6. Zigzag Conversion

Medium String

Leetcode Link

Problem Description

The task is to rearrange the characters of a given string `s` in a zigzag pattern on a specified number of rows and then read off the characters line by line to create a new string. The zigzag pattern means that the characters of `s` are placed diagonally in a zigzag manner in downwards and upwards directions successively. After placing all the characters in this pattern, we would read the characters in each row horizontally and concatenate to form the final string.

To visualize the process, you can think of writing down the characters in the following way:

- Starting from the top row, write characters downwards.
- Upon reaching the last row, switch direction and start writing characters upwards, forming a diagonal line until you reach the top row again.
- Alternate the process until every character is written in this zigzag fashion.

For example, given a string `"PAYPALISHIRING"` and 3 rows, the characters should be placed like this:

```
1 P   A   H   N
2 A P L S I I G
3 Y   I   R
```

After arranging the characters, the string `"PAHNAPLSIIGYIR"` is obtained by reading each row sequentially. To form this new string programmatically, the code should simulate the reading process.

Intuition

The intuitive approach to simulate the zigzag conversion process involves creating an imaginary 2D grid that represents rows and columns of the pattern. A `group` is defined which denotes the cycle length of the zigzag pattern, calculated as `2 * numRows - 2`. If `numRows` is 1, then the zigzag pattern is not possible, and the original string is returned.

The solution uses a loop to construct the new string by considering each row individually, assembling the characters that would appear in that row in the zigzag pattern. The `interval` between characters of the same row varies depending on their position (at the top or bottom, it's equal to one cycle; in between, it alternates between moving vertically down and obliquely up) and is used to calculate the index of the next character to be added.

1. Initialize an empty list `ans` to hold characters in the order they should be added to form the zigzag string.
2. Loop through each row. For the top and bottom rows, characters occur after every full cycle (the `group`), so the interval is constant. For intermediate rows, the interval alternates. We calculate the initial `interval` based on the current row.
3. Within each row, loop to add characters to `ans`, incrementing the index by the `interval` each time. After each addition, update the interval to alternate between moving vertically and diagonally, accounting for edge cases to avoid a zero interval.
4. After processing all rows, join the list of characters `ans` into a string to get the final result.

The core of the solution relies on finding the pattern of intervals between the positions within the zigzag alignment and using it to construct the final string efficiently without simulating the entire grid.

Solution Approach

The implementation for the given problem follows a direct approach that calculates the next index in the original string for each character in the transformed zigzag string. It does this by determining the appropriate index intervals without needing to simulate the entire 2D zigzag structure.

Here's how the implementation works:

1. **Early Return for Single Row**: If `numRows` is 1, the zigzag pattern is trivial and the original string can be returned immediately, as there is no alteration in the order of characters.

```
if numRows == 1:
    return s
```

2. **Calculating the Group Cycle**: The group cycle (`group`) is calculated as the number of characters that form the complete vertical and diagonal cycle in the zigzag pattern, given by `2 * numRows - 2`. This number is central to the implementation as it helps to understand the symmetry in the pattern, which repeats every `group` characters.

```
group = 2 * numRows - 2
```

3. **Initializing the Answer List**: An empty list `ans` is initialized to store the characters in their new order.
4. **Populating Each Row**: The algorithm iterates over each row and identifies the characters that would appear in that row in a zigzag pattern:
   - Determine the initial `interval` for characters in the current row. For the first and last rows, it remains constant and equal to `group`. For intermediate rows, the `interval` alternates between two values.
   - Use a `while` loop to continue adding characters to the `ans` list until the end of the string is reached.

```
for i in range(1, numRows + 1):
    interval = group if i == numRows else 2 * numRows - 2 * i
    idx = i - 1
    while idx < len(s):
        ans.append(s[idx])
        idx += interval
        interval = group - interval if interval != group else interval
```

   Here, `i` is the row index, `idx` is the index in the original string for the current character, and the `interval` determines the number of steps to the next character in the current row.

