Lesson 12a: Recursion

Introduction

Recursion

- Recursion occurs when something is defined in terms of itself
- Recursive function: when a function calls to itself

Recursion

Semantically: it is a programming technique in which a function calls to itself

Algorithmically: it is a way of designing solutions to divide-and-conquer problems. This is achieved by reducing the problem into simpler versions.

- It can be used instead of **iteration**

Reasons of use:

- For those "almost" unsolvable problems with iterative structures.
- Elegant solutions.
- Simpler solutions.

Conditions of the problem so that recursion can be applied:

- The problem can be solved from the same problem, but with other input parameters, so that the initial problem is simplified.
- The problem must have 1 or more base cases that are easy to solve.

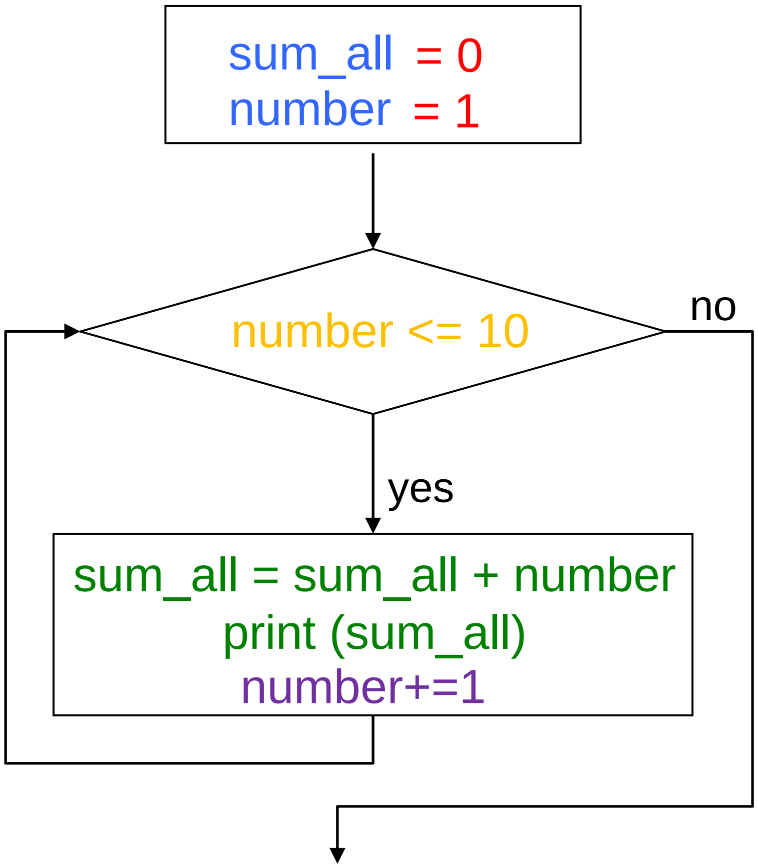
Requirements of the computer (Hardware):

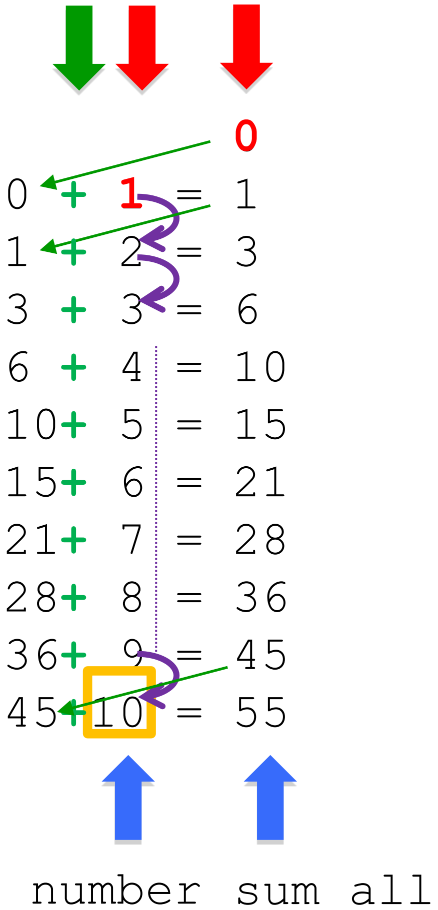
- Availability of RAM memory.

