03. Loop Tracing & Recursion Call Stack Analysis

> **Module Focus**: Systematic variable state tables for single and nested loops, call stack winding/unwinding phases, recursion tree diagrams, static variable state retention, and execution order mechanics for placement examinations.

---

1. Loop Tracing Methodology & Variable State Tables

In timed pseudocode assessments, manual mental math leads to avoidable errors. Constructing a **Variable State Table** tracks state transitions deterministically across loop iterations.

Key Rules for Constructing State Tables:

1. **Define Columns**: List Iteration Number, Loop Control Variables (LCVs), Conditions Evaluated, Internal Variable Modifications, and Final Outputs.
2. **Track Exit Conditions**: Always include the iteration step where the loop test condition evaluates to `FALSE` to capture final post-loop variable states.
3. **Sequence Operations**: Execute updates strictly top-to-bottom within each iteration.

Example 1.1: Nested `FOR` Loop with Dynamic Inner Bounds

```
Integer i, j, k = 0
FOR i = 1 TO 3
    FOR j = 1 TO i
        k = k + i + j
    END FOR
END FOR
PRINT k
```

Step-by-Step Variable State Table:

Outer `i`	Inner `j`	Inner Loop Condition (`j <= i`)	Calculation (`k = k + i + j`)	Updated `k` Value	Next Action
1	1	$1 \le 1$ (TRUE)	$0 + 1 + 1$	**2**	Increment `j` to 2
1	2	$2 \le 1$ (FALSE)	-	2	Exit inner loop; Increment `i` to 2
2	1	$1 \le 2$ (TRUE)	$2 + 2 + 1$	**5**	Increment `j` to 2
2	2	$2 \le 2$ (TRUE)	$5 + 2 + 2$	**9**	Increment `j` to 3

Outer `i`	Inner `j`	Inner Loop Condition (`j <= i`)	Calculation (`k = k + i + j`)	Updated `k` Value	Next Action
2	3	$3 \le 2$ (FALSE)	-	9	Exit inner loop; Increment `i` to 3
3	1	$1 \le 3$ (TRUE)	$9 + 3 + 1$	**13**	Increment `j` to 2
3	2	$2 \le 3$ (TRUE)	$13 + 3 + 2$	**18**	Increment `j` to 3
3	3	$3 \le 3$ (TRUE)	$18 + 3 + 3$	**24**	Increment `j` to 4
3	4	$4 \le 3$ (FALSE)	-	**24**	Exit inner loop; Increment `i` to 4
4	-	$4 \le 3$ (FALSE)	-	**24**	Exit outer loop; Print `k`

Final Output: `k = 24`

Example 1.2: Dynamic `WHILE` Loop with Multi-Variable Mutation

```
Integer i = 1, j = 10, count = 0
WHILE (i < j)
    count = count + 1
    i = i * 2
    j = j - 1
END WHILE
PRINT count, i, j
```

State Table Trace:

Iteration	Condition (`i < j`)	`count = count + 1`	`i = i * 2`	`j = j - 1`	State at End of Iteration
Initial	-	0	1	10	`i=1, j=10, count=0`
1	$1 < 10$ (TRUE)	1	2	9	`i=2, j=9, count=1`
2	$2 < 9$ (TRUE)	2	4	8	`i=4, j=8, count=2`
3	$4 < 8$ (TRUE)	3	8	7	`i=8, j=7, count=3`
4	$8 < 7$ (**FALSE**)	-	-	-	Loop Terminates

Final Output: `count = 3`, `i = 8`, `j = 7`

Example 1.3: REPEAT-UNTIL Loop Mechanics vs WHILE Loop Traps

Unlike a `WHILE` loop (which tests its condition at entry), a `REPEAT-UNTIL` loop evaluates its condition **at the end of every iteration**. Furthermore, `REPEAT-UNTIL` **continues when the condition is FALSE** and **terminates when the condition becomes TRUE**.

