1880. Check if Word Equals Summation of Two Words

Easy String

Problem Description

The problem presents a scenario in which each letter from 'a' to 'j' has an associated numeric value based on its position in the English alphabet, starting with 'a' as 0 through 'j' as 9. This conversion applies a positional numbering system where each character has a place value that is a power of 10 based on its position in the string (similar to decimal numbers). Given three lowercase strings `firstWord`, `secondWord`, and `targetWord`, the task is to determine whether the sum of the numeric values of `firstWord` and `secondWord` is equal to the numeric value of `targetWord`.

To illustrate:

- Suppose `firstWord` is "abc" which converts to numerical value as "012" ⇒ 12 in integer.
- Suppose `secondWord` is "de" which converts to numerical value as "34" ⇒ 34 in integer.
- Suppose `targetWord` is "fg" which converts to numerical value as "56" ⇒ 56 in integer.

One would then check if 12 (`firstWord`) + 34 (`secondWord`) equals 56 (`targetWord`), and based on this, return `true` or `false`.

Intuition

The solution hinges on converting each string into its corresponding numeric value and then comparing the sum of the numeric values of `firstWord` and `secondWord` with the numeric value of `targetWord`. This involves understanding the positional value of each letter, akin to how positional value works in numerical digits.

By defining a function `f(s)` which converts a given string `s` into its numeric equivalent, we can simplify the problem into three conversions followed by a numeric comparison. The function takes the following steps for each character in the string:

- It subtracts the ASCII value of 'a' from the ASCII value of the character to find the numeric value of the letter (as 'a' maps to 0, 'b' maps to 1, and so on).
- It multiplies the current result by 10 (since we're moving one place to the left in a positional number system) and adds the numeric value of the letter.

Solution Approach

The solution applies a simple algorithmic approach that parses each character of the input strings and converts it to a numerical value based on its position in the alphabet. No complex data structures are needed — just a fundamental loop and some basic arithmetic operations. Here's an analysis of the steps:

1. The helper function `f(s)` is designed to process a single string `s` and convert it into its corresponding numeric value.
   - We initialize `res` to 0, which will serve as an accumulator for the resultant numeric value.
   - For every character `c` in the string `s`, the function converts it from a letter to a number by using the expression `(ord(c) - ord('a'))`. Here, `ord(c)` gives the ASCII value of character `c` and `ord('a')` gives the ASCII value of the letter 'a'. Subtracting the latter from the former yields a number from 0 to 9, corresponding to the letters 'a' to 'j'.
   - The accumulator `res` is then updated by multiplying it by 10 (to shift its digit one place to the left) and adding the numeric value of the current character. This effectively "appends" the numeric character value to the result, mimicking the concatenation of numbers.
2. After defining the conversion function, the main function `isSumEqual` applies this helper function to each of the three input words: `firstWord`, `secondWord`, and `targetWord`.
   - We then sum the results of `firstWord` and `secondWord`, and compare it with the result of `targetWord`.
   - If the sum is equal to the numeric value of `targetWord`, the function returns `True`. Otherwise, it returns `False`.

The solution doesn't use any complex patterns or algorithms—it's a direct application of basic programming constructs such as loops and conditions along with ASCII operations to perform the task. The elegance of the solution lies in its simplicity and the realization that string manipulation can be dealt with fundamental arithmetic operations and character encoding principles.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach with three strings `firstWord`, `secondWord`, and `targetWord`.

Suppose we have:

- `firstWord` as "acd" which should convert to numerical value as "023" ⇒ 23 in integer.
- `secondWord` as "ba" which should convert to numerical value as "10" ⇒ 10 in integer.
- `targetWord` as "cde" which should convert to numerical value as "234" ⇒ 234 in integer.

Using the algorithm described in the solution approach, let's apply these steps:

1. Convert `firstWord` ("acd") to its numeric equivalent:
   - 'a' is the 0th letter, so the current result is 0.
   - 'c' is the 2nd letter, so we take the current result of 0, multiply by 10 (giving 0), and add 2 to get 2.
   - 'd' is the 3rd letter, so we take the current result of 2, multiply by 10 (giving 20), and add 3 to get 23.

   The numeric value of `firstWord` is hence 23.
2. Convert `secondWord` ("ba") to its numeric equivalent:
   - 'b' is the 1st letter, so the current result is 1.
   - 'a' is the 0th letter, so we take the current result of 1, multiply by 10 (giving 10), and add 0 to get 10.

   The numeric value of `secondWord` is hence 10.
3. Convert `targetWord` ("cde") to its numeric equivalent:
   - 'c' is the 2nd letter, so the current result is 2.
   - 'd' is the 3rd letter, so we take the current result of 2, multiply by 10 (giving 20), and add 3 to get 23.
   - 'e' is the 4th letter, so we take the current result of 23, multiply by 10 (giving 230), and add 4 to get 234.

   The numeric value of `targetWord` is hence 234.
4. Sum the results of `firstWord` and `secondWord`:
   - The sum of 23 (`firstWord`) and 10 (`secondWord`) is 33.

Finally, we compare this sum with the numeric value of `targetWord`:

- The sum we have is 33, which is not equal to the numeric value of `targetWord`, 234.

