## **Problem Description**

subsequence of a string t is a new string generated from t after removing some (can be none) characters without changing the order of the remaining characters. For example, "ace" is a subsequence of "abcde" while "aec" is not. The task is to construct the shortest common supersequence that has both str1 and str2 as its subsequences. It is important to note that there can be multiple valid supersequences that satisfy the conditions, and any valid one can be returned.

The problem provided requires us to find the shortest string that contains two given strings, str1 and str2, as subsequences. A

Intuition

and str2.

(LCS) of the two strings. 1. The LCS assists us in identifying the common base around which we can organize the other characters in the supersequence. Since the LCS is the longest sequence found in both strings, we need to include it only once to cover its occurrence in both str1

The intuition behind the solution starts by recognizing that this is a classic problem of finding the Longest Common Subsequence

- 2. Starting with an empty result string, we move through str1 and str2 in parallel, adding characters from both strings to the result. Whenever we find a pair of characters in str1 and str2 that are part of the LCS, we only add this character once to the result.
- 3. To be more practical, we create a dynamic programming table, denoted as f, that stores the lengths of LCS for different substrings of str1 and str2. This table can be built by iterating over all characters of str1 and str2 where f[i][j] represents the length of the LCS between str1[:i] and str2[:j].
- 4. Once we have filled this table, we can backtrack from f[m][n] (where m is the length of str1 and n is the length of str2) to construct the shortest common supersequence by choosing characters from str1 or str2 or both when they are the same and they match the LCS character.
- By using the dynamic programming approach to find the LCS and careful backtracking to build the actual supersequence, we can construct a supersequence that is the shortest possible combination of str1 and str2, containing both as subsequences. **Solution Approach**

The solution utilizes dynamic programming (DP) to compute the shortest common supersequence. Here's how the steps break down in the provided code:

# of str1 and str2.

move left in the table (decrement i).

in finding the LCS, and backtracking to build the supersequence from the LCS.

and move up (decrement j).

2. We fill in the DP table f using a nested loop. Here, f[i][j] will be filled with the length of the LCS of substrings str1[:i] and str2[:j]. If str1[i - 1] is equal to str2[j - 1], this means the characters match and can be part of the LCS, so we set f[i][j]

to be 1 plus the length of the LCS at f[i - 1][j - 1]. Otherwise, we take the maximum of the lengths of the two possible LCS,

1. A two-dimensional DP array f with dimensions  $(m+1) \times (n+1)$  is initialized with zeros, where m and n are the lengths of str1 and

str2, respectively. This array will eventually contain the lengths of the longest common subsequences for all possible substrings

3. After constructing the DP table, we backtrack to build the shortest common supersequence. The list ans is used to store the characters of the supersequence in reverse as we'll be starting from the bottom-right corner of the DP table and moving towards the top-left corner.

by either including the last character of str1 (f[i - 1][j]) or str2 (f[i][j - 1]).

4. The backtracking logic works as follows: ∘ If i or j reaches 0, it means we have reached the end of str1 or str2, so we append the remaining characters of the other string. ∘ If f[i][j] is equal to f[i-1][j], it means the current character of str1 is not part of the LCS, so we include str1[i-1] and

∘ If f[i][j] is equal to f[i][j-1], it's the character from str2 that's not part of the LCS, so we add str2[j-1] to the answer

o If none of the above conditions meet, it means both characters from str1 and str2 are part of the LCS, so we add either one

- to the answer and move diagonally up-left in the table (decrement both i and j). 5. Because ans contains the characters in reverse order (from backtracking), we reverse it back to get the correct order of the shortest common supersequence and return it as a string.
- Example Walkthrough

Let's take two strings str1 = "abac" and str2 = "cab" to demonstrate how to find the shortest common supersequence using the

This algorithm ensures the generation of the shortest string which is a supersequence of both str1 and str2, using DP for efficiency

1. First, we initialize an empty DP table f with dimensions  $(4+1) \times (3+1)$  since len(str1) is 4 and len(str2) is 3. Each cell in f will represent the longest common subsequence (LCS) length for substrings ending at str1[i-1] and str2[j-1].

