1324. Print Words Vertically

Medium Array String Simulation Leetcode Link

Problem Description

The problem presents a situation where we are given a string `s` which contains a sequence of words separated by spaces. The goal is to return all these words "vertically" in the order they appear in the string. Returning the words vertically means that each word will start at a new column, and the letters from different words at the same column index will be combined into a new string (still preserving the order in which words appear). If the words are of different lengths and we reach a column index where some words have no characters, we use spaces to fill those places. It is also important to note that we should not have any trailing spaces in our final strings.

For example, if `s` is `"HOW ARE YOU"`, the output should be `["HAY","ORO","WEU"]` as each column gives `"HAY"`, `"ORO"`, and `"WEU"` respectively without the trailing spaces.

Intuition

To solve this problem, we first split the input string into words. Then we need to determine the maximum length among these words because this will tell us how many "vertical" strings we need to form — one for each character position in the longest word.

Next, we iterate over each character position up to the maximum length, and for each position, we build a new vertical word by taking the character from each original word at that position if it exists, or using a space as a placeholder if that word is too short.

While building each vertical word, we should also ensure that we trim trailing spaces. This is crucial so that the resulting vertical words do not end with any unnecessary spaces.

This solution focuses on solving the problem step by step, considering each vertical word as a snapshot of each column in the original words. It's a straightforward approach that uses simple loops and list operations to accomplish the task.

Solution Approach

The implementation of the solution follows a clear and structured algorithmic approach.

1. **Splitting the String:** The first step involves splitting the string `s` into words. This is done using the `.split()` method in Python, which by default splits a string by spaces, thereby separating individual words into a list.

```
words = s.split()
```

2. **Determining the Maximum Word Length:** Once we have a list of words, we find the length of the longest word using a generator expression inside the `max` function. This step is crucial as it determines how many vertical strings we need to construct (one for each character of the longest word).

```
n = max(len(w) for w in words)
```

3. **Creating Vertical Words:** The solution then iterates through each character position (using a range of `n`, the maximum length found). For each position `j`, we construct a temporary list `t`, where each element is the `j`-th character of a word from the original list if that word is long enough; otherwise, it's a space `' '`.

```
for j in range(n):
    t = [w[j] if j < len(w) else ' ' for w in words]
```

This is achieved using a list comprehension that also applies conditional logic - an efficient way to construct lists based on conditional operations in Python.

4. **Trimming Trailing Spaces:** After constructing each vertical word, the algorithm trims any trailing spaces from the list `t`. It does this by checking and popping the last element repeatedly until the last character is not a space.

```
while t[-1] == ' ':
    t.pop()
```

5. **Building the Final Answer:** Finally, all the characters in the list `t` are joined to make a string representing a vertical word, which is then appended to the answer list `ans`.

```
ans.append(''.join(t))
```

The algorithm completes when it has created a vertical word for each position in the maximum word length, and the answer list `ans` is returned containing the correctly formatted vertical representation of the given string. This approach leverages simple data structures, namely lists and strings, combined with straightforward logic for an intuitive solution.

Overall, this solution approach uses common Python data manipulation techniques to transform the input string into the desired vertical orientation step by step. It employs fundamental programming concepts such as loops, list comprehension, conditional statements, and string manipulation to achieve the result.

Example Walkthrough

Let's walk through the solution approach with a small example. Suppose our input string is `"TO BE OR NOT TO BE"`.

