2075. Decode the Slanted Ciphertext

Medium String Simulation Leetcode Link

Problem Description

This problem presents a string `originalText` that has been encoded using a "slanted transposition cipher" into a new string `encodedText` with the aid of a matrix with a defined number of rows `rows`. The encoding process consists of placing the characters of `originalText` into a matrix slantingly, from top to bottom and left to right in order of the text. The matrix is filled such that it would not have any empty columns on the right side after placing the entire `originalText`. Any remaining empty spaces in the matrix are filled with spaces ' '. `encodedText` is then obtained by reading the characters out of the matrix row by row and concatenating them.

For instance, if `originalText = "cipher"` and `rows = 3`, the process is visualized as placing "c", "i", "p" in the first row, shifting one position to the right for the second row to start with "h", and then "e" for the row after. Once the characters are placed, they are read off as "ch ie pr" to form `encodedText`. The goal is to reverse this process and recover `originalText` from `encodedText` and the given number of rows `rows`. Note that `originalText` doesn't contain any trailing spaces, and it is guaranteed there is only one `originalText` that corresponds to the input.

Intuition

To decode `encodedText`, we need to reverse-engineer the encoding process. We know the number of rows in which `encodedText` was originally laid out. By dividing the length of `encodedText` by the number of rows, we determine the number of columns that the plaintext was wrapped into. With this matrix's dimensions in mind, we can simulate the reading order the encoded text would have had if it were to be read slantingly.

The solution starts at each column of the top row and proceeds diagonally downward to the rightmost column, mimicking the slanted filling from the encoding process. We continue this diagonal reading for all starting positions in the top row.

To implement this in the solution, a simple loop iterates over each possible starting position (each column of the first row). For each start position, it reads off the characters diagonally until it either reaches the last row or the last column. These characters are appended to the `ans` list, which accumulates the original text. Finally, since `encodedText` could have trailing spaces (due to the padding of the matrix), but `originalText` doesn't, we use the `.rstrip()` function to remove any trailing whitespace from the reconstructed original text.

This process gives us the `originalText` without the need to actually build the matrix, which would be computationally more expensive and memory consuming.

Solution Approach

The implementation of the solution uses a simple yet efficient approach to decode `encodedText`. It avoids constructing the entire matrix and instead calculates the positions of the characters that would be in the original diagonal sequence based on their indices in the encoded string.

Here's how the algorithm unfolds:

1. Determine the number of columns in the encoded matrix by dividing the length of `encodedText` by `rows`, the number of rows (`cols = len(encodedText) // rows`).

2. Iterate over the range of columns to determine the starting point of each diagonal read process (`for j in range(cols)`).

3. For each starting point, initialize variables `x` and `y` to keep track of the current row and column during the diagonal traversal. Initially `x` is set to 0 because we always start from the top, and `y` is set to the current column we are iterating over (`x, y = 0, j`).

4. Start a while loop that continues as long as `x` is less than `rows` and `y` is less than `cols`. These conditions ensure we stay within the bounds of the conceptual matrix.

5. Calculate the linear index of the current character in the encoded string (`encodedText[x * cols + y]`) and append it to the `ans` list. The multiplication `x * cols` skips entire rows to get to the current one, and `y` moves us along to the correct column.

6. Increment both `x` and `y` to move diagonally down to the right in the matrix (`x, y = x + 1, y + 1`). This essentially simulates the row and column shift that occurs in a diagonal traversal.

7. After completing the while loop for a diagonal line, the loop will iterate to the next starting column, and the process repeats until all columns have been used as starting points for the diagonal reads.

8. Once all characters are read diagonally and stored in the `ans` list, combine them into a string (`''.join(ans)`) and strip any trailing spaces (`rstrip()`). This yields the original non-encoded text, which is then returned.

This solution effectively decodes the slanted cipher text without constructing the encoding matrix. It directly accesses the required characters by calculating their original and encoded positions through index arithmetic, which is both space and time-efficient. The choice of using a list to accumulate characters before joining them into a string is due to the fact that string concatenation can be costly in Python due to strings being immutable, while appending to a list and then joining is more efficient.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the problem content provided.

