Learning Aims

Recursion

- Explain how recursive functions works, their good and bad points
- Compare recursive and iterative solutions to a problem
- Trace recursive functions

Recursion

- Recursion happens when a **function calls itself**.
- It's a *divide and conquer* method — breaking a big problem into smaller ones of the same type, solving them, and combining the results.
- It can make code **shorter and more elegant.**
- Each recursive call creates a new copy of the function with its own variables.
- **However**, recursion can use **a lot of memory** because each call stays in the call stack until it finishes.
- A recursive function has two main parts:
  - **Recursive part:** where the function calls itself.
  - **Base case:** where the function stops calling itself.
- Recursion is often used for problems like **factorials**, **Fibonacci sequences**, **tree traversal**, and **search algorithms**.

Countdown using Iteration

```
def countdown_iterative(n):
    while n > 0:
        print(n)
        n = n - 1
    print("Blast off!")

# Example usage
countdown_iterative(5)
```

```
5
4
3
2
1
Blast off!
```

Uses loops (for, while) to repeat actions until a condition fails.

The same block of code executes repeatedly

Uses a **loop condition** (while n > 0) to stop iteration.

Trace Table (n = 3)

n	Output
3	print(3)
2	print(2)
1	print(1)
0	print("Blast off!")

Countdown

```
def countdown(n):
    if n <= 0:
        print("Blast off!")
    else:
        print(n)
        countdown(n - 1)

# Example usage
countdown(5)
```

Recursive routines have an if statement (not a while) to specify the terminating condition.

A function that calls itself until a base condition is met.

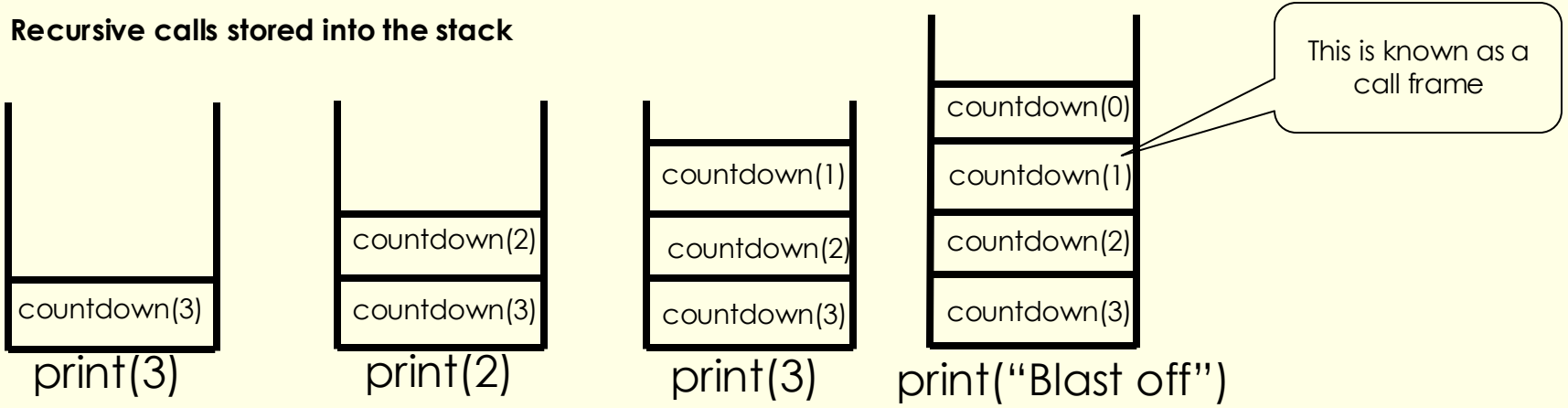
Function calls itself multiple times — each call is added to the **call stack**.

Uses a **base case** (if n <= 0) to stop recursion.

