Tema 3. Recursivitat

Algorísmica

Grau d'Enginyeria Informàtica

Mireia Ribera, ribera@ub.edu
Daniel Ortiz, Daniel.Ortiz@ub.edu

Recursivitat: Conceptes clau

1. Motivació
2. Definició i anatomia: cas base i reducció
3. Cost i eficiència: pila d'execució. Versió iterativa

Motivació

Ja hem vist que una funció en pot cridar a una altra... però té sentit que *una funció es cridi a si mateixa?*

Veiem un exemple: el càlcul del factorial.

$$n! = \begin{cases} 1 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ n \cdot (n-1)! & \text{si } n > 1 \end{cases}$$

Definició

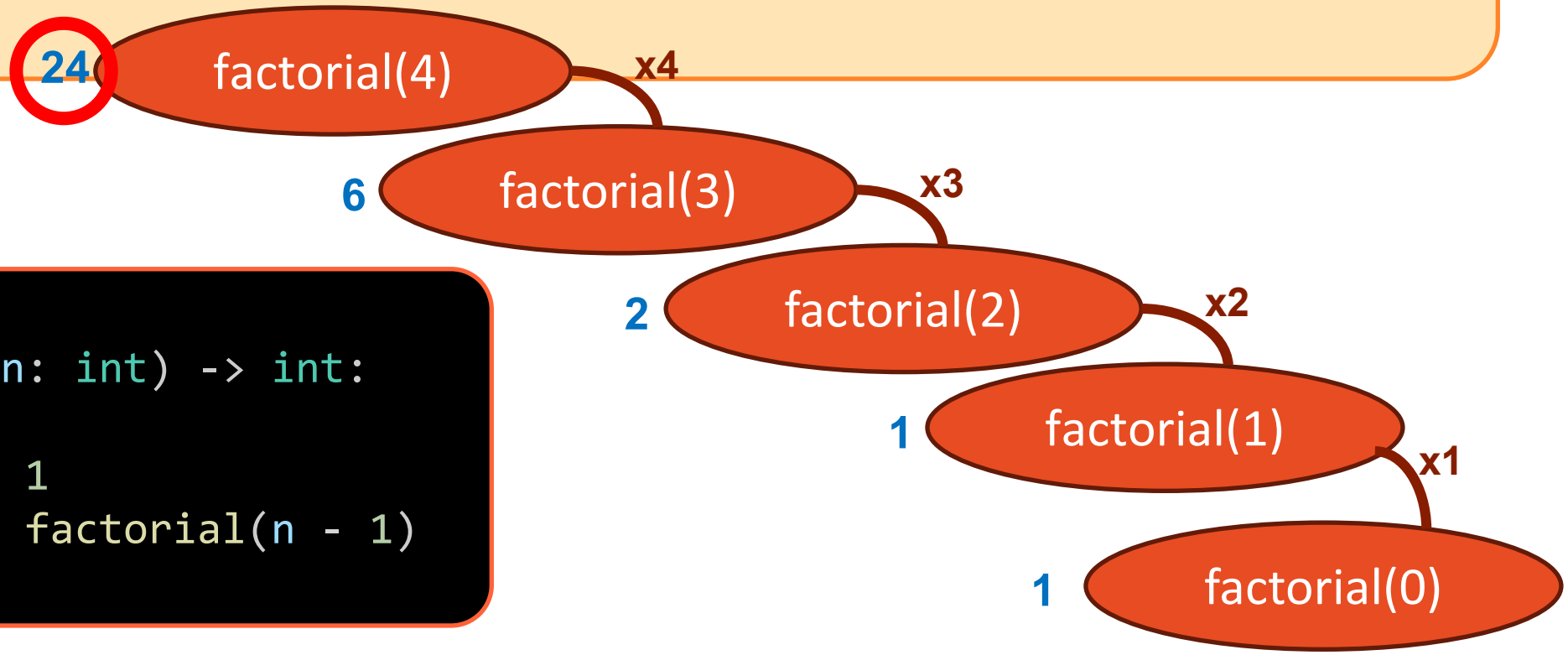
La recursivitat és una tècnica de resoldre un problema computacional en el que la solució depèn de solucions a instàncies més petites del mateix problema. Font: Wikipedia

El problema es redueix a versions més petites fins a arribar a un cas molt senzill que podem resoldre directament.

Exemple de càlcul del factorial

La recursivitat és una tècnica de resoldre un problema computacional en el que **la solució depèn de solucions a instàncies més petites** del mateix problema.

