```
// Ultimate Java DSA + Logic Patterns for SDET/Senior SDET Interviews
/

// Pattern 1: Two Sum (Brute Force + Optimized)
class TwoSum {
    public static int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement)) {
                return new int[] { map.get(complement), i };
            }
            map.put(nums[i], i);
        }
        return new int[]{};
    }
}



// Pattern 2: Merge Sort
class MergeSort {
    public static void mergeSort(int[] arr, int l, int r) {
        if (l < r) {
            int m = l + (r - l) / 2;
            mergeSort(arr, l, m);
            mergeSort(arr, m + 1, r);
            merge(arr, l, m, r);
        }
    }

    public static void merge(int[] arr, int l, int m, int r) {
        int[] left = Arrays.copyOfRange(arr, l, m + 1);
        int[] right = Arrays.copyOfRange(arr, m + 1, r + 1);

        int i = 0, j = 0, k = l;
        while (i < left.length && j < right.length) {
            arr[k++] = left[i] <= right[j] ? left[i++] : right[j++];
        }
        while (i < left.length) arr[k++] = left[i++];
        while (j < right.length) arr[k++] = right[j++];
    }
}
```

```
// Pattern 3: Reverse a LinkedList
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) { this.val = val; }
}

class ReverseLinkedList {
    public static ListNode reverseList(ListNode head) {
        ListNode prev = null;
        while (head != null) {
            ListNode next = head.next;
            head.next = prev;
            prev = head;
            head = next;
        }
        return prev;
    }
}
```

```
// Pattern 4: Detect Cycle in a Linked List
class LinkedListCycle {
    public static boolean hasCycle(ListNode head) {
        if (head == null) return false;
        ListNode slow = head, fast = head.next;
        while (slow != fast) {
            if (fast == null || fast.next == null) return false;
            slow = slow.next;
            fast = fast.next.next;
        }
        return true;
    }
}
```

```java
// Pattern 5: HashMap Frequency Count
class FrequencyCounter {
    public static Map<Integer, Integer> frequency(int[] nums) {
        Map<Integer, Integer> freq = new HashMap<>();
        for (int num : nums) {
            freq.put(num, freq.getOrDefault(num, 0) + 1);
        }
        return freq;
    }
}
```

```java
// Pattern 6: Sliding Window - Maximum Sum Subarray of Size K
class MaxSumSubarrayK {
    public static int maxSum(int[] nums, int k) {
        int windowSum = 0, maxSum = 0;
        for (int i = 0; i < k; i++) windowSum += nums[i];
        maxSum = windowSum;
        for (int i = k; i < nums.length; i++) {
            windowSum += nums[i] - nums[i - k];
            maxSum = Math.max(maxSum, windowSum);
        }
        return maxSum;
    }
}
```

```java
// Pattern 7: Two Pointers - Sorted Array Two Sum
class TwoPointersSorted {
    public static boolean hasTwoSum(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int sum = nums[left] + nums[right];
            if (sum == target) return true;
            else if (sum < target) left++;
            else right--;
        }
        return false;
    }
}
```

Pattern 8 : Three sum

```
class ThreeSum {
    public static List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        Arrays.sort(nums);
        for (int i = 0; i < nums.length - 2; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) continue; // Skip duplicates
            int left = i + 1, right = nums.length - 1;
            while (left < right) {
                int sum = nums[i] + nums[left] + nums[right];
                if (sum == 0) {
                    result.add(Arrays.asList(nums[i], nums[left], nums[right]));
                    while (left < right && nums[left] == nums[left + 1]) left++; // Skip duplicates
                    while (left < right && nums[right] == nums[right - 1]) right--; // Skip duplicates
                    left++;
                    right--;
                } else if (sum < 0) {
                    left++;
                } else {
                    right--;
                }
            }
        }
        return result;
    }
}
```

```
Pattern 9 : class LongestPalindromicSubstring {
    public static String longestPalindrome(String s) {
        if (s == null || s.length() < 1) return "";
        int start = 0, end = 0;
        for (int i = 0; i < s.length(); i++) {
            int len1 = expandAroundCenter(s, i, i); // Odd length palindrome
            int len2 = expandAroundCenter(s, i, i + 1); // Even length palindrome
