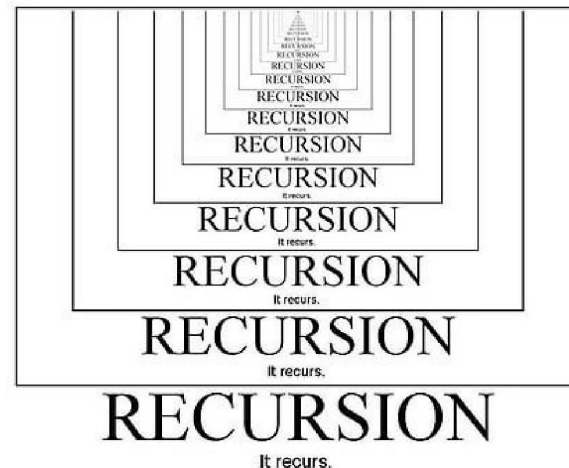


Recursión

- Definiciones
- Recursión en python
- Pilas
- Ejemplos
- Memoization



*La recursividad es el proceso mediante el que una **función se llama a sí misma** de forma repetida, hasta que se satisface alguna determinada condición.*



El concepto de recursividad se usa en muchos aspectos de la vida.

Ejemplo: supongamos que para ser ciudadano de un determinado país debe cumplirse alguna de las siguientes dos condiciones:

- A. **Nació** en el territorio
- B. Nació fuera del territorio, pero uno de sus **progenitores tiene la ciudadanía**

Problema: ¿cómo sabemos si una persona es ciudadana de ese país?

Recursividad

Es ciudadano?

Recursividad

Es ciudadano? 🙄

Recursividad

Es ciudadano? 🙄

↳ Nació en territorio? NO

Recursividad

Es ciudadano? 🙄

└─> Nació en territorio? NO

└─> Progenitor ciudadano? 🙄

Es ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

Es ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

Es ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

Es ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

Es ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

Es ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano?

Es ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? SÍ

Es ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? **SÍ**



Es ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? **SÍ**



Recursividad

Es ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? **SÍ**



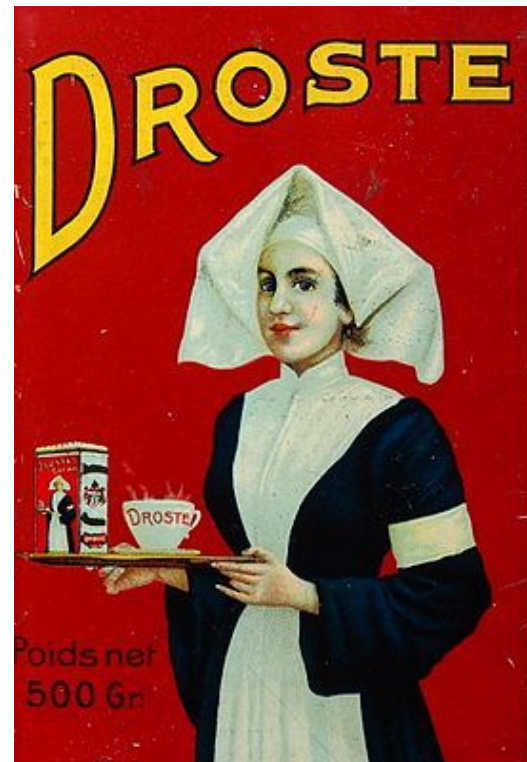
Recursividad

Es ciudadano? SÍ



En general una definición recursiva consta de dos partes:

1. **Caso recursivo:** resolver un problema reduciéndolo a una versión más pequeña del mismo problema.
2. **Caso base:** especifica directamente el resultado para un caso particular. Garantiza que el algoritmo recursivo termine y se resuelva en un tiempo finito.



Es ciudadano? 🧑

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🧑

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🧑

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🧑

↳ Nació en territorio? NO

CASO BASE



Progenitor ciudadano? SÍ

Es ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

↳ Nació en territorio? NO

↳ Progenitor ciudadano? 🙄

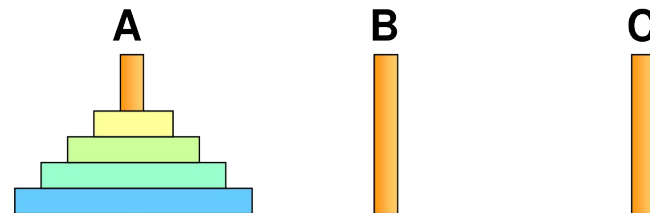
↳ Nació en territorio? NO

CASO BASE



Progenitor ciudadano? NO

Algorítmicamente: Una manera de diseñar soluciones a problemas utilizando el patrón **Divide y Reinarás**. Reducir el problema a versiones más simples de ese mismo problema.



Semánticamente: Una técnica de programación que usa una función que se invoca a sí misma.

```
def func(n):  
    print(n)  
    func(n-1)
```

Recursión en python

```
def func(x):  
    print(x)
```

```
func(3)
```

Vimos que en la definición de cada función puede llamarse a otras funciones


```
def func(x):  
    print(x)
```

```
func(3)
```

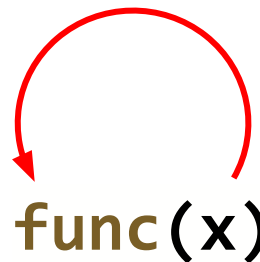
Vimos que en la definición de cada función puede llamarse a otras funciones

```
def func(x):  
    func(x)
```

```
func(3)
```

Python (al igual que otros lenguajes) permite que una función se llame así misma.

RECURSIVIDAD



Pila (stack)

Pila (Stack)

1. Cuando **se llama** a una función, se crea una nueva tabla de símbolos (a menudo llamada **stack frame** o **marco de pila**).
2. Cuando la función **finaliza**, el **stack frame** desaparece.
3. Si se invoca una función dentro de otra, los *stack frames* **se van acumulando**.
4. A medida que las funciones terminan se libera primero el *stack frame* añadido más recientemente y por último el más antiguo.

*Se comporta como si fueran panqueques apilados recién cocidos: **"último en entrar primero en salir"** (el último en cocinarse es el primero en servirse).*



Pila (Stack)

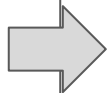
```
def func0(a,b):  
    print(a+b)  
  
def func1(x,y):  
    func0(x*y,x-y)  
  
def func2(z):  
    func1(z,z**2)  
  
func2(5)
```

Pila (Stack)

```
def func0(a,b):  
    print(a+b)
```

```
def func1(x,y):  
    func0(x*y,x-y)
```

```
def func2(z):  
    func1(z,z**2)
```




```
func2(5)
```

Pila (Stack)

```
def func0(a,b):  
    print(a+b)
```

```
def func1(x,y):  
    func0(x*y,x-y)
```



```
def func2(z):  
    func1(z,z**2)
```

```
func2(5)
```

Pila (Stack)

```
def func0(a,b):  
    print(a+b)  
  
def func1(x,y):  
    func0(x*y,x-y)  
  
def func2(z):  
    func1(z,z**2)
```

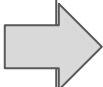
func2(5)

stack frame
func2()

z

Pila (Stack)

```
def func0(a,b):  
    print(a+b)
```



```
def func1(x,y):  
    func0(x*y,x-y)
```

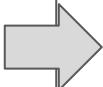
```
def func2(z):  
    func1(z,z**2)
```

```
func2(5)
```

stack frame func2()	z
------------------------	---

Pila (Stack)

```
def func0(a,b):  
    print(a+b)
```



```
def func1(x,y):  
    func0(x*y,x-y)
```

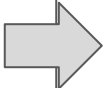
```
def func2(z):  
    func1(z,z**2)
```

```
func2(5)
```

stack frame func1()	x y
------------------------	-----

stack frame func2()	z
------------------------	---

Pila (Stack)

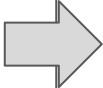


```
def func0(a,b):  
    print(a+b)  
  
def func1(x,y):  
    func0(x*y,x-y)  
  
def func2(z):  
    func1(z,z**2)  
  
func2(5)
```

stack frame func1()	x y
------------------------	-----

stack frame func2()	z
------------------------	---

Pila (Stack)



```
def func0(a,b):  
    print(a+b)
```

```
def func1(x,y):  
    func0(x*y,x-y)
```

```
def func2(z):  
    func1(z,z**2)
```

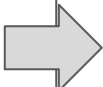
```
func2(5)
```

stack frame func0()	a b
------------------------	-----

stack frame func1()	x y
------------------------	-----

stack frame func2()	z
------------------------	---

Pila (Stack)



```
def func0(a,b):  
    print(a+b)
```

```
def func1(x,y):  
    func0(x*y,x-y)
```

```
def func2(z):  
    func1(z,z**2)
```

```
func2(5)
```

stack frame func0()	a b
------------------------	-----

stack frame func1()	x y
------------------------	-----

stack frame func2()	z
------------------------	---

Pila (Stack)

```
def func0(a,b):  
    print(a+b)
```

```
def func1(x,y):  
    func0(x*y,x-y)
```

```
def func2(z):  
    func1(z,z**2)
```

```
func2(5)
```

stack frame func1()	x y
------------------------	-----

stack frame func2()	z
------------------------	---

Pila (Stack)

```
def func0(a,b):  
    print(a+b)  
  
def func1(x,y):  
    func0(x*y,x-y)  
  
def func2(z):  
    func1(z,z**2)  
  
func2(5)
```

stack frame
func2()

