2800. Shortest String That Contains Three Strings

Medium Greedy String Enumeration Leetcode Link

Problem Description

The problem requires us to construct a string with the shortest possible length that contains three given strings `a`, `b`, and `c` as its substrings. The goal is to not only find the shortest concatenation but also to ensure that if there are multiple such shortest strings, we return the one that comes first lexicographically (i.e., in dictionary order).

A few key points from the problem statement:

- A substring is a contiguous sequence of characters within another string.
- If two strings differ at some position, the lexicographically smaller one is the string with the character earlier in the alphabet at that position.

Essentially, we are tasked with merging the strings in such a way that they are all present and the resulting string is as short as possible, and among the candidates of equal length, the lexicographically smallest is chosen.

Intuition

The solution relies on finding the best way to overlap the given strings `a`, `b`, and `c` to make the shortest possible string that contains all three.

Here's the intuition behind the solution:

1. Start by considering all permutations of the given strings because the order in which we concatenate them could affect both the final length and lexicographic order.

2. To find the best overlap between two strings, we check if one is entirely contained within the other. If so, we can use the longer string as it already encompasses the other.

3. If neither string contains the other, we look for the longest suffix of one that matches a prefix of the other. This allows us to overlap these parts, reducing the combined length of the concatenated string.

4. We iterate the process, starting with a pair of strings and then combining the result with the third string.

5. We keep track of the overall shortest string and update it for each permutation if the new combination is shorter or equal in length but lexicographically smaller.

The key approach is the function `f(s, t)`, which merges strings `s` and `t`. It either returns one of the input strings if one contains the other or finds the longest overlap possible to merge them efficiently. We use this function iteratively to combine the permutations of input strings and update our result accordingly.

Using the permutation approach ensures that we consider every possible order of the input strings, while the merging function `f` ensures that we combine the strings as efficiently as possible for each permutation.

Solution Approach

The implementation of the solution uses Python's built-in permutation function from the `itertools` module to generate all possible orders in which the strings can be concatenated.

Here's the breakdown of the code:

- The key part of the implementation is the nested function `f(s: str, t: str)` which takes two strings `s` and `t`, and attempts to merge them.
  - It first checks if `s` is a substring of `t` or `t` is a substring of `s`, and if so, returns the larger string.
  - If not, it attempts to find the longest overlap between `s` and `t`. It does this by checking the suffix of `s` against the prefix of `t` starting from the largest possible overlap and decreasing the size until it finds a match or reaches zero.
  - If an overlap is found, it returns the concatenated string excluding the overlapped part in `t` to avoid duplication.
  - If no overlap is found (`i` reaches 0), it returns the concatenation of `s` and `t`.
- The solution function `minimumString(self, a: str, b: str, c: str)` starts with an empty string `ans` to store the final result.
- It then iterates through each permutation of strings `a`, `b`, and `c`, and for each permutation:
  - It uses the merge function `f` to combine the first two strings, and then merges the result with the third string.
  - It compares the resulting combined string with the current `ans`. If `ans` is empty, or if the new string is shorter, or if it's the same length but lexicographically smaller, `ans` is updated to this new string.
- The algorithm uses iterative comparison and string manipulation to build the most efficient string (shortest and lexicographically smallest) for every permutation of input strings.
- Finally, after all permutations are considered and the best merged strings are computed, the function returns the optimal string stored in `ans`.

By using permutations and a merge function, the algorithm efficiently combines the input strings while always ensuring that the output is the minimum length required to contain all three substrings. It also guarantees that if there are strings of the same minimal length, the lexicographically smallest is chosen.

Example Walkthrough

Let's apply the solution approach to a small example with strings `a`, `b`, and `c`. Suppose the given strings are:

- `a = "abc"`
- `b = "bcx"`
- `c = "cxy"`

Following the solution approach:

Step 1: Consider All Permutations

We'll consider all permutations of `a`, `b`, and `c`:

- Permutation 1: "abc", "bcx", "cxy"
- Permutation 2: "abc", "cxy", "bcx"
- Permutation 3: "bcx", "abc", "cxy"
- Permutation 4: "bcx", "cxy", "abc"
- Permutation 5: "cxy", "abc", "bcx"
- Permutation 6: "cxy", "bcx", "abc"

Step 2: Find Best Overlap between Two Strings

Using the function `f(s, t)` for merging strings, we aim to find the best merge:

Taking the first permutation "abc", "bcx", "cxy" as an example:

- Merge "abc" and "bcx" using `f`: "abc" already contains "bc", so we take the longer string "abcx".
- Now merge "abcx" with "cxy": The "cx" part is overlapping, so the result is "abcxy".

Step 3: Iterate the Process for All Other Permutations

Repeat step 2 for all permutations to find the merged string for each.

