```
/**
 * Problem 1: Encode and Decode Strings
 * Problem Statement:
 * Implement encode and decode functions that convert a list of strings to a single string and vice versa.
 * Intuition:
 * Use a delimiter or length prefix to join strings uniquely so decoding is unambiguous.
 * Logic:
 * 1. Encode each string with its length followed by a separator.
 * 2. Decode by reading lengths and extracting substrings.
 */
function encode(strs) {
    return strs.map(s => s.length + '#' + s).join('');
}
function decode(s) {
    const res = [];
    let i = 0;
    while (i < s.length) {
        let j = i;
        while (s[j] !== '#') j++;
        const len = parseInt(s.slice(i, j));
        res.push(s.slice(j + 1, j + 1 + len));
        i = j + 1 + len;
    }
    return res;
}
/**
 * Problem 2: Container With Most Water
 * Problem Statement:
 * Given an array of heights, find two lines that together with x-axis forms container with most water.
 * Intuition:
 * Use two pointers, moving the shorter line inward increases potential area.
 * Logic:
 * 1. Initialize left and right pointers.
 * 2. Calculate area, update max.
 * 3. Move pointer with shorter height.
 */
function maxArea(height) {
    let left = 0, right = height.length - 1, maxA = 0;
    while (left < right) {
        const area = Math.min(height[left], height[right]) * (right - left);
        maxA = Math.max(maxA, area);
        if (height[left] < height[right]) left++;
        else right--;
    }
    return maxA;
}
/**
 * Problem 3: Permutation in a String
 * Problem Statement:
 * Check if s1's permutation is substring of s2.
 * Intuition:
 * Use sliding window with character count to match s1 in s2.
 * Logic:
 * 1. Count chars in s1.
 * 2. Slide window of length s1 in s2, update counts.
 * 3. Check if counts match.
 */
function checkInclusion(s1, s2) {
    const a = Array(26).fill(0), b = Array(26).fill(0);
    const n = s1.length, m = s2.length;
    for (let i = 0; i < n; i++) {
        a[s1.charCodeAt(i) - 97]++;
        b[s2.charCodeAt(i) - 97]++;
    }
    if (a.toString() === b.toString()) return true;
    for (let i = n; i < m; i++) {
        b[s2.charCodeAt(i) - 97]++;
        b[s2.charCodeAt(i - n) - 97]--;
        if (a.toString() === b.toString()) return true;
    }
    return false;
}
/**
 * Problem 4: Evaluate Reverse Polish Notation
 * Problem Statement:
 * Evaluate arithmetic expression in Reverse Polish Notation.
 * Intuition:
 * Use a stack to store operands and evaluate when operator is found.
 * Logic:
 * 1. Traverse tokens.
 * 2. Push numbers to stack.
 * 3. Pop two numbers and apply operator, push result.
 */
function evalRPN(tokens) {
    const stack = [];
    for (const t of tokens) {
        if ("+-*/".includes(t)) {
            const b = stack.pop(), a = stack.pop();
            stack.push(
                t === '+' ? a + b :
                t === '-' ? a - b :
                t === '*' ? a * b :
                Math.trunc(a / b)
            );
        } else stack.push(parseInt(t));
    }
    return stack[0];
}
/**
 * Problem 5: Daily Temperatures
 * Problem Statement:
 * For each day, find how many days until a warmer temperature.
 * Intuition:
 * Use a stack to store indices; next warmer temperature resolves previous days.
 * Logic:
 * 1. Traverse temps.
 * 2. Pop stack while current temp > stack top, compute days difference.
 * 3. Push current index.
 */
function dailyTemperatures(T) {
    const res = Array(T.length).fill(0), stack = [];
    for (let i = 0; i < T.length; i++) {
        while (stack.length && T[i] > T[stack[stack.length - 1]]) {
            const idx = stack.pop();
            res[idx] = i - idx;
        }
        stack.push(i);
    }
    return res;
}
/**
 * Problem 6: Car Fleet
 * Problem Statement:
 * Count number of car fleets that will arrive at target.
 * Intuition:
 * Cars behind can't pass cars ahead if slower; sort by position, calculate time to target.
 * Logic:
 * 1. Sort cars by position descending.
 * 2. Compute time to target.
 * 3. Merge fleets if time behind <= time ahead.
 */
function carFleet(target, position, speed) {
    const cars = position.map((p, i) => [p, speed[i]]).sort((a, b) => b[0] - a[0]);
    let fleets = 0, curTime = 0;
    for (const [pos, spd] of cars) {
        const time = (target - pos) / spd;
        if (time > curTime) { fleets++; curTime = time; }
    }
    return fleets;
}
/**
 * Problem 7: Search a 2D Matrix
 * Problem Statement:
 * Search for a target in a matrix where rows and columns are sorted.
 * Intuition:
 * Start from top-right; move left if greater, down if smaller.
```

```
/**
 * Problem 28: Min Cost to Connect All Points
 * Problem Statement:
 * Given points in 2D plane, connect all points with minimal total distance.
 * Intuition:
 * Use Minimum Spanning Tree (MST) to connect all points efficiently.
 * Logic:
 * 1. Use Prim's algorithm with min-heap for next cheapest edge.
 * 2. Track visited points to avoid cycles.
