```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * Given two non-negative integers `num1` and `num2` represented as strings,
 * return the product of these two numbers, also as a string.
 * You must implement the multiplication manually (without using BigInt or built-in
 * arbitrary-precision libraries).
 *
 * Intuition:
 * ----------
 * This simulates the grade-school multiplication method:
 * - Multiply each digit of `num1` with each digit of `num2` from right to left.
 * - Keep track of carries and add partial products in the correct positions.
 * - Finally, remove leading zeros and return the result as a string.
 *
 * Logic:
 * ------
 * 1. Initialize an array `res` of length `m + n` (where m and n are lengths of `num1` and `num2`)
 *    filled with zeros to store intermediate sums.
 * 2. Iterate backward through both strings:
 *    - Multiply digits `(num1[i] - '0') * (num2[j] - '0')`.
 *    - Add the product to the correct position in `res` using `i + j + 1`.
 *    - Manage carry by adding `Math.floor(sum / 10)` to `res[i + j]`.
 * 3. After finishing the multiplication, remove leading zeros.
 * 4. Join the array into a string and return it.
 */

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('');
}

// -------- Driver Code --------
function main() {
    const num1 = "123";
    const num2 = "45";
    const result = multiply(num1, num2);
    console.log(`${num1} × ${num2} = ${result}`);
}

main();
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * There are n cars going to the same destination along a one-lane road. The destination is at position 'target'.
 * You are given two arrays: 'position' and 'speed', where position[i] is the position of the i-th car and speed[i] is its speed.
 *
 * A car fleet is a group of cars traveling at the same speed, where a faster car catches up to a slower car and they move together.
 * Return the number of car fleets that will arrive at the destination.
 *
 * Example:
 * ----------
 * Input: target = 12, position = [10, 8, 0, 5, 3], speed = [2, 4, 1, 1, 3]
 * Output: 3
 *
 * Explanation:
 * - Car at 10 and speed 2 -> reaches in 1 hour
 * - Car at 8 and speed 4 -> reaches in 1 hour
 *   These two meet at destination forming a fleet.
 * - Car at 0 and speed 1 -> reaches in 12 hours
 * - Car at 5 and speed 1 -> reaches in 7 hours
 * - Car at 3 and speed 3 -> reaches in 3 hours
 *   Cars at 5 and 3 form a fleet eventually with the slower car at 0 remaining separate.
 *
 * Intuition:
 * ----------
 * 1. Sort cars by starting position from closest to the destination to farthest.
 * 2. Iterate over cars:
 *    - Calculate the time each car takes to reach the target.
 *    - If a car's time is greater than the current fleet's time, it forms a new fleet.
 *    - Otherwise, it joins the fleet ahead.
 * 3. Count the number of fleets.
 */

function carFleet(target, position, speed) {
    // Combine positions and speeds, sort descending by position
    const cars = position.map((p, i) => [p, speed[i]])
                         .sort((a, b) => b[0] - a[0]);

    let fleets = 0;     // Number of fleets
    let curTime = 0;    // Time of the last fleet to reach the target

    for (const [pos, spd] of cars) {
        const time = (target - pos) / spd;  // Time to reach target
        // If this car takes longer than the last fleet, it forms a new fleet
        if (time > curTime) {
            fleets++;
            curTime = time;
        }
    }

    return fleets;
}

// Driver code to test the function
const testCases = [
    { target: 12, position: [10, 8, 0, 5, 3], speed: [2, 4, 1, 1, 3], expected: 3 },
    { target: 10, position: [3], speed: [3], expected: 1 },
    { target: 100, position: [0, 2, 4], speed: [4, 2, 1], expected: 1 }
];

testCases.forEach(({target, position, speed, expected}, i) => {
    const result = carFleet(target, position, speed);
    console.log(`Test Case ${i + 1}: Expected = ${expected}, Got = ${result}`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given a list of daily temperatures T, return a list such that, for each day in the input,
 * tells you how many days you would have to wait until a warmer temperature.
 * If there is no future day for which this is possible, put 0 instead.
 *
 * Example:
 * ----------
 * Input: [73, 74, 75, 71, 69, 72, 76, 73]
 * Output: [1, 1, 4, 2, 1, 1, 0, 0]
 *
 * Explanation:
 * - For day 0 (73), the next warmer temperature is on day 1 (74) -> 1 day
 * - For day 1 (74), the next warmer temperature is on day 2 (75) -> 1 day
 * - For day 2 (75), the next warmer temperature is on day 6 (76) -> 4 days
 * - For day 3 (71), the next warmer temperature is on day 5 (72) -> 2 days
 * - For day 4 (69), the next warmer temperature is on day 5 (72) -> 1 day
 * - For day 5 (72), the next warmer temperature is on day 6 (76) -> 1 day
 * - For day 6 (76), no warmer day ahead -> 0
 * - For day 7 (73), no warmer day ahead -> 0
 *
 * Intuition:
 * ----------
 * Use a stack to keep track of indices of temperatures for which we haven't found a warmer day yet.
 * 1. Iterate through each day.
 * 2. While the stack is not empty and the current temperature is higher than the temperature
 *    at the index at the top of the stack:
 *      - Pop that index from the stack
 *      - Compute the difference in days (current index - popped index) and store in result
 * 3. Push the current index onto the stack.
 * 4. At the end, indices still in the stack have no warmer future day, so their result is already 0.
 */

function dailyTemperatures(T) {
    const res = Array(T.length).fill(0); // Initialize result array with 0
    const stack = []; // Stack to keep indices of unresolved temperatures

    for (let i = 0; i < T.length; i++) {
        // While current temperature is higher than temperature at stack top
        while (stack.length && T[i] > T[stack[stack.length - 1]]) {
            const idx = stack.pop(); // Get the index of previous cooler temperature
            res[idx] = i - idx;      // Compute the number of days until a warmer temperature
        }
        stack.push(i); // Push current day's index to stack
    }

    return res;
}

// Driver code to test the function
const testCases = [
    { T: [73, 74, 75, 71, 69, 72, 76, 73], expected: [1, 1, 4, 2, 1, 1, 0, 0] },
    { T: [30, 40, 50, 60], expected: [1, 1, 1, 0] },
    { T: [30, 60, 90], expected: [1, 1, 0] }
];

testCases.forEach(({T, expected}, i) => {
    const result = dailyTemperatures(T);
    console.log(`Test Case ${i + 1}: Expected = [${expected}], Got = [${result}]`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given a string containing digits from 2-9 inclusive, return all possible letter combinations
 * that the number could represent, based on the mapping of digits to letters on a phone keypad.
 *
 * Example:
 * ----------
 * Input: digits = "23"
 * Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
 *
 * Input: digits = ""
 * Output: []
 *
 * Input: digits = "2"
 * Output: ["a","b","c"]
 *
 * Intuition:
 * ----------
 * 1. Use backtracking to explore all combinations.
 * 2. Map each digit to its corresponding letters.
 * 3. Recursively build combinations by appending letters for each digit.
 * 4. Once the path length equals the input digits length, store the combination.
 */

function letterCombinations(digits) {
    if (!digits) return []; // No digits -> return empty array

    // Mapping from digit to letters
    const map = {
        "2": "abc",
        "3": "def",
        "4": "ghi",
        "5": "jkl",
        "6": "mno",
        "7": "pqrs",
        "8": "tuv",
        "9": "wxyz"
    };

    const res = [];

    // Backtracking helper function
    const backtrack = (idx, path) => {
        if (path.length === digits.length) { // Completed a combination
            res.push(path);
            return;
        }

        // Explore all letters for the current digit
        for (const c of map[digits[idx]]) {
            backtrack(idx + 1, path + c);
        }
    };

    backtrack(0, ""); // Start backtracking from index 0
    return res;
}

// Driver code to test letterCombinations
const testCases = [
    { digits: "23", expected: ["ad","ae","af","bd","be","bf","cd","ce","cf"] },
    { digits: "", expected: [] },
    { digits: "2", expected: ["a","b","c"] }
];

testCases.forEach(({digits, expected}, i) => {
    const result = letterCombinations(digits);
    console.log(`Test Case ${i + 1}: Input = "${digits}", Output = [${result}]`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given a binary tree, a node X in the tree is named "good" if in the path from the root to X
 * there are no nodes with a value greater than X.
 *
 * Return the number of good nodes in the binary tree.
 *
 * Example:
 * ----------
 * Input: root = [3,1,4,3,null,1,5]
 * Output: 4
 * Explanation: Nodes 3 (root), 3 (left child of 1), 4, and 5 are good nodes.
 *
 * Input: root = [3,3,null,4,2]
 * Output: 3
 *
 * Input: root = [1]
 * Output: 1
 *
 * Intuition:
 * ----------
 * 1. Perform DFS (Depth-First Search) traversal of the tree.
 * 2. Keep track of the maximum value encountered along the path from root to current node.
 * 3. If current node's value >= max value so far, increment count (good node).
 * 4. Update max value for the path and recurse into left and right subtrees.
 */

function TreeNode(val, left, right) {
    this.val = val;
    this.left = left || null;
    this.right = right || null;
}

function goodNodes(root) {
    let count = 0; // To track the number of good nodes

    // DFS helper function
    const dfs = (node, maxVal) => {
        if (!node) return; // Base case: empty node
        // Check if current node is good
        if (node.val >= maxVal) count++;
        const newMax = Math.max(maxVal, node.val); // Update max value for path
        dfs(node.left, newMax);  // Recurse left
        dfs(node.right, newMax); // Recurse right
    };

    dfs(root, root.val); // Start DFS from root
    return count;
}

// Helper function to create tree from array (level-order) for testing
function arrayToTree(arr) {
    if (!arr.length) return null;
    const root = new TreeNode(arr[0]);
    const queue = [root];
    let i = 1;
    while (i < arr.length) {
        const node = queue.shift();
        if (arr[i] !== null) {
            node.left = new TreeNode(arr[i]);
            queue.push(node.left);
        }
        i++;
        if (i < arr.length && arr[i] !== null) {
            node.right = new TreeNode(arr[i]);
            queue.push(node.right);
        }
        i++;
    }
    return root;
}

// Driver code to test goodNodes
const testCases = [
    { tree: [3,1,4,3,null,1,5], expected: 4 },
    { tree: [3,3,null,4,2], expected: 3 },
    { tree: [1], expected: 1 },
];

testCases.forEach(({tree, expected}, i) => {
    const root = arrayToTree(tree);
    const result = goodNodes(root);
    console.log(`Test Case ${i + 1}: Expected = ${expected}, Got = ${result}`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given two binary trees, write a function to check if they are the same.