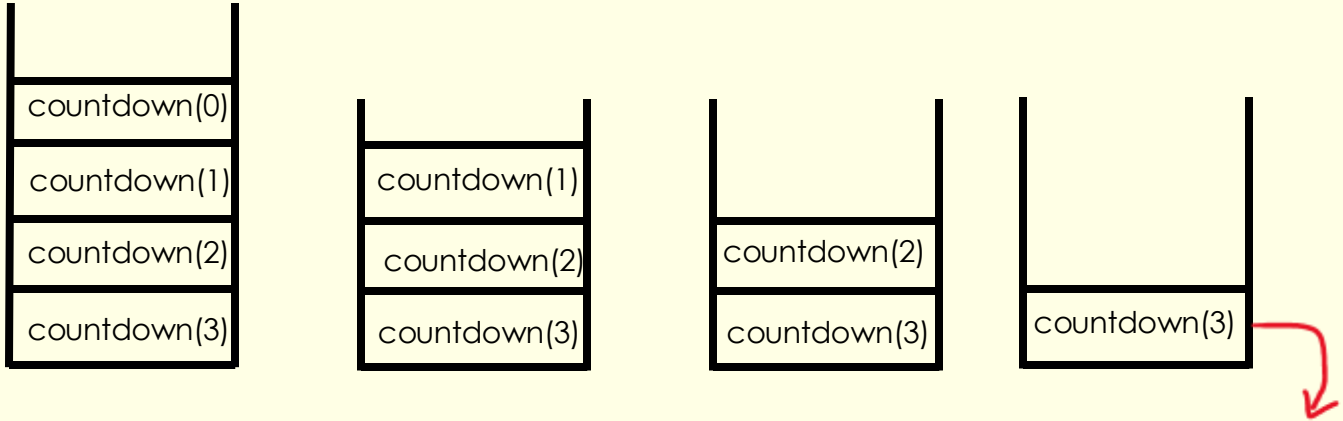
Call	n value	Output	Return
countdown(3)	3	print(3)	
countdown(2)	2	print(2)	
countdown(1)	1	print(1)	
countdown(0)	0	print("Blast off!")	
return	—	—	Pop all recursive calls from the stack

Using stacks

Recursive calls stored into the stack



Returning values once base case is met



An example is a procedure to produce a list of the square numbers. The procedure will be passed two parameters stating what the largest and smallest squares should be.

This can be written non-recursively using a while loop as follows:

```
PROCEDURE Squares(Low, High)
    Count = Low
    WHILE Count <= High
        PRINT (Count * Count)
        Count = Count + 1
    ENDWHILE
ENDPROC
```

This could be written as a recursive procedure as follows:

```
PROCEDURE Squares(Low, High)
    IF Low <= High
        PRINT (Low * Low)
        Squares(Low+1,High)
    ENDIF
ENDPROC
```

Recursive routines have an if statement (not a while) to specify the terminating condition.
The recursive call asks that Squares be executed again with new parameters.

Why less memory efficient

- Recursion is **less memory efficient** because each function call is stored on a stack until the base case is reached.
- If there are too many recursive calls, it can cause a **stack overflow** (the program runs out of memory and crashes).
- It must remember all variable values for each call, which uses extra memory.
- When a recursive function runs, **each function call is stored on the call stack**, and the program must remember the value of its variables (like n)
- This **uses extra memory** compared to iteration, because iteration reuses the **same variables** in a single loop rather than creating a new stack frame each time.

So when is recursion is better…

- In the countdown example, **recursion stops at the base case** and uses a **stack** to keep track of each call.
- For the countdown problem, **iteration is better** because it's simpler and uses less memory.
- So when is recursion better?

