Java Algorithm Templates

25 Essential Patterns for Coding Interviews

Helper Classes

```
// Standard data structures
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) {
        this.val = val;
        this.next = next;
    }
}

class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

// Java doesn't have tuples - use Pair
class Pair<K, V> {
    K key;
    V value;
    Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

1. Two Pointers

Use when: Sorted array, pair/triplet sums, palindromes
Time: O(n) **Space:** O(1)

```
// Opposite ends pattern
int[] twoSumSorted(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    while (left < right) {
        int sum = arr[left] + arr[right];
        if (sum == target)
            return new int[]{left, right};
        else if (sum < target)
            left++;
        else
            right--;
    }
    return new int[]{-1, -1};
}

// Same direction (fast/slow)
int removeDuplicates(int[] arr) {
    if (arr.length == 0) return 0;
    int slow = 0;
    for (int fast = 1; fast < arr.length; fast++) {
        if (arr[fast] != arr[slow]) {
            slow++;
            arr[slow] = arr[fast];
        }
    }
    return slow + 1;
}
```

2. Sliding Window

Use when: Substring/subarray with constraints
Time: O(n) **Space:** O(k)

```
// Fixed size window
int maxSumSubarray(int[] arr, int k) {
    int windowSum = 0;
    for (int i = 0; i < k; i++)
        windowSum += arr[i];

    int maxSum = windowSum;
    for (int i = k; i < arr.length; i++) {
        windowSum += arr[i] - arr[i - k];
        maxSum = Math.max(maxSum, windowSum);
    }
    return maxSum;
}

// Variable size window
int longestSubstringKDistinct(String s, int k) {
    Map<Character, Integer> charCount = new HashMap
    <>();
    int left = 0, maxLen = 0;

    for (int right = 0; right < s.length(); right++) {
        char c = s.charAt(right);
        charCount.put(c, charCount.getOrDefault(c, 0)
    + 1);

        while (charCount.size() > k) {
            char leftChar = s.charAt(left);
            charCount.put(leftChar, charCount.get(
    leftChar) - 1);
            if (charCount.get(leftChar) == 0)
                charCount.remove(leftChar);
            left++;
        }
        maxLen = Math.max(maxLen, right - left + 1);
    }
    return maxLen;
}
```

3. Binary Search

Use when: Sorted array, find first/last, optimization
Time: O(log n) **Space:** O(1)

```
// Standard binary search
int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}

// Find first occurrence
int findFirst(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    int result = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            result = mid;
            right = mid - 1;  // Continue left
        } else if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return result;
}

// Search space binary search
// Tip: Binary search on ANSWER range
int minCapacity(int[] weights, int days) {
    int left = Arrays.stream(weights).max().getAsInt()
    ;
    int right = Arrays.stream(weights).sum();

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (canShip(weights, days, mid))
            right = mid;
        else
            left = mid + 1;
    }
    return left;
}
```

4. Slow and Fast Pointers

Use when: Linked list cycles, find middle
Time: O(n) **Space:** O(1)

```
// Cycle detection
boolean hasCycle(ListNode head) {
    ListNode slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast)
            return true;
    }
    return false;
}

// Find cycle start
ListNode detectCycle(ListNode head) {
    ListNode slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) {
            slow = head;
            while (slow != fast) {
                slow = slow.next;
                fast = fast.next;
            }
            return slow;
        }
    }
    return null;
}
```

```java
// Find middle
ListNode findMiddle(ListNode head) {
    ListNode slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}
```

5. Linked List Reversal

Use when: Reverse list/partial, reorder
Time: O(n) **Space:** O(1)

```java
// Reverse entire list
ListNode reverseList(ListNode head) {
    ListNode prev = null, curr = head;
    while (curr != null) {
        ListNode next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

// Reverse between positions m and n
ListNode reverseBetween(ListNode head, int m, int n) {
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode prev = dummy;

    for (int i = 0; i < m - 1; i++)
        prev = prev.next;

    ListNode curr = prev.next;
    for (int i = 0; i < n - m; i++) {
        ListNode temp = curr.next;
        curr.next = temp.next;
        temp.next = prev.next;
        prev.next = temp;
    }
    return dummy.next;
}
```

6. Binary Tree Traversal

Use when: Tree navigation, level processing
Time: O(n) **Space:** O(h) recursive, O(n) iterative

```java
// Recursive inorder
List<Integer> inorder(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    inorderHelper(root, result);
    return result;
}
void inorderHelper(TreeNode node, List<Integer> result
    ) {
    if (node == null) return;
    inorderHelper(node.left, result);
    result.add(node.val);
    inorderHelper(node.right, result);
}

// Iterative inorder
List<Integer> inorderIterative(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();
    TreeNode curr = root;

    while (curr != null || !stack.isEmpty()) {
        while (curr != null) {
            stack.push(curr);
            curr = curr.left;
        }
        curr = stack.pop();
        result.add(curr.val);
        curr = curr.right;
    }
    return result;
}