Suppose `encodedText = "ch ie pr"` and `rows = 3`. We want to decode this string to find the original text, supposedly `originalText = "cipher"`.

1. Calculate the number of columns: `len("ch ie pr") // 3 = 8 // 3 = 2`. So, we have 2 columns (ignoring the extra spaces).

2. Iterate over the range of columns. Since we have 2 columns, the loop will run twice, for column indices 0 and 1.

3. Loop iteration for column index 0:

   - Initialize `x = 0`, `y = 0`. This represents the first character of the top row.
   - While `x < 3` and `y < 2`:
     - `x = 0`, `y = 0`, the current character is `encodedText[0 * 2 + 0]`, which is "c".
     - Append "c" to the `ans` list.
     - Increment `x` and `y` to move diagonally, `x = 1`, `y = 1`.
   - Now `x = 1`, `y = 1`, the current character is `encodedText[1 * 2 + 1]`, which is "i".
     - Append "i" to the `ans` list.
     - Increment `x` and `y` to move diagonally, `x = 2`, `y = 2`.
   - Since `y` is not less than `cols`, we have reached the end of the diagonal traversal for this starting point.
4. Loop iteration for column index 1:

   - Initialize `x = 0`, `y = 1`. This represents the second column of the top row.
   - While `x < 3` and `y < 2`:
     - `x = 0`, `y = 1`, the current character is `encodedText[0 * 2 + 1]`, which is "h".
     - Append "h" to the `ans` list.
     - Increment `x` and `y` to move diagonally, `x = 1`, `y = 2`.
   - Since `y` is not less than `cols`, we move to the next row with `x = 1` and reset `y = 0`.
   - Now `x = 1`, `y = 0`, the current character is `encodedText[1 * 2 + 0]`, which is " ".
     - This is a space, but append it to the `ans` list anyway since we need to preserve the sequence.
     - Increment `x` and `y` to move diagonally, `x = 2`, `y = 1`.
   - Now `x = 2`, `y = 1`, the current character is `encodedText[2 * 2 + 1]`, which is "p".
     - Append "p" to the `ans` list.
     - Increment `x` and `y` to move diagonally, `x = 3`, `y = 2`.
   - Since `x` is not less than `rows`, we have reached the end of the traversal for this starting point.
5. The `ans` list now contains `["c", "i", "h", " ", "p"]`.

6. Combine the characters into a string and strip trailing spaces: `'cih p'.rstrip()` gives us `"cihp"`.

7. However, we notice that this is not the correct original text as the decode process puts the space in the wrong position. We should refine our solution by handling the spaces correctly.

So, let's correct the steps to account for the shift that occurs at each row of the slanted transposition:

- Increment `x` and `y` to move diagonally, if `y` reaches the number of columns, reset `y` to 0 and increase `x`: This mimics the wrapping to the next line in the slanted transposition.

Here's the corrected list of characters following the approach and the adjustment:

```
 ch
  i
  e
pr
```

- The diagonal traversal from column 0: "c", "i", "p" (top-to-bottom).
- The diagonal traversal from column 1: "h", "e" (top-to-bottom).

So the correct `ans` list would contain `["c", "i", "p", "h", "e"]`. We join them into a string without needing to strip spaces (since we handled the spaces correctly): `''.join(["c", "i", "p", "h", "e"])` gives us `"cipher"`, which is our desired original text.

