```
/**
 * Problem 1: Encode and Decode Strings
 * Problem Statement:
 * Implement encode and decode functions that convert a list of strings to a sing
 * Intuition:
 * Use a delimiter or length prefix to join strings uniquely so decoding is unamb
 * 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 cont
 * 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
 * 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.
 * Logic:
 * 1. Initialize row=0, col=matrix[0].length-1.
 * 2. While in bounds, adjust row or col based on comparison.
 */
function searchMatrix(matrix, target) {
```

```
        let r = 0, c = matrix[0].length - 1;
        while (r < matrix.length && c >= 0) {
            if (matrix[r][c] === target) return true;
            else if (matrix[r][c] > target) c--;
            else r++;
        }
        return false;
    }

    /**
     * Problem 8: Time Based Key-Value Store
     * Problem Statement:
     * Store values with timestamps and retrieve value for key at given timestamp.
     * Intuition:
     * Store timestamped values in array; binary search to find closest <= timestamp.
     * Logic:
     * 1. Map key -> array of [timestamp,value].
     * 2. Binary search in array for timestamp <= given.
     */
    class TimeMap {
        constructor() { this.map = new Map(); }
        set(key, value, timestamp) {
            if (!this.map.has(key)) {
                this.map.set(key, []);
            }

            this.map.get(key).push([timestamp, value]);
        }
        get(key, timestamp) {
            const arr = this.map.get(key) || [];
            let l = 0, r = arr.length - 1, res = "";
            while (l <= r) {
                const m = Math.floor((l + r) / 2);
                if (arr[m][0] <= timestamp) {
                    res = arr[m][1]; l = m + 1;
                }
                else r = m - 1;
            }
            return res;
        }
```

```
}

/**
 * Problem 9: Add Two Numbers (Linked List)
 * Problem Statement:
 * Given two linked lists representing numbers in reverse, return sum as linked l
 * Intuition:
 * Traverse both lists, sum digits and carry.
 * Logic:
 * 1. Initialize dummy head and carry.
 * 2. Traverse lists, compute sum and carry.
 * 3. Append new node for each digit.
 */
function ListNode(val, next) {
    this.val = val;
    this.next = next || null;
}

function addTwoNumbers(l1, l2) {
    const dummy = new ListNode(0);
    let p = l1, q = l2, curr = dummy, carry = 0;
    while (p || q || carry) {
        const sum = (p ? p.val : 0) + (q ? q.val : 0) + carry;
        carry = Math.floor(sum / 10);
        curr.next = new ListNode(sum % 10);
        curr = curr.next;
        if (p) p = p.next;
        if (q) q = q.next;
    }
    return dummy.next;
}

/**
 * Problem 10: Find Duplicate Number in Linked List
 * Problem Statement:
 * Given linked list with values 1..n and one duplicate, find duplicate.
 * Intuition:
 * Use Floyd’s Tortoise and Hare cycle detection to find duplicate.
 * Logic:
 * 1. Use slow and fast pointers to detect cycle.
```

```
 * 2. Find entry point which is duplicate.
 */
function findDuplicate(nums) {
    let slow = nums[0], fast = nums[0];
    do {
        slow = nums[slow]; fast = nums[nums[fast]];
    } while (slow !== fast);

    slow = nums[0];
    while (slow !== fast) {
        slow = nums[slow];
        fast = nums[fast];
    }
    return slow;
}

/**
 * Problem 11: Reverse Nodes in k-Group
 * Problem Statement:
 * Reverse every k nodes in a linked list.
 * Intuition:
 * Reverse linked list in segments of k.
 * Logic:
 * 1. Count k nodes, reverse them.
 * 2. Connect reversed segment to next.
 */
function reverseKGroup(head, k) {
    const reverse = (start, end) => {
        let prev = end, curr = start;
        while (curr !== end) {
            const tmp = curr.next;
            curr.next = prev;
            prev = curr;
            curr = tmp;
        }
        return prev;
    };
    let node = head, count = 0;
    while (node) { count++; node = node.next; }
    if (count < k) return head;
```

```
        let newHead = reverse(head, node);
        head.next = reverseKGroup(node, k);
        return newHead;
    }

    /**
     * Problem 12: Maximum Depth of Binary Tree
     * Problem Statement:
     * Find maximum depth of binary tree.
     * Intuition:
     * Use DFS to traverse tree.