Sub-Example A: REPEAT-UNTIL Post-Test Execution Trace

```
Integer i = 5, sum = 0
REPEAT
    sum = sum + i
    i = i - 2
UNTIL (i <= 0)
PRINT sum, i
```

Variable State Table:

Iteration	Initial `i`	`sum = sum + i`	`i = i - 2`	Condition Evaluated `(i <= 0)`	Result	Loop Action
Pass 1	5	$0 + 5 = \mathbf{5}$	3	`(3 <= 0)`	**FALSE**	Continue loop
Pass 2	3	$5 + 3 = \mathbf{8}$	1	`(1 <= 0)`	**FALSE**	Continue loop
Pass 3	1	$8 + 1 = \mathbf{9}$	-1	`(-1 <= 0)`	**TRUE**	**TERMINATE**

- **Final Output**: `sum = 9`, `i = -1`
- **Key Takeaway**: Notice that even when `i` reached `1`, the body executed *before* `i` dropped to `-1`.

Sub-Example B: CoCubes Infinite Loop Trap with Loop Control Modification

```
Integer n
FOR (n = 3; n != 0; n--)
    PRINT n
    n = n - 1
END FOR
```

Step-by-Step State Table Trace:

Loop Step	Current `n`	Condition `n != 0`	Action Inside Loop	Variable Mutation	Post-Iteration Increment `n--`	Next `n` State
Start	3	`3 != 0` (**TRUE**)	Print `3`	`n = 3 - 1 = 2`	`n = 2 - 1 = 1`	`n = 1`

Loop Step	Current `n`	Condition `n != 0`	Action Inside Loop	Variable Mutation	Post-Iteration Increment `n--`	Next `n` State
Iter 2	1	`1 != 0` (**TRUE**)	Print `1`	`n = 1 - 1 = 0`	`n = 0 - 1 = -1`	`n = -1`
Iter 3	-1	`-1 != 0` (**TRUE**)	Print `-1`	`n = -1 - 1 = -2`	`n = -2 - 1 = -3`	`n = -3`

- **Final Output**: **Infinite Loop** (Prints `3, 1, -1, -3, -5...`).
- **Exam Trap Warning**: Candidates assume `n = n - 1` inside the loop causes `n` to hit `0` and terminate. However, inside `n` becomes `0`, and then `n--` in the FOR header decrements it to `-1` *before* checking `n != 0`, missing `0` completely!

---

2. Recursion Call Stack & Unwinding Analysis

Recursion relies on the Call Stack (LIFO - Last In, First Out). Execution occurs in two distinct phases:

1. **Winding Phase (Push)**: Function calls itself recursively, pushing activation frames onto the call stack until reaching the **Base Case**.
2. **Unwinding Phase (Pop)**: Base case returns a concrete value; activation frames pop off the stack, passing return values back to parent callers while executing remaining post-recursive statements.

Example 2.1: Single Linear Recursion (Russian Peasant Multiplication)

```
Integer fun(Integer a, Integer b)
    If (b == 0)
        Return 0
    End If
    If (b % 2 == 0)
        Return fun(a + a, b / 2)
    End If
    Return fun(a + a, b / 2) + a
End Function

Execution Target: fun(3, 5)
```

1. Winding Phase (Stack Push Sequence):

- **Call 1**: `fun(3, 5)` $\rightarrow$ `b=5` (Odd) $\implies$ Returns `fun(6, 2) + 3`
- **Call 2**: `fun(6, 2)` $\rightarrow$ `b=2` (Even) $\implies$ Returns `fun(12, 1)`
- **Call 3**: `fun(12, 1)` $\rightarrow$ `b=1` (Odd) $\implies$ Returns `fun(24, 0) + 12`
- **Call 4**: `fun(24, 0)` $\rightarrow$ `b=0` (Base Case) $\implies$ Returns `0`

2. Call Stack Representation:

```
+-----------------------------------+  |
| Frame 4: fun(24, 0) -> Returns 0  |  | Base Case Reached
+-----------------------------------+  | Unwinding Phase Begins
| Frame 3: fun(12, 1) -> fun(24,0)+12|  v
+-----------------------------------+
| Frame 2: fun(6, 2)  -> fun(12,1)  |
+-----------------------------------+
| Frame 1: fun(3, 5)  -> fun(6,2)+3 |
+-----------------------------------+
```

