DYNAMIC PROGRAMMING PATTERN STUDY GUIDE

A Complete Handbook for Interview Mastery

> Transform from beginner to FAANG-ready DP master in 30 days

---

DOCUMENT PURPOSE

This is a professional algorithm handbook designed to systematically teach Dynamic Programming through:

- **10 Core Patterns** with mathematical rigor
- **State Design Framework** for any DP problem
- **60+ Curated Problems** organized by pattern
- **30-Day Study Roadmap** for structured learning
- **Mental Models** for intuitive understanding

Output Format: Recurrence relations, state definitions, pseudocode templates **NOT Included**: Full C++ implementations (focus on understanding, not memorization)

---

1. THEORETICAL FOUNDATION

1.1 What is Dynamic Programming?

Dynamic Programming is an algorithmic paradigm solving optimization problems by:

1. **Decomposing** into overlapping subproblems
2. **Solving** each subproblem once
3. **Storing** solutions to avoid recomputation
4. **Building** solutions bottom-up or top-down

Key Insight: Trade space for time to reduce exponential complexity to polynomial.

The DP Formula

```
Optimization Problem
+ Optimal Substructure
+ Overlapping Subproblems
= Dynamic Programming Solution
```

1.2 Why Greedy and Pure Recursion Fail

Pure Recursion Limitations

Example: Fibonacci

```
fib(5) calls fib(4) + fib(3)
fib(4) calls fib(3) + fib(2)
fib(3) called TWICE ⟶ Exponential waste!

Recurrence: T(n) = T(n-1) + T(n-2)
Complexity: O(φ^n) where φ = golden ratio ≈ 1.618
```

Problems:

- Repeated subproblem computation
- Exponential time complexity
- Stack overflow for large inputs

Greedy Algorithm Limitations

Counterexample: Coin change with coins {1, 3, 4} for amount 6

```
Greedy: Always pick largest coin
 6 ⟶ 4 (rem 2) ⟶ 1 (rem 1) ⟶ 1 = 3 coins ❌

Optimal:
 6 ⟶ 3 + 3 = 2 coins ✓
```

Why greedy fails: Local optimum ≠ Global optimum

1.3 Overlapping Subproblems

Definition: Same subproblem solved multiple times in recursive decomposition.

Detection Method: Draw recursion tree

```
If parameters (n, k, ...) appear multiple times ⟶ Overlap exists
```

Contrast with Divide & Conquer:

- Merge Sort: Disjoint subarrays ⟶ No overlap
- Fibonacci: fib(3) computed many times ⟶ Overlap!

1.4 Optimal Substructure

Definition: Optimal solution constructed from optimal solutions of subproblems.

Mathematical Form:

$$OPT(P) = \text{combine}(OPT(P_1), OPT(P_2), ..., OPT(P_k))$$
where P decomposes into subproblems $P_1, P_2, ..., P_k$

Test for Optimal Substructure:

If optimal path A⟶C goes through B,
then subpaths A⟶B and B⟶C must also be optimal

Examples:

- ✓ Shortest Path: Has optimal substructure
- ✗ Longest Simple Path: Does NOT (subpaths may need revisiting nodes)

1.5 Bellman's Principle of Optimality

> *"An optimal policy has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."*

Translation: Every sub-path of an optimal path is optimal.

Mathematical Expression:

$$V(s) = \max/\min \text{ over actions } a \{ \text{reward}(s, a) + \gamma \cdot V(\text{next\_state}(s, a)) \}$$

1.6 DP vs Divide & Conquer vs Greedy

Feature	Divide & Conquer	Dynamic Programming	Greedy
Subproblems	Disjoint	Overlapping	N/A
Approach	Recursive split	Memoize/Tabulate	Local choice
Guarantees	Correctness	Optimal (if applicable)	May not be optimal
Examples	Merge Sort, Binary Search	LCS, Knapsack	Huffman, Dijkstra

1.7 Memoization vs Tabulation

Memoization (Top-Down DP)

Approach: Recursive with caching

Template:

```
memo = {}

function dp(state):
    if state in memo:
        return memo[state]

    if base_case(state):
        return base_value

    result = combine(dp(subproblem1), dp(subproblem2), ...)
    memo[state] = result
    return result
```

Pros: Intuitive, only computes needed states **Cons**: Recursion overhead, stack overflow risk

Tabulation (Bottom-Up DP)

Approach: Iterative table filling

Template:

```
dp = initialize_table()
dp[base_cases] = base_values

for state in topological_order:
    dp[state] = combine(dp[previous_states])

return dp[final_state]
```

Pros: No recursion, faster, predictable memory **Cons**: Must compute all states, less intuitive

---

2. MATHEMATICAL MODEL OF DP

2.1 State Space

State: Complete specification of problem configuration

Formal Definition:

```
State S = (p₁, p₂, ..., pₖ)
where pᵢ uniquely identifies a subproblem instance
```

State Space: Set of all possible states Σ

Examples:

```
Fibonacci: S = {n} where n ∈ [0, N]
Knapsack: S = {(i, w)} where i ∈ [0,n], w ∈ [0,W]
LCS: S = {(i, j)} where i ∈ [0,m], j ∈ [0,n]
TSP: S = {(mask, city)} where mask ⊆ {0,1}ⁿ, city ∈ [0,n)
```

2.2 Recurrence Relations

General Form:

```
dp[state] = optimal_function({
   combine(dp[prev_state], cost_of_transition)
   for each valid prev_state
})
```

Components:

1. **Current State**: dp[state]
2. **Previous States**: States that state depends on
3. **Transition Function**: How to compute current from previous
4. **Optimal Function**: min, max, sum, count, etc.

