Topic Introduction

Today's theme: Dynamic Programming for Strings — Finding Common Patterns

Dynamic Programming (DP) is a go-to technique for solving optimization problems where choices overlap and subproblems repeat. In string problems, DP shines when you need to compare, align, or merge two strings to find patterns like matches, subsequences, or supersequences.

What is String Dynamic Programming?

String DP is about breaking down complex string problems into smaller, manageable pieces, storing results, and reusing them. Typical questions include:

- How do you find the *longest* or *shortest* string that satisfies a certain property for two given strings?
- What's the minimal number of operations to convert `A` into `B`?

How does it work?
Imagine a grid where one string runs across the top and the other down the side. Each cell in the grid represents a subproblem: aligning prefixes of the two strings up to certain lengths. You fill this grid by building up from smaller cases to bigger ones.

Why is it useful?
String DP pops up everywhere: DNA sequence alignment, spell-check, file diff tools, and, of course, coding interviews! Interviewers love these problems because they test your ability to break problems into subproblems and understand overlapping work.

Simple Example (not one of our main problems):
Suppose you want to check if one string is a subsequence of another. You can use two pointers or DP to check if all characters of the first string appear in order within the second.

Introducing Today's 3 Problems

Today, we'll explore three classic string DP problems, all about finding *common patterns* between two strings:

- **Longest Common Subsequence** (LeetCode 1143): Find the length of the longest sequence that appears in both strings (not necessarily contiguous).
- **Longest Common Substring** (LeetCode 718): Find the length of the longest contiguous substring that appears in both strings.
- **Shortest Common Supersequence** (LeetCode 1092): Find the shortest string that contains both given strings as subsequences.

Why are these grouped together?
All three use string DP to uncover shared structure between two strings. The first two focus on what's common (subsequence vs substring), while the last asks how to combine both strings efficiently.

Let's dive in!

Problem 1: Longest Common Subsequence

Problem link: LeetCode 1143

Problem Statement (Rephrased):
Given two strings, return the length of their longest common subsequence.
A *subsequence* is a sequence you can get from a string by deleting some (or no) characters, without changing the order of the remaining characters.

Example:
Input: `text1 = "abcde"`, `text2 = "ace"`
Output: `3`
Explanation: The longest common subsequence is `"ace"`.

Thought Process:
The brute-force way is to try every possible subsequence of both strings and check which ones match. But that's *exponentially* slow.

Try this example on paper:
`text1 = "aggtab"`, `text2 = "gxtxayb"`
What is their longest common subsequence?

Another test case to try:
`text1 = "abc"`, `text2 = "def"` (what should the answer be?)

Brute-force Approach

Try all subsequences of both strings and check which ones match.

- Exponential time (O(2^N * 2^M)), not practical for interviews.

Optimal Approach: Dynamic Programming

Core idea:

- Use a DP table where `dp[i][j]` is the length of the LCS between the first `i` characters of `text1` and first `j` of `text2`.
- If `text1[i-1] == text2[j-1]`, the character is part of the LCS:

`dp[i][j] = 1 + dp[i-1][j-1]`

- Otherwise, skip one character from either string:

`dp[i][j] = max(dp[i-1][j], dp[i][j-1])`

Step-by-step logic:

- Create a DP table of size `(len(text1)+1) x (len(text2)+1)`, initialized to 0.
- Loop through each character pair, filling the table according to the rules above.
- The answer is in `dp[len(text1)][len(text2)]`.

Python Solution

```
def longestCommonSubsequence(text1, text2):
    # Create a DP table with an extra row and column (for the 0 case)
    n, m = len(text1), len(text2)
    dp = [[0] * (m + 1) for _ in range(n + 1)]

    # Fill the DP table
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = 1 + dp[i - 1][j - 1]
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
    return dp[n][m]
```

- **Time Complexity:** O(N*M), where N and M are lengths of the two strings.
- **Space Complexity:** O(N*M) for the DP table.

What does each part do?

- The table stores solutions to subproblems: `dp[i][j]` is LCS for first `i` and `j` chars.
- The nested loops fill in the answers, building up from the smallest substrings.
- At each step, if the characters match, we add 1 to the answer from the previous diagonal cell; otherwise, we take the best result from either skipping one character from `text1` or `text2`.

Walkthrough Example

Let’s trace `text1 = "abcde"`, `text2 = "ace"`:

Filling the table, we get:

		a	c	e
	0	0	0	0
a	0	1	1	1
b	0	1	1	1
c	0	1	2	2
d	0	1	2	2
e	0	1	2	3

Final answer: `3`.

Try this test case yourself:
`text1 = "abcba"`, `text2 = "abcbcba"`
What is the LCS length?

Take a moment to try this on your own before reading further!

Did you know?

For space optimization, you only need two rows of the DP table at a time, since each row only depends on the previous one.

Problem 2: Longest Common Substring

Problem link: LeetCode 718

Problem Statement (Rephrased):

Given two arrays (or strings), find the length of the longest contiguous subarray (substring) that appears in both.

