



# Cambridge (CIE) A Level Computer Science



Your notes

## Recursion

### Contents

\* Recursion



## Features of recursion

### What is recursion?

- Recursion is a highly effective programming technique where a **function calls** itself to **solve a problem** or **execute a task**
- Recursion doesn't rely on iterative loops
- Instead, it uses the idea of **self-reference** to **break down** complicated problems into **more manageable** subproblems
- A recursive algorithm has three features:
  - the function must **call itself**
  - a **base case** - this means that it can return a value without further recursive calls
  - a **stopping base** - this must be reachable after a finite number of times

### How does recursion work?

- In a recursive function, **the function calls itself with a modified input parameter** until it reaches a base case — a condition that stops the recursion and provides the final result
- Each recursive call breaks down the problem into more minor instances until it reaches the base case

### Example: factorial calculation

```
def factorial(n):
```

```
# Base case
```

```
if n == 0 or n == 1:
```

```
    return 1
```

```
else:
```

```
    # Recursive call with a smaller instance of the problem
```

```
    return n * factorial(n - 1)
```

```
result = factorial(5)
```

```
print(result)
```

```
# Output: 120 (5! = 5 * 4 * 3 * 2 * 1)
```

- In this example, the **factorial** function calculates the **factorial** of a positive integer **n**
- It does so by breaking down the problem into smaller instances, multiplying **n** with the factorial of **(n - 1)** until it reaches the base case (**n == 0** or **n == 1**)



## Importance of a proper stopping condition

- It is important to have a proper **stopping condition or base case** when using recursion to avoid **stack overflow** errors which result in program crashes
- If a recursive function does not have a stopping condition, it will continue to call itself indefinitely, which can use up excessive **memory** and cause the program to malfunction

## Designing a stopping condition

- When creating a stopping condition, it's important to consider the problem being solved
- Identify the easiest scenario where the function can provide a direct result. This scenario should be defined as the base case, covering the simplest instances of the problem
- By doing so, the function will be able to stop the recursion when those conditions are met
- The difference between line 7 and the function declaration on line 1, is that num1 is replaced with result + 1 so we'll need to set num1 equal to result + 1

## Recursion: Benefits & drawbacks

- Programs can be written using either recursion or **iteration** - which one is used will depend on the problem being solved
- There are many benefits and drawbacks to using either, depending on the situation:

### Recursion

| Benefits                                                                                                                           | Drawbacks                                                                                                 |
|------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| <b>Concise</b> - can often be expressed in a more concise way, especially for structures like trees or fractals                    | <b>Performance</b> - repeated function calls can be CPU and Memory intensive, leading to slower execution |
| <b>Simple</b> - simply stating what needs to be done without having to focus on the how can make it more readable and maintainable | <b>Debugging</b> - recursive code can be much more difficult to track the state of the program            |
|                                                                                                                                    | <b>Limited application</b> - not all problems are suited to recursive solutions                           |

### Iteration

| Benefits                                                               | Drawbacks                                                                                       |
|------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| <b>Performance</b> - more efficient than recursion, less memory usage. | <b>Complexity</b> - can get very complex and use more lines of code than recursive alternatives |

|                                                                       |                                                                                            |
|-----------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| <b>Debugging</b> – easier to understand and debug                     | <b>Less concise</b> – compared to recursive alternatives, making them harder to understand |
| <b>Wider application</b> – more suitable to a wider range of problems |                                                                                            |



Your notes

# Writing recursive algorithms

- Here is a **recursive algorithm** for a simple countdown program written in **Python** to countdown from 10 to 0

## Step 1

- Create the **subroutine** (in this example it will be a **function** as it will **return a value**) and identify any **parameters**

def countdown\_rec(n): #n is the parameter passed when we call the subroutine

## Step 2

- Create a **stopping condition** – when n is 0 the function will stop

```
def countdown_rec(n):  
  
    print(n) #output the starting number  
    if n == 0: #stopping condition  
        return
```

## Step 3

- Add a **recursive function call**

```
def countdown_rec(n):  
    print(n)  
    if n == 0:  
        return  
    countdown_rec(n - 1) #recursive functional call
```

