**Code Challenge Problem 4: Three Ways to Sum to n**

**1. Recursive Approach (Basic Recursion - O(n))**

This method uses a recursive function that repeatedly calls itself with n - 1 until it reaches the base case (n === 1).

**Implementation:**

function sum\_to\_n\_recursive(n: number): number {  
 if (n <= 0) return 0;   
 if (n === 1) return 1;   
 return n + sum\_to\_n\_recursive(n - 1);  
}

**Complexity & Analysis:**

Time Complexity: O(n) (each call reduces n by 1, leading to n recursive calls).

Space Complexity: O(n) (each call is stored in the call stack, leading to possible stack overflow for large n).

Pros: Simple, easy to understand.

Cons: Inefficient for large n due to high memory usage.

**2. Iterative Approach (Loop - O(n))**

This method uses a loop to add numbers from 1 to n, avoiding the overhead of recursion.

**Implementation:**

function sum\_to\_n\_iterative(n: number): number {

if (n < 0) return 0;  
 let sum = 0;  
 for (let i = 1; i <= n; i++) {  
 sum += i;  
 }  
 return sum;  
}

**Complexity & Analysis:**

Time Complexity: O(n) (one loop iteration per number up to n).

Space Complexity: O(1) (only one variable sum is used, no recursion overhead).

Pros: More efficient than recursion, no risk of stack overflow.

Cons: Still requires O(n) iterations, not the fastest approach.

**3. Mathematical Approach (Constant Time - O(1))**

This method uses the mathematical formula for summation of the first n natural numbers: total = n \* (n + 1) / 2

**Implementation:**

function sum\_to\_n\_math(n: number): number {

if (n < 0) return 0;  
 return (n \* (n + 1)) / 2;  
}

**Complexity & Analysis:**

Time Complexity: O(1) (single computation, independent of n).

Space Complexity: O(1) (no extra memory usage).

Pros: Extremely fast, most efficient approach.

Cons: May cause integer overflow for very large n, but safe within Number.MAX\_SAFE\_INTEGER.