     * Logic:
     * 1. Depth = 1 + max(depth of left, depth of right)
     */
    function maxDepth(root) {
        if (!root) return 0;
        return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
    }

    /**
     * Problem 13: Same Tree
     * Problem Statement:
     * Determine if two binary trees are identical.
     * Intuition:
     * Recursively compare nodes.
     * Logic:
     * 1. Check root value, left and right subtrees.
     */
    function isSameTree(p, q) {
        if (!p && !q) return true;
        if (!p || !q) return false;
        return p.val === q.val && isSameTree(p.left, q.left) && isSameTree(p.right, q
    }

    /**
     * Problem 14: Subtree of Another Tree
     * Problem Statement:
     * Check if tree t is subtree of s.
     * Intuition:
     * Check every node in s to see if it matches t.
```

```
 * Logic:
 * 1. Traverse s, call isSameTree at each node.
 */
function isSubtree(s, t) {
    if (!s) return false;
    return isSameTree(s, t) || isSubtree(s.left, t) || isSubtree(s.right, t);
}

/**
 * Problem 15: Count Good Nodes in Binary Tree
 * Problem Statement:
 * Count nodes where value >= max on path from root.
 * Intuition:
 * DFS with tracking max value.
 * Logic:
 * 1. If node.val >= maxVal, increment count.
 * 2. Recurse left and right with updated max.
 */
function goodNodes(root) {
    let count = 0;
    const dfs = (node, maxVal) => {
        if (!node) return;
        if (node.val >= maxVal) count++;
        const newMax = Math.max(maxVal, node.val);
        dfs(node.left, newMax);
        dfs(node.right, newMax);
    };
    dfs(root, root.val);
    return count;
}

/**
 * Problem 16: Validate Binary Search Tree
 * Problem Statement:
 * Check if tree is BST.
 * Intuition:
 * Recursively check node values between min and max.
 * Logic:
 * 1. Each node.val must be in (min, max)
 */
```

```
function isValidBST(root) {
    const helper = (node, min, max) => {
        if (!node) return true;
        if (node.val <= min || node.val >= max) return false;
        return helper(node.left, min, node.val) && helper(node.right, node.val, m
    };
    return helper(root, -Infinity, Infinity);
}

/**
 * Problem 17: Kth Smallest Element in BST
 * Problem Statement:
 * Given a binary search tree, return the kth smallest element.
 * Intuition:
 * In a BST, inorder traversal visits nodes in sorted order.
 * So, performing an inorder traversal allows us to count nodes until the kth is
 * Logic:
 * 1. Perform inorder traversal recursively.
 * 2. Keep a counter to track number of nodes visited.
 * 3. When counter reaches k, return the node's value.
 */
function kthSmallest(root, k) {
    let count = 0, result = null;
    const inorder = (node) => {
        if (!node || result !== null) return;
        inorder(node.left);
        count++;
        if (count === k) { result = node.val; return; }
        inorder(node.right);
    };
    inorder(root);
    return result;
}

/**
 * Problem 18: Trie Prefix
 * Problem Statement:
 * Implement Trie with insert, search, and startsWith.
 * Intuition:
 * Use tree structure where each node represents a character.
```

```
 * Logic:
 * 1. Insert: traverse/create nodes for each character.
 * 2. Search: traverse nodes and check endOfWord.
 * 3. startsWith: traverse nodes, return true if path exists.
 */
class TrieNode {
    constructor() {
        this.children = {};
        this.endOfWord = false;
    }
}

class Trie {
    constructor() { this.root = new TrieNode(); }
    insert(word) {
        let node = this.root;
        for (const c of word) {
            if (!node.children[c]) node.children[c] = new TrieNode();
            node = node.children[c];
        }
        node.endOfWord = true;
    }
    search(word) {
        let node = this.root;
        for (const c of word) {
            if (!node.children[c]) return false;
            node = node.children[c];
        }
        return node.endOfWord;
    }
    startsWith(prefix) {
        let node = this.root;
        for (const c of prefix) {
            if (!node.children[c]) return false;
            node = node.children[c];
        }
        return true;
    }
}
```

```
/**
 * Problem 19: Word Search
 * Problem Statement:
 * Check if a word exists in 2D board by connecting adjacent letters.
 * Intuition:
 * Backtracking search for word from each cell.
 * Logic:
 * 1. DFS from each cell, mark visited.
 * 2. Recurse to neighbors, backtrack after.