## Step 4

- Call the **function**

```
def countdown_rec(n):  
    print(n)  
    if n == 0:  
        return  
    countdown_rec(n - 1)
```

countdown\_rec(10) #call the function and pass 10 as a starting value

## Output to user

10  
9  
8

7  
6  
5  
4  
3  
2  
1  
0

## Tracing recursive algorithms

- Now lets **trace** the **recursive algorithm** we have just written to check what happens during the execution of the program
- Here is the completed program and we are going to start it using the command `countdown_rec(5)`

```
def countdown_rec(n):  
    print(n)  
    if n == 0:  
        return  
    countdown_rec(n - 1)
```

- Using a simple **trace table** we can trace the **recursive** function call

| Function call    | print(n) | countdown_rec(n - 1) |
|------------------|----------|----------------------|
| countdown_rec(5) | 5        | 4                    |
| countdown_rec(4) | 4        | 3                    |
| countdown_rec(3) | 3        | 2                    |
| countdown_rec(2) | 2        | 1                    |
| countdown_rec(1) | 1        | 0 (return)           |

## Translate between iteration & recursion

- Recursive** algorithms can be **translated** to use **iteration**, and **vice versa**
- Let's look at the previous example **recursive** program and see how it would change to **solve the same problem** but using an **iterative** approach

### Recursive approach

```
01 def countdown_rec(n):  
02     print(n)  
03     if n == 0:  
04         return
```

```
05 countdown_rec(n-1)
06 countdown_rec(10)
```

### Iterative approach

```
01 def countdown_rec(n):
02     while n > 0:
03         print(n)
04         n = n-1
05     return
06 countdown_rec(10)
```

- The **recursive function** call on line **05** has been replaced with a **while loop** (line **02**) which checks if  $n > 0$
- Using an **iterative** approach we use exactly the same amount of code (6 lines) **BUT...**
  - less memory would be used (**increased performance**)
  - **easier to debug**

## Compilation & stack

### What happens when recursion runs?

- When a recursive function is called, the **compiler (or interpreter)** doesn't treat it like a loop
- Instead, it uses a special structure called the **call stack** to keep track of each function call

### The call stack

- Each time a function calls itself, the system **stores a snapshot of that call** on the **call stack**
- This snapshot (called a **stack frame**) contains:
  - The function name
  - The value of parameters and variables at that level
  - The place to return to when the function finishes
- This continues until the **base case is reached**

### Stack frames and unwinding

- Once the base case is reached, no more function calls are made
- At this point, the stack **unwinds**, which means:
  - The **most recent call** completes and **returns a value**
  - Control goes back to the **previous stack frame**
  - This continues until the original call receives the final result



Your notes



- This is called **stack unwinding**

## Example: factorial(3)

- Let's trace the function **factorial(3)** using a call stack

```
def factorial(n):  
    if n == 0 or n == 1:  
        return 1  
    return n * factorial(n - 1)
```

## Call stack trace:

factorial(3) → 3 \* factorial(2)  
factorial(2) → 2 \* factorial(1)  
factorial(1) → base case → returns 1

Unwinding:

factorial(2) = 2 \* 1 = 2  
factorial(3) = 3 \* 2 = 6

- At the deepest point, the stack held:
  - **factorial(3)**
  - **factorial(2)**
  - **factorial(1)**
- Each was popped off (unwound) as the return values passed back up

## Why understanding the call stack matters

- If **no base case** is present, the stack **never stops growing**
- This leads to a **stack overflow**, causing the program to crash
- Recursive programs should always be designed to **reach the base case** in a finite number of steps

## Summary

| Concept        | Description                                                               |
|----------------|---------------------------------------------------------------------------|
| Call stack     | Structure used by the compiler to store function calls                    |
| Stack frame    | Snapshot of a single function call (with local variables and return info) |
| Unwinding      | The process of returning values back up the stack after reaching the base |
| Stack overflow | Happens when recursive calls never stop, exhausting available memory      |