A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

19.2 Recursion

Name: _____ Class: _____ Date: _____ **Total: 17 marks**

KEY WORDS recursion 递归 • call stack 调用栈 • stack frame 栈帧 • recursive case 递归情形 • stack overflow 栈溢出

Objective

Build the skills to answer exam questions on **recursion** 递归 — base/recursive cases, tracing, and how the compiler uses the call stack.

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

1 Worked examples

Study these first. Each one shows the method for a question type used later — follow the steps and you can do the Practice and Exam-style questions yourself.

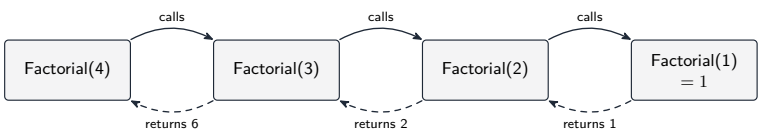
What recursion is

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

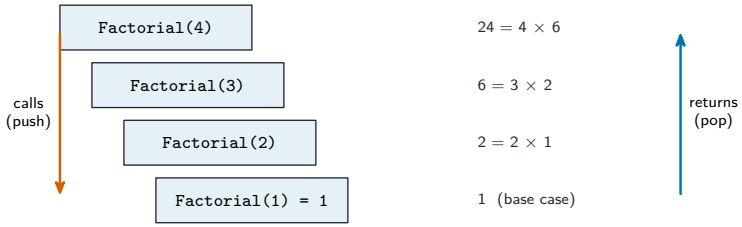
```
FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
    IF n = 0 OR n = 1 THEN
        RETURN 1               // base case
    ELSE
        RETURN n * Factorial(n - 1)   // recursive case
    ENDIF
ENDFUNCTION
```

Tracing

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



result: 24



Recursion builds up a call stack

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**.

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).

2 Practice

Now apply the methods above.

2.1 State what a **base case** is. [1]

2.2 State what the **recursive case** does. [1]

2.3 State what happens if a recursive function has **no base case**. [1]

2.4 State **two** things each **stack frame** holds. [2]

3 Exam-style questions

3.1 Describe what is meant by **recursion**. [2]

3.2 Trace `Factorial(4)`. State the value **returned** at each level of the recursion. [3]

3.3 Explain what happens on the **call stack** each time a recursive function is called. [3]

3.4 Give **one** feature that makes a problem suitable for recursion, and **one** example application. [2]

3.5 State **one** advantage and **one** disadvantage of writing a routine **recursively** instead of **iteratively**. [2]

Solutions

Each answer shows the steps, then the **result**.

2.1

- the **simplest case**, solved **directly without** another recursive call — it **stops** the recursion

2.2

- **reduces** the input and **calls the function again** (on the smaller problem)

2.3

- the recursion **never stops** — it runs until the **call stack overflows** and the program crashes

2.4

- any two: the **parameters**; the **local variables**; the **return address**

3.1

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

3.2

- `Factorial(1)` returns **1**; `Factorial(2)` returns **2**; `Factorial(3)` returns **6**; `Factorial(4)` returns **24**

3.3

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

3.4

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

3.5

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