Dynamic Programming Patterns: Complete Learning Guide

Part 1: Foundations

Chapter 1: Understanding Dynamic Programming

Core Concept: Breaking down problems into overlapping subproblems and storing their solutions to avoid recomputation.

Key Principles:

- Optimal Substructure: Optimal solution contains optimal solutions to subproblems
- Overlapping Subproblems: Same subproblems are solved multiple times
- Memoization (Top-Down) vs Tabulation (Bottom-Up)

When to Use DP:

- Problem asks for optimization (min/max/count)
- Problem can be broken into similar subproblems
- Decisions lead to subproblems with similar structure

Part 2: Fundamental Patterns

Chapter 2: Linear DP (1D)

Pattern: Problems where state depends only on previous states in a single sequence.

State Definition: `dp[i]` = solution for first i elements

Classic Problems:

- Climbing Stairs (each step depends on previous 1-2 steps)
- House Robber (rob or skip current house)
- Decode Ways (count decodings up to position i)
- Maximum Subarray (Kadane's Algorithm)

Recurrence Template:

```
dp[i] = function(dp[i-1], dp[i-2], ..., arr[i])
```

Chapter 3: Fibonacci-Style Pattern

Pattern: Current state depends on fixed number of previous states.

State Definition: `dp[i]` = answer for index i

Characteristics:

- Usually O(n) time, O(1) space optimizable
- Fixed lookback distance

Classic Problems:

- Fibonacci Numbers
- Tribonacci Numbers
- N-th Tribonacci Number
- Min Cost Climbing Stairs

Space Optimization: Keep only last k states instead of entire array.

Part 3: 2D Grid Patterns

Chapter 4: Grid Path Problems

Pattern: Finding paths in 2D grids with constraints.

State Definition: `dp[i][j]` = answer at cell (i, j)

Movement: Usually right/down or all four directions

Classic Problems:

- Unique Paths (count paths from top-left to bottom-right)
- Minimum Path Sum (find minimum cost path)
- Dungeon Game (minimum health needed)
- Unique Paths II (with obstacles)

Recurrence Template:

```
dp[i][j] = function(dp[i-1][j], dp[i][j-1], grid[i][j])
```

Chapter 5: Two-Sequence DP (String Matching)

Pattern: Comparing or combining two sequences.

State Definition: `dp[i][j]` = solution using first i elements of seq1 and first j elements of seq2

Classic Problems:

- Longest Common Subsequence (LCS)
- Edit Distance (Levenshtein)
- Distinct Subsequences
- Interleaving String
- Regular Expression Matching
- Wildcard Matching

Recurrence Template:

```
if (s1[i] == s2[j]):
    dp[i][j] = function(dp[i-1][j-1])
else:
    dp[i][j] = function(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
```

Part 4: Optimization Patterns

Chapter 6: 0/1 Knapsack Pattern

Pattern: Select items with constraints to optimize value.

State Definition: `dp[i][w]` = maximum value using first i items with weight limit w

Characteristics:

- Each item: take it or leave it (binary choice)
- Constraint: capacity/weight/size limit

Classic Problems:

- 0/1 Knapsack (original problem)
- Partition Equal Subset Sum
- Target Sum
- Last Stone Weight II
- Ones and Zeroes (2D knapsack)

Recurrence Template:

```
dp[i][w] = max(
    dp[i-1][w],              // don't take item i
    dp[i-1][w-weight[i]] + value[i]  // take item i
)
```

Space Optimization: Use 1D array, iterate backwards.

Chapter 7: Unbounded Knapsack Pattern

Pattern: Similar to 0/1 knapsack but items can be used unlimited times.

State Definition: `dp[i][w]` = maximum value with first i item types and capacity w

Classic Problems:

- Coin Change (minimum coins)
- Coin Change II (count ways)
- Perfect Squares (min squares summing to n)
- Minimum Cost For Tickets

Recurrence Template:

```
dp[i][w] = max(
    dp[i-1][w],              // don't take item i
    dp[i][w-weight[i]] + value[i]  // take item i (can use again)
)
```

Key Difference from 0/1: Use `dp[i]` instead of `dp[i-1]` when taking item.

Chapter 8: Longest Increasing Subsequence (LIS) Pattern

Pattern: Finding optimal subsequences with increasing/decreasing property.