○ Start with i = 1 and j = 1. If str1[i - 1] == str2[j - 1], set f[i][j] to f[i - 1][j - 1] + 1. Otherwise, set f[i][j] to

## • Repeat this process to fill the entire table. Since "abac" has no common characters with "cab" in the first position, the first row and column will be filled with incremental counts.

dynamic programming approach.

2. We then populate the table f as follows:

 $\max(f[i-1][j], f[i][j-1]).$ 

answer and decrement i to 3.

The filled DP table f would look something like this:

This table tells us that the length of the LCS for "abac" and "cab" is 2.

6 c 0 1 2 2

3. Starting from f[4][3], we backtrack to construct the supersequence. Initializing i = 4, j = 3, we go backwards and:

decrement i. • Finally, str1[0] matches str2[0] (c), so we add "c" to our answer and decrement each index. The result string (ans) at this point is "cbac" in reverse order.

At f[2][1], the characters str1[1] (b) and str2[0] (c) don't match. Since f[2][1] == f[1][1], we add "b" from str1 and

Note that the characters str1[3] (c) and str2[2] (a) aren't equal. Since f[4][3] == f[3][3], we add "c" from str1 to our

Now, str1[2] (a) matches str2[2] (a), so we add "a" from either string to our answer and decrement both i and j.

have successfully constructed the shortest string containing str1 = "abac" and str2 = "cab" as subsequences. **Python Solution** 

In this way, by using dynamic programming to determine the LCS of the two strings and carefully backtracking from the end, we

# longest common subsequence of str1[:i] and str2[:j]  $dp_table = [[0] * (len_str2 + 1) for _ in range(len_str1 + 1)]$ 8 9 # Build the table in bottom-up manner 10

4. Reversing ans, we get "cabc" which is the shortest common supersequence of str1 and str2.

# Dynamic programming table f, where f[i][j] will store the length of the

# If characters match, add 1 to the diagonal value

dp\_table[i][j] = dp\_table[i - 1][j - 1] + 1

# Otherwise, take the maximum value from above or left cell

# If we've finished strl, add remaining str2 characters

# Follow the path of the longest common subsequence

def shortestCommonSupersequence(self, str1: str, str2: str) -> str:

# Get the lengths of both input strings

for i in range(1, len\_str1 + 1):

len\_str1, len\_str2 = len(str1), len(str2)

for j in range(1, len\_str2 + 1):

# Initialize pointers for both strings

sscs.append(str1[i])

i, j = len\_str1, len\_str2

while i > 0 or j > 0:

if i == 0:

else:

if str1[i - 1] == str2[j - 1]:

# Trace back from the bottom-right corner of the table

if dp\_table[i][j] == dp\_table[i - 1][j]:

18  $dp_{table[i][j]} = max(dp_{table[i - 1][j], dp_{table[i][j - 1])}$ 19 20 # This will hold the shortest common supersequence characters 21 sscs = []22

```
30
31
                    sscs.append(str2[j])
32
                elif j == 0:
33
                    # If we've finished str2, add remaining str1 characters
34
                    i -= 1
```

class Solution:

11

12

13

14

15

16

17

23

24

25

26

27

28

29

35

36

37

38

```
39
                         i -= 1
 40
                         sscs.append(str1[i])
 41
                     elif dp_table[i][j] == dp_table[i][j - 1]:
 42
                          i -= 1
 43
                          sscs.append(str2[j])
 44
                     else:
 45
                         # If characters match, go diagonally up-left and add the character
 46
                         i -= 1
 47
                          i -= 1
 48
                         sscs.append(str1[i])
 49
 50
             # The sscs list contains the shortest common supersequence in reverse order;
 51
             # reverse it to get the correct sequence
             return ''.join(sscs[::-1])
 52
 53
Java Solution
     class Solution {
         public String shortestCommonSupersequence(String str1, String str2) {
             int str1Length = str1.length(), str2Length = str2.length();
  3
             int[][] dp = new int[str1Length + 1][str2Length + 1];
  6
             // Calculate the length of longest common subsequence using dynamic programming
             for (int i = 1; i <= str1Length; ++i) {</pre>
                 for (int j = 1; j <= str2Length; ++j) {</pre>
  8
                     if (str1.charAt(i - 1) == str2.charAt(j - 1)) {
  9
 10
                         // Characters match, take diagonal value plus one
 11
                         dp[i][j] = dp[i - 1][j - 1] + 1;
 12
                     } else {
 13
                         // Choose the maximum value from the cell above or the cell to the left
 14
                         dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
 15
 16
 17
 18
 19
             // Reconstruct the shortest common supersequence from the dp table
 20
             StringBuilder shortestSupersequence = new StringBuilder();
 21
             int i = str1Length, j = str2Length;
 22
             while (i > 0 | | j > 0) {
 23
 24
                 if (i == 0) {
 25
                     // If we have reached the beginning of strl, add remaining characters from str2
 26
                     shortestSupersequence.append(str2.charAt(--j));
 27
                 } else if (j == 0) {
                     // If we have reached the beginning of str2, add remaining characters from str1
 28
                     shortestSupersequence.append(str1.charAt(--i));
 29
 30
                 } else {
                     // Move diagonally if characters match
 31
 32
                     if (dp[i][j] == dp[i - 1][j - 1] + 1) {
 33
                         shortestSupersequence.append(str1.charAt(--i));
 34
                         --j;
                     } else if (dp[i][j] == dp[i - 1][j]) {
 35
                         // Move up if the character from strl is part of the supersequence
                         shortestSupersequence.append(str1.charAt(--i));
  37
 38
                     } else {
```

// Move left if the character from str2 is part of the supersequence

// The constructed supersequence is in reverse, so reverse it to get the final string

shortestSupersequence.append(str2.charAt(--j));

return shortestSupersequence.reverse().toString();

C++ Solution

39

40

41

42

43

44

45

46

47

48

49

```
1 #include <vector>
  2 #include <algorithm>
     #include <string>
    class Solution {
    public:
         // Method calculates the shortest common supersequence of two strings
         std::string shortestCommonSupersequence(std::string str1, std::string str2) {
             int m = str1.size(); // Length of str1
 10
             int n = str2.size(); // Length of str2
 11
 12
             // dp table with dimensions m+1 by n+1, initialized to 0
 13
             std::vector<std::vector<int>> dp(m + 1, std::vector<int>(n + 1, 0));
 14
 15
             // Fill the dp table
             for (int i = 1; i \le m; ++i) {
 16
 17
                 for (int j = 1; j \le n; ++j) {
                     if (str1[i - 1] == str2[j - 1]) {
 18
 19
                         // Characters match, increment the length of common subsequence
                         dp[i][j] = dp[i - 1][j - 1] + 1;
 20
 21
                     } else {
                         // Characters do not match, take the maximum from either the top or left cell
 22
                         dp[i][j] = std::max(dp[i - 1][j], dp[i][j - 1]);
 23
 24
 25
 26
 27
 28
             // Reconstruct the shortest common supersequence from the dp table
 29
             std::string sequence;
 30
             int i = m, j = n;
 31
             while (i > 0 || j > 0) {
 32
                 if (i == 0) {
 33
                     // If we have reached the beginning of strl, append remaining str2
                     sequence += str2[--j];
 34
 35
                 } else if (j == 0) {
 36
                     // If we have reached the beginning of str2, append remaining str1
 37
                     sequence += str1[--i];
                 } else {
 38
                     // Decide which character to append from either str1 or str2
 39
 40
                     if (dp[i][j] == dp[i - 1][j]) {
 41
                         // Coming from top, append strl's character
 42
                         sequence += str1[--i];
                     } else if (dp[i][j] == dp[i][j - 1]) {
 43
 44
                         // Coming from left, append str2's character
 45
                         sequence += str2[--j];
 46
                     } else {
 47
                         // When characters match, append one of them and move diagonally
 48
                         sequence += str1[--i];
 49
                         --j;
 50
 51
 52
 53
 54
             // Since the construction was from back to front, reverse the string
 55
             std::reverse(sequence.begin(), sequence.end());
 56
 57
             return sequence;
 58
 59
    };
 60
Typescript Solution
    function shortestCommonSupersequence(str1: string, str2: string): string {
```