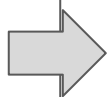
z

Pila (Stack)

```
def func0(a,b):  
    print(a+b)
```

```
def func1(x,y):  
    func0(x*y,x-y)
```

```
def func2(z):  
    func1(z,z**2)
```



```
func2(5)
```

Pila en la recursión


Recursión

```
def func(n):  
    print(n)  
    func(n-1)  
  
func(3)
```

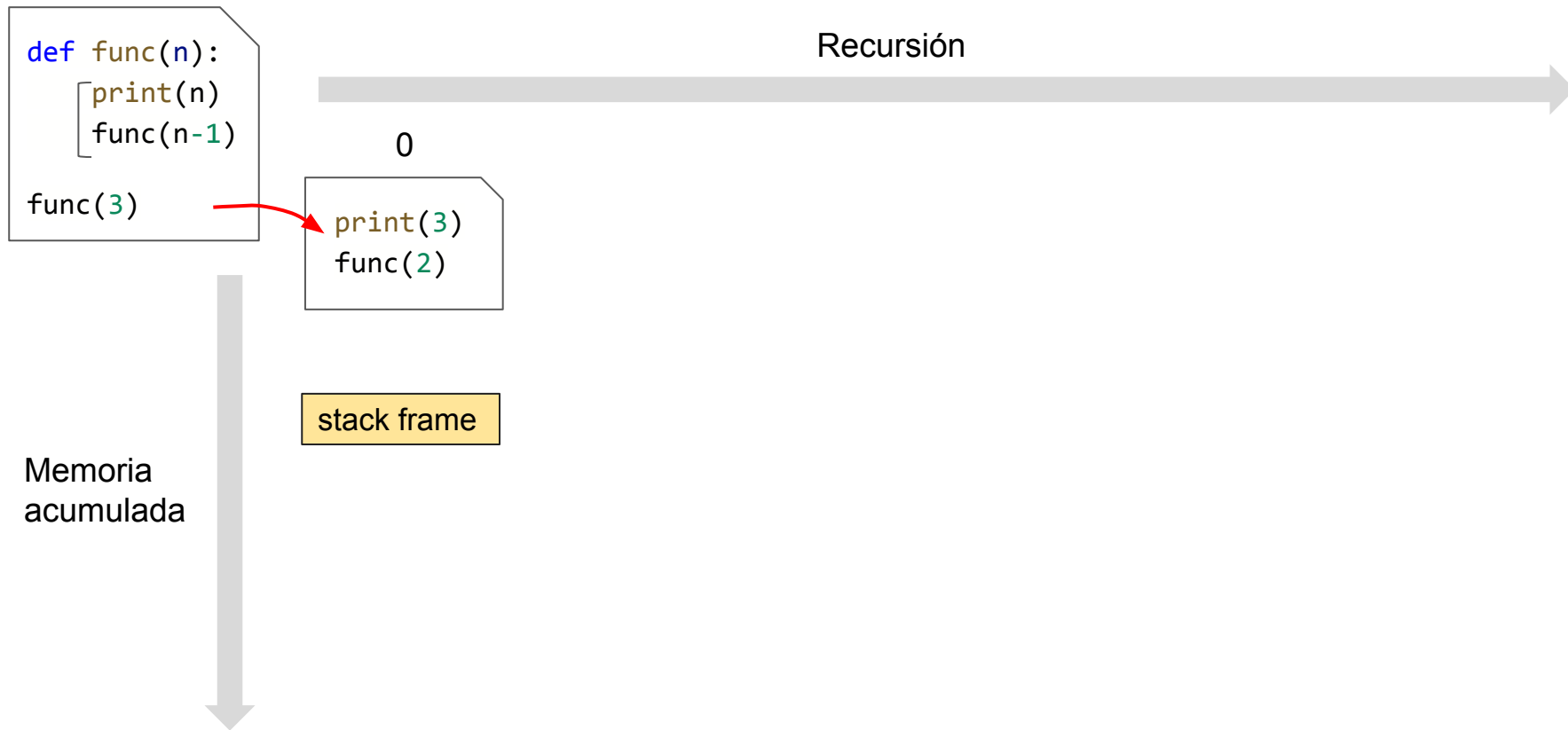
Recursión



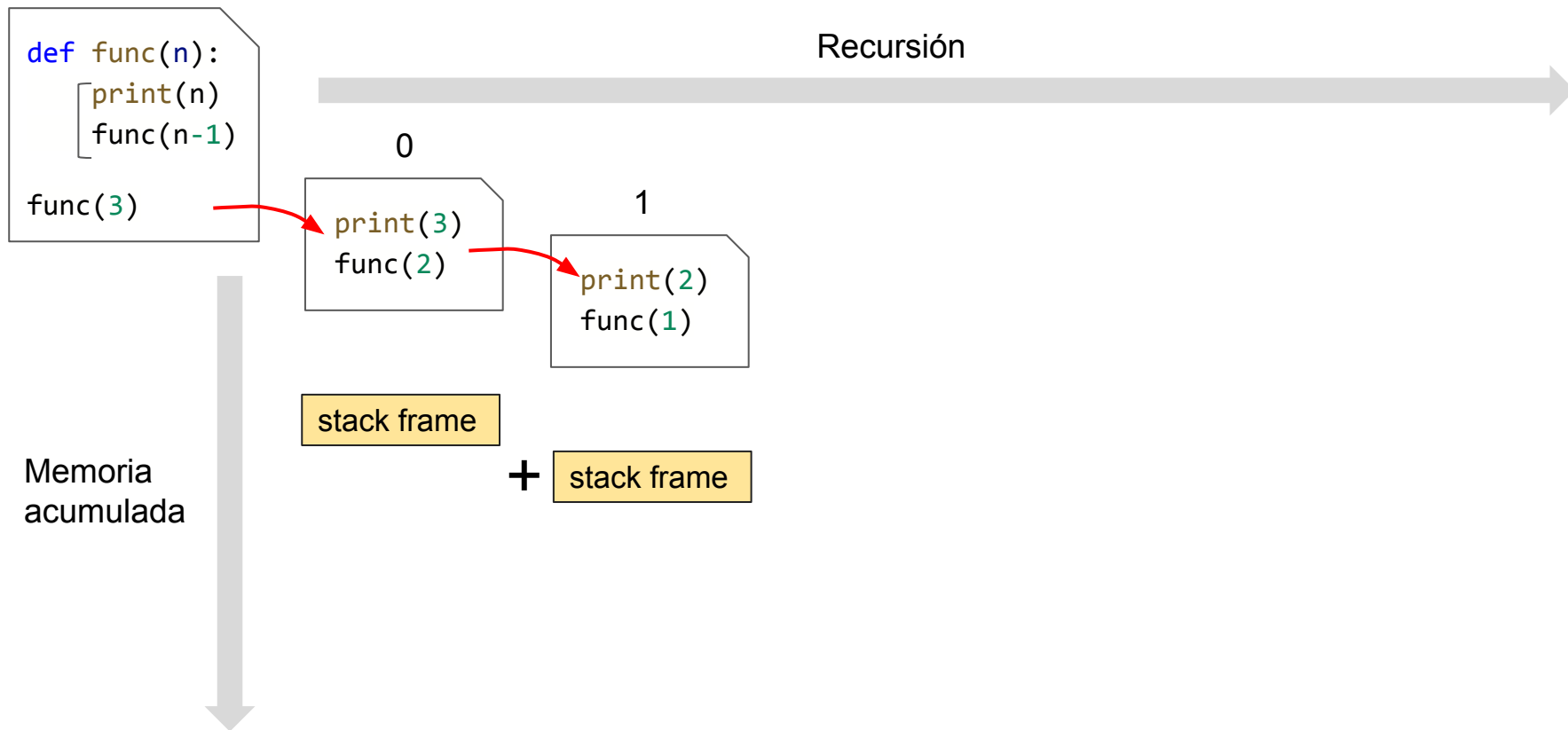
Memoria
acumulada



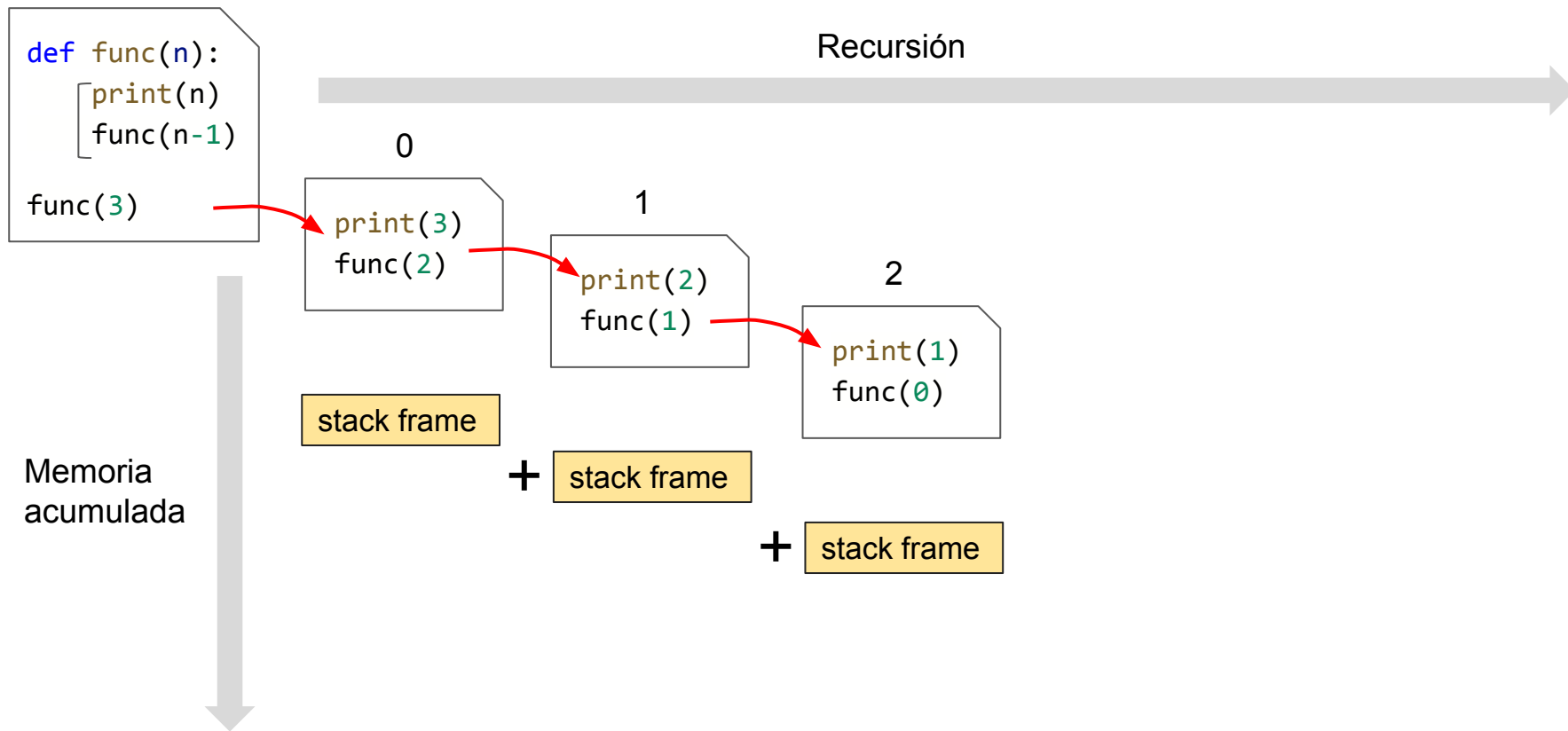
Recursión



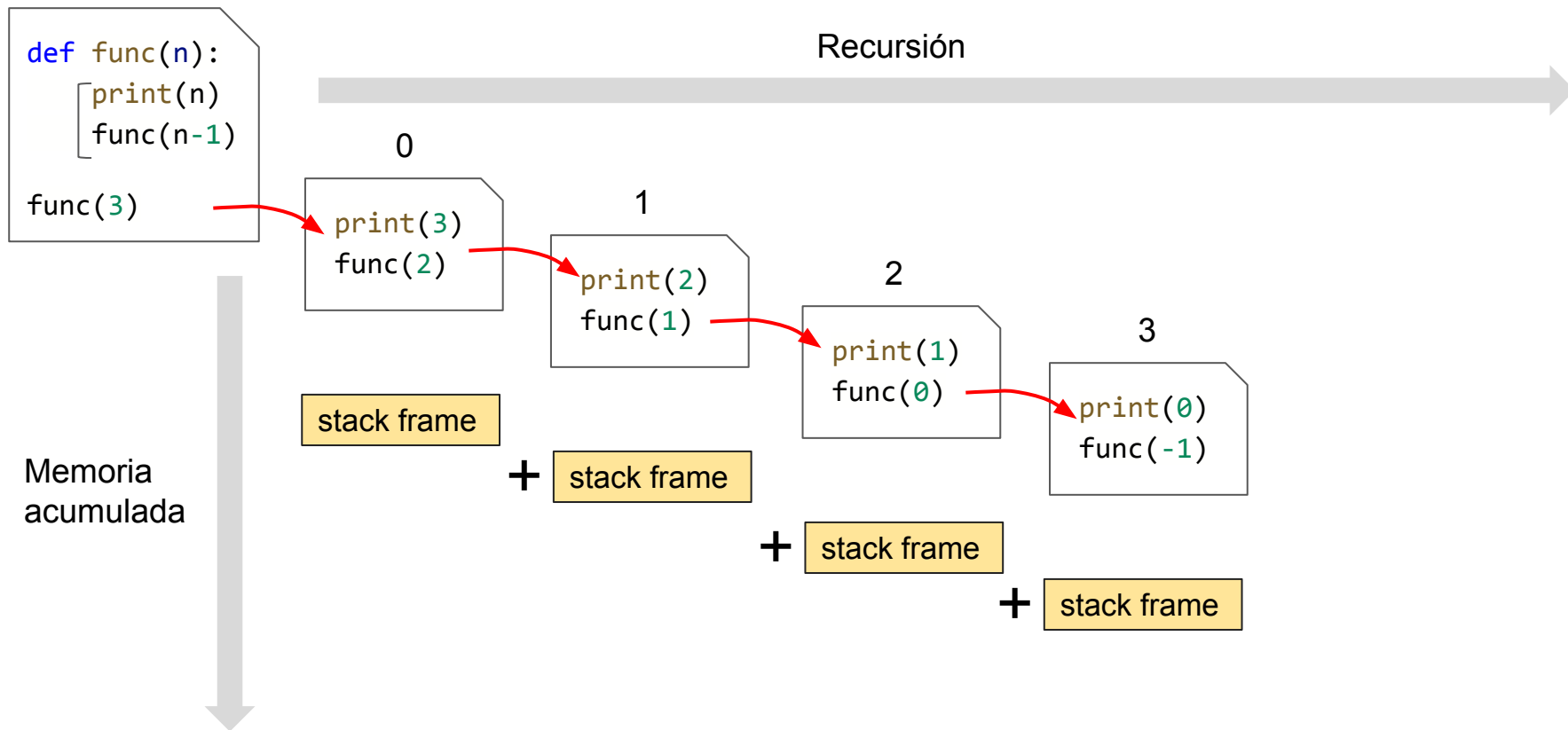
Recursión



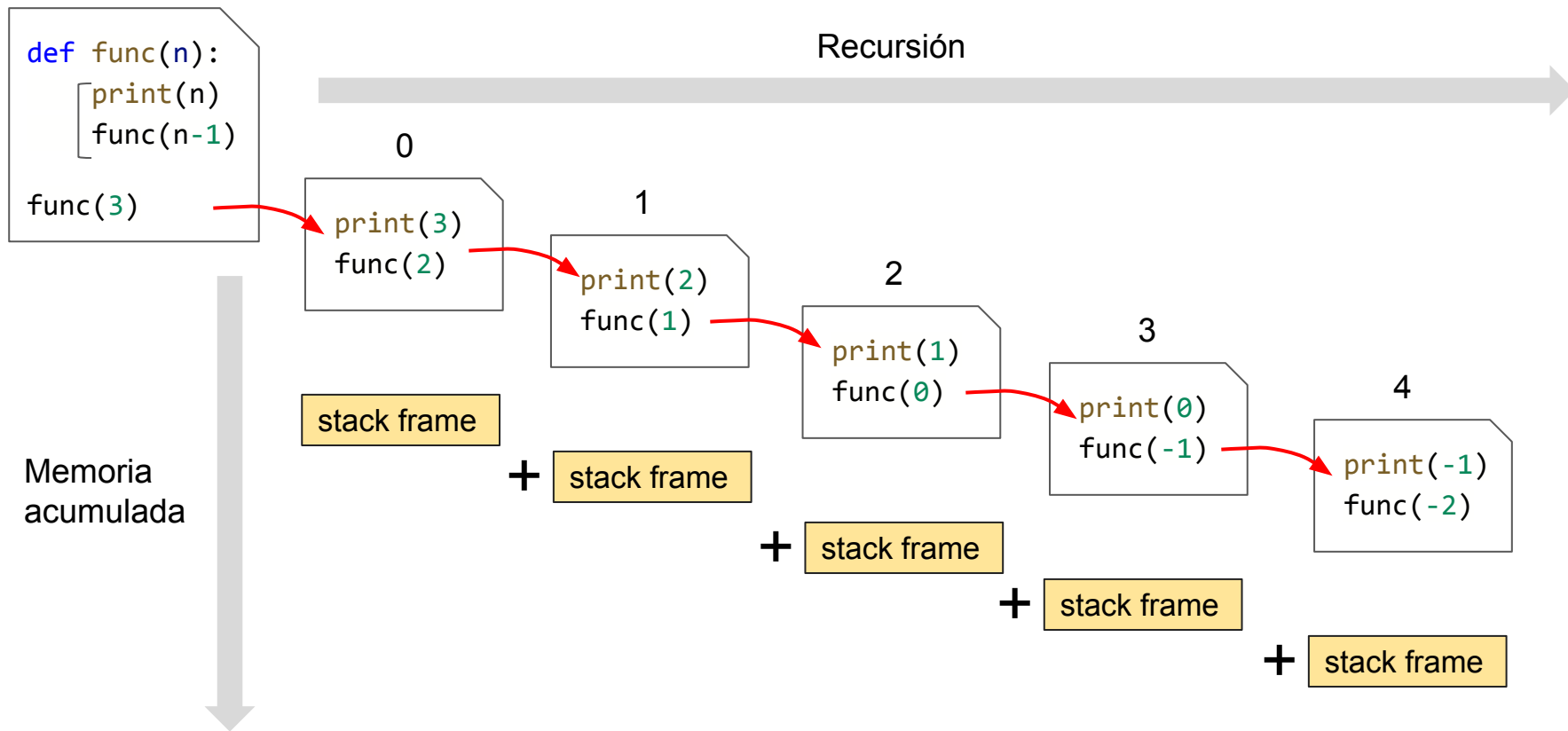
Recursión



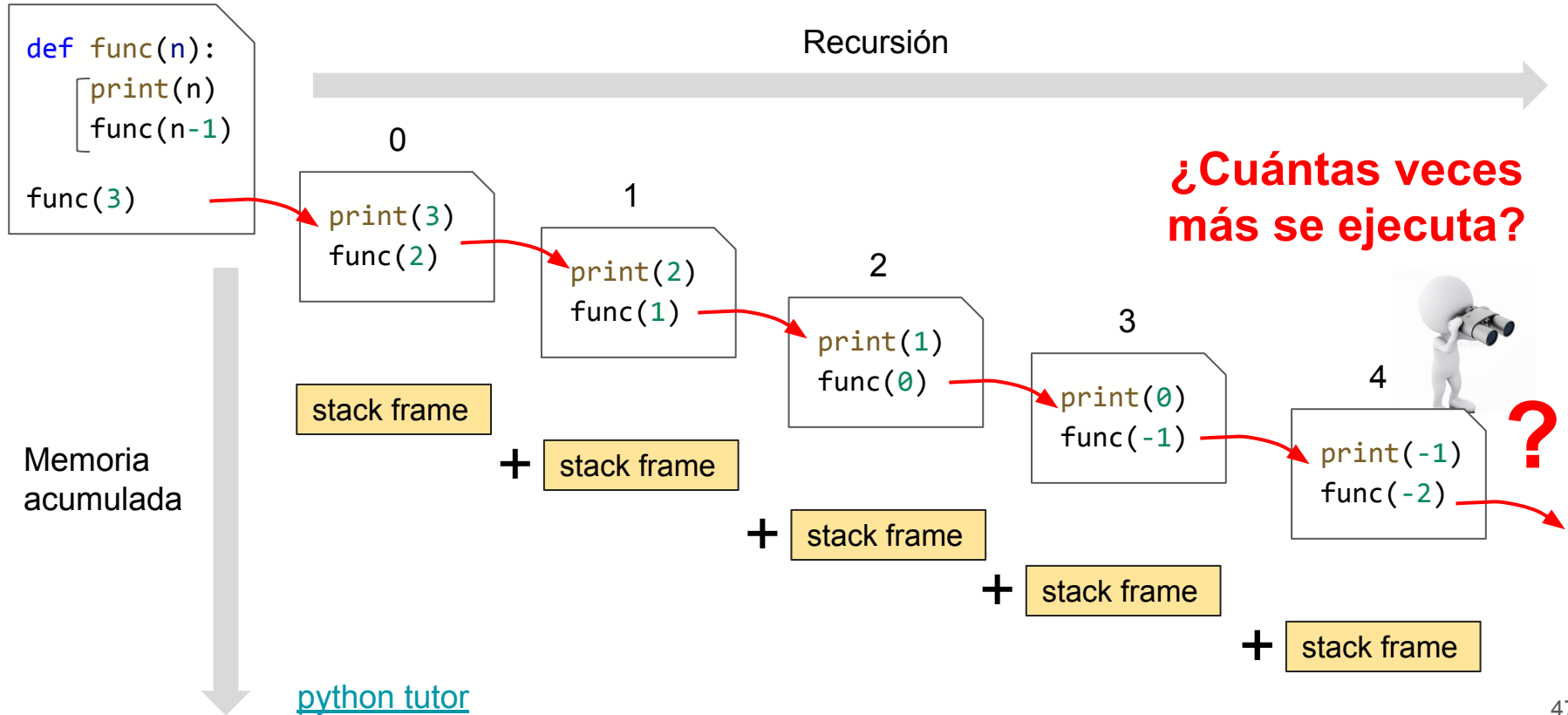
Recursión



Recursión