Example:

Input: `nums1 = [1,2,3,2,1]`, `nums2 = [3,2,1,4,7]`

Output: `3`

Explanation: The longest common subarray is `[3,2,1]`.

How is this different from LCS?

Now, the match must be *contiguous*. So, you can't skip characters — as soon as the match is broken, you must restart.

Brute-force Approach

Try every pair of starting indices in both strings, and compare substrings until they mismatch.

- Time: O(N^3)

Optimal Approach: Dynamic Programming

Core pattern:

- Use a DP table: `dp[i][j]` is the length of the longest common substring ending at `A[i-1]` and `B[j-1]`.
- If `A[i-1] == B[j-1]`:

`dp[i][j] = dp[i-1][j-1] + 1`

- Else:

`dp[i][j] = 0`

- Track the maximum value as you fill the table.

Pseudocode

```
function longestCommonSubstring(A, B):
    n = length of A
    m = length of B
    create dp[n+1][m+1] initialized to 0
    maxLen = 0
    for i from 1 to n:
        for j from 1 to m:
            if A[i-1] == B[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
                maxLen = max(maxLen, dp[i][j])
            else:
                dp[i][j] = 0
    return maxLen
```

Step-by-step Example

Input: `A = "ababc"`, `B = "babca"`

Fill the table. The longest common substring is `"babc"`, length 4.

Try this test case:
`A = "abcdef"`, `B = "zabcf"` (What's the answer?)

Tracing with code:

- For every matching pair, increment based on the previous diagonal cell.
- Reset to 0 on mismatch.

Time Complexity: O(N*M)
Space Complexity: O(N*M)

Problem 3: Shortest Common Supersequence

Problem link: LeetCode 1092

Problem Statement (Rephrased):
Given two strings, find the shortest string that has both as subsequences.

Example:
Input: `str1 = "abac"`, `str2 = "cab"`
Output: `"cabac"`
Explanation: `"cabac"` is the smallest string that contains both as subsequences.

How is this different from the previous ones?
Now, instead of finding what's common, you want to *merge* both strings efficiently, minimizing extra characters.

Brute-force Approach

Try all possible ways to merge the strings while maintaining their order.

- Exponential time (impractical).

Optimal Approach: Dynamic Programming

Key insight:

- The length of the SCS is `len(str1) + len(str2) - length of LCS(str1, str2)`.
- To find the actual sequence, reconstruct it using the DP table for LCS.

Pseudocode

```
function shortestCommonSupersequence(str1, str2):
    # First, compute the LCS table for str1 and str2
    dp = lcs_table(str1, str2)
    i = len(str1)
    j = len(str2)
    result = empty string
    while i > 0 and j > 0:
        if str1[i-1] == str2[j-1]:
            prepend str1[i-1] to result
            i -= 1
            j -= 1
        else if dp[i-1][j] > dp[i][j-1]:
            prepend str1[i-1] to result
            i -= 1
        else:
            prepend str2[j-1] to result
            j -= 1
    # Add remaining characters from str1 or str2
    while i > 0:
        prepend str1[i-1] to result
        i -= 1
    while j > 0:
        prepend str2[j-1] to result
        j -= 1
    return result reversed
```

Try this test case:

`str1 = "geek"`, `str2 = "eke"`

What is the shortest common supersequence?

Guidance:

- Build the LCS table as in Problem 1.
- Walk backwards from the end, appending characters to your result (when both match, append once; otherwise, append the character from the string with the higher DP value).
- Don't forget to add leftovers when one string is exhausted.

Time Complexity: O(N*M)

Space Complexity: O(N*M)

Take a moment to implement this and verify the output for the test case above!

Can you spot how the LCS helps minimize extra characters? Try to reason why this works!

Summary and Next Steps

Recap:

Today, you tackled three fundamental string DP problems:

- **LCS:** Find maximum overlap (not necessarily contiguous).
- **Longest Common Substring:** Find maximum contiguous overlap.
- **Shortest Common Supersequence:** Combine both strings efficiently by minimizing extra characters.

Key patterns & insights:

- Use a 2D DP table to compare prefixes of the two strings.
- LCS is the backbone for many string alignment problems.
- For substrings, reset on mismatch; for subsequences, allow skipping.
- Many supersequence problems can be reduced to LCS computations.

Common traps:

- Mixing up substring and subsequence (contiguous vs not).
- Off-by-one errors in DP table indices.
- Not considering all remaining characters when constructing the supersequence.

Action List

- Implement all three problems yourself, even those with full code shown!
- For Problem 2 and 3, try solving with a different approach (e.g., sliding window, suffix automata, or recursion with memoization).
- Explore related problems: Edit Distance, Sequence Alignment, Interleaving Strings.
- Compare your DP table traces and code with others for better style and bug-spotting.
- If you get stuck, step through the DP table by hand — it's a great way to learn!

Keep going! Mastering these patterns will unlock a whole class of interview questions for you. Happy coding!