1754. Largest Merge Of Two Strings

Medium Greedy Two Pointers String

Leetcode Link

Problem Description

The problem presents a task where we need to merge two given strings `word1` and `word2` into one new string `merge`. The goal is to create the lexicographically largest string possible. The process of merging is defined by repeatedly taking the first character from either `word1` or `word2` and appending it to `merge`, then removing that character from the string it was taken from. The lexicographically largest string means that if you sort all possible `merge` strings, the one we want would appear last. It should be constructed in such a way that at every choice, if possible, the character that will make `merge` lexicographically larger should be chosen.

We're asked to implement a function that, given two strings `word1` and `word2`, returns the lexicographically largest `merge` string that can be constructed from them.

Intuition

The intuition behind the solution is to always pick the lexicographically larger character to append to the `merge` string. However, simply comparing the characters at the current positions in `word1` and `word2` is not enough. We should look ahead because picking a character from one string might lead to a suboptimal result if the subsequent characters in the other string would create a lexicographically larger string.

The solution approach is to compare the substrings starting from the current characters of `word1` and `word2`, not just the characters themselves. This comparison tells us which string leads to a lexicographically larger outcome if we were to take all remaining characters from it. Whenever the substring of `word1` from the current index `i` is greater than the substring of `word2` from the current index `j`, we append the character from `word1` to `merge`, and vice versa.

We use a while loop to conduct this process repeatedly until one of the strings is empty. Once one of the strings is empty, there are no more decisions to be made—we simply append the remaining characters of the non-empty string to `merge`. The Python `>` operator is used for the comparison, which conveniently compares strings lexicographically. The `.join()` method is then used to combine the list of characters into a single string before returning it as the solution.

Solution Approach

The implemented solution uses two pointers, `i` and `j`, which start at 0 corresponding to the first characters in `word1` and `word2` respectively. An empty list named `ans` is initialized to collect the characters that will form the `merge` string.

The main algorithm is composed of a while loop, which runs as long as there are characters left in both `word1` and `word2`. Within this loop, the key operation is to compare the substrings of `word1` starting from `i` and `word2` starting from `j`. This is done with the expression `word1[i:] > word2[j:]`.

- If `word1[i:]` is lexicographically larger than `word2[j:]`, the first character of `word1` at index `i` is appended to the `ans` list using `ans.append(word1[i])`, and the pointer `i` is incremented by 1 with `i += 1`.
- If `word2[j:]` is lexicographically larger or equal to `word1[i:]`, the first character of `word2` at index `j` is appended to the `ans` list using `ans.append(word2[j])`, and the pointer `j` is incremented by 1 with `j += 1`.

Once the while loop exits (meaning at least one of the strings is exhausted), the remaining characters from both strings (if any) are appended to the `ans` list using `ans.append(word1[i:])` and `ans.append(word2[j:])`. These operations effectively concatenate the leftover substring to the `merge` string.

Finally, the `merge` string is constructed by joining the characters in the `ans` list with the `"".join(ans)` expression, which combines all elements of the list into a single string. The resulting string is then returned as the largest lexicographical `merge` that can be constructed from `word1` and `word2`.

This solution makes use of simple data structures (strings and lists) and an algorithm that optimally decides which character to append to `merge` at each step, ensuring the lexicographically largest result.

Example Walkthrough

Let's assume we are given the following input strings:

```
word1 = "ace"
word2 = "bdf"
```

Our goal is to merge `word1` and `word2` into the lexicographically largest string possible as per the solution approach described. Here is a step-by-step walkthrough of how the algorithm will work with these inputs:

1. Initialize pointers `i` and `j` both to `0` and an empty list `ans` to collect characters.
2. Compare `word1[0:]` ("ace") with `word2[0:]` ("bdf").
3. Since `'ace' < 'bdf'` lexicographically, we append the first character of `word2` to `ans` (`['b']`), and increment `j` to `1`.
4. Now compare `word1[0:]` ("ace") with `word2[1:]` ("df").
5. `'ace' < 'df'` lexicographically, so append the first character of `word2` at index `j` to `ans` (`['b', 'd']`), and increment `j` to `2`.
6. Now compare `word1[0:]` ("ace") with `word2[2:]` ("f").
7. `'ace' > 'f'` lexicographically, so append the first character of `word1` (`'a'`) to `ans` (`['b', 'd', 'a']`), and increment `i` to `1`.
8. Compare `word1[1:]` ("ce") with `word2[2:]` ("f").
9. `'ce' > 'f'` lexicographically, append the first character of `word1` at index `i` to `ans` (`['b', 'd', 'a', 'c']`), and increment `i` to `2`.
10. Compare `word1[2:]` ("e") with `word2[2:]` ("f").
11. `'e' < 'f'` lexicographically, append the character of `word2` at index `j` to `ans` (`['b', 'd', 'a', 'c', 'f']`), and increment `j` to `3`.
12. `word2` is now empty, so we append the remaining characters of `word1` to `ans`.
13. Adding `word1[2:]` ("e") to `ans` gives us `['b', 'd', 'a', 'c', 'f', 'e']`.
14. Join the characters in `ans` with `"".join(ans)` to get the final merged string.

The resulting `merge` string is "bdacfe", which is the lexicographically largest string constructible from the input `word1` and `word2`.