Until now ... Iterative algorithms

Repetitive instructions (while and for) allows us to develop iterative algorithms

The result is calculated using a set of "state variables" that are updated at each iteration of the loop





Recursive algorithms: summation

We express the summation in mathematical form:

$$Summ(x) = \sum_{1}^{n} x = n + \sum_{1}^{n-1} x = \ ...$$

We can define it as follows:

$$Summ(x) = \begin{cases} 1 & \text{if } \ x = 1 \\ Summ(x) = x + Summ(x-1) & \text{if } \ x > 1 \end{cases}$$

Base case

Algorithm:

```
def S(x):
"""
Summation computed recursively
"""
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x
```

Base case

Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```

Scope of the main program

x	4

Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```

Scope of the main program

x	4

Scope S(4)

x	4

Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```

Scope of the main program		Scope S(4)	
x	4	x	4

Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```

Scope of the main program	Scope S(4)	Scope S(3)
x 4	x 4	x 3

Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```

Scope of the main program		Scope S(4)		Scope S(3)	
x	4	x	4	x	3

Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```

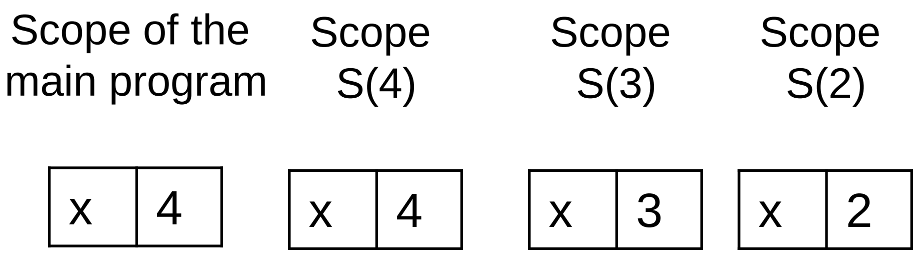
Scope of the main program		Scope S(4)		Scope S(3)		Scope S(2)	
x	4	x	4	x	3	x	2

