Ultimate Coding Patterns Mastery - Volume 3

Dynamic Programming, Backtracking, Advanced Data Structures, and Optimization Patterns

Pattern 8: Dynamic Programming (DP)

8.1 Core Understanding

Layman's Intuition

Imagine climbing stairs—at each step, you can go up 1 or 2 steps. To find total ways to reach step N, you realize: ways(N) = ways(N-1) + ways(N-2). Instead of recalculating ways(N-1) and ways(N-2) repeatedly, you save (memoize) each answer. That's DP—solving big problems by combining solutions to smaller overlapping subproblems.

Technical Foundation

Definition: Dynamic Programming solves optimization problems by breaking them into overlapping subproblems, storing solutions to avoid redundant computation.

Core Principles:

1. **Optimal Substructure:** Optimal solution contains optimal solutions to subproblems
2. **Overlapping Subproblems:** Same subproblems solved multiple times

Two Approaches:

- **Top-Down (Memoization):** Recursion + caching
- **Bottom-Up (Tabulation):** Iterative table filling

Complexity Comparison:

Method	Time	Space	When to Use
Naive Recursion	Exponential	O(depth)	Never (educational only)
Top-Down DP	O(states × transition)	O(states) + recursion	Natural recursive structure
Bottom-Up DP	O(states × transition)	O(states)	Optimal space, iterative

When to Use DP:

- **Optimization:** Maximum/minimum value
- **Counting:** Number of ways
- **Decision-making:** Yes/no feasibility

- **Keywords:** "maximum", "minimum", "count ways", "longest", "shortest", "optimal"

8.2 Problem-Solving Framework

Step 1: Identify DP Problem

Signals:

- Asks for optimal value (max/min)
- Count number of ways
- Make decisions at each step
- Constraints suggest exponential brute force

Step 2: Define State

State = Information needed to solve subproblem

Examples:

- Fibonacci: `dp[i]` = fib(i)
- Knapsack: `dp[i][w]` = max value using first i items with weight limit w
- LCS: `dp[i][j]` = longest common subsequence of s1[0..i], s2[0..j]

Step 3: Find Recurrence Relation

Express current state in terms of previous states

Examples:

- Fibonacci: `dp[i] = dp[i-1] + dp[i-2]`
- Knapsack: `dp[i][w] = max(dp[i-1][w], dp[i-1][w-weight[i]] + value[i])`

Step 4: Determine Base Cases

Smallest subproblems with known answers

Examples:

- Fibonacci: `dp[0] = 0, dp[1] = 1`
- Knapsack: `dp[0][w] = 0` (no items)

Step 5: Decide Computation Order

Bottom-up: Fill table in order that dependencies are ready
Top-down: Recursive with memoization handles order automatically

8.3 Complete Problem Solutions

Problem 1: Climbing Stairs (LeetCode 70)

All Approaches from Worst to Best:

Approach 1: Pure Recursion - O(2^n) time, O(n) space

```
int climbStairsRecursive(int n) {
    if (n &lt;= 1) return 1;
    return climbStairsRecursive(n - 1) + climbStairsRecursive(n - 2);
}
// Time: O(2^n) - exponential
// Space: O(n) - recursion stack
// NEVER use in production!
```

Approach 2: Top-Down DP (Memoization) - O(n) time, O(n) space

```
/**
 * Climbing stairs with memoization
 *
 * State: dp[i] = ways to reach step i
 * Recurrence: dp[i] = dp[i-1] + dp[i-2]
 * Base: dp[0] = 1, dp[1] = 1
 *
 * Time: O(n) - each state computed once
 * Space: O(n) - memo array + recursion stack
 */
class Solution {
private:
    unordered_map&lt;int, int&gt; memo;

    int helper(int n) {
        if (n &lt;= 1) return 1;

        if (memo.count(n)) {
            return memo[n];
        }

        memo[n] = helper(n - 1) + helper(n - 2);
        return memo[n];
    }

public:
    int climbStairs(int n) {
        return helper(n);
    }
};
```