 *
 * Two binary trees are considered the same if they are structurally identical and the nodes have the same values.
 *
 * Example:
 * ----------
 * Input:
 *   Tree 1:    1         Tree 2:    1
 *             / \                   / \
 *            2   3                 2   3
 * Output: true
 *
 * Input:
 *   Tree 1:    1         Tree 2:    1
 *             /                     \
 *            2                       2
 * Output: false
 *
 * Intuition:
 * ----------
 * 1. Use recursion to traverse both trees simultaneously.
 * 2. Base cases:
 *    - If both nodes are null, they are the same (return true).
 *    - If one is null and the other is not, trees differ (return false).
 * 3. Compare current node values and recursively check left and right subtrees.
 */

function TreeNode(val, left, right) {
    this.val = val;
    this.left = left || null;
    this.right = right || null;
}

function isSameTree(p, q) {
    if (!p && !q) return true;        // Both nodes null -> same
    if (!p || !q) return false;       // One null -> different
    // Check current node value and recurse on left and right
    return p.val === q.val && isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
}

// Helper function to create tree from array (level-order) for testing
function arrayToTree(arr) {
    if (!arr.length) return null;
    const root = new TreeNode(arr[0]);
    const queue = [root];
    let i = 1;
    while (i < arr.length) {
        const node = queue.shift();
        if (arr[i] !== null) {
            node.left = new TreeNode(arr[i]);
            queue.push(node.left);
        }
        i++;
        if (i < arr.length && arr[i] !== null) {
            node.right = new TreeNode(arr[i]);
            queue.push(node.right);
        }
        i++;
    }
    return root;
}

// Driver code to test isSameTree
const testCases = [
    { p: [1,2,3], q: [1,2,3], expected: true },
    { p: [1,2], q: [1,null,2], expected: false },
    { p: [], q: [], expected: true },
];

testCases.forEach(({p, q, expected}, i) => {
    const treeP = arrayToTree(p);
    const treeQ = arrayToTree(q);
    const result = isSameTree(treeP, treeQ);
    console.log(`Test Case ${i + 1}: Expected = ${expected}, Got = ${result}`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given two non-empty binary trees `s` and `t`, check whether `t` is a subtree of `s`.
 *
 * A subtree of `s` is a tree that consists of a node in `s` and all of its descendants.
 * `s` could also be considered a subtree of itself.
 *
 * Example:
 * ----------
 * Input:
 * s = [3,4,5,1,2], t = [4,1,2]
 * Output: true
 *
 * Input:
 * s = [3,4,5,1,2,null,null,null,null,0], t = [4,1,2]
 * Output: false
 *
 * Intuition:
 * ----------
 * 1. Traverse tree `s` and at each node check if the subtree rooted at that node is identical to `t`.
 * 2. Use the previously defined `isSameTree` function to compare trees.
 * 3. Recursively check the left and right subtrees if current node is not a match.
 */

function TreeNode(val, left, right) {
    this.val = val;
    this.left = left || null;
    this.right = right || null;
}

// Reuse isSameTree function from previous problem
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.right);
}

function isSubtree(s, t) {
    if (!s) return false; // Reached end of tree `s` without match
    // Check if current subtree matches or recurse into left/right subtrees
    return isSameTree(s, t) || isSubtree(s.left, t) || isSubtree(s.right, t);
}

// Helper function to create tree from array (level-order) for testing
function arrayToTree(arr) {
    if (!arr.length) return null;
    const root = new TreeNode(arr[0]);
    const queue = [root];
    let i = 1;
    while (i < arr.length) {
        const node = queue.shift();
        if (arr[i] !== null) {
            node.left = new TreeNode(arr[i]);
            queue.push(node.left);
        }
        i++;
        if (i < arr.length && arr[i] !== null) {
            node.right = new TreeNode(arr[i]);
            queue.push(node.right);
        }
        i++;
    }
    return root;
}

// Driver code to test isSubtree
const testCases = [
    { s: [3,4,5,1,2], t: [4,1,2], expected: true },
    { s: [3,4,5,1,2,null,null,null,null,0], t: [4,1,2], expected: false },
    { s: [1], t: [1], expected: true }
];

testCases.forEach(({s, t, expected}, i) => {
    const treeS = arrayToTree(s);
    const treeT = arrayToTree(t);
    const result = isSubtree(treeS, treeT);
    console.log(`Test Case ${i + 1}: Expected = ${expected}, Got = ${result}`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given a binary tree, determine if it is a valid Binary Search Tree (BST).
 *
 * A BST must satisfy:
 * 1. The left subtree of a node contains only nodes with keys **less than** the node's key.
 * 2. The right subtree of a node contains only nodes with keys **greater than** the node's key.
 * 3. Both left and right subtrees must also be BSTs.
 *
 * Example:
 * ----------
 * Input: root = [2,1,3]
 * Output: true
 *
 * Input: root = [5,1,4,null,null,3,6]
 * Output: false
 *
 * Intuition:
 * ----------
 * 1. Use recursion with a range for each node.
 * 2. Each node must satisfy: min < node.val < max.
 * 3. For left child, update max to current node's value.
 * 4. For right child, update min to current node's value.
 * 5. If any node violates this, the tree is not a BST.
 */

function TreeNode(val, left, right) {
    this.val = val;
    this.left = left || null;
    this.right = right || null;
}

function isValidBST(root) {
    const helper = (node, min, max) => {
        if (!node) return true; // Base case: empty node is valid
        if (node.val <= min || node.val >= max) return false; // BST property violation
        // Check left subtree with updated max, right subtree with updated min
        return helper(node.left, min, node.val) && helper(node.right, node.val, max);
    };
    return helper(root, -Infinity, Infinity);
}

// Helper function to create tree from array (level-order) for testing
function arrayToTree(arr) {
    if (!arr.length) return null;
    const root = new TreeNode(arr[0]);
    const queue = [root];
    let i = 1;
    while (i < arr.length) {
        const node = queue.shift();
        if (arr[i] !== null) {
            node.left = new TreeNode(arr[i]);
            queue.push(node.left);
        }
        i++;
        if (i < arr.length && arr[i] !== null) {
            node.right = new TreeNode(arr[i]);
            queue.push(node.right);
        }
        i++;
    }
    return root;
}

// Driver code to test isValidBST
const testCases = [
    { tree: [2,1,3], expected: true },
    { tree: [5,1,4,null,null,3,6], expected: false },
    { tree: [10,5,15,null,null,6,20], expected: false },
];

testCases.forEach(({tree, expected}, i) => {
    const root = arrayToTree(tree);
    const result = isValidBST(root);
    console.log(`Test Case ${i + 1}: Expected = ${expected}, Got = ${result}`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given a binary search tree (BST), find the kth smallest element in it.
 *
 * Note: A BST's in-order traversal produces elements in **ascending order**.
 *
 * Example:
 * ----------
 * Input: root = [3,1,4,null,2], k = 1
 * Output: 1
 *
 * Input: root = [5,3,6,2,4,null,null,1], k = 3
 * Output: 3
 *
 * Intuition:
 * ----------
 * 1. Perform an in-order traversal of the BST.
 * 2. Keep a counter to track how many nodes have been visited.
 * 3. When count reaches k, store the current node's value as the result.
 * 4. Stop traversal once the kth smallest element is found.
 */

function TreeNode(val, left, right) {
    this.val = val;
    this.left = left || null;
    this.right = right || null;
}

function kthSmallest(root, k) {
    let count = 0, result = null;

    // In-order traversal helper function
    const inorder = (node) => {
        if (!node || result !== null) return; // Stop if node is null or result found
        inorder(node.left);                   // Traverse left subtree
        count++;
        if (count === k) {                    // Check if current node is kth
            result = node.val;
            return;
        }
        inorder(node.right);                  // Traverse right subtree
    };

    inorder(root);
    return result;
}

// Helper function to create tree from array (level-order) for testing
function arrayToTree(arr) {
    if (!arr.length) return null;
    const root = new TreeNode(arr[0]);
    const queue = [root];
    let i = 1;
    while (i < arr.length) {
        const node = queue.shift();
        if (arr[i] !== null) {
            node.left = new TreeNode(arr[i]);
            queue.push(node.left);
        }
        i++;
        if (i < arr.length && arr[i] !== null) {
            node.right = new TreeNode(arr[i]);
            queue.push(node.right);
        }
        i++;
    }
    return root;
}

// Driver code to test kthSmallest
const testCases = [
    { tree: [3,1,4,null,2], k: 1, expected: 1 },
    { tree: [5,3,6,2,4,null,null,1], k: 3, expected: 3 },
    { tree: [5,3,6,2,4,null,null,1], k: 6, expected: 6 },
];

testCases.forEach(({tree, k, expected}, i) => {
    const root = arrayToTree(tree);
    const result = kthSmallest(root, k);
    console.log(`Test Case ${i + 1}: Expected = ${expected}, Got = ${result}`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given a binary tree, find its maximum depth.
 *
 * The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.
 *
 * Example:
 * ----------
 * Input:    3
 *          / \
 *         9  20
 *           / \
 *          15  7
 * Output: 3
 *
 * Input: null
 * Output: 0
 *
 * Intuition:
 * ----------
 * 1. Use recursion to traverse the tree.
 * 2. For each node, the depth is 1 + max(depth of left subtree, depth of right subtree).
 * 3. Base case: If the node is null, return 0.
 * 4. Recursively compute the depth for left and right subtrees.
 */

function TreeNode(val, left, right) {
    this.val = val;
    this.left = left || null;
    this.right = right || null;
}

function maxDepth(root) {
    if (!root) return 0; // Base case: empty tree has depth 0
    // Depth is 1 + maximum depth of left and right subtrees
    return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
}

// Helper function to create tree from array (level-order) for testing
function arrayToTree(arr) {
    if (!arr.length) return null;
    const root = new TreeNode(arr[0]);
    const queue = [root];
    let i = 1;
    while (i < arr.length) {
        const node = queue.shift();
        if (arr[i] !== null) {
            node.left = new TreeNode(arr[i]);
            queue.push(node.left);
        }
        i++;
        if (i < arr.length && arr[i] !== null) {
            node.right = new TreeNode(arr[i]);
            queue.push(node.right);
        }
        i++;
    }
    return root;
}

// Driver code to test maxDepth
const testCases = [
    { tree: [3,9,20,null,null,15,7], expected: 3 },
    { tree: [], expected: 0 },
    { tree: [1,null,2], expected: 2 },
];

testCases.forEach(({tree, expected}, i) => {
    const root = arrayToTree(tree);
    const result = maxDepth(root);
    console.log(`Test Case ${i + 1}: Expected = ${expected}, Got = ${result}`);
});
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * Given n balloons, each balloon has a number on it. You are asked to burst all the balloons.
