example to illustrate the solution approach.

Assume we are given the string `s = "abc"` and `k = 3`. Our task is to transform `s` into an integer following the given rules and then reduce it `k` times by digit summation.

1. **String to Integer Conversion**:

   First, we convert each character of `s` into its corresponding position in the alphabet.

   - 'a' becomes "1" since 'a' is the first letter of the alphabet.
   - 'b' becomes "2" since 'b' is the second letter.
   - 'c' becomes "3" since 'c' is the third letter.

   After conversion, the string `s` becomes "123".

2. **Sum of Digits Transformation**:

   Now we start the process of summing the digits of the string "123".

   - **Transform #1**: We sum the digits: 1 + 2 + 3 = 6. We then set `s` to "6" (the sum of the digits).
   - **Transform #2**: We sum the digits: 6 (since `s` is just "6" at this point). Nothing changes because `s` only contains one digit, so we set `s` to "6" again.
   - **Transform #3**: Again, we sum the digits: 6. Since `s` is already "6", it remains unchanged.

3. **Repeat Transformation**:

   We were required to perform the digit sum transformation a total of `k` (3) times, and now we have completed this process.

4. **Return the Result**:

   After completing the `k` transformations, the final value of `s` is "6". We convert this string to an integer using `int()` and return the result.

The returned integer after all transformations is 6. The process correctly reduces the initial string "abc" through an integer representation into a single digit representing the sum of its parts, repeated `k` times.

Python Solution

```python
class Solution:
    def getLucky(self, s: str, k: int) -> int:
        # Convert the string into its corresponding numerical string representation
        # where 'a' -> 1, 'b' -> 2, ..., 'z' -> 26.
        numeric_string = ''.join(str(ord(char) - ord('a') + 1) for char in s)

        # Repeat the transformation process k times.
        for _ in range(k):
            # Calculate the sum of digits in the numeric string.
            sum_of_digits = sum(int(digit) for digit in numeric_string)
            # Convert the sum back to string for the next iteration.
            numeric_string = str(sum_of_digits)

        # Return the final integer after k transformations.
        return int(numeric_string)
```

Java Solution

```java
class Solution {

    // Function to convert a string into a "lucky" number based on the specified rules and transformations.
    public int getLucky(String s, int k) {
        // StringBuilder to create the initial numerical string representation
        StringBuilder numericalRepresentation = new StringBuilder();

        // Convert each character in the string to its corresponding numerical value ('a' -> 1, 'b' -> 2, etc.)
        for (char c : s.toCharArray()) {
            // Append the numerical string equivalent of character to the StringBuilder
            numericalRepresentation.append(c - 'a' + 1);
        }

        // Store the numerical string representation to s for further manipulation
        s = numericalRepresentation.toString();

        // Perform transformation k times
        while (k-- > 0) {
            int sum = 0; // Initialize sum to accumulate the digits

            // Sum the digits of the string
            for (char c : s.toCharArray()) {
                sum += c - '0'; // Convert char digit to integer and add to sum
            }

            // Convert the calculated sum back to string for the next iteration
            s = String.valueOf(sum);
        }

        // Convert the final string back to an integer and return it as the "lucky" number
        return Integer.parseInt(s);
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Converts a string s to its "transformed" value as per the problem statement
    // and performs the transformation k times.
    int getLucky(string s, int k) {
        string transformed; // Holds the numeric representation of the string 's'

        // Convert each character in string 's' to its corresponding number
        // ('a' -> 1, 'b' -> 2, ..., 'z' -> 26
        for (char c : s) {
            transformed += to_string(c - 'a' + 1);
        }

        // Perform k transformations
        while (k--) {
            int sumOfDigits = 0; // Placeholder for the sum of digits

            // Calculate the sum of the digits in the transformed string
            for (char digitChar : transformed) {
                sumOfDigits += digitChar - '0'; // Convert char to int and add to sum
            }

            // Update the transformed string to the new value
            transformed = to_string(sumOfDigits);
        }

        // Finally, convert the transformed string to an integer and return it
        return stoi(transformed);
    }
};
```

Typescript Solution

```typescript
/**
 * Converts a string into a transformed integer by a specific process.
 * @param {string} s - The input string consisting of lowercase English letters.
 * @param {number} k - The number of transformations to perform.
 * @returns {number} - The resulting integer after k transformations.
 */
function getLucky(s: string, k: number): number {
    // Initialize an empty string to store the numeric representation.
    let numericRepresentation = '';

    // Convert each character to its alphabetic position and concatenate to numericRepresentation.
    for (const character of s) {
        numericRepresentation += character.charCodeAt(0) - 'a'.charCodeAt(0) + 1;
    }

    // Perform the transformation k times.
    for (let i = 0; i < k; ++i) {
        let sum = 0; // Initialize the sum for this iteration.

        // Calculate the sum of the digits in the numeric representation.
        for (const digit of numericRepresentation) {
            sum += Number(digit);
        }

        // Convert the sum back to string for the next iteration.
        numericRepresentation = `${sum}`;
    }

    // Convert the final numeric representation to a number and return.
    return Number(numericRepresentation);
}
```

Time and Space Complexity

Time Complexity

The time complexity of the function involves several parts:

1. Converting each character in the string `s` to its corresponding numerical value and then to a string. This takes `O(n)` time where `n` is the length of the input string `s`.
2. The concatenation of the string representations of these numerical values in the worst case can create a string of length `O(n)` because each character in the string `s` can at most add 2 digits to the new string (`'z'` being the 26th letter results in '26').
3. Repeatedly summing the digits for `k` times. In each iteration, the length of the string `s` decreases roughly by a factor of 9 since we are summing the digits (except the first iteration which will operate on the concatenated string). Therefore, each subsequent summing operation will potentially work on a shorter string. The overall time for the summing part can be approximated by `O(k * m)` where `m` represents the length of the numerical string after the first transformation.

Combining these, the total time complexity is `O(n + k * m)`.

Space Complexity

The space complexity is primarily affected by:

1. The space required to store the numerical string equivalent to the input string which could be `O(n)` in the worst case.
2. The space required to store the intermediate sum results, which is `O(m)` where `m` is the length of the numerical string after the first transformation.

Since `m` is at most `2n`, and the numerical string is overwritten in each iteration with a potentially shorter one, the total space complexity is `O(n)`.

Note: The value of `m` here refers to the length of the initial input string, and the value of `m` is dynamic but starts off as at most `2n`.