Exercise walk-through

Recursion ▪ 19.2

eXercise

You must be able to

- explain **recursion** (base case + recursive case)
- **trace** a recursive function
- explain what the **compiler / call stack** does for each call

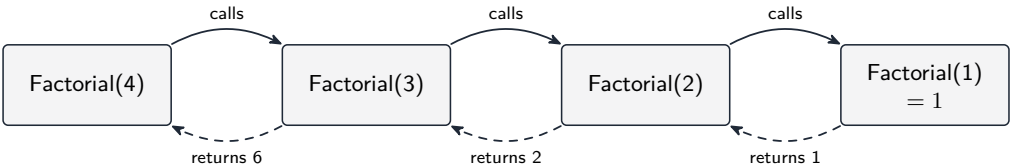
Method — What recursion is

A **recursive** function calls **itself** with a smaller input, until a **base case** stops it:

```
FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
    IF n = 0 OR n = 1 THEN
        RETURN 1                // base case
    ELSE
        RETURN n * Factorial(n - 1)   // recursive case
    ENDIF
ENDFUNCTION
```

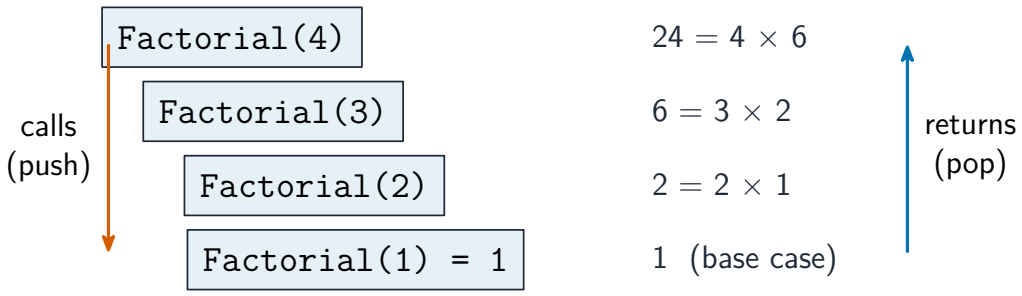
Method — Tracing

`Factorial(4)` calls down to the base case, then the returns **unwind** back up:



result: 24

Method — Tracing



Recursion builds up a call stack

Method — The call stack

The compiler gives every call its own **stack frame** holding its **parameters**, **local variables** and **return address**. On return, the value is passed back, the frame is **popped**, and execution resumes at the return address — so calls don't overwrite each other's variables. Missing the base case → **infinite recursion** → **stack overflow**.

Method — Recursion vs iteration

Any recursive routine can be rewritten as an **iterative** one (a loop, plus your own stack if needed). Iteration usually uses **less memory** (no stack frames) and runs faster; recursion is often **shorter and clearer** for self-similar problems (trees, divide-and-conquer).

Practice 2.1 ▪ 1 mark

State what a **base case** is.

Solution 2.1

Worked steps

The **simplest case**, solved **directly without** another recursive call — it **stops** the recursion B1.

Practice 2.2 ▪ 1 mark

State what the **recursive case** does.

Solution 2.2

Method: What recursion is

Worked steps

Reduces the input and **calls the function again** (on the smaller problem) B1.

Practice 2.3 ▪ 1 mark

State what happens if a recursive function has **no base case**.

Solution 2.3

Method: What recursion is

Worked steps

The recursion **never stops** — it runs until the **call stack overflows** and the program crashes B1.

Practice 2.4 ▪ 2 marks

State **two** things each **stack frame** holds.

Solution 2.4

Method: The call stack

Worked steps

Any two: the **parameters**; the **local variables**; the **return address** B1 B1.

Exam-style 3.1 ▪ 2 marks

Describe what is meant by **recursion**.

Solution 3.1

Worked steps

A technique where a function **calls itself** B1 with a **smaller version** of the problem until a **base case** is reached B1.

Exam-style 3.2 ▪ 3 marks

Trace `Factorial(4)`. State the value **returned** at each level of the recursion.

Solution 3.2

Method: Tracing

Worked steps

`Factorial(1)` returns **1**; `Factorial(2)` returns **2**; `Factorial(3)` returns **6**; `Factorial(4)` returns **24** B1 B1 B1 *for base case = 1, for the multiplications, for final 24.*

Exam-style 3.3 ▪ 3 marks

Explain what happens on the **call stack** each time a recursive function is called.

Solution 3.3

Worked steps

A new **stack frame** is **pushed**, holding that call's parameters, local variables and return address B1; the call runs, and on return its **value is passed back** and the frame is **popped** B1; execution resumes at the **return address** in the caller B1.

Exam-style 3.4 ▪ 2 marks

Give **one** feature that makes a problem suitable for recursion, and **one** example application.

Solution 3.4

Worked steps

Feature: the problem is **self-similar** / can be broken into smaller versions of itself B1. Example: traversing a **tree**, binary search, merge sort, factorial/Fibonacci B1.

Exam-style 3.5 ▪ 2 marks

State **one** advantage and **one** disadvantage of writing a routine **recursively** instead of **iteratively**.

Solution 3.5

Worked steps

Advantage: **shorter, clearer** code for self-similar problems B1. Disadvantage: it uses **more memory** (a stack frame per call) and can cause a **stack overflow**, and is often slower B1.