## 30 if (dpMatrix[i][j] === dpMatrix[i - 1][j]) { supersequence.push(str1[--i]); 31 32 } else if (dpMatrix[i][j] === dpMatrix[i][j - 1]) { 33 supersequence.push(str2[--j]); 34 } else { 35 supersequence.push(str1[--i]);

const str1Length = str1.length;

const str2Length = str2.length;

} else {

let i = str1Length;

let j = str2Length;

while (i > 0 || j > 0) {

} else if (j === 0) {

--j;

Time and Space Complexity

if (i === 0) {

} else {

let supersequence: string[] = [];

for (let i = 1; i <= str1Length; ++i) {</pre>

for (let j = 1; j <= str2Length; ++j) {</pre>

supersequence.push(str2[--j]);

supersequence.push(str1[--i]);

return supersequence.reverse().join('');

strings str1 and str2. There are two main steps in this algorithm:

if (str1[i - 1] === str2[j - 1]) {

// Create a 2D matrix to store the lengths of the longest common subsequences

// Build the matrix with the lengths of the longest common subsequences

dpMatrix[i][j] = dpMatrix[i - 1][j - 1] + 1;

// Initialize an array to build the shortest common supersequence

// Backtrack through the matrix to find the shortest common supersequence

// Reverse the array and join it to form the final supersequence string

const dpMatrix = new Array(str1Length + 1).fill(0).map(() => new Array(str2Length + 1).fill(0));

dpMatrix[i][j] = Math.max(dpMatrix[i - 1][j], dpMatrix[i][j - 1]);

3

4

5

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

36

37

38

39

40

41

42

43

44

The given Python code defines a method shortestCommonSupersequence to find the shortest common supersequence of two strings str1 and str2. To analyze the computational complexities, we need to consider both the time taken to execute the code, which depends on the number of operations performed, and the space required to store the data during execution.

The time complexity of the algorithm is determined by the nested loop structure which iterates over the lengths of the two input

1. Filling out the dynamic programming table f, which has dimensions (m + 1) \* (n + 1), where m is the length of str1 and n is the

length of str2. Each cell in the table is filled in constant time, so the time required to fill this table is directly proportional to the

## number of cells, resulting in a time complexity of 0(m \* n). 2. Constructing the shortest common supersequence by traversing back from the bottom-right corner of the table to the top-left,

Time Complexity

appending characters accordingly. The length of this path is m + n in the worst case because each step either moves left or up in the table, and there can be at most m upward moves and n left moves. Therefore, the time complexity for the reconstruction step is 0(m + n).

- The overall time complexity of the algorithm is the sum of these two parts, predominantly the first since it involves iterating over a two-dimensional array. Thus, the total time complexity is 0(m \* n) + 0(m + n), which simplifies to 0(m \* n) because m \* n
- dominates m + n for large values of m and n. **Space Complexity** The space complexity is dictated by the space required to store the dynamic programming table f, which has (m + 1) \* (n + 1)

cells, each storing an integer. Hence, the space complexity is 0(m \* n) to accommodate this table. Additional space is used for the

In summary, the time complexity of the code is 0(m \* n), and the space complexity of the code is also 0(m \* n).

list ans which is used to construct the shortest common supersequence. In the worst case, ans could grow to a size of m + n, when all characters from both str1 and str2 are used in the supersequence. However, the space complexity remains 0(m \* n) because m \* n dominates m + n.