Sheet 4: Recursion with Statement *Between* Recursive Calls (Middle-Call Pattern)

Focus: Used in **divide-merge**, **in-order traversal**, or **sorting algorithms**.

MCQ 1: In-Order Print

```
function process(n):
    if n == 0:
        return
    process(n - 1)
    print(n)
    process(n - 1)

process(2)
```

Output?
A) `1 2 1`
B) `2 1 1`
C) `1 1 2`
D) `1 2 2`

Learning: Middle call = In-order traversal behavior.

MCQ 2: Binary Divide and Merge

```
function divide(n):
    if n <= 1:
        print(n)
        return
    divide(n // 2)
    print(n)
    divide(n // 2)

divide(4)
```

Output?
A) `1 2 4 2 1`
B) `1 4 1`
C) `1 1 2 4 2 1 1`
D) `2 4 2`

MCQ 3: Middle of Two Recursive Calls

```
function middle(n):
    if n == 0:
        return
    middle(n - 1)
    print(n)
    middle(n - 2)

middle(3)
```

Output?
A) `1 2 3 1`
B) `1 2 1 3`
C) `1 2 3`
D) `3 2 1`

MCQ 4: In-Order Tree Walk Analogy

```
function walk(n):
    if n == 0:
        return
    walk(n - 1)
    print("node", n)
    walk(n - 1)

walk(2)
```

Output?
A) `node 2 node 1 node 1`
B) `node 1 node 2 node 1`
C) `node 1 node 1 node 2`
D) `node 1 node 2`

MCQ 5: Merge Sort Style

```
function mergeStep(n):
    if n == 1:
        print(n)
        return
    mergeStep(n // 2)
    print(n)
    mergeStep(n // 2)

mergeStep(4)
```

Output?
A) `1 2 4 2 1`
B) `1 4 1`
C) `4 2 1`
D) `1 2 1 4 1 2 1`

Summary

Sheet No.	Category	Key Use Case
Sheet 1	**Post-recursion (Unwinding)**	Reverse operations, post-order traversal
Sheet 2	**Pre-recursion (Winding)**	Top-down logic, pre-order traversal
Sheet 3	**Tail Recursion**	Accumulator-based, space-efficient
Sheet 4	**Middle Recursive Call**	In-order logic, divide-and-conquer algorithms