// Level order (BFS)
List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) return result;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        int size = queue.size();
        List<Integer> level = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
            level.add(node.val);
            if (node.left != null) queue.offer(node.
    left);
            if (node.right != null) queue.offer(node.
    right);
        }
        result.add(level);
    }
    return result;
}
```

7. DFS (Depth-First Search)

Use when: Tree/graph traversal, connectivity
Time: O(V+E) **Space:** O(V)

```java
// Recursive DFS
void dfs(int node, Set<Integer> visited,
         Map<Integer, List<Integer>> graph) {
    if (visited.contains(node)) return;
    visited.add(node);
    for (int neighbor : graph.get(node)) {
        dfs(neighbor, visited, graph);
    }
}

// Iterative DFS
Set<Integer> dfsIterative(int start,
                          Map<Integer, List<Integer>>
    graph) {
    Stack<Integer> stack = new Stack<>();
    Set<Integer> visited = new HashSet<>();
    stack.push(start);

    while (!stack.isEmpty()) {
        int node = stack.pop();
        if (!visited.contains(node)) {
            visited.add(node);
            for (int neighbor : graph.get(node)) {
                if (!visited.contains(neighbor))
                    stack.push(neighbor);
            }
        }
    }
    return visited;
}

// Count connected components
int countComponents(int n, int[][] edges) {
    Map<Integer, List<Integer>> graph = new HashMap
    <>();
    for (int i = 0; i < n; i++)
        graph.put(i, new ArrayList<>());

    for (int[] edge : edges) {
        graph.get(edge[0]).add(edge[1]);
        graph.get(edge[1]).add(edge[0]);
    }

    Set<Integer> visited = new HashSet<>();
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (!visited.contains(i)) {
            dfs(i, visited, graph);
            count++;
        }
    }
    return count;
}
```

8. BFS (Breadth-First Search)

Use when: Shortest path, level-order
Time: O(V+E) **Space:** O(V)

```java
// Standard BFS
Set<Integer> bfs(int start,
                 Map<Integer, List<Integer>> graph) {
    Queue<Integer> queue = new LinkedList<>();
    Set<Integer> visited = new HashSet<>();
    queue.offer(start);
    visited.add(start);

    while (!queue.isEmpty()) {
        int node = queue.poll();
        for (int neighbor : graph.get(node)) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                queue.offer(neighbor);
            }
        }
    }
    return visited;
}

// Multi-source BFS (matrix)
void bfsMatrix(int[][] matrix) {
    Queue<int[]> queue = new LinkedList<>();

    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[0].length; j++) {
            if (matrix[i][j] == target)
                queue.offer(new int[]{i, j});
        }
    }

    int[][] dirs = {{0,1}, {1,0}, {0,-1}, {-1,0}};
    while (!queue.isEmpty()) {
        int[] pos = queue.poll();
```

```
        int x = pos[0], y = pos[1];
        for (int[] dir : dirs) {
            int nx = x + dir[0], ny = y + dir[1];
            if (nx >= 0 && nx < matrix.length &&
                ny >= 0 && ny < matrix[0].length) {
                // Process neighbor
            }
        }
    }
}

// Shortest path with distance
int shortestPath(int start, int end,
                 Map<Integer, List<Integer>> graph) {
    Queue<int[]> queue = new LinkedList<>();
    Set<Integer> visited = new HashSet<>();
    queue.offer(new int[]{start, 0});
    visited.add(start);

    while (!queue.isEmpty()) {
        int[] curr = queue.poll();
        int node = curr[0], dist = curr[1];
        if (node == end) return dist;

        for (int neighbor : graph.get(node)) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                queue.offer(new int[]{neighbor, dist +
     1});
            }
        }
    }
    return -1;
}
```

9. Dynamic Programming

Use when: Optimization, overlapping subproblems
Time: O(n) to $O(n^3)$ **Space:** O(n) to $O(n^2)$

```
// Top-down (Memoization)
// Public wrapper for clean API
int fib(int n) {
    return fibMemo(n, new HashMap<>());
}

// Private implementation with memo
private int fibMemo(int n, Map<Integer, Integer> memo)
     {
    if (memo.containsKey(n))
        return memo.get(n);
    if (n <= 1) return n;

    int result = fibMemo(n-1, memo) + fibMemo(n-2,
     memo);
    memo.put(n, result);
    return result;
}

// Bottom-up (Tabulation)
int fibTab(int n) {
    if (n <= 1) return 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];
}
```

```
// House robber pattern
int rob(int[] nums) {
    if (nums.length == 0) return 0;
    int prev2 = 0, prev1 = nums[0];

    for (int i = 1; i < nums.length; i++) {
        int temp = Math.max(prev1, prev2 + nums[i]);
        prev2 = prev1;
        prev1 = temp;
    }
    return prev1;
}