Recursión



Recursión

```
def func(n):  
    print(n)  
    func(n-1)
```

```
func(3)
```

```
3  
2  
1  
0  
-1  
-2  
...  
-958
```

RecursionError: maximum recursion depth exceeded while calling a Python object

Recursión

```
def func(n):  
    print(n)  
    func(n-1)  
  
func(3)
```

Si no se incluye un caso **base** continúa indefinidamente.

En realidad, se dispara una excepción cuando alcanza el máximo de recursiones permitidas

```
3  
2  
1  
0  
-1  
-2  
...  
-958
```

RecursionError: maximum recursion depth exceeded while calling a Python object

Recursión

```
def func2(n):  
    if n>0:  
        print(n)  
        func2(n-1)  
func2(5)
```

Agregamos un **caso base** en el que concluye la recursión (en este ejemplo, cuando n deja de ser un número positivo)

Recursión

```
def func2(n):  
    if n>0:  
        print(n)  
        func2(n-1)  
func2(5)
```

Recursión

```
def func2(n):
```

```
    if n>0:
```

```
        if 5>0:
```

```
            print(5)
```

```
            func2(4)
```

```
func2(5)
```

5

Recursión

```
def func2(n):
```

```
    if n>0:
```

```
func2(5)
```

```
    if 5>0:
```

```
        print(5)
```

```
        func2(4)
```

```
    if 4>0:
```

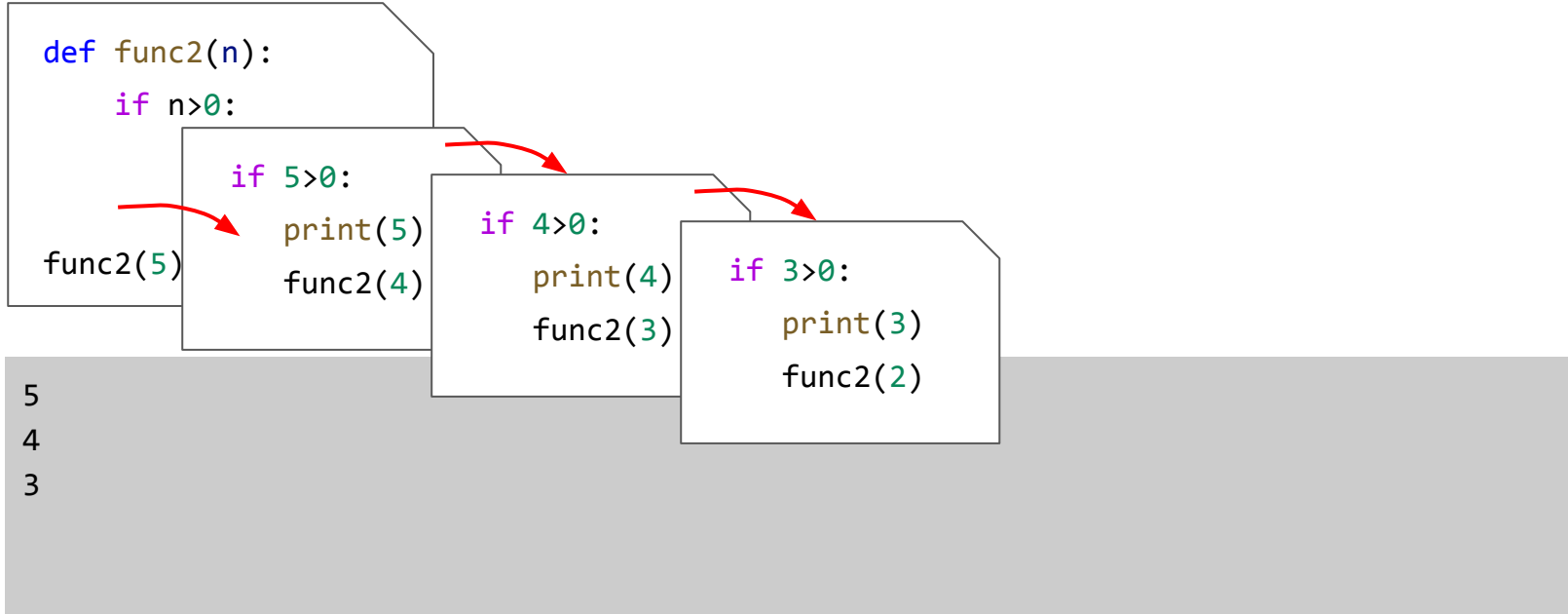
```
        print(4)
```

```
        func2(3)
```