Scope of the variables

Summation

```python
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```



Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```

Scope of the main program	Scope S(4)	Scope S(3)	Scope S(2)	Scope S(1)
x | 4	x | 4	x | 3	x | 2	x | 1

Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```

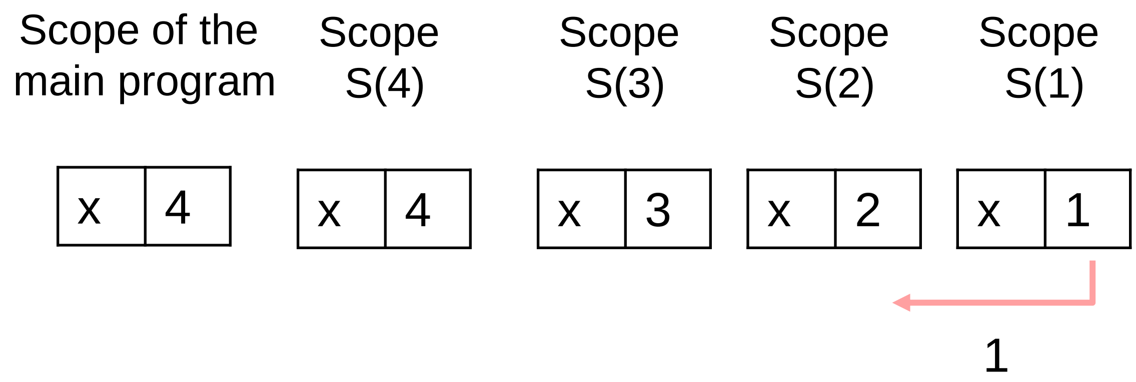
Scope of the main program	Scope S(4)	Scope S(3)	Scope S(2)	Scope S(1)
x 4	x 4	x 3	x 2	x 1

Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```



Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```

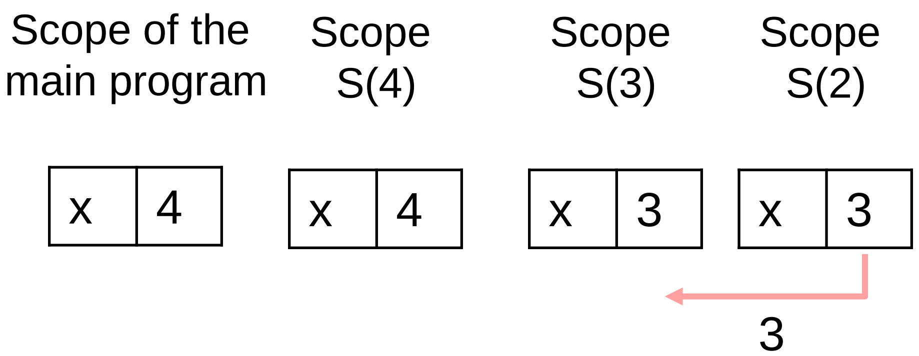
Scope of the main program	Scope S(4)	Scope S(3)	Scope S(2)
x 4	x 4	x 3	x 3

Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```



Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```

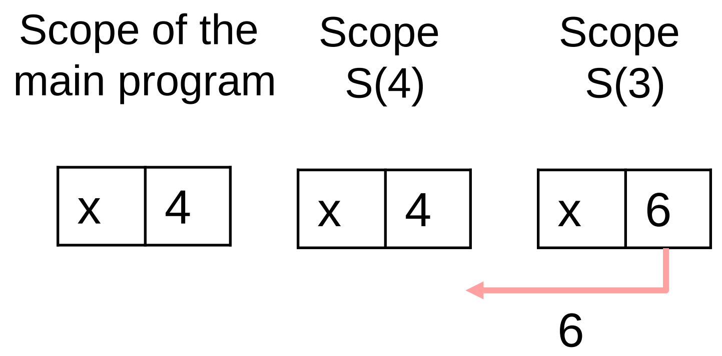
Scope of the main program	Scope S(4)	Scope S(3)
x | 4	x | 4	x | 6

Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```



Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```

Scope of the main program

x	4

Scope S(4)

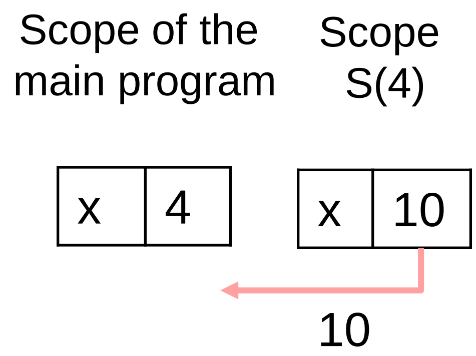
x	10

Scope of the variables

Summation

```
def S(x):
    if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```



Scope of the variables

Summation

```
def S(x):
if (x == 1):
        x = 1
    else
        x = x + S(x-1)
    return x

x = 4
print(S(x))
```

Scope of the main program

x	4

```
In [70]: S(4)
Out[70]: 10

In [71]:
```

Important:

- Each recursive call to a function creates its own scope / environment
- The variables in an environment do not change due to the recursive call
- The control flow moves to the previous environment once the function returns the value.

Recursion vs Iteration

Summation

```
def S_iter(x):
    sum = 0
    for i in range(1,x+1):
        sum += i
    return sum
```

```
def S_recursive(x):

    if (x == 1):
        x = 1
    else
        x = x + S_recursive(x-1)
    return x
```

Differences:

- Recursion can be simpler: it's more intuitive.
- Recursion can be efficient from a programmer's point of view.
- Recursion may not be computer efficient.