Construction Process:

```
Step 1: Identify last decision/action
Step 2: Express current state using previous states
Step 3: Apply optimal substructure
Step 4: Add base cases
```

Example: Climbing Stairs

```
Problem: n stairs, climb 1 or 2 steps at a time

Last step: Either 1-step from (n-1) or 2-step from (n-2)

Recurrence:
dp[n] = dp[n-1] + dp[n-2]

Base cases:
dp[0] = 1 (one way: don't climb)
dp[1] = 1 (one way: single step)
```

2.3 State Transition Graph (DAG View)

Key Insight: Every DP problem is implicitly a DAG

Graph Components:

- **Vertices V**: All possible states
- **Edges E**: Valid transitions between states
- **Weights w(u,v)**: Cost/value of transition u→v
- **Source**: Initial state(s)
- **Target**: Final state (answer)

DP Solution = Optimal Path in DAG

Example: Coin Change

```
States: remaining amounts [0, target]
Edges: using a coin reduces amount
Weights: +1 (one more coin)
Goal: Shortest path from target to 0
```

2.4 Topological Ordering

For Tabulation: Process states such that all dependencies are computed first

Common Orderings:

```
Linear 1D: i = 0, 1, 2, ..., n
Grid 2D: row-by-row or column-by-column
Interval: by increasing length
Reverse: i = n, n-1, ..., 0 (for some patterns)
```

Why it matters: Ensures dp[prev_state] available when computing dp[state]

2.5 Complexity Analysis

Time Complexity

Formula:

```
T(n) = (Number of States) × (Time per State Computation)
```

Examples:

```
Fibonacci:
  States: n
  Time per state: O(1)
  Total: O(n)

LCS:
  States: m × n
  Time per state: O(1)
  Total: O(m·n)

Matrix Chain:
  States: n²
  Time per state: O(n) (trying split points)
  Total: O(n³)

TSP (DP):
  States: 2ⁿ · n
  Time per state: O(n)
  Total: O(2ⁿ · n²)
```

Space Complexity

Formula:

```
S(n) = Number of States Stored
```

Optimization Opportunity:

```
If dp[i] depends only on dp[i-1], dp[i-2], ..., dp[i-k]:
  → Use rolling array
  → Reduce from O(n) to O(k)
```

---

3. DP PATTERN TAXONOMY

Overview of 10 Core Patterns

#	Pattern	State	Time	Problems
1	1D Linear	dp[i]	O(n)	Fibonacci, House Robber
2	2D Grid	dp[i][j]	O(m·n)	Unique Paths, LCS
3	Subsequence	dp[i] or dp[i][j]	$O(n^2)$	LIS, LCS
4	Knapsack	dp[i][w]	O(n·W)	0/1 Knapsack, Subset Sum
5	Interval	dp[i][j]	$O(n^3)$	Matrix Chain, Burst Balloons
6	Partition	dp[i] or dp[i][k]	$O(n^2)$	Word Break, Palindrome Cut
7	Tree	dp[node][state]	O(n)	House Robber III
8	Digit	dp[pos][tight][...]	O(digits·states)	Digit counting
9	Bitmask	dp[mask][i]	$O(2^n \cdot n^2)$	TSP, Subset problems
10	Graph	dp[node][k]	O(V·E·k)	K-steps shortest path

Pattern 1: 1D Linear DP

Definition

Sequential decision making on array/string where `dp[i]` represents optimal solution for prefix/position i.

When to Use

- Problems on single sequence
- Decision at each position
- Current state depends on small number of previous positions
- Keywords: "first i elements", "up to position i"

State Design

```
dp[i] = optimal value considering elements [0..i] or position i
```

Recurrence Template

```
dp[i] = optimal_function(
    dp[i-1],            // previous position
    dp[i-2],            // two positions back
    f(arr[i], dp[...])    // include current element
)
```

Transition Complexity

O(1) to O(i) per state

Examples

House Robber:

```
dp[i] = max money robbing houses [0..i]

dp[i] = max(
    dp[i-1],          // skip house i
    dp[i-2] + arr[i]   // rob house i
)

Base: dp[0] = arr[0], dp[1] = max(arr[0], arr[1])
```

Decode Ways:

```
dp[i] = number of ways to decode s[0..i-1]

dp[i] = (s[i-1] valid single digit ? dp[i-1] : 0)
      + (s[i-2:i] valid two digits ? dp[i-2] : 0)

Base: dp[0] = 1
```

Space Optimization

```
// From dp[n] to O(1)
prev2 = dp[0]
prev1 = dp[1]

for i from 2 to n:
    curr = f(prev1, prev2)
```

```
    prev2 = prev1
    prev1 = curr
```

Pattern 2: 2D Grid DP

Definition

States form 2D table where position (i,j) represents grid location or comparison of two sequences.