El problema es redueix a versions més petites **fins a arribar a un cas molt senzill** que podem resoldre directament.




```python
def factorial(n: int) -> int:
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Anatomia d’una funció recursiva

🧠 Quan resolem un problema amb recursivitat ens cal definir:

1. 🛑 El cas base: la condició que atura la recursió
2. 🔁 La crida recursiva: com avancem cap al cas base, fent el problema més petit cada vegada

⚠️ Errors típics:

- Oblidar el cas base: l’execució no pararà mai, serà infinita
- No reduir el problema correctament
- Pensar només en el primer pas sense tenir clar tot el problema

Ara et toca...

Escriu una funció que calculi x^n amb recursivitat. x i n enters.
Defineix quin és el cas base i com vas reduint el problema a cada crida.

La recursivitat no sempre és la millor solució...

Observem què passa amb les crides de Fibonacci

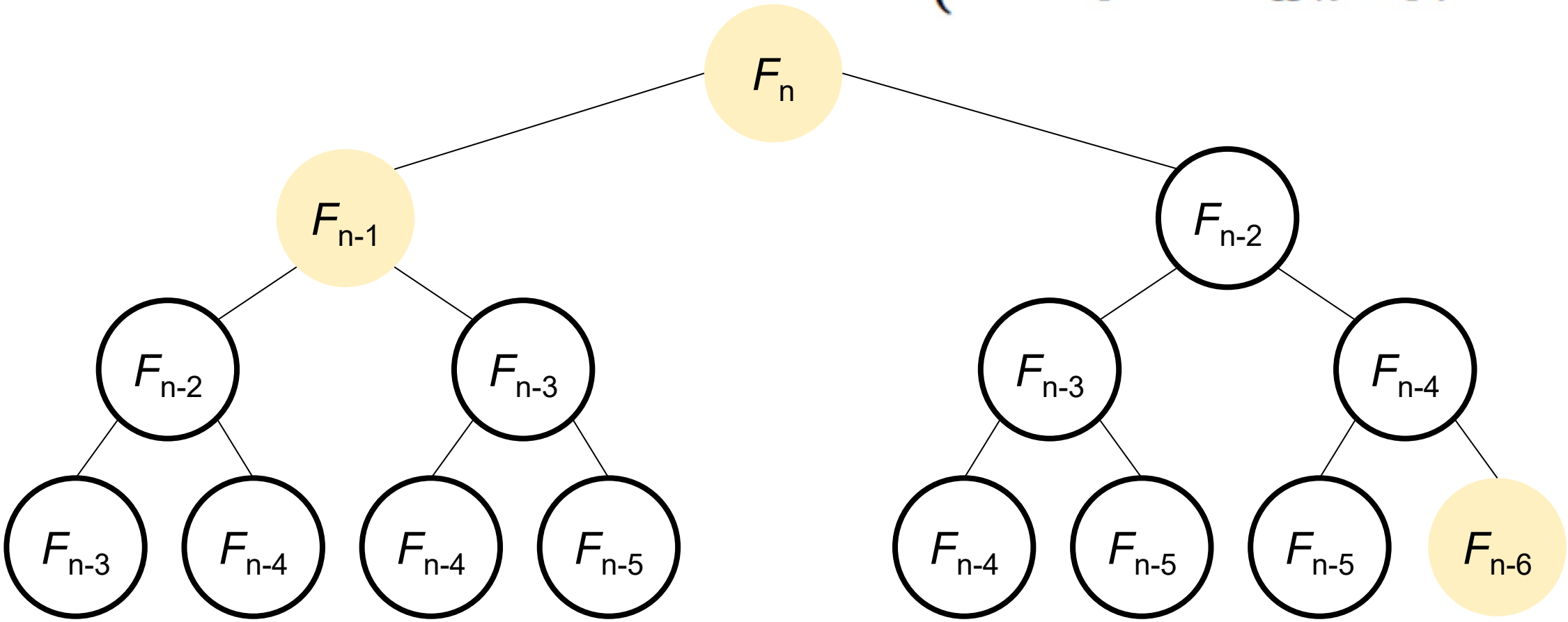
$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0\,. \end{cases}$$

```python
def fibonacci(n: int) -> int:
    if n == 0:
        return n
    if n == 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

La recursivitat no sempre és la millor solució...

