```
/**
 * COMPLEXITY ANALYSIS PRACTICE PROBLEMS
 * Try to figure out the time and space complexity before looking at answers!
 */

// ============================================
// PRACTICE PROBLEM 1
// ============================================
function problem1(arr) {
   return arr[arr.length - 1];
}
/**
 * YOUR ANALYSIS:
 * Time Complexity: ?
 * Space Complexity: ?
 *
 * ANSWER:
 * Time: O(1) - Just accessing one element by index
 * Space: O(1) - No extra memory used
 */

// ============================================
// PRACTICE PROBLEM 2
// ============================================
function problem2(arr) {
   let sum = 0;
   for (let i = 0; i < arr.length; i++) {
      sum += arr[i];
   }
   return sum;
}
/**
 * YOUR ANALYSIS:
 * Time Complexity: ?
 * Space Complexity: ?
 *
 * ANSWER:
 * Time: O(n) - Loop runs n times
 * Space: O(1) - Only uses 'sum' variable
 */

// ============================================
// PRACTICE PROBLEM 3
// ============================================
function problem3(arr) {
   const doubled = [];
   for (let i = 0; i < arr.length; i++) {
      doubled.push(arr[i] * 2);
```

```
    }
    return doubled;
}
/**
 * YOUR ANALYSIS:
 * Time Complexity: ?
 * Space Complexity: ?
 *
 * ANSWER:
 * Time: O(n) - Loop runs n times
 * Space: O(n) - Creates new array of size n
 */

// ============================================
// PRACTICE PROBLEM 4
// ============================================
function problem4(arr) {
    for (let i = 0; i < arr.length; i++) {
        for (let j = 0; j < arr.length; j++) {
            if (i !== j && arr[i] === arr[j]) {
                return true;
            }
        }
    }
    return false;
}
/**
 * YOUR ANALYSIS:
 * Time Complexity: ?
 * Space Complexity: ?
 *
 * ANSWER:
 * Time: O(n²) - Nested loops, each runs n times
 * Space: O(1) - Only uses loop variables
 */

// ============================================
// PRACTICE PROBLEM 5
// ============================================
function problem5(arr, target) {
    let left = 0;
    let right = arr.length - 1;

    while (left <= right) {
        let mid = Math.floor((left + right) / 2);

        if (arr[mid] === target) {
            return mid;
```

```
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
/**
 * YOUR ANALYSIS:
 * Time Complexity: ?
 * Space Complexity: ?
 *
 * ANSWER:
 * Time: O(log n) - Cuts search space in half each iteration
 * Space: O(1) - Only uses a few variables
 */

// ============================================
// PRACTICE PROBLEM 6
// ============================================
function problem6(n) {
    if (n <= 1) return n;
    return problem6(n - 1) + problem6(n - 2);
}
/**
 * YOUR ANALYSIS:
 * Time Complexity: ?
 * Space Complexity: ?
 *
 * ANSWER:
 * Time: O(2^n) - Each call makes 2 more calls (exponential!)
 * Space: O(n) - Recursion depth goes up to n levels
 *
 * This is the naive Fibonacci - very inefficient!
 */

// ============================================
// PRACTICE PROBLEM 7
// ============================================
function problem7(arr1, arr2) {
    const result = [];
    for (let i = 0; i < arr1.length; i++) {
        for (let j = 0; j < arr2.length; j++) {
            result.push([arr1[i], arr2[j]]);
        }
    }
    return result;
```

```
}
/**
 * YOUR ANALYSIS:
 * Time Complexity: ?
 * Space Complexity: ?
 *
 * ANSWER:
 * Time: O(a * b) where a = arr1.length, b = arr2.length
 * Space: O(a * b) - Creates a*b pairs
 *
 * IMPORTANT: This is NOT O(n²) because we have two different inputs!
 */

// ============================================
// PRACTICE PROBLEM 8
// ============================================
function problem8(str) {
   const chars = {};
   for (let i = 0; i < str.length; i++) {
      chars[str[i]] = (chars[str[i]] || 0) + 1;
   }
   return chars;
}
/**
 * YOUR ANALYSIS:
 * Time Complexity: ?
 * Space Complexity: ?
 *
 * ANSWER:
 * Time: O(n) - Loop through string once
 * Space: O(k) where k = number of unique characters
 * In worst case, k = n, so O(n) space
 */

// ============================================
// TRICKY PROBLEM 9
// ============================================
function problem9(arr) {
   for (let i = 0; i < arr.length; i++) {
      for (let j = i + 1; j < arr.length; j++) {
         for (let k = j + 1; k < arr.length; k++) {
            console.log(arr[i], arr[j], arr[k]);
         }
      }
   }
}
/**
 * YOUR ANALYSIS:
```

```
 * Time Complexity: ?