Python Solution

```python
class Solution:
    def largestMerge(self, word1: str, word2: str) -> str:
        index1 = index2 = 0  # Initialize pointers for word1 and word2
        merged = []  # Initialize the list to store the result

        # Loop until the end of one of the words is reached
        while index1 < len(word1) and index2 < len(word2):
            # Compare the suffix starting from the current indices of both words
            if word1[index1:] > word2[index2:]:
                # If word1 has lexicographically greater suffix, add its current character to merged
                merged.append(word1[index1])
                index1 += 1  # Move to the next character in word1
            else:
                # Otherwise, add word2's current character to merged
                merged.append(word2[index2])
                index2 += 1  # Move to the next character in word2

        # Append the remaining part of word1 if there's any left
        merged.append(word1[index1:])
        # Append the remaining part of word2 if there's any left
        merged.append(word2[index2:])

        # Join all pieces into a single string and return
        return "".join(merged)

```

Java Solution

```java
class Solution {
    // Method to find the largest merge of two strings.
    public String largestMerge(String word1, String word2) {
        int lengthWord1 = word1.length(), lengthWord2 = word2.length(); // Lengths of both words
        int indexWord1 = 0, indexWord2 = 0; // Pointers to the current characters in word1 and word2
        StringBuilder largestMerge = new StringBuilder(); // Builder for the result string

        // Iterate until one of the strings is fully added to the merge
        while (indexWord1 < lengthWord1 && indexWord2 < lengthWord2) {
            // Compare the suffixes starting from current pointers of word1 and word2
            boolean greaterThan = word1.substring(indexWord1).compareTo(word2.substring(indexWord2)) > 0;

            // Append the character from the word which has the 'greater' current suffix
            // And increment the pointer for that word
            if (greaterThan) {
                largestMerge.append(word1.charAt(indexWord1++));
            } else {
                largestMerge.append(word2.charAt(indexWord2++));
            }
        }

        // Append the remaining parts of word1 and word2, if any.
        largestMerge.append(word1.substring(indexWord1));
        largestMerge.append(word2.substring(indexWord2));

        return largestMerge.toString(); // Return the largest merge
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to create the largest merge of two strings
    string largestMerge(string word1, string word2) {
        int lengthWord1 = word1.size(); // Length of word1
        int lengthWord2 = word2.size(); // Length of word2
        int indexWord1 = 0; // Index for traversing word1
        int indexWord2 = 0; // Index for traversing word2
        string mergedString; // String to store the result

        // Loop until one of the strings is fully traversed
        while (indexWord1 < lengthWord1 && indexWord2 < lengthWord2) {
            // Determine if the substring of word1 starting from current index
            // is greater than that of word2
            bool isWord1Greater = word1.substr(indexWord1) > word2.substr(indexWord2);

            // If word1's substring is greater, append the next character
            // from word1, else append the next character from word2.
            mergedString += isWord1Greater ? word1[indexWord1++] : word2[indexWord2++];
        }

        // If there are remaining characters in word1, append them to mergedString
        mergedString += word1.substr(indexWord1);

        // If there are remaining characters in word2, append them to mergedString
        mergedString += word2.substr(indexWord2);

        // Return the final merged string
        return mergedString;
    }
};

```

Typescript Solution

```typescript
// Function to merge two strings into the largest lexicographical order.
function largestMerge(word1: string, word2: string): string {
    const word1Length = word1.length; // Length of the first word
    const word2Length = word2.length; // Length of the second word
    let mergedString = ''; // Variable to store the merged string result
    let indexWord1 = 0; // Index pointer for word1
    let indexWord2 = 0; // Index pointer for word2

    // Main loop to construct the merged string
    while (indexWord1 < word1Length && indexWord2 < word2Length) {
        // Compare the substrings starting from current index positions
        // Append the greater (lexicographically) character to the merged string result
        mergedString += word1.slice(indexWord1) > word2.slice(indexWord2)
                        ? word1[indexWord1++]
                        : word2[indexWord2++];
    }

    // Append the remaining substring from word1 if any
    mergedString += word1.slice(indexWord1);
    // Append the remaining substring from word2 if any
    mergedString += word2.slice(indexWord2);

    // Return the final merged string
    return mergedString;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be analyzed by looking at the operations inside the while loop and the operations that happen after the while loop.

1. The while loop runs until `i < len(word1)` and `j < len(word2)`. At each iteration, it checks the lexicographical order of the suffixes starting at the current indices `i` in `word1` and `j` in `word2`. Comparing the suffixes (`word1[i:] > word2[j:]`) is an `O(m)` operation, where `m` is the length of the longer suffix at each step because in the worst case, comparison could go on till the end of the string.

2. The loop runs up to `len(word1) + len(word2)` times in total since at each iteration at least one character is appended to `ans`.

Therefore, the worst-case time complexity is `O((len(word1) + len(word2)) * m)`, where `m` is the length of the longer suffix at each step.

However, if we consider that string comparison in Python is done lexicographically and character by character, `m` will be the smaller of the two suffix lengths at each comparison point.

So a tighter bound considering the average lengths as average sizes of the compared suffixes, the time complexity would be `O(n * k)`, where `n` is `len(word1) + len(word2)` and `k` is the average size of these suffixes during comparison operations.

Space Complexity

The space complexity can be analyzed by considering the extra space used by the algorithm.

1. The list `ans` can grow up to `len(word1) + len(word2)` characters in size.

2. The string slices `word1[i:]` and `word2[j:]`, if implemented naively, could potentially create new strings each iteration, but in Python, slicing strings doesn't create copies, but rather, new references to the existing string's elements. So, these operations are `O(1)` in space.

Therefore, the space complexity is `O(n)`, where `n` is `len(word1) + len(word2)` for the result list that is generated.