1. **Splitting the String:** First, we split the input string `"TO BE OR NOT TO BE"` into individual words.

```
words = "TO BE OR NOT TO BE".split()  # ["TO", "BE", "OR", "NOT", "TO", "BE"]
```

2. **Determining the Maximum Word Length:** We find the longest word's length, which determines the number of vertical strings we need to construct.

```
n = max(len(w) for w in words)  # The longest word is "NOT", so n = 3.
```

3. **Creating Vertical Words:** We iterate over each character position (0 to 2, since the longest word has 3 characters) and build temporary lists for each position, adding spaces if a word is shorter than the current index.

```
# First iteration (j = 0)
t = [w[0] if 0 < len(w) else ' ' for w in words]  # ["T", "B", "O", "N", "T", "B"]

# Second iteration (j = 1)
t = [w[1] if 1 < len(w) else ' ' for w in words]  # ["O", "E", "R", "O", "O", "E"]

# Third iteration (j = 2)
t = [w[2] if 2 < len(w) else ' ' for w in words]  # [" ", " ", " ", "T", " ", " "]
```

4. **Trimming Trailing Spaces:** We make sure to remove any trailing spaces from our temporary lists after each iteration.

```
# In the third iteration, the list `t` is [" ", " ", " ", "T", " ", " "]
while t[-1] == ' ':
    t.pop()  # After trimming, `t` becomes [" ", " ", " ", "T"]
```

5. **Building the Final Answer:** Each trimmed list `t` is then converted to a string and added to our answer list `ans`.

```
# After the first iteration:
ans = ["TBONTB"]

# After the second iteration:
ans.append(''.join(t))  # ["TBONTB", "ERONTOE"]

# After the third iteration:
ans.append(''.join(t))  # ["TBONTB", "ERONTOE", "   T"]
```

After completing these steps for each character position, the answer list `ans` will look like this: `["TBONTB", "ERONTOE", " T"]`. The last step is to make sure no trailing spaces are in the final strings:

```
final_ans = [s.rstrip() for s in ans]  # ["TBONTB", "ERONTOE", "T"]
```

The final output is `["TBONTB", "ERONTOE", "T"]`. In this example, each vertical string corresponds to each column of characters from top to bottom, accurately representing the vertical words without trailing spaces as required.

Python Solution

```python
class Solution:
    def printVertically(self, s: str) -> List[str]:
        # Split the string into words.
        words = s.split()
        # Find the length of the longest word to determine the number of rows.
        max_length = max(len(word) for word in words)

        # Create a list to hold the vertical print result.
        vertical_print = []

        # Iterate over the range of the maximum length found.
        for i in range(max_length):
            # Collect the i-th character of each word if it exists,
            # otherwise use a space.
            column_chars = [(word[i] if i < len(word) else ' ') for word in words]

            # Trim trailing spaces from the right side.
            while column_chars and column_chars[-1] == ' ':
                column_chars.pop()

            # Join the characters to form the vertical word and
            # append it to the result list.
            vertical_print.append(''.join(column_chars))

        # Return the list of vertical words.
        return vertical_print