When to Use

- Problems on 2D grid/matrix
- Comparing two sequences
- Two independent dimensions

State Design

```
// Grid problems
dp[i][j] = optimal value at grid position (i, j)

// Two sequences
dp[i][j] = optimal for s1[0..i-1] and s2[0..j-1]
```

Recurrence Template

```
dp[i][j] = optimal_function(
    dp[i-1][j],      // from top
    dp[i][j-1],      // from left
    dp[i-1][j-1],    // from diagonal
    grid[i][j]       // current cell
)
```

Examples

Unique Paths:

```
dp[i][j] = number of paths to reach (i, j)

dp[i][j] = dp[i-1][j] + dp[i][j-1]

Base: dp[0][j] = 1, dp[i][0] = 1
```

Longest Common Subsequence:

```
dp[i][j] = LCS length of s1[0..i-1] and s2[0..j-1]

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

Space Optimization

```
// From dp[m][n] to dp[n]
prev = array[n+1]
curr = array[n+1]

for i from 1 to m:
    for j from 1 to n:
        curr[j] = f(prev[j], curr[j-1], prev[j-1])
    swap(prev, curr)
```

Pattern 3: Subsequence DP

Definition

Finding optimal subsequences (non-contiguous) preserving relative order.

When to Use

- Keyword: "subsequence" (not substring!)
- Can skip elements
- Order matters
- Comparing sequences

State Design

```
// Single sequence
dp[i] = optimal subsequence ending at or within [0..i]

// Two sequences
dp[i][j] = optimal for A[0..i-1] and B[0..j-1]
```

Key Patterns

Longest Increasing Subsequence (LIS):

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

dp[i] = max(dp[j] + 1) for all j < i where arr[j] < arr[i]

Base: dp[i] = 1 for all i
Answer: max(dp[i])

Time: O(n²), Space: O(n)
```

LIS Optimization (Binary Search):

```
tail[len] = smallest ending element of LIS of length len+1

for x in arr:
    pos = binary_search_lower_bound(tail, x)
    tail[pos] = x
    if pos == len:
        len++

Time: O(n log n), Space: O(n)
```

Longest Common Subsequence (LCS):

```
dp[i][j] = LCS of s1[0..i-1] and s2[0..j-1]

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

Time: O(m·n), Space: O(m·n) → O(min(m,n))
```

Pattern 4: Knapsack DP

Definition

Selecting items with constraints to optimize value.

When to Use

- "Choose items" with weight/capacity limit
- "Maximize value" with constraint
- "Can each item be used?" → 0/1 vs unbounded

State Design

```
dp[i][w] = max value using first i items with capacity w
```

Recurrence Templates

0/1 Knapsack (each item at most once):

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

Condition: only if w >= wt[i]
Base: dp[0][w] = 0 for all w
Answer: dp[n][W]
```

Unbounded Knapsack (unlimited items):

```
dp[i][w] = max(
    dp[i-1][w],                 // don't use item i anymore
    dp[i][w-wt[i]] + val[i]     // use item i again
)

Note: Second term uses dp[i][...], not dp[i-1][...]
```

Space Optimization (0/1 Knapsack)

```
dp = array[W+1] initialized to 0

for i from 0 to n-1:
    for w from W down to wt[i]:   // REVERSE!
        dp[w] = max(dp[w], dp[w-wt[i]] + val[i])

return dp[W]
```

Why reverse? Prevents using same item multiple times in one iteration.

Variations

Subset Sum:

```
dp[sum] = true if sum achievable

for x in arr:
   for s from target down to x:
      dp[s] = dp[s] OR dp[s-x]
```

Coin Change (Min Coins):

```
dp[amount] = min coins for amount

dp[amt] = min over coins c {
   dp[amt - c] + 1
}

Base: dp[0] = 0
Invalid: dp[amt] = ∞
```

Coin Change 2 (Count Ways):

```
dp[amt] = number of ways to make amount

for coin in coins:
   for amt from coin to target:
      dp[amt] += dp[amt - coin]

Base: dp[0] = 1
```

Pattern 5: Interval DP

Definition

Optimize operations on intervals [i, j], typically merging or splitting.

When to Use

- "Optimal way to split/merge range"
- "Cost of operations on [i, j]"
- Parenthesization problems

- Palindrome partitioning

State Design

```
dp[i][j] = optimal value for interval [i, j]
```

Recurrence Template

```
dp[i][j] = optimal over all split points k ∈ [i, j) {
   dp[i][k] + dp[k+1][j] + cost_merge(i, k, j)
}

Base: dp[i][i] = base_value (single element)
```

Iteration Order (Critical!)