 * If you burst balloon k, you get nums[i] * nums[k] * nums[j] coins where i and j are the
 * adjacent balloons left and right of k after bursting.
 * Find the maximum coins you can collect by bursting balloons wisely.
 * (LeetCode 312: Burst Balloons)
 *
 * Example:
 *   nums = [3,1,5,8] -> Expected result = 167
 *
 * Intuition:
 * ----------
 * Bursting balloons in order affects future choices. Instead of choosing which to burst first,
 * think in reverse: choose which balloon to burst **last** in a range.
 * Add two "virtual balloons" with value 1 on each end to simplify edge handling.
 * Use dynamic programming where dp[i][j] represents the maximum coins obtainable
 * by bursting all balloons between i and j (exclusive).
 *
 * Logic:
 * ------
 * 1. Pad nums with 1 at both ends: nums = [1, ...nums, 1].
 * 2. Let dp[i][j] = maximum coins from bursting balloons strictly between i and j.
 * 3. Iterate over subarray lengths (len = 2 to n):
 *    - For each (i, j) = (start, start+len):
 *      - Try all positions k in (i, j) as the last balloon to burst:
 *        dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + nums[i]*nums[k]*nums[j])
 * 4. Final answer is dp[0][n-1].
 */

function maxCoins(nums) {
    nums = [1, ...nums, 1];           // Pad with 1 on both ends
    const n = nums.length;
    const dp = Array.from({ length: n }, () => Array(n).fill(0));

    // len is the distance between i and j
    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] * nums[k] * nums[j]
                );
            }
        }
    }

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

// -------- Driver Code --------
function main() {
    const nums = [3,1,5,8];
    const result = maxCoins(nums);
    console.log(`Balloons: [${nums.join(', ')}]`);
    console.log(`Maximum coins: ${result}`); // Expected: 167
}

main();
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * Given an array of positive integers `nums`, determine if it can be partitioned into
 * two subsets whose sums are equal. This is the "Partition Equal Subset Sum" problem.
 *
 * Intuition:
 * ----------
 * If the total sum of `nums` is odd, it's impossible to split into two equal halves.
 * If even, check whether any subset sums to half the total. If such a subset exists,
 * the remaining numbers form the other half—making equal partitions possible.
 *
 * Logic:
 * ------
 * 1. Compute the total sum of `nums`. If odd, return false.
 * 2. Let `target = sum / 2`. Use a 1D boolean DP array `dp` of length `target + 1`.
 *    - `dp[i]` means: "Is a subset sum of `i` achievable?"
 * 3. Initialize `dp[0] = true` (sum of 0 is always achievable with no elements).
 * 4. For each `num` in `nums`, iterate `i` backward from `target` to `num`:
 *       dp[i] = dp[i] || dp[i - num];
 *    (Iterating backward ensures each number is only used once per iteration.)
 * 5. Return `dp[target]`.
 */

function canPartition(nums) {
    const sum = nums.reduce((a,b)=>a+b,0);
    if (sum % 2 !== 0) return false;      // Odd sum cannot be split equally
    const target = sum / 2;
    const dp = Array(target + 1).fill(false);
    dp[0] = true;                         // Base case: sum of 0 is always possible

    for (const num of nums) {
        for (let i = target; i >= num; i--) {
            dp[i] = dp[i] || dp[i - num];
        }
    }
    return dp[target];
}

// -------- Driver Code --------
function main() {
    const nums = [1, 5, 11, 5]; // Example input
    const result = canPartition(nums);
    console.log(`Can partition [${nums.join(', ')}]: ${result}`);
}

main();
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * Given three strings s1, s2, and s3, determine if s3 is formed by an interleaving
 * of s1 and s2. Interleaving means s3 can be formed by merging s1 and s2 in a way
 * that preserves the order of characters from each string.
 * (LeetCode 97: Interleaving String)
 *
 * Example:
 *   s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
 *   Output: true
 *   Explanation: s3 can be formed by taking characters from s1 and s2 in order.
 *
 * Intuition:
 * ----------
 * Use dynamic programming to decide whether prefixes of s1 and s2 can form a
 * prefix of s3. If a prefix of s3 ends with a character from s1 or s2,
 * check whether removing that character leaves a smaller interleaving problem
 * that is already solvable.
 *
 * Logic:
 * ------
 * 1. If lengths don't match (s1.length + s2.length !== s3.length), return false.
 * 2. Define a DP table dp[m+1][n+1] where dp[i][j] indicates whether the first i
 *    characters of s1 and first j characters of s2 can form the first i+j characters of s3.
 * 3. Base case: dp[0][0] = true (empty strings match).
 * 4. Fill the table:
 *    - If s1[i-1] == s3[i+j-1], inherit dp[i-1][j].
 *    - If s2[j-1] == s3[i+j-1], inherit dp[i][j-1].
 *    - Use logical OR because either choice may lead to a match.
 * 5. Return dp[m][n] as the final answer.
 */

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];
}

// -------- Driver Code --------
function main() {
    const s1 = "aabcc";
    const s2 = "dbbca";
    const s3 = "aadbbcbcac";
    const result = isInterleave(s1, s2, s3);
    console.log(`s1: "${s1}", s2: "${s2}", s3: "${s3}"`);
    console.log(`Is interleaving: ${result}`); // Expected: true
}

main();
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * Given an m x n integer matrix, return the length of the longest strictly increasing path.
 * You can move in four directions: up, down, left, or right.
 * (LeetCode 329: Longest Increasing Path in a Matrix)
 *
 * Example:
 *   matrix = [
 *     [9,9,4],
 *     [6,6,8],
 *     [2,1,1]
 *   ]
 *   Output: 4
 *   Explanation: The longest path is [1, 2, 6, 9].
 *
 * Intuition:
 * ----------
 * Use DFS to explore all paths starting from each cell, but cache results (memoization)
 * to avoid recomputing for overlapping subproblems.
 * If a neighbor has a greater value, continue DFS to extend the path.
 * Keep track of the longest path found so far.
 *
 * Logic:
 * ------
 * 1. Create a memo table `memo[m][n]` initialized to 0.
 * 2. Define directions `dirs` for movement in 4 directions.
 * 3. DFS(x, y):
 *    - If memo[x][y] > 0, return cached value.
 *    - Initialize maxLen = 1 (the cell itself).
 *    - For each neighbor (nx, ny):
 *        a. Check bounds and ensure matrix[nx][ny] > matrix[x][y].
 *        b. Recurse: maxLen = max(maxLen, 1 + dfs(nx, ny)).
 *    - Cache and return maxLen.
 * 4. Loop through all cells, compute dfs(i, j), and track the global maximum.
 * 5. Return the global maximum as the result.
 */

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]; // Return cached result
        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]) {
                maxLen = Math.max(maxLen, 1 + dfs(nx, ny));
            }
        }
        memo[x][y] = maxLen; // Cache result
        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;
}

// -------- Driver Code --------
function main() {
    const matrix = [
        [9, 9, 4],
        [6, 6, 8],
        [2, 1, 1]
    ];
    const result = longestIncreasingPath(matrix);
    console.log("Matrix:");
    console.log(matrix.map(row => row.join(' ')).join('\n'));
    console.log(`Longest Increasing Path Length: ${result}`); // Expected: 4
}

main();
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * Implement regular expression matching with support for '.' and '*'.
 * - '.' Matches any single character.
 * - '*' Matches zero or more of the preceding element.
 * Given a string s and pattern p, return true if p matches the entire string s.
 * (LeetCode 10: Regular Expression Matching)
 *
 * Example:
 *   s = "aab", p = "c*a*b" -> true
 *   Explanation: c* can match empty, a* matches "aa", and b matches "b".
 *
 * Intuition:
 * ----------
 * Use Dynamic Programming to build a table dp[i][j] indicating if the first i characters
 * of s match the first j characters of p.
 * - Direct matches or '.' consume one character from both s and p.
 * - '*' can eliminate the preceding character (match zero times) or extend a match
 *   if the preceding character matches the current character in s.
 *
 * Logic:
 * ------
 * 1. Initialize dp[m+1][n+1] with false; dp[0][0] = true (empty string matches empty pattern).
 * 2. Pre-fill dp[0][j] for patterns like a*, a*b*, etc. where '*' can eliminate pairs.
 * 3. Iterate i from 1..m and j from 1..n:
 *    - If p[j-1] matches s[i-1] or is '.', set dp[i][j] = dp[i-1][j-1].
 *    - If p[j-1] is '*', two possibilities:
 *        a. dp[i][j-2] (zero occurrences of preceding char).
 *        b. (p[j-2] matches s[i-1] or '.') && dp[i-1][j] (extend match).
 * 4. Return dp[m][n] as the result.
 */

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 with '*' that can match empty prefixes (e.g., a*, a*b*, etc.)
    for (let j = 1; j <= n; j++) {
        if (p[j - 1] === '*') {
            dp[0][j] = dp[0][j - 2];
        }
    }

    // Build the DP table
    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] || // '*' matches zero of the preceding element
                    ((p[j - 2] === s[i - 1] || p[j - 2] === '.') && dp[i - 1][j]);
            }
        }
    }

    return dp[m][n];
}

// -------- Driver Code --------
function main() {
    const s = "aab";
    const p = "c*a*b";
    const result = isMatch(s, p);
    console.log(`String: "${s}", Pattern: "${p}"`);
    console.log(`Matches: ${result}`); // Expected: true
}

main();
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * A message containing letters A-Z is encoded into numbers using the mapping:
 *     'A' -> "1", 'B' -> "2", ..., 'Z' -> "26".
 * Given a string s containing only digits, return the number of ways to decode it.
 * (LeetCode 91: Decode Ways)
 *
 * Example:
 *   s = "226" -> Output: 3
 *   Explanation: "2 2 6" -> "BBF", "22 6" -> "VF", "2 26" -> "BZ"
 *
 * Intuition:
 * ----------
 * Dynamic Programming is ideal because the number of decodings for a prefix
 * depends on its previous states:
 * - Single-digit decode: If the current digit is not '0', add ways from dp[i-1].
 * - Double-digit decode: If the two-digit number is between 10 and 26, add ways from dp[i-2].
 *
 * Logic:
 * ------
 * 1. Let dp[i] = number of ways to decode substring s[0..i-1].
 * 2. Initialize:
 *     - dp[0] = 1 (empty string has one way).
 * 3. For each position i (1..n):
 *     - If s[i-1] != '0', then dp[i] += dp[i-1] (valid single-digit decode).
