Dynamic Programming (DP) in DSA: Beginner to Advanced Guide

Subtitle: Master dynamic programming concepts, patterns, and common problems to solve complex coding challenges efficiently.

Website Name: haas.dev
Website Link: https://dev-roast-app.vercel.app

Introduction

Dynamic Programming (DP) is a technique to solve problems by breaking them into **smaller overlapping subproblems** and storing results to avoid recomputation. DP is widely used in optimization problems, sequences, and combinatorial problems. This guide teaches DP step by step.

Step 1: Understanding DP

- **Key Idea:** Store results of subproblems (memoization or tabulation)
- **Two Approaches:**
  1. **Top-Down (Memoization)** → recursion + cache
  2. **Bottom-Up (Tabulation)** → iterative filling of table

Example: Fibonacci using memoization

```
let memo = {};

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

Step 2: Common DP Patterns

1. 1D DP

- Problems like **Fibonacci, climbing stairs, min cost**

2. 2D DP

- Problems like **LCS, matrix paths, knapsack**

3. DP on Subsets / Bitmask

- Problems with **combinations of items**

4. DP on Trees / Graphs

- Solve tree-based optimizations
- Path count in DAG

Step 3: Classic DP Problems

1. Climbing Stairs

```
function climbStairs(n) {
  let dp = Array(n+1).fill(0);
  dp[0] = 1; dp[1] = 1;
  for (let i=2; i<=n; i++) dp[i] = dp[i-1] + dp[i-2];
  return dp[n];
}
```

2. 0/1 Knapsack

```
function knapsack(weights, values, W) {
  let n = weights.length;
  let dp = Array(n+1).fill(0).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(values[i-1] + dp[i-1][w - weights[i-1]], dp[i-1][w]);
      } else dp[i][w] = dp[i-1][w];
    }
  }
  return dp[n][W];
}
```

3. Longest Common Subsequence (LCS)

```
function LCS(a, b) {
  let n = a.length, m = b.length;
  let dp = Array(n+1).fill(0).map(()=>Array(m+1).fill(0));

  for (let i=1; i<=n; i++) {
    for (let j=1; j<=m; j++) {
      if (a[i-1] === b[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[n][m];
}
```

Step 4: Key DP Concepts

- **Overlapping Subproblems:** Same subproblems occur multiple times
- **Optimal Substructure:** Solution can be constructed from solutions of subproblems
- **State Definition:** Identify variables that define subproblems
- **Transition:** Formula to move from subproblem to problem

Step 5: Common Mistakes

- Forgetting base cases
- Misdefining states → wrong transitions
- Using recursion without memoization → TLE
- Ignoring edge cases

Step 6: Practice Plan for DP

Day 1:

- Fibonacci, climbing stairs → 5 problems

Day 2:

- Min cost, coin change → 5–7 problems

Day 3:

- 0/1 knapsack, subset sum → 5–7 problems

Day 4:

- LCS, LIS (longest increasing subsequence) → 5–7 problems

Day 5:

- Matrix path problems → 5 problems

Day 6–7:

- Mixed DP problems + revision → 10–12 problems

Mini Exercises

1. Minimum steps to reduce a number to 1
2. Maximum sum of non-adjacent elements
3. Count number of ways to reach end of stairs
4. Longest palindromic subsequence

Key Takeaways

- DP is essential for **optimization and sequence problems**
- Learn **top-down vs bottom-up approaches**
- Define **states, base cases, and transitions carefully**
- Start with small examples and gradually tackle larger problems
- Consistent practice builds **intuition for DP patterns**

Visit haas.dev for more DSA dynamic programming guides, problem sets, and interview preparation resources.

Website Name: haas.dev
Website Link: https://dev-roast-app.vercel.app