```
// Fill by increasing interval length
for len from 2 to n:
   for i from 0 to n-len:
      j = i + len - 1
      dp[i][j] = compute(i, j)
```

Why length-based? Ensures dp[i][k] and dp[k+1][j] computed before dp[i][j].

Examples

Matrix Chain Multiplication:

```
dp[i][j] = min operations to multiply matrices [i..j]

dp[i][j] = min over k ∈ [i, j) {
   dp[i][k] + dp[k+1][j] + dim[i-1]·dim[k]·dim[j]
}

Cost of multiplying (i→k) with (k+1→j):
 Result dimensions: dim[i-1] × dim[j]
 Operations: dim[i-1] · dim[k] · dim[j]
```

Burst Balloons:

```
dp[i][j] = max coins bursting balloons (i, j) exclusive

for k from i+1 to j-1:  // k is LAST balloon to burst
   dp[i][j] = max(
      dp[i][k] + dp[k][j] + arr[i]·arr[k]·arr[j]
   )

Trick: Add dummy balloons arr[-1]=1, arr[n]=1
```

Palindrome Partitioning:

```
dp[i][j] = min cuts to make s[i..j] all palindromes

if is_palindrome(s[i..j]):
   dp[i][j] = 0
else:
   dp[i][j] = min over k {
      dp[i][k] + dp[k+1][j] + 1
   }
```

Pattern 6: Partition DP

Definition

Split array/string into parts with specific properties.

When to Use

- "Partition into k parts"
- "Minimum partitions/cuts"
- "Valid partitions" (palindrome, dictionary words)

State Design

```
// Minimum partitions
dp[i] = min partitions for s[0..i-1]

// k partitions
dp[i][k] = optimal to partition [0..i-1] into k parts
```

Recurrence Templates

Minimum Partitions:

```
dp[i] = min over all j < i {
    dp[j] + 1    // if [j, i) is valid partition
}

Base: dp[0] = 0
```

k Partitions:

```
dp[i][k] = min/max over all j < i {
    dp[j][k-1] + cost(j, i)
}

Base: dp[0][0] = 0
```

Examples

Word Break:

```
dp[i] = true if s[0..i-1] can be segmented

dp[i] = OR over all j < i {
    dp[j] AND (s[j..i-1] in dictionary)
}

Base: dp[0] = true
```

Palindrome Partitioning II:

```
dp[i] = min cuts for s[0..i-1] to be all palindromes

dp[i] = min over j < i {
    dp[j] + 1    // if s[j..i-1] is palindrome
}

Optimization: Precompute is_palin[i][j]
```

Split Array Largest Sum:

```
Split array into k subarrays minimizing largest subarray sum

dp[i][k] = min largest sum using first i elements in k parts

dp[i][k] = min over j {
   max(dp[j][k-1], sum(j+1, i))
}
```

Pattern 7: Tree DP

Definition

DP on tree structures where state depends on subtree solutions.

When to Use

- Problems on trees
- "Optimal node selection"
- "Tree paths/distances"
- Binary/N-ary trees

State Design

```
dp[node][state] = optimal for subtree rooted at node

Common states:
 0 = node not selected/used
 1 = node selected/used
```

Recurrence Template

```
// Post-order traversal
function dfs(node):
   if node is null:
      return base_value

   // Process children first
   for child in node.children:
      dfs(child)

   // Combine child results
```

```
    dp[node][0] = combine(dp[child][0], dp[child][1])
    dp[node][1] = value[node] + combine(dp[child][0])
```

Examples

House Robber III:

```
dp[node][0] = max money if node NOT robbed
dp[node][1] = max money if node IS robbed

dp[node][0] = max(dp[left][0], dp[left][1])
            + max(dp[right][0], dp[right][1])

dp[node][1] = node.val + dp[left][0] + dp[right][0]

Answer: max(dp[root][0], dp[root][1])
```

Binary Tree Maximum Path Sum:

```
For each node, compute:
  max_single = max path going through node to ONE subtree
  max_through = max path going through node connecting subtrees

max_single[node] = max(
    node.val,
    node.val + max_single[left],
    node.val + max_single[right]
)

max_through[node] = max(
    max_single[node],
    node.val + max_single[left] + max_single[right]
)

global_max = max(global_max, max_through[node])
```

Pattern 8: Digit DP

Definition

Count numbers with digit properties in range [L, R].

When to Use

- "Count numbers with property"
- "Numbers in range [L, R]"
- Digit constraints (sum, pattern, uniqueness)

State Design

```
dp[pos][tight][...state]

where:
  pos = current digit position (left to right)
  tight = still bounded by upper limit? (0/1)
  ...state = problem-specific (sum, last digit, etc.)
```

Template

```
function count(pos, tight, state):
    if pos == num_digits:
        return is_valid(state)

    if memoized(pos, tight, state):
        return memo[pos][tight][state]

    limit = tight ? digit[pos] : 9
    result = 0

    for d from 0 to limit:
        new_tight = tight AND (d == limit)
        new_state = update_state(state, d)
        result += count(pos+1, new_tight, new_state)

    memo[pos][tight][state] = result
    return result
```

Examples

Count Numbers with Unique Digits:

```
State: (pos, tight, mask_used_digits)

For each digit d:
  if d already used (mask & (1<<d)):
    skip
  else:
    recurse with mask | (1<<d)
```

Numbers at Most N Given Digit Set:

```
State: (pos, tight, started)

started = have we placed a non-zero digit yet?