3. Unwinding Phase (Stack Pop & Result Aggregation):

Stack Level	Frame Popped	Expression Evaluated	Value Returned
Level 4	`fun(24, 0)`	Base Case `0`	**`0`**
Level 3	`fun(12, 1)`	`fun(24, 0) + 12` $\rightarrow 0 + 12$	**`12`**
Level 2	`fun(6, 2)`	`fun(12, 1)` $\rightarrow 12$	**`12`**
Level 1	`fun(3, 5)`	`fun(6, 2) + 3` $\rightarrow 12 + 3$	**`15`**

Final Return Value: **`15`** (Computes $3 \times 5 = 15$).

Example 2.2: Tree / Branching Recursion (Call Stack Unwinding Tree)

Tree recursion occurs when a function makes multiple self-referential recursive calls within a single invocation (e.g., Fibonacci or Divide-and-Conquer).

```
Integer calc(Integer n)
    If (n <= 1)
        Return 1
    End If
    Return calc(n - 1) + calc(n - 2) + n
End Function

Execution Target: calc(4)
```

Call Stack Unwinding Tree Diagram:

```
                          calc(4)  [Returns 16]
                         /   |    \
                  (Left)    (+)    (Right)
                   /                  \
          calc(3) [Returns 8]        calc(2) [Returns 4]
```

```
            /   |   \                  /   |   \
      calc(2)  (+)  calc(1)      calc(1)  (+)  calc(0)
    [Returns 4]     [Ret 1]      [Ret 1]       [Ret 1]
     /  |  \
 calc(1) (+) calc(0)
 [Ret 1]    [Ret 1]
```

Step-by-Step Depth-First Search (DFS) Evaluation Order:

Step	Call / Node	Action / Base Case Test	Value Returned / Expression
1	`calc(4)`	Evaluates `calc(3)` (Left Branch)	Waiting for `calc(3)`
2	`calc(3)`	Evaluates `calc(2)` (Left Branch)	Waiting for `calc(2)`
3	`calc(2)`	Evaluates `calc(1)` (Left Branch)	Returns `1`
4	`calc(2)`	Evaluates `calc(0)` (Right Branch)	Returns `1`
5	`calc(2)`	Computes `calc(1) + calc(0) + 2` $\rightarrow 1 + 1 + 2$	**4**
6	`calc(3)`	Evaluates `calc(1)` (Right Branch)	Returns `1`
7	`calc(3)`	Computes `calc(2) + calc(1) + 3` $\rightarrow 4 + 1 + 3$	**8**
8	`calc(4)`	Evaluates `calc(2)` (Right Branch)	Waiting for `calc(2)`
9	`calc(2)`	Evaluates `calc(1)` (Left) $\rightarrow 1$; `calc(0)` (Right) $\rightarrow 1$	-
10	`calc(2)`	Computes `calc(1) + calc(0) + 2` $\rightarrow 1 + 1 + 2$	**4**
11	`calc(4)`	Computes `calc(3) + calc(2) + 4` $\rightarrow 8 + 4 + 4$	**16**

Final Return Value: **16**

Example 2.3: Static / Global Variables in Recursion (Common Exam Trap)

Static variables are initialized **only once** and retain their modified values across all recursive calls during both winding **and** unwinding phases.

```
Integer fun(Integer n)
    Static Integer x = 0
    If (n <= 0)
        Return 0
    End If
    x = x + 1
    Return fun(n - 1) + x
End Function
```

```
Execution Target: fun(3)
```

Execution Trace with Static Variable Retention:

1. **Winding Phase**:
2. `fun(3)` : `x` becomes $0 + 1 = \mathbf{1}$. Calls `fun(2) + x`.
3. `fun(2)` : `x` becomes $1 + 1 = \mathbf{2}$. Calls `fun(1) + x`.
4. `fun(1)` : `x` becomes $2 + 1 = \mathbf{3}$. Calls `fun(0) + x`.
5. `fun(0)` : Base case reached (`n <= 0`). Returns `0`.
6. **Unwinding Phase** (Crucial: `x` is now **`3`** globally across all frames!):
7. `fun(1)` receives `0` from `fun(0)` $\rightarrow$ Computes `0 + x` $\rightarrow 0 + 3 =$ **`3`**
8. `fun(2)` receives `3` from `fun(1)` $\rightarrow$ Computes `3 + x` $\rightarrow 3 + 3 =$ **`6`**
9. `fun(3)` receives `6` from `fun(2)` $\rightarrow$ Computes `6 + x` $\rightarrow 6 + 3 =$ **`9`**