// Coin change (unbounded knapsack)
int coinChange(int[] coins, int amount) {
    int[] dp = new int[amount + 1];
    Arrays.fill(dp, amount + 1);
    dp[0] = 0;

    for (int i = 1; i <= amount; i++) {
        for (int coin : coins) {
            if (i >= coin)
                dp[i] = Math.min(dp[i], dp[i-coin] +
     1);
        }
    }
    return dp[amount] > amount ? -1 : dp[amount];
}

// Kadane's Algorithm (Maximum Subarray Sum)
// Classic DP - often asked by name
int maxSubarraySum(int[] nums) {
    if (nums.length == 0) return 0;
    int maxSoFar = nums[0], currentMax = nums[0];

    for (int i = 1; i < nums.length; i++) {
        currentMax = Math.max(nums[i], currentMax +
     nums[i]);
        maxSoFar = Math.max(maxSoFar, currentMax);
    }
    return maxSoFar;
}
```

10. Backtracking

Use when: Generate all solutions, CSP
Time: Exponential **Space:** O(n)

```
// Permutations
List<List<Integer>> permute(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    backtrackPermute(new ArrayList<>(), nums,
                     new boolean[nums.length], result)
     ;
    return result;
}
void backtrackPermute(List<Integer> path, int[] nums,
                      boolean[] used, List<List<
     Integer>> result) {
    if (path.size() == nums.length) {
        result.add(new ArrayList<>(path));
        return;
    }
    for (int i = 0; i < nums.length; i++) {
        if (used[i]) continue;
        path.add(nums[i]);
        used[i] = true;
        backtrackPermute(path, nums, used, result);
        path.remove(path.size() - 1);
        used[i] = false;
    }
}

// Subsets
List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    backtrackSubsets(0, new ArrayList<>(), nums,
     result);
    return result;
}
void backtrackSubsets(int start, List<Integer> path,
                      int[] nums, List<List<Integer>>
     result) {
    result.add(new ArrayList<>(path));
    for (int i = start; i < nums.length; i++) {
        path.add(nums[i]);
        backtrackSubsets(i + 1, path, nums, result);
        path.remove(path.size() - 1);
    }
}

// Combinations (choose k elements)
List<List<Integer>> combine(int[] nums, int k) {
    List<List<Integer>> result = new ArrayList<>();
    backtrackCombine(0, new ArrayList<>(), nums, k,
     result);
    return result;
}
void backtrackCombine(int start, List<Integer> path,
                      int[] nums, int k,
                      List<List<Integer>> result) {
    if (path.size() == k) {
        result.add(new ArrayList<>(path));
        return;
    }
    for (int i = start; i < nums.length; i++) {
        path.add(nums[i]);
        backtrackCombine(i + 1, path, nums, k, result)
     ;
        path.remove(path.size() - 1);
    }
}
```

11. Bit Manipulation

Use when: Set operations, unique elements
Time: O(1) per operation **Space:** O(1)

```
// Common operations
x & (x - 1)          // Clear rightmost set bit
x & -x               // Get rightmost set bit
x | (1 << i)         // Set bit i
x & ~(1 << i)        // Clear bit i
x ^ (1 << i)         // Toggle bit i
(x >> i) & 1         // Check if bit i is set
(x & (x-1)) == 0     // Check if power of 2

// Find single number (XOR)
int singleNumber(int[] nums) {
    int result = 0;
    for (int num : nums)
        result ^= num;
    return result;
}

// Count set bits
int countBits(int n) {
    int count = 0;
```

```
    while (n != 0) {
        n &= n - 1;
        count++;
    }
    return count;
}

// Subset generation using bits
List<List<Integer>> subsetsBits(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    int n = nums.length;

    for (int mask = 0; mask < (1 << n); mask++) {
        List<Integer> subset = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            if ((mask >> i & 1) == 1)
                subset.add(nums[i]);
        }
        result.add(subset);
    }
    return result;
}
```

12. Prefix Sum

Use when: Range queries, subarray sums
Time: O(n) build, O(1) query **Space:** O(n)

```
// 1D prefix sum
class PrefixSum {
    private int[] prefix;

    public PrefixSum(int[] nums) {
        prefix = new int[nums.length + 1];
        for (int i = 0; i < nums.length; i++)
            prefix[i+1] = prefix[i] + nums[i];
    }

    public int rangeSum(int left, int right) {
        return prefix[right+1] - prefix[left];
    }
}

// Subarray sum equals k
int subarraySum(int[] nums, int k) {
    int count = 0, prefixSum = 0;
    Map<Integer, Integer> sumFreq = new HashMap<>();
    sumFreq.put(0, 1);

    for (int num : nums) {
        prefixSum += num;
        count += sumFreq.getOrDefault(prefixSum - k,
    0);
        sumFreq.put(prefixSum,
                    sumFreq.getOrDefault(prefixSum, 0)
     + 1);
    }
    return count;
}