Approach 3: Bottom-Up DP - O(n) time, O(n) space

```
int climbStairsDP(int n) {
    if (n &lt;= 1) return 1;

    vector&lt;int&gt; dp(n + 1);
    dp[0] = 1;
    dp[1] = 1;

    for (int i = 2; i &lt;= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }

    return dp[n];
}
```

Approach 4: Space-Optimized DP - O(n) time, O(1) space ⭐ OPTIMAL

```
/**
 * Space-optimized climbing stairs
 *
 * Observation: Only need last two values
 *
 * Time: O(n)
 * Space: O(1) - two variables
 */
int climbStairsOptimal(int n) {
    if (n &lt;= 1) return 1;

    int prev2 = 1;  // dp[i-2]
    int prev1 = 1;  // dp[i-1]

    for (int i = 2; i &lt;= n; i++) {
        int current = prev1 + prev2;
        prev2 = prev1;
        prev1 = current;
    }

    return prev1;
}
```

Problem 2: Coin Change (LeetCode 322)

All Approaches:

Approach 1: Recursive Backtracking - O(S^n) time (S = amount, n = coins)

```
int coinChangeRecursive(vector&lt;int&gt;&amp;amp; coins, int amount) {
    if (amount == 0) return 0;
    if (amount &lt; 0) return -1;

    int min_coins = INT_MAX;
```

```
    for (int coin : coins) {
        int result = coinChangeRecursive(coins, amount - coin);
        if (result &gt;= 0) {
            min_coins = min(min_coins, result + 1);
        }
    }

    return min_coins == INT_MAX ? -1 : min_coins;
}
```

Approach 2: Top-Down DP - O(amount × coins) time, O(amount) space

```
class Solution {
private:
    unordered_map&lt;int, int&gt; memo;

    int dp(vector&lt;int&gt;&amp; coins, int amount) {
        if (amount == 0) return 0;
        if (amount &lt; 0) return -1;

        if (memo.count(amount)) {
            return memo[amount];
        }

        int min_coins = INT_MAX;

        for (int coin : coins) {
            int result = dp(coins, amount - coin);
            if (result &gt;= 0) {
                min_coins = min(min_coins, result + 1);
            }
        }

        memo[amount] = (min_coins == INT_MAX) ? -1 : min_coins;
        return memo[amount];
    }

public:
    int coinChange(vector&lt;int&gt;&amp; coins, int amount) {
        return dp(coins, amount);
    }
};
```

Approach 3: Bottom-Up DP - O(amount × coins) time, O(amount) space ⭐ OPTIMAL

```
/**
 * Coin change with bottom-up DP
 *
 * State: dp[i] = min coins to make amount i
 * Recurrence: dp[i] = min(dp[i], dp[i - coin] + 1) for each coin
 * Base: dp[0] = 0
 *
 * Time: O(amount × coins)
 * Space: O(amount)
```

```
 */
int coinChange(vector&lt;int&gt;&amp; coins, int amount) {
    vector&lt;int&gt; dp(amount + 1, amount + 1);  // Initialize to impossible value
    dp[0] = 0;  // Base case

    for (int i = 1; i &lt;= amount; i++) {
        for (int coin : coins) {
            if (coin &lt;= i) {
                dp[i] = min(dp[i], dp[i - coin] + 1);
            }
        }
    }

    return dp[amount] &gt; amount ? -1 : dp[amount];
}
```

Detailed Walkthrough:

Example: coins = [1,2,5], amount = 11

```
Building DP table:

i=0: dp[0] = 0 (base case)

i=1: Try coins [1,2,5]
  coin=1: dp[1] = min(∞, dp[0]+1) = 1
  Result: dp[1] = 1

i=2: Try coins [1,2,5]
  coin=1: dp[2] = min(∞, dp[1]+1) = 2
  coin=2: dp[2] = min(2, dp[0]+1) = 1
  Result: dp[2] = 1

i=5: Try coins [1,2,5]
  coin=1: dp[5] = min(∞, dp[4]+1) = 3
  coin=2: dp[5] = min(3, dp[3]+1) = 3
  coin=5: dp[5] = min(3, dp[0]+1) = 1
  Result: dp[5] = 1

...continuing...

i=11: Try coins [1,2,5]
  coin=1: dp[11] = dp[10]+1 = 3
  coin=2: dp[11] = dp[9]+1 = 3
  coin=5: dp[11] = dp[6]+1 = 3
  Result: dp[11] = 3

Answer: 3 coins (5+5+1)
```

Problem 3: Longest Increasing Subsequence (LeetCode 300)

Approach 1: DP - O(n²) time, O(n) space

```
/**
 * LIS using dynamic programming
 *
 * State: dp[i] = length of LIS ending at index i
 * Recurrence: dp[i] = max(dp[j] + 1) for all j &lt; i where nums[j] &lt; nums[i]
 *
 * Time: O(n²)
 * Space: O(n)
 */
int lengthOfLIS_DP(vector&lt;int&gt;&amp; nums) {
    int n = nums.size();
    vector&lt;int&gt; dp(n, 1);  // Each element is subsequence of length 1
    int max_len = 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);
            }
        }
        max_len = max(max_len, dp[i]);
    }

    return max_len;
}
```

Approach 2: Binary Search + DP - O(n log n) time, O(n) space ⭐ OPTIMAL

```
/**
 * LIS using binary search optimization
 *
 * Maintain array tails where tails[i] = smallest ending element
 * of all increasing subsequences of length i+1
 *
 * Time: O(n log n)
 * Space: O(n)
 */
int lengthOfLIS(vector&lt;int&gt;&amp; nums) {
    vector&lt;int&gt; tails;

    for (int num : nums) {
        // Binary search for position to insert/replace
        auto it = lower_bound(tails.begin(), tails.end(), num);

        if (it == tails.end()) {
            tails.push_back(num);  // Extend sequence
        } else {
            *it = num;  // Replace to keep tail smaller
        }
    }
```

```
    return tails.size();
}
```

8.4 DP Pattern Categories

Linear DP (1D)

Problems: House Robber, Jump Game, Decode Ways

Template:

```
vector&lt;int&gt; dp(n);
dp[0] = base_value;

for (int i = 1; i &lt; n; i++) {
    dp[i] = function_of(dp[i-1], dp[i-2], ...);
}
```

Grid DP (2D)

Problems: Unique Paths, Minimum Path Sum, Edit Distance

Template:

```
vector&lt;vector&lt;int&gt;&gt; dp(m, vector&lt;int&gt;(n));
// Initialize first row and column

for (int i = 1; i &lt; m; i++) {
    for (int j = 1; j &lt; n; j++) {
        dp[i][j] = function_of(dp[i-1][j], dp[i][j-1], ...);
    }
}
```

Knapsack DP

Problems: 0/1 Knapsack, Partition Equal Subset Sum

Template:

```
vector&lt;vector&lt;int&gt;&gt; dp(n+1, vector&lt;int&gt;(capacity+1, 0));

for (int i = 1; i &lt;= n; i++) {
    for (int w = 1; w &lt;= capacity; w++) {
        if (weight[i] &lt;= w) {
            dp[i][w] = max(dp[i-1][w],
                           dp[i-1][w-weight[i]] + value[i]);
        } else {
            dp[i][w] = dp[i-1][w];
```

```
        }
    }
}
```

Interval DP

Problems: Longest Palindromic Substring, Matrix Chain Multiplication

Template:

```
vector&lt;vector&lt;int&gt;&gt; dp(n, vector&lt;int&gt;(n));

// Iterate by interval length
for (int len = 2; len &lt;= n; len++) {
    for (int i = 0; i &lt;= n - len; i++) {
        int j = i + len - 1;
        dp[i][j] = compute_from(dp[i][k], dp[k+1][j]);
    }
}
```

Pattern 9: Recursion & Backtracking

9.1 Core Understanding

Layman's Intuition

Imagine filling a Sudoku puzzle. At each empty cell, you try a number (1-9). If it leads to a valid solution, great! If you get stuck later, you backtrack—erase that number and try the next one. Backtracking is systematically trying all possibilities with the ability to undo (backtrack) when hitting dead ends.

Technical Foundation

Recursion: Function calling itself with smaller input
Backtracking: Recursion + pruning invalid branches + undoing choices

Template Structure:

```
function backtrack(state):
    if is_solution(state):
        record solution
        return

    if should_prune(state):
        return  // Optimization: skip invalid branches

    for choice in get_choices(state):
        make_choice(choice)       // Modify state
```

```
        backtrack(new_state)      // Recurse
        undo_choice(choice)       // Backtrack
```

When to Use:

- **Generate all combinations/permutations**
- **Constraint satisfaction:** Sudoku, N-Queens
- **Pathfinding:** All paths in maze
- **Subset problems:** Generate all subsets
- **Keywords:** "all possible", "generate", "find all solutions"

9.2 Complete Problem Solutions

Problem: Permutations (LeetCode 46)

Approach: Backtracking with Swapping

```
/**
 * Generate all permutations
 *
 * Approach: Backtracking by swapping elements
 *
 * Time: O(n! × n) - n! permutations, n to copy each
 * Space: O(n) - recursion depth
 */
class Solution {
private:
    void backtrack(vector&lt;int&gt;&amp; nums, int start,
                   vector&lt;vector&lt;int&gt;&gt;&amp; result) {
        // Base case: reached end → found permutation
        if (start == nums.size()) {
            result.push_back(nums);
            return;
        }

        // Try each element in remaining positions
        for (int i = start; i &lt; nums.size(); i++) {
            // Choose: swap to current position
            swap(nums[start], nums[i]);

            // Explore: recurse on next position
            backtrack(nums, start + 1, result);

            // Unchoose: backtrack (undo swap)
            swap(nums[start], nums[i]);
        }
    }

public:
    vector&lt;vector&lt;int&gt;&gt; permute(vector&lt;int&gt;&amp; nums) {
        vector&lt;vector&lt;int&gt;&gt; result;
```

```
        backtrack(nums, 0, result);
        return result;
    }
};
```

Detailed Execution Trace for [1,2,3]:

```
backtrack([1,2,3], 0)
├─ swap(0,0): [1,2,3]
│  └─ backtrack([1,2,3], 1)
│     ├─ swap(1,1): [1,2,3]
│     │  └─ backtrack([1,2,3], 2)
│     │     ├─ swap(2,2): [1,2,3] → ADD [1,2,3]
│     │     └─ swap(2,2): restore
│     ├─ swap(1,2): [1,3,2]
│     │  └─ backtrack([1,3,2], 2) → ADD [1,3,2]
│     └─ swap(1,2): restore to [1,2,3]
├─ swap(0,1): [2,1,3]
│  └─ ... generates [2,1,3], [2,3,1]
└─ swap(0,2): [3,2,1]
   └─ ... generates [3,2,1], [3,1,2]

Result: [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,2,1], [3,1,2]]
```

Problem: N-Queens (LeetCode 51)

```
/**
 * Solve N-Queens problem
 *
 * Place N queens on N×N board such that no two attack each other
 *
 * Backtracking with pruning:
 * - Try each column in current row
 * - Check if safe (no queens in same column, diagonals)
 * - Recurse on next row
 * - Backtrack if no solution
 *
 * Time: O(N!) - upper bound, pruning reduces actual complexity
 * Space: O(N²) - board
 */
class Solution {
private:
    bool isSafe(vector&lt;string&gt;&amp; board, int row, int col, int n) {
        // Check column
        for (int i = 0; i &lt; row; i++) {
            if (board[i][col] == 'Q') return false;
        }

        // Check upper-left diagonal
        for (int i = row - 1, j = col - 1; i &gt;= 0 &amp;&amp; j &gt;= 0; i--, j--) {
            if (board[i][j] == 'Q') return false;
        }
```

```
        // Check upper-right diagonal
        for (int i = row - 1, j = col + 1; i &gt;= 0 &amp;&amp; j &lt; n; i--, j++) {
            if (board[i][j] == 'Q') return false;
        }

        return true;
    }

    void backtrack(vector&lt;vector&lt;string&gt;&gt;&amp; result,
                   vector&lt;string&gt;&amp; board, int row, int n) {
        // Base case: all queens placed
        if (row == n) {
            result.push_back(board);
            return;
        }

        // Try placing queen in each column of current row
        for (int col = 0; col &lt; n; col++) {
            if (isSafe(board, row, col, n)) {
                // Choose: place queen
                board[row][col] = 'Q';

                // Explore: recurse on next row
                backtrack(result, board, row + 1, n);

                // Unchoose: remove queen (backtrack)
                board[row][col] = '.';
            }
        }
    }

public:
    vector&lt;vector&lt;string&gt;&gt; solveNQueens(int n) {
        vector&lt;vector&lt;string&gt;&gt; result;
        vector&lt;string&gt; board(n, string(n, '.'));
        backtrack(result, board, 0, n);
        return result;
    }
};
```

Optimization: Use Sets for O(1) Conflict Checking

```
class SolutionOptimized {
private:
    unordered_set&lt;int&gt; cols;       // Occupied columns
    unordered_set&lt;int&gt; diag1;      // Occupied \ diagonals (row - col)
    unordered_set&lt;int&gt; diag2;      // Occupied / diagonals (row + col)

    void backtrack(vector&lt;vector&lt;string&gt;&gt;&amp; result,
                   vector&lt;string&gt;&amp; board, int row, int n) {
        if (row == n) {
            result.push_back(board);
            return;
        }
```

```
        for (int col = 0; col &lt; n; col++) {
            int d1 = row - col;
            int d2 = row + col;

            // Check if position is safe (O(1) with sets)
            if (cols.count(col) || diag1.count(d1) || diag2.count(d2)) {
                continue;
            }

            // Choose
            board[row][col] = 'Q';
            cols.insert(col);
            diag1.insert(d1);
            diag2.insert(d2);

            // Explore
            backtrack(result, board, row + 1, n);

            // Unchoose
            board[row][col] = '.';
            cols.erase(col);
            diag1.erase(d1);
            diag2.erase(d2);
        }
    }

public:
    vector&lt;vector&lt;string&gt;&gt; solveNQueens(int n) {
        vector&lt;vector&lt;string&gt;&gt; result;
        vector&lt;string&gt; board(n, string(n, '.'));
        backtrack(result, board, 0, n);
        return result;
    }
};
```

Pattern 10: Monotonic Stack

10.1 Core Understanding

Layman's Intuition

Imagine standing in line at a theme park. You can see ahead to find the next person taller than you. Instead of checking everyone one-by-one, you maintain a "stack" of people in increasing height order. When someone taller arrives, you pop everyone shorter—they've found their "next greater element." Efficient!

Technical Foundation

Definition: A stack that maintains elements in monotonically increasing or decreasing order, enabling O(n) solutions for "next greater/smaller" problems.

Two Types:

1. **Monotonic Increasing Stack:** Elements increase from bottom to top
2. **Monotonic Decreasing Stack:** Elements decrease from bottom to top

Key Insight: Each element pushed/popped at most once → O(n) total

When to Use:

- **Next Greater Element (NGE)**
- **Next Smaller Element (NSE)**
- **Largest Rectangle problems**
- **Stock span problems**

10.2 Template & Implementation

Monotonic Decreasing Stack (for Next Greater Element)

```
/**
 * Find next greater element for each array element
 *
 * Approach: Monotonic decreasing stack
 *
 * Invariant: Stack contains elements in decreasing order
 * When new element &gt; stack top: found NGE for stack elements
 *
 * Time: O(n) - each element pushed/popped once
 * Space: O(n) - stack
 */
vector&lt;int&gt; nextGreaterElement(vector&lt;int&gt;&amp; nums) {
    int n = nums.size();
    vector&lt;int&gt; result(n, -1);  // -1 means no greater element
    stack&lt;int&gt; stk;  // Store indices

    for (int i = 0; i &lt; n; i++) {
        // Pop elements smaller than current
        while (!stk.empty() &amp;&amp; nums[stk.top()] &lt; nums[i]) {
            int idx = stk.top();
            stk.pop();
            result[idx] = nums[i];  // Found NGE
        }

        // Push current index
        stk.push(i);
    }
```

```
    return result;
}
```

Problem: Daily Temperatures (LeetCode 739)

```
/**
 * Find number of days until warmer temperature
 *
 * Approach: Monotonic decreasing stack
 *
 * Time: O(n)
 * Space: O(n)
 */
vector&lt;int&gt; dailyTemperatures(vector&lt;int&gt;&amp; temperatures) {
    int n = temperatures.size();
    vector&lt;int&gt; result(n, 0);
    stack&lt;int&gt; stk;  // Indices of days with pending warmer day

    for (int i = 0; i &lt; n; i++) {
        // Found warmer day for days in stack
        while (!stk.empty() &amp;&amp;
               temperatures[stk.top()] &lt; temperatures[i]) {
            int prevDay = stk.top();
            stk.pop();
            result[prevDay] = i - prevDay;
        }

        stk.push(i);
    }

    return result;  // Remaining elements have 0 (no warmer day)
}
```

Example Trace:

```
Input: [73, 74, 75, 71, 69, 72, 76, 73]

i=0: stk=[0]
i=1: 74 &gt; 73 → pop 0, result[0]=1, stk=[1]
i=2: 75 &gt; 74 → pop 1, result[1]=1, stk=[2]
i=3: 71 &lt; 75 → stk=[2,3]
i=4: 69 &lt; 71 → stk=[2,3,4]
i=5: 72 &gt; 69,71 → pop 4,3, result[4]=1, result[3]=2, stk=[2,5]
i=6: 76 &gt; 72,75 → pop 5,2, result[5]=1, result[2]=4, stk=[6]
i=7: 73 &lt; 76 → stk=[6,7]

Result: [1, 1, 4, 2, 1, 1, 0, 0]
```

Pattern 11: Union-Find (Disjoint Set Union)

11.1 Core Understanding

Layman's Intuition

Imagine managing friend groups on social media. Initially, everyone is their own group. When two people become friends, merge their groups. To check if two people are in the same group, trace up to their group leaders and compare. This is Union-Find—efficiently tracking disjoint sets!

Technical Foundation

Operations:

1. **Find(x):** Which set does x belong to?
2. **Union(x, y):** Merge sets containing x and y

Optimizations:

- **Path Compression:** Make nodes point directly to root during Find
- **Union by Rank:** Attach smaller tree to larger tree

Complexity with Both Optimizations:

- **Find/Union:** O(α(n)) ≈ O(1) where α is inverse Ackermann (extremely slow-growing)
- **Space:** O(n)

When to Use:

- **Connected components:** Count/identify disjoint groups
- **Cycle detection:** In undirected graphs
- **Minimum spanning tree:** Kruskal's algorithm
- **Dynamic connectivity:** Queries about connectivity

11.2 Implementation

```
/**
 * Union-Find data structure with path compression and union by rank
 *
 * Operations:
 * - find(x): O(α(n)) ≈ O(1)
 * - unite(x, y): O(α(n)) ≈ O(1)
 *
 * α(n) is inverse Ackermann function (grows incredibly slowly)
 */
class UnionFind {
private:
    vector&lt;int&gt; parent;
    vector&lt;int&gt; rank;
    int num_components;
```

```
public:
    UnionFind(int n) : parent(n), rank(n, 0), num_components(n) {
        // Initially each element is its own parent
        for (int i = 0; i &lt; n; i++) {
            parent[i] = i;
        }
    }

    /**
     * Find root of element x with path compression
     *
     * Path compression: make all nodes on path point directly to root
     * This flattens the tree structure for faster future finds
     */
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);  // Path compression
        }
        return parent[x];
    }

    /**
     * Union sets containing x and y
     *
     * Union by rank: attach smaller tree to larger tree
     * Keeps tree height logarithmic
     *
     * Returns: true if x and y were in different sets (union performed)
     */
    bool unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX == rootY) {
            return false;  // Already in same set
        }

        // Union by rank
        if (rank[rootX] &lt; rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] &gt; rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }

        num_components--;
        return true;
    }

    /**
     * Check if x and y are in same set
     */
    bool connected(int x, int y) {
```

```
        return find(x) == find(y);
    }

    /**
     * Get number of disjoint sets
     */
    int getNumComponents() {
        return num_components;
    }
};
```

Problem: Number of Provinces (LeetCode 547)

```
/**
 * Count number of provinces (connected components)
 *
 * Approach: Union-Find
 *
 * Time: O(n² × α(n)) ≈ O(n²)
 * Space: O(n)
 */
int findCircleNum(vector&lt;vector&lt;int&gt;&gt;&amp; isConnected) {
    int n = isConnected.size();
    UnionFind uf(n);

    // Unite directly connected cities
    for (int i = 0; i &lt; n; i++) {
        for (int j = i + 1; j &lt; n; j++) {
            if (isConnected[i][j] == 1) {
                uf.unite(i, j);
            }
        }
    }

    return uf.getNumComponents();
}
```

Pattern 12: Topological Sort

12.1 Core Understanding

Layman's Intuition

Imagine ordering college courses with prerequisites. You can't take Advanced Algorithms before Data Structures. Topological sort finds a valid order to take all courses, respecting prerequisites. It's like untangling dependencies!

Technical Foundation

Definition: Linear ordering of vertices in directed acyclic graph (DAG) such that for every edge u→v, u comes before v.

Requirements:

- Graph must be **DAG** (no cycles)
- May have multiple valid orderings

Two Algorithms:

1. **Kahn's Algorithm (BFS-based):** Remove nodes with in-degree 0 iteratively
2. **DFS-based:** Post-order DFS, reverse the finish order

When to Use:

- **Course scheduling:** Prerequisites
- **Build systems:** Dependency resolution
- **Task scheduling:** Order tasks with dependencies

12.2 Complete Implementations

Approach 1: Kahn's Algorithm (BFS)

```
/**
 * Topological sort using Kahn's algorithm
 *
 * Algorithm:
 * 1. Compute in-degree for each node
 * 2. Add all nodes with in-degree 0 to queue
 * 3. Process nodes, decrement neighbors' in-degrees
 * 4. Add newly zero in-degree nodes to queue
 *
 * Cycle detection: If topological order has &lt; n nodes, cycle exists
 *
 * Time: O(V + E)
 * Space: O(V)
 */
vector&lt;int&gt; topologicalSort(int numCourses, vector&lt;vector&lt;int&gt;&gt;&amp; pr
    // Build adjacency list and in-degree array
    vector&lt;vector&lt;int&gt;&gt; graph(numCourses);
    vector&lt;int&gt; indegree(numCourses, 0);

    for (auto&amp; prereq : prerequisites) {
        int course = prereq[0];
        int prerequisite = prereq[1];
        graph[prerequisite].push_back(course);
        indegree[course]++;
    }

    // Initialize queue with nodes of in-degree 0
```

```
    queue&lt;int&gt; q;
    for (int i = 0; i &lt; numCourses; i++) {
        if (indegree[i] == 0) {
            q.push(i);
        }
    }

    // Process nodes in topological order
    vector&lt;int&gt; result;

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        result.push_back(node);

        // Decrement in-degree of neighbors
        for (int neighbor : graph[node]) {
            indegree[neighbor]--;
            if (indegree[neighbor] == 0) {
                q.push(neighbor);
            }
        }
    }

    // Check for cycle
    if (result.size() != numCourses) {
        return {};  // Cycle detected, no valid ordering
    }

    return result;
}
```

Approach 2: DFS-based

```
/**
 * Topological sort using DFS
 *
 * Algorithm:
 * 1. Perform DFS from each unvisited node
 * 2. Add node to stack in post-order (after visiting children)
 * 3. Reverse stack for topological order
 *
 * Time: O(V + E)
 * Space: O(V)
 */
class Solution {
private:
    bool hasCycle = false;

    void dfs(int node, vector&lt;vector&lt;int&gt;&gt;&amp; graph,
             vector&lt;int&gt;&amp; state, stack&lt;int&gt;&amp; stk) {
        // States: 0=unvisited, 1=visiting, 2=visited
        state[node] = 1;  // Mark as visiting

        for (int neighbor : graph[node]) {
```

```
            if (state[neighbor] == 1) {
                // Back edge → cycle
                hasCycle = true;
                return;
            }

            if (state[neighbor] == 0) {
                dfs(neighbor, graph, state, stk);
            }
        }

        state[node] = 2;  // Mark as visited
        stk.push(node);   // Add to stack in post-order
    }

public:
    vector&lt;int&gt; topologicalSortDFS(int n, vector&lt;vector&lt;int&gt;&gt;&amp; edge
        // Build adjacency list
        vector&lt;vector&lt;int&gt;&gt; graph(n);
        for (auto&amp; edge : edges) {
            graph[edge[1]].push_back(edge[0]);
        }

        vector&lt;int&gt; state(n, 0);
        stack&lt;int&gt; stk;

        // DFS from each unvisited node
        for (int i = 0; i &lt; n; i++) {
            if (state[i] == 0) {
                dfs(i, graph, state, stk);
                if (hasCycle) return {};
            }
        }

        // Pop stack to get topological order
        vector&lt;int&gt; result;
        while (!stk.empty()) {
            result.push_back(stk.top());
            stk.pop();
        }

        return result;
    }
};
```

Pattern 13: Trie (Prefix Tree)

13.1 Core Understanding

Layman's Intuition

Imagine a dictionary where words share common prefixes. Instead of storing "cat", "car", "card" separately, you store: c→a→(t, r→d). This tree structure (Trie) enables fast prefix searches and autocomplete.

Technical Foundation

Operations:

- **Insert:** O(m) where m = word length
- **Search:** O(m)
- **StartsWith:** O(m)
- **Space:** O(alphabet_size × total_characters)

When to Use:

- **Autocomplete:** Find all words with prefix
- **Spell checker:** Suggest corrections
- **IP routing:** Longest prefix match
- **Word games:** Boggle, Scrabble

13.2 Complete Implementation

```
/**
 * Trie (Prefix Tree) implementation
 *
 * Supports:
 * - insert(word): O(m) where m = word length
 * - search(word): O(m)
 * - startsWith(prefix): O(m)
 *
 * Space: O(ALPHABET_SIZE × N × L) where N=words, L=avg length
 */
class TrieNode {
public:
    unordered_map&lt;char, TrieNode*&gt; children;
    bool is_end_of_word;

    TrieNode() : is_end_of_word(false) {}
};

class Trie {
private:
    TrieNode* root;

public:
    Trie() {
        root = new TrieNode();
    }
```

```
    /**
     * Insert word into trie
     * Time: O(m) where m = word length
     */
    void insert(string word) {
        TrieNode* node = root;

        for (char ch : word) {
            if (!node-&gt;children.count(ch)) {
                node-&gt;children[ch] = new TrieNode();
            }
            node = node-&gt;children[ch];
        }

        node-&gt;is_end_of_word = true;
    }

    /**
     * Search for exact word
     * Time: O(m)
     */
    bool search(string word) {
        TrieNode* node = root;

        for (char ch : word) {
            if (!node-&gt;children.count(ch)) {
                return false;
            }
            node = node-&gt;children[ch];
        }

        return node-&gt;is_end_of_word;
    }

    /**
     * Check if any word starts with prefix
     * Time: O(m)
     */
    bool startsWith(string prefix) {
        TrieNode* node = root;

        for (char ch : prefix) {
            if (!node-&gt;children.count(ch)) {
                return false;
            }
            node = node-&gt;children[ch];
        }

        return true;
    }

    /**
     * Delete word from trie
     * Time: O(m)
     */
    bool deleteWord(string word) {
```

```
        return deleteHelper(root, word, 0);
    }

private:
    bool deleteHelper(TrieNode* node, const string&amp;amp; word, int index) {
        if (index == word.length()) {
            if (!node-&gt;is_end_of_word) {
                return false;  // Word not in trie
            }
            node-&gt;is_end_of_word = false;
            return node-&gt;children.empty();  // Can delete if no children
        }

        char ch = word[index];
        if (!node-&gt;children.count(ch)) {
            return false;
        }

        TrieNode* child = node-&gt;children[ch];
        bool shouldDeleteChild = deleteHelper(child, word, index + 1);

        if (shouldDeleteChild) {
            node-&gt;children.erase(ch);
            return node-&gt;children.empty() &amp;amp;&amp;amp; !node-&gt;is_end_of_word;
        }

        return false;
    }
};
```

This comprehensive Volume 3 concludes the Ultimate Coding Patterns series with production-grade implementations, optimization techniques, and systematic mastery frameworks for all critical patterns.