Step 4: Keep Track of the Overall Shortest and Lexicographically Smallest String

Let's assume that after computing all permutations, we got another shorter string from a different permutation:

- From permutation 4: Merging "bcx" with "cxy" gives "bcxy", and then merging with "abc" gives "abcxy".

In this case, although "abcxy" is not shorter than the one from permutation 1, it is not lexicographically smaller either, so no change is made.

Step 5: Return the Optimal String

After computing all permutations, the shortest string that contains all substrings `a`, `b`, and `c` and is also the first lexicographically is "abcxy".

The optimal string "abcxy" is finally returned as the result.

Python Solution

```python
from itertools import permutations

class Solution:
    def minimumString(self, string_a: str, string_b: str, string_c: str) -> str:
        def merge_strings(s1: str, s2: str) -> str:
            # Check if one string is a subsequence of the other and return the longer one.
            if s1 in s2:
                return s2
            if s2 in s1:
                return s1

            # Identify the longest common suffix of s1 and prefix of s2
            # Then merge s1 and s2 by overlapping this common part
            len_s1, len_s2 = len(s1), len(s2)
            for overlap_length in range(min(len_s1, len_s2), 0, -1):
                if s1[-overlap_length:] == s2[:overlap_length]:
                    return s1 + s2[overlap_length:]

            # If there is no overlap, concatenate the strings
            return s1 + s2

        # Initialize the answer with an empty string.
        shortest_merged_string = ""

        # Check all permutations of the input strings to find the shortest merged string
        for permutation in permutations((string_a, string_b, string_c)):
            merged_string = merge_strings(merge_strings(permutation[0], permutation[1]), permutation[2])
            if not shortest_merged_string or len(merged_string) < len(shortest_merged_string) or \
               (len(merged_string) == len(shortest_merged_string) and merged_string < shortest_merged_string):
                shortest_merged_string = merged_string

        # Return the shortest string found
        return shortest_merged_string

```

Java Solution

```java
class Solution {
    // Method to find the minimum string composed by overlapping or concatenating a, b, and c
    public String minimumString(String a, String b, String c) {
        // Store strings in an array for easy access
        String[] strings = {a, b, c};

        // All possible permutations of indices 0, 1, and 2
        int[][] permutations = {
            {0, 1, 2}, {0, 2, 1},
            {1, 0, 2}, {1, 2, 0},
            {2, 1, 0}, {2, 0, 1}
        };

        // Initialize answer string to be empty at the start
        String answer = "";

        // Iterate over all permutations to find the smallest possible overlap string
        for (int[] perm : permutations) {
            // Access indices based on the current permutation
            int i = perm[0], j = perm[1], k = perm[2];

            // Overlap the first two strings, then overlap the result with the third
            String temp = overlap(overlap(strings[i], strings[j]), strings[k]);

            // Check if the current string `temp` is smaller or lexicographically smaller than `answer`
            if ("".equals(answer) || temp.length() < answer.length()
                    || (temp.length() == answer.length() && temp.compareTo(answer) < 0)) {
                answer = temp; // Update `answer` with the new minimum string `temp`
            }
        }

        return answer; // Return the smallest string after all permutations are checked
    }

    // Helper method to find the overlap between two strings or concatenate if no overlap exists
    private String overlap(String s, String t) {
        // If one string contains the other, return the larger string
        if (s.contains(t)) {
            return s;
        }
        if (t.contains(s)) {
            return t;
        }

        int m = s.length(), n = t.length();

        // Find the maximum overlap starting from the end of `s` and the beginning of `t`
        for (int i = Math.min(m, n); i > 0; --i) {
            // If the end of `s` matches the beginning of `t`, concatenate the non-overlapping part
            if (s.substring(m - i).equals(t.substring(0, i))) {
                return s + t.substring(i);
            }
        }

        // If there's no overlap, concatenate the strings `s` and `t`
        return s + t;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the minimum string obtained by concatenating
    // three input strings in any order without overlapping.
    string minimumString(string stringA, string stringB, string stringC) {
        // The input strings are stored in a vector for easy permutation.
        vector<string> strings = {stringA, stringB, stringC};
        // All possible permutations of the three strings.
        vector<vector<int>> permutations = {
            {0, 1, 2}, {0, 2, 1}, {1, 0, 2}, {1, 2, 0}, {2, 0, 1}, {2, 1, 0}
        };
        // String to store the minimum concatenated result.
        string minimumResult = "";
        // Loop over all the permutations.
        for (auto& permutation : permutations) {
            // Indices for the permutation
            int first = permutation[0], second = permutation[1], third = permutation[2];
            // Concatenate strings based on the current permutation and minimize them.
            string temp = minimize(strings[first], minimize(strings[second], strings[third]));
            // Update the minimumResult if a smaller string is found.
            if (minimumResult.empty() || temp.size() < minimumResult.size() ||
                (temp.size() == minimumResult.size() && temp < minimumResult)) {
                minimumResult = temp;
            }
        }
        return minimumResult;
    }

    // Helper function to minimize two strings by concatenating
    // without overlapping the maximum number of characters.
    string minimize(string firstString, string secondString) {
        // If one string contains the other, return the larger one.
        if (firstString.find(secondString) != string::npos) {
            return firstString;
        }
        if (secondString.find(firstString) != string::npos) {
            return secondString;
        }
        // Get the lengths of the two strings.
        int lengthFirst = firstString.size(), lengthSecond = secondString.size();
        // Try to find the largest overlap starting from the largest possible size.
        for (int overlapSize = min(lengthFirst, lengthSecond); overlapSize > 0; --overlapSize) {
            if (firstString.substr(lengthFirst - overlapSize) == secondString.substr(0, overlapSize)) {
                // If an overlap is found, return the concatenation without the overlapping part.
                return firstString + secondString.substr(overlapSize);
            }
        }
        // If no overlap is found, return the concatenation of both strings.
        return firstString + secondString;
    }
};

