

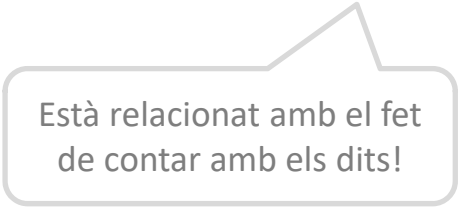
Algorísmica

Algorismes Numèrics

Mireia Ribera & Jordi Vitrià


Una mica d'història...

Cap a l'any 600, a l'Índia, es va inventar el **sistema decimal de numeració**.



Està relacionat amb el fet de contar amb els dits!

El seu principal avantatge sobre els que es coneixien a Europa, com el romà, és la seva base **posicional** i la **simplicitat** de les operacions (algorismes) aritmètiques.



Aquestes propietats estan compartides amb totes les bases!

Una mica d'història...

Un **sistema de numeració** és un conjunt de símbols i regles de generació que permeten construir tots els nombres vàlids en el sistema.

Un sistema de numeració ve definit doncs per:

- el conjunt S dels símbols permesos en el sistema.

En el cas del sistema decimal són $\{0,1...9\}$; en el binari són $\{0,1\}$; en l'octal són $\{0,1,...7\}$; en l'hexadecimal són $\{0,1,...9,A,B,C,D,E,F\}$

- el conjunt R de les regles de generació que ens indiquen quins nombres són vàlids i quins no són vàlids en el sistema.

Una mica d'història...

Els sistemes de numeració romans i egipcis no són estrictament posicionals. Per això, és molt complex dissenyar algoritmes d'ús general (per exemple, per a sumar, restar, multiplicar o dividir).



1	𐍇	10	𐍏	100	𐍑	1000	𐍓
2	𐍆𐍆	20	𐍏𐍏	200	𐍑𐍑	2000	𐍓𐍓
3	𐍆𐍆𐍆	30	𐍏𐍏𐍏	300	𐍑𐍑𐍑	3000	𐍓𐍓𐍓
4	𐍆𐍆𐍆𐍆	40	𐍏𐍏𐍏𐍏	400	𐍑𐍑𐍑𐍑	4000	𐍓𐍓𐍓𐍓
5	𐍆𐍆𐍆𐍆𐍆	50	𐍏𐍏𐍏𐍏𐍏	500	𐍑𐍑𐍑𐍑𐍑	5000	𐍓𐍓𐍓𐍓𐍓
6	𐍆𐍆𐍆𐍆𐍆𐍆	60	𐍏𐍏𐍏𐍏𐍏𐍏	600	𐍑𐍑𐍑𐍑𐍑𐍑	6000	𐍓𐍓𐍓𐍓𐍓𐍓
7	𐍆𐍆𐍆𐍆𐍆𐍆𐍆	70	𐍏𐍏𐍏𐍏𐍏𐍏𐍏	700	𐍑𐍑𐍑𐍑𐍑𐍑𐍑	7000	𐍓𐍓𐍓𐍓𐍓𐍓𐍓
8	𐍆𐍆𐍆𐍆𐍆𐍆𐍆𐍆	80	𐍏𐍏𐍏𐍏𐍏𐍏𐍏𐍏	800	𐍑𐍑𐍑𐍑𐍑𐍑𐍑𐍑	8000	𐍓𐍓𐍓𐍓𐍓𐍓𐍓𐍓
9	𐍆𐍆𐍆𐍆𐍆𐍆𐍆𐍆𐍆	90	𐍏𐍏𐍏𐍏𐍏𐍏𐍏𐍏𐍏	900	𐍑𐍑𐍑𐍑𐍑𐍑𐍑𐍑𐍑	9000	𐍓𐍓𐍓𐍓𐍓𐍓𐍓𐍓𐍓
Hieratic numerals							

Bases i representació de nombres

Quantes “unitats” hi ha a 642?

Simple!


$$600 + 40 + 2$$

642 és $600 + 40 + 2$ en **BASE 10**

La **base** d'un nombre determina el nombre de dígit
diferents i el valor de les posicions dels dígit.