// 2D prefix sum
class Matrix2D {
    private int[][] prefix;

    public Matrix2D(int[][] matrix) {
        int m = matrix.length, n = matrix[0].length;
        prefix = new int[m+1][n+1];

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                prefix[i][j] = matrix[i-1][j-1] +
                               prefix[i-1][j] +
                               prefix[i][j-1] -
                               prefix[i-1][j-1];
            }
        }
    }

    public int regionSum(int r1, int c1, int r2, int
    c2) {
        return prefix[r2+1][c2+1] - prefix[r1][c2+1] -
               prefix[r2+1][c1] + prefix[r1][c1];
    }
}
```

13. Top K Elements / Heaps

Use when: Priority problems, k largest/smallest
Time: O(n log k) **Space:** O(k)

```
// K largest elements (use min-heap)
List<Integer> kLargest(int[] nums, int k) {
    PriorityQueue<Integer> heap = new PriorityQueue
    <>();
    for (int num : nums) {
        heap.offer(num);
        if (heap.size() > k)
            heap.poll();
    }
    return new ArrayList<>(heap);
}

// K smallest (use max-heap)
List<Integer> kSmallest(int[] nums, int k) {
    PriorityQueue<Integer> heap =
        new PriorityQueue<>((a, b) -> b - a);
    for (int num : nums) {
        heap.offer(num);
        if (heap.size() > k)
            heap.poll();
    }
    return new ArrayList<>(heap);
}

// Merge k sorted lists
ListNode mergeKLists(ListNode[] lists) {
    PriorityQueue<ListNode> heap = new PriorityQueue
    <>(
        (a, b) -> Integer.compare(a.val, b.val)
    );

    for (ListNode list : lists)
        if (list != null)
            heap.offer(list);

    ListNode dummy = new ListNode(0);
    ListNode curr = dummy;

    while (!heap.isEmpty()) {
        ListNode node = heap.poll();
        curr.next = node;
        curr = curr.next;
        if (node.next != null)
            heap.offer(node.next);
    }
    return dummy.next;
}
```

```
// Running median
class MedianFinder {
    PriorityQueue<Integer> small; // max-heap
    PriorityQueue<Integer> large; // min-heap

    public MedianFinder() {
        small = new PriorityQueue<>((a, b) -> b - a);
        large = new PriorityQueue<>();
    }

    public void addNum(int num) {
        small.offer(num);
        large.offer(small.poll());
        if (large.size() > small.size())
            small.offer(large.poll());
    }

    public double findMedian() {
        if (small.size() > large.size())
            return small.peek();
        return (small.peek() + large.peek()) / 2.0;
    }
}
```

14. Monotonic Stack

Use when: Next greater/smaller element
Time: O(n) **Space:** O(n)

```
// Next greater element
int[] nextGreater(int[] nums) {
    Stack<Integer> stack = new Stack<>();
    int[] result = new int[nums.length];
    Arrays.fill(result, -1);

    for (int i = 0; i < nums.length; i++) {
        while (!stack.isEmpty() &&
               nums[stack.peek()] < nums[i]) {
            result[stack.pop()] = nums[i];
        }
        stack.push(i);
    }
    return result;
}

// Next smaller element
int[] nextSmaller(int[] nums) {
    Stack<Integer> stack = new Stack<>();
    int[] result = new int[nums.length];
    Arrays.fill(result, -1);

    for (int i = 0; i < nums.length; i++) {
        while (!stack.isEmpty() &&
               nums[stack.peek()] > nums[i]) {
            result[stack.pop()] = nums[i];
        }
        stack.push(i);
    }
    return result;
}

// Largest rectangle in histogram
int largestRectangle(int[] heights) {
    Stack<Integer> stack = new Stack<>();
    int maxArea = 0;
    int[] h = Arrays.copyOf(heights, heights.length +
    1);

    for (int i = 0; i < h.length; i++) {
```

```
        while (!stack.isEmpty() && h[stack.peek()] > h
    [i]) {
            int height = h[stack.pop()];
            int width = stack.isEmpty() ? i :
                        i - stack.peek() - 1;
            maxArea = Math.max(maxArea, height * width
    );
        }
        stack.push(i);
    }
    return maxArea;
}
```

15. Overlapping Intervals

Use when: Scheduling, merging intervals
Time: O(n log n) **Space:** O(n)

```
// Merge intervals
int[][] merge(int[][] intervals) {
    Arrays.sort(intervals, (a, b) -> Integer.compare(a
    [0], b[0]));
    List<int[]> merged = new ArrayList<>();
    merged.add(intervals[0]);

    for (int i = 1; i < intervals.length; i++) {
        int[] curr = intervals[i];
        int[] last = merged.get(merged.size() - 1);
        if (curr[0] <= last[1]) {
            last[1] = Math.max(last[1], curr[1]);
        } else {
            merged.add(curr);
        }
    }
    return merged.toArray(new int[merged.size()][]);
}