Exercise: Factorial

Write the function **Factorial** of an integer **n**. Implement 2 versions:

- Iterative version
- Recursive version

Exercise: Factorial

Iterative Version

```
def factorial(n):

    fac = 1
    while (n > 0):

        fac = fac * n
        n = n - 1

    return fac
```

Exercise: Factorial

Recursive Version

- Basic case: for what value can we give a direct solution of the factorial?
  - If n = 0, because 0! = 1

- General case:
  - **n> 0**
  - Suppose we know how to calculate (n-1)!
  - How can we calculate n! from (n-1)! ?

  - 3! = 3 * 2 * 1
  - 4! = 4 * 3 * 2 * 1
  - 4! = 4 * 3!
  - …
  - n! = n * (n-1)!

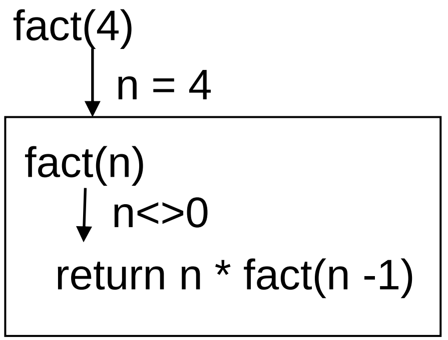
Exercise: Factorial

- Base case: if n = 0: 0! = 1
- General case: if n > 0: n! = n * (n-1)!

```
def fact(n):
    if (n == 0): # Base case
        return 1

    # General case
    return (n * fact(n-1))
```

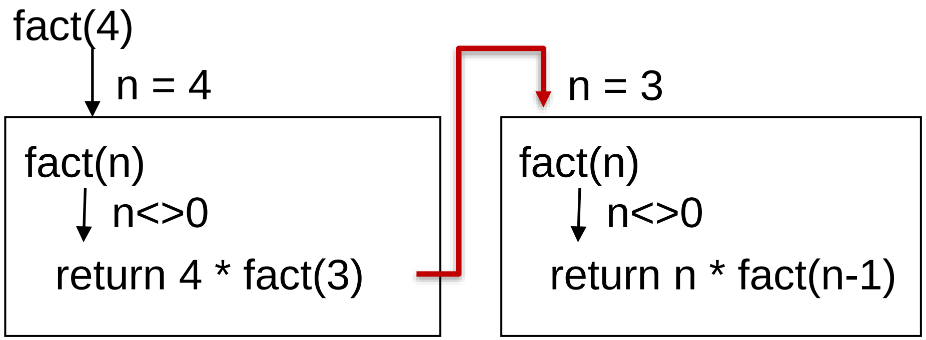
Exercise: Factorial




```python
def fact(n):
  if (n == 0):
    return 1

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

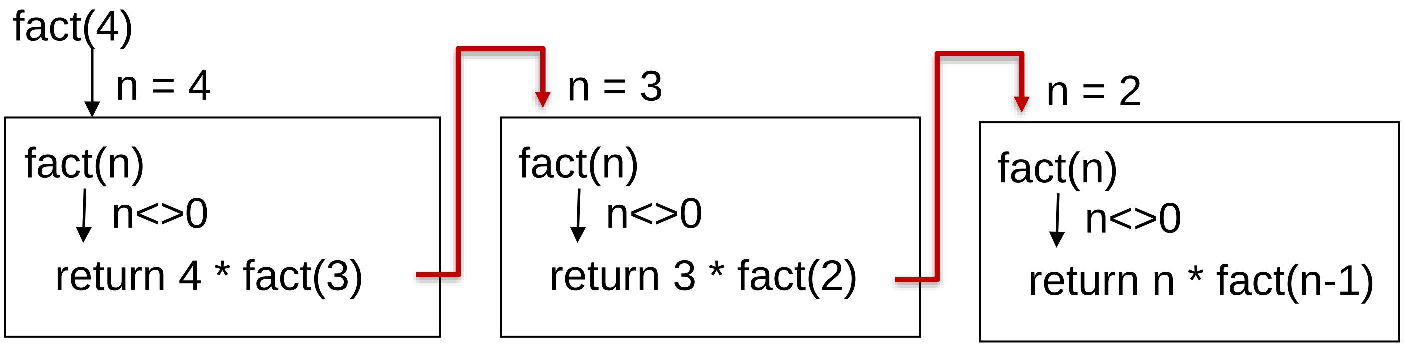
Exercise: Factorial