 */
function existWord(board, word) {
    const m = board.length, n = board[0].length;
    const dfs = (i, j, idx) => {
        if (idx === word.length) return true;
        if (i < 0 || j < 0 || i >= m || j >= n || board[i][j] !== word[idx]) retu

        const tmp = board[i][j]; board[i][j] = '#';
        const res = dfs(i+1,j,idx+1)||dfs(i-1,j,idx+1)||dfs(i,j+1,idx+1)||dfs(i,j
        board[i][j] = tmp;
        return res;
    };
    for (let i=0;i{
        if(!node) return;
        if(node.endOfWord) res.add(path);
        const c = board[i][j];
        board[i][j] = '#';

        for(const [dx,dy] of [[1,0],[0,1],[-1,0],[0,-1]]){
            const x=i+dx,y=j+dy;
            if(x>=0&&y>=0&&xa-b).slice(-k);
    }
    add(val) {
        this.heap.push(val);
        this.heap.sort((a,b)=>a-b);
        if(this.heap.length>this.k) {
            this.heap.shift();
        }
        return this.heap[0];
    }
}
```

```
/**
 * Problem 22: Letter Combinations of a Phone Number
 * Problem Statement:
 * Return all possible letter combinations for digit string.
 * Intuition:
 * Use backtracking to explore each digit’s letters.
 * Logic:
 * 1. Map digits to letters.
 * 2. Backtrack recursively.
 */
function letterCombinations(digits) {
    if(!digits) return [];
    const map = {"2":"abc","3":"def","4":"ghi","5":"jkl","6":"mno","7":"pqrs","8"
    const res=[];
    const backtrack=(idx,path)=>{
        if(path.length===digits.length){
            res.push(path); return;
        }
        for(const c of map[digits[idx]]) {
            backtrack(idx+1,path+c);
        }
    };
    backtrack(0,"");
    return res;
}

/**
 * Problem 23: Surrounded Regions
 * Problem Statement:
 * Capture regions

}

/**
 * Problem 23: Surrounded Regions
 * Problem Statement:
 * Capture regions surrounded by 'X' on 2D board.
 * Intuition:
 * DFS from borders, mark non-surrounded 'O's, then flip rest.
```

```
 * Logic:
 * 1. DFS from borders, mark 'O' as 'T'.
 * 2. Flip 'O' to 'X', 'T' back to 'O'.
 */
function solveSurroundedRegions(board) {
    const m = board.length, n = board[0].length;
    const dfs = (i, j) => {
        if (i < 0 || j < 0 || i >= m || j >= n || board[i][j] !== 'O') return;
        board[i][j] = 'T';
        dfs(i + 1, j); dfs(i - 1, j); dfs(i, j + 1); dfs(i, j - 1);
    };
    for (let i = 0; i < m; i++) { dfs(i, 0); dfs(i, n - 1); }
    for (let j = 0; j < n; j++) { dfs(0, j); dfs(m - 1, j); }
    for (let i = 0; i < m; i++)
         for (let j = 0; j < n; j++)
        {
            board[i][j] = board[i][j] === 'T' ? 'O' : 'X';
        }
}

/**
 * Problem 24: Course Schedule II
 * Problem Statement:
 * Return order to finish courses given prerequisites.
 * Intuition:
 * Topological sort via DFS.
 * Logic:
 * 1. Build adjacency list.
 * 2. DFS with cycle detection.
 */
function findOrder(numCourses, prerequisites) {
    const adj = Array.from({ length: numCourses }, () => []);
    for (const [c, p] of prerequisites) adj[p].push(c);
    const res = [], visited = Array(numCourses).fill(0);
    const dfs = (i) => {
        if (visited[i] === 1) return false;
        if (visited[i] === 2) return true;
        visited[i] = 1;
        for (const nei of adj[i]) if (!dfs(nei)) return false;
        visited[i] = 2; res.push(i); return true;
```

```
        };
        for (let i = 0; i < numCourses; i++) if (!dfs(i)) return [];
        return res.reverse();
    }

    /**
     * Problem 25: Redundant Connection
     * Problem Statement:
     * Find edge creating cycle in undirected graph.
     * Intuition:
     * Use Union-Find.
     * Logic:
     * 1. Union nodes, detect if already connected.