            int len = Math.max(len1, len2);
            if (len > (end - start)) {
                start = i - (len - 1) / 2;
                end = i + len / 2;
            }
        }
        return s.substring(start, end + 1);
    }

    private static int expandAroundCenter(String s, int left, int right) {
        while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
            left--;
            right++;
        }
        return right - left - 1;
    }
}
```

```
// Pattern 10: HashSet to Find Duplicates
class ContainsDuplicate {
    public static boolean containsDuplicate(int[] nums) {
        Set<Integer> set = new HashSet<>();
        for (int num : nums) {
            if (!set.add(num)) return true;
        }
        return false;
    }
}

// Pattern 11: Subarrays with Equal Sum
class SubarraySumEqualsK {
    public static int subarraySum(int[] nums, int k) {
        Map<Integer, Integer> map = new HashMap<>();
        map.put(0, 1);
        int sum = 0, count = 0;
        for (int num : nums) {
            sum += num;
            count += map.getOrDefault(sum - k, 0);
            map.put(sum, map.getOrDefault(sum, 0) + 1);
        }
        return count;
    }
}

// Pattern 12: Sliding Window - Longest Substring Without Repeating Characters
class LongestUniqueSubstring {
    public static int lengthOfLongestSubstring(String s) {
        Set<Character> set = new HashSet<>();
        int left = 0, maxLen = 0;
        for (int right = 0; right < s.length(); right++) {
            while (!set.add(s.charAt(right))) {
                set.remove(s.charAt(left++));
            }
            maxLen = Math.max(maxLen, right - left + 1);
        }
        return maxLen;
    }
}

// Pattern 13: Trapping Rain Water
class TrappingRainWater {
    public static int trap(int[] height) {
        int left = 0, right = height.length - 1;
        int leftMax = 0, rightMax = 0, water = 0;
        while (left < right) {
            if (height[left] < height[right]) {
                if (height[left] >= leftMax) leftMax = height[left];
                else water += leftMax - height[left];
                left++;
            } else {
                if (height[right] >= rightMax) rightMax = height[right];
                else water += rightMax - height[right];
                right--;
            }
        }
        return water;
    }
}

// Pattern 14: Merge Intervals
class MergeIntervals {
    public static int[][] merge(int[][] intervals) {
        Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
        List<int[]> merged = new ArrayList<>();
        int[] current = intervals[0];
        for (int[] interval : intervals) {
            if (interval[0] <= current[1]) {
                current[1] = Math.max(current[1], interval[1]);
            } else {
                merged.add(current);
                current = interval;
            }
        }
        merged.add(current);
        return merged.toArray(new int[merged.size()][]);
    }
}

// Pattern 15: Kth Largest Element (Min Heap)
class KthLargest {
    public static int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        for (int num : nums) {
            pq.add(num);
            if (pq.size() > k) pq.poll();
        }
        return pq.peek();
    }
}




// Pattern 16: Word Ladder (BFS)
class WordLadder {
    public static int ladderLength(String beginWord, String endWord, List<String> wordList) {
        Set<String> dict = new HashSet<>(wordList);
        Queue<String> queue = new LinkedList<>();
        queue.offer(beginWord);
        int level = 1;
        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                String word = queue.poll();
                if (word.equals(endWord)) return level;
                char[] chars = word.toCharArray();
                for (int j = 0; j < chars.length; j++) {
                    char old = chars[j];
                    for (char c = 'a'; c <= 'z'; c++) {
                        chars[j] = c;
                        String next = new String(chars);
                        if (dict.contains(next)) {
                            queue.offer(next);
                            dict.remove(next);
                        }
                    }
                    chars[j] = old;
                }
            }
            level++;
        }
        return 0;
    }
}
```

```
// Pattern 17: Dijkstra’s Shortest Path
class DijkstraGraph {
    static class Pair {
        int node, dist;