State Definition: `dp[i]` = length of LIS ending at index i

Classic Problems:

- Longest Increasing Subsequence
- Longest Increasing Path in Matrix
- Number of LIS
- Russian Doll Envelopes
- Maximum Length of Pair Chain

Recurrence Template:

```
for i in range(n):
    for j in range(i):
        if arr[j] < arr[i]:
            dp[i] = max(dp[i], dp[j] + 1)
```

Advanced: O(n log n) solution using binary search with patience sorting.

Part 5: Advanced Patterns

Chapter 9: Interval DP

Pattern: Problems involving intervals/subarrays where solution depends on smaller intervals.

State Definition: `dp[i][j]` = solution for interval [i, j]

Iteration Order: By increasing interval length

Classic Problems:

- Longest Palindromic Subsequence
- Palindrome Partitioning II
- Burst Balloons
- Remove Boxes
- Minimum Cost Tree From Leaf Values
- Stone Game series

Recurrence Template:

```
for length in range(2, n+1):
    for i in range(n-length+1):
        j = i + length - 1
        for k in range(i, j+1):  # split point
            dp[i][j] = function(dp[i][k], dp[k+1][j])
```

Chapter 10: Partition DP

Pattern: Dividing array into subarrays/partitions optimally.

State Definition: `dp[i]` = optimal solution for first i elements

Classic Problems:

- Palindrome Partitioning II (min cuts)
- Partition Array for Maximum Sum
- Split Array Largest Sum
- Decode Ways II

Recurrence Template:

```
for i in range(n):
    for j in range(i):
        if is_valid(j, i):
            dp[i] = optimize(dp[i], dp[j] + cost[j:i])
```

Chapter 11: State Machine DP

Pattern: Problems with distinct states and transitions between states.

State Definition: `dp[i][state]` = optimal solution at position i in given state

Classic Problems:

- Best Time to Buy and Sell Stock (all variants)
- Best Time to Buy and Sell Stock with Cooldown
- Best Time to Buy and Sell Stock with Transaction Fee

States Example (Stock Problem):

- State 0: No stock, can buy
- State 1: Holding stock, can sell
- State 2: Just sold, cooldown

Recurrence Template:

```
dp[i][state0] = max(dp[i-1][state0], dp[i-1][state2])
dp[i][state1] = max(dp[i-1][state1], dp[i-1][state0] - price)
dp[i][state2] = dp[i-1][state1] + price
```

Chapter 12: Digit DP

Pattern: Counting numbers with specific digit properties in a range.

State Definition: `dp[pos][tight][other_states]`

- pos: current digit position
- tight: whether we're still bounded by the limit
- other_states: problem-specific (sum, count, etc.)

Classic Problems:

- Count Numbers with Unique Digits
- Numbers At Most N Given Digit Set
- Numbers With Repeated Digits
- Count Special Numbers

Approach:

1. Convert number to digits
2. Build number digit by digit
3. Track constraints during construction

Chapter 13: DP on Trees

Pattern: Computing optimal solutions on tree structures.

State Definition: `dp[node][state]` = solution for subtree rooted at node in given state

Approaches:

- DFS with memoization
- Post-order traversal

Classic Problems:

- House Robber III (rob tree nodes)
- Binary Tree Cameras
- Maximum Path Sum in Binary Tree
- Diameter of Binary Tree
- Longest Path with Different Adjacent Characters

Recurrence Template:

```
def dfs(node, parent):
    # Base case
    if not node:
        return base_value

    # Recurse on children
    left = dfs(node.left, node)
    right = dfs(node.right, node)

    # Combine results
    dp[node] = function(left, right, node.val)
    return dp[node]
```

Chapter 14: Bitmask DP

Pattern: Using bitmasks to represent states/subsets for optimization.

State Definition: `dp[mask]` or `dp[i][mask]` where mask represents a subset

Use Cases:

- Small set sizes (typically n ≤ 20)
- Need to track which elements are used
- Permutation/combination problems

Classic Problems:

- Traveling Salesman Problem (TSP)
- Shortest Path Visiting All Nodes
- Minimum Cost to Connect All Points
- Find the Shortest Superstring
- Number of Ways to Wear Different Hats

Bit Operations:

```
Set bit i: mask | (1 << i)
Clear bit i: mask & ~(1 << i)
Check bit i: mask & (1 << i)
Count set bits: bin(mask).count('1')
```

Chapter 15: Probability DP

Pattern: Computing probabilities or expected values.