// Insert interval
int[][] insert(int[][] intervals, int[] newInterval) {
    List<int[]> result = new ArrayList<>();
    int i = 0;

    // Before overlap
    while (i < intervals.length &&
           intervals[i][1] < newInterval[0])
        result.add(intervals[i++]);

    // Merge overlapping
    while (i < intervals.length &&
           intervals[i][0] <= newInterval[1]) {
        newInterval[0] = Math.min(newInterval[0],
                                  intervals[i][0]);
        newInterval[1] = Math.max(newInterval[1],
                                  intervals[i][1]);
        i++;
    }
    result.add(newInterval);

    // After overlap
    while (i < intervals.length)
        result.add(intervals[i++]);

    return result.toArray(new int[result.size()][]);
}

// Minimum intervals to remove
int eraseOverlapIntervals(int[][] intervals) {
    Arrays.sort(intervals, (a, b) -> Integer.compare(a
    [1], b[1]));
    int count = 0, end = Integer.MIN_VALUE;

    for (int[] interval : intervals) {
        if (interval[0] >= end) {
            end = interval[1];
        } else {
            count++;
        }
    }
    return count;
}
```

16. Trie (Prefix Tree)

Use when: Prefix matching, autocomplete
Time: O(m) per operation **Space:** O(total chars)

```
class TrieNode {
    Map<Character, TrieNode> children = new HashMap
    <>();
    boolean isEnd = false;
}

class Trie {
    private TrieNode root = new TrieNode();

    public void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            node.children.putIfAbsent(c, new TrieNode
    ());
            node = node.children.get(c);
        }
        node.isEnd = true;
    }

    public boolean search(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            if (!node.children.containsKey(c))
                return false;
            node = node.children.get(c);
        }
        return node.isEnd;
    }

    public boolean startsWith(String prefix) {
        TrieNode node = root;
        for (char c : prefix.toCharArray()) {
            if (!node.children.containsKey(c))
                return false;
            node = node.children.get(c);
        }
        return true;
    }
}
```

17. Union-Find (Disjoint Set)

Use when: Connectivity, components
Time: $O(\alpha(n)) \approx O(1)$ **Space:** O(n)

```
class UnionFind {
    private int[] parent, rank;
    private int components;

    public UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        components = n;
        for (int i = 0; i < n; i++)
            parent[i] = i;
    }

    public int find(int x) {
        if (parent[x] != x)
            parent[x] = find(parent[x]);
        return parent[x];
    }

    public boolean union(int x, int y) {
        int rootX = find(x), rootY = find(y);
        if (rootX == rootY) return false;

        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
        components--;
        return true;
    }

    public boolean connected(int x, int y) {
        return find(x) == find(y);
    }

    public int getComponents() {
        return components;
    }
}

// Count components
int countComponents(int n, int[][] edges) {
    UnionFind uf = new UnionFind(n);
    for (int[] edge : edges)
        uf.union(edge[0], edge[1]);
    return uf.getComponents();
}
```

18. Greedy Algorithms

Use when: Local optimum leads to global
Time: Varies **Space:** O(1) typically

```
// Activity selection
List<int[]> activitySelection(int[][] activities) {
    Arrays.sort(activities, (a, b) -> Integer.compare(
    a[1], b[1]));
    List<int[]> result = new ArrayList<>();
    result.add(activities[0]);
    int lastEnd = activities[0][1];

    for (int i = 1; i < activities.length; i++) {
        if (activities[i][0] >= lastEnd) {
            result.add(activities[i]);
            lastEnd = activities[i][1];
        }
    }
    return result;
}

// Jump game
```

```java
boolean canJump(int[] nums) {
    int maxReach = 0;
    for (int i = 0; i < nums.length; i++) {
        if (i > maxReach) return false;
        maxReach = Math.max(maxReach, i + nums[i]);
        if (maxReach >= nums.length - 1)
            return true;
    }
    return true;
}

// Gas station
int canCompleteCircuit(int[] gas, int[] cost) {
    int totalGas = 0, totalCost = 0;
    for (int i = 0; i < gas.length; i++) {
        totalGas += gas[i];
        totalCost += cost[i];
    }
    if (totalGas < totalCost) return -1;

    int tank = 0, start = 0;
    for (int i = 0; i < gas.length; i++) {
        tank += gas[i] - cost[i];
        if (tank < 0) {
            tank = 0;
            start = i + 1;
        }
    }
    return start;
}
```

19. Advanced DP

2D DP, State Machines, DP on Trees

```java
// 2D DP: Edit distance
int minDistance(String word1, String word2) {
    int m = word1.length(), n = word2.length();
    int[][] dp = new int[m+1][n+1];

    for (int i = 0; i <= m; i++) dp[i][0] = i;
    for (int j = 0; j <= n; j++) dp[0][j] = j;