```

Typescript Solution

```typescript
function minimumString(firstString: string, secondString: string, thirdString: string): string {
    // Function to merge two strings with minimum additional characters
    const mergeMinimum = (stringA: string, stringB: string): string => {
        // If one string includes the other, return the longer one
        if (stringA.includes(stringB)) {
            return stringA;
        }
        if (stringB.includes(stringA)) {
            return stringB;
        }

        // Determine the lengths of both strings
        const lengthA = stringA.length;
        const lengthB = stringB.length;

        // Find the longest suffix of stringA that is a prefix of stringB
        for (let overlapLength = Math.min(lengthA, lengthB); overlapLength > 0; --overlapLength) {
            if (stringA.slice(-overlapLength) === stringB.slice(0, overlapLength)) {
                // Combine strings without the overlapping part
                return stringA + stringB.slice(overlapLength);
            }
        }

        // If no overlap, return the concatenation of both strings
        return stringA + stringB;
    };

    // Declare an array for the input strings
    const strings: string[] = [firstString, secondString, thirdString];

    // Define all permutations of the indices for string combinations
    const permutations: number[][] = [
        [0, 1, 2],
        [0, 2, 1],
        [1, 0, 2],
        [1, 2, 0],
        [2, 0, 1],
        [2, 1, 0],
    ];

    // Initialize the answer string
    let answer = '';

    // Iterate over each permutation
    for (const [index1, index2, index3] of permutations) {
        // Merge the strings according to the current permutation
        const currentCombo = mergeMinimum(mergeMinimum(strings[index1], strings[index2]), strings[index3]);

        // Update the answer if the current combination is shorter or lexicographically smaller
        if (answer === '' || currentCombo.length < answer.length || (currentCombo.length === answer.length && currentCombo < answer
            answer = currentCombo;
        }
    }

    // Return the answer that represents the shortest merged string
    return answer;
}

```

Time and Space Complexity

Time Complexity

The `minimumString` function is trying to concatenate three strings `a`, `b`, `c`, in all possible permutations, and find the minimum length string that captures all three original strings possibly overlapping. It uses a nested helper function `f`, which is applied twice in the worst case, to combine two strings efficiently by checking for overlaps.

- There are `6` permutations of `(a, b, c)`, which comes from `3!` permutations for three elements.
- The helper function `f` compares every suffix of one string with the prefix of another, this could take up to `O(min(m, n))` time where `m` and `n` are the lengths of the two strings being merged.
- Since `f` is called twice for each permutation, merging first two and then the result with the third, the cost per permutation is `O(m + n) + O(max(m, n, o))`, with `o` being the length of the third string.
- Therefore, the total time complexity for the worst case overlaps checking and merging is `O(6 * (m + n + max(m, n, o)))` which simplifies to `O(m + n + o)` as constants are dropped in big O notation.

Space Complexity

- There is a variable `ans` which keeps track of the best (shortest) answer found so far. It can take up to `O(m + n + o)` space in the situation there is no overlap, however since strings are immutable in Python, each concatenation can create a new string, so this will use space proportional to the sum of lengths of `a`, `b`, and `c`.
- The function doesn't use any additional data structures which grow with the input size, aside from temporary variables for iteration.
- Each call to `f` creates substrings, which in Python are typically pointers to sections of the original strings and therefore do not consume more space than the original strings.
- Hence, the space complexity is `O(m + n + o)` as well, primarily used by the output string and the input strings.

In conclusion, the time complexity of the given code is `O(m + n + o)` and the space complexity is also `O(m + n + o)`.