🧮 Dynamic Programming (DP) Interview Quick-Check Pattern

A crystal-clear cheat sheet to recognize, plan, and implement DP problems like a pro.

---

1. 🧠 How to Recognize a DP Problem

Ask yourself:

- ❓ Does the problem have **overlapping subproblems**?
- ❓ Can it be **broken into smaller, similar decisions**?
- ❓ Are there **multiple paths** to the solution?
- ❓ Do I need to find the **min/max/count of something**?
- ❓ Are there **constraints** like weights, lengths, coins, k steps?

> 💡 If you're **recomputing results** — it's likely a DP problem.

---

2. 🧱 5-Step DP Framework

Step	Question
1. 🤔 What is the **problem asking**?	
2. 📦 What is the **state**? (What varies?)	
3. 🔁 What is the **recurrence relation**?	
4. 🧮 What is the **base case**?	
5. 🧾 Should I use **Memoization (Top-Down)** or **Tabulation (Bottom-Up)**?	

---

3. 🔧 Core Templates

✅ Fibonacci (Memoized Top-Down)

```
function fib(n, memo = {}) {
  if (n <= 1) return n;
  if (n in memo) return memo[n];
  memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

✅ Fibonacci (Tabulated Bottom-Up)

```
function fib(n) {
  if (n <= 1) return n;
  const dp = [0, 1];
  for (let i = 2; i <= n; i++) {
    dp[i] = dp[i - 1] + dp[i - 2];
  }
  return dp[n];
}
```

✅ 0/1 Knapsack (Classic DP Table)

```
function knapsack(weights, values, W) {
  const n = weights.length;
  const dp = Array(n + 1).fill().map(() => Array(W + 1).fill(0));

  for (let i = 1; i <= n; i++) {
    for (let w = 0; w <= W; w++) {
      if (weights[i - 1] <= w) {
        dp[i][w] = Math.max(dp[i - 1][w],
                            values[i - 1] + dp[i - 1][w - weights[i - 1]]);
      } else {
        dp[i][w] = dp[i - 1][w];
      }
    }
  }

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

✅ House Robber (Linear DP)

```
function rob(nums) {
  if (!nums.length) return 0;
  if (nums.length === 1) return nums[0];
  const dp = [nums[0], Math.max(nums[0], nums[1])];
  for (let i = 2; i < nums.length; i++) {
    dp[i] = Math.max(dp[i - 1], dp[i - 2] + nums[i]);
  }
  return dp[nums.length - 1];
}
```

✅ Longest Common Subsequence (2D DP)

```
function longestCommonSubsequence(text1, text2) {
  const m = text1.length, n = text2.length;
  const dp = Array(m + 1).fill().map(() => Array(n + 1).fill(0));

  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      if (text1[i - 1] === text2[j - 1]) {
        dp[i][j] = 1 + dp[i - 1][j - 1];
      } else {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
      }
    }
  }

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

4. 📊 Top DP Problem Types

Type	Examples	Pattern
1D DP	Climb Stairs, Rob Houses	Linear
2D DP (Grid)	Unique Paths, Min Path Sum	Bottom-Up Table
Subsequence/Substring	LCS, LIS, Edit Distance	2D DP Matrix
Knapsack-like	0/1 Knapsack, Coin Change	DP on weights
Partitioning	Palindrome Partition, Burst Balloons	Recursive + memo
Decision Trees	Game Theory, Minimax Problems	DP with max/min
Bitmask DP	Traveling Salesman, N-Queens	State = bits

5. 🧱 Edge Cases & Pitfalls

- Off-by-one indexing (`dp[i-1]` vs `dp[i]`)
- Memo keys must be unique per state (`i + ',' + j`)
- Mutating arrays when copying (deep copy!)
- Forgetting base case → leads to stack overflow
- Comparing `===` instead of `==` for characters
- Overlapping subproblem detection (recurse → fails if not memoized)

6. 🧠 Mental Model to Store Forever

> 🧠 **Dynamic Programming = Remembering the Past**

Think Like This:

- 🎯 Goal → What do I need to calculate?
- 🔁 Recurse → How can I break the problem down?
- 💾 Cache → Can I save results to avoid recomputing?
- 🧮 Build → Should I build up from the base case?

✅ Final Interview Checklist

- Did I identify the **state variables**?
- Did I define a **clear base case**?
- Do I understand the **recurrence relation**?
- Is it better solved **top-down with memo** or **bottom-up tabulation**?
- Are my **dimensions correct**? (e.g., 1D vs 2D DP)
- Can I **optimize space** (e.g., `dp[i-1]` only)?