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