Recursion is better when:

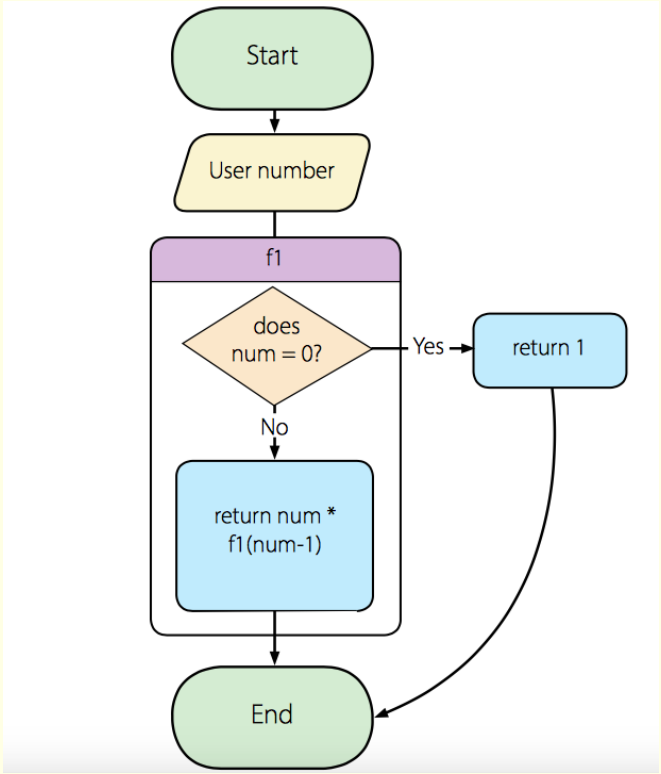
- A problem can be **broken into smaller subproblems** of the same type.
- The structure is **naturally recursive**, like trees or linked lists.
- It makes the code **shorter and easier to understand** than a complex loop.

Examples:

- Calculating a **factorial**
- Finding a **Fibonacci sequence**
- Traversing **trees or directories**

Factorial Finder

- The Factorial of a positive integer, n, is defined as the product of the sequence n, n-1, n-2,
- The factorial of a number is the product of all the integers from 1 to that number.
- For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.
- Factorial is not defined for negative numbers and the factorial of zero is one, 0! = 1



Comparison

Recursive method

```
function fact(num)
    if num == 0: #base case
        return 1
    return num * fact(number-1)
```

Iteration method

```
function fact(num)
    fact = 1
    for i = 1 to num
        fact = fact * i
    return fact
```