 *     - If two-digit number s[i-2..i-1] ∈ [10,26], then dp[i] += dp[i-2].
 * 4. Return dp[n] (total ways to decode entire string).
 */

function numDecodings(s) {
    const n = s.length;
    const dp = Array(n + 1).fill(0);
    dp[0] = 1; // Base case: empty string has one decoding

    for (let i = 1; i <= n; i++) {
        // Single-digit decode
        if (s[i - 1] !== '0') dp[i] += dp[i - 1];

        // Double-digit decode
        if (i > 1) {
            const twoDigit = parseInt(s.slice(i - 2, i), 10);
            if (twoDigit >= 10 && twoDigit <= 26) {
                dp[i] += dp[i - 2];
            }
        }
    }
    return dp[n];
}

// -------- Driver Code --------
function main() {
    const s = "226";
    const result = numDecodings(s);
    console.log(`String: "${s}"`);
    console.log(`Number of decodings: ${result}`); // Expected: 3
}

main();
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * A robot is located at the top-left corner of an m x n grid.
 * The robot can only move either down or right at any point.
 * The robot tries to reach the bottom-right corner of the grid.
 * Return the total number of unique paths possible.
 *
 * Example:
 *   m = 3, n = 7
 *   Output: 28
 *
 * Intuition:
 * ----------
 * - Recursive: From a cell (i,j), the robot can come either from (i-1,j) or (i,j-1).
 *   The total paths to (i,j) is the sum of paths to its top and left neighbors.
 * - Dynamic Programming: Use a table to build solutions bottom-up and avoid
 *   recalculating overlapping subproblems.
 *
 * Approach 1: Recursive (Top-Down)
 * --------------------------------
 * Base cases:
 *   - If either m == 1 or n == 1, there's only 1 path (all the way right or down).
 * Recurrence:
 *   - uniquePathsRec(m, n) = uniquePathsRec(m-1, n) + uniquePathsRec(m, n-1)
 * Complexity:
 *   - Time: O(2^(m+n)) without memoization (inefficient for large grids).
 *   - Space: O(m+n) recursion depth.
 */

function uniquePathsRec(m, n) {
    if (m === 1 || n === 1) return 1;
    return uniquePathsRec(m - 1, n) + uniquePathsRec(m, n - 1);
}

/**
 * Approach 2: Dynamic Programming (Bottom-Up)
 * -------------------------------------------
 * Build a DP table where dp[i][j] = dp[i-1][j] + dp[i][j-1].
 * Base initialization:
 *   - First row and first column are all 1 (only one way along edges).
 * Complexity:
 *   - Time: O(m * n)
 *   - Space: O(m * n)
 */

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];
}

// -------- Driver Code --------
function main() {
    const m = 3, n = 7;

    const resultDP = uniquePaths(m, n);
    const resultRec = uniquePathsRec(m, n);

    console.log(`Grid size: ${m} x ${n}`);
    console.log(`Unique paths (DP): ${resultDP}`);  // Expected: 28
    console.log(`Unique paths (Recursive): ${resultRec}`); // Expected: 28
}

main();
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * There are n cities connected by flights. flights[i] = [u, v, w] means there is a flight
 * from city u to city v with cost w. Given the number of cities n, the list of flights,
 * a source city src, a destination city dst, and an integer K (maximum allowed stops),
 * return the cheapest price from src to dst with at most K stops. If there is no such route, return -1.
 * (LeetCode 787: Cheapest Flights Within K Stops)
 *
 * Example:
 *   n = 4, flights = [[0,1,100],[1,2,100],[2,3,100],[0,2,500]],
 *   src = 0, dst = 3, K = 1
 *   Output: 600
 *
 * Intuition:
 * ----------
 * Perform a BFS-like traversal where each layer represents an additional stop. Track:
 * - current city,
 * - accumulated cost,
 * - number of stops so far.
 * Always expand paths only if they do not exceed K stops.
 * Keep track of the minimum cost found to reach dst.
 *
 * Logic:
 * ------
 * 1. Build an adjacency list `adj` for the graph from flights.
 * 2. Use a queue for BFS: each element = [node, cost, stops].
 * 3. Initialize queue with [src, 0, 0] (starting at src, cost 0, stops 0).
 * 4. While the queue is not empty:
 *    - Pop front: [node, cost, stops].
 *    - If node === dst, update res = min(res, cost).
 *    - If stops > K, skip further expansion.
 *    - For each neighbor nei with weight w, push [nei, cost+w, stops+1].
 * 5. Return -1 if res was never updated; else return res.
 *
 * Complexity:
 * -----------
 * - Time: O(E) in worst case (edges processed multiple times depending on stops).
 * - Space: O(V + E) for adjacency list and queue.
 */

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]]; // [current node, total cost, stops used]
    let 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;
}

// -------- Driver Code --------
function main() {
    const n = 4;
    const flights = [[0,1,100],[1,2,100],[2,3,100],[0,2,500]];
    const src = 0, dst = 3, K = 1;

    const result = findCheapestPrice(n, flights, src, dst, K);
    console.log(`Flights: ${JSON.stringify(flights)}`);
    console.log(`Cheapest price from ${src} to ${dst} with at most ${K} stops: ${result}`);
    // Expected: 600
}

main();
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * You are given n points on a 2D plane where points[i] = [xi, yi].
 * The cost to connect two points is their Manhattan distance:
 *    dist(i, j) = |xi - xj| + |yi - yj|.
 * Return the minimum cost to connect all points so that there is exactly one simple path
 * between any two points.
 * (LeetCode 1584: Min Cost to Connect All Points)
 *
 * Example:
 *   points = [[0,0],[2,2],[3,10],[5,2],[7,0]]
 *   Output: 20
 *
 * Intuition:
 * ----------
 * This is a classic **Minimum Spanning Tree (MST)** problem on a complete graph.
 * Use **Prim's algorithm**:
 * - Start from any point (node).
 * - Greedily pick the smallest edge connecting the visited set to an unvisited point.
 * - Repeat until all points are connected.
 *
 * Logic:
 * ------
 * 1. Initialize:
 *    - visited array of length n.
 *    - Min-heap priority queue with [0,0] (cost,node) to start.
 *    - res = 0 (total cost), count = 0 (visited nodes count).
 * 2. While count < n:
 *    - Pop the smallest cost edge from the heap.
 *    - If the node is already visited, skip.
 *    - Mark it visited, add its cost to res, increment count.
 *    - For each unvisited neighbor, push [dist(u,v), v] to the heap.
 * 3. Return res.
 *
 * Complexity:
 * -----------
 * - Time: O(n² log n) since for each node we push up to n edges into a min-heap.
 * - Space: O(n²) for the heap in the worst case.
 */

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) {
        // Extract smallest edge from min-heap
        heap.sort((a, b) => a[0] - b[0]);
        const [cost, u] = heap.shift();

        if (visited[u]) continue; // Skip if already visited
        visited[u] = true;
        res += cost;
        count++;

        // Add edges from current node to all unvisited nodes
        for (let v = 0; v < n; v++) {
            if (!visited[v]) {
                heap.push([dist(u, v), v]);
            }
        }
    }

    return res;
}

// -------- Driver Code --------
function main() {
    const points = [[0,0],[2,2],[3,10],[5,2],[7,0]];
    const result = minCostConnectPoints(points);
    console.log("Points:", points.map(p => `[${p}]`).join(', '));
    console.log(`Minimum cost to connect all points: ${result}`); // Expected: 20
}

main();
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * In a connected undirected graph with n nodes labeled from 1 to n,
 * there is exactly one extra edge that creates a cycle.
 *
 * Given a list of edges where each edge = [u, v], return the edge that can be removed to make the graph a tree.
 * If there are multiple answers, return the one that appears last in the input.
 *
 * Example:
 * ----------
 * Input: edges = [[1,2],[1,3],[2,3]]
 * Output: [2,3]
 *
 * Input: edges = [[1,2],[2,3],[3,4],[1,4],[1,5]]
 * Output: [1,4]
 *
 * Intuition:
 * ----------
 * 1. Use Union-Find (Disjoint Set Union, DSU) to detect cycles efficiently.
 * 2. Initialize each node as its own parent.
 * 3. For each edge [u, v]:
 *    - If u and v have the same root, adding this edge creates a cycle → it's redundant.
 *    - Otherwise, union u and v.
 * 4. Return the first edge detected that forms a cycle.
 */

function findRedundantConnection(edges) {
    const parent = [];

    // Find with path compression
    const find = (x) => {
        if (parent[x] === x) return x;
        parent[x] = find(parent[x]);
        return parent[x];
    };

    // Union two sets; return false if they are already connected
    const union = (x, y) => {
        const px = find(x), py = find(y);
        if (px === py) return false; // Cycle detected
        parent[px] = py;
        return true;
    };

    // Initialize parent array (1-based indexing)
    for (let i = 0; i <= edges.length; i++) parent[i] = i;

    // Process all edges
    for (const [u, v] of edges) {
        if (!union(u, v)) return [u, v]; // Redundant edge detected
    }
}

// Driver code to test findRedundantConnection
const testCases = [
    { edges: [[1,2],[1,3],[2,3]], expected: [2,3] },
    { edges: [[1,2],[2,3],[3,4],[1,4],[1,5]], expected: [1,4] },
    { edges: [[1,2],[2,3],[3,1]], expected: [3,1] }
];

testCases.forEach(({edges, expected}, i) => {
    const result = findRedundantConnection(edges);
    console.log(`Test Case ${i + 1}: Expected = [${expected}], Got = [${result}]`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.
 *
 * A region is captured by flipping all 'O's into 'X's in that surrounded region.
 * An 'O' on the border, or connected to a border 'O', is not surrounded.
 *
 * Example:
 * ----------
 * Input:
 * [
 *   ['X','X','X','X'],
 *   ['X','O','O','X'],
 *   ['X','X','O','X'],
 *   ['X','O','X','X']
 * ]
 * Output:
 * [
 *   ['X','X','X','X'],
 *   ['X','X','X','X'],
 *   ['X','X','X','X'],
 *   ['X','O','X','X']
 * ]
 *
 * Intuition:
 * ----------
 * 1. All 'O's that are connected to the border cannot be captured.
 * 2. Use DFS to mark all border-connected 'O's with a temporary marker (e.g., 'T').
 * 3. Traverse the entire board:
 *    - Convert 'T' back to 'O' (safe).
 *    - Convert remaining 'O's to 'X' (captured).