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (word1.charAt(i-1) == word2.charAt(j-1)
    ) {
                dp[i][j] = dp[i-1][j-1];
            } else {
                dp[i][j] = 1 + Math.min(dp[i-1][j],
                    Math.min(dp[i][j-1], dp[i-1][j-1])
    );
            }
        }
    }
    return dp[m][n];
}

// State machine DP: Stock with cooldown
int maxProfitCooldown(int[] prices) {
    int held = -prices[0];
    int sold = 0, rest = 0;

    for (int i = 1; i < prices.length; i++) {
        int newHeld = Math.max(held, rest - prices[i])
    ;
        int newSold = held + prices[i];
        int newRest = Math.max(rest, sold);
        held = newHeld;
        sold = newSold;
        rest = newRest;
    }
    return Math.max(sold, rest);
}

// DP on trees: House robber III
int robTree(TreeNode root) {
    int[] result = dfsRob(root);
    return Math.max(result[0], result[1]);
}
int[] dfsRob(TreeNode node) {
    if (node == null) return new int[]{0, 0};

    int[] left = dfsRob(node.left);
    int[] right = dfsRob(node.right);

    int rob = node.val + left[1] + right[1];
    int notRob = Math.max(left[0], left[1]) +
                 Math.max(right[0], right[1]);
    return new int[]{rob, notRob};
}
```

20. Graph Algorithms

Dijkstra, Floyd-Warshall, MST

```java
// Dijkstra's shortest path
Map<Integer, Integer> dijkstra(
        Map<Integer, List<int[]>> graph, int start) {
    Map<Integer, Integer> distances = new HashMap<>();
    for (int node : graph.keySet())
        distances.put(node, Integer.MAX_VALUE);
    distances.put(start, 0);

    PriorityQueue<int[]> pq = new PriorityQueue<>(
        (a, b) -> Integer.compare(a[0], b[0])
    );
    pq.offer(new int[]{0, start});

    while (!pq.isEmpty()) {
        int[] curr = pq.poll();
        int dist = curr[0], node = curr[1];
        if (dist > distances.get(node)) continue;

        for (int[] neighbor : graph.get(node)) {
            int next = neighbor[0], weight = neighbor
    [1];
            int newDist = dist + weight;
            if (newDist < distances.get(next)) {
                distances.put(next, newDist);
                pq.offer(new int[]{newDist, next});
            }
        }
    }
    return distances;
}

// Floyd-Warshall (all-pairs shortest)
int[][] floydWarshall(int n, int[][] edges) {
    int[][] dist = new int[n][n];
    for (int[] row : dist)
        Arrays.fill(row, Integer.MAX_VALUE / 2);
    for (int i = 0; i < n; i++)
        dist[i][i] = 0;

    for (int[] edge : edges)
        dist[edge[0]][edge[1]] = edge[2];
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                dist[i][j] = Math.min(dist[i][j],
                                      dist[i][k] +
    dist[k][j]);
    return dist;
}

// Bellman-Ford (handles negative weights)
int[] bellmanFord(int n, int[][] edges, int start) {
    int[] dist = new int[n];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[start] = 0;

    for (int i = 0; i < n - 1; i++) {
        for (int[] edge : edges) {
            int u = edge[0], v = edge[1], w = edge[2];
            if (dist[u] != Integer.MAX_VALUE &&
                dist[u] + w < dist[v])
                dist[v] = dist[u] + w;
        }
    }

    // Check for negative cycles
    for (int[] edge : edges) {
        int u = edge[0], v = edge[1], w = edge[2];
        if (dist[u] != Integer.MAX_VALUE &&
            dist[u] + w < dist[v])
            return null; // Negative cycle
    }
    return dist;
}

// Prim's MST
List<int[]> primMST(int n, int[][] edges) {
    Map<Integer, List<int[]>> graph = new HashMap<>();
    for (int i = 0; i < n; i++)
        graph.put(i, new ArrayList<>());

    for (int[] edge : edges) {
        int u = edge[0], v = edge[1], w = edge[2];
        graph.get(u).add(new int[]{v, w});
        graph.get(v).add(new int[]{u, w});
    }

    List<int[]> mst = new ArrayList<>();
    Set<Integer> visited = new HashSet<>();
    visited.add(0);

    PriorityQueue<int[]> pq = new PriorityQueue<>(
        (a, b) -> Integer.compare(a[0], b[0])
    );
    for (int[] neighbor : graph.get(0))
        pq.offer(new int[]{neighbor[1], 0, neighbor
    [0]});

    while (!pq.isEmpty() && visited.size() < n) {
        int[] edge = pq.poll();
        int weight = edge[0], u = edge[1], v = edge
    [2];

        if (visited.contains(v)) continue;
        visited.add(v);
        mst.add(new int[]{u, v, weight});

        for (int[] neighbor : graph.get(v)) {
            int next = neighbor[0], w = neighbor[1];
            if (!visited.contains(next))
                pq.offer(new int[]{w, v, next});
        }
```