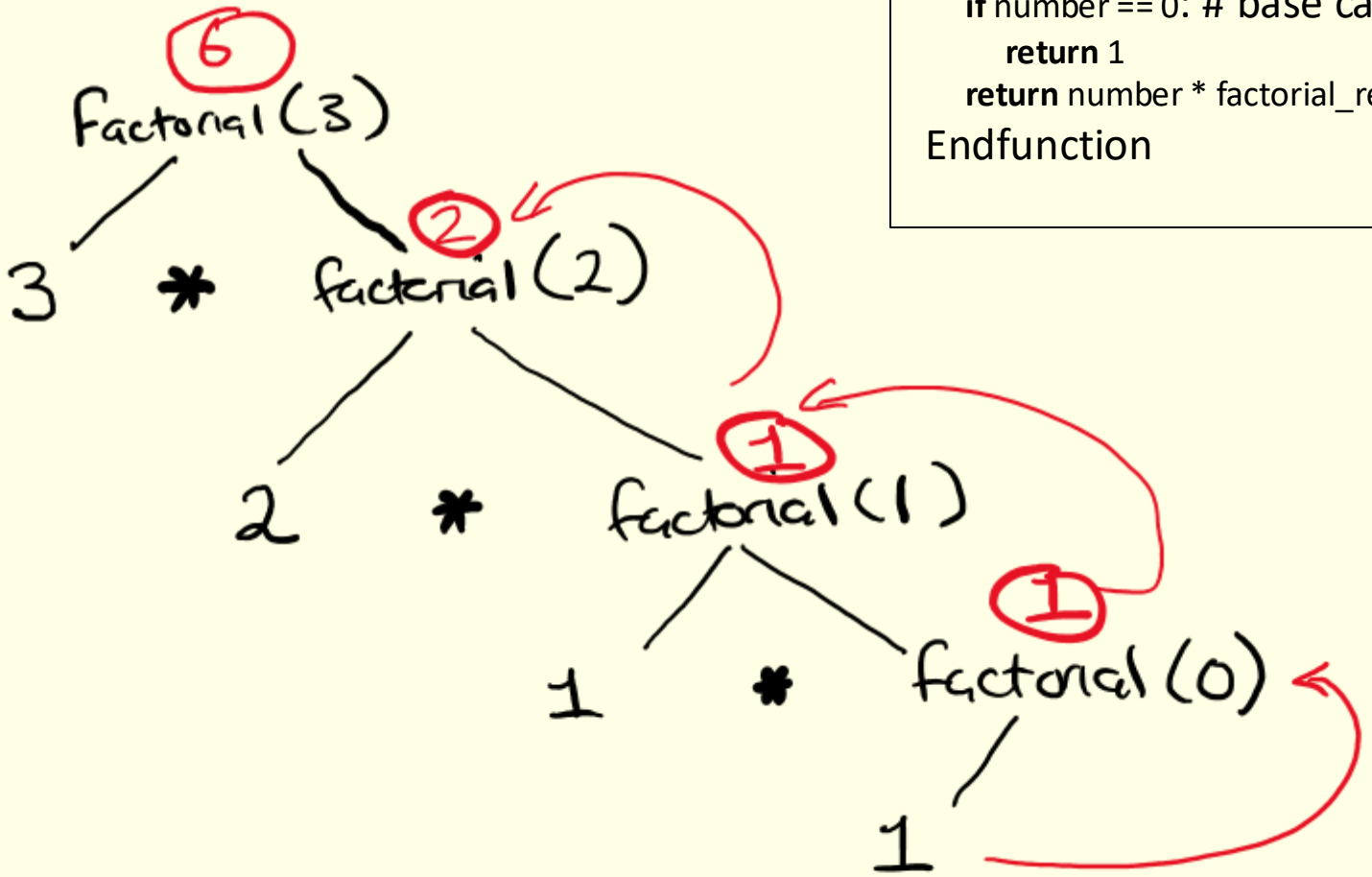
Recursive routines have an if statement (not a while) to specify the terminating condition.

Tracing the factorial recursive algorithm

Function Call	**n**	**Return / Explanation**
fact(3)	3	fact(3) = 3 * fact(2) → 3 * 2 = 6
fact(2)	2	fact(2) = 2 * fact(1) → 2 * 1 = 2
fact(1)	1	fact(1) = 1 * fact(0) → 1 * 1 = 1
fact(0)	0	Base case → return 1

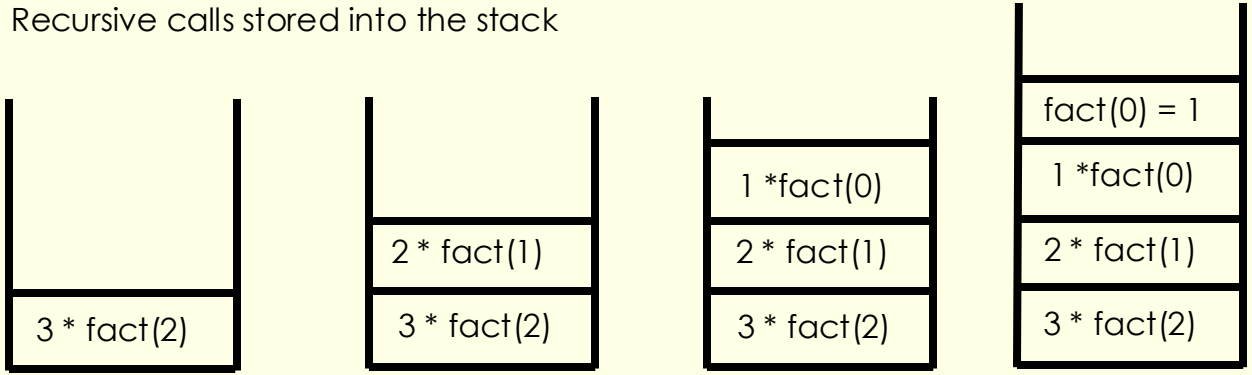
Recursive Tree

```
function factorial_recursion(number):
    if number == 0: # base case
        return 1
    return number * factorial_recursion(number-1)
Endfunction
```