Bases i representació de nombres

Fòrmula:

$$d_n * R^{n-1} + d_{n-1} * R^{n-2} + \dots + d_2 * R + d_1$$

R és la base del
nombre

$$642 \text{ is } 6_3 * 10^2 + 4_2 * 10^1 + 2_1$$

d és el dígit de la
ièssima posició
del nombre

Bases i representació de nombres

642 en base 13 és equivalent a 1068 en base 10

$$\begin{aligned} + 6 \times 13^2 &= 6 \times 169 = 1014 \\ + 4 \times 13^1 &= 4 \times 13 = 52 \\ + 2 \times 13^0 &= 2 \times 1 = 2 \\ &= 1068 \text{ in base 10} \end{aligned}$$

Bases i representació de nombres

Decimal és base 10 i té 10 dígit:

0,1,2,3,4,5,6,7,8,9

Binari és base 2 i té 2 dígit:

0,1

Per què un nombre existeixi en un sistema de numeració, el sistema ha d'incloure els seus dígit. Per exemple, el nombre 284 només existeix en base 9 i superiors.

La base 16 té 16 dígit: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E, and F

Bases i representació de nombres

Quants **dígits** necessitem per representar un nombre N en base b ?

Si tenim k dígits en base b podem representar els nombres fins a $b^k - 1$.

En el sistema digital, amb tres dígits podem representar fins $999 = 10^3 - 1$

Per tant, necessitem $\lceil \log_b (N + 1) \rceil \approx \lceil \log_b N \rceil$ dígits per escriure N en base b .

Resolem per k :
 $b^k - 1 = N$

Una mica d'història...

El **sistema decimal de numeració** va trigar molts anys en arribar a Europa.



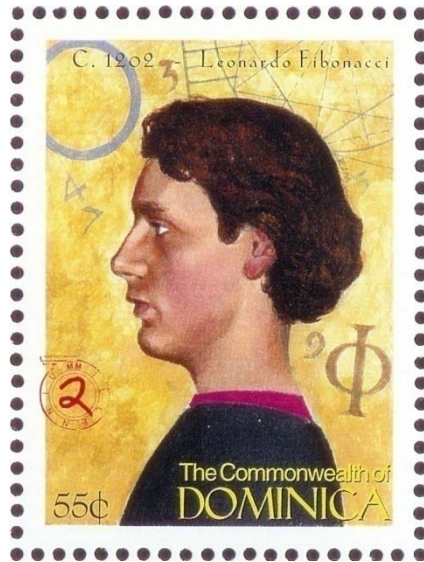
El medi de transmissió més important va ser un manual, escrit en àrab durant el segle IX a Bagdad, obra de **Al Khwarizmi**, en el que **especificava els procediments per sumar, multiplicar i dividir nombres escrits en base deu.**

Els procediments eren precisos, no ambigus, mecànics, eficients i correctes.

És a dir, eren algorismes (per a ser implementats sobre paper i no amb un ordinador!)

Una mica d'història...

Una de les persones que més van valorar aquesta aportació va ser **Leonardo Fibonacci**.



Fibonacci és avui conegut sobre tot per la seva seqüència:

0,1,1,2,3,5,8,13,21,34...

La seqüència es pot calcular amb la següent regla:

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

Això encara no és un algorisme. A les següents pàgines veurem diferents algorismes per implementar aquesta definició.