     */
    function findRedundantConnection(edges) {
        const parent = [];
        const find = (x) => {
            if (parent[x] === x) return x;
            //else
            parent[x] = find(parent[x]);
            return parent[x]; };
        const union = (x, y) => {
            const px = find(x), py = find(y);
            if (px === py) return false;
            parent[px] = py;
            return true;
        };
        for (let i = 0; i <= edges.length; i++) parent[i] = i;
        for (const [u, v] of edges) if (!union(u, v)) return [u, v];
    }

    /**
     * Problem 26: Graph Valid Tree
     * Problem Statement:
     * Check if undirected graph is tree.
     * Intuition:
     * Tree = connected + no cycle.
     * Logic:
     * 1. Use Union-Find to detect cycle.
     * 2. Ensure n-1 edges.
```

```
 */
function validTree(n, edges) {
    if (edges.length !== n - 1) return false;
    const parent = Array.from({ length: n }, (_, i) => i);
    const find = x => {
        if (parent[x] === x) return x;
        //else
        parent[x] = find(parent[x]);
        return parent[x]; };
    for (const [u, v] of edges) {
        const pu = find(u), pv = find(v);
        if (pu === pv) return false;
        parent[pu] = pv;
    }
    return true;
}

/**
 * Problem 27: Cheapest Flights Within K Stops
 * Problem Statement:
 * Find cheapest flight from src to dst with at most k stops.
 * Intuition:
 * BFS with level = stops, keep best cost.
 * Logic:
 * 1. Build adjacency list.
 * 2. BFS while tracking cost and stops.
 */
function findCheapestPrice(n, flights, src, dst, K) {
    const adj = Array.from({ length: n }, () => []);
    for (const [u, v, w] of flights)
        {
            adj[u].push([v, w]);
        }
    let q = [[src, 0, 0]], res = Infinity;
    while (q.length) {
        const [node, cost, stops] = q.shift();
        if (node === dst) {
            res = Math.min(res, cost);
        }
        if (stops > K) continue;
```

```
            for (const [nei, w] of adj[node]) {
                q.push([nei, cost + w, stops + 1]);
            }
        }
        return res === Infinity ? -1 : res;
    }

    /**
     * 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] - pc
        }

        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 wa
 * 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...2
 * 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,
            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
            if (j > 0 && s2[j - 1] === s3[i + j - 1]) dp[i][j] = dp[i][j] || dp[i
        }
    }
    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 value
 * 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
                    maxLen = Math.max(maxLen, 1 + dfs(nx, ny));
                }
            }
            memo[x][y] = maxLen;
            return maxLen;
        }

        let res = 0;
        for (let i = 0; i < m; i++) {
            for (let j = 0; j < n; j++) {
                res = Math.max(res, dfs(i, j));
            }
        }
        return res;
    }
    /**
     * Problem 34: Burst Balloons
     * Problem Statement:
     * Max coins by bursting balloons in optimal order.
     * Intuition:
     * DP: consider last balloon in subarray; sum coins from left, right, and itself.
     * Logic:
     * 1. Pad nums with 1 at both ends.
     * 2. dp[i][j] = max coins for bursting balloons in (i,j).
     * 3. Iterate all possible k for last balloon in subarray.
     */
    function maxCoins(nums) {
        nums = [1, ...nums, 1];
        const n = nums.length;
        const dp = Array.from({ length: n }, () => Array(n).fill(0));
        for (let len = 2; len < n; len++) {
            for (let i = 0; i + len < n; i++) {
                const j = i + len;
                for (let k = i + 1; k < j; k++) {
                    dp[i][j] = Math.max(dp[i][j], dp[i][k] + dp[k][j] + nums[i] * num
                }
```

```
            }
        }
        return dp[0][n - 1];
    }

    /**
     * Problem 35: Regular Expression Matching
     * Problem Statement:
     * Match string s with pattern p including '.' and '*'.
     * Intuition:
     * DP stores match results for prefixes; '*' handles zero or more of previous cha
     * Logic:
     * 1. dp[i][j] = s[0..i-1] matches p[0..j-1].
     * 2. Fill dp using single character match or '*' zero/more handling.