For each allowed digit d:
  if d > limit and tight: skip
  else: recurse
```

Pattern 9: Bitmask DP (State Compression)

Definition

Using bitmasks to represent subsets for problems with small n (≤ 20).

When to Use

- "All subsets" consideration
- "Which elements used/visited"
- $n \leq 20$ ($2^{20} \approx 10^6$)
- TSP, assignment, subset problems

State Design

```
dp[mask] = optimal for subset represented by mask
dp[mask][i] = optimal for subset mask ending at element i

Bit representation:
  bit i = 1 → element i included
  bit i = 0 → element i not included
```

Common Bitmask Operations

```
// Check if i-th bit set
if (mask & (1 << i))

// Set i-th bit
mask | (1 << i)

// Clear i-th bit
mask & ~(1 << i)

// Toggle i-th bit
mask ^ (1 << i)

// Count set bits
__builtin_popcount(mask)

// Iterate all subsets of mask
for (sub = mask; sub > 0; sub = (sub-1) & mask)
```

Examples

Traveling Salesman Problem:

```
dp[mask][i] = min cost to visit cities in mask, ending at i

Base: dp[1<<start][start] = 0

Transition:
for each city j in mask:
   new_mask = mask ^ (1 << j)  // remove j
   for each city k not in new_mask:
      dp[mask][j] = min(
         dp[mask][j],
         dp[new_mask][k] + dist[k][j]
      )

Answer: min over all i { dp[(1<<n)-1][i] + dist[i][start] }
```

Shortest Path Visiting All Nodes:

```
dp[mask][node] = min path to visit cities in mask, at node

BFS initialization:
  dp[1<<i][i] = 0 for all i

Transition:
  for neighbors v of node:
    new_mask = mask | (1 << v)
    dp[new_mask][v] = min(dp[new_mask][v], dp[mask][node] + 1)

Answer: min over all i { dp[(1<<n)-1][i] }
```

Pattern 10: DP on Graphs

Definition

DP combined with graph traversal for path problems with constraints.

When to Use

- "Shortest/longest path" with constraints
- "k-hop paths"
- "Paths visiting specific nodes"
- DAG optimization

State Design

```
// Constrained paths
dp[node][k] = optimal path to node using k edges/steps

// DAG DP
dp[node] = optimal value reaching node
```

Templates

Shortest Path with k Edges (Bellman-Ford DP):

```
dp[v][k] = shortest path to v using at most k edges

dp[v][0] = 0 if v == source else ∞

for i from 1 to k:
    for edge (u, v) with weight w:
        dp[v][i] = min(dp[v][i], dp[u][i-1] + w)

Answer: dp[target][k]
```

Longest Path in DAG:

```
// Topological sort first
dp[v] = longest path to v

dp[source] = 0

for v in topological_order:
    for edge (u, v):
        dp[v] = max(dp[v], dp[u] + weight(u,v))

Answer: max(dp[v]) for all v
```

Examples

Cheapest Flights Within K Stops:

```
dp[city][stops] = min cost to reach city in at most stops

dp[src][0] = 0

for s from 0 to k:
    for flight (from, to, price):
        dp[to][s+1] = min(
            dp[to][s+1],
            dp[from][s] + price
        )

Answer: min over s { dp[dst][s] }
```

---

4. DP STATE DESIGN FRAMEWORK

Systematic 8-Step Process

Step 1: Identify Problem Type

Questions:

- What are we optimizing? (min/max value, count ways)
- What are we choosing? (items, cuts, paths)
- What constraints exist? (capacity, budget, order)

Classifications:

- **Selection**: Knapsack-like
- **Partition**: Split into parts
- **Matching**: Two sequences
- **Path**: Grid/graph traversal
- **Combination**: Subsets/assignments

Step 2: Identify Dimensions

Dimension Checklist:

- □ Position in array/string? → Add index (i)
- □ Two sequences? → Add (i, j)
- □ Capacity constraint? → Add (w) or (capacity)
- □ Count constraint? → Add (k) (# of items/partitions)
- □ State changes? → Add state variable
- □ Subset tracking? → Add bitmask

Common Patterns:

```
1D: dp[i]
2D: dp[i][j], dp[i][w], dp[i][k]
3D: dp[i][j][k]
Bitmask: dp[mask], dp[mask][i]
```

Step 3: Define State Meaning

Two Conventions:

Convention A (Exclusive end):

```
dp[i] = optimal for first i elements (elements [0..i-1])

Pros: Clean base case dp[0]
Cons: Final answer is dp[n], not dp[n-1]
```

Convention B (Inclusive):

```
dp[i] = optimal including element i (or [0..i])

Pros: Intuitive indexing
Cons: Special handling for dp[0]
```

Choose ONE and be consistent!

Step 4: Enumerate Decision Choices

Common Decision Types:

1. **Binary**: Take or skip

```
Knapsack: Include item i OR skip it
```

2. **Multiple**: Which previous to extend

```
LIS: Extend from which j < i
```

3. **Split Point**: Where to partition

```
Interval DP: Split at position k
```

4. **Next State**: State machine transition

```
Tree DP: Children states
```

Step 5: Write Recurrence

General Template:

```
dp[current_state] = optimal_function {
   over all valid decisions:
      combine(dp[previous_state], decision_cost)
}
```

Optimal Functions:

- Minimize: `min(...)`
- Maximize: `max(...)`
- Count ways: `sum(...)`
- Feasibility: `OR(...)`

Example Construction:

