Dynamic Programming Cheatsheet

Topic Overview

Dynamic Programming (DP) in Java optimizes recursive problems by storing subproblem results. This cheatsheet covers DP techniques.

Prerequisites

Arrays

List of Subtopics

- Memoization
- Tabulation
- Fibonacci Sequence
- Knapsack Problem
- Longest Common Subsequence (LCS)
- Longest Increasing Subsequence (LIS)
- Coin Change
- Matrix Chain Multiplication
- Edit Distance
- Partition Problem

Key Concepts Explained

- **Memoization**: Top-down DP with a cache to avoid recalculation.
- **Tabulation**: Bottom-up DP using a table.
- **LCS**: Finds longest common subsequence between two strings.

Approaches to Solve Problems with Step-by-Step Algorithms

- **Memoization**:
  - **Algorithm**:
    1. Initialize a cache (e.g., array or map).
    2. For each recursive call, check cache, return if present.
    3. Compute, store in cache, return result.
  - **Context**: O(n) space, reduces time from exponential to polynomial.
- **Tabulation**:
  - **Algorithm**:
    1. Initialize a table with base cases.
    2. Fill table iteratively based on subproblems.
    3. Return the final table value.
  - **Context**: O(n) space, often more efficient than memoization.
- **Fibonacci Sequence**:
  - **Algorithm**:
    1. Use a DP array, set base cases (0, 1).
    2. Iterate, set each value as sum of previous two.
  - **Context**: O(n) time, O(n) space.
- **Knapsack Problem**:
  - **Algorithm**:
    1. Create DP table for weights and values.
    2. For each item, choose max of including or excluding.
  - **Context**: O(n*W) time, O(n*W) space.
- **Longest Common Subsequence (LCS)**:
  - **Algorithm**:
    1. Create DP table for string lengths.
    2. If chars match, add 1 + diagonal, else max of left or up.
  - **Context**: O(mn) time, O(mn) space.
- **Longest Increasing Subsequence (LIS)**:
  - **Algorithm**:
    1. Use DP array, initialize with 1 for each element.
    2. For each pair, update with max length if increasing.
  - **Context**: O(nš) time, O(n) space.

- **Coin Change**:
  - **Algorithm**:
    1. Use DP array for amounts, initialize with infinity.
    2. For each coin, update min ways to make amount.
  - **Context**: O(amount*coins) time, O(amount) space.
- **Matrix Chain Multiplication**:
  - **Algorithm**:
    1. Create DP table for subproblem costs.
    2. Fill table with min cost of all splits.
  - **Context**: O(nş) time, O(nš) space.
- **Edit Distance**:
  - **Algorithm**:
    1. Use DP table, initialize with row/column indices.
    2. If chars match, copy diagonal, else min of left, up, diagonal + 1.
  - **Context**: O(mn) time, O(mn) space.
- **Partition Problem**:
  - **Algorithm**:
    1. Use DP for subset sums up to target.
    2. Check if target sum is achievable.
  - **Context**: O(n*sum) time, O(sum) space.

Common LeetCode Problems with Approaches

- **Climbing Stairs (70)**: Use DP for Fibonacci-like sequence.
- **Coin Change (322)**: Use tabulation for minimum coins.
- **Longest Increasing Subsequence (300)**: Use DP with length tracking.
-
- **Word Break (139)**: Use DP for substring matching.

Time & Space Complexities

- Varies by problem: O(n) to O(nş)
- Space: O(n) to O(nš)

Important Tips & Tricks

- Identify overlapping subproblems for DP applicability.
- Use memoization for top-down, tabulation for bottom-up.
- Optimize space with rolling arrays.
- Handle base cases carefully.
- Test with small inputs to verify DP logic.