```
def fact(n):
  if (n == 0):
    return 1

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

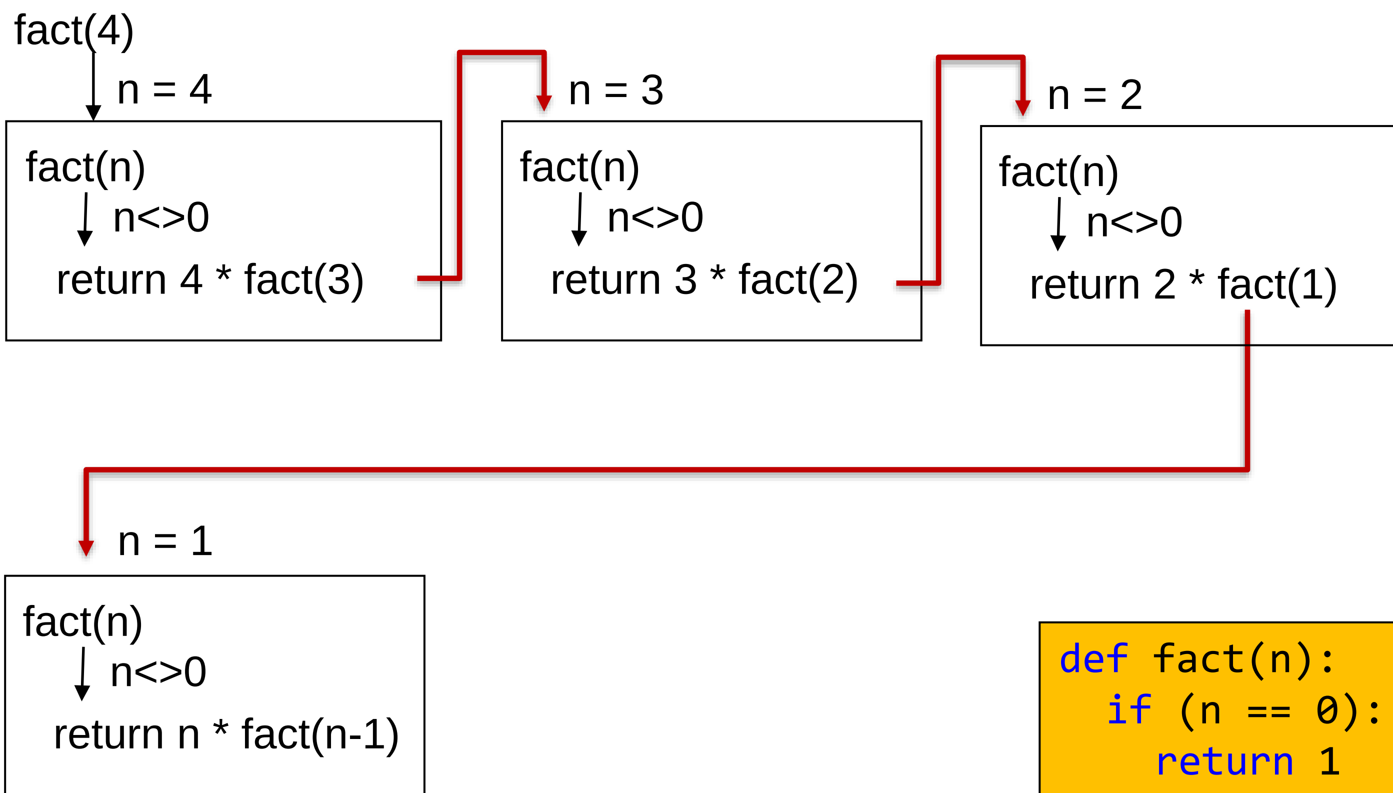
Exercise: Factorial




