4.1.1 Recursion

What is Recursion?

Recursion is a programming technique where **a function calls itself** to solve a problem.

Each time the recursive call works on a **smaller version of the original problem**, until it reaches a stopping point called a **base case** or **terminating case**.

Every recursive function must have **two essential parts**:

1. **Base case**
    - The condition that stops the recursion.
    - Prevents the function from calling itself forever.
2. **Recursive case (general case)**
    - The function calls itself with a smaller or simpler version of the problem.
    - Moves the function closer to the base case.

Without a base case, the program would continue calling the function until it runs out of memory, causing a **stack overflow**.

Advantages and Disadvantages of Recursion

Advantages

- Produces short, elegant code for some problems.
- Natural solution for problems that are defined recursively.
- Useful for tree traversal, graph algorithms, and divide-and-conquer algorithms (such as merge sort).

Disadvantages

- Uses more memory because each function call is stored on the call stack.
- Can be slower than an equivalent iterative solution because of the overhead of repeated function calls.
- Can cause a stack overflow if there is no base case or if the recursion is too deep.
- Some problems are simpler to solve using loops.

Example 1: Countdown Trace

```
def countdown(number):
    if number == 0:          # Base case
        print("Blast off!")
    else:                    # Recursive case
        print(number)
        countdown(number - 1)

countdown(3)
```

How it works for `countdown(3)`

`countdown(3)`
↓
`countdown(2)`
↓
`countdown(1)`
↓
`countdown(0)`

At `countdown(0)`, the **base case** is reached where *"blast off!"* is printed and the recursion stops.

Example 2: Factorials Trace

The factorial of a number is the product of all positive integers from that number down to 1.

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

```python
def factorial(n):
    if n == 1:                # Base case
        return 1
    else:                     # Recursive case
        return n * factorial(n - 1)

print(factorial(5))
```

How it works for `factorial(5)`

`factorial(5)`

= 5 × `factorial(4)`

= 5 × 4 × `factorial(3)`

= 5 × 4 × 3 × `factorial(2)`

= 5 × 4 × 3 × 2 × `factorial(1)`

= 5 × 4 × 3 × 2 × 1

= 120

Each function must wait for the next one to finish before it can calculate the answer and return.

- `factorial(5)` waits for `factorial(4)` to return.
- `factorial(4)` waits for `factorial(3)` to return.
- `factorial(3)` waits for `factorial(2)` to return.
- `factorial(2)` waits for `factorial(1)` to return.
- `factorial(1)` triggers a base case and returns 1 back up to `factorial(2)`.
- `factorial(2)` calculates 2 × 1 and returns 2 back up to `factorial(3)`.
- `factorial(3)` calculates 3 × 2 and returns 6 back up to `factorial(4)`.
- `factorial(4)` calculates 4 × 6 and returns 24 back up to `factorial(5)`.
- `factorial(5)` calculates 5 × 24 and returns 120.

Once the base case returns, **the results cascade back up through the waiting function calls** until the original function call returns its final result.