     */
    function isMatch(s, p) {
        const m = s.length, n = p.length;
        const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(false));
        dp[0][0] = true;

        // Handle patterns like a*, a*b*, etc.
        for (let j = 1; j <= n; j++) {
            if (p[j - 1] === '*') {
                dp[0][j] = dp[0][j - 2];
            }
        }

        for (let i = 1; i <= m; i++) {
            for (let j = 1; j <= n; j++) {
                if (p[j - 1] === s[i - 1] || p[j - 1] === '.') {
                    dp[i][j] = dp[i - 1][j - 1];
                } else if (p[j - 1] === '*') {
                    dp[i][j] = dp[i][j - 2] || ((p[j - 2] === s[i - 1] || p[j - 2] ==
                }
            }
        }

        return dp[m][n];
    }
    // Continue similarly for remaining problems 36-48
```

```
// (Jump Game 2, Hand of Straights, Merge Triples to Form Triplet, Partition Labe
/**
 * Problem 36: Jump Game II
 * Problem Statement:
 * Given array of max jump lengths, return minimum number of jumps to reach last
 * Intuition:
 * Greedy approach: always jump to farthest reachable index within current range.
 * Logic:
 * 1. Track current max reachable index.
 * 2. When reaching end of current jump, increase jump count and update range.
 */
function jumps2(nums) {
    let jumps = 0, curEnd = 0, curFarthest = 0;

    for (let i = 0; i < nums.length - 1; i++) {
        curFarthest = Math.max(curFarthest, i + nums[i]);
        if (i === curEnd) {
            jumps++;
            curEnd = curFarthest;
        }
    }
    return jumps;
}

/**
 * Problem 37: Hand of Straights
 * Problem Statement:
 * Check if hand can be rearranged into groups of W consecutive cards.
 * Intuition:
 * Count frequencies and always form consecutive groups from smallest available c
 * Logic:
 * 1. Count cards using map.
 * 2. Sort keys, reduce frequencies for next W-1 consecutive numbers.
 */
function isNStraightHand(hand, W) {
    if (hand.length % W !== 0) return false;
    const count = new Map();
    for (const h of hand) count.set(h, (count.get(h) || 0) + 1);
    const keys = [...count.keys()].sort((a, b) => a - b);
    for (const k of keys) {
```

```
            const freq = count.get(k);
            if (freq > 0) {
                for (let i = 0; i < W; i++) {
                    if ((count.get(k + i) || 0) < freq) return false;
                    count.set(k + i, count.get(k + i) - freq);
                }
            }
        }
        return true;
    }

    /**
     * Problem 38: Merge Triples to Form Triplet
     * Problem Statement:
     * Given list of triples [a,b,c], return max triplet possible by merging smaller
     * Intuition:
     * Track max values for each index; only update if a new triple increases any com
     * Logic:
     * 1. Sort triples by first element.
     * 2. Update max second and third values while iterating.
     */
    function mergeTriplets(triples, target) {
        let maxB = 0, maxC = 0;
        for (const [a, b, c] of triples) {
            if (a <= target[0] && b <= target[1] && c <= target[2]) {
                if (b > maxB) maxB = b;
                if (c > maxC) maxC = c;
            }
        }
        return maxB === target[1] && maxC === target[2];
    }

    /**
     * Problem 39: Partition Labels
     * Problem Statement:
     * Partition string so that each letter appears in at most one part; return sizes
     * Intuition:
     * Track last occurrence of each character; partition when current index reaches
     * Logic:
     * 1. Record last index for each char.
```

```
 * 2. Expand partition until current index reaches end of chars in partition.
 */
function partitionLabels(S) {
    const last = {};
    for (let i = 0; i < S.length; i++) last[S[i]] = i;
    const res = [];
    let start = 0, end = 0;
    for (let i = 0; i < S.length; i++) {
        end = Math.max(end, last[S[i]]);
        if (i === end) {
            res.push(end - start + 1);
            start = i + 1;
        }
    }
    return res;
}

/**
 * Problem 40: Meeting Rooms II
 * Problem Statement:
 * Find minimum number of meeting rooms needed for given intervals.
 * Intuition:
 * Sort start and end times; track overlapping intervals using two pointers.
 * Logic:
 * 1. Sort startTimes and endTimes separately.
 * 2. Increment rooms when new meeting starts before previous ends.
 */
function minMeetingRooms(intervals) {
    const starts = intervals.map(i => i[0]).sort((a, b) => a - b);
    const ends = intervals.map(i => i[1]).sort((a, b) => a - b);
    let s = 0, e = 0, rooms = 0, maxRooms = 0;
    while (s < starts.length) {
        if (starts[s] < ends[e]) {
            rooms++;
            s++;
            maxRooms = Math.max(maxRooms, rooms);
        }
        else { rooms--;
            e++;
        }
```

```
        }
        return maxRooms;
    }

    /**
     * Problem 41: Minimum Interval to Include Each Query
     * Problem Statement:
     * For each query, find smallest interval that contains it.