```
def fact(n):
  if (n == 0):
    return 1

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

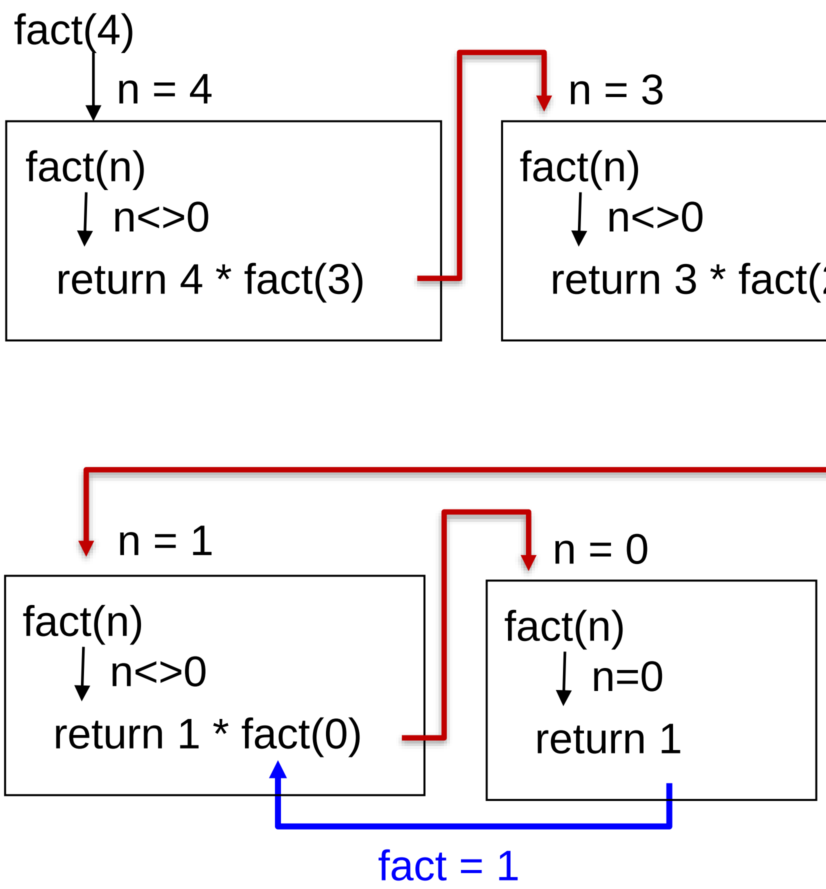
Exercise: Factorial