   - For intermediate rows (rows between the first and last), after each character is added, the interval for the next character is updated to the complementary interval (the difference between the `group` and the current interval), thus simulating the zigzag pattern without explicitly constructing a 2D grid structure.
5. **Concatenating the Results**: After the rows are processed and all characters are placed in the `ans` list, the list is joined into a string to form the final zigzag transformation.

```
return ''.join(ans)
```

This implementation optimizes space by avoiding the creation of a 2D grid and computes the string in O(n) time by analyzing the inherent pattern in the positions of characters in the zigzag arrangement.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the string `"HELLO"` and the number of rows `numRows` is 3. The task is to arrange the characters in a zigzag pattern and then read them line by line.

Step 1: Early Return for Single Row

Since `numRows` is not 1, we don't return the string as is, and we proceed with the rest of the algorithm.

Step 2: Calculating the Group Cycle

We calculate the group cycle (`group = 2 * numRows - 2`) which in this case is `2 * 3 - 2 = 4`. This means that the vertical and diagonal pattern will repeat every 4 characters.

Step 3: Initializing the Answer List

We initialize an empty `ans` list where we will store the characters.

Step 4: Populating Each Row

We iterate over each row and add the relevant characters to the `ans` list:

- For the first row (`i = 1`), the interval will remain constant at 4. The first character is `'H'` at index 0. The next character in the zigzag pattern for the first row would be 4 characters away, but since our string is shorter, we stop here for this row. So we add `'H'`.
- For the second row (`i = 2`), the interval starts at `2 * 3 - 2 * 2 = 2` and will alternate between 2 and (4 - 2) = 2 for subsequent characters. The first character for this row is `'E'` at index 1. The next character in the zigzag pattern for the second row is 2 characters away, which is `'L'` at index 3. There are no more characters 2 or 4 steps away, so we stop for this row. We add `'E'` and `'L'`.
- For the third row (`i = 3`), the interval is constant at 4, the same as for the first row. The first character is `'L'` at index 2, and there are no more characters 4 steps away. So, we add `'L'`.

Step 5: Concatenating the Results

We concatenate the characters in the `ans` list to form the final string which is `'HEL'`.

Following the steps above, the zigzag arrangement for the string `"HELLO"` with 3 rows would look like this:

```
1 H   .
2 . E .
3 . L .
4 . . L
5 . . O
```

When reading off each line, the new string formed is `"HEL"`, which matches the characters collected in our answer list.

This walkthrough of a smaller example demonstrates how the designed algorithm would simulate the placement of characters in the zigzag pattern and construct the transformed string accordingly.

Python Solution

```
class Solution:
    def convert(self, s: str, numRows: int) -> str:
        # If there's only one row or the string is shorter than the number of rows,
        # it means the pattern will be the same as the input string
        if numRows == 1 or numRows >= len(s):
            return s

        # Create an array to hold the rows
        rows = [''] * numRows
        # This variable controls the direction the "zigzag" is going.
        # It starts as 1, meaning "downwards", and will change to -1 to go "upwards".
        step = 1
        # Start from the first row
        curr_row = 0

        # Iterate over each character in the string
        for char in s:
            # Append the current character to the current row
            rows[curr_row] += char
            # If we're at the top or bottom row, switch direction
            if curr_row == 0 or curr_row == numRows - 1:
                step = -step
            # Move to the next row in the current direction
            curr_row += step

        # Concatenate all rows to form the final string
        return ''.join(rows)

# Example usage:
# solution = Solution()
# converted_string = solution.convert("PAYPALISHIRING", 3)
# print(converted_string)  # Output should be "PAHNAPLSIIGYIR"
```