```
Problem: Maximum sum, no adjacent elements

Decision at i:
  Choice 1: Skip i → dp[i-1]
  Choice 2: Take i → arr[i] + dp[i-2]

Recurrence:
  dp[i] = max(dp[i-1], arr[i] + dp[i-2])
```

Step 6: Define Base Cases

Principles:

1. Smallest/trivial subproblem
2. Directly computable (no recursion)
3. Foundation for building up

Common Bases:

```
Empty: dp[0] = 0 or 1
Single: dp[1] = trivial_value
Boundary: dp[i][0] = boundary, dp[0][j] = boundary
Invalid: dp[...] = ∞ (minimize) or -∞ (maximize)
```

Validation:

- Can all valid states be reached from base cases?
- Are base cases correct by problem definition?

Step 7: Determine Iteration Order

For Tabulation: Process in topological order

Standard Orders:

```
1D Linear:
 for i from 0 to n:
    dp[i] = f(dp[i-1], dp[i-2], ...)

2D Row-by-Row:
 for i from 0 to m:
    for j from 0 to n:
       dp[i][j] = f(dp[i-1][j], dp[i][j-1], ...)

Reverse (Space Optimization):
 for i from 0 to n:
    for w from W down to weight[i]:
       dp[w] = ...

Interval (By Length):
 for len from 2 to n:
    for i from 0 to n-len:
       j = i + len - 1
       dp[i][j] = ...
```

Golden Rule: Process state only after ALL dependencies computed

Step 8: Optimize Memory

Technique 1: Constant Space (1D → O(1))

```
// Original: dp[n]
// Optimized: 2-3 variables

prev2 = dp[0]
prev1 = dp[1]

for i from 2 to n:
   curr = f(prev1, prev2)
   prev2 = prev1
   prev1 = curr

return prev1
```

Technique 2: Rolling Array (2D → 1D)

```
// Original: dp[m][n]
// Optimized: dp[n]

for i from 0 to m:
    for j from 0 to n:
        new_dp[j] = f(dp[j], new_dp[j-1], ...)
    dp = new_dp
```

Technique 3: Two-Row (2D → 2×1D)

```
prev_row = [0] * (n+1)
curr_row = [0] * (n+1)

for i from 1 to m:
    for j from 1 to n:
        curr_row[j] = f(prev_row[j], curr_row[j-1], prev_row[j-1])
    prev_row, curr_row = curr_row, prev_row
```

---

5. CORE DP PATTERNS — DEEP DIVE

5.1 Longest Increasing Subsequence (LIS)

Problem

Find length of longest strictly increasing subsequence.

State Definition

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

Recurrence

```
dp[i] = max(dp[j] + 1) for all j < i where arr[j] < arr[i]

Base: dp[i] = 1 for all i
Answer: max(dp[i]) over all i
```

Pseudocode ($O(n^2)$)

```
for i from 0 to n-1:
    dp[i] = 1
    for j from 0 to i-1:
        if arr[j] < arr[i]:
            dp[i] = max(dp[i], dp[j] + 1)

return max(dp)
```

Optimization: Binary Search (O(n log n))

```
tail = []  // tail[k] = smallest ending value of LIS length k+1

for x in arr:
    pos = binary_search_lower_bound(tail, x)
    if pos == len(tail):
        tail.append(x)
    else:
        tail[pos] = x

return len(tail)
```

Key Insight: Among all LIS of same length, prefer one with smallest ending element.

5.2 Longest Common Subsequence (LCS)

Problem

Find length of longest common subsequence of two strings.

State Definition

```
dp[i][j] = LCS length of s1[0..i-1] and s2[0..j-1]
```

Recurrence

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

```
Base: dp[i][0] = 0, dp[0][j] = 0
Answer: dp[m][n]
```

Pseudocode

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

Space Optimization

```
// O(min(m,n)) space
prev = [0] * (n+1)
curr = [0] * (n+1)

for i from 1 to m:
    for j from 1 to n:
        if s1[i-1] == s2[j-1]:
            curr[j] = prev[j-1] + 1
        else:
            curr[j] = max(prev[j], curr[j-1])
    prev, curr = curr, [0]*(n+1)
```

5.3 Edit Distance

Problem

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

State Definition

```
dp[i][j] = min edits to convert s1[0..i-1] to s2[0..j-1]
```

Recurrence

```
if s1[i-1] == s2[j-1]:
    dp[i][j] = dp[i-1][j-1]  // no operation
else:
    dp[i][j] = 1 + min(
```

```
    dp[i-1][j],     // delete from s1
    dp[i][j-1],     // insert into s1
    dp[i-1][j-1]    // replace
  )

Base: dp[i][0] = i, dp[0][j] = j
```

Interpretation

```
dp[i-1][j]   : Delete s1[i-1], then convert s1[0..i-2] to s2[0..j-1]
dp[i][j-1]   : Convert s1[0..i-1] to s2[0..j-2], then insert s2[j-1]
dp[i-1][j-1] : Replace s1[i-1] with s2[j-1], convert rest
```

5.4 Coin Change

Problem

Minimum coins to make target amount.

State Definition