     * Intuition:
     * Sort intervals and queries; use min-heap to track smallest interval covering c
     * Logic:
     * 1. Sort intervals by start, queries with indices.
     * 2. Push intervals starting before query into heap by size.
     * 3. Remove intervals ending before query.
     */
    function minInterval(intervals, queries) {
        intervals.sort((a, b) => a[0] - b[0]);
        const q = queries.map((v, i) => [v, i]).sort((a, b) => a[0] - b[0]);
        const res = Array(queries.length).fill(-1);
        const heap = [];
        let i = 0;
        for (const [val, idx] of q) {
            while (i < intervals.length && intervals[i][0] <= val) {
                const [l, r] = intervals[i];
                heap.push([r - l + 1, r]);
                i++;
            }
            heap.sort((a, b) => a[0] - b[0]);
            while (heap.length && heap[0][1] < val) heap.shift();
            if (heap.length) res[idx] = heap[0][0];
        }
        return res;
    }

    /**
     * Problem 42: Rotate Image
     * Problem Statement:
     * Rotate n x n 2D matrix 90 degrees clockwise in-place.
     * Intuition:
     * First transpose matrix, then reverse each row.
```

```
 * Logic:
 * 1. Transpose matrix (swap[i][j] with [j][i]).
 * 2. Reverse each row.
 */
function rotate(matrix) {
    const n = matrix.length;
    // Transpose the matrix
    for (let i = 0; i < n; i++) {
        for (let j = i; j < n; j++) {
            [matrix[i][j], matrix[j][i]] = [matrix[j][i], matrix[i][j]];
        }
    }
    // Reverse each row
    for (let row of matrix) {
        row.reverse();
    }
}

/**
 * Problem 43: Happy Number
 * Problem Statement:
 * Determine if number eventually reaches 1 when repeatedly replaced by sum of sq
 * Intuition:
 * Detect cycles using Set.
 * Logic:
 * 1. Replace number by sum of squares of digits.
 * 2. Store seen numbers; if repeat occurs, not happy.
 */
function isHappy(n) {
    const seen = new Set();
    while (n !== 1 && !seen.has(n)) {
        seen.add(n);
        n = n.toString().split('').reduce((acc, d) => acc + Math.pow(parseInt(d),
    }
    return n === 1;
}

/**
 * Problem 44: Set Matrix Zeroes
 * Problem Statement:
```

```
 * If matrix element is 0, set its row and column to 0.
 * Intuition:
 * Use first row and column as markers to avoid extra space.
 * Logic:
 * 1. Mark zeros in first row/col.
 * 2. Set zeros accordingly.
 */
function setZeroes(matrix) {
    const m = matrix.length, n = matrix[0].length;
    let firstRow = false, firstCol = false;
    for (let i = 0; i < m; i++) if (matrix[i][0] === 0) firstCol = true;
    for (let j = 0; j < n; j++) if (matrix[0][j] === 0) firstRow = true;
    for (let i = 1; i < m; i++)
        for (let j = 1; j < n; j++)
            if (matrix[i][j] === 0) matrix[i][0] = matrix[0][j] = 0;
    for (let i = 1; i < m; i++)
        for (let j = 1; j < n; j++)
            if (matrix[i][0] === 0 || matrix[0][j] === 0) matrix[i][j] = 0;
    if (firstRow) matrix[0].fill(0);
    if (firstCol) for (let i = 0; i < m; i++) matrix[i][0] = 0;
}

/**
 * Problem 45: Multiply Strings
 * Problem Statement:
 * Multiply two non-negative numbers represented as strings.
 * Intuition:
 * Simulate multiplication like hand-calculation using array to store digits.
 * Logic:
 * 1. Multiply each digit pair and store in correct position.
 * 2. Handle carries, convert array to string.
 */
function multiply(num1, num2) {
    const m = num1.length, n = num2.length;
    const res = Array(m + n).fill(0);
    for (let i = m - 1; i >= 0; i--) {
        for (let j = n - 1; j >= 0; j--) {
            const mul = (num1[i] - '0') * (num2[j] - '0');
            const sum = mul + res[i + j + 1];
            res[i + j + 1] = sum % 10;
```

```
                res[i + j] += Math.floor(sum / 10);
            }
        }
        while (res[0] === 0 && res.length > 1) res.shift();
        return res.join('');
    }

    /**
     * Problem 46: Palindromic Substrings
     * Problem Statement:
     * Count all palindromic substrings in string.
