848. Shifting Letters

Medium Array String Leetcode Link

Problem Description

The problem requires us to perform a series of shifts on a string `s` composed of lowercase English letters, based on the integers contained in a parallel array `shifts`. For each letter in `s`, we define the operation `shift()` which takes a letter to the next one in the alphabetical order, wrapping around from 'z' to 'a'. For every element `shifts[i]`, representing a number `x`, we apply the shift operation `x` times to the first `i + 1` letters of the string `s`. Our goal is to apply all these shifts accordingly and return the modified string.

Intuition

The solution capitalizes on the observation that the effect of shifting letters is cumulative. That is, a shift on the first letter affects all subsequent shifts. Instead of applying each shift one by one to all applicable letters—which would be time-consuming—, we can compute the total shifts required for each letter starting from the end of the string.

Here's a step-by-step breakdown of the intuition:

1. Recognize that if we shift the first letter of the string, it affects the shift operations for all other letters in the string. For example, if the first letter should be shifted by 5, then all subsequent letters should also account for this shift.

2. Utilize the fact that shifts wrap around the alphabet, which means we can use modulus (`%`) to simplify the shifts. Since there are 26 letters in the English alphabet, any shift count will only have a unique effect within a range of 26 (e.g., shifting 27 times is the same as shifting once).

3. Start from the end of the string and work backwards. By doing so, we can maintain a running total of shifts needed as we move towards the start, adding the current shift value to our total. Each letter's shift is then a simple combination of its original position and the cumulative shifts applied to it.

4. Apply the total shift value to each letter by converting the letter to its corresponding alphabet index (0 for 'a', 1 for 'b', etc.), adding the shift value, and then taking the result modulo 26 to handle wrap-around.

5. Build the resulting string by converting the shifted indices back into characters, appending the result to form the final string.

The provided solution code implements this approach efficiently, resulting in a final string that reflects all the required shifts.

Solution Approach

The solution approach for the problem uses a reverse iteration through the `shifts` array, and the python `ord()` and `ascii_lowercase` functions for character manipulation. Here's a detailed step-by-step explanation:

1. **Initialization**: We start by converting the input string `s` into a list of individual characters to allow easy manipulation since strings in Python are immutable. We also define a variable `t` to keep track of the cumulative shift.

2. **Reverse Iteration**: We loop through the `shifts` array in reverse order, using `range(n - 1, -1, -1)`. This allows us to accumulate shifts starting from the end of the string, ensuring that each letter is shifted the correct number of times as per the problem definition.

3. **Cumulative Shift**: In each iteration, we update `t` by adding the current shift value `shifts[i]`. This effectively creates a cumulative shift count `t` that is applied to every character at index `i` and to all characters before it.

4. **Character Shifting**: For each character at index `i`, we calculate the new character after shifting:

   - First, by finding the current character's position in the alphabet `ord(s[i]) - ord('a')`
   - Then, we add the total shift value `t` to this position.
   - We use modulo 26 `( ... ) % 26` to ensure that if the shift takes us past 'z', we wrap around back to the start of the alphabet.
   - We find the new character using `ascii_lowercase[j]` where `j` is the result of the modulo operation. This provides us with the appropriate shifted character.
5. **Building the Result**: We update each character in our list with its corresponding shifted character.

6. **Returning the Final String**: Finally, we join the list of characters into a string using `''.join(s)` and return this as the final result.

Here is the code snippet that encapsulates the solution approach:

```python
class Solution:
    def shiftingLetters(self, s: str, shifts: List[int]) -> str:
        n, t = len(s), 0
        s = list(s)
        for i in range(n - 1, -1, -1):
            t += shifts[i]
            j = (ord(s[i]) - ord('a') + t) % 26
            s[i] = ascii_lowercase[j]
        return ''.join(s)
```

This code carefully combines the character shifting with cumulative totals and the alphanumeric positioning of each letter to deliver an optimal and effective solution with linear-time complexity.

Example Walkthrough

Let's walk through a simple example using the solution approach described.

Given:

- String `s = "abc"`
- Shifts `shifts = [3, 5, 9]`

The goal is to shift each letter in the string based on the corresponding value in the `shifts` array, applying this value as a shift to the first `i + 1` letters in the string.

Here's how we apply the solution approach:

1. **Initialization**: Convert string `s` into a list `['a', 'b', 'c']` to allow for easy manipulation.

2. **Reverse Iteration**: Loop through `shifts` in reverse order: indices `2, 1, 0`.

3. **Cumulative Shift**:

   - For index `2`, `t += shifts[2]`, so `t = 9`.
   - For index `1`, `t += shifts[1]`, so `t = 9 + 5 = 14`.
   - For index `0`, `t += shifts[0]`, so `t = 14 + 3 = 17`.
4. **Character Shifting**:

   - At index `2 (c)`: The shift is `9`. New character index: `(2 + 9) % 26 = 11`. The new character is `ascii_lowercase[11] = 'l'`.
   - At index `1 (b)`: The cumulative shift is `14`. New character index: `(1 + 14) % 26 = 15`. The new character is `ascii_lowercase[15] = 'p'`.
   - At index `0 (a)`: The cumulative shift is `17`. New character index: `(0 + 17) % 26 = 17`. The new character is `ascii_lowercase[17] = 'r'`.
5. **Building the Result**: Update the list with shifted characters to get `['r', 'p', 'l']`.