 */

function solveSurroundedRegions(board) {
    const m = board.length, n = board[0].length;

    // DFS to mark connected 'O's starting from border
    const dfs = (i, j) => {
        if (i < 0 || j < 0 || i >= m || j >= n || board[i][j] !== 'O') return;
        board[i][j] = 'T'; // Mark as temporary safe
        dfs(i + 1, j);
        dfs(i - 1, j);
        dfs(i, j + 1);
        dfs(i, j - 1);
    };

    // Start DFS from all border cells
    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);
    }

    // Flip captured 'O's to 'X' and revert temporary marks to 'O'
    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            board[i][j] = board[i][j] === 'T' ? 'O' : 'X';
        }
    }
}

// Driver code to test solveSurroundedRegions
const board = [
    ['X','X','X','X'],
    ['X','O','O','X'],
    ['X','X','O','X'],
    ['X','O','X','X']
];

console.log("Before:");
board.forEach(row => console.log(row.join(' ')));

solveSurroundedRegions(board);

console.log("After:");
board.forEach(row => console.log(row.join(' ')));
// Expected Output:
// X X X X
// X X X X
// X X X X
// X O X X
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * You are given an n x n grid where the value grid[i][j] indicates the elevation at that point.
 * Water rises over time t. At time t, you can enter any square with elevation ≤ t.
 * Starting at the top-left corner (0,0), return the minimum time t required
 * to reach the bottom-right corner (n-1,n-1).
 * (LeetCode 778: Swim in Rising Water)
 *
 * Example:
 *   grid = [
 *     [0,2],
 *     [1,3]
 *   ]
 *   Output: 3
 *
 * Intuition:
 * ----------
 * Model the grid as a weighted graph where each cell is a node and edges represent
 * movement to adjacent cells. The "weight" of a path is determined by the highest
 * elevation encountered so far. Use a **Dijkstra-like** approach:
 * - Always expand the lowest-elevation option first (min-heap).
 * - Update the maximum elevation encountered on the path.
 *
 * Logic:
 * ------
 * 1. Initialize a min-heap with [grid[0][0], 0, 0] (elevation, x, y) and a visited matrix.
 * 2. Pop the lowest elevation cell from the heap:
 *    - If it's the bottom-right cell, return its elevation as the minimum time.
 *    - Mark it visited.
 *    - For each neighbor (up, down, left, right):
 *        * If not visited, push into the heap with elevation = max(current path elevation, neighbor's value).
 * 3. Continue until destination is reached.
 *
 * Complexity:
 * -----------
 * Time: O(n² log n²) = O(n² log n) due to sorting heap operations.
 * Space: O(n²) for visited and heap.
 */

function swimInWater(grid) {
    const n = grid.length;
    const visited = Array.from({ length: n }, () => Array(n).fill(false));
    const heap = new MinHeap([grid[0][0], 0, 0]); // [elevation, x, y]
    const dirs = [[1, 0], [0, 1], [-1, 0], [0, -1]];

    while (heap.length) {
        // Pop the cell with the lowest elevation (simulated min-heap using sort)
        heap.sort((a, b) => a[0] - b[0]);
        const [h, x, y] = heap.shift();

        if (x === n - 1 && y === n - 1) return h; // Reached destination

        if (visited[x][y]) continue;
        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]);
            }
        }
    }
}

// -------- Driver Code --------
function main() {
    const grid = [
        [0, 2],
        [1, 3]
    ];
    const result = swimInWater(grid);
    console.log("Grid:");
    console.log(grid.map(row => row.join(' ')).join('\n'));
    console.log(`Minimum time to reach destination: ${result}`); // Expected: 3
}

main();
```

```
/*#*/
/**
 * Union-Find (Disjoint Set Union - DSU)
 * -------------------------------------
 * This is a data structure to efficiently keep track of a set of elements
 * partitioned into disjoint (non-overlapping) subsets.
 *
 * Key Operations:
 *  - find(x): Find the representative (root) of the set containing x.
 *  - union(x, y): Merge the sets containing x and y.
 *  - connected(x, y): Check if x and y belong to the same set.
 *
 * Use Cases:
 *  - Detecting cycles in an undirected graph.
 *  - Kruskal's Minimum Spanning Tree algorithm.
 *  - Counting connected components in a graph.
 *  - Network connectivity and clustering problems.
 *
 * Optimizations:
 *  - Path Compression: Flattens the tree structure during find(),
 *    making future operations faster.
 *  - Union by Rank (or Size): Always attach the smaller tree to the root
 *    of the larger tree to avoid tall trees.
 *
 * Time Complexity:
 *  - With optimizations, both find() and union() run in nearly O(1)
 *    (specifically O(α(n)), where α is the inverse Ackermann function).
 */

class UnionFind {
    constructor(n) {
        // Initially, each element is its own parent (self root)
        this.parent = Array.from({ length: n }, (_, i) => i);
        // Rank array stores tree height approximation for union-by-rank
        this.rank = Array(n).fill(1);
    }

    // Find the representative of set x with path compression
    find(x) {
        if (this.parent[x] !== x) {
            // Recursively find the root and compress path
            this.parent[x] = this.find(this.parent[x]);
        }
        return this.parent[x];
    }

    // Union two sets using union by rank
    union(x, y) {
        const rootX = this.find(x);
        const rootY = this.find(y);

        if (rootX === rootY) return false; // Already in the same set

        // Attach the smaller tree under the larger one
        if (this.rank[rootX] > this.rank[rootY]) {
            this.parent[rootY] = rootX;
        } else if (this.rank[rootX] < this.rank[rootY]) {
            this.parent[rootX] = rootY;
        } else {
            this.parent[rootY] = rootX;
            this.rank[rootX] += 1;
        }
        return true;
    }

    // Check if two elements are in the same set
    connected(x, y) {
        return this.find(x) === this.find(y);
    }
}

// -------- Driver Code Example --------
function main() {
    const uf = new UnionFind(5); // Elements: 0,1,2,3,4

    uf.union(0, 1); // Merge sets containing 0 and 1
    uf.union(1, 2); // Merge sets containing 1 and 2
    console.log(uf.connected(0, 2)); // true (0 and 2 are connected)
    console.log(uf.connected(0, 3)); // false (0 and 3 are not connected)

    uf.union(3, 4); // Merge sets containing 3 and 4
    console.log(uf.connected(3, 4)); // true

    uf.union(2, 3); // Merge sets containing 2 and 3
    console.log(uf.connected(0, 4)); // true (all 0-4 are now connected)
}

main();
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given n nodes labeled from 0 to n-1 and a list of edges,
 * determine if these edges make up a valid tree.
 *
 * A valid tree must satisfy:
 * 1. It is connected (all nodes are reachable).
 * 2. It contains no cycles.
 *
 * Example:
 * ----------
 * Input: n = 5, edges = [[0,1],[0,2],[0,3],[1,4]]
 * Output: true
 *
 * Input: n = 5, edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]
 * Output: false
 *
 * Intuition:
 * ----------
 * 1. For a tree with n nodes, there must be exactly n-1 edges.
 * 2. Use Union-Find to detect cycles efficiently:
 *    - Initialize each node as its own parent.
 *    - For each edge [u,v], union them.
 *    - If u and v are already connected, there is a cycle → not a tree.
 * 3. If edges.length !== n-1 → not connected or too many edges → not a tree.
 */

function validTree(n, edges) {
    // A tree must have exactly n-1 edges
    if (edges.length !== n - 1) return false;

    // Initialize Union-Find
    const parent = Array.from({ length: n }, (_, i) => i);

    // Find with path compression
    const find = x => {
        if (parent[x] === x) return x;
        parent[x] = find(parent[x]);
        return parent[x];
    };

    // Union edges
    for (const [u, v] of edges) {
        const pu = find(u), pv = find(v);
        if (pu === pv) return false; // Cycle detected
        parent[pu] = pv;             // Union
    }

    return true; // No cycles and n-1 edges → valid tree
}

// Driver code to test validTree
const testCases = [
    { n: 5, edges: [[0,1],[0,2],[0,3],[1,4]], expected: true },
    { n: 5, edges: [[0,1],[1,2],[2,3],[1,3],[1,4]], expected: false },
    { n: 4, edges: [[0,1],[2,3]], expected: false }
];

testCases.forEach(({n, edges, expected}, i) => {
    const result = validTree(n, edges);
    console.log(`Test Case ${i + 1}: Expected = ${expected}, Got = ${result}`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given a 2D board and a word, check if the word exists in the grid.
 *
 * - The word can be constructed from letters of sequentially adjacent cells,
 *   where "adjacent" cells are horizontally or vertically neighboring.
 * - The same letter cell may not be used more than once.
 *
 * Example:
 * ----------
 * Input:
 * board = [
 *   ['A','B','C','E'],
 *   ['S','F','C','S'],
 *   ['A','D','E','E']
 * ], word = "ABCCED"
 * Output: true
 *
 * Input: word = "SEE"
 * Output: true
 *
 * Input: word = "ABCB"
 * Output: false
 *
 * Intuition:
 * ----------
 * 1. Perform DFS starting from each cell in the board.
 * 2. At each step, check if the current cell matches the corresponding character in the word.
 * 3. Mark the cell as visited (e.g., with a temporary marker) to avoid revisiting.
 * 4. Explore all four directions (up, down, left, right).
 * 5. If we match all characters, return true; otherwise backtrack and continue.
 */

function existWord(board, word) {
    const m = board.length, n = board[0].length;

    // DFS helper function
    const dfs = (i, j, idx) => {
        // If all characters are matched
        if (idx === word.length) return true;

        // Check boundaries and character match
        if (i < 0 || j < 0 || i >= m || j >= n || board[i][j] !== word[idx]) return false;

        // Mark current cell as visited
        const tmp = board[i][j];
        board[i][j] = '#';

        // Explore all four directions
        const res = dfs(i+1, j, idx+1) || dfs(i-1, j, idx+1) ||
                    dfs(i, j+1, idx+1) || dfs(i, j-1, idx+1);

        // Backtrack: restore original character
        board[i][j] = tmp;
        return res;
    };

    // Start DFS from every cell
    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            if (dfs(i, j, 0)) return true;
        }
    }

    return false;
}

// Driver code to test existWord
const board = [
    ['A','B','C','E'],
    ['S','F','C','S'],
    ['A','D','E','E']
];

const testCases = [
    { word: "ABCCED", expected: true },
    { word: "SEE", expected: true },
    { word: "ABCB", expected: false }
];

testCases.forEach(({word, expected}, i) => {
    const result = existWord(board.map(row => [...row]), word); // copy board for each test
    console.log(`Test Case ${i + 1}: Word = "${word}", Expected = ${expected}, Got = ${result}`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given a 2D board of letters and a list of words, find all words from the list that exist in the board.
