1768. Merge Strings Alternately

Easy | Two Pointers | String

Leetcode Link

Problem Description

In this problem, you are given two input strings, `word1` and `word2`. Your task is to combine these two strings into a new string, by alternating characters from each. You start merging from `word1`, then `word2`, and repeat this process until you use all characters from both strings. If one string is longer than the other, you simply continue adding the remaining characters from the longer string to the end of the merged string. The goal is to return the merged string as the result.

Intuition

The intuition behind the solution involves iterating through both strings concurrently and pairing up characters to the new string. A key point is determining what to do when the strings are of different lengths. One approach is to use the `zip` function that pairs elements from both strings. However, `zip` stops when the shortest input is exhausted. To overcome this, we can use the `zip_longest` function from Python's `itertools` module, which allows pairing beyond the length of the shorter word by filling in missing values with a specified value (`fillvalue=''` in this case, which represents an empty string).

The function `zip_longest(word1, word2, fillvalue='')` pairs corresponding characters from each string and fills in with an empty string when either `word1` or `word2` runs out of characters. We join these pairs using a generator expression that concatenates each pair of characters into a single string, accounting for the possibility of one character being an empty string when the other string has extra characters. We then use `''.join()` on this sequence to merge all pairs into the final merged string.

Solution Approach

The solution leverages Python's `itertools.zip_longest` function to iterate over both input strings `word1` and `word2` in parallel. This function can handle input sequences of different lengths by filling in missing values with a specified value (`''` in this case).

Here's a step-by-step explanation of the algorithm:

1. We call the `zip_longest` function with `word1`, `word2`, and `fillvalue=''` to create an iterator that pairs elements from both `word1` and `word2`. If one string is shorter than the other, `zip_longest` will use the `fillvalue` to ensure that the iterator can still return pairs of values until the longer string is fully processed.
2. We use a generator expression `(a + b for a, b in zip_longest(word1, word2, fillvalue=''))` which goes through each pair produced by `zip_longest` and concatenates the elements of each pair. This handles the alternating of characters from `word1` and `word2` and takes care of any additional characters from the longer string.
3. Finally, `''.join()` is called on the generator expression to concatenate all string pairs into a single string without any separators. This results in the merged string that comprises characters from `word1` and `word2` in alternating order, fulfilling the requirements laid out in the problem description.

This solution elegantly utilizes the `zip_longest` to abstract away the complexity of dealing with different string lengths and allows the process to be expressed in a single, readable, and efficient line of code.

Example Walkthrough

Let's consider two example input strings for our walk-through:

- `word1 = "Hello"`
- `word2 = "World"`

Applying the solution approach:

1. We call `zip_longest(word1, word2, fillvalue='')`, which pairs `H` with `W`, `e` with `o`, `l` with `r`, `l` with `l`, and `o` with `d`. Since `word1` and `word2` are of equal length, `fillvalue=''` is not utilized here.
2. As we go through the generator expression `(a + b for a, b in zip_longest(word1, word2, fillvalue=''))`, it yields `'HW'`, `'eo'`, `'lr'`, `'ll'`, and `'od'`.
3. By calling `''.join()` on the generator expression, we concatenate all these string pairs to form the final merged string: `'HW' + 'eo' + 'lr' + 'll' + 'od'` which results in `"HWeorlld"`, achieving an alternating character string using elements from both `word1` and `word2`.

This example clearly demonstrates how each step of the process works in unison to solve the problem, even when the strings are of equal length, and the method remains robust for strings of differing lengths as well.

Python Solution

```python
from itertools import zip_longest  # Importing zip_longest from itertools for handling uneven iterables

class Solution:
    def merge_alternately(self, word1: str, word2: str) -> str:
        # This method merges two strings alternately, filling in with an empty string if one is shorter

        # Use itertools' zip_longest to pair up characters from both strings. If one string is shorter, fill the missing values with
        # Join the paired characters into a single string with list comprehension and join method.
        merged_string = ''.join(a + b for a, b in zip_longest(word1, word2, fillvalue=''))

        return merged_string  # Return the resulting merged string

```

Java Solution

```java
class Solution {
    // Method to merge two strings alternately.
    public String mergeAlternately(String word1, String word2) {
        // Length of the first and second words
        int lengthWord1 = word1.length(), lengthWord2 = word2.length();
        // StringBuilder to create the result string efficiently
        StringBuilder mergedString = new StringBuilder();

        // Iterate as long as we have characters remaining in at least one string
        for (int index = 0; index < lengthWord1 || index < lengthWord2; ++index) {
            // If the current index is within the bounds of word1, append its character
            if (index < lengthWord1) {
                mergedString.append(word1.charAt(index));
            }
            // If the current index is within the bounds of word2, append its character
            if (index < lengthWord2) {
                mergedString.append(word2.charAt(index));
            }
        }
        // Return the resulting string
        return mergedString.toString();
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to merge two strings alternately
    string mergeAlternately(string word1, string word2) {
        int length1 = word1.size(); // length of the first word
        int length2 = word2.size(); // length of the second word
        string result; // string to store the result of merging

        // Loop through the maximum length of the two strings
        for (int i = 0; i < max(length1, length2); ++i) {
            // If the current index is less than the length of word1, append the character to result
            if (i < length1) result += word1[i];
            // If the current index is less than the length of word2, append the character to result
            if (i < length2) result += word2[i];
        }

        return result; // Return the merged string
    }
};

```

Typescript Solution

```typescript
function mergeAlternately(word1: string, word2: string): string {
    // Initialize an array to hold the results of the merge.
    const mergedArray: string[] = [];

    // Determine the longer length from both words.
    const maxLength: number = Math.max(word1.length, word2.length);

    // Loop through each character up to the maximum length.
    for (let index = 0; index < maxLength; index++) {
        // If the current index is within the range of word1, push the character into mergedArray.
        if (index < word1.length) {
            mergedArray.push(word1[index]);
        }
        // If the current index is within the range of word2, push the character into mergedArray.
        if (index < word2.length) {
            mergedArray.push(word2[index]);
        }
    }

    // Join the array elements into a string and return the result.
    return mergedArray.join('');
}

```

Time and Space Complexity

The time complexity of the code is `O(max(M, N))`, where `M` is the length of `word1` and `N` is the length of `word2`. This complexity arises because the code iterates through both strings simultaneously until the end of the longer string is reached, making use of `zip_longest`. At each step of iteration, it performs a constant amount of work by concatenating a single character or an empty string (represented by `fillvalue=''` when one of the inputs is exhausted).

The space complexity is also `O(max(M, N))`, primarily due to the output string. The size of the output string will be the sum of the sizes of `word1` and `word2`. In Python, strings are immutable, so each concatenation creates a new string. Since we are using a generator expression with the `join` method, the space for the intermediate tuples generated by `zip_longest` is negligible, as they're created and discarded. However, the final string that is returned will have a length equivalent to the sum of the lengths of both input strings, thus leading to the `O(max(M, N))` space complexity.