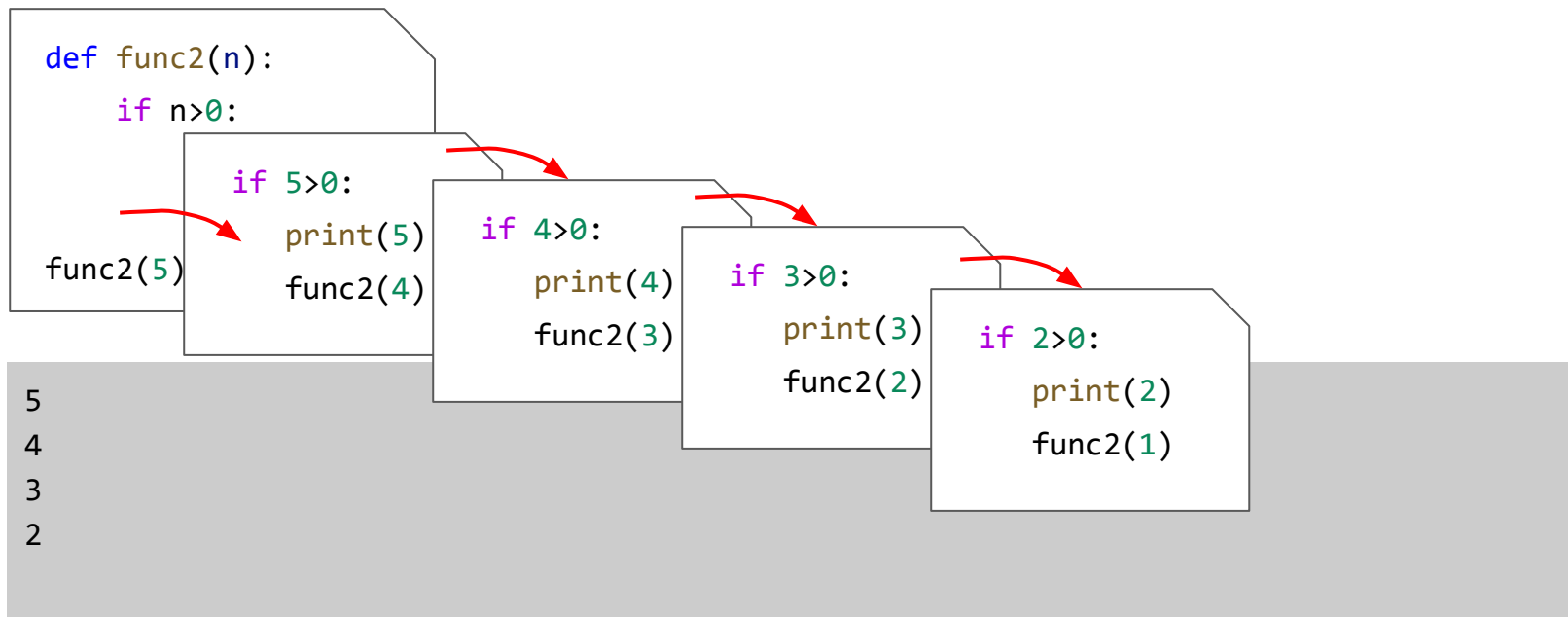
5

4

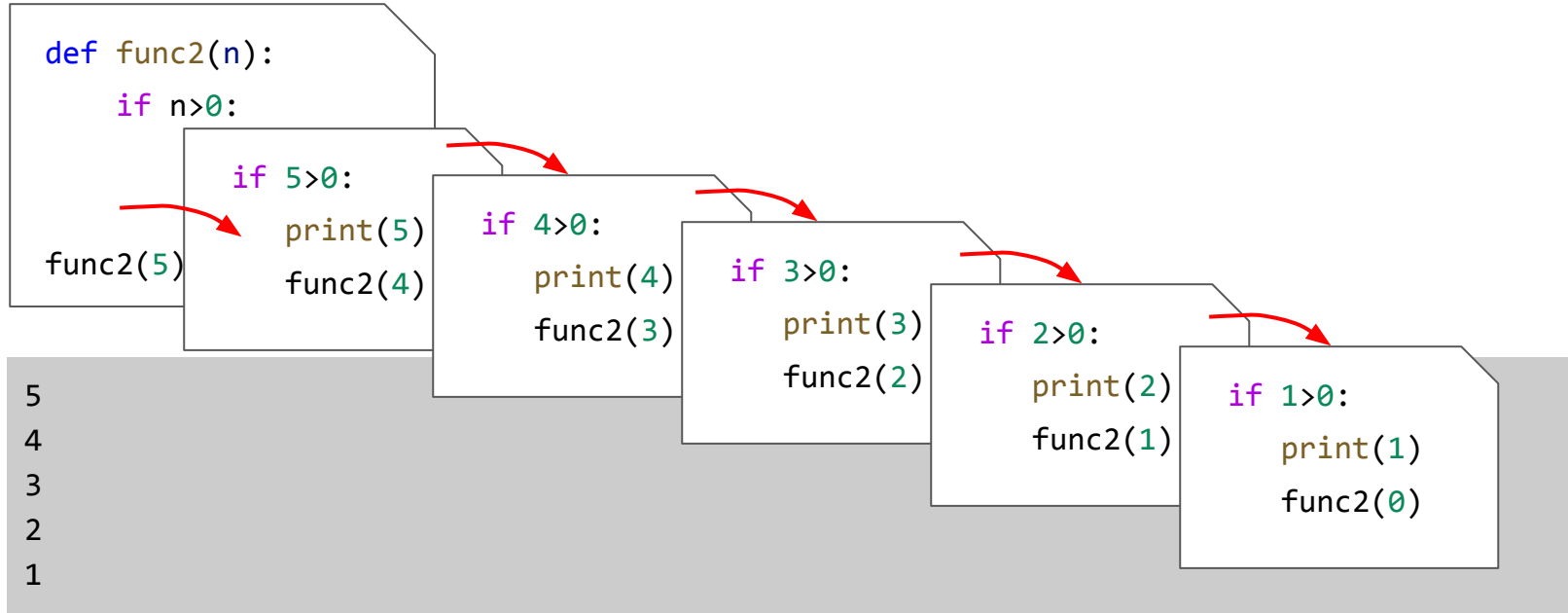
Recursión



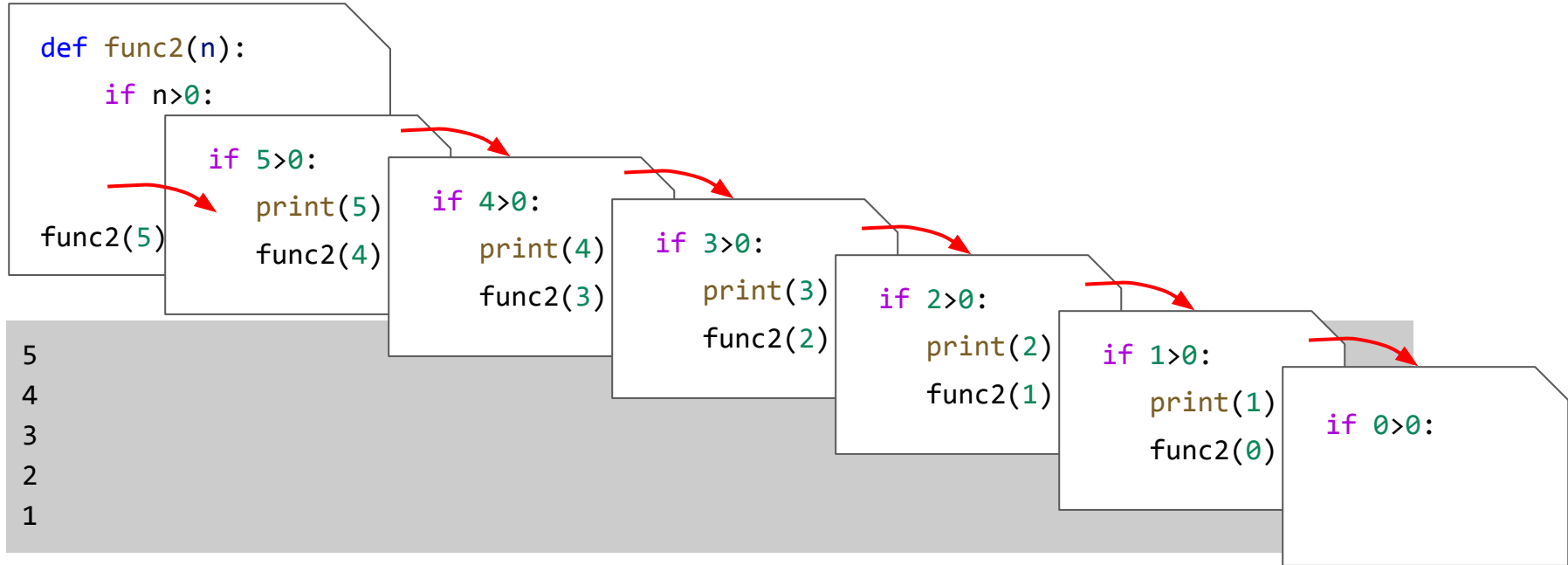
Recursión



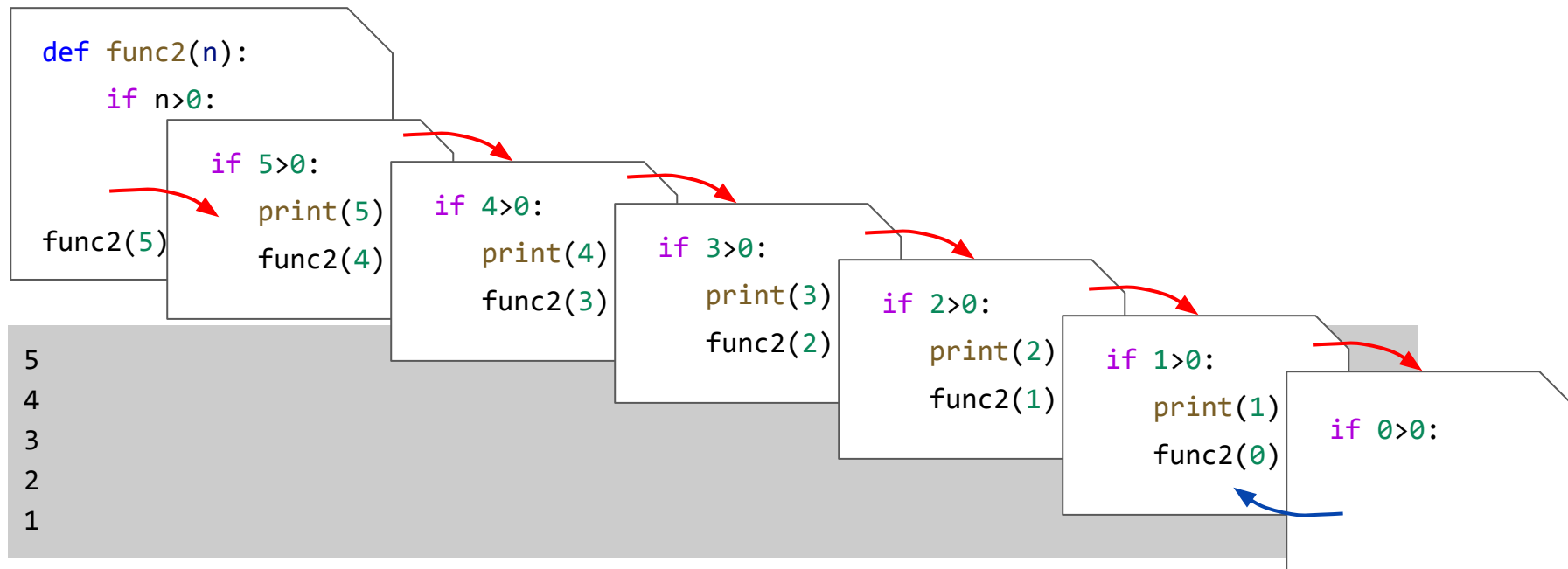
Recursión



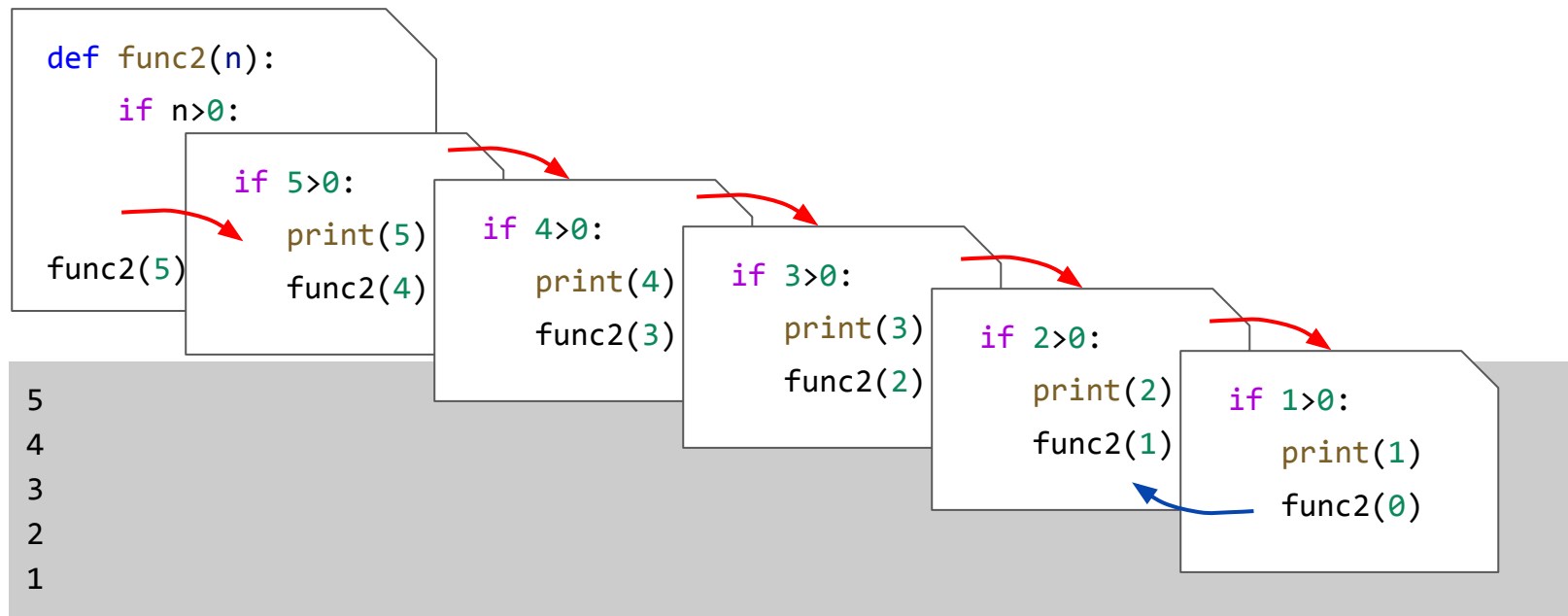
Recursión



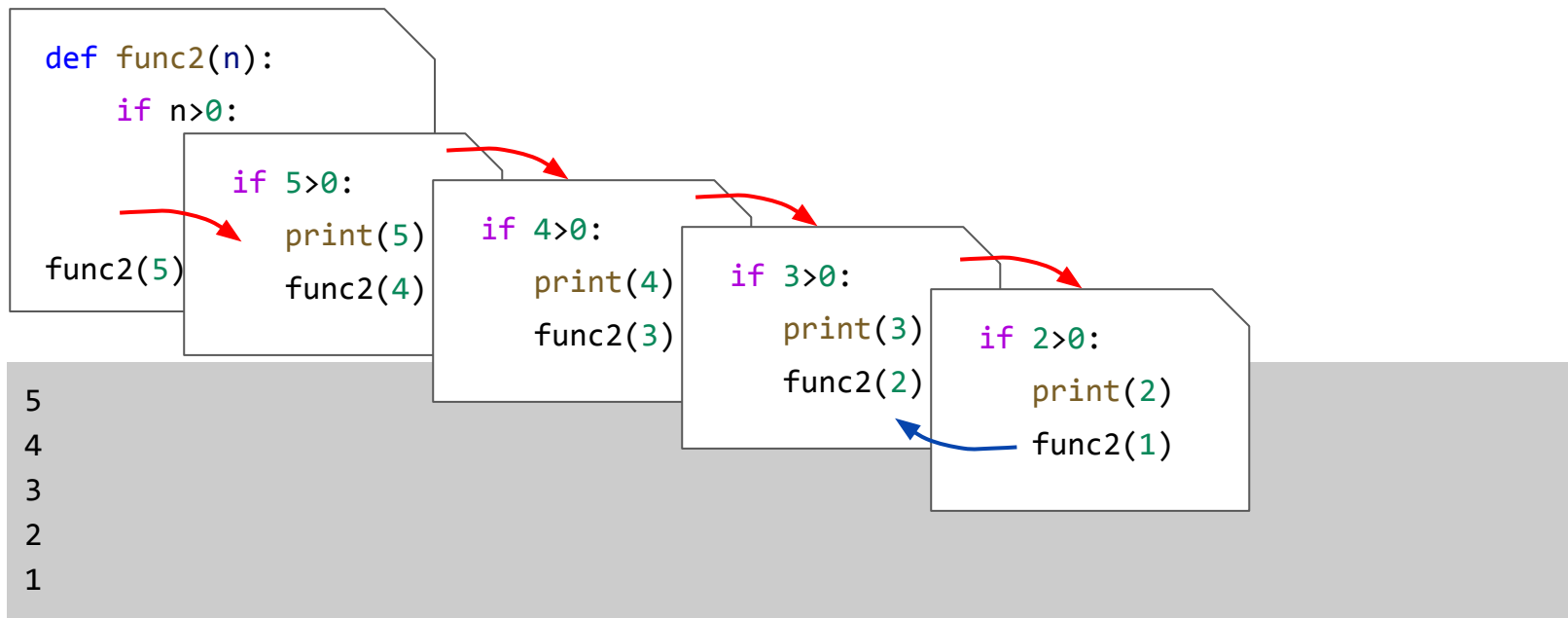
Recursión



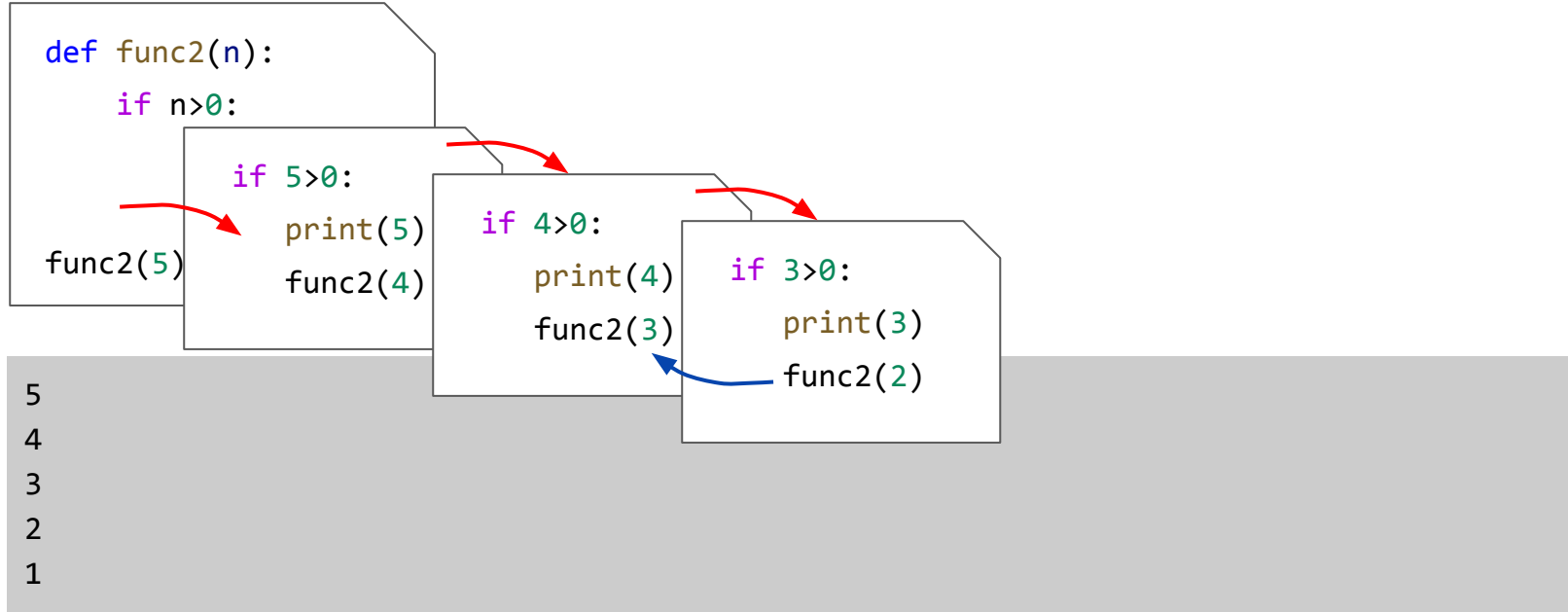
Recursión



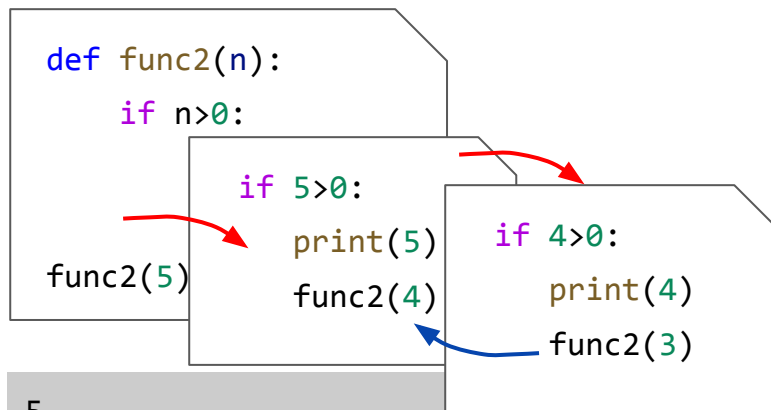
Recursión



Recursión



Recursión



5
4
3
2
1

Recursión

```
def func2(n):
```

```
    if n>0:
```

```
        if 5>0:
```

```
            print(5)
```

```
func2(5)
```

```
        func2(4)
```

5
4
3
2
1

Recursión

```
def func2(n):  
    if n>0:  
        print(n)  
        func2(n-1)  
func2(5)
```

5
4
3
2
1

[python tutor](#)

Recursión

```
def func2(n):  
    if n>0:  
        func2(n-1)  
        print(n)  
func2(5)
```

¿Por qué en este caso se imprime en el orden inverso al ejemplo anterior?

1
2
3
4
5

Ejemplos

Multiplicación - Perspectiva Iterativa

Problema: multiplicación entre dos números enteros usando solo sumas