     * Intuition:
     * Expand around each center to detect palindromes.
     * Logic:
     * 1. Consider centers at each index (odd and even length).
     * 2. Expand and count.
     */
    function countSubstrings(s) {
        let count = 0;
        const expand = (l, r) => {
            while (l >= 0 && r < s.length && s[l] === s[r]) {
                count++; l--; r++;
            }
        };
        for (let i = 0; i < s.length; i++) {
            expand(i, i);
            expand(i, i + 1);
        }
        return count;
    }

    /**
     * Problem 47: Partition Equal Subsets
     * Problem Statement:
     * Determine if array can be partitioned into two subsets with equal sum.
     * Intuition:
     * Use DP to check if subset sum equals half of total sum.
     * Logic:
     * 1. Compute total sum, target = total/2.
     * 2. dp[i] = whether subset sum i is achievable.
```

```
 */
function canPartition(nums) {
    const sum = nums.reduce((a, b) => a + b, 0);
    if (sum % 2 !== 0) return false;
    const target = sum / 2;
    const dp = Array(target + 1).fill(false);
    dp[0] = true;
    for (const num of nums) {
        for (let i = target; i >= num; i--) dp[i] = dp[i] || dp[i - num];
    }
    return dp[target];
}

/**
 * Problem 48: Last Stone Weight
 * Problem Statement:
 * Smash two largest stones until one or none left; return weight of remaining st
 * Intuition:
 * Use max-heap to repeatedly extract largest stones.
 * Logic:
 * 1. Sort array in descending order.
 * 2. Pop two largest, push difference if non-zero, repeat.
 */
function lastStoneWeight(stones) {
    stones.sort((a, b) => b - a);
    while (stones.length > 1) {
        const y = stones.shift(), x = stones.shift();
        if (y !== x) {
            stones.push(y - x);
            stones.sort((a, b) => b - a);
        }
    }
    return stones.length ? stones[0] : 0;
}

/**
 * Problem 41: Minimum Interval to Include Each Query
 * Problem Statement:
 * For each query, find the length of smallest interval that includes it.
 * Intuition:
```

```
 * Sort intervals and queries; use a min-heap to keep track of interval lengths c
 * Logic:
 * 1. Sort intervals by start, queries by value.
 * 2. Push intervals starting before query into min-heap.
 * 3. Remove intervals ending before query.
 * 4. Top of heap gives smallest interval.
 */
function minInterval(intervals, queries) {
    intervals.sort((a, b) => a[0] - b[0]);
    const sortedQueries = queries.map((q, i) => [q, i]).sort((a, b) => a[0] - b[0
    const res = Array(queries.length).fill(-1);
    const heap = [];
    let i = 0;
    for (const [q, idx] of sortedQueries) {
        while (i < intervals.length && intervals[i][0] <= q) {
            const [start, end] = intervals[i];
            heap.push([end - start + 1, end]);
            i++;
        }
        heap.sort((a, b) => a[0] - b[0]); // min-heap by interval length
        while (heap.length && heap[0][1] < q) heap.shift();
        if (heap.length) res[idx] = heap[0][0];
    }
    return res;
}

/**
 * Problem 42: Rotate Image
 * Problem Statement:
 * Rotate n x n matrix 90 degrees clockwise.
 * Intuition:
 * Transpose + reverse rows.
 * Logic:
 * 1. Swap matrix[i][j] with matrix[j][i].
 * 2. Reverse each row.
 */
function rotate(matrix) {
    const n = matrix.length;
    for (let i = 0; i < n; i++) {
        for (let j = i; j < n; j++) {
```

```
                [matrix[i][j], matrix[j][i]] = [matrix[j][i], matrix[i][j]];
            }
        }
        for (let i = 0; i < n; i++) matrix[i].reverse();
    }

    /**
     * Problem 43: Happy Number
     * Problem Statement:
     * Determine if number eventually reaches 1 by replacing number with sum of squar
     * Intuition:
     * Detect cycle using set or Floyd's cycle detection.
     * Logic:
     * 1. Loop: compute sum of squares.
     * 2. If 1, return true. If seen before, return false.
