Topic Introduction

Today we're diving into a family of classic coding interview questions that all rely on a powerful concept: **Dynamic Programming on Strings** (specifically, *edit operations*). If you've ever wondered about the minimum number of changes needed to transform one string into another, you've already met this theme!

What is String Dynamic Programming (Edit Operations)?

In string DP, we often want to compare two strings and figure out how to "edit" one into the other using a set of allowed operations (insert, delete, replace, etc.). Each operation has a cost, and our goal is to reach the target string with the smallest total cost or fewest operations.

How does it work?

• We break the problem into subproblems: What if I only cared about the first i characters of string A and first j of string B?

• We use a table (matrix) to store solutions for these subproblems so we never recompute them.

• We build up the answer, usually row by row or recursively with memoization.

When and why is this useful?

• These problems are classic because they test your ability to model a problem, recognize overlapping subproblems, and optimize using DP.

• They appear in spell checkers, DNA sequence analysis, file comparison, and more.

Quick Example (not one of today's main problems):
Suppose you want to find the length of the longest common subsequence (LCS) of "abcde" and "ace". You'd use a DP table to compare characters one by one, storing the best answer at each prefix.

Let's look at three famous problems that use these ideas, each with a unique twist:

• **Edit Distance** (Levenshtein Distance): Minimum edits (insert, delete, replace) to convert one string to another.

• **Delete Operation for Two Strings**: Minimum deletes (only delete allowed) to make two strings equal.

• **Minimum ASCII Delete Sum for Two Strings**: Like Delete Operation, but each character you delete has a cost equal to its ASCII value. Find the minimum total cost.

Why are these grouped together?
All three are classic string DP problems using edit operations. The core logic is very similar: compare prefixes, decide on an operation, and record the minimum cost. The difference is which operations are allowed and how each operation is "scored."

Problem 1: Edit Distance

[Leetcode 72: Edit Distance]

Problem Statement (in my words):
Given two strings word1 and word2, find the minimum number of operations required to convert word1 into word2. You can insert a character, delete a character, or replace a character. Each operation costs 1.

Example

```
Input: word1 = "horse", word2 = "ros"
Output: 3

Explanation:
- horse -> rorse (replace 'h' with 'r')
- rorse -> rose (remove 'r')
- rose -> ros (remove 'e')
```

Try manually: What if word1 = "intention", word2 = "execution"? (Answer: 5)

Thought Process

This is a classic dynamic programming problem. The brute-force way is to try all possible ways to edit word1 into word2, which is exponential in time.

But, if we think recursively:

- If the last characters of the current prefixes are the same, we don't need to edit them. Move to the previous characters.
- If they’re different, consider three options:
    - Insert a character (advance in word2)
    - Delete a character (advance in word1)
    - Replace a character (advance in both)

Our subproblem: what is the minimum edit distance for the first i characters of word1 and the first j of word2?

Let's formalize it:

- **dp[i][j]** = minimum edits to convert word1[:i] to word2[:j]

Base cases:

- dp[0][j] = j (need to insert all of word2[:j])
- dp[i][0] = i (need to delete all of word1[:i])

Brute-force approach

Try all combinations recursively. Time: $O(3^{(m+n)})$ (very slow).

Optimal DP approach

We fill a 2D table of size (m+1) by (n+1) (m, n are lengths of the strings), building it from the ground up.

Clean Python Solution

```
def minDistance(word1, word2):
    m, n = len(word1), len(word2)
    # dp[i][j] = min edits to convert word1[:i] to word2[:j]
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Fill base cases
    for i in range(m + 1):
        dp[i][0] = i  # delete all characters
    for j in range(n + 1):
        dp[0][j] = j  # insert all characters

    # Fill the table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(
                    dp[i - 1][j],     # delete
                    dp[i][j - 1],     # insert
                    dp[i - 1][j - 1]  # replace
                )
    return dp[m][n]
```

Time Complexity: O(mn)

Space Complexity: O(mn) (can be reduced to O(min(m, n)) with some tweaks)

Code Explanation

- `dp[i][j]` means: min edits to convert word1[:i] to word2[:j]
- We fill the base cases: if one string is empty, edits = length of the other string.
- For each character, if they match, move diagonally (no cost). If not, take min of insert, delete, or replace (each adding 1).

Trace with "horse" and "ros"

- Start with dp[0][0] (empty to empty = 0 edits).
- Step through the table, comparing "h" vs "r", "o" vs "o", etc.
- Eventually, dp[5][3] = 3 (as in the example above).

Try this test case:

word1 = "kitten", word2 = "sitting" (Expected: 3)

Take a moment to solve this on your own before jumping into the solution!

Problem 2: Delete Operation for Two Strings

[Leetcode 583: Delete Operation for Two Strings]

Problem Statement (in my words):

Given two strings, return the minimum number of delete operations needed to make both strings equal. Only deletions are allowed.

Example

```
Input: word1 = "sea", word2 = "eat"
Output: 2

Explanation:
- Delete 's' from "sea" → "ea"
- Delete 't' from "eat" → "ea"
```

Try this: word1 = "leetcode", word2 = "etco" (Expected: 4)

Similarities and Differences

This is similar to Edit Distance, but *only* deletes are allowed. No insert or replace.