Java Solution

```
class Solution {
    public String convert(String inputString, int numRows) {
        // If numRows is 1, no pattern is required, so return the string as it is.
        if (numRows == 1) {
            return inputString;
        }

        // StringBuilder is more efficient when appending characters in a loop.
        StringBuilder convertedStringBuilder = new StringBuilder();
        // Length of the pattern cycle.
        int cycleLength = 2 * numRows - 2;

        // Loop over each row.
        for (int row = 0; row < numRows; ++row) {
            // Calculate the interval for the current row.
            // For the first and last row, it is the cycle length,
            // for the others, it depends on the row number.
            int interval = (row == numRows - 1) ? cycleLength : 2 * (numRows - row - 1);
            // Index to keep track of the position on the string.
            int index = row;

            // Continue looping until the end of the string is reached.
            while (index < inputString.length()) {
                // Append character at index to the result.
                convertedStringBuilder.append(inputString.charAt(index));
                // Proceed to the next character in the current row.
                index += interval;
                // Toggle the interval for the middle rows.
                // This does not affect the first and last rows.
                interval = (interval == cycleLength || interval == 0) ? cycleLength : cycleLength - interval;
            }
        }

        // Convert StringBuilder back to String and return.
        return convertedStringBuilder.toString();
    }
}
```

C++ Solution

```
#include <string>

class Solution {
public:
    /**
     * Converts the input string to a new string arranged in a zigzag pattern on a given number of rows.
     *
     * @param s Input string to be converted.
     * @param numRows The number of rows in which the string will be rearranged into a zigzag pattern.
     * @return A string representing the zigzag pattern.
     */
    std::string convert(std::string s, int numRows) {
        // If there is only one row, then the zigzag pattern is the same as the original string.
        if (numRows == 1) return s;

        std::string ans;  // The final answer string in zigzag order.
        int cycleLength = 2 * numRows - 2;  // The length of the repeating zigzag cycle.

        // Loop through each row.
        for (int currentRow = 1; currentRow <= numRows; ++currentRow) {
            // The interval depends on the current row and alternates within each zigzag cycle.
            int interval = (currentRow == numRows) ? cycleLength : 2 * numRows - 2 * currentRow;
            int currentIndex = currentRow - 1;  // The starting index in the original string for this row.

            // Loop through characters in the row.
            while (currentIndex < s.length()) {
                ans.push_back(s[currentIndex]);  // Append character to the answer string.
                currentIndex += interval;  // Move to the next character in the zigzag pattern.
                // Switch the interval for the next character in the zigzag pattern.
                // Unless it's the first or last row, where the interval remains constant.
                interval = (interval == cycleLength || interval == 0) ? cycleLength : cycleLength - interval;
            }
        }

        return ans;  // Return the zigzag pattern string.
    }
};
```

Typescript Solution

```
function convert(text: string, rowQuantity: number): string {
    // If the numRows is 1, the "zigzag" pattern is the same as the original string
    if (rowQuantity === 1) {
        return text;
    }

    // Create an array of strings to represent each row initialized with empty strings
    const rows = new Array(rowQuantity).fill('');
    let currentRow = 0; // Initialize the current row tracker
    let goingDown = true; // Flag to determine the direction of iteration through the rows

    // Iterate through each character in the input string
    for (const char of text) {
        rows[currentRow] += char; // Append the current character to the current row

        // If we are at the top or bottom row, reverse the direction
        if (currentRow === 0 || currentRow === rowQuantity - 1) {
            goingDown = !goingDown;
        }

        // Move to the next row depending on the direction
        currentRow += goingDown ? 1 : -1;
    }

    // Combine all the rows into a single string and return the result
    return rows.join('');
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`. This is determined by the observation that we iterate over each character of the string `s`, which has length `n`. Every character is visited once during the construction of the `ans` list. The nested while loop does not increase the complexity because the inner loop's increments are such that the total number of iterations remains linearly dependent on the length of the `s`.

Space Complexity

The space complexity of the code is `O(n)`. This is because we use an additional list `ans` to store the rearranged characters. In the worst case, the `ans` list can contain all characters of the input string `s`, which requires space that is linearly proportional to the length of the input string.