Summary

I delved into some classic problems that can be solved efficiently using Dynamic Programming. My focus was on understanding not just the problems themselves, but the underlying strategy of breaking them down into simpler, overlapping subproblems to build an optimal solution.

The Longest Common Subsequence (LCS) Problem

I started with the problem of finding the longest common subsequence (LCS) between two sequences. I learned that a subsequence doesn't have to be contiguous, which makes the problem interesting. The key takeaway was how to apply Dynamic Programming to find the *length* of the LCS, and from there, how to reconstruct the subsequence itself.

- **Core Concept:** The optimal substructure is key. The LCS of two sequences X and Y depends on the LCS of their prefixes. I came to understand that if the last characters of the two sequences match, the LCS length is simply 1 + LCS(of prefixes). If they don't match, I must take the best result from two subproblems: (1) ignoring the last character of X and (2) ignoring the last character of Y.
- **DP Formulation:** This logic is captured perfectly in a recursive formula that builds a 2D table, C[i, j], representing the LCS length for prefixes X[1..i] and Y[1..j].
- **Complexity & Application:** I was surprised to learn this O(mn) algorithm powers fundamental tools like git and diff for version control, and has significant applications in bioinformatics for DNA sequence comparison.

The Knapsack Problem

Next, I explored the Knapsack Problem, a classic optimization challenge. The goal is to pack a knapsack with a limited weight capacity (W) by selecting items, each with a specific weight and value, to maximize the total value. I learned to distinguish between its two main variants.

1. **The Unbounded Knapsack Problem**
   This version assumes an infinite supply of each item. I learned that the DP approach here is simpler because the subproblems only depend on the remaining capacity of the knapsack, not on which items have already been used.
   - **DP Formulation:** The solution, K[x], for a knapsack of capacity x is found by trying every item i and picking the one that maximizes v_i + K[x - w_i]. This means for each capacity, I'm simply asking: "What's the best item to add to the optimal solution for the remaining space?"
   - **Complexity:** The algorithm runs in O(nW) time.
2. **The 0/1 Knapsack Problem**
   This is the more constrained version where each item is unique—I can either take it or leave it. This constraint adds a new dimension to the subproblems. Now, a subproblem is defined not only by the knapsack's capacity but also by the set of items available to choose from.

- **DP Formulation:** The solution required a two-dimensional table, K[x, j], representing the maximum value for a capacity x using only the first j items. For each item j, I have a clear choice: either I don't include it, in which case the value is the same as the solution for j-1 items (K[x, j-1]), or I do include it, adding its value to the solution for the remaining capacity and items (v_j + K[x - w_j, j-1]). The final value is the maximum of these two choices.
- **Complexity:** Despite the added complexity in logic, the time complexity remains O(nW).

Overall, my main takeaway is the power of the Dynamic Programming mindset: identifying an optimal substructure and a recursive relationship allows complex problems to be solved by building up from simple, calculated solutions to smaller subproblems.

Tasks 2-4 provided in submission as .ipynb file: