Ultimate Coding Patterns Mastery - Volume 2

Fast/Slow Pointers, DFS, BFS, Binary Search, and Merge Intervals

Pattern 3: Fast & Slow Pointers (Floyd's Cycle Detection)

3.1 Core Understanding

Layman's Intuition

Imagine two runners on a circular track. One runs twice as fast as the other. If the track is truly circular, the faster runner will eventually lap the slower one—they'll meet. But if the track ends (no loop), the faster runner will reach the end first and they'll never meet. This is how we detect cycles in linked lists!

Technical Foundation

Definition: Fast & slow pointers (also called "tortoise and hare") is a technique where two pointers traverse a data structure at different speeds to detect cycles, find middle elements, or solve linked list problems efficiently.

Mathematical Proof of Cycle Detection:

Let cycle start at distance k from head, cycle length L.

Meeting Point:

- When slow enters cycle (traveled k steps), fast is k steps ahead
- Fast gains 1 step per iteration on slow
- They meet after $L - (k \mod L)$ steps inside cycle

Cycle Start Detection:

- Reset one pointer to head, both move at same speed
- They meet at cycle start (distance k from head)

Proof: If meeting point is m from cycle start:

- Distance from head to cycle start: k
- Slow traveled: $k + m$
- Fast traveled: $2(k + m) = k + m + nL$ for some n
- Simplifies to: $k = nL - m$

- Starting from head and meeting point simultaneously → meet at cycle start

When to Use:

- **Linked list cycle detection**
- **Finding middle of linked list**
- **Palindrome linked list checking**
- **Finding cycle start**
- **Happy number problem**

3.2 Associated Data Structures

Linked Lists (Primary)

```
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
};

// Floyd's cycle detection
ListNode* detectCycle(ListNode* head) {
    ListNode* slow = head;
    ListNode* fast = head;

    // Phase 1: Detect if cycle exists
    while (fast != nullptr &amp;&amp; fast-&gt;next != nullptr) {
        slow = slow-&gt;next;
        fast = fast-&gt;next-&gt;next;

        if (slow == fast) {
            // Cycle detected
            // Phase 2: Find cycle start
            slow = head;
            while (slow != fast) {
                slow = slow-&gt;next;
                fast = fast-&gt;next;
            }
            return slow;  // Cycle start
        }
    }

    return nullptr;  // No cycle
}
```

3.3 Complete Problem Solutions

Problem 1: Linked List Cycle (LeetCode 141)

All Approaches with Full Implementations:

Approach 1: Hash Set - O(n) time, O(n) space

```
bool hasCycle_HashSet(ListNode *head) {
    unordered_set&lt;ListNode*&gt; visited;
    ListNode* current = head;

    while (current != nullptr) {
        if (visited.count(current)) {
            return true;  // Cycle detected
        }
        visited.insert(current);
        current = current-&gt;next;
    }

    return false;  // No cycle
}
```

Approach 2: Fast & Slow Pointers - O(n) time, O(1) space ⭐ OPTIMAL

```
/**
 * Detect if linked list has cycle using Floyd's algorithm
 *
 * Time: O(n)
 * - If no cycle: fast reaches end in n/2 steps
 * - If cycle exists: pointers meet in at most n steps
 *
 * Space: O(1) - only two pointers
 *
 * Proof of correctness:
 * - Fast pointer moves 2x speed of slow
 * - Inside cycle, fast catches up to slow by 1 step per iteration
 * - Gap closes linearly → guaranteed meeting
 */
bool hasCycle(ListNode *head) {
    if (!head || !head-&gt;next) return false;

    ListNode* slow = head;
    ListNode* fast = head;

    while (fast != nullptr &amp;&amp; fast-&gt;next != nullptr) {
        slow = slow-&gt;next;        // Move 1 step
        fast = fast-&gt;next-&gt;next;  // Move 2 steps

        if (slow == fast) {
            return true;  // Pointers met → cycle exists
        }
    }
    return false;  // Fast reached end → no cycle
}
```

Problem 2: Find Middle of Linked List (LeetCode 876)

```
/**
 * Find middle node of linked list
 *
 * Approach: When fast reaches end, slow is at middle
 *
 * For even-length list, returns second middle node
 * Example: 1→2→3→4 returns 3
 *
 * Time: O(n) - one pass
 * Space: O(1) - two pointers
 */
ListNode* middleNode(ListNode* head) {
    ListNode* slow = head;
    ListNode* fast = head;

    while (fast != nullptr &amp;&amp; fast-&gt;next != nullptr) {
        slow = slow-&gt;next;
        fast = fast-&gt;next-&gt;next;
    }

    return slow;  // Middle node
}
```

Variation: First Middle for Even Length

```
ListNode* middleNodeFirstOfTwo(ListNode* head) {
    ListNode* slow = head;
    ListNode* fast = head;

    while (fast-&gt;next != nullptr &amp;&amp; fast-&gt;next-&gt;next != nullptr) {
        slow = slow-&gt;next;
        fast = fast-&gt;next-&gt;next;
    }

    return slow;  // For even, returns first middle
}
```

Problem 3: Happy Number (LeetCode 202)

```
/**
 * Determine if number is happy
 *
 * Happy number: sum of squares of digits eventually reaches 1
 * Unhappy: enters cycle that doesn't include 1
 *
 * Approach: Treat as cycle detection problem
 * - Fast pointer: computes sum twice per iteration
 * - Slow pointer: computes sum once per iteration
 * - If cycle exists and doesn't include 1 → unhappy
 *
 * Time: O(log n) - number of digits reduces
 * Space: O(1) - two pointers
 */
class Solution {
private:
    int sumSquares(int n) {
        int sum = 0;
        while (n &gt; 0) {
            int digit = n % 10;
            sum += digit * digit;
            n /= 10;
        }
        return sum;
    }

public:
    bool isHappy(int n) {
        int slow = n;
        int fast = n;

        do {
            slow = sumSquares(slow);              // Move 1 step
            fast = sumSquares(sumSquares(fast));  // Move 2 steps
        } while (slow != fast);

        return slow == 1;  // If met at 1 → happy
    }
};
```

Pattern 4: Depth-First Search (DFS)

4.1 Core Understanding

Layman's Intuition

Imagine exploring a maze by always going as deep as possible down one path before backtracking. You keep going forward until you hit a dead end, then you backtrack to the last intersection and try another path. This "go deep, then backtrack" approach is DFS.

Technical Foundation

Definition: DFS is a graph/tree traversal algorithm that explores as far as possible along each branch before backtracking.

Key Properties:

- **LIFO (Last In First Out):** Uses stack (implicit via recursion or explicit)
- **Backtracking:** Returns to previous state when dead end reached
- **Memory:** O(h) where h is maximum depth
- **Order:** Pre-order, in-order, post-order for trees

When to Use:

- **Explore all paths:** Count all paths, find all solutions
- **Backtracking problems:** N-Queens, Sudoku, permutations
- **Connected components:** Find islands, regions
- **Topological sort:** Prerequisites ordering
- **Cycle detection:** In directed graphs

4.2 Implementation Templates

Recursive DFS (Most Common)

```
void dfs(Node* node, unordered_set&lt;Node*&gt;&amp; visited) {
    if (!node || visited.count(node)) return;

    // Mark visited
    visited.insert(node);

    // Process current node
    process(node);

    // Recurse on neighbors
    for (Node* neighbor : node-&gt;neighbors) {
        dfs(neighbor, visited);
    }
}
```

Iterative DFS with Stack

```
void dfsIterative(Node* start) {
    stack&lt;Node*&gt; stk;
    unordered_set&lt;Node*&gt; visited;

    stk.push(start);

    while (!stk.empty()) {
        Node* node = stk.top();
        stk.pop();

        if (visited.count(node)) continue;

        visited.insert(node);
        process(node);

        // Push neighbors (reverse order for left-to-right)
        for (auto it = node-&gt;neighbors.rbegin();
             it != node-&gt;neighbors.rend(); ++it) {
            if (!visited.count(*it)) {
                stk.push(*it);
            }
        }
    }
}
```

DFS on 2D Grid

```
void dfsGrid(vector&lt;vector&lt;int&gt;&gt;&amp; grid, int row, int col,
             vector&lt;vector&lt;bool&gt;&gt;&amp; visited) {
    int rows = grid.size();
    int cols = grid[0].size();

    // Base cases
    if (row &lt; 0 || row &gt;= rows || col &lt; 0 || col &gt;= cols) return;
    if (visited[row][col] || grid[row][col] == 0) return;

    // Mark visited
    visited[row][col] = true;

    // Process cell
    process(grid[row][col]);

    // Explore 4 directions
    int directions[4][2] = {{-1,0}, {1,0}, {0,-1}, {0,1}};
    for (auto&amp; dir : directions) {
        int newRow = row + dir[0];
        int newCol = col + dir[1];
        dfsGrid(grid, newRow, newCol, visited);
    }
}
```

4.3 Complete Problem Solutions

Problem: Number of Islands (LeetCode 200)

All Approaches:

Approach 1: DFS Recursive - O(m×n) time, O(m×n) space

```
class Solution {
private:
    void dfs(vector&lt;vector&lt;char&gt;&gt;&amp; grid, int row, int col) {
        int rows = grid.size();
        int cols = grid[0].size();

        // Boundary check and water check
        if (row &lt; 0 || row &gt;= rows || col &lt; 0 || col &gt;= cols ||
            grid[row][col] == '0') {
            return;
        }

        // Mark as visited by converting to '0'
        grid[row][col] = '0';

        // Explore all 4 directions
        dfs(grid, row + 1, col);  // Down
        dfs(grid, row - 1, col);  // Up
        dfs(grid, row, col + 1);  // Right
        dfs(grid, row, col - 1);  // Left
    }

public:
    int numIslands(vector&lt;vector&lt;char&gt;&gt;&amp; grid) {
        if (grid.empty()) return 0;

        int rows = grid.size();
        int cols = grid[0].size();
        int island_count = 0;

        for (int r = 0; r &lt; rows; r++) {
            for (int c = 0; c &lt; cols; c++) {
                if (grid[r][c] == '1') {
                    // Found unvisited land → new island
                    island_count++;
                    dfs(grid, r, c);  // Mark entire island
                }
            }
        }

        return island_count;
    }
};
```

Approach 2: DFS Iterative with Stack

```
int numIslandsIterative(vector&lt;vector&lt;char&gt;&gt;&amp; grid) {
    if (grid.empty()) return 0;
    int rows = grid.size();
    int cols = grid[0].size();
    int island_count = 0;

    for (int r = 0; r &lt; rows; r++) {
        for (int c = 0; c &lt; cols; c++) {
            if (grid[r][c] == '1') {
                island_count++;

                // DFS using stack
                stack&lt;pair&lt;int,int&gt;&gt; stk;
                stk.push({r, c});
                grid[r][c] = '0';  // Mark visited

                while (!stk.empty()) {
                    auto [row, col] = stk.top();
                    stk.pop();

                    // Check 4 directions
                    int directions[4][2] = {{-1,0}, {1,0}, {0,-1}, {0,1}};
                    for (auto&amp; dir : directions) {
                        int newRow = row + dir[0];
                        int newCol = col + dir[1];

                        if (newRow &gt;= 0 &amp;&amp; newRow &lt; rows &amp;&amp;
                            newCol &gt;= 0 &amp;&amp; newCol &lt; cols &amp;&amp;
                            grid[newRow][newCol] == '1') {
                            stk.push({newRow, newCol});
                            grid[newRow][newCol] = '0';
                        }
                    }
                }
            }
        }
    }

    return island_count;
}
```

Problem: Clone Graph (LeetCode 133)

```
/**
 * Clone undirected graph
 *
 * Approach: DFS with hash map to track cloned nodes
 *
 * Time: O(N + E) where N=nodes, E=edges
 * Space: O(N) for hash map and recursion stack
 */
class Node {
public:
    int val;
    vector&lt;Node*&gt; neighbors;
    Node(int _val) : val(_val) {}
};

class Solution {
private:
    unordered_map&lt;Node*, Node*&gt; cloned;

    Node* dfs(Node* node) {
        if (!node) return nullptr;

        // If already cloned, return clone
        if (cloned.count(node)) {
            return cloned[node];
        }

        // Create clone
        Node* clone = new Node(node-&gt;val);
        cloned[node] = clone;

        // Clone neighbors recursively
        for (Node* neighbor : node-&gt;neighbors) {
            clone-&gt;neighbors.push_back(dfs(neighbor));
        }

        return clone;
    }

public:
    Node* cloneGraph(Node* node) {
        return dfs(node);
    }
};
```

Problem: Path Sum II (LeetCode 113)

```
/**
 * Find all root-to-leaf paths with given sum
 *
 * Approach: DFS with backtracking
 *
 * Time: O(N) to visit all nodes
 * Space: O(H) for recursion stack (H=height)
 */
class Solution {
private:
    void dfs(TreeNode* node, int targetSum,
             vector&lt;int&gt;&amp; path, vector&lt;vector&lt;int&gt;&gt;&amp; result) {
        if (!node) return;

        // Add current node to path
        path.push_back(node-&gt;val);

        // Check if leaf with target sum
        if (!node-&gt;left &amp;&amp; !node-&gt;right &amp;&amp;
            targetSum == node-&gt;val) {
            result.push_back(path);
        } else {
            // Recurse on children
            dfs(node-&gt;left, targetSum - node-&gt;val, path, result);
            dfs(node-&gt;right, targetSum - node-&gt;val, path, result);
        }

        // Backtrack: remove current node
        path.pop_back();
    }

public:
    vector&lt;vector&lt;int&gt;&gt; pathSum(TreeNode* root, int targetSum) {
        vector&lt;vector&lt;int&gt;&gt; result;
        vector&lt;int&gt; path;
        dfs(root, targetSum, path, result);
        return result;
    }
};
```

Pattern 5: Breadth-First Search (BFS)

5.1 Core Understanding

Layman's Intuition

Imagine dropping a stone in a pond—ripples spread outward in expanding circles. BFS explores a graph/tree similarly: first visit all nodes one step away, then all nodes two steps away, and so on. It explores level by level, like ripples.

Technical Foundation

Definition: BFS explores nodes level-by-level, visiting all neighbors before moving to next level.

Key Properties:

- **FIFO (First In First Out):** Uses queue
- **Level-order:** Processes nodes by distance from start
- **Shortest path:** In unweighted graphs
- **Memory:** O(w) where w is maximum width

When to Use:

- **Shortest path:** Unweighted graphs
- **Level-order traversal:** Tree level by level
- **Minimum steps:** State-space search
- **Connectivity:** Connected components

5.2 Implementation Templates

Standard BFS

```
void bfs(Node* start) {
    queue&lt;Node*&gt; q;
    unordered_set&lt;Node*&gt; visited;

    q.push(start);
    visited.insert(start);

    while (!q.empty()) {
        Node* node = q.front();
        q.pop();

        process(node);

        for (Node* neighbor : node-&gt;neighbors) {
            if (!visited.count(neighbor)) {
                visited.insert(neighbor);
                q.push(neighbor);
            }
        }
    }
}
```

Level-by-Level BFS

```
void bfsLevels(Node* start) {
    queue&lt;Node*&gt; q;
    unordered_set&lt;Node*&gt; visited;

    q.push(start);
    visited.insert(start);
    int level = 0;

    while (!q.empty()) {
        int level_size = q.size();

        // Process entire level
        for (int i = 0; i &lt; level_size; i++) {
            Node* node = q.front();
            q.pop();

            process(node, level);

            for (Node* neighbor : node-&gt;neighbors) {
                if (!visited.count(neighbor)) {
                    visited.insert(neighbor);
                    q.push(neighbor);
                }
            }
        }
        level++;
    }
}
```

5.3 Complete Problem Solutions

Problem: Binary Tree Level Order Traversal (LeetCode 102)

```
/**
 * Level-order traversal of binary tree
 *
 * Approach: BFS with level tracking
 *
 * Time: O(N) - visit each node once
 * Space: O(W) - queue holds at most W nodes (width)
 */
vector&lt;vector&lt;int&gt;&gt; levelOrder(TreeNode* root) {
    vector&lt;vector&lt;int&gt;&gt; result;
    if (!root) return result;

    queue&lt;TreeNode*&gt; q;
    q.push(root);

    while (!q.empty()) {
        int level_size = q.size();
        vector&lt;int&gt; current_level;

        // Process all nodes at current level
        for (int i = 0; i &lt; level_size; i++) {
            TreeNode* node = q.front();
            q.pop();

            current_level.push_back(node-&gt;val);

            // Add children for next level
            if (node-&gt;left) q.push(node-&gt;left);
            if (node-&gt;right) q.push(node-&gt;right);
        }

        result.push_back(current_level);
    }

    return result;
}
```

Problem: Shortest Path in Binary Matrix (LeetCode 1091)

```
/**
 * Find shortest path from top-left to bottom-right
 * Can move in 8 directions, only through 0s
 *
 * Approach: BFS for shortest path
 *
 * Time: O(N) where N = total cells
 * Space: O(N) for queue and visited
 */
int shortestPathBinaryMatrix(vector&lt;vector&lt;int&gt;&gt;&amp; grid) {
    int n = grid.size();

    // Check if start or end is blocked
    if (grid[0][0] == 1 || grid[n-1][n-1] == 1) {
        return -1;
    }

    // BFS
    queue&lt;tuple&lt;int,int,int&gt;&gt; q;  // (row, col, distance)
    q.push({0, 0, 1});
    grid[0][0] = 1;  // Mark visited

    // 8 directions
    int directions[8][2] = {
        {-1,-1}, {-1,0}, {-1,1},
        {0,-1},          {0,1},
        {1,-1},  {1,0},  {1,1}
    };

    while (!q.empty()) {
        auto [row, col, dist] = q.front();
        q.pop();

        // Reached destination
        if (row == n-1 &amp;&amp; col == n-1) {
            return dist;
        }

        // Explore 8 neighbors
        for (auto&amp; dir : directions) {
            int newRow = row + dir[0];
            int newCol = col + dir[1];

            if (newRow &gt;= 0 &amp;&amp; newRow &lt; n &amp;&amp;
                newCol &gt;= 0 &amp;&amp; newCol &lt; n &amp;&amp;
                grid[newRow][newCol] == 0) {
                q.push({newRow, newCol, dist + 1});
                grid[newRow][newCol] = 1;  // Mark visited
            }
        }
    }
    return -1;  // No path found
}
```

Pattern 6: Binary Search

6.1 Core Understanding

Layman's Intuition

Looking for a word in a dictionary? You don't check every page. You open to the middle, see if your word comes before or after, then repeat on the relevant half. Each step eliminates half the remaining pages. That's binary search!

Technical Foundation

Definition: Binary search finds target in sorted array by repeatedly dividing search space in half.

Time Complexity: O(log n)
Space Complexity: O(1) iterative, O(log n) recursive

Invariant: Target, if exists, is always within [left, right]

When to Use:

- **Sorted array:** Search, find boundary
- **Search space:** Answer lies in range [min, max]
- **Monotonic function:** Can determine if guess is too high/low
- **Keywords:** "sorted", "find first/last", "minimize/maximize"

6.2 Implementation Templates

Standard Binary Search

```
int binarySearch(vector&lt;int&gt;&amp; arr, int target) {
    int left = 0;
    int right = arr.size() - 1;

    while (left &lt;= right) {
        int mid = left + (right - left) / 2;  // Prevent overflow

        if (arr[mid] == target) {
            return mid;  // Found
        } else if (arr[mid] &lt; target) {
            left = mid + 1;  // Search right half
        } else {
            right = mid - 1;  // Search left half
        }
    }
    return -1;  // Not found
}
```

Find First Occurrence

```
int findFirst(vector&lt;int&gt;&amp; arr, int target) {
    int left = 0;
    int right = arr.size() - 1;
    int result = -1;

    while (left &lt;= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            result = mid;        // Record position
            right = mid - 1;     // Continue searching left
        } else if (arr[mid] &lt; target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return result;
}
```

Find Last Occurrence

```
int findLast(vector&lt;int&gt;&amp; arr, int target) {
    int left = 0;
    int right = arr.size() - 1;
    int result = -1;

    while (left &lt;= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            result = mid;        // Record position
            left = mid + 1;      // Continue searching right
        } else if (arr[mid] &lt; target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return result;
}
```

6.3 Complete Problem Solutions

Problem: Search in Rotated Sorted Array (LeetCode 33)

```
/**
 * Search in rotated sorted array
 *
 * Key insight: At least one half is always sorted
 *
 * Time: O(log n)
 * Space: O(1)
 */
int search(vector&lt;int&gt;&amp; nums, int target) {
    int left = 0;
    int right = nums.size() - 1;

    while (left &lt;= right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] == target) {
            return mid;
        }

        // Determine which half is sorted
        if (nums[left] &lt;= nums[mid]) {
            // Left half is sorted
            if (nums[left] &lt;= target &amp;&amp; target &lt; nums[mid]) {
                right = mid - 1;  // Target in left half
            } else {
                left = mid + 1;   // Target in right half
            }
        } else {
            // Right half is sorted
            if (nums[mid] &lt; target &amp;&amp; target &lt;= nums[right]) {
                left = mid + 1;   // Target in right half
            } else {
                right = mid - 1;  // Target in left half
            }
        }
    }

    return -1;
}
```

Problem: Find Minimum in Rotated Sorted Array (LeetCode 153)

```
/**
 * Find minimum element in rotated sorted array
 *
 * Approach: Binary search comparing mid with right
 *
 * Time: O(log n)
 * Space: O(1)
 */
int findMin(vector&lt;int&gt;&amp; nums) {
    int left = 0;
    int right = nums.size() - 1;

    while (left &lt; right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] &gt; nums[right]) {
            // Minimum is in right half
            left = mid + 1;
        } else {
            // Minimum is in left half (including mid)
            right = mid;
        }
    }

    return nums[left];  // left == right at end
}
```

Pattern 7: Merge Intervals

7.1 Core Understanding

Layman's Intuition

Imagine your calendar with multiple appointments. Some overlap—you want to combine them into single blocks of busy time. You'd sort by start time, then merge any that overlap. That's interval merging!

Technical Foundation

Definition: Merge intervals combines overlapping ranges into non-overlapping intervals.

Key Steps:

1. Sort intervals by start time
2. Iterate and merge overlapping intervals
3. Track current merged interval

Overlap Condition:
Intervals [a, b] and [c, d] overlap if:

$$a \leq d \text{ and } c \leq b$$

When to Use:

- **Interval scheduling:** Meeting rooms, overlaps
- **Range queries:** Merge continuous ranges

- **Calendar problems:** Available time slots

7.2 Complete Problem Solutions

Problem: Merge Intervals (LeetCode 56)

```
/**
 * Merge overlapping intervals
 *
 * Approach:
 * 1. Sort by start time
 * 2. Iterate and merge overlapping intervals
 *
 * Time: O(n log n) - dominated by sorting
 * Space: O(n) - result array
 */
vector&lt;vector&lt;int&gt;&gt; merge(vector&lt;vector&lt;int&gt;&gt;&amp; intervals) {
    if (intervals.empty()) return {};

    // Sort by start time
    sort(intervals.begin(), intervals.end());

    vector&lt;vector&lt;int&gt;&gt; merged;
    merged.push_back(intervals[0]);

    for (int i = 1; i &lt; intervals.size(); i++) {
        vector&lt;int&gt;&amp; last = merged.back();
        vector&lt;int&gt;&amp; current = intervals[i];

        if (current[0] &lt;= last[1]) {
            // Overlapping → merge
            last[1] = max(last[1], current[1]);
        } else {
            // Non-overlapping → add new interval
            merged.push_back(current);
        }
    }

    return merged;
}
```

Problem: Insert Interval (LeetCode 57)

```
/**
 * Insert new interval and merge if necessary
 *
 * Three phases:
 * 1. Add all intervals ending before new interval
 * 2. Merge overlapping intervals
 * 3. Add all intervals starting after new interval
 *
 * Time: O(n)
 * Space: O(n)
 */
vector&lt;vector&lt;int&gt;&gt; insert(vector&lt;vector&lt;int&gt;&gt;&amp; intervals,
                          vector&lt;int&gt;&amp; newInterval) {
    vector&lt;vector&lt;int&gt;&gt; result;
    int i = 0;
    int n = intervals.size();

    // Phase 1: Add non-overlapping intervals before newInterval
    while (i &lt; n &amp;&amp; intervals[i][1] &lt; newInterval[0]) {
        result.push_back(intervals[i]);
        i++;
    }

    // Phase 2: Merge overlapping intervals
    while (i &lt; n &amp;&amp; intervals[i][0] &lt;= newInterval[1]) {
        newInterval[0] = min(newInterval[0], intervals[i][0]);
        newInterval[1] = max(newInterval[1], intervals[i][1]);
        i++;
    }
    result.push_back(newInterval);

    // Phase 3: Add remaining intervals
    while (i &lt; n) {
        result.push_back(intervals[i]);
        i++;
    }

    return result;
}
```

This comprehensive Volume 2 continues with the same exhaustive depth for all remaining patterns. The complete series provides production-grade implementations, multiple solution approaches, detailed complexity analysis, and systematic mastery tracking for competitive programming excellence.