State Definition: `dp[state]` = probability or expected value of reaching this state

Classic Problems:

- Knight Probability in Chessboard
- Soup Servings
- New 21 Game
- Minimum Number of Flips to Convert Binary Matrix

Approach:

- Forward DP: Propagate probabilities forward
- Backward DP: Calculate from end state

Recurrence Template:

```
dp[next_state] += dp[current_state] * probability
```

Part 6: Advanced Techniques

Chapter 16: DP with Data Structures

Pattern: Combining DP with auxiliary data structures for optimization.

Techniques:

- **Segment Tree**: Range queries with DP
- **Binary Indexed Tree**: Prefix sums in DP
- **Monotonic Queue/Stack**: Sliding window optimization
- **Priority Queue**: Selection of optimal subproblems

Classic Problems:

- Largest Rectangle in Histogram (Monotonic Stack)
- Sliding Window Maximum (Monotonic Queue)
- Russian Doll Envelopes (Binary Search + DP)

Chapter 17: Game Theory DP

Pattern: Two-player games with optimal strategies.

State Definition: `dp[state]` = true if current player can win from this state

Classic Problems:

- Stone Game (all variants)
- Predict the Winner
- Can I Win
- Nim Game variants

Minimax Principle:

- Maximize your score
- Minimize opponent's best move

Recurrence Template:

```
dp[state] = max over all moves (
    score_from_move - dp[resulting_state]
)
```

Chapter 18: DP Space Optimization Techniques

Technique 1: Rolling Array

- Keep only last k rows/states
- Reduces $O(n^2)$ to $O(n)$ space

Technique 2: In-Place Update

- Update DP array while iterating
- Careful about dependencies

Technique 3: State Compression

- Combine multiple states into one
- Use bitmasks or clever encoding

Technique 4: Implicit DP

- Calculate states on-the-fly

- Don't store all intermediate results

Part 7: Problem-Solving Framework

Chapter 19: How to Identify DP Problems

Red Flags:

1. Keywords: "maximize", "minimize", "count ways", "longest", "shortest"
2. Constraints: Small enough for exponential → polynomial
3. Optimal substructure: Can break into smaller similar problems
4. Overlapping subproblems: Same calculations repeated

Not DP:

- Greedy works (no need for exploring all options)
- No overlapping subproblems
- Online algorithms (streaming data)

Chapter 20: Step-by-Step DP Solution Process

Step 1: Identify if it's DP

- Check for optimal substructure
- Look for overlapping subproblems

Step 2: Define State

- What information do you need to solve subproblem?
- How many dimensions needed?

Step 3: Find Recurrence Relation

- How does current state relate to previous states?
- What are the choices/decisions?

Step 4: Determine Base Cases

- What are the simplest subproblems?
- What are boundary conditions?

Step 5: Decide Iteration Order

- Top-down (recursion + memoization) or bottom-up (tabulation)?
- What order ensures dependencies are computed first?

Step 6: Implement

- Start with brute force
- Add memoization
- Convert to tabulation if needed

Step 7: Optimize

- Space optimization
- Time complexity improvements

Practice Roadmap

Beginner Level

1. Climbing Stairs
2. House Robber
3. Maximum Subarray
4. Unique Paths
5. Minimum Path Sum

Intermediate Level

1. Longest Common Subsequence
2. Coin Change
3. Edit Distance
4. Longest Increasing Subsequence
5. Partition Equal Subset Sum

Advanced Level

1. Burst Balloons
2. Regular Expression Matching
3. Shortest Path Visiting All Nodes
4. Stone Game III
5. Minimum Cost Tree From Leaf Values

Expert Level

1. Count All Palindromic Subsequences
2. Profitable Schemes
3. Strange Printer
4. Find the Shortest Superstring
5. Number of Ways to Wear Different Hats

Key Takeaways

1. **Pattern Recognition**: Most problems fit known patterns
2. **State Design**: Most important and challenging step
3. **Start Simple**: Begin with recursive solution, then optimize
4. **Practice**: Pattern recognition improves with experience
5. **Optimization**: Space can often be reduced without affecting correctness

Remember: Dynamic Programming is about **breaking problems into subproblems** and **storing solutions to avoid recomputation**. Master the patterns, and you'll solve problems you've never seen before!