Python Solution

```python
class Solution:
    def decode_ciphertext(self, encoded_text: str, rows: int) -> str:
        # Initialize a list to hold the decoded characters
        decoded_characters = []

        # Calculate the number of columns based on the length of the encoded text and number of rows
        cols = len(encoded_text) // rows

        # Iterate over each column index starting from 0
        for col_index in range(cols):
            # Initialize starting point
            row, col = 0, col_index

            # Traverse the encoded text by moving diagonally in the matrix
            # constructed by rows and columns
            while row < rows and col < cols:
                # Determine the linear index for the current position in the (row, col) matrix
                linear_index = row * cols + col

                # Append the corresponding character to the decoded list
                decoded_characters.append(encoded_text[linear_index])

                # Move diagonally: go to next row and next column
                row, col = row + 1, col + 1

        # Join the decoded characters to form the decoded string.
        # Strip trailing spaces if any.
        return ''.join(decoded_characters).rstrip()

```

Java Solution

```java
class Solution {

    // Function to decode the cipher text given the number of rows
    public String decodeCiphertext(String encodedText, int rows) {
        // StringBuilder to build the decoded string
        StringBuilder decodedText = new StringBuilder();

        // Calculate the number of columns based on the length of the encoded text and number of rows
        int columns = encodedText.length() / rows;

        // Loop through the columns; start each diagonal from a new column
        for (int colStart = 0; colStart < columns; ++colStart) {
            // Initialize the row and column pointers for the start of the diagonal
            for (int row = 0, col = colStart; row < rows && col < columns; ++row, ++col) {
                // Calculate the index in the encodedText string and append the character at this index
                decodedText.append(encodedText.charAt(row * columns + col));
            }
        }

        // Remove trailing spaces from the decoded string
        while (decodedText.length() > 0 && decodedText.charAt(decodedText.length() - 1) == ' ') {
            decodedText.deleteCharAt(decodedText.length() - 1);
        }

        // Return the decoded string
        return decodedText.toString();
    }
}

```

C++ Solution

```cpp
#include <string>

class Solution {
public:
    // Decodes the ciphertext from the encoded text given the number of rows of the encoded grid
    std::string decodeCiphertext(std::string encodedText, int rows) {
        // Initialize the decoded string
        std::string decoded;
        // Find out the number of columns based on the size of encoded text and number of rows
        int cols = encodedText.size() / rows;

        // Iterate over each column of the grid
        for (int col = 0; col < cols; ++col) {
            // For every column, traverse diagonally starting from (0, col)
            for (int row = 0, y = col; row < rows && y < cols; ++row, ++y) {
                // Add character at (row, y) to the decoded string
                decoded += encodedText[row * cols + y];
            }
        }

        // Trim any trailing spaces from the decoded string
        while (!decoded.empty() && decoded.back() == ' ') {
            decoded.pop_back();
        }

        // Return the decoded string
        return decoded;
    }
};

```

Typescript Solution

```typescript
function decodeCiphertext(encodedText: string, rows: number): string {
    // Compute the number of columns based on the length of encoded text and the number of rows.
    const columns = Math.ceil(encodedText.length / rows);
    let decodedCharacters: string[] = [];

    // Traverse the encoded text diagonally starting at each column.
    for (let columnOffset = 0; columnOffset <= columns; columnOffset++) {
        // Start from the first row and the current column offset,
        // moving diagonally through the text and adding the characters to the result.
        for (let row = 0, col = columnOffset; row < rows && col < columns; row++, col++) {
            decodedCharacters.push(encodedText.charAt(row * columns + col));
        }
    }

    // Combine the characters to form the decoded string and trim any trailing spaces.
    return decodedCharacters.join('').trimEnd();
}

```

Time and Space Complexity

The time complexity of the code is `O(rows * cols)` since the main computation happens in a nested loop where `x` goes from `0` to `rows - 1`, and `y` goes from `0` to `cols - 1`. In each iteration of the loop, it performs a constant time operation of adding a single character to the `ans` list. Since `rows * cols` is also the length of the `encodedText`, the complexity could also be given as `O(n)` where `n` is the length of `encodedText`.

The space complexity is `O(n)` as well, due to the `ans` list which at most will contain `n` characters (where `n` is the length of `encodedText`). The `.rstrip()` function is called on a `''.join(ans)` which is a string of the same length as `ans`, but since strings are immutable in Python, this operation generates a new string so it doesn't increase the space complexity beyond `O(n)`.