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

3.6

(a)

`Index`	`Target`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`	`[6]`	`[7]`
1	12	4	9	12	18	25	30	0
2	12							
3	12			18				
4	12				25			
5	12					30		
6	12						0	
7	12							

- `Index` and `Target` correct throughout
- `[1]` and `[2]` unchanged, because 4 and 9 are less than 12
- `[3]` becomes 18
- `[4]` becomes 25 and `[5]` becomes 30
- `[6]` becomes 0; at `Index` 7, `Numbers[7]` is 0, so the recursion **ends**

(b)

- it **deletes** the first value that is **greater than or equal to** `Target` (here 12), moving the later values **left** to close the gap

(c)

- no element is `0`, so the **base case is never reached** before the end of the array
- `Numbers[Index + 1]` is read **beyond the upper bound**: an **index out of range** error / the recursion would not stop

3.7

(a)

```
FUNCTION SumScores(N : INTEGER) RETURNS INTEGER
    IF N = 0 THEN
        RETURN 0
    ELSE
        RETURN Scores[N] + SumScores(N - 1)
    ENDIF
ENDFUNCTION
```

- function header with parameter `N` and `RETURNS INTEGER`
- base case: `IF N = 0 THEN RETURN 0`
- recursive call with `N - 1`
- adds `Scores[N]` to the value returned

(b)

- **4** frames: `SumScores(3)`, `SumScores(2)`, `SumScores(1)` and `SumScores(0)`

- each call **waits** on the stack until the call it made returns, so all four exist when the base case runs