Therefore, the function would return `False` for this example since 33 does not equal 234. This illustrates the solution approach of converting the alphabetic string into a numerical value and then adding to compare with a third numeric string value.

Solution Implementation

Python

```python
class Solution:
    def isSumEqual(self, first_word: str, second_word: str, target_word: str) -> bool:
        # Helper function to convert a string to a numerical value
        # based on the position of each character in the alphabet.
        def convert_to_number(s: str) -> int:
            result = 0
            for char in s:
                # Subtract 'a' from the char to get its position in the alphabet
                # where 'a' = 0, 'b' = 1, ..., 'j' = 9, and then shift the result
                # left by one decimal place (multiply by 10) for each new character.
                result = result * 10 + (ord(char) - ord('a'))
            return result

        # Check if the sum of the numerical values for first_word and second_word
        # is equal to the numerical value of target word.
        return convert_to_number(first_word) + convert_to_number(second_word) == convert_to_number(target_word)
```

Java

```java
class Solution {
    // Method to determine if the numeric value of targetWord equals the sum of numeric values of firstWord and secondWord
    public boolean isSumEqual(String firstWord, String secondWord, String targetWord) {
        // Utilizing helper method 'convertToNumericValue' to get numeric values and checking for equality
        return convertToNumericValue(firstWord) + convertToNumericValue(secondWord) == convertToNumericValue(targetWord);
    }

    // Helper method to convert a string where each character represents a digit from 'a' = 0 to 'j' = 9, into its numeric value
    private int convertToNumericValue(String word) {
        int numericValue = 0; // Initialize result to zero
        // Iterate through the characters in the string
        for (char character : word.toCharArray()) {
            // Shift existing numericValue one decimal place and add the numeric equivalent of the character
            numericValue = numericValue * 10 + (character - 'a');
        }
        // Return the total numeric value of the string
        return numericValue;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to check if the sum of numerical values of two words is equal to the numerical value of a target word
    bool isSumEqual(string firstWord, string secondWord, string targetWord) {
        // Check if the sum of the numerical equivalents of firstWord and secondWord is equal to that of targetWord
        return convertToNumber(firstWord) + convertToNumber(secondWord) == convertToNumber(targetWord);
    }

    // Helper function to convert a string word to its numerical equivalent based on the problem's rule
    // 'a' corresponds to 0, 'b' to 1, ..., 'j' to 9.
    int convertToNumber(string word) {
        int result = 0; // Initialize result to hold the numerical value of the word
        // Iterate through each character of the string
        for (char ch : word) {
            // Multiply the current result by 10 and add the numerical equivalent of character 'ch'
            result = result * 10 + (ch - 'a');
        }
        return result; // Return the final numerical value of the word
    }
};
```

TypeScript

```typescript
function isSumEqual(firstWord: string, secondWord: string, targetWord: string): boolean {
    // Calculates the numeric value of a given string based on the problem's rules,
    // where 'a' corresponds to 0, 'b' to 1, ... 'j' to 9.
    const calculateStringValue = (word: string) => {
        let value = 0; // Initialize the numeric value as 0.

        // Loop through each character in the string.
        for (const char of word) {
            value = value * 10 + (char.charCodeAt(0) - 'a'.charCodeAt(0));
        }

        // Return the final numeric value of the string.
        return value;
    };

    // Compare the sum of the numeric values of the first two words to the numeric value of the target word.
    return calculateStringValue(firstWord) + calculateStringValue(secondWord) === calculateStringValue(targetWord);
}
```

```python
class Solution:
    def isSumEqual(self, first_word: str, second_word: str, target_word: str) -> bool:
        # Helper function to convert a string to a numerical value
        # based on the position of each character in the alphabet.
        def convert_to_number(s: str) -> int:
            result = 0
            for char in s:
                # Subtract 'a' from the char to get its position in the alphabet
                # where 'a' = 0, 'b' = 1, ..., 'j' = 9, and then shift the result
                # left by one decimal place (multiply by 10) for each new character.
                result = result * 10 + (ord(char) - ord('a'))
            return result

        # Check if the sum of the numerical values for first_word and second_word
        # is equal to the numerical value of target word.
        return convert_to_number(first_word) + convert_to_number(second_word) == convert_to_number(target_word)
```

Time and Space Complexity

The given Python code defines a method `isSumEqual` that checks if the numerical value of the sum of two words is equal to the numerical value of a third word, considering the words are treated as base-10 numbers where 'a' corresponds to 0, 'b' to 1, up to 'j' corresponding to 9.

The time complexity of the function `f` is `O(N)`, where `N` is the length of the input string `s`. This is because for each character in the string, it performs a constant number of operations (calculating the difference between the character code and the code of 'a', then multiplying the previous result by 10, and adding the numerical value of the character).

The time complexity of the overall `isSumEqual` function is determined by the lengths of the input strings. If we denote the lengths of `firstWord`, `secondWord`, and `targetWord` as `N1`, `N2`, and `N3` respectively, then the total runtime is `O(N1 + N2 + N3)`, since each word is processed individually by the function `f`.

As for the space complexity, the auxiliary space used by the function `f` is `O(1)`, since it only uses a fixed number of integer variables, regardless of the input size. This means the space complexity of the `isSumEqual` function is also `O(1)` because it only calls `f` for each of the words without storing any additional information that grows with the input size.