2800. Shortest String That Contains Three Strings Enumeration Medium Greedy String

### Leetcode Link

# The problem requires us to construct a string with the shortest possible length that contains three given strings a, b, and c as its

that position.

Problem Description

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

- 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

## Here's the intuition behind the solution:

all three.

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.

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

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

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 no overlap is found (i reaches 0), it returns the concatenation of s and t.

same length but lexicographically smaller, ans is updated to this new string.

smallest) for every permutation of input strings.

length, the lexicographically smallest is chosen.

Example Walkthrough

**Step 1: Consider All Permutations** 

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

Permutation 1: "abc", "bcx", "cxy"

Permutation 6: "cxy", "bcx", "abc"

Step 2: Find Best Overlap between Two Strings

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

Step 3: Iterate the Process for All Other Permutations

a = "abc"

b = "bcx"

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.

The implementation of the solution uses Python's built-in permutation function from the itertools module to generate all possible

# • The solution function minimumString(self, a: str, b: str, c: str) starts with an empty string ans to store the final result.

merge them.

Solution Approach

 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

The algorithm uses iterative comparison and string manipulation to build the most efficient string (shortest and lexicographically

Finally, after all permutations are considered and the best merged strings are computed, the function returns the optimal string

output is the minimum length required to contain all three substrings. It also guarantees that if there are strings of the same minimal

stored in ans. By using permutations and a merge function, the algorithm efficiently combines the input strings while always ensuring that the

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

c = "cxy" Following the solution approach:

 Permutation 2: "abc", "cxy", "bcx" Permutation 3: "bcx", "abc", "cxy" Permutation 4: "bcx", "cxy", "abc" Permutation 5: "cxy", "abc", "bcx"

#### 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 5: Return the Optimal String** 

Python Solution

"abcxy".

11

12

18

19

20

21

24

25

26

27

28

29

30

31

32

33

34

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

42

43

44

45

46

47

48

49

50

52

51 };

C++ Solution

2 public:

1 class Solution {

**}**;

Java Solution

class Solution {

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

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".

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

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

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

from itertools import permutations class Solution:

def merge\_strings(s1: str, s2: str) -> str:

if s1 in s2:

if s2 in s1:

return s1 + s2

shortest\_merged\_string = ""

# Return the shortest string found

String[] strings = {a, b, c};

if (s.contains(t)) {

if (t.contains(s)) {

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

for (int i = Math.min(m, n); i > 0; --i) {

return s + t.substring(i);

if (s.substring(m - i).equals(t.substring(0, i))) {

// If there's no overlap, concatenate the strings `s` and `t`

// Function to find the minimum string obtained by concatenating

vector<string> strings = {stringA, stringB, stringC};

// All possible permutations of the three strings.

// String to store the minimum concatenated result.

// Helper function to minimize two strings by concatenating

string minimize(string firstString, string secondString) {

if (firstString.find(secondString) != string::npos) {

// If one string contains the other, return the larger one.

// without overlapping the maximum number of characters.

vector<vector<int>> permutations = {

// Loop over all the permutations.

for (auto& permutation : permutations) {

// Indices for the permutation

minimumResult = temp;

string minimumResult = "";

return minimumResult;

return firstString;

return firstString + secondString;

string minimumString(string stringA, string stringB, string stringC) {

// The input strings are stored in a vector for easy permutation.

// Update the minimumResult if a smaller string is found.

if (minimumResult.empty() || temp.size() < minimumResult.size() ||</pre>

 $\{0, 1, 2\}, \{0, 2, 1\}, \{1, 0, 2\}, \{1, 2, 0\}, \{2, 0, 1\}, \{2, 1, 0\}$ 

int first = permutation[0], second = permutation[1], third = permutation[2];

string temp = minimize(strings[first], minimize(strings[second], strings[third]));

for (int overlapSize = min(lengthFirst, lengthSecond); overlapSize > 0; --overlapSize) {

return firstString + secondString.substr(overlapSize);

// If no overlap is found, return the concatenation of both strings.

if (firstString.substr(lengthFirst - overlapSize) == secondString.substr(0, overlapSize)) {

// If an overlap is found, return the concatenation without the overlapping part.

// Concatenate strings based on the current permutation and minimize them.

(temp.size() == minimumResult.size() && temp < minimumResult)) {</pre>

// three input strings in any order without overlapping.

return s;

return t;

return s + t;

return shortest\_merged\_string

return s2

return s1

def minimumString(self, string\_a: str, string\_b: str, string\_c: str) -> str:

# Identify the longest common suffix of s1 and prefix of s2

# Check if one string is a subsequence of the other and return the longer one.

# Check all permutations of the input strings to find the shortest merged string

// Method to find the minimum string composed by overlapping or concatenating a, b, and c

// Find the maximum overlap starting from the end of `s` and the beginning of `t`

// If the end of `s` matches the beginning of `t`, concatenate the non-overlapping part

merged\_string = merge\_strings(merge\_strings(permutation[0], permutation[1]), permutation[2])

(len(merged\_string) == len(shortest\_merged\_string) and merged\_string < shortest\_merged\_string):</pre>

if not shortest\_merged\_string or len(merged\_string) < len(shortest\_merged\_string) or \</pre>