- Usamos un ciclo
- Multiplicar $a * b$ es igual que sumar a b -veces, ¿verdad?

```
def mult_iter(a, b):  
    """ retorna c = a * b """  
    c = 0  
    for i in range(b):  
        c += a  
    return c
```

$a + a + a + a + \dots + a$
1 2 3 4 ... b veces

[python tutor](#)

Multiplicación - Perspectiva Recursiva

Problema: multiplicación entre dos números enteros usando solo sumas

- **Caso Recursivo:** Reducir el problema a uno más simple del mismo tipo
- **Caso Base:** caso en el cual ya sabemos el resultado, $a * 1 = a$

```
def mult_rec(a, b):  
    """ retorna a * b """  
    if b == 1:  
        return a  
    else:  
        return a + mult_rec(a, b-1)
```

[python tutor](#)

$$a * b = a + \underbrace{a + a + a + \dots + a}_{a * (b - 1)}$$

$$a * b = a + a * (b - 1)$$

Ejemplo: Suma de Gauss

$$S(n) = 1 + 2 + 3 + 4 + 5 + \dots + n-2 + n-1 + n$$

Ejemplo: Suma de Gauss

$$S(n) = 1 + 2 + 3 + 4 + 5 + \dots + n-2 + n-1 + n = S(n-1) + n$$

$\underbrace{\hspace{15em}}_{S(n-1)}$

Ejemplo: Suma de Gauss

$$S(n) = 1 + 2 + 3 + 4 + 5 + \dots + n-2 + n-1 + n = S(n-1) + n$$

