Volume 4: Dynamic Programming Mastery

Table of Contents

From Recursion to Tabulation - Pattern Recognition System

Target: Infosys L3 Specialist Programmer
Edition: 2025
Focus: Hard DP problems (35% of Q3 in coding round)

Preface

Dynamic Programming is the **highest-value skill** for Infosys L3:

- Q3 (Hard) is almost always DP or advanced graph
- DP mastery separates L2 from L3 candidates
- Pattern recognition is everything (not memorization)

FAST Method (from Dynamic Programming for Interviews[1]):

1. **First solution**: Recursive brute force
2. **Analyze**: Find overlapping subproblems
3. **Subproblems**: Define DP state clearly
4. **Turn around**: Convert to bottom-up

This volume: 12 core patterns covering 90% of DP problems.

Table of Contents

Chapter 1: Fibonacci & 1D DP

1.1 The DP Awakening

Problem: Compute Fibonacci(n)

Recursive (Brute Force):

```
int fib(int n) {
    if (n &lt;= 1) return n;
    return fib(n-1) + fib(n-2);
}
```

Time: O(2^n) — exponential explosion!

Why slow? Recomputing same values:

```
fib(5)
├─ fib(4)
│  ├─ fib(3)
│  │  ├─ fib(2)
│  │  │  ├─ fib(1)
│  │  │  └─ fib(0)
│  │  └─ fib(1)
│  └─ fib(2)  ← RECOMPUTED
└─ fib(3)     ← RECOMPUTED
```

1.2 Memoization (Top-Down DP)

Store results to avoid recomputation:

```
int fibMemo(int n, vector&lt;int&gt;&amp; memo) {
    if (n &lt;= 1) return n;
    if (memo[n] != -1) return memo[n]; // Cached

    memo[n] = fibMemo(n-1, memo) + fibMemo(n-2, memo);
    return memo[n];
}

int fib(int n) {
    vector&lt;int&gt; memo(n + 1, -1);
    return fibMemo(n, memo);
}
```

Time: O(n), **Space**: O(n)

1.3 Tabulation (Bottom-Up DP)

Iterative approach, no recursion:

```
int fib(int n) {
    if (n &lt;= 1) return n;

    vector&lt;int&gt; dp(n + 1);
    dp[0] = 0;
    dp[1] = 1;

    for (int i = 2; i &lt;= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

Space Optimized (only need last 2 values):

```
int fib(int n) {
    if (n &lt;= 1) return n;

    int prev2 = 0, prev1 = 1;
    for (int i = 2; i &lt;= n; i++) {
        int curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

Time: O(n), **Space**: O(1)

1.4 Problem: Climbing Stairs (Infosys Easy)

Statement: n stairs, can climb 1 or 2 steps. Count ways to reach top.

Input: n = 3
Output: 3 (ways: 1+1+1, 1+2, 2+1)

Recurrence: `ways(n) = ways(n-1) + ways(n-2)`

Why? To reach step n, we either:

- Came from step n-1 (1 step)
- Came from step n-2 (2 steps)

C++ Solution:

```
int climbStairs(int n) {
    if (n &lt;= 2) return n;

    int prev2 = 1, prev1 = 2;
    for (int i = 3; i &lt;= n; i++) {
        int curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

1.5 Problem: House Robber (Infosys Medium)

Statement: Rob houses in a row, can't rob adjacent houses. Maximize money.

Input: nums = [2, 7, 9, 3, 1]
Output: 12 (rob houses 0, 2, 4 → 2 + 9 + 1)

Recurrence:

$$dp[i] = \max(dp[i-1], nums[i] + dp[i-2]) \quad (1)$$

Why? At house i, either:

- Skip it → take dp[i-1]
- Rob it → nums[i] + dp[i-2] (can't rob i-1)

C++ Solution:

```
int rob(vector&lt;int&gt;&amp; nums) {
    int n = nums.size();
    if (n == 1) return nums[0];

    int prev2 = 0, prev1 = nums[0];
    for (int i = 1; i &lt; n; i++) {
        int curr = max(prev1, nums[i] + prev2);
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

Time: O(n), **Space**: O(1)

Chapter 2: Grid DP

2.1 Problem: Unique Paths (Infosys Medium)

Statement: m×n grid, robot at top-left, goal at bottom-right. Can only move right or down. Count paths.

Input: m = 3, n = 2
Output: 3

Recurrence:

$$dp[i][j] = dp[i-1][j] + dp[i][j-1] \qquad (2)$$

Why? To reach (i,j), we either came from:

- Above: (i-1, j)
- Left: (i, j-1)

C++ Solution:

```
int uniquePaths(int m, int n) {
    vector&lt;vector&lt;int&gt;&gt; dp(m, vector&lt;int&gt;(n, 1));

    for (int i = 1; i &lt; m; i++) {
        for (int j = 1; j &lt; n; j++) {
            dp[i][j] = dp[i-1][j] + dp[i][j-1];
        }
    }
    return dp[m-1][n-1];
}
```

Space Optimized (1D array):

```
int uniquePaths(int m, int n) {
    vector&lt;int&gt; dp(n, 1);

    for (int i = 1; i &lt; m; i++) {
        for (int j = 1; j &lt; n; j++) {
            dp[j] += dp[j-1];
        }
    }
    return dp[n-1];
}
```

Time: O(m×n), **Space**: O(n)

2.2 Problem: Minimum Path Sum (Infosys Medium)

Statement: m×n grid with costs, find path with minimum sum from top-left to bottom-right.

Input:

```
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
```

Output: 7 (path: 1→3→1→1→1)

Recurrence:

$$dp[i][j] = grid[i][j] + \min(dp[i-1][j], dp[i][j-1]) \quad (3)$$

C++ Solution:

```
int minPathSum(vector&lt;vector&lt;int&gt;&gt;&amp; grid) {
    int m = grid.size(), n = grid[0].size();

    // In-place DP (modify grid)
    for (int i = 1; i &lt; m; i++) {
        grid[i][0] += grid[i-1][0];
    }
    for (int j = 1; j &lt; n; j++) {
        grid[0][j] += grid[0][j-1];
    }

    for (int i = 1; i &lt; m; i++) {
        for (int j = 1; j &lt; n; j++) {
            grid[i][j] += min(grid[i-1][j], grid[i][j-1]);
        }
    }
    return grid[m-1][n-1];
}
```

Chapter 3: DP State Design Principles

3.1 The State Question

Most important decision: What does `dp[i]` or `dp[i][j]` represent?

Good state:

- Captures essential information
- Enables recurrence relation
- Avoids redundancy

Example States:

Problem	State	Meaning
Fibonacci	dp[i]	Fibonacci(i)
Climbing Stairs	dp[i]	Ways to reach step i
Knapsack	dp[i][w]	Max value using first i items, weight limit w
LCS	dp[i][j]	LCS length of s1[0..i-1], s2[0..j-1]

3.2 Dimensionality Analysis

How many dimensions?

1D DP: Single changing parameter

- Fibonacci (n)

- Climbing Stairs (n)
- House Robber (current house index)

2D DP: Two changing parameters

- Grid problems (row, column)
- Knapsack (item index, weight)
- String DP (index in s1, index in s2)

3D DP: Rare, complex

- LCS of 3 strings (index in s1, s2, s3)
- 3D grid problems

Chapter 4: Pattern 1 - 0/1 Knapsack Family

4.1 Core Problem: 0/1 Knapsack

Statement: Given weights, values, capacity W. Maximize value without exceeding weight. Each item used 0 or 1 times.

Input: wt = [1,3,4,5], val = [1,4,5,7], W = 7
Output: 9 (items with weights 3,4)

State: `dp[i][w]` = max value using first i items, weight limit w

Recurrence:

$$dp[i][w] = \begin{cases} dp[i-1][w] & amp; \text{if } wt[i-1] \quad gt; w \\ \max(dp[i-1][w], val[i-1] + dp[i-1][w - wt[i-1]]) & amp; \text{otherwise} \end{cases} \tag{4}$$

C++ Solution:

```
int knapsack(vector&lt;int&gt;&amp; wt, vector&lt;int&gt;&amp; val, int W) {
    int n = wt.size();
    vector&lt;vector&lt;int&gt;&gt; dp(n + 1, vector&lt;int&gt;(W + 1, 0));

    for (int i = 1; i &lt;= n; i++) {
        for (int w = 1; w &lt;= W; w++) {
            if (wt[i-1] &lt;= w) {
                dp[i][w] = max(dp[i-1][w],
                               val[i-1] + dp[i-1][w - wt[i-1]]);
            } else {
                dp[i][w] = dp[i-1][w];
            }
        }
    }
    return dp[n][W];
}
```

Space Optimized (2D → 1D):

```
int knapsack(vector&lt;int&gt;&amp; wt, vector&lt;int&gt;&amp; val, int W) {
    int n = wt.size();
    vector&lt;int&gt; dp(W + 1, 0);

    for (int i = 0; i &lt; n; i++) {
        for (int w = W; w &gt;= wt[i]; w--) { // REVERSE!
            dp[w] = max(dp[w], val[i] + dp[w - wt[i]]);
        }
```

```
    }
    return dp[W];
}
```

Why reverse? Prevents using same item twice (we only want previous row data).

4.2 Variation: Subset Sum

Problem: Can we select subset with sum = target?

Input: nums = [3, 34, 4, 12, 5, 2], target = 9
Output: true (4 + 5)

State: `dp[i][s]` = can we make sum s using first i elements?

Recurrence: Same as knapsack (weights = values = nums)

C++ Solution:

```
bool canPartition(vector&lt;int&gt;&amp; nums, int target) {
    int n = nums.size();
    vector&lt;bool&gt; dp(target + 1, false);
    dp[0] = true; // Empty subset

    for (int num : nums) {
        for (int s = target; s &gt;= num; s--) {
            dp[s] = dp[s] || dp[s - num];
        }
    }
    return dp[target];
}
```

4.3 Variation: Partition Equal Subset Sum (Infosys Medium)

Problem: Can we partition array into two subsets with equal sum?

Input: nums = [1, 5, 11, 5]
Output: true (partitions: [1,5,5] and [11])

Insight: If total sum is odd → impossible. If even, find subset with sum = total/2.

C++ Solution:

```
bool canPartition(vector&lt;int&gt;&amp; nums) {
    int sum = accumulate(nums.begin(), nums.end(), 0);
    if (sum % 2 != 0) return false;

    int target = sum / 2;
    vector&lt;bool&gt; dp(target + 1, false);
    dp[0] = true;

    for (int num : nums) {
        for (int s = target; s &gt;= num; s--) {
            dp[s] = dp[s] || dp[s - num];
        }
    }
    return dp[target];
}
```

Chapter 5: Pattern 2 - Unbounded Knapsack

5.1 Core Problem: Coin Change (Minimum Coins)

Problem: Given coins, find minimum coins to make amount.

Input: coins = [1, 2, 5], amount = 11
Output: 3 (5 + 5 + 1)

State: `dp[a]` = min coins to make amount a

Recurrence:

$$dp[a] = \min_{c \in \text{coins}} (dp[a - c] + 1) \tag{5}$$

C++ Solution:

```
int coinChange(vector&lt;int&gt;&amp; coins, int amount) {
    vector&lt;int&gt; dp(amount + 1, INT_MAX);
    dp[0] = 0;

    for (int a = 1; a &lt;= amount; a++) {
        for (int c : coins) {
            if (a &gt;= c &amp;&amp; dp[a - c] != INT_MAX) {
                dp[a] = min(dp[a], dp[a - c] + 1);
            }
        }
    }
    return dp[amount] == INT_MAX ? -1 : dp[amount];
}
```

Time: O(amount × coins), **Space**: O(amount)

5.2 Variation: Coin Change (Count Ways)

Problem: Count ways to make amount.

Input: coins = [1, 2, 5], amount = 5
Output: 4 (ways: 5, 2+2+1, 2+1+1+1, 1+1+1+1+1)

Recurrence:

$$dp[a] = \sum_{c \in \text{coins}} dp[a - c] \tag{6}$$

C++ Solution:

```
int change(int amount, vector&lt;int&gt;&amp; coins) {
    vector&lt;int&gt; dp(amount + 1, 0);
    dp[0] = 1; // One way to make 0

    for (int c : coins) {
        for (int a = c; a &lt;= amount; a++) {
            dp[a] += dp[a - c];
        }
    }
    return dp[amount];
}
```

Critical: Iterate coins in outer loop to avoid counting permutations.

Chapter 6: Pattern 3 - String DP (LCS Family)

6.1 Longest Common Subsequence (Infosys Hard)

Problem: Find length of LCS of two strings.

Input: s1 = "abcde", s2 = "ace"
Output: 3 ("ace")

State: `dp[i][j]` = LCS of s1[0..i-1], s2[0..j-1]

Recurrence:

$$dp[i][j] = \begin{cases} 1 + dp[i-1][j-1] & amp; \text{if } s1[i-1] = s2[j-1] \\ \max(dp[i-1][j], dp[i][j-1]) & amp; \text{otherwise} \end{cases} \quad (7)$$

C++ Solution:

```
int longestCommonSubsequence(string s1, string s2) {
    int m = s1.length(), n = s2.length();
    vector&lt;vector&lt;int&gt;&gt; dp(m + 1, vector&lt;int&gt;(n + 1, 0));

    for (int i = 1; i &lt;= m; i++) {
        for (int j = 1; j &lt;= n; j++) {
            if (s1[i-1] == s2[j-1]) {
                dp[i][j] = 1 + dp[i-1][j-1];
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }
    return dp[m][n];
}
```

Python Solution:

```
def longestCommonSubsequence(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i-1] == s2[j-1]:
                dp[i][j] = 1 + dp[i-1][j-1]
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]
```

6.2 Edit Distance (Infosys Hard)

Problem: Minimum operations (insert, delete, replace) to convert s1 to s2.

Input: s1 = "horse", s2 = "ros"
Output: 3 (horse → rorse → rose → ros)

Recurrence:

$$dp[i][j] = \begin{cases} dp[i-1][j-1] & amp; \text{if } s1[i-1] = s2[j-1] \\ 1 + \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) & amp; \text{otherwise} \end{cases} \quad (8)$$

C++ Solution:

```
int minDistance(string s1, string s2) {
    int m = s1.length(), n = s2.length();
    vector&lt;vector&lt;int&gt;&gt; dp(m + 1, vector&lt;int&gt;(n + 1));

    // Base cases
    for (int i = 0; i &lt;= m; i++) dp[i][0] = i;
    for (int j = 0; j &lt;= n; j++) dp[0][j] = j;

    for (int i = 1; i &lt;= m; i++) {
        for (int j = 1; j &lt;= n; j++) {
            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],    // Delete
                                    dp[i][j-1],    // Insert
                                    dp[i-1][j-1]}); // Replace
            }
        }
    }
    return dp[m][n];
}
```

Chapter 7: Pattern 4 - Longest Increasing Subsequence

7.1 LIS (O(n²) DP)

Problem: Find length of longest increasing subsequence.

Input: nums = [10, 9, 2, 5, 3, 7, 101, 18]
Output: 4 ([2, 3, 7, 101])

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

Recurrence:

C++ Solution:

```
int lengthOfLIS(vector&lt;int&gt;&amp; nums) {
    int n = nums.size();
    vector&lt;int&gt; dp(n, 1); // Each element is LIS of length 1

    for (int i = 1; i &lt; n; i++) {
        for (int j = 0; j &lt; i; j++) {
            if (nums[j] &lt; nums[i]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
```

```
        }
    }
    return *max_element(dp.begin(), dp.end());
}
```

Time: O(n²)

7.2 LIS (O(n log n) Binary Search)

Optimized approach using patience sorting:

```
int lengthOfLIS(vector&lt;int&gt;&amp; nums) {
    vector&lt;int&gt; tail; // tail[i] = smallest ending value of LIS of length i+1

    for (int num : nums) {
        auto it = lower_bound(tail.begin(), tail.end(), num);
        if (it == tail.end()) {
            tail.push_back(num);
        } else {
            *it = num;
        }
    }
    return tail.size();
}
```

Time: O(n log n)

Chapter 8: Interview Strategy

8.1 State Space Reduction (2D → 1D)

When possible: If `dp[i]` only depends on `dp[i-1]`, use 1D array.

Example: Knapsack

2D → 1D transformation:

```
// 2D (O(n×W) space):
for (int i = 1; i &lt;= n; i++) {
    for (int w = 1; w &lt;= W; w++) {
        dp[i][w] = max(dp[i-1][w], val[i-1] + dp[i-1][w - wt[i-1]]);
    }
}

// 1D (O(W) space):
for (int i = 0; i &lt; n; i++) {
    for (int w = W; w &gt;= wt[i]; w--) { // REVERSE
        dp[w] = max(dp[w], val[i] + dp[w - wt[i]]);
    }
}
```

Key: Reverse iteration prevents overwriting needed values.

8.2 Common Mistakes

Mistake 1: Wrong Base Case

```
// WRONG (fails for all-negative arrays):
int maxSoFar = INT_MIN;

// CORRECT:
int maxSoFar = nums[0];
```

Mistake 2: Off-by-One in State

```
// WRONG (s1[i] vs dp[i+1] mismatch):
if (s1[i] == s2[j]) ...

// CORRECT:
if (s1[i-1] == s2[j-1]) ... // dp[i] represents s1[0..i-1]
```

Mistake 3: Forgetting Space Optimization Direction

```
// WRONG (forward iteration overwrites needed values):
for (int w = wt[i]; w &lt;= W; w++)

// CORRECT:
for (int w = W; w &gt;= wt[i]; w--)
```

Practice Problem Set (50 DP Problems)

Easy (15 problems)

1. Climbing Stairs
2. Min Cost Climbing Stairs
3. Fibonacci Number
4. N-th Tribonacci Number
5. House Robber

Medium (25 problems)

6. House Robber II
7. Unique Paths
8. Minimum Path Sum
9. Coin Change
10. Partition Equal Subset Sum
11. Longest Common Subsequence
12. Longest Increasing Subsequence
13. Decode Ways
14. Word Break
15. Maximum Product Subarray

Hard (10 problems - Infosys L3 Level)

16. Edit Distance
17. Regular Expression Matching
18. Wildcard Matching
19. Burst Balloons
20. Longest Valid Parentheses
21. Distinct Subsequences
22. Interleaving String
23. Scramble String
24. Maximum Rectangle
25. Best Time to Buy Sell Stock IV

References

[1] Byte by Byte. (2019). Dynamic Programming for Interviews. https://www.byte-by-byte.com/wp-content/uploads/2019/04/Dynamic-Programming-for-Interviews.pdf

[2] DesignGurus. (2024). Grokking Dynamic Programming Patterns. https://www.designgurus.io/course/grokking-dynamic-programming

[3] CLRS. (2009). *Introduction to Algorithms* (3rd ed.). Chapter 15: Dynamic Programming.