Reflection

Introduction to Dynamic Programming

Dynamic Programming (DP) is an effective problem-solving paradigm designed to optimize solutions for complex problems. It works by breaking a problem into smaller overlapping subproblems, storing their results, and building the global optimal solution from them. Unlike divide-and-conquer, DP specifically handles overlapping subproblems and optimal substructure. Through approaches such as top-down memorization and bottom-up tabulation, DP reduces exponential complexity into polynomial time.

Longest Common Subsequence (LCS)

The LCS problem aims to find the longest subsequence common to two given sequences, where order matters but continuity is not required. For example, between “ABAZDC” and “BACBAD,” the LCS is “ABAD.” The DP solution constructs a two-dimensional table to record the length of LCS for all prefix pairs. The final result is obtained by filling this table and backtracking. The time and space complexity are both O(mn), where m and n are the string lengths.

Knapsack Problem

The Knapsack Problem is a classic optimization problem. In the 0/1 Knapsack, each item has a weight and value, and the goal is to maximize the total value without exceeding capacity, where each item can be chosen at most once.

Unbounded Knapsack

In the Unbounded Knapsack, each item can be chosen multiple times. The DP state can be reduced to a one-dimensional array, where dp[w] stores the best value at capacity w.

0/1 Knapsack (Revisited)

Unlike the unbounded version, the 0/1 Knapsack restricts each item to be used once. The DP transition is similar but carefully ensures that items are only considered once per capacity. Space optimization techniques can further reduce memory usage, showing the flexibility of DP approaches.

**Task 2: Combinatorial DP**

You are climbing a staircase with a total of n steps. At each move, you can hop 1, 2, or 3 steps.

- Write a recursive solution to calculate the number of distinct ways to reach the top.
- Then refactor your solution into a top-down dynamic programming approach using memoization.
- Finally, implement a bottom-up DP solution for comparison.
- Test your implementation for various values of n, and report the time or memory performance (optional but encouraged).

The stair climbing problem requires computing the number of different ways to climb n stairs, taking 1, 2, or 3 steps at a time.

The code implements only a recursive solution, resulting in a time complexity of O(3^n) and a space complexity of O(n).

In contrast, using memorization or bottom-up dynamic programming, the time complexity can be optimized to O(n), while the space complexity remains O(n). Even for n=1000, the problem can be completed in milliseconds, demonstrating the significant performance boost that dynamic programming can bring to algorithms.

**Task 5: Fewest Coins**

Given coins of different denominations and a total amount, find the minimum number of coins needed to make up that amount. Your task is to implement both:

- Top-down memoized solution,
- Bottom-up tabulation version.
- Ensure your solution handles cases where it's not possible to reach the amount.

Example:

Input: coins = [1, 2, 5], amount = 11

Output: 3

Explanation: 11 = 5 + 5 + 1

Input: coins = [2], amount = 3

Output: -1

Explanation: No combination of coins adds to 3.

*Make sure you are ready to get your strategy discussed with your tutor.*

The minimum coin problem requires finding a target amount using coins of given denominations while using the minimum number of coins.

The code provides a bottom-up solution, creating a dp array dp[i] representing the minimum number of coins required to produce amount i, starting with the simplest case and building up the solution step by step.

For example, with coins = [1, 2, 5] and amount = 11, the calculation shows that 3 coins are ultimately required (5 + 5 + 1).

This method has a time complexity of O(amount × coins_count), making it much more efficient than a recursive approach.

The top-down (memoization) approach starts with the target problem, recursively decomposes it into subproblems, and uses memoization to store already computed

results. Its advantages are that it is more intuitive and only computes the necessary subproblems. However, its disadvantage is that it incurs recursive overhead.

The bottom-up (tabulation) approach starts with the simplest subproblem and gradually builds toward the target problem. Its advantages are that it has no recursive overhead and is generally more efficient. However, its disadvantage is that it may compute unnecessary subproblems.

Both approaches guarantee an optimal solution, but the bottom-up approach is generally preferred in practical applications because it avoids the performance overhead of recursive calls and the risk of stack overflow, making it more stable and reliable when handling large-scale problems.