Infosys SP/DSE Dynamic Programming Encyclopedia

Table of Contents

Volume E: From Recursion to Mastery - All DP Patterns

Target: Infosys L3/SP/DSE Question 3 (Hard DP)
Edition: 2025
Weightage: 35% of Coding Round
Difficulty: Codeforces 2400-3000

Preface

Question 3 is **almost always Dynamic Programming**. This is where L3 candidates are separated from L2.

Reality Check:

- Q3 "Hard" = Codeforces 2600-3000 (Master/Grandmaster level)
- **Partial credit matters**: 50% test cases = competitive score
- 70-90 minutes allocated

This volume provides:

- 15 core DP patterns
- Every Infosys DP problem type
- Template-based approach for instant recognition

Table of Contents

PART I: DP FUNDAMENTALS

Chapter 1: The DP Transformation

1.1 Problem: Fibonacci

Naive Recursion:

```
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 subproblems:

```
fib(5)
├─ fib(4)
│   ├─ fib(3)      ← Computed
```

```
│   │   ├─ fib(2)
│   │   └─ fib(1)
│   └─ fib(2)      ← RECOMPUTED
└─ fib(3)          ← RECOMPUTED AGAIN
```

1.2 Memoization (Top-Down DP)

Store computed results:

```
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) — each subproblem computed once
Space: O(n) — memo array + recursion stack

1.3 Tabulation (Bottom-Up DP)

Iterative, 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];
}
```

Time: O(n)
Space: O(n)

1.4 Space Optimization

Observation: 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) ✓

Chapter 2: State Design Principles

2.1 The Most Important Question

What does `dp[i]` or `dp[i][j]` represent?

Good state design:

- Captures all necessary information
- Enables clear recurrence relation
- Minimizes dimensions

2.2 Common State Patterns

Problem Type	State	Meaning
Fibonacci	`dp[i]`	Fibonacci(i)
Climbing Stairs	`dp[i]`	Ways to reach step i
House Robber	`dp[i]`	Max money robbing houses 0..i
0/1 Knapsack	`dp[i][w]`	Max value using items 0..i-1, capacity w
LCS	`dp[i][j]`	LCS length of s1[0..i-1], s2[0..j-1]
Edit Distance	`dp[i][j]`	Min operations to convert s1[0..i-1] to s2[0..j-1]

2.3 Dimensionality Decision Tree

```
How many changing parameters?
│
├─ One parameter (e.g., index i)
│   └─ Use 1D DP: dp[i]
│       Examples: Fibonacci, Climbing Stairs, House Robber
│
├─ Two parameters (e.g., index i, capacity w)
│   └─ Use 2D DP: dp[i][j]
│       Examples: Knapsack, LCS, Edit Distance
│
└─ Three+ parameters
    └─ Use 3D+ DP or optimize to 2D
        Examples: LCS of 3 strings, 3D grid
```

PART II: 1D DP PATTERNS

Chapter 4: Fibonacci Variants

4.1 Climbing Stairs

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

Recurrence:

$$dp[i] = dp[i-1] + dp[i-2] \tag{1}$$

Why? To reach step i, either:

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

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;
}
```

4.2 House Robber

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

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

Recurrence:

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

Why? At house i, either:

- Skip it → take dp[i-1]
- Rob it → nums[i] + dp[i-2]

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)

4.3 House Robber II (Circular)

Problem: Houses in circle, can't rob first and last together.

Approach: Solve twice:

1. Rob houses 0 to n-2 (exclude last)
2. Rob houses 1 to n-1 (exclude first)

Solution:

```
int robLinear(vector&lt;int&gt;&amp; nums, int start, int end) {
    int prev2 = 0, prev1 = 0;

    for (int i = start; i &lt;= end; i++) {
        int curr = max(prev1, nums[i] + prev2);
        prev2 = prev1;
        prev1 = curr;
    }

    return prev1;
}

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

    return max(robLinear(nums, 0, n-2),
               robLinear(nums, 1, n-1));
}
```

Chapter 5: Longest Increasing Subsequence (LIS)

5.1 $O(n^2)$ DP Solution

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:

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: $O(n^2)$
Space: $O(n)$

5.2 O(n log n) Binary Search Solution

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)
Space: O(n)

PART III: 2D DP PATTERNS

Chapter 8: 0/1 Knapsack

8.1 Classic Problem

Input:

- `weights[]`: Weight of each item
- `values[]`: Value of each item
- `W`: Knapsack capacity

Output: Maximum value without exceeding weight.