```
dp[amount] = minimum coins needed
```

Recurrence

```
dp[amt] = min over coins c {
  dp[amt - c] + 1
}

Base: dp[0] = 0
Invalid: dp[amt] = ∞ if unreachable
Answer: dp[target]
```

Pseudocode

```
dp = [∞] * (amount + 1)
dp[0] = 0

for amt from 1 to amount:
  for coin in coins:
    if amt >= coin:
      dp[amt] = min(dp[amt], dp[amt - coin] + 1)
```

```
return dp[amount] if dp[amount] != ∞ else -1
```

5.5 0/1 Knapsack

Problem

Maximum value within weight capacity, each item used at most once.

State Definition

```
dp[i][w] = max value using first i items with capacity w
```

Recurrence

```
dp[i][w] = max(
    dp[i-1][w],                 // skip item i
    dp[i-1][w-wt[i]] + val[i]   // take item i
)

Condition: Second option only if w >= wt[i]
Base: dp[0][w] = 0
Answer: dp[n][W]
```

Space-Optimized (1D)

```
dp = [0] * (W + 1)

for i from 0 to n-1:
    for w from W down to wt[i]:  // REVERSE
        dp[w] = max(dp[w], dp[w - wt[i]] + val[i])

return dp[W]
```

Why reverse? Ensure we don't use item i multiple times.

5.6 Matrix Chain Multiplication

Problem

Minimum scalar multiplications for matrix multiplication.

State Definition

```
dp[i][j] = min operations to multiply matrices [i..j]
```

Recurrence

```
dp[i][j] = min over k in [i, j) {
   dp[i][k] + dp[k+1][j] + dims[i-1] * dims[k] * dims[j]
}

Base: dp[i][i] = 0
```

Pseudocode

```
for len from 2 to n:
   for i from 1 to n-len+1:
      j = i + len - 1
      dp[i][j] = ∞

      for k from i to j-1:
         cost = dp[i][k] + dp[k+1][j] + dims[i-1]*dims[k]*dims[j]
         dp[i][j] = min(dp[i][j], cost)
```

5.7 Palindrome Problems

Longest Palindromic Subsequence

State: `dp[i][j]` = LPS length in s[i..j]

Recurrence:

```
if s[i] == s[j]:
   dp[i][j] = dp[i+1][j-1] + 2
else:
   dp[i][j] = max(dp[i+1][j], dp[i][j-1])

Base: dp[i][i] = 1
```

Longest Palindromic Substring

State: `dp[i][j]` = true if s[i..j] is palindrome

Recurrence:

```
dp[i][j] = (s[i] == s[j]) AND (j - i <= 2 OR dp[i+1][j-1])
```

---

6. DP OPTIMIZATION TECHNIQUES

6.1 Space Compression

Observation

If `dp[i]` depends only on `dp[i-1], dp[i-2], ..., dp[i-k]`, we need only k variables.

Examples

Fibonacci:

```
// From O(n) to O(1)
a, b = 0, 1
for i from 2 to n:
    a, b = b, a+b
return b
```

Knapsack 2D → 1D:

```
// From dp[n][W] to dp[W]
for each item:
    for w from W down to weight:
        dp[w] = max(dp[w], dp[w-weight] + value)
```

6.2 Rolling Arrays

Technique

Keep only last k rows/columns.

LCS Example

```
// From dp[m][n] to 2 rows
prev_row = [0] * (n+1)
curr_row = [0] * (n+1)

for i from 1 to m:
    for j from 1 to n:
```

```
    # compute curr_row[j] using prev_row
prev_row, curr_row = curr_row, prev_row
```

6.3 Prefix/Suffix Optimization

Use Case

Avoid inner loops by precomputing aggregates.

Example

```
// Instead of O(n²)
for i:
    for j < i:
        dp[i] = max(dp[i], dp[j] + ...)

// Optimize to O(n) using prefix max
prefix_max[i] = max(dp[0..i])
for i:
    dp[i] = prefix_max[i-1] + ...
```

6.4 Monotonic Queue/Stack

When

Sliding window maximum/minimum in DP.

Example: Sliding Window Maximum in DP

```
Use deque to maintain max in window
for each i:
    # Remove elements outside window
    while deque and deque[0] < i - k:
        deque.popleft()