Final Return Value: **`9`**
(Note: If `x` were a non-static local variable, `fun(3)` would have evaluated to $1 + 2 + 3 = 6$. The static retention increases the total output to 9).

Example 2.4: CoCubes Parameter-Swapping Recursive Call Stack Unwinding

```
Function fun(Integer a, Integer b)
    If (a < b)
        Return fun(b, a)
    Else If (b != 0)
        Return a + fun(a, b - 1)
    Else
        Return 0
    End If
End Function

Execution Target: fun(8, 9)
```

Call Stack Winding Phase & Activation Sequence:

1. **Call 1**: `fun(8, 9)` $\to$ Condition `(8 < 9)` is **TRUE**. Returns `fun(9, 8)` (Swaps parameters so $a \ge b$).
2. **Call 2**: `fun(9, 8)` $\to$ `9 < 8` FALSE, `8 != 0` TRUE. Returns `9 + fun(9, 7)`.
3. **Call 3**: `fun(9, 7)` $\to$ Returns `9 + fun(9, 6)`.
4. ... Repeatedly subtracts 1 from `b` while holding `a = 9` ...
5. **Call 10**: `fun(9, 0)` $\to$ Base Case `b == 0` reached! Returns `0`.

Call Stack Unwinding Table:

Stack Frame Level	Function Call	Branch Executed	Value Returned / Expression	Accumulated Value
Level 10	`fun(9, 0)`	Base Case `b == 0`	Returns `0`	**0**
Level 9	`fun(9, 1)`	`9 + fun(9, 0)`	$9 + 0$	**9**
Level 8	`fun(9, 2)`	`9 + fun(9, 1)`	$9 + 9$	**18**
Level 7	`fun(9, 3)`	`9 + fun(9, 2)`	$9 + 18$	**27**
Level 6	`fun(9, 4)`	`9 + fun(9, 3)`	$9 + 27$	**36**
Level 5	`fun(9, 5)`	`9 + fun(9, 4)`	$9 + 36$	**45**
Level 4	`fun(9, 6)`	`9 + fun(9, 5)`	$9 + 45$	**54**
Level 3	`fun(9, 7)`	`9 + fun(9, 6)`	$9 + 54$	**63**
Level 2	`fun(9, 8)`	`9 + fun(9, 7)`	$9 + 63$	**72**
Level 1	`fun(8, 9)`	`fun(9, 8)`	Returns value from Level 2	**72**

- **Final Return Value**: `72` (Computes multiplication $9 \times 8 = 72$).
- **Exam Rule**: Whenever a recursive call swaps parameters on `a < b`, it standardizes the larger parameter as the multiplicand and the smaller as the iteration count.

---

3. Quick Reference: Recursion Classification & Shortcuts

Recursion Type	Structure	Stack Behavior	Shortcut / Optimization Strategy
Tail Recursion	Recursive call is the *last statement* (no operations after call).	Stack frame can be reused (O(1) space optimization).	Equivalent to an iterative loop: convert parameter changes directly to loop updates.
Head Recursion	Recursive call occurs *before* processing operations.	Output operations execute in reverse stack order (unwinding phase).	Reverses processing sequence (e.g., printing digits of a number backwards/forwards).
Tree Recursion	Multiple recursive calls per function invocation.	Exponential stack frames $O(2^n)$.	Draw call tree; identify overlapping subproblems or solve bottom-up using recurrence relations.
Nested Recursion	Recursive call passed as argument to another recursive call: `fun(fun(n-1))`.	Rapidly escalating stack depth.	Trace manually from inside call outward; base case values lock sequence.