6. **Returning the Final String**: Join the list to form the final string `"rpl"`.

The resulted shifted string after applying all shifts is `"rpl"`.

This example demonstrates how applying the cumulative shift from the end of the string reduces the time complexity of the algorithm, as each shift value is added once to the total, and each letter is shifted individually only once.

Python Solution

```python
from string import ascii_lowercase

class Solution:
    def shiftingLetters(self, s: str, shifts: List[int]) -> str:
        # Initialize necessary variables
        # Get the length of the string
        string_length = len(s)
        # Total cumulative shifts count
        cumulative_shift = 0
        # Convert string to a list of characters to allow modification
        str_list = list(s)

        # Process each character from the end to the beginning
        for i in range(string_length - 1, -1, -1):
            # Increase cumulative shift by the current value
            cumulative_shift += shifts[i]
            # Calculate new character position by adding cumulative_shifts
            # Take modulus by 26 to find the correct position after 'z'
            new_char_index = (ord(str_list[i]) - ord('a') + cumulative_shift) % 26
            # Update the character in the list using the new index
            str_list[i] = ascii_lowercase[new_char_index]

        # Join the list of characters into a string
        # and return the resulting string
        return ''.join(str_list)

# The changes made include the following:
# - Import ascii_lowercase to be used later in the code.
# - Rename variables to be more descriptive.
# - Add comments to explain each step of the code for clarity.

```

Java Solution

```java
class Solution {
    // This method shifts the letters of a given string based on the values in the 'shifts' array
    public String shiftingLetters(String s, int[] shifts) {
        // Convert the string to a character array for in-place manipulation
        char[] characters = s.toCharArray();
        // Get the length of the character array
        int length = characters.length;
        // 'totalShifts' will accumulate the amount of shift to be applied
        long totalShifts = 0;

        // Iterate over the characters from the end to the beginning
        for (int i = length - 1; i >= 0; --i) {
            // Cumulative addition of shifts for the current character
            totalShifts += shifts[i];
            // Calculate the new position for the current character after shift
            int newPosition = (int) ((characters[i] - 'a' + totalShifts) % 26);
            // Update the character in the array to its new shifted character
            characters[i] = (char) ('a' + newPosition);
        }

        // Return the new string created from the shifted character array
        return new String(characters);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to shift letters in the string based on the shifts vector.
    string shiftingLetters(string s, vector<int>& shifts) {
        long long totalShifts = 0;  // Initialize total shifts to 0
        int stringLength = s.size(); // Get the size of the string

        // Loop through the string starting from the end
        for (int i = stringLength - 1; i >= 0; --i) {
            totalShifts += shifts[i];  // Add shifts for current position to total shifts
            totalShifts %= 26;         // Avoid overflow and keep within valid alphabet indexes

            // Calculate the new offset for current letter based on the total shifts
            int newLetterIndex = (s[i] - 'a' + totalShifts) % 26;
            s[i] = 'a' + newLetterIndex;  // Set the new letter in the string
        }

        return s;  // Return the modified string after shifts
    }
};

```

Typescript Solution

```typescript
// Function to shift letters in a string based on the shifts array.
function shiftingLetters(s: string, shifts: number[]): string {
    let totalShifts: number = 0; // Initialize total shifts to 0
    const stringLength: number = s.length; // Get the length of the string
    let shiftedString: string[] = s.split(''); // Convert the string to an array for easy modification

    // Loop through the string starting from the end
    for (let i: number = stringLength - 1; i >= 0; --i) {
        totalShifts += shifts[i]; // Add the shift for the current position to the total shifts
        totalShifts %= 26; // Avoid overflow and keep within valid alphabet indexes

        // Calculate the new offset for the current letter based on the total shifts
        let newLetterIndex: number = (shiftedString[i].charCodeAt(0) - 'a'.charCodeAt(0) + totalShifts) % 26;
        shiftedString[i] = String.fromCharCode('a'.charCodeAt(0) + newLetterIndex); // Update the letter in the array
    }

    return shiftedString.join(''); // Join the array back into a string and return it
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the string `s`. This is because the function contains a single loop that iterates backward through the length of the string and `shifts` list only once. Inside the loop, it performs constant time operations such as addition, modulo operation, and index access, which do not depend on the size of `n`.

Space Complexity

The space complexity of the code is `O(n)`, mainly because the string `s` is converted to a list which will contain `n` characters. Temporary variables `t` and `j` use constant space, and the incremental updates within the loop do not increase space usage with respect to the input size. Thus, the space used is proportional to the length of the input string.