Dynamic Programming Cheatsheet (Java Focus)

When to Use DP

Use DP if the problem has:
1. Overlapping Subproblems
2. Optimal Substructure

Examples: Fibonacci, Climbing Stairs, LCS, Knapsack

Where DP is Used (Patterns)

- Fibonacci-like: Climbing Stairs, Tiling Problems
- Knapsack: Subset Sum, Resource Allocation
- Grid DP: Unique Paths, Matrix Path Sum
- String DP: LCS, Edit Distance
- Subset DP: Target Sum, Coin Change
- Partition DP: Palindrome Partitioning
- Bitmask DP: Travelling Salesman
- Interval DP: Matrix Chain Multiplication

Memoization Template (Top-Down)

```
Map<String, Integer> memo = new HashMap<>();
int dp(int i, int j) {
    String key = i + "," + j;
    if (memo.containsKey(key)) return memo.get(key);
    // Base case
    int result = some_recursion(i, j);
    memo.put(key, result);
    return result;
}
```

Tabulation Template (Bottom-Up)

```
int[][] dp = new int[n + 1][m + 1];
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        dp[i][j] = ...;
    }
}
```

Common Problems with Java Code

1. Fibonacci:

```
int fib(int n) {
  int[] dp = new int[n + 1];
  dp[1] = 1;
  for (int i = 2; i <= n; i++)
    dp[i] = dp[i - 1] + dp[i - 2];
```

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

2. Climbing Stairs:

```
int climbStairs(int n) {
  if (n <= 2) return n;
  int[] dp = new int[n + 1];
  dp[1] = 1; dp[2] = 2;
  for (int i = 3; i <= n; i++)
    dp[i] = dp[i - 1] + dp[i - 2];
  return dp[n];
}
```

3. Knapsack:

```
int knapsack(int[] wt, int[] val, int W) {
  int[][] dp = new int[wt.length + 1][W + 1];
  for (int i = 1; i <= wt.length; i++) {
    for (int w = 0; w <= W; w++) {
      if (wt[i - 1] <= w)
        dp[i][w] = Math.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[wt.length][W];
}
```

4. LCS:

```
int lcs(String a, String b) {
  int[][] dp = new int[a.length() + 1][b.length() + 1];
  for (int i = 1; i <= a.length(); i++) {
    for (int j = 1; j <= b.length(); j++) {
      if (a.charAt(i-1) == b.charAt(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[a.length()][b.length()];
}
```