```
    }
    return mst;
}
```

21. Topological Sort

Use when: DAG ordering, dependencies
Time: O(V+E) **Space:** O(V)

```
// Kahn's algorithm (BFS-based)
List<Integer> topologicalSort(int n, int[][] edges) {
    Map<Integer, List<Integer>> graph = new HashMap
    <>();
    int[] inDegree = new int[n];

    for (int i = 0; i < n; i++)
        graph.put(i, new ArrayList<>());

    for (int[] edge : edges) {
        graph.get(edge[0]).add(edge[1]);
        inDegree[edge[1]]++;
    }

    Queue<Integer> queue = new LinkedList<>();
    for (int i = 0; i < n; i++)
        if (inDegree[i] == 0)
            queue.offer(i);

    List<Integer> result = new ArrayList<>();
    while (!queue.isEmpty()) {
        int node = queue.poll();
        result.add(node);
        for (int neighbor : graph.get(node)) {
            inDegree[neighbor]--;
            if (inDegree[neighbor] == 0)
                queue.offer(neighbor);
        }
    }
    return result.size() == n ? result :
           new ArrayList<>();
}

// Course schedule with prerequisites
boolean canFinish(int numCourses, int[][]
    prerequisites) {
    Map<Integer, List<Integer>> graph = new HashMap
    <>();
    int[] inDegree = new int[numCourses];

    for (int i = 0; i < numCourses; i++)
        graph.put(i, new ArrayList<>());

    for (int[] prereq : prerequisites) {
        graph.get(prereq[1]).add(prereq[0]);
        inDegree[prereq[0]]++;
    }

    Queue<Integer> queue = new LinkedList<>();
    for (int i = 0; i < numCourses; i++)
        if (inDegree[i] == 0)
            queue.offer(i);

    int count = 0;
    while (!queue.isEmpty()) {
        int course = queue.poll();
        count++;
        for (int next : graph.get(course)) {
            inDegree[next]--;
            if (inDegree[next] == 0)
                queue.offer(next);
        }
    }
    return count == numCourses;
}
```

22. Cyclic Sort

Use when: Array with numbers in range [1, n]
Time: O(n) **Space:** O(1)

```
// Core cyclic sort pattern
void cyclicSort(int[] nums) {
    int i = 0;
    while (i < nums.length) {
        int j = nums[i] - 1;
        if (j >= 0 && j < nums.length &&
            nums[i] != nums[j]) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        } else {
            i++;
        }
    }
}

// Find missing number
int findMissingNumber(int[] nums) {
    int i = 0;
    while (i < nums.length) {
        int j = nums[i];
        if (j < nums.length && nums[i] != nums[j]) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        } else {
            i++;
        }
    }

    for (i = 0; i < nums.length; i++)
        if (nums[i] != i)
            return i;
    return nums.length;
}

// Find all duplicates
List<Integer> findDuplicates(int[] nums) {
    int i = 0;
    while (i < nums.length) {
        int j = nums[i] - 1;
        if (nums[i] != nums[j]) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        } else {
            i++;
        }
    }

    List<Integer> duplicates = new ArrayList<>();
    for (i = 0; i < nums.length; i++)
        if (nums[i] != i + 1)
            duplicates.add(nums[i]);
    return duplicates;
}
```

23. Lowest Common Ancestor

Use when: Finding shared ancestor in tree
Time: O(n) **Space:** O(h)

```
// LCA in binary tree
TreeNode lowestCommonAncestor(TreeNode root,
                              TreeNode p, TreeNode q)
    {
    if (root == null || root == p || root == q)
        return root;

    TreeNode left = lowestCommonAncestor(root.left, p,
     q);
    TreeNode right = lowestCommonAncestor(root.right,
    p, q);

    if (left != null && right != null)
        return root;
    return left != null ? left : right;
}

// LCA in BST
TreeNode lcaBST(TreeNode root, TreeNode p, TreeNode q)
     {
    while (root != null) {
        if (p.val < root.val && q.val < root.val)
            root = root.left;
        else if (p.val > root.val && q.val > root.val)
            root = root.right;
        else
            return root;
    }
    return null;
}
```

24. Matrix Traversal

Use when: Spiral traversal, in-place rotation
Time: O(m*n) **Space:** O(1)

```
// Spiral order traversal
List<Integer> spiralOrder(int[][] matrix) {
    List<Integer> result = new ArrayList<>();
    if (matrix.length == 0) return result;

    int left = 0, right = matrix[0].length - 1;
    int top = 0, bottom = matrix.length - 1;

    while (left <= right && top <= bottom) {
        for (int i = left; i <= right; i++)
            result.add(matrix[top][i]);
        top++;

        for (int i = top; i <= bottom; i++)
            result.add(matrix[i][right]);
        right--;

        if (top <= bottom) {
            for (int i = right; i >= left; i--)
                result.add(matrix[bottom][i]);
            bottom--;
        }

        if (left <= right) {
            for (int i = bottom; i >= top; i--)
                result.add(matrix[i][left]);
            left++;
        }
```