 */
function minCostConnectPoints(points) {
    const n = points.length;
    const visited = Array(n).fill(false);
    const heap = [[0, 0]]; // [cost, node]
    let res = 0, count = 0;
    function dist(i, j) {
        return Math.abs(points[i][0] - points[j][0]) + Math.abs(points[i][1] - points[j][1]);
    }
    while (count < n) {
        heap.sort((a, b) => a[0] - b[0]);
        const [cost, u] = heap.shift();
        if (visited[u]) continue;
        visited[u] = true;
        res += cost;
        count++;
        for (let v = 0; v < n; v++) {
            if (!visited[v]) heap.push([dist(u, v), v]);
        }
    }
    return res;
}
/**
 * Problem 29: Swim in Rising Water
 * Problem Statement:
 * Find minimum time to reach bottom-right in grid where water rises.
 * Intuition:
 * Use Dijkstra-like approach with priority queue to always move to lowest max water level.
 * Logic:
 * 1. Push starting cell with its height into heap.
 * 2. Expand neighbors, updating max height along path.
 */
function swimInWater(grid) {
    const n = grid.length;
    const visited = Array.from({ length: n }, () => Array(n).fill(false));
    const heap = [[grid[0][0], 0, 0]];
    const dirs = [[1, 0], [0, 1], [-1, 0], [0, -1]];
    while (heap.length) {
        heap.sort((a, b) => a[0] - b[0]);

        const [h, x, y] = heap.shift();
        if (x === n - 1 && y === n - 1) return h;
        visited[x][y] = true;

        for (const [dx, dy] of dirs) {
            const nx = x + dx, ny = y + dy;
            if (nx >= 0 && ny >= 0 && nx < n && ny < n && !visited[nx][ny]) {
                heap.push([Math.max(h, grid[nx][ny]), nx, ny]);
            }
        }
    }
}
/**
 * Problem 30: Decode Ways
 * Problem Statement:
 * Count the number of ways to decode a digit string into letters (1->A, 2->B...26->Z).
 * Intuition:
 * Use dynamic programming to check valid single and double-digit decodings.
 * Logic:
 * 1. dp[i] = ways to decode s[0..i-1].
 * 2. dp[0] = 1.
 * 3. Check 1-digit and 2-digit validity, add dp[i-1] and dp[i-2] accordingly.
 */
function numDecodings(s) {
    const n = s.length;
    const dp = Array(n + 1).fill(0);
    dp[0] = 1;
    for (let i = 1; i <= n; i++) {
        if (s[i - 1] !== '0') dp[i] += dp[i - 1];
        if (i > 1 && parseInt(s.slice(i - 2, i)) >= 10 && parseInt(s.slice(i - 2, i)) <= 26)
            dp[i] += dp[i - 2];
    }
    return dp[n];
}
/**
 * Problem 31: Unique Paths
 * Problem Statement:
 * Find the number of unique paths from top-left to bottom-right in m x n grid.
 * Intuition:
 * Use DP; each cell's paths = paths from top + left.
 * Logic:
 * 1. Initialize dp[m][n] with 1s.
 * 2. dp[i][j] = dp[i-1][j] + dp[i][j-1].
 */
function uniquePaths(m, n) {
    const dp = Array.from({ length: m }, () => Array(n).fill(1));
    for (let i = 1; i < m; i++) {
        for (let j = 1; j < n; j++)
            {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
    }
    return dp[m - 1][n - 1];
}
/**
 * Problem 32: Interleaving Strings
 * Problem Statement:
 * Check if s3 is formed by interleaving s1 and s2.
 * Intuition:
 * DP tracks feasible prefixes from s1 and s2 forming s3.
 * Logic:
 * 1. dp[i][j] = true if s3[0..i+j-1] formed by s1[0..i-1] and s2[0..j-1].
 * 2. Fill dp table considering single-character extensions from s1 or s2.
 */
function isInterleave(s1, s2, s3) {
    if (s1.length + s2.length !== s3.length) return false;
    const m = s1.length, n = s2.length;
    const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(false));
    dp[0][0] = true;
    for (let i = 0; i <= m; i++) {
        for (let j = 0; j <= n; j++) {
            if (i > 0 && s1[i - 1] === s3[i + j - 1]) dp[i][j] = dp[i][j] || dp[i - 1][j];
            if (j > 0 && s2[j - 1] === s3[i + j - 1]) dp[i][j] = dp[i][j] || dp[i][j - 1];
        }
    }
    return dp[m][n];
}
/**
 * Problem 33: Longest Increasing Path in a Matrix
 * Problem Statement:
 * Find the length of longest increasing path in a 2D matrix.
 * Intuition:
 * DFS with memoization from each cell; only move to neighbors with greater values.
 * Logic:
 * 1. dfs(i,j) returns longest increasing path starting at (i,j).
 * 2. Memoize results to avoid recomputation.
 */
function longestIncreasingPath(matrix) {
    const m = matrix.length, n = matrix[0].length;
    const memo = Array.from({ length: m }, () => Array(n).fill(0));
    const dirs = [[1, 0], [0, 1], [-1, 0], [0, -1]];
    function dfs(x, y) {
        if (memo[x][y]) return memo[x][y];
        let maxLen = 1;
        for (const [dx, dy] of dirs) {
            const nx = x + dx, ny = y + dy;
            if (nx >= 0 && ny >= 0 && nx < m && ny < n && matrix[nx][ny] > matrix[x][y]) {
```