Brute-force

Try all deletion combinations. Exponential time.

DP Approach

Let's observe:

- To make both strings equal with only deletes, we need to remove all characters that aren't shared.
- The best we can do is to keep their **Longest Common Subsequence** (LCS).
- So, answer = (len(word1) - LCS) + (len(word2) - LCS)

Step-by-step logic

- Find the length of the LCS between the two strings.
- Number of deletes = (len(word1) - LCS) + (len(word2) - LCS)

Pseudocode

```
function minDistance(word1, word2):
    m = length of word1
```

```
    n = length of word2
    dp = 2D array (m+1) x (n+1), initialized to 0

    for i from 1 to m:
        for j from 1 to n:
            if word1[i-1] == word2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    lcs = dp[m][n]
    return (m - lcs) + (n - lcs)
```

Example Walkthrough

word1 = "sea", word2 = "eat"

- LCS is "ea" (length 2)
- Deletes: (3 - 2) + (3 - 2) = 2

Another test case:
word1 = "delete", word2 = "leet" (Expected: 2)

Time Complexity: O(mn)
Space Complexity: O(mn)

Take a moment to try this approach on your own! Can you spot the useful pattern? (Hint: LCS!)

Problem 3: Minimum ASCII Delete Sum for Two Strings

[Leetcode 712: Minimum ASCII Delete Sum for Two Strings]

Problem Statement (in my words):
Given two strings, delete characters from either string so that the two strings are equal. The "cost" of deleting a character is its ASCII value. Return the minimum total cost to make the strings equal.

What's New or More Challenging?

This problem builds on Problem 2, but instead of counting deletions, each character has a "weight" (its ASCII value). We want to minimize the total ASCII sum of deleted characters.

Brute-force

Try all delete combinations, keeping track of total ASCII sum. Not feasible for long strings.

DP Approach

We adapt the LCS-style DP, but track the minimum sum of deleted ASCII values:

Step-by-step logic

Let dp[i][j] = minimum total ASCII sum of deleted characters to make word1[:i] and word2[:j] equal.

- If word1[i-1] == word2[j-1]: no deletion needed, move to dp[i-1][j-1]
- Else: we have two choices:
  - Delete word1[i-1]: cost += ord(word1[i-1]), move to dp[i-1][j]
  - Delete word2[j-1]: cost += ord(word2[j-1]), move to dp[i][j-1]

Base cases:

- dp[0][j]: sum of ASCII values of word2[:j] (delete all of word2)
- dp[i][0]: sum of ASCII values of word1[:i] (delete all of word1)

Pseudocode

```
function minimumDeleteSum(s1, s2):
    m = length of s1
    n = length of s2
    dp = 2D array (m+1) x (n+1), initialized to 0

    for i from 1 to m:
        dp[i][0] = dp[i-1][0] + ASCII(s1[i-1])
    for j from 1 to n:
        dp[0][j] = dp[0][j-1] + ASCII(s2[j-1])

    for i from 1 to m:
        for j from 1 to n:
            if s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = min(
                    dp[i-1][j] + ASCII(s1[i-1]),
                    dp[i][j-1] + ASCII(s2[j-1])
                )
    return dp[m][n]
```

Example

Input: s1 = "sea", s2 = "eat"

- ASCII('s') = 115, 'e' = 101, 'a' = 97, 't' = 116
- Delete 's' (115), delete 't' (116): Total = 231

Try this test case:
s1 = "delete", s2 = "leet" (Expected: 403)

Time Complexity: O(mn)
Space Complexity: O(mn)

Take a moment to implement this in code and test it! Does the pattern look familiar? Can you think of ways to optimize the space?

Summary and Next Steps

These three problems are grouped together because they all use dynamic programming to solve string transformation questions, focusing on *edit operations*. The core pattern is to model the problem using prefixes and recursively build a DP table. The main differences are which operations are allowed, and how each operation is scored (step cost or ASCII value).

Key insights to remember:
- Many string transformation problems boil down to comparing prefixes and making optimal choices at each step.
- Recognize when a problem is a variant of Edit Distance or LCS.
- Careful base case handling is crucial in DP problems.
- Don’t forget to reconstruct the solution if asked (not just the cost).

Common mistakes:
- Off-by-one errors in indices.
- Not initializing base cases properly.
- Forgetting what each cell in your DP table represents.

Action List:
- Solve all three problems on your own, using both code and pen-and-paper.
- Try solving Problems 2 and 3 with a recursive + memoization approach.
- Explore space optimization for these DP tables (can you use just two rows?).
- Check out similar problems: "Longest Common Subsequence", "Shortest Common Supersequence", or "One Edit Distance".
- Compare your solutions with others for edge cases and code clarity.
- If you get stuck, take a break and revisit — understanding comes with practice!

Keep practicing and you’ll soon find these “scary” string DP problems are just clever applications of a single, powerful pattern. Happy coding!