    # Remove smaller elements
    while deque and arr[deque[-1]] < arr[i]:
        deque.pop()

    deque.append(i)
    dp[i] = arr[deque[0]]
```

---

7. INTERVIEW PATTERN MATCHING

Decision Tree: Problem → Pattern

By Objective

Find Maximum/Minimum VALUE:

- Single sequence → 1D Linear / House Robber
- Two sequences → LCS / Edit Distance
- With capacity → Knapsack
- Range/interval → Interval DP

Count NUMBER OF WAYS:

- Paths → Grid DP / Linear DP
- Form target → Knapsack variants
- Partitions → Partition DP

Find LONGEST/SHORTEST:

- Substring → Sliding Window / Interval DP
- Subsequence → LIS / LCS

By Structure

Linear Array → 1D DP **2D Grid** → 2D Grid DP **Two Sequences** → 2D DP (LCS-like) **Tree** → Tree DP **Graph** → Graph DP **Small n (≤20)** → Bitmask DP

By Keywords

Keyword	Pattern
"maximum sum"	1D Linear / Kadane's
"ways to reach"	Grid DP / Counting DP
"longest increasing"	LIS
"longest common"	LCS
"edit/transform"	Edit Distance
"choose items" + "capacity"	Knapsack
"partition"	Partition DP

Keyword	Pattern
"range [i,j]"	Interval DP
"all subsets"	Bitmask DP
"tree nodes"	Tree DP

8. COMMON DP PITFALLS

8.1 Wrong State Definition

Problem: State doesn't capture necessary information. **Example**: Max sum without adjacent - need to track if last taken. **Solution**: Add state dimension.

8.2 Wrong Base Case

Problem: Incorrect initialization. **Example**: Coin change - dp[0] = 0, not 1. **Solution**: Manually verify smallest case.

8.3 Wrong Transition

Problem: Using incorrect previous states. **Example**: 0/1 Knapsack using dp[i][...] instead of dp[i-1][...]. **Solution**: Trace small example carefully.

8.4 TLE (Time Limit Exceeded)

Causes:

- Too many states (n^2 with $n=10^5$)
- Expensive transitions **Solutions**:
- Optimize state space
- Use better data structures
- Consider different algorithm

8.5 MLE (Memory Limit Exceeded)

Causes:

- Large DP table **Solutions**:
- Space optimization (rolling array)
- State elimination

8.6 Debugging Strategy

1. Print DP table for small input
2. Trace example by hand
3. Verify base cases
4. Check iteration order
5. Validate recurrence

---

9. 60+ CURATED DP INTERVIEW PROBLEMS

Organized by Pattern

1D Linear DP

- LC 70: Climbing Stairs
- LC 198: House Robber
- LC 213: House Robber II
- LC 746: Min Cost Climbing Stairs
- LC 91: Decode Ways

2D Grid DP

- LC 62: Unique Paths
- LC 63: Unique Paths II
- LC 64: Minimum Path Sum
- LC 120: Triangle
- LC 221: Maximal Square

Subsequence DP

- LC 300: Longest Increasing Subsequence
- LC 1143: Longest Common Subsequence
- LC 72: Edit Distance
- LC 115: Distinct Subsequences
- LC 583: Delete Operation for Two Strings

Knapsack DP

- LC 322: Coin Change
- LC 518: Coin Change 2
- LC 416: Partition Equal Subset Sum
- LC 494: Target Sum
- LC 279: Perfect Squares

Interval DP

- LC 516: Longest Palindromic Subsequence
- LC 312: Burst Balloons
- LC 1130: Minimum Cost Tree From Leaf Values
- LC 647: Palindromic Substrings

Partition DP

- LC 139: Word Break
- LC 132: Palindrome Partitioning II
- LC 410: Split Array Largest Sum

Tree DP

- LC 337: House Robber III
- LC 124: Binary Tree Maximum Path Sum
- LC 543: Diameter of Binary Tree

Bitmask DP

- LC 847: Shortest Path Visiting All Nodes
- LC 698: Partition to K Equal Sum Subsets
- LC 1125: Smallest Sufficient Team

Graph DP

- LC 787: Cheapest Flights Within K Stops
- LC 1976: Number of Ways to Arrive at Destination

---

10. 30-DAY DP MASTERY ROADMAP

Week 1: Foundations

Day 1-2: Theory + Fibonacci + 1D Linear
Day 3-4: 2D Grid DP
Day 5-7: Subsequence DP (LIS, LCS)

Week 2: Core Patterns

Day 8-10: Knapsack variants
Day 11-12: Interval DP
Day 13-14: Partition DP + Review

Week 3: Advanced

Day 15-16: Tree DP + Digit DP
Day 17-18: Bitmask DP
Day 19-21: Graph DP + Optimizations

Week 4: Mastery

Day 22-24: Hard problems
Day 25-27: Mock interviews
Day 28-30: Review + weak areas

---

11. MENTAL MODELS

Model 1: Decision Tree to Table

Every DP starts as a recursive tree. DP converts it to a table by identifying and computing each unique node once.

Model 2: Recurrence Mindset

"How to solve current using smaller?" Trust recursion, focus on current decision, express in terms of smaller.

Model 3: State Machine View

States are configurations, transitions are decisions, goal is optimal path through states.

Model 4: Knapsack Lens

Many problems are hidden knapsacks: selecting items with constraints to optimize value.

Model 5: Build-Up Principle

Start from smallest (base cases), incrementally build to larger, combine smaller solutions.

CONCLUSION

You now have:

- ✓ Theoretical foundation
- ✓ 10 comprehensive patterns
- ✓ State design framework
- ✓ Optimization techniques
- ✓ 60+ problems roadmap
- ✓ Mental models

Next Steps:

1. Follow 30-day roadmap
2. Solve actively, not passively
3. Focus on patterns, not problems
4. Practice explaining your approach

Remember: DP mastery is a journey. Every problem strengthens pattern recognition. Stay consistent!

END OF GUIDE

Document Statistics:

- Patterns: 10
- Problems: 60+
- Study Days: 30
- Mental Models: 5+