$\underbrace{\hspace{15em}}_{S(n-1)}$

$$S(n) = S(n-1) + n$$

**Hallamos una
expresión recursiva
para la función $S(n)$!!**

Ejemplo: Suma de Gauss

$$S(n) = 1 + 2 + 3 + 4 + 5 + \dots + n-2 + n-1 + n = S(n-1) + n$$

$\underbrace{\hspace{10em}}_{S(n-1)}$

Solución iterativa

```
def suma_gauss(n):  
    s=0  
    for i in range(1,n+1):  
        s=s+i  
    return s  
  
print(suma_gauss(6))
```

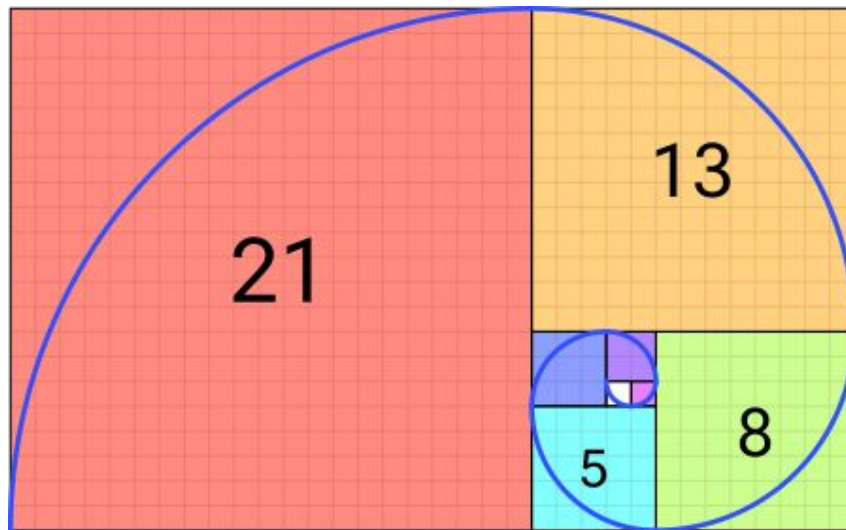
Solución recursiva

```
def suma_gauss(n):  
    if n == 1:  
        return 1  
    return suma_gauss(n-1) + n  
  
print(suma_gauss(6))
```


Ejemplo: serie de fibonacci

Ecuaciones que definen los
números de Fibonacci

$$\left\{ \begin{array}{l} f_0 = 0 \\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \end{array} \right.$$



$$\begin{aligned} 0 + 1 &= 1 \\ 1 + 1 &= 2 \\ 2 + 1 &= 3 \\ 3 + 2 &= 5 \\ 5 + 3 &= 8 \\ 8 + 5 &= 13 \\ 13 + 8 &= 21 \\ 21 + 13 &= 34 \\ 34 + 21 &= 55 \\ 55 + 34 &= 89 \end{aligned}$$

f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	...
0	1	1	2	3	5	8	13	21	34	55	89	144	...