 * Space Complexity: ?
 *
 * ANSWER:
 * Time: O(n³) - Triple nested loops!
 * Space: O(1) - Only loop variables
 *
 * This prints all combinations of 3 elements
 */

// ============================================
// TRICKY PROBLEM 10
// ============================================
function problem10(arr) {
   let n = arr.length;
   while (n > 1) {
      for (let i = 0; i < n; i++) {
         console.log(arr[i]);
      }
      n = Math.floor(n / 2);
   }
}
/**
 * YOUR ANALYSIS:
 * Time Complexity: ?
 * Space Complexity: ?
 *
 * ANSWER:
 * Time: O(n) - First loop prints n, then n/2, then n/4...
 * Total: n + n/2 + n/4 + ... = 2n = O(n)
 * Space: O(1) - Only loop variables
 *
 * This is tricky! Even though there's a loop inside another loop,
 * the outer loop runs fewer times each iteration.
 */

// ============================================
// COMPLEXITY CHEAT SHEET
// ============================================

/**
 * COMMON PATTERNS TO RECOGNIZE:
 *
 * 1. Single loop through array: O(n)
 * 2. Nested loops (same array): O(n²)
 * 3. Nested loops (different arrays): O(a * b)
 * 4. Binary search pattern: O(log n)
 * 5. Recursive with single call: O(n) time, O(n) space
```

```
 * 6. Recursive with multiple calls: Often O(2^n) time
 * 7. Creating new array of same size: O(n) space
 * 8. Only using variables: O(1) space
 *
 * SORTING ALGORITHMS:
 * - Bubble, Selection, Insertion: O(n²)
 * - Merge Sort: O(n log n) time, O(n) space
 * - Quick Sort: O(n log n) average, O(n²) worst
 * - Heap Sort: O(n log n) time, O(1) space
 *
 * DATA STRUCTURE OPERATIONS:
 * - Array access: O(1)
 * - Array search: O(n)
 * - Hash table: O(1) average
 * - Binary Search Tree: O(log n) average
 * - Linked List: O(n) search, O(1) insert/delete at known position
 */

// ============================================
// PERFORMANCE TESTING EXAMPLE
// ============================================

function performanceTest() {
    console.log("=== Performance Testing ===");

    // Test different array sizes
    const sizes = [1000, 10000, 100000];

    sizes.forEach(size => {
        const arr = Array.from({length: size}, (_, i) => i);

        // O(1) operation
        console.time(`O(1) - size ${size}`);
        console.log(`First element: ${arr[0]}`);
        console.timeEnd(`O(1) - size ${size}`);

        // O(n) operation
        console.time(`O(n) - size ${size}`);
        let sum = 0;
        for (let i = 0; i < arr.length; i++) {
            sum += arr[i];
        }
        console.timeEnd(`O(n) - size ${size}`);

        // O(log n) operation
        console.time(`O(log n) - size ${size}`);
        binarySearch(arr, size - 1);
        console.timeEnd(`O(log n) - size ${size}`);
```

```
        console.log("---");
    });
}

function binarySearch(arr, target) {
    let left = 0, right = arr.length - 1;
    while (left <= right) {
        const mid = Math.floor((left + right) / 2);
        if (arr[mid] === target) return mid;
        if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}

// Uncomment to run performance test
// performanceTest();

// ============================================
// STUDY PLAN
// ============================================

/**
 * DAY 1-2: Master the Basics
 * - Understand what Big O means
 * - Learn O(1), O(n), O(n²), O(log n)
 * - Practice identifying simple patterns
 *
 * DAY 3-4: Practice Analysis
 * - Analyze your own code
 * - Work through the practice problems above
 * - Time yourself solving problems
 *
 * DAY 5-7: Advanced Patterns
 * - Recursive complexity
 * - Multiple variables (O(a*b))
 * - Space complexity in detail
 * - Common algorithm complexities
 *
 * ONGOING: Apply to Everything
 * - Always analyze your solutions
 * - Compare different approaches
 * - Optimize when needed
 */

console.log("Ready to master complexity analysis! 🚀");
console.log("Start with the practice problems above!");
```

```
console.log("Remember: Practice makes perfect! 💪");
```