```

Java Solution

```java
import java.util.ArrayList;
import java.util.List;

public class Solution {

    // Function to print words of a string in a vertical order
    public List<String> printVertically(String s) {

        // Split the input string into words
        String[] words = s.split(" ");

        // The variable 'maxWordLength' will hold the length of the longest word
        int maxWordLength = 0;

        // Find the longest word to determine the number of rows in the output
        for (String word : words) {
            maxWordLength = Math.max(maxWordLength, word.length());
        }

        // Initialize a list to store the resulting vertical strings
        List<String> result = new ArrayList<>();

        // Loop through each character index up to the length of the longest word
        for (int j = 0; j < maxWordLength; ++j) {

            // Use StringBuilder for efficient string concatenation
            StringBuilder currentLineBuilder = new StringBuilder();

            // Loop through each word and append the character at current index,
            // or append a space if the word is not long enough
            for (String word : words) {
                currentLineBuilder.append(j < word.length() ? word.charAt(j) : ' ');
            }

            // Remove trailing spaces from the current line
            while (currentLineBuilder.length() > 0 &&
                   currentLineBuilder.charAt(currentLineBuilder.length() - 1) == ' ') {
                currentLineBuilder.deleteCharAt(currentLineBuilder.length() - 1);
            }

            // Add the trimmed line to the result list
            result.add(currentLineBuilder.toString());
        }

        // Return the list of vertical strings
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <sstream>
#include <algorithm>

class Solution {
public:
    // Function to print words of a string vertically
    std::vector<std::string> printVertically(std::string s) {
        // Initialize stringstream for parsing words
        std::stringstream stream(s);

        // Container for storing individual words
        std::vector<std::string> words;

        // Placeholder for current word extraction
        std::string word;

        // Maximum length of words
        int maxLength = 0;
        while (stream >> word) { // Extract words one by one
            words.emplace_back(word); // Add current word to words vector
            maxLength = std::max(maxLength, static_cast<int>(word.size())); // Update maxLength if current word is longer
        }

        // Container for the answer
        std::vector<std::string> result;

        // Loop to form words for vertical printing
        for (int columnIndex = 0; columnIndex < maxLength; ++columnIndex) {
            std::string verticalWord; // String to hold each vertical word

            // Forming each vertical word by taking character at the current column index
            for (auto& currentWord : words) {
                // Add the character if the column index is less than the word length, otherwise, add a space
                verticalWord += columnIndex < currentWord.size() ? currentWord[columnIndex] : ' ';
            }

            // Trim the trailing spaces in the vertical word
            while (!verticalWord.empty() && verticalWord.back() == ' ') {
                verticalWord.pop_back(); // Remove the last character if it is a space
            }

            // Add the trimmed vertical word to the result
            result.emplace_back(verticalWord);
        }

        // Return the vector containing the vertically printed words
        return result;
    }
};

```

Typescript Solution

```typescript
// Import statements for TypeScript (if needed)
// Notably, TypeScript does not have a direct equivalent of C++'s <sstream>, <vector>, or <algorithm>
// No import is needed here since TypeScript has built-in support for arrays and strings.

// Function to print words of a string vertically
function printVertically(s: string): string[] {
    // Split the input string into words based on spaces
    const words: string[] = s.split(' ');

    // Find the maximum length of the words
    const maxLength: number = Math.max(...words.map(word => word.length));

    // Initialize an array to hold the results
    const result: string[] = [];

    // Loop through each column index (0 to maxLength - 1)
    for (let columnIndex = 0; columnIndex < maxLength; columnIndex++) {
        // Variable to store the current vertical word
        let verticalWord: string = '';

        // Loop through each word to form one vertically
        for (const currentWord of words) {
            // Add the character at the current index or a space if the word is too short
            verticalWord += columnIndex < currentWord.length ? currentWord.charAt(columnIndex) : ' ';
        }

        // Trim the trailing spaces from the current vertical word
        verticalWord = verticalWord.replace(/\s+$/, '');

        // Add the trimmed vertical word to the result array
        result.push(verticalWord);
    }

    // Return the formatted vertical words array
    return result;
}

```

Time and Space Complexity

The time complexity of the code can be determined by analyzing the two main operations in the function: splitting the input string into words and forming the vertical print.

1. Splitting the input string into words takes `O(m)` time, where `m` is the length of the input string `s`, since the split operation goes through the string once.

2. The main loop runs `n` times, where `n` is the maximum length of the words obtained after the split. Inside this loop, forming the temporary list `t` takes `O(k)` time for each iteration, where `k` is the number of words. The `while` loop inside may run up to `k` times in the worst-case scenario, which is when all but the first word are shorter than the current index `j`.

Combining these two points, the overall time complexity is `O(m + n*k^2)`. However, typically the while loop is expected to perform fewer operations as it stops once a non-space character is found from the end. Therefore, the general expected time complexity is `O(m + n*k)`.

The space complexity is determined by the space required to store the `words`, the `ans` list, and the temporary list `t`. The `words` list will store `k` words occupying `O(m)` space (since all words together cannot be longer than the input string `s`), and the `ans` list will store at most `n * k` characters, which accounts for `O(n*k)` space. The temporary list `t` requires `O(k)` space.

Considering these factors, the overall space complexity is `O(m + n*k)` (since `m` could be less than or equal to `n*k` and both need to be considered).