```
    }
    return result;
}

// Rotate matrix 90 degrees clockwise
void rotate(int[][] matrix) {
    int n = matrix.length;

    // Transpose
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = temp;
        }
    }

    // Reverse each row
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n / 2; j++) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[i][n - 1 - j];
            matrix[i][n - 1 - j] = temp;
        }
    }
}
```

25. TreeMap/TreeSet Patterns

Java-specific ordered collections
Time: O(log n) **Space:** O(n)

```
// TreeMap for ordered operations
TreeMap<Integer, Integer> map = new TreeMap<>();

// Get floor/ceiling keys
Integer floor = map.floorKey(x);     // Largest <= x
Integer ceiling = map.ceilingKey(x); // Smallest >= x
Integer lower = map.lowerKey(x);     // Largest < x
Integer higher = map.higherKey(x);   // Smallest > x

// Get entries
Map.Entry<Integer, Integer> first = map.firstEntry();
Map.Entry<Integer, Integer> last = map.lastEntry();

// Range operations
NavigableMap<Integer, Integer> sub =
    map.subMap(from, true, to, false);

// TreeSet for ordered set
TreeSet<Integer> set = new TreeSet<>();
Integer floor = set.floor(x);
Integer ceiling = set.ceiling(x);
```

Java-Specific Gotchas

```
// 1. Integer caching (-128 to 127)
Integer a = 127, b = 127;
a == b;  // true (cached)
Integer c = 128, d = 128;
c == d;  // false (not cached)
c.equals(d);  // true (always use equals!)

// 2. String comparison
String s1 = "hello", s2 = "hello";
s1 == s2;  // true (string pool)
String s3 = new String("hello");
s1 == s3;  // false
s1.equals(s3);  // true (correct way)

// 3. Array initialization
int[] arr = new int[5];  // [0, 0, 0, 0, 0]
Arrays.fill(arr, -1);    // [-1, -1, -1, -1, -1]

// 4. StringBuilder for string manipulation
StringBuilder sb = new StringBuilder();
sb.append("hello").append(" world");
String result = sb.toString();

// 5. Comparator for custom sorting
// WRONG - can overflow with large values:
// Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
// RIGHT - safe from overflow:
Arrays.sort(intervals, (a, b) -> Integer.compare(a[0],
      b[0]))

// 6. Character methods
Character.isDigit('5');     // true
Character.isLetter('a');    // true
Character.toLowerCase('A'); // 'a'
```

Pattern Recognition Guide

- **Sorted array + pairs:** Two Pointers
- **Substring with constraints:** Sliding Window
- **Search in sorted:** Binary Search
- **Linked list cycles:** Fast/Slow Pointers
- **Tree traversal:** DFS/BFS/Preorder/Inorder
- **Tree ancestor problems:** LCA
- **Optimization + overlapping:** Dynamic Programming
- **Max subarray sum:** Kadane's Algorithm
- **All solutions:** Backtracking
- **Choose k elements:** Combinations
- **Range queries:** Prefix Sum
- **Priority-based:** Heap
- **Next greater/smaller:** Monotonic Stack
- **Intervals:** Sort + Merge/Greedy
- **Prefix matching:** Trie
- **Connectivity:** Union-Find
- **DAG ordering:** Topological Sort
- **Array with numbers 1 to n:** Cyclic Sort
- **Spiral/rotation matrix:** Matrix Traversal
- **Ordered operations:** TreeMap/TreeSet
- **Negative edge weights:** Bellman-Ford

Complexity Quick Reference

Pattern	Time	Space
Two Pointers	O(n)	O(1)
Sliding Window	O(n)	O(k)
Binary Search	O(log n)	O(1)
Fast/Slow Pointers	O(n)	O(1)
DFS/BFS	O(V+E)	O(V)
DP (1D)	O(n)	O(n)
DP (2D)	$O(n^2)$	$O(n^2)$
Kadane's	O(n)	O(1)
Backtracking	Exponential	O(n)
Prefix Sum	O(n) build, O(1) query	O(n)
Heap	O(n log k)	O(k)
Monotonic Stack	O(n)	O(n)
Intervals	O(n log n)	O(n)
Trie	O(m)	O(total)
Union-Find	$O(\alpha(n))$	O(n)
Topological Sort	O(V+E)	O(V)
Cyclic Sort	O(n)	O(1)
LCA	O(n)	O(h)
Matrix Traversal	O(m*n)	O(1)
TreeMap/TreeSet	O(log n)	O(n)
Bellman-Ford	O(V*E)	O(V)

Common Java Collections

Python	Java
list	ArrayList, LinkedList
dict	HashMap, TreeMap
set	HashSet, TreeSet
deque	ArrayDeque, LinkedList
heapq	PriorityQueue