        Pair(int node, int dist) { this.node = node; this.dist = dist; }
    }

    public static int[] dijkstra(int V, List<List<Pair>> graph, int src) {
        PriorityQueue<Pair> pq = new PriorityQueue<>((a, b) -> a.dist - b.dist);
        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[src] = 0;
        pq.offer(new Pair(src, 0));

        while (!pq.isEmpty()) {
            Pair cur = pq.poll();
            for (Pair neighbor : graph.get(cur.node)) {
                if (dist[cur.node] + neighbor.dist < dist[neighbor.node]) {
                    dist[neighbor.node] = dist[cur.node] + neighbor.dist;
                    pq.offer(new Pair(neighbor.node, dist[neighbor.node]));
                }
            }
        }
        return dist;
    }
}
```

```
// Pattern 18: Top K Frequent Elements
class TopKFrequent {
    public static int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> count = new HashMap<>();
        for (int num : nums) count.put(num, count.getOrDefault(num, 0) + 1);

        PriorityQueue<Map.Entry<Integer, Integer>> pq =
            new PriorityQueue<>((a, b) -> a.getValue() - b.getValue());

        for (Map.Entry<Integer, Integer> entry : count.entrySet()) {
            pq.offer(entry);
            if (pq.size() > k) pq.poll();
        }

        int[] result = new int[k];
        for (int i = 0; i < k; i++) {
            result[i] = pq.poll().getKey();
        }
        return result;
    }
}
```

```
// Pattern 19: Group Anagrams
class GroupAnagrams {
    public static List<List<String>> groupAnagrams(String[] strs) {
        Map<String, List<String>> map = new HashMap<>();
        for (String s : strs) {
            char[] chars = s.toCharArray();
            Arrays.sort(chars);
            String key = new String(chars);
            map.computeIfAbsent(key, k -> new ArrayList<>()).add(s);
        }
        return new ArrayList<>(map.values());
    }
}
```

```
// Pattern 20: Best Time to Buy and Sell Stock I (1 transaction)
class BuySellStockI {
    public static int maxProfit(int[] prices) {
        int minPrice = Integer.MAX_VALUE, maxProfit = 0;
        for (int price : prices) {
            if (price < minPrice) minPrice = price;
            else maxProfit = Math.max(maxProfit, price - minPrice);
        }
        return maxProfit;
    }
}
```

```
// Pattern 21: Best Time to Buy and Sell Stock II (multiple transactions)
class BuySellStockII {
    public static int maxProfit(int[] prices) {
        int profit = 0;
        for (int i = 1; i < prices.length; i++) {
            if (prices[i] > prices[i - 1]) profit += prices[i] - prices[i - 1];
        }
        return profit;
    }
}
```

```
// Pattern 22: Best Time to Buy and Sell Stock with Cooldown
class BuySellStockCooldown {
    public static int maxProfit(int[] prices) {
        if (prices.length == 0) return 0;
        int sell = 0, hold = -prices[0], cooldown = 0;
        for (int i = 1; i < prices.length; i++) {
            int prevSell = sell;
            sell = Math.max(sell, hold + prices[i]);
            hold = Math.max(hold, cooldown - prices[i]);
            cooldown = prevSell;
        }
        return sell;
    }
}
```

```
// Pattern 23: Fibonacci Number (DP)
class FibonacciDP {
    public static int fib(int n) {
        if (n <= 1) return n;
        int[] dp = new int[n + 1];
        dp[0] = 0; dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
}

// Pattern 24: Climbing Stairs (DP)
class ClimbingStairs {
    public static int climbStairs(int n) {
        if (n <= 2) return n;
        int a = 1, b = 2, c = 0;
        for (int i = 3; i <= n; i++) {
            c = a + b;
            a = b;
            b = c;
        }
        return b;
    }
}

// Pattern 25: Longest Increasing Subsequence (DP)
class LongestIncreasingSubsequence {
    public static int lengthOfLIS(int[] nums) {
        if (nums.length == 0) return 0;
        int[] dp = new int[nums.length];
        Arrays.fill(dp, 1);
        for (int i = 1; i < nums.length; i++) {
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
        }
        return Arrays.stream(dp).max().getAsInt();
    }
}
```

```
// Pattern 26: Longest Common Subsequence (DP)
class LongestCommonSubsequence {
    public static int longestCommonSubsequence(String text1, String text2) {
        int m = text1.length(), n = text2.length();
        int[][] dp = new int[m + 1][n + 1];
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
        return dp[m][n];
    }
}
```