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

# Then merge s1 and s2 by overlapping this common part 13  $len_s1$ ,  $len_s2 = len(s1)$ , len(s2)14 15 for overlap\_length in range(min(len\_s1, len\_s2), 0, -1): if s1[-overlap\_length:] == s2[:overlap\_length]: 16 return s1 + s2[overlap\_length:]

# Initialize the answer with an empty string.

# If there is no overlap, concatenate the strings

shortest\_merged\_string = merged\_string

public String minimumString(String a, String b, String c) {

// Store strings in an array for easy access

for permutation in permutations((string\_a, string\_b, string\_c)):

```
6
            // All possible permutations of indices 0, 1, and 2
            int[][] permutations = {
 8
                {0, 1, 2}, {0, 2, 1},
 9
                {1, 0, 2}, {1, 2, 0},
10
                \{2, 1, 0\}, \{2, 0, 1\}
11
12
            };
13
14
            // Initialize answer string to be empty at the start
15
            String answer = "";
16
17
            // Iterate over all permutations to find the smallest possible overlap string
18
            for (int[] perm : permutations) {
19
                // Access indices based on the current permutation
20
                int i = perm[0], j = perm[1], k = perm[2];
21
                // Overlap the first two strings, then overlap the result with the third
22
23
                String temp = overlap(overlap(strings[i], strings[j]), strings[k]);
24
25
                // Check if the current string `temp` is smaller or lexicographically smaller than `answer`
                if ("".equals(answer) || temp.length() < answer.length()</pre>
26
27
                        || (temp.length() == answer.length() && temp.compareTo(answer) < 0)) {</pre>
                    answer = temp; // Update `answer` with the new minimum string `temp`
28
29
30
31
32
            return answer; // Return the smallest string after all permutations are checked
33
34
35
        // Helper method to find the overlap between two strings or concatenate if no overlap exists
36
        private String overlap(String s, String t) {
37
            // If one string contains the other, return the larger string
```

#### 36 if (secondString.find(firstString) != string::npos) { 37 return secondString; 38 // Get the lengths of the two strings. 39 int lengthFirst = firstString.size(), lengthSecond = secondString.size(); // Try to find the largest overlap starting from the largest possible size. 41

```
Typescript Solution
    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
  4
             if (stringA.includes(stringB)) {
                 return stringA;
             if (stringB.includes(stringA)) {
  8
                 return stringB;
  9
 10
 11
 12
             // Determine the lengths of both strings
 13
             const lengthA = stringA.length;
 14
             const lengthB = stringB.length;
 15
 16
             // Find the longest suffix of stringA that is a prefix of stringB
 17
             for (let overlapLength = Math.min(lengthA, lengthB); overlapLength > 0; --overlapLength) {
 18
                 if (stringA.slice(-overlapLength) === stringB.slice(0, overlapLength)) {
                     // Combine strings without the overlapping part
 19
 20
                     return stringA + stringB.slice(overlapLength);
 21
 22
 23
 24
            // If no overlap, return the concatenation of both strings
 25
            return stringA + stringB;
 26
        };
 27
 28
        // Declare an array for the input strings
 29
         const strings: string[] = [firstString, secondString, thirdString];
 30
 31
        // Define all permutations of the indices for string combinations
 32
         const permutations: number[][] = [
 33
             [0, 1, 2],
 34
             [0, 2, 1],
 35
             [1, 0, 2],
 36
             [1, 2, 0],
 37
             [2, 0, 1],
 38
             [2, 1, 0],
        1;
 39
 40
 41
        // Initialize the answer string
 42
        let answer = '';
 43
 44
        // Iterate over each permutation
 45
         for (const [index1, index2, index3] of permutations) {
 46
             // Merge the strings according to the current permutation
 47
             const currentCombo = mergeMinimum(mergeMinimum(strings[index1], strings[index2]), strings[index3]);
 48
```

if (answer === '' || currentCombo.length < answer.length || (currentCombo.length === answer.length && currentCombo < answer

// Update the answer if the current combination is shorter or lexicographically smaller

## 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.

**Time Complexity** 

49

50

51

52

53

54

55

56

58

57 }

return answer;

**Time and Space Complexity** 

answer = currentCombo;

// Return the answer that represents the shortest merged string

#### 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 0(m + n) + O(max(m, n, o)), with o being the length of the third string.

Space Complexity

 Therefore, the total time complexity for the worst case overlaps checking and merging is 0(6 \* (m + n + max(m, n, o))) which simplifies to O(m + n + o) as constants are dropped in big O notation.

The minimumString function is trying to concatenate three strings a, b, c, in all possible permutations, and find the minimum length

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

• There is a variable ans which keeps track of the best (shortest) answer found so far. It can take up to 0(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.

There are 6 permutations of (a, b, c), which comes from 3! permutations for three elements.

- 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.
- In conclusion, the time complexity of the given code is 0(m + n + o) and the space complexity is also 0(m + n + o).

• Hence, the space complexity is 0(m + n + o) as well, primarily used by the output string and the input strings.