 *
 * - Words can be constructed from letters of sequentially adjacent cells (horizontally or vertically).
 * - The same letter cell may not be used more than once in a word.
 *
 * Example:
 * ----------
 * Input:
 * board = [
 *   ['o','a','a','n'],
 *   ['e','t','a','e'],
 *   ['i','h','k','r'],
 *   ['i','f','l','v']
 * ], words = ["oath","pea","eat","rain"]
 * Output: ["oath","eat"]
 *
 * Intuition:
 * ----------
 * 1. Build a Trie from the given word list for fast prefix matching.
 * 2. Perform DFS from each cell in the board.
 * 3. At each step, check if the current path is a valid prefix in the Trie.
 * 4. If a word is complete in the Trie, add it to the result set.
 * 5. Use backtracking to mark visited cells temporarily.
 */

// Trie Node
class TrieNode {
    constructor() {
        this.children = {};  // Map of character -> TrieNode
        this.endOfWord = false;
    }
}

// Trie
class Trie {
    constructor() {
        this.root = new TrieNode();
    }

    insert(word) {
        let node = this.root;
        for (const char of word) {
            if (!node.children[char]) node.children[char] = new TrieNode();
            node = node.children[char];
        }
        node.endOfWord = true;
    }
}

// Main function to find words
function findWords(board, words) {
    const res = new Set();
    const trie = new Trie();

    // Insert all words into Trie
    for (const w of words) trie.insert(w);

    const m = board.length, n = board[0].length;

    const dfs = (i, j, node, path) => {
        if (!node) return;

        if (node.endOfWord) res.add(path);

        const c = board[i][j];
        board[i][j] = '#'; // mark as visited

        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 && x < m && y < n && node.children[board[x][y]]) {
                dfs(x, y, node.children[board[x][y]], path + board[x][y]);
            }
        }

        board[i][j] = c; // backtrack
    };

    // Start DFS from each cell
    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            if (trie.root.children[board[i][j]]) {
                dfs(i, j, trie.root.children[board[i][j]], board[i][j]);
            }
        }
    }

    return Array.from(res); // Convert set to array
}

// Driver code to test findWords
const board = [
    ['o','a','a','n'],
    ['e','t','a','e'],
    ['i','h','k','r'],
    ['i','f','l','v']
];
const words = ["oath","pea","eat","rain"];
const result = findWords(board, words);
console.log("Words found:", result); // Expected: ["oath","eat"]
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * Given an array `nums` where each element represents the maximum jump length at that position,
 * return the minimum number of jumps needed to reach the last index.
 * (LeetCode 45: Jump Game II)
 *
 * Example:
 *   nums = [2,3,1,1,4]
 *   Minimum jumps to reach end = 2  (2 -> 3 -> 4)
 *
 * Intuition:
 * ----------
 * Greedily track the furthest position we can reach while scanning.
 * When we reach the end of the current "jump range", increment the jump count
 * and update the range to the furthest we can reach so far.
 *
 * Logic:
 * ------
 * 1. Initialize:
 *    - jumps = 0 (number of jumps taken)
 *    - curEnd = 0 (end of the current jump range)
 *    - curFarthest = 0 (farthest position reachable within the current range)
 * 2. Traverse `nums` up to the second-to-last element:
 *    - Update curFarthest = max(curFarthest, i + nums[i]).
 *    - If i == curEnd, we've reached the end of the current jump range:
 *        a. Increment jumps.
 *        b. Update curEnd = curFarthest (start a new jump range).
 * 3. Return jumps at the end.
 */

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;
}

// -------- Driver Code --------
function main() {
    const nums = [2,3,1,1,4];
    const result = jumps2(nums);
    console.log(`Array: [${nums.join(', ')}]`);
    console.log(`Minimum jumps to reach the end: ${result}`); // Expected: 2
}

main();
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * Given a string S of lowercase letters, partition the string into as many parts
 * as possible so that each letter appears in at most one part.
 * Return a list of integers representing the size of these parts.
 *
 * Intuition:
 * ----------
 * Each character's last occurrence determines how far we must extend a partition
 * to include all instances of that character. By tracking the furthest last index
 * encountered while scanning, we know the earliest point we can safely cut.
 *
 * Logic:
 * ------
 * 1. Traverse S once to record the last index of every character in an object `last`.
 * 2. Initialize `start = 0` and `end = 0`.
 * 3. Traverse S again:
 *    - Update `end` to the furthest last index seen for the current character.
 *    - If the current index `i` equals `end`, it means all characters in this partition
 *      end by `end`. Append the partition size `(end - start + 1)` to the result array,
 *      and set `start = i + 1` for the next partition.
 * 4. Return the result array.
 */

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;
}

// -------- Driver Code --------
function main() {
    const S = "ababcbacadefegdehijhklij";
    const result = partitionLabels(S);
    console.log(`Input: "${S}"`);
    console.log(`Partition Sizes: ${result}`); // Expected: [9,7,8]
}

main();
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * Given an array of positive integers `stones`, each value represents the weight of a stone.
 * Each turn, you pick the two heaviest stones (x <= y), smash them together:
 *  - If x === y, both are destroyed.
 *  - If x !== y, the smaller one is destroyed, and the larger one is reduced by x (new weight = y - x),
 *    then put the new stone back into the pile.
 * Repeat until at most one stone remains. Return the weight of the remaining stone, or 0 if none remain.
 *
 * Intuition:
 * ----------
 * To simulate the smashing process, we need quick access to the two heaviest stones each time.
 * Sorting the array descendingly allows us to take the largest stones from the front.
 * After smashing, the resulting stone (if any) must be reinserted into the sorted array to keep the
 * heaviest stones at the front for the next iteration.
 *
 * Logic:
 * ------
 * 1. Sort stones in descending order.
 * 2. While more than one stone remains:
 *    a. Remove the two largest stones (front of the array).
 *    b. If their weights differ, compute their difference.
 *    c. Insert this new stone back into the array at the correct position to keep it sorted.
 * 3. If one stone remains, return its weight; otherwise, return 0.
 */
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;
}

// Top comment explains the algorithm above (optional)

// Example driver code
function main() {
  const testCases = [
    [2,7,4,1,8,1],  // Example: Expected output = 1
    [1],            // Only one stone: Expected output = 1
    [3,3],          // Two equal stones: Expected output = 0
    [9,3,2,10],     // Mixed weights: Expected output = 0
    [5,5,6,7],      // Expected output = 1
  ];

  testCases.forEach((stones, idx) => {
    const result = lastStoneWeight([...stones]); // copy array to avoid mutation
    console.log(`Test Case ${idx + 1}: Stones = [${stones.join(', ')}] -> Remaining Weight = ${result}`);
  });
}

main();
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * Given an array of integers `hand` representing cards and an integer `W` representing group size,
 * determine if the cards can be rearranged into groups of `W` consecutive cards.
 * Each card can only be used once. Return true if possible, else false.
 *
 * Example:
 *   hand = [1,2,3,6,2,3,4,7,8], W = 3
 *   Possible grouping: [1,2,3], [2,3,4], [6,7,8] -> returns true.
 *
 * Intuition:
 * ----------
 * Sort the unique card values. For each smallest card still available, attempt to build a group of W
 * consecutive cards using it as the start. Reduce their frequencies accordingly.
 * If at any point we don't have enough cards to form a group, return false.
 *
 * Logic:
 * ------
 * 1. If the total number of cards is not divisible by W, it's impossible — return false.
 * 2. Count the frequency of each card using a map.
 * 3. Sort the unique card values ascending.
 * 4. For each card value `k`:
 *    - Let `freq` = how many of card `k` remain.
 *    - If `freq > 0`, try to form `freq` groups starting at `k`:
 *      a. For i in [0..W-1], check if `count[k+i]` >= freq.
 *      b. If any count is too small, return false.
 *      c. Otherwise, subtract `freq` from `count[k+i]`.
 * 5. If all groups are formed successfully, return true.
 */

function isNStraightHand(hand, W) {
    if (hand.length % W !== 0) return false; // must be divisible to form equal groups
    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) {
            // Try forming freq groups starting from k
            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;
}

// -------- Driver Code --------
function main() {
    const hand = [1,2,3,6,2,3,4,7,8];
    const W = 3;
    const result = isNStraightHand(hand, W);
    console.log(`Hand: [${hand.join(', ')}], W = ${W}`);
    console.log(`Can be rearranged: ${result}`); // Expected: true
}

main();
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * You are given an array of triplets `triples` (each triple = [a,b,c])
 * and a `target` triple = [x,y,z].
 * You can "merge" triplets by choosing values from any triple at a position:
 * - The merged triple's i-th value is chosen from any triple's i-th value,
 *   but only if all selected triples' values do not exceed the target at that position.
 * Return true if you can form the `target` triple exactly by merging some of the triplets.
 *
 * Intuition:
 * ----------
 * Ignore any triple that exceeds the target in any position, since they can't contribute.
 * Track the largest b and c values among the valid triples (we don't need to track 'a'
 * because to match target[0], one valid triple must have a == target[0] already).
 * If, after processing all triples, the maximum b and c equal the target's b and c,
 * and some triple also matched target[0], then the target is achievable.
 *
 * Logic:
 * ------
 * 1. Initialize maxB = 0 and maxC = 0.
 * 2. For each triple [a,b,c]:
 *    - If any value exceeds the target in that position, skip it.
 *    - Otherwise, update maxB and maxC if b or c are larger than current maxima.
 * 3. After processing:
 *    - Return true if maxB == target[1] and maxC == target[2].
 *    - 'a' must be satisfied implicitly by having a triple where a == target[0].
 */

function mergeTriplets(triples, target) {
    let maxB = 0, maxC = 0;
    for (const [a, b, c] of triples) {
        // Only consider triples that don't exceed target in any dimension
        if (a <= target[0] && b <= target[1] && c <= target[2]) {
            if (b > maxB) maxB = b;  // track largest b
            if (c > maxC) maxC = c;  // track largest c
        }
    }
    // To form target, we must match target's b and c
    return maxB === target[1] && maxC === target[2];
}

// -------- Driver Code --------
function main() {
    const triples = [[2,5,3],[1,8,4],[1,7,5],[2,3,5]];
    const target = [2,7,5];
    const result = mergeTriplets(triples, target);
    console.log(`Triples: ${JSON.stringify(triples)}`);
    console.log(`Target: ${JSON.stringify(target)}`);
    console.log(`Can merge to target: ${result}`); // Expected: true
}

main();
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Design a class to find the kth largest element in a stream of numbers.