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

Recurrence:

$$dp[i][w] = \begin{cases} dp[i-1][w] & amp; \text{if } weights[i-1] \quad gt; w \\ \max(dp[i-1][w], values[i-1] + dp[i-1][w - weights[i-1]]) & amp; \text{otherwise} \end{cases} \tag{3}$$

8.2 2D Solution

```
int knapsack(vector&lt;int&gt;&amp; weights, vector&lt;int&gt;&amp; values, int W) {
    int n = weights.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++) {
            // Option 1: Don't include item i-1
            dp[i][w] = dp[i-1][w];

            // Option 2: Include item i-1 (if fits)
            if (weights[i-1] &lt;= w) {
                dp[i][w] = max(dp[i][w],
                              values[i-1] + dp[i-1][w - weights[i-1]]);
            }
        }
    }
```

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

Time: O(n × W)
Space: O(n × W)

8.3 1D Space-Optimized Solution

Key: Process weights in **reverse** to avoid overwriting needed values.

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

    for (int i = 0; i &lt; n; i++) {
        // REVERSE iteration is critical!
        for (int w = W; w &gt;= weights[i]; w--) {
            dp[w] = max(dp[w], values[i] + dp[w - weights[i]]);
        }
    }

    return dp[W];
}
```

Time: O(n × W)
Space: O(W) ✓

Why reverse? Forward iteration would allow using same item multiple times (unbounded knapsack).

8.4 Knapsack Variants

Variant 1: Subset Sum

Problem: Can we select subset with sum = target?

Solution: Same as knapsack with `weights = values = nums`.

```
bool canPartition(vector&lt;int&gt;&amp; nums, int target) {
    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];
}
```

Variant 2: Partition Equal Subset Sum

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

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

```
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 9: Unbounded Knapsack

9.1 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)

Recurrence:

$$dp[a] = \min_{c \in coins} (dp[a-c] + 1) \tag{4}$$

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)

9.2 Coin Change (Count Ways)

Problem: Count ways to make amount.

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

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

```
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) { // Outer loop on coins!
        for (int a = c; a &lt;= amount; a++) {
            dp[a] += dp[a - c];
        }
    }
```

```
    return dp[amount];
}
```

Chapter 10: Longest Common Subsequence (LCS)

10.1 Classic LCS

Problem: Find length of LCS of two strings.

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

Recurrence:

$$dp[i][j] = \begin{cases} 1 + dp[i-1][j-1] & amp; \text{if } s1[i-1] = s2[j-1] \\ \max(dp[i-1][j], dp[i][j-1]) & amp; \text{otherwise} \end{cases} \tag{5}$$

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];
}
```

Time: O(m × n)
Space: O(m × n)

Chapter 11: Edit Distance

11.1 Problem

Input: `word1 = "horse", word2 = "ros"`
Output: `3` (horse → rorse → rose → ros)

Operations: Insert, Delete, Replace

Recurrence:

$$dp[i][j] = \begin{cases} dp[i-1][j-1] & amp; \text{if } s1[i-1] = s2[j-1] \\ 1 + \min \begin{cases} dp[i-1][j] & amp; \text{(delete)} \\ dp[i][j-1] & amp; \text{(insert)} \\ dp[i-1][j-1] & amp; \text{(replace)} \end{cases} & amp; \text{otherwise} \end{cases} \tag{6}$$

Solution:

```
int minDistance(string word1, string word2) {
    int m = word1.length(), n = word2.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 (word1[i-1] == word2[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];
}
```

References

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

[2] Dynamic Programming for Interviews (Byte by Byte, 2019)

[3] LeetCode DP Tag Problems (2025)