Observem què passa amb les crides de Fibonacci

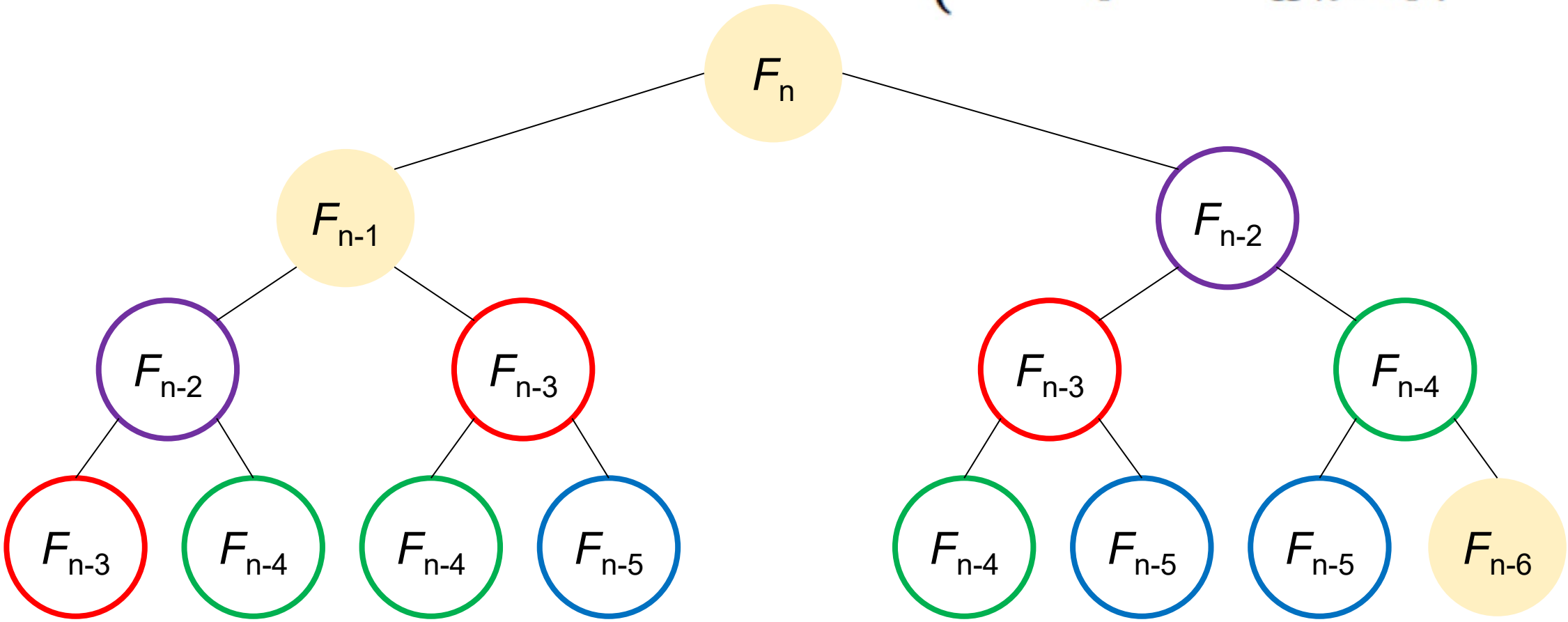
$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0\,. \end{cases}$$



La recursivitat no sempre és la millor solució...