     */
    function isHappy(n) {
        const seen = new Set();
        while (n !== 1 && !seen.has(n)) {
            seen.add(n);
            n = String(n).split('').reduce((sum, d) => sum + d*d, 0);
        }
        return n === 1;
    }

    /**
     * Problem 44: Set Matrix Zeroes
     * Problem Statement:
     * Set entire row and column to 0 if an element is 0 in matrix.
     * Intuition:
     * Use first row/col as markers to save space.
     * Logic:
     * 1. Track if first row/col has zero.
     * 2. Use rest of matrix to mark zero rows/cols.
     * 3. Zero rows/cols accordingly.
     */
    function setZeroes(matrix) {
        const m = matrix.length, n = matrix[0].length;
        let firstRow = false, firstCol = false;
        for (let i = 0; i < m; i++) if (matrix[i][0] === 0) firstCol = true;
```

```
        for (let j = 0; j < n; j++) if (matrix[0][j] === 0) firstRow = true;
        for (let i = 1; i < m; i++) {
            for (let j = 1; j < n; j++) {
                if (matrix[i][j] === 0) { matrix[i][0] = 0; matrix[0][j] = 0; }
            }
        }
        for (let i = 1; i < m; i++) {
            for (let j = 1; j < n; j++) {
                if (matrix[i][0] === 0 || matrix[0][j] === 0) matrix[i][j] = 0;
            }
        }
        if (firstRow) for (let j = 0; j < n; j++) matrix[0][j] = 0;
        if (firstCol) for (let i = 0; i < m; i++) matrix[i][0] = 0;
    }

    /**
     * Problem 45: Multiply Strings
     * Problem Statement:
     * Multiply two non-negative integers given as strings.
     * Intuition:
     * Simulate digit-by-digit multiplication.
     * Logic:
     * 1. Initialize array for result.
     * 2. Multiply digits, add to positions.
     * 3. Carry over.
     */
    function multiply(num1, num2) {
        const m = num1.length, n = num2.length;
        const res = Array(m + n).fill(0);
        for (let i = m - 1; i >= 0; i--) {
            for (let j = n - 1; j >= 0; j--) {
                const mul = (num1[i] - '0') * (num2[j] - '0');
                const sum = mul + res[i + j + 1];
                res[i + j + 1] = sum % 10;
                res[i + j] += Math.floor(sum / 10);
            }
        }
        while (res[0] === 0 && res.length > 1) res.shift();
        return res.join('');
    }
```

```
/**
 * Problem 46: Palindromic Substrings
 * Problem Statement:
 * Count all palindromic substrings in string.
 * Intuition:
 * Expand around center.
 * Logic:
 * 1. For each center (i, i) and (i, i+1), expand while palindrome.
 */
function countSubstrings(s) {
    let count = 0;
    const expand = (l, r) => {
        while (l >= 0 && r < s.length && s[l] === s[r]) { count++; l--; r++; }
    };
    for (let i = 0; i < s.length; i++) { expand(i, i); expand(i, i+1); }
    return count;
}

/**
 * Problem 47: Partition Equal Subsets
 * Problem Statement:
 * Determine if array can be partitioned into two subsets with equal sum.
 * Intuition:
 * Subset sum problem.
 * Logic:
 * 1. dp[i] = can achieve sum i.
 * 2. Iterate nums, update dp.
 */
function canPartition(nums) {
    const sum = nums.reduce((a,b)=>a+b,0);
    if (sum % 2 !== 0) return false;
    const target = sum/2;
    const dp = Array(target+1).fill(false);
    dp[0] = true;
    for (const num of nums) {
        for (let i = target; i >= num; i--) dp[i] = dp[i] || dp[i-num];
    }
    return dp[target];
}
```

```
/**
 * Problem 48: Last Stone Weight
 * Problem Statement:
 * Smash largest two stones until one or none remains.
 * Intuition:
 * Use max-heap to repeatedly smash heaviest stones.
 * Logic:
 * 1. Push all stones into heap.
 * 2. Pop two largest, push difference if non-zero.
 */
function lastStoneWeight(stones) {
    stones.sort((a,b)=>b-a);
    while (stones.length > 1) {
        const y = stones.shift(), x = stones.shift();
        if (y !== x) {
            const diff = y - x;
            let idx = stones.findIndex(s => s < diff);
            if (idx === -1) stones.push(diff);
            else stones.splice(idx, 0, diff);
        }
    }
    return stones.length ? stones[0] : 0;
}
```