Dynamic Programming Cheatsheet

A concise guide to Dynamic Programming concepts, techniques, and common patterns for algorithm design and interview preparation.

Core Concepts

Understanding DP

What is Dynamic Programming?

Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

Key Properties:

- **Optimal Substructure:** An optimal solution can be constructed from optimal solutions of its subproblems.
- **Overlapping Subproblems:** The problem can be broken down into subproblems which are reused several times.

DP vs Divide & Conquer:

Unlike Divide & Conquer (e.g., Merge Sort), which divides the problem into independent subproblems, DP is applicable when subproblems are not independent, and subproblems share subsubproblems.

Approaches

Top-Down (Memoization)	Start with the main problem and recursively solve subproblems. Store results of subproblems to avoid recomputation.
Bottom-Up (Tabulation)	Solve subproblems first and build up to the main problem. Store results in a table (array).
When to use which approach?	Memoization can be more intuitive, while tabulation can be more efficient due to reduced function call overhead.

Steps to Solve DP Problems

1. **Define the subproblem:** Clearly state what the subproblem is trying to compute.
2. **Identify the base cases:** Determine the simplest subproblems that can be solved directly.
3. **Write the recurrence relation:** Express the solution to a subproblem in terms of solutions to smaller subproblems.
4. **Implement the algorithm:** Use either memoization (top-down) or tabulation (bottom-up) to solve the problem efficiently.

Common DP Patterns

1D DP

Characteristics:
Involves a single changing variable, often the length of an array or a value within a range.

Example: Fibonacci Sequence

Compute the nth Fibonacci number.

Recurrence relation: `F(n) = F(n-1) + F(n-2)`
Base cases: `F(0) = 0, F(1) = 1`

```python
def fibonacci(n):
    dp = [0] * (n + 1)
    dp[0] = 0
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]
```

2D DP

Characteristics:
Involves two changing variables, often indices of two arrays or a 2D grid.

Example: Edit Distance

Compute the minimum number of operations (insert, delete, replace) to convert one string to another.

Recurrence relation: Based on whether the characters match or not.
Base cases: Empty strings.

```python
def edit_distance(s1, s2):
    dp = [[0] * (len(s2) + 1) for _ in range(len(s1) + 1)]
    for i in range(len(s1) + 1):
        dp[i][0] = i
    for j in range(len(s2) + 1):
        dp[0][j] = j
    for i in range(1, len(s1) + 1):
        for j in range(1, len(s2) + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])
    return dp[len(s1)][len(s2)]
```

Knapsack Pattern

0/1 Knapsack	Each item can either be included or excluded. Recurrence: `dp[i][w] = max(dp[i-1][w], value[i] + dp[i-1][w - weight[i]])`
Unbounded Knapsack	Each item can be included multiple times. Recurrence: `dp[i][w] = max(dp[i-1][w], value[i] + dp[i][w - weight[i]])`
Variations	Subset Sum, Partition Equal Subset Sum, etc. Often involve modifying the knapsack recurrence.

Optimization Techniques

Space Optimization

Reducing Space Complexity

In some DP problems, you only need the previous row or column to compute the current one. In these cases, you can reduce space complexity by using only two rows or columns instead of storing the entire table.

Example: Fibonacci (Space Optimized)

```
def fibonacci_space_optimized(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

Time Optimization

Avoiding Redundant Calculations

Memoization is a key technique to avoid recomputing the same subproblems. Ensure your base cases are correctly defined to prevent infinite recursion.

Careful Recurrence Design

A well-defined recurrence relation can significantly impact the time complexity. Consider alternative formulations that might lead to faster computation.

Bitmasking

When to Use	When the problem involves sets or subsets of elements, and the size of the set is relatively small (<= 20).
Representation	Represent a set as an integer, where the ith bit is 1 if the ith element is in the set, and 0 otherwise.
Example	Traveling Salesman Problem (TSP) variations, Set Cover Problem.

Practice Problems

Classic Problems

- **Longest Common Subsequence (LCS)**
- **Longest Increasing Subsequence (LIS)**
- **Coin Change Problem**
- **Rod Cutting Problem**

Medium Difficulty

- **Maximum Subarray Problem (Kadane's Algorithm)**
- **Word Break Problem**
- **Minimum Cost Path in a Grid**

Hard Difficulty

- **Regular Expression Matching**
- **Edit Distance with Constraints**
- **Matrix Chain Multiplication**

Tips for Interview Prep

- **Understand the Problem:** Clarify constraints and edge cases.
- **Explain Your Approach:** Articulate your thought process clearly.
- **Write Clean Code:** Pay attention to variable names, comments, and code structure.
- **Test Thoroughly:** Test with various inputs, including edge cases.
- **Analyze Complexity:** Determine time and space complexity.