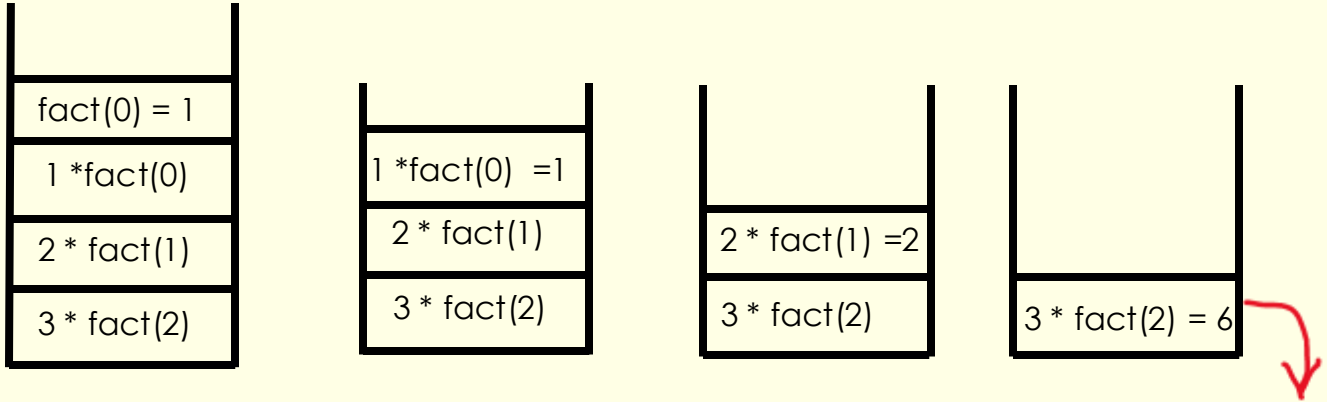


Stack Diagram

Recursive calls stored into the stack



Returning values once base case is met



Why Use Recursion for Factorial?

- **Matches the math definition:**
  Factorial is defined recursively — n! = n × (n-1)! — so the code mirrors the formula.
- **Simpler and easier to read:**
  Recursive code is shorter and clearly shows the problem being broken into smaller parts.
- **Great for learning:**
  Factorial is an easy example to understand how recursion works with base cases and the call stack.

- *Note:* Iteration is faster and uses less memory, but recursion is clearer for showing the concept.

Comparison

Recursion	**Iteration**
more memory (each call adds a new stack frame)	less memory (same variables reused)
Declares new variables//variables are put onto the stack each time	iteration reuses the same variables in the loop
Can run out of memory/stack space	while iteration cannot run out of memory
Express a problem more elegantly in fewer lines of code	Often needs **more lines** and explicit loop control.
Slower, because each call has overhead (function call setup).	**Faster**, because it's a simple loop with no call overhead.
Be harder to follow/ trace/debug	**Easier to follow** and debug step-by-step.
Needs a **base case** to stop recursion.	Needs a **loop condition** to stop iteration
Recursion will be self-referential // will call itself	Uses **loop constructs** (for, while), not self-calling.

Summary

Advantages of Recursion

-Recursive functions make the code look clean and elegant.
-A complex task can be broken down into simpler sub-problems using recursion.
-Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

-Sometimes the logic behind recursion is hard to follow through.
-Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
-Recursive functions are hard to debug.