 *
 * - Implement the KthLargest class:
 *   - KthLargest(int k, int[] nums) Initializes the object with the integer k and the stream of integers nums.
 *   - int add(int val) Appends the integer val to the stream and returns the kth largest element in the stream.
 *
 * Example:
 * ----------
 * Input:
 * ["KthLargest","add","add","add","add","add"]
 * [[3,[4,5,8,2]],[3],[5],[10],[9],[4]]
 * Output:
 * [null,4,5,5,8,8]
 *
 * Intuition:
 * ----------
 * 1. Keep a sorted array of the k largest elements seen so far.
 * 2. When adding a new element:
 *    - Insert it into the array.
 *    - Sort the array.
 *    - If length exceeds k, remove the smallest element.
 * 3. The kth largest element is always the smallest in this array.
 */

class KthLargest {
    constructor(k, nums) {
        this.k = k;
        // Keep only the k largest elements initially
        this.heap = nums.sort((a, b) => a - b).slice(-k);
    }

    add(val) {
        this.heap.push(val);                // Add new value
        this.heap.sort((a, b) => a - b);    // Sort ascending
        if (this.heap.length > this.k) {    // Keep only k largest elements
            this.heap.shift();              // Remove smallest
        }
        return this.heap[0];                // kth largest element
    }
}

// Driver code to test KthLargest
const kthLargest = new KthLargest(3, [4,5,8,2]);
console.log(kthLargest.add(3));  // Expected: 4
console.log(kthLargest.add(5));  // Expected: 5
console.log(kthLargest.add(10)); // Expected: 5
console.log(kthLargest.add(9));  // Expected: 8
console.log(kthLargest.add(4));  // Expected: 8
```

```
/*#*/
/**
 * Problem Definition:
 * -------------------
 * Given an array of intervals `intervals` where intervals[i] = [start, end] and
 * an array of query points `queries`, return an array `res` where `res[j]` is the
 * length of the smallest interval that contains `queries[j]`. If no such interval
 * exists for a query, return -1 for that position.
 *
 * Intuition:
 * ----------
 * To efficiently answer queries, sort both intervals and queries by their start points/values.
 * As you process each query in ascending order, add to a "min-heap" all intervals whose
 * start <= current query. Remove intervals from the heap that end < current query (they
 * no longer contain it). The heap's top element gives the smallest interval containing
 * the query.
 *
 * Logic:
 * ------
 * 1. Sort intervals by start time ascending.
 * 2. Pair queries with their original indices, then sort queries ascending by value.
 * 3. Initialize an empty heap (here simulated with an array and manual sort for simplicity)
 *    and a result array filled with -1.
 * 4. For each query:
 *    a. Add all intervals starting <= query to the heap, storing [intervalLength, end].
 *    b. Sort the heap to keep the smallest intervalLength at the front.
 *    c. Remove intervals whose end < query (they don't cover the query).
 *    d. If the heap is not empty, the front element's length is the answer for this query.
 * 5. Return the result array in the original query order.
 */

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 = new MinHeap();
    let i = 0;

    for (const [q, idx] of sortedQueries) {
        // Add intervals that start before or at the current query
        while (i < intervals.length && intervals[i][0] <= q) {
            const [start, end] = intervals[i];
            heap.push([end - start + 1, end]); // store [length, end]
            i++;
        }

        // Maintain heap property by sorting by interval length
        heap.sort((a, b) => a[0] - b[0]);

        // Remove intervals that end before the current query
        while (heap.length && heap[0][1] < q) heap.shift();

        // If any interval covers the query, pick the smallest length
        if (heap.length) res[idx] = heap[0][0];
    }
    return res;
}

// -------- Driver Code --------
function main() {
    const intervals = [[1,4],[2,4],[3,6],[4,4]];
    const queries = [2,3,4,5];
    const result = minInterval(intervals, queries);
    console.log(`Intervals: ${JSON.stringify(intervals)}`);
    console.log(`Queries: ${queries}`);
    console.log(`Result: ${result}`); // Expected: [3,3,1,4]
}

main();
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * You are given two non-empty linked lists representing two non-negative integers.
 * The digits are stored in reverse order, and each node contains a single digit.
 * Add the two numbers and return the sum as a linked list, also in reverse order.
 *
 * Example:
 * ----------
 * Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
 * Output: 7 -> 0 -> 8
 * Explanation: 342 + 465 = 807
 *
 * Input: (0) + (0)
 * Output: 0
 *
 * Input: (9 -> 9 -> 9 -> 9 -> 9 -> 9 -> 9) + (9 -> 9 -> 9 -> 9)
 * Output: 8 -> 9 -> 9 -> 9 -> 0 -> 0 -> 0 -> 1
 *
 * Intuition:
 * ----------
 * 1. Use a dummy head node to simplify handling the head of the result list.
 * 2. Traverse both linked lists simultaneously, adding corresponding digits along with carry.
 * 3. If one list is shorter, treat missing nodes as 0.
 * 4. Maintain carry for sums >= 10.
 * 5. After processing both lists, if carry > 0, add a final node for it.
 */

function ListNode(val, next) {
    this.val = val;
    this.next = next || null;
}

function addTwoNumbers(l1, l2) {
    const dummy = new ListNode(0); // Dummy head for result linked list
    let p = l1, q = l2, curr = dummy; // Pointers for traversing lists
    let carry = 0; // Carry for sum >= 10

    while (p || q || carry) {
        const sum = (p ? p.val : 0) + (q ? q.val : 0) + carry; // Add digits + carry
        carry = Math.floor(sum / 10); // Update carry
        curr.next = new ListNode(sum % 10); // Create new node for current digit
        curr = curr.next; // Move current pointer
        if (p) p = p.next; // Move pointer in l1
        if (q) q = q.next; // Move pointer in l2
    }

    return dummy.next; // Return the head of the result list
}

// Helper function to create a linked list from an array
function arrayToList(arr) {
    const dummy = new ListNode(0);
    let curr = dummy;
    for (const val of arr) {
        curr.next = new ListNode(val);
        curr = curr.next;
    }
    return dummy.next;
}

// Helper function to print linked list as array
function listToArray(head) {
    const arr = [];
    while (head) {
        arr.push(head.val);
        head = head.next;
    }
    return arr;
}

// Driver code to test addTwoNumbers
const testCases = [
    { l1: [2,4,3], l2: [5,6,4], expected: [7,0,8] },
    { l1: [0], l2: [0], expected: [0] },
    { l1: [9,9,9,9,9,9,9], l2: [9,9,9,9], expected: [8,9,9,9,0,0,0,1] }
];

testCases.forEach(({l1, l2, expected}, i) => {
    const result = addTwoNumbers(arrayToList(l1), arrayToList(l2));
    console.log(`Test Case ${i + 1}: Expected = [${expected}], Got = [${listToArray(result)}]`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given an array nums containing n + 1 integers where each integer is between 1 and n (inclusive),
 * there is only **one repeated number**. Find this duplicate number **without modifying the array** and
 * using constant extra space.
 *
 * Example:
 * ----------
 * Input: [1,3,4,2,2]
 * Output: 2
 *
 * Input: [3,1,3,4,2]
 * Output: 3
 *
 * Constraints:
 * - Must use O(1) extra space.
 * - Must not modify the input array.
 *
 * Intuition:
 * ----------
 * This problem can be solved using **Floyd's Tortoise and Hare (Cycle Detection) algorithm**:
 * 1. Treat the array as a linked list where the value points to the next index.
 * 2. A duplicate number creates a cycle in this "linked list".
 * 3. Use slow and fast pointers to detect the cycle.
 * 4. Once a cycle is detected, reset one pointer to the start and move both one step at a time.
 * 5. They meet at the duplicate number.
 */

function findDuplicate(nums) {
    let slow = nums[0], fast = nums[0];

    // Phase 1: Detect cycle using Floyd's Tortoise and Hare
    do {
        slow = nums[slow];          // Move slow pointer 1 step
        fast = nums[nums[fast]];    // Move fast pointer 2 steps
    } while (slow !== fast);

    // Phase 2: Find entrance to cycle (duplicate number)
    slow = nums[0]; // Reset slow to start
    while (slow !== fast) {
        slow = nums[slow]; // Move both pointers 1 step
        fast = nums[fast];
    }

    return slow; // Duplicate number
}

// Driver code to test the function
const testCases = [
    { nums: [1,3,4,2,2], expected: 2 },
    { nums: [3,1,3,4,2], expected: 3 },
    { nums: [1,1], expected: 1 },
    { nums: [1,1,2], expected: 1 }
];

testCases.forEach(({nums, expected}, i) => {
    const result = findDuplicate(nums);
    console.log(`Test Case ${i + 1}: Expected = ${expected}, Got = ${result}`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.
 *
 * - k is a positive integer and is less than or equal to the length of the linked list.
 * - If the number of nodes is not a multiple of k then the remaining nodes at the end should remain as-is.
 * - You may not alter the values in the nodes—only nodes themselves may be changed.
 *
 * Example:
 * ----------
 * Input: head = 1->2->3->4->5, k = 2
 * Output: 2->1->4->3->5
 *
 * Input: head = 1->2->3->4->5, k = 3
 * Output: 3->2->1->4->5
 *
 * Intuition:
 * ----------
 * 1. Use a helper function to reverse a segment of the list from `start` to `end` (exclusive).
 * 2. Count if there are at least k nodes; if not, return head as is.
 * 3. Reverse the first k nodes and recursively process the remaining list.
 * 4. Connect the reversed portion to the recursively reversed rest.
 */

function ListNode(val, next) {
    this.val = val;
    this.next = next || null;
}

function reverseKGroup(head, k) {
    // Helper function to reverse a segment from start to end (exclusive)
    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; // New head of reversed segment
    };

    // Count nodes to ensure we have at least k nodes
    let node = head, count = 0;
    while (node) { count++; node = node.next; }
    if (count < k) return head; // Less than k nodes, return as-is

    // Reverse first k nodes
    let newHead = reverse(head, head);
    let temp = head;
    for (let i = 0; i < k; i++) temp = temp.next; // Move temp k steps forward

    // Recursively reverse the remaining list and connect
    head.next = reverseKGroup(temp, k);
    return newHead;
}

// Helper function to create a linked list from array
function arrayToList(arr) {
    const dummy = new ListNode(0);
    let curr = dummy;
    for (const val of arr) {
        curr.next = new ListNode(val);
        curr = curr.next;
    }
    return dummy.next;
}

// Helper function to convert linked list to array
function listToArray(head) {
    const arr = [];
    while (head) {
        arr.push(head.val);
        head = head.next;
    }
    return arr;
}

// Driver code to test reverseKGroup
const testCases = [
    { list: [1,2,3,4,5], k: 2, expected: [2,1,4,3,5] },
    { list: [1,2,3,4,5], k: 3, expected: [3,2,1,4,5] },
    { list: [1,2,3,4,5,6], k: 3, expected: [3,2,1,6,5,4] } // For full groups only
];

testCases.forEach(({list, k, expected}, i) => {
    const result = reverseKGroup(arrayToList(list), k);
    console.log(`Test Case ${i + 1}: Expected = [${expected}], Got = [${listToArray(result)}]`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Given an m x n matrix where:
 *  - Integers in each row are sorted in ascending order from left to right.