Ejemplo: serie de fibonacci

Solución iterativa

```
def fibo(n):  
    if n == 0 or n == 1:  
        return n  
    fn_1 = 1  
    fn_2 = 0  
    for i in range(2, n + 1):  
        fn = fn_1 + fn_2  
        fn_2 = fn_1  
        fn_1 = fn  
    return fn
```

Solución recursiva

```
def fibo(n):  
    if n == 0 or n == 1:  
        return n  
    return fibo(n - 1) + fibo(n - 2)
```

El código es más elegante y simple, pero más ineficiente

Ejemplo: serie de fibonacci

Probando la solución iterativa

```
def fibo(n):  
    if n == 0 or n == 1:  
        return n  
    fn_1 = 1  
    fn_2 = 0  
    for i in range(2, n + 1):  
        print(f'f({i}) Iteración')  
        fn = fn_1 + fn_2  
        fn_2 = fn_1  
        fn_1 = fn  
    return fn
```

```
fibo(6)
```

```
f(2) Iteración  
f(3) Iteración  
f(4) Iteración  
f(5) Iteración  
f(6) Iteración  
8
```

Ejemplo: serie de fibonacci

Probando la solución recursiva

```
def fibo(n):  
    if n == 0 or n == 1:  
        return n  
    print(f'f({n}) Recursión')  
    fn = fibo(n - 1) + fibo(n - 2)  
    return fn
```

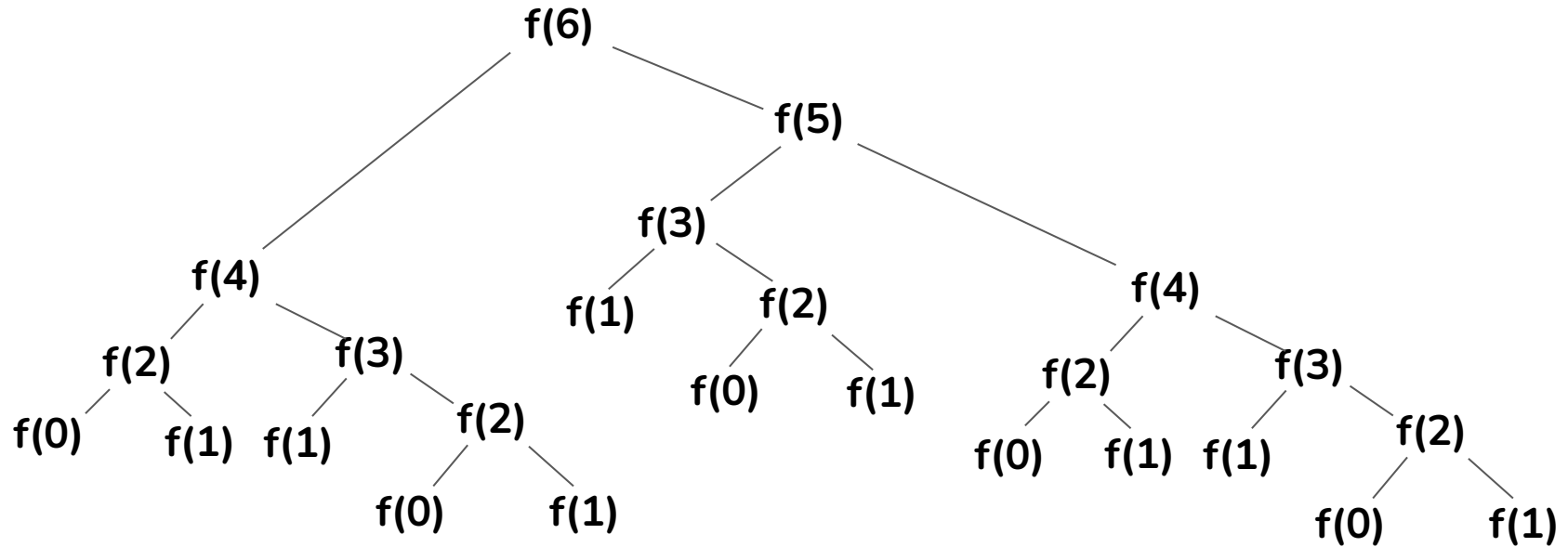
```
fibo(6)
```

```
f(6) Recursión  
f(5) Recursión  
f(4) Recursión  
f(3) Recursión  
f(2) Recursión  
f(2) Recursión  
f(3) Recursión  
f(2) Recursión  
f(4) Recursión  
f(3) Recursión  
f(2) Recursión  
f(2) Recursión  
8
```

Muy
ineficiente!

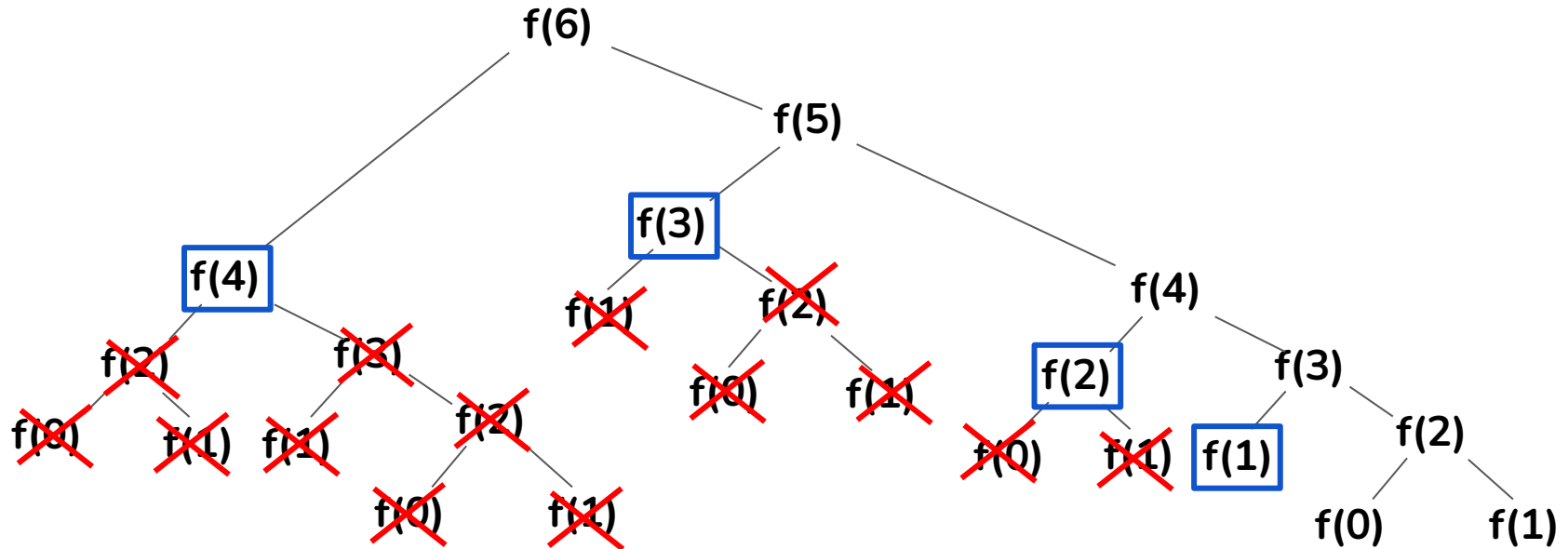
[python tutor](#)

Ejemplo: serie de fibonacci



Memoization

Ejemplo: serie de fibonacci



Ejemplo: serie de fibonacci

Probando la solución recursiva más eficiente (**MEMOIZATION**)

```
def fibo(n, d):  
    if n in d:  
        return d[n]  
  
    print(f'f({n}) Recursión')  
    d[n] = fibo(n-1,d) + fibo(n-2,d)  
    return d[n]
```

```
d = {0:0, 1:1}  
fibo(6, d)
```

```
f(6) Recursión  
f(5) Recursión  
f(4) Recursión  
f(3) Recursión  
f(2) Recursión  
8
```

Se pueden utilizar diccionarios para guardar las $f(n)$ ya evaluadas y evitar repetir ejecuciones redundantes. Esto se denomina **memoization**

[python tutor](#)