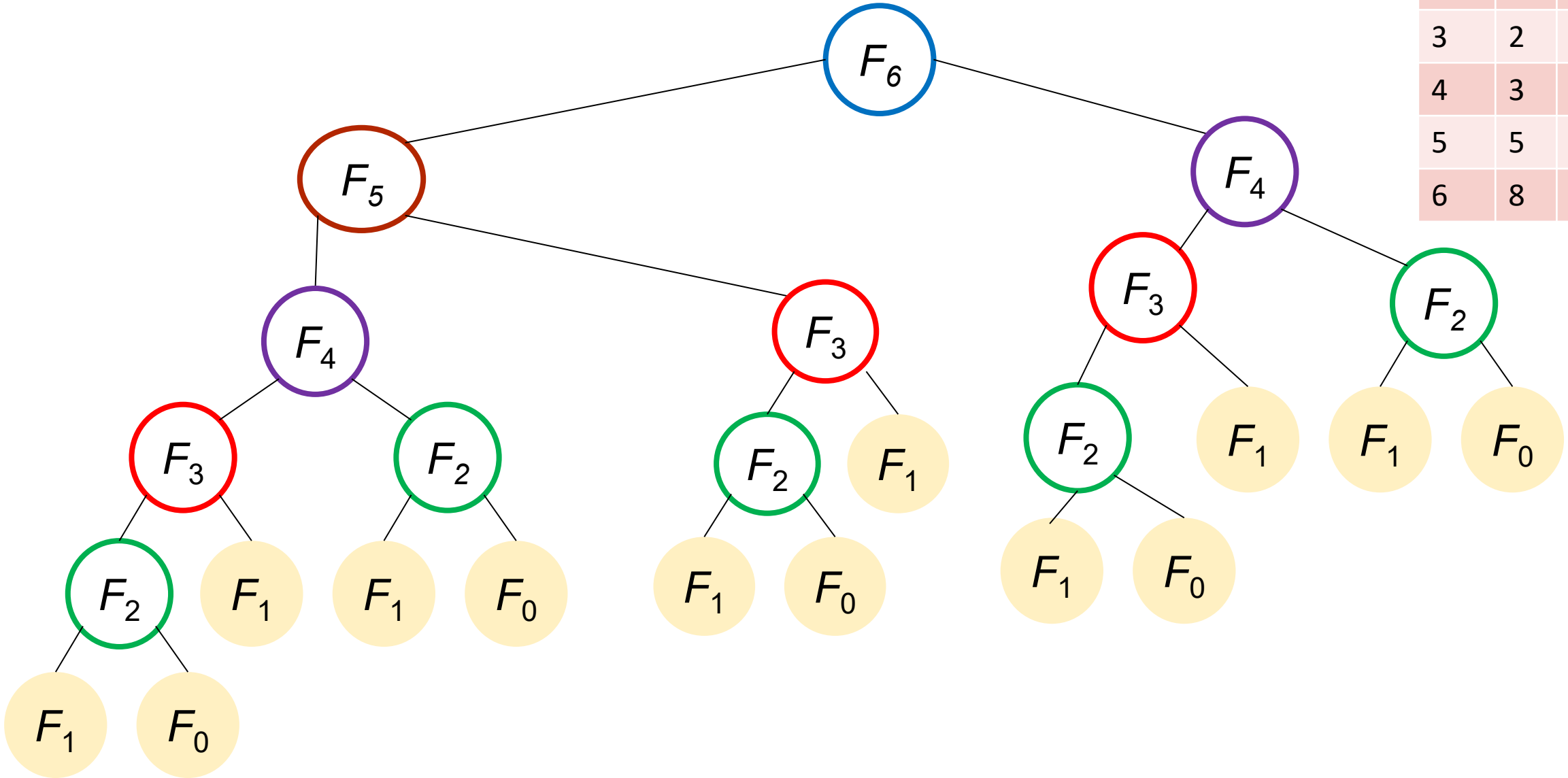
Observem què passa amb les crides de Fibonacci

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0\,. \end{cases}$$



Quants casos base hi ha per cada pas?

n	F	c.b
0	0	0
1	1	0
2	1	2
3	2	3
4	3	5
5	5	8
6	8	13



Podem millorar l'eficiència...

```python
def fibonacci_iteratiu(n: int) -> int:
    if n == 0:
        return 0

    seq: List[int] = [0, 1]
    for i in range(2, n + 1):
        seq.append(seq[i - 1] + seq[i - 2])

    return seq[n]
```

```python
def fibonacci_iteratiu_optimitzat(n: int) -> int:
    a: int = 0
    b: int = 1

    for i in range(1, n + 1):
        a, b = b, a + b

    return a
```

Fins i tot quan no hi ha repeticions...

```python
def factorial(n: int) -> int:
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

- La recursivitat ajuda a conceptualitzar un problema de manera elegant.
- Cal assegurar-se de definir el cas base i que a cada pas ens hi apropem.
- L'execució no sempre és eficient, si hi ha opció iterativa cal valorar avantatges i inconvenients tenint molt en compte l'eficiència.

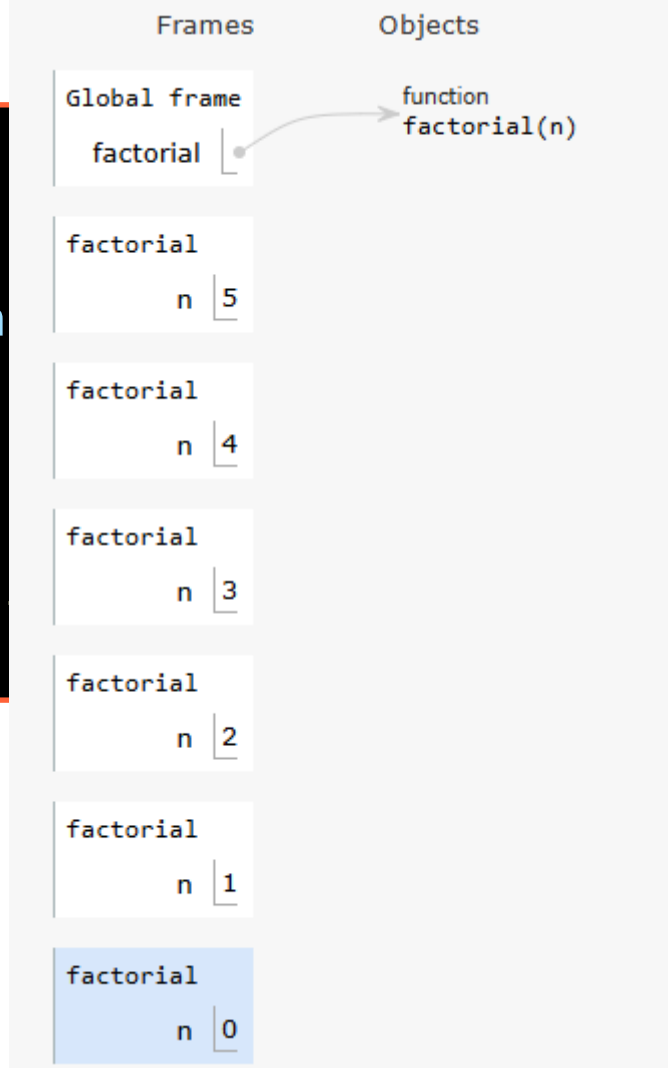
Què esta passant dins l'ordinador?