```python
def fact(n):
  if (n == 0):
    return 1

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

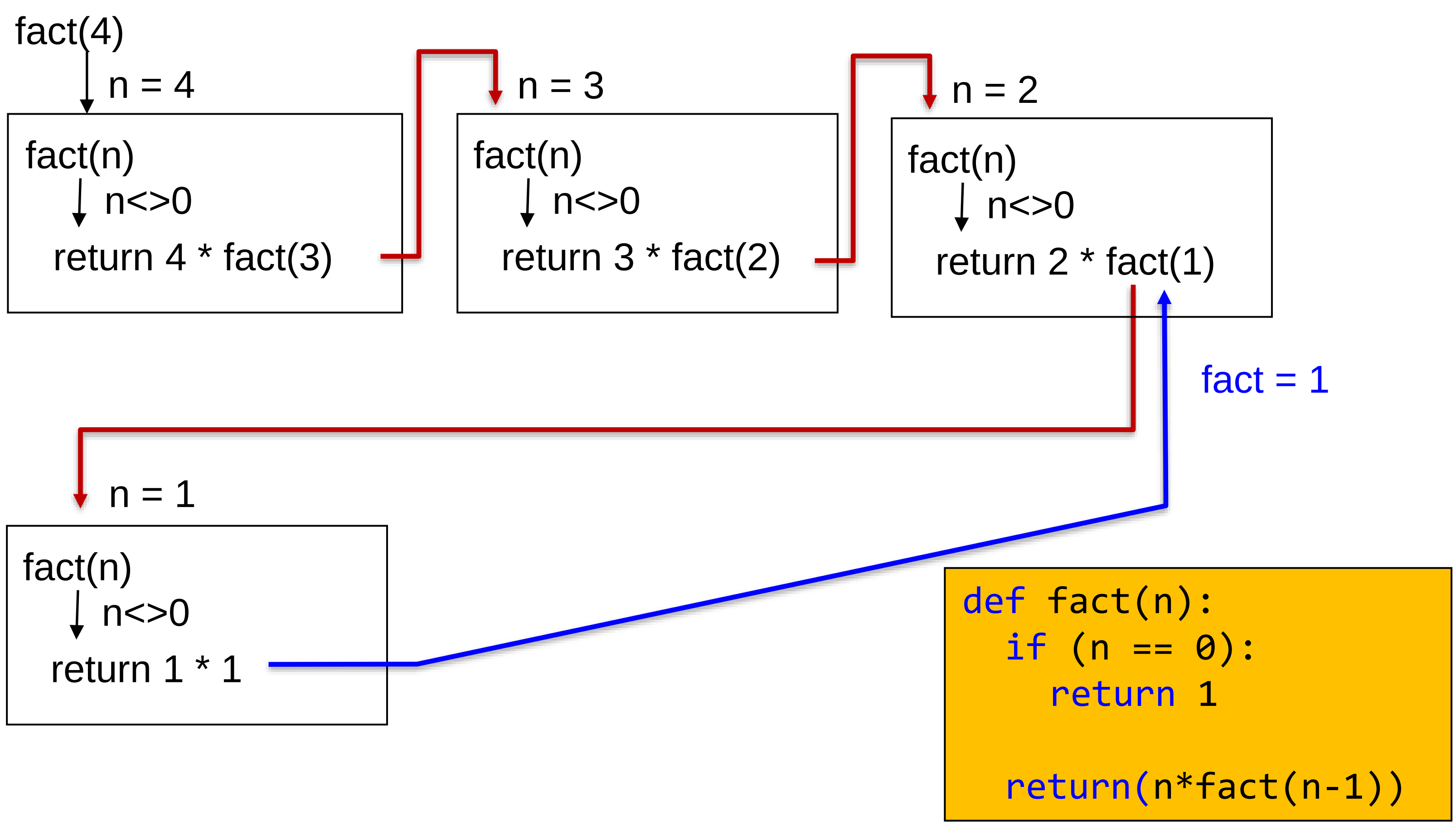
Exercise: Factorial



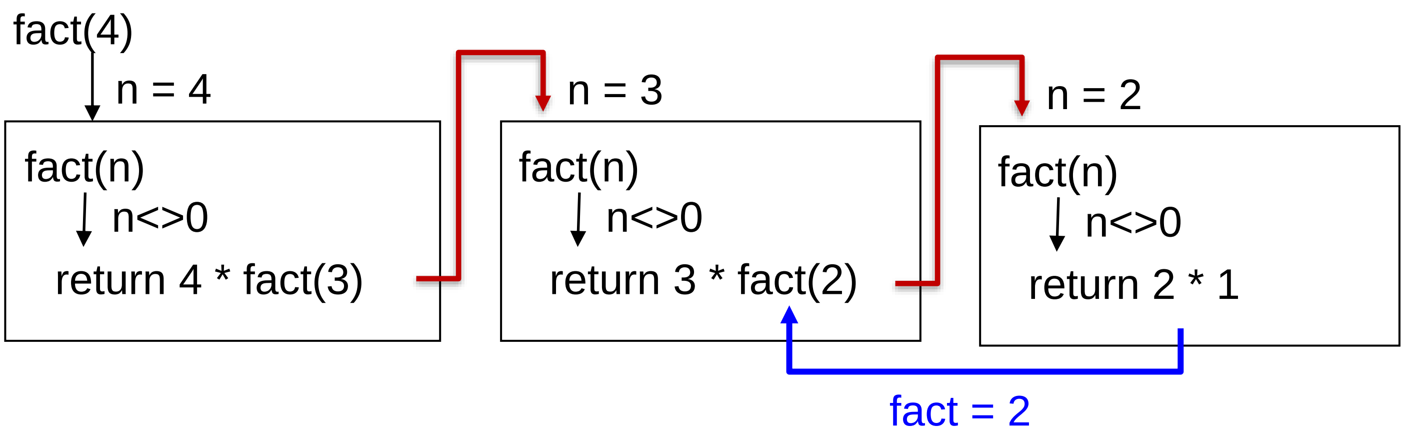

```
def fact(n):
  if (n == 0):
    return 1

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

Exercise: Factorial



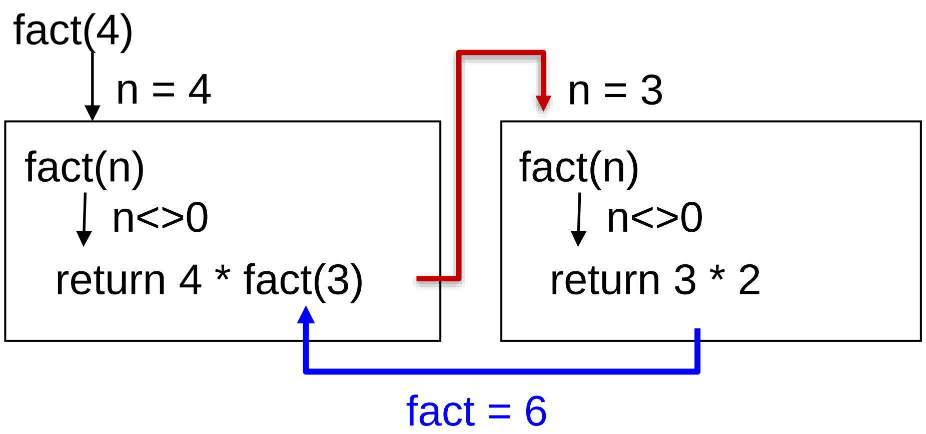
Exercise: Factorial




```
def fact(n):
  if (n == 0):
    return 1

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

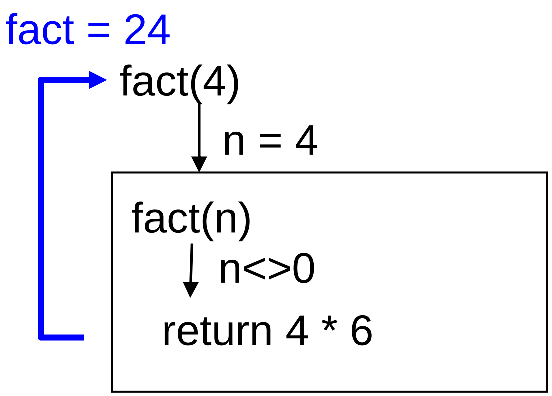
Exercise: Factorial




```python
def fact(n):
  if (n == 0):
    return 1

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

Exercise: Factorial




```python
def fact(n):
  if (n == 0):
    return 1

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

Execution of recursive algorithms

- In each recursive call it is necessary to save the local objects (parameters and local variables) of the current call.
- It uses a pile: “the call stack”.