 *  - Integers in each column are sorted in ascending order from top to bottom.
 *
 * Write a function to determine if a given 'target' exists in the matrix.
 *
 * Example:
 * ----------
 * Input:
 * matrix = [
 *   [1, 4, 7, 11, 15],
 *   [2, 5, 8, 12, 19],
 *   [3, 6, 9, 16, 22],
 *   [10, 13, 14, 17, 24],
 *   [18, 21, 23, 26, 30]
 * ], target = 5
 * Output: true
 *
 * Input: same matrix, target = 20
 * Output: false
 *
 * Intuition:
 * ----------
 * 1. Start from the top-right corner of the matrix.
 * 2. Compare the current element with the target:
 *    - If equal, return true.
 *    - If current > target, move left (decrease column) because all elements below are bigger.
 *    - If current < target, move down (increase row) because all elements to the left are smaller.
 * 3. Repeat until you find the target or go out of bounds.
 *
 * This works in O(m + n) time.
 */

function searchMatrix(matrix, target) {
    let r = 0; // Start row
    let c = matrix[0].length - 1; // Start column (top-right corner)

    // Traverse the matrix
    while (r < matrix.length && c >= 0) {
        if (matrix[r][c] === target) {
            return true; // Found the target
        } else if (matrix[r][c] > target) {
            c--; // Move left
        } else {
            r++; // Move down
        }
    }

    return false; // Target not found
}

// Driver code to test the function
const testCases = [
    { matrix: [
        [1, 4, 7, 11, 15],
        [2, 5, 8, 12, 19],
        [3, 6, 9, 16, 22],
        [10, 13, 14, 17, 24],
        [18, 21, 23, 26, 30]
      ], target: 5, expected: true
    },
    { matrix: [
        [1, 4, 7, 11, 15],
        [2, 5, 8, 12, 19],
        [3, 6, 9, 16, 22],
        [10, 13, 14, 17, 24],
        [18, 21, 23, 26, 30]
      ], target: 20, expected: false
    }
];

testCases.forEach(({matrix, target, expected}, i) => {
    const result = searchMatrix(matrix, target);
    console.log(`Test Case ${i + 1}: Expected = ${expected}, Got = ${result}`);
});
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Design a time-based key-value store that can store multiple values for the same key at different timestamps.
 *
 * Implement the TimeMap class:
 * - set(key, value, timestamp): Stores the key with the value at the given timestamp.
 * - get(key, timestamp): Returns the value such that set(key, value, t) was called previously with t <= timestamp.
 *   If there are multiple such values, return the one with the largest t. If none, return "".
 *
 * Example:
 * ----------
 * Input:
 * tm = new TimeMap();
 * tm.set("foo", "bar", 1);
 * tm.get("foo", 1); // returns "bar"
 * tm.get("foo", 3); // returns "bar"
 * tm.set("foo", "bar2", 4);
 * tm.get("foo", 4); // returns "bar2"
 * tm.get("foo", 5); // returns "bar2"
 *
 * Intuition:
 * ----------
 * 1. Use a Map to store key → list of [timestamp, value] pairs.
 * 2. For get():
 *    - Use binary search on the timestamp list to find the largest timestamp <= query timestamp.
 *    - This ensures O(log n) retrieval for each key's history.
 */

class TimeMap {
    constructor() {
        this.map = new Map(); // key -> list of [timestamp, value]
    }

    /**
     * Stores the value for a key at a specific timestamp
     * @param {string} key
     * @param {string} value
     * @param {number} timestamp
     */
    set(key, value, timestamp) {
        if (!this.map.has(key)) {
            this.map.set(key, []); // Initialize array if key doesn't exist
        }
        this.map.get(key).push([timestamp, value]); // Append [timestamp, value]
    }

    /**
     * Retrieves the value with largest timestamp <= query timestamp
     * @param {string} key
     * @param {number} timestamp
     * @returns {string}
     */
    get(key, timestamp) {
        const arr = this.map.get(key) || []; // Get the array of [timestamp, value]
        let l = 0, r = arr.length - 1;
        let res = ""; // Default if no timestamp <= query

        // Binary search for the largest timestamp <= query timestamp
        while (l <= r) {
            const m = Math.floor((l + r) / 2);
            if (arr[m][0] <= timestamp) {
                res = arr[m][1]; // Update result
                l = m + 1;       // Look for a later timestamp
            } else {
                r = m - 1;       // Move left
            }
        }

        return res;
    }
}

// Driver code to test TimeMap
const tm = new TimeMap();
tm.set("foo", "bar", 1);
console.log(tm.get("foo", 1)); // Expected: "bar"
console.log(tm.get("foo", 3)); // Expected: "bar"
tm.set("foo", "bar2", 4);
console.log(tm.get("foo", 4)); // Expected: "bar2"
console.log(tm.get("foo", 5)); // Expected: "bar2"
console.log(tm.get("foo", 0)); // Expected: ""
```

```javascript
/*#*/
/**
 * 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
 * and keep track of total count for both odd and even length palidromes
 */
function countSubstrings(s) {
    let count = 0;

    function 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 Definition:
 * -------------------
 * Given two strings s1 and s2, return true if s2 contains a permutation of s1,
 * otherwise return false. In other words, check if s2 has any substring
 * that is an anagram of s1.
 * (LeetCode 567: Permutation in String)
 *
 * Example:
 *   s1 = "ab", s2 = "eidbaooo"
 *   Output: true
 *   Explanation: "ba" is a permutation of "ab".
 *
 * Intuition:
 * ----------
 * Use a sliding window of size equal to s1's length over s2. Maintain frequency
 * counts of characters for both s1 and the current window in s2. If at any point
 * the frequency arrays match, a permutation exists.
 *
 * Logic:
 * ------
 * 1. Initialize two frequency arrays of length 26 (for lowercase letters).
 * 2. Populate counts for s1 and the first window of s2.
 * 3. If the counts match, return true.
 * 4. Slide the window through s2:
 *    - Add the next character's count.
 *    - Remove the outgoing character's count.
 *    - Compare arrays. If they match, return true.
 * 5. If no match found, return false.
 *
 * Complexity:
 * -----------
 * - Time: O(m * 26) in worst case due to string comparisons,
 *   but effectively O(m) where m = s2.length.
 * - Space: O(26) = O(1).
 */

function checkInclusion(s1, s2) {
    const a = Array(26).fill(0), b = Array(26).fill(0);
    const aCode = 'a'.charCodeAt(0);

    const n = s1.length;
    const m = s2.length;
    if (n > m) return false;

    // Count frequencies for s1 and the first window in s2
    for (let i = 0; i < n; i++) {
        a[s1.charCodeAt(i) - aCode]++;
        b[s2.charCodeAt(i) - aCode]++;
    }

    // Check initial window
    if (a.toString() === b.toString()) return true;

    // Slide the window through s2
    for (let i = n; i < m; i++) {
        b[s2.charCodeAt(i) - aCode]++;           // add new character
        b[s2.charCodeAt(i - n) - aCode]--;       // remove outgoing character
        if (a.toString() === b.toString()) return true;
    }

    return false;
}

// -------- Driver Code --------
function main() {
    const s1 = "ab";
    const s2 = "eidbaooo";
    const result = checkInclusion(s1, s2);
    console.log(`s1 = "${s1}", s2 = "${s2}"`);
    console.log(`Does s2 contain a permutation of s1? ${result}`); // Expected: true
}

main();
```

```
/*#*/
/**
 * Problem Statement:
 * ------------------
 * Evaluate the value of an arithmetic expression in Reverse Polish Notation (RPN).
 *
 * In Reverse Polish Notation, every operator follows all of its operands.
 * For example:
 * - The RPN expression ["2","1","+","3","*"] evaluates to ((2 + 1) * 3) = 9
 * - The RPN expression ["4","13","5","/","+"] evaluates to (4 + (13 / 5)) = 6
 *
 * Valid operators are: +, -, *, /
 * Each operand may be an integer or another expression.
 * Division between two integers should truncate toward zero.
 *
 * Intuition:
 * ----------
 * Use a stack to keep track of operands.
 * 1. Iterate through each token in the input array.
 * 2. If the token is a number, push it onto the stack.
 * 3. If the token is an operator:
 *    - Pop the top two numbers from the stack (right-hand side first).
 *    - Apply the operator.
 *    - Push the result back onto the stack.
 * 4. At the end, the stack will have a single element which is the result.
 *
 * Example:
 * ----------
 * Input: ["2","1","+","3","*"]
 * Step 1: Push 2, Push 1
 * Step 2: Encounter "+", pop 1 and 2 -> 2+1=3, push 3
 * Step 3: Push 3
 * Step 4: Encounter "*", pop 3 and 3 -> 3*3=9, push 9
 * Output: 9
 */

function evalRPN(tokens) {
    const stack = []; // Stack to hold numbers during evaluation

    for (const t of tokens) {
        // If the token is an operator
        if ("+-*/".includes(t)) {
            const b = stack.pop(); // Second operand
            const a = stack.pop(); // First operand
            // Perform operation and push result back to stack
            stack.push(
                t === '+' ? a + b :
                t === '-' ? a - b :
                t === '*' ? a * b :
                Math.trunc(a / b) // truncate division toward zero
            );
        } else {
            // If token is a number, parse it and push to stack
            stack.push(parseInt(t));
        }
    }

    // Final result is the only element in stack
    return stack[0];
}

// Driver code to test the function
const testCases = [
    { tokens: ["2","1","+","3","*"], expected: 9 },
    { tokens: ["4","13","5","/","+"], expected: 6 },
    { tokens: ["10","6","9","3","+","-11","*","/","*","17","+","5","+"], expected: 22 }
];

testCases.forEach(({tokens, expected}, i) => {
    const result = evalRPN(tokens);
    console.log(`Test Case ${i + 1}: Expected = ${expected}, Got = ${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;
    }
}
```