```python
# recursiu
def factorial(n: int) -> int:
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

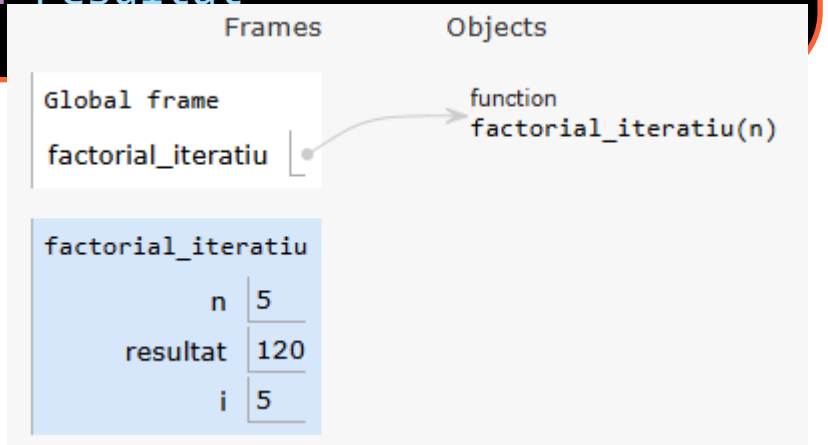
```python
# iteratiu
def factorial_iteratiu(n: int) -> int:
    resultat = 1
    for i in range(1, n + 1):
        resultat *= i
    return resultat
```

Què esta passant dins l'ordinador?

```
# recursiu
def factorial(n
    if n == 0:
        return
    return n *
```




```
# iteratiu
def factorial_iteratiu(n: int) -> int:
    resultat = 1
    for i in range(1, n + 1):
        resultat *= i
    return resultat
```



Notebook de suport

Compara les tres versions de Fibonacci: RecursiusTeoriaFibonacci.ipynb

Ara et toca...

Escriu una funció per comptar els elements d'una llista d'enters, i una altra per revertir una cadena. No pots fer servir funcions de Python.
De cadascuna fes-ne una versió recursiva i una iterativa i càlcula'n el cost.
Pots inspirar-te en el següent notebook de suport, suma d'una llista.

Més difícil...

Escriu una funció que donat un nombre enter retorni una cadena amb nombre parèntesis ben formats. Per exemple parèntesi(3) retornarà “((()))” fes-ne la versió recursiva i la iterativa.

Notebook de suport

Un exemple complet, la suma d'una llista: RecursiusTeoriaSumaLlista.ipynb

Gràcies i recapitulació

MIREIA RIBERA | ACCESSIBILITAT DIGITAL
| EXPERIÈNCIA D'USUARI
| VISUALITZACIÓ DE DADES

UNIVERSITAT DE BARCELONA

Recapitulació

✅ **Avantatges de la recursivitat**

✨ Codi més curt, més expressiu

📐 Més proper a la definició matemàtica

⚠️ **Inconvenients de la recursivitat**

💾 Sobrecàrrega de memòria (per la pila d'execució)

⏳ Sovint ofereix menor rendiment

Les versions iteratives dels algorismes solen ser més eficients, tot i que a vegades són menys elegants. Cal valorar avantatges i inconvenients en cada cas.