La seqüència de Fibonacci

La seqüència creix molt ràpid i es pot demostrar que

$$F_n \approx 2^{0.694n}$$

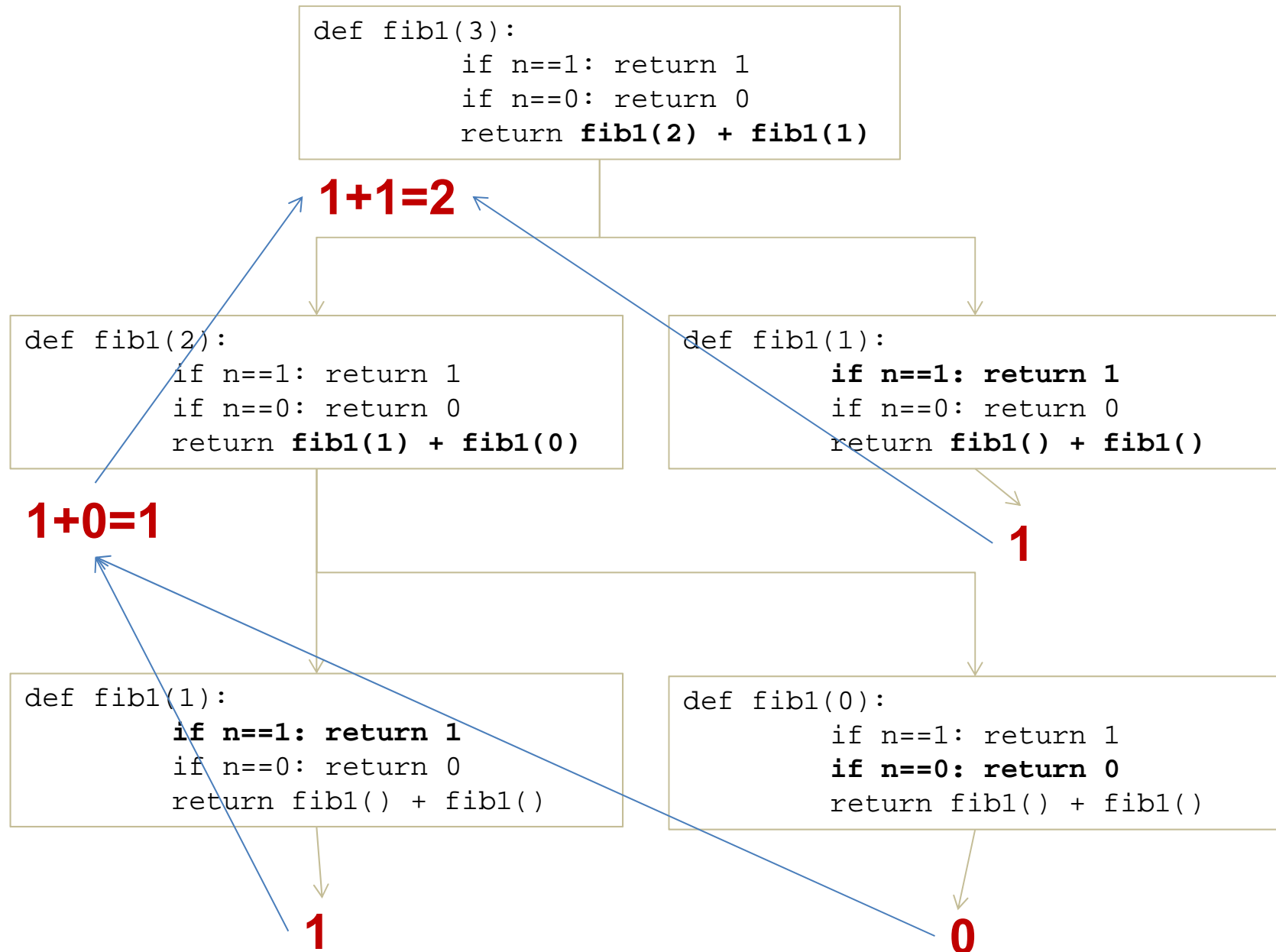
Però per calcular un terme concret necessitem un **algorisme**!

Una primera possibilitat és aquesta (**algorisme recursiu**):

```
def fib1(n):  
    if n==1:  
        return n  
    if n==0:  
        return n  
    return fib1(n-1) + fib1(n-2)  
  
fib1(10)
```

Els algorismes recursius són una família molt important dins del món de l'algorísmica, que es caracteritzen per "cridar-se" a ells mateixos.

55



La seqüència de Fibonacci

Les tres preguntes
bàsiques del l'algorísmica!

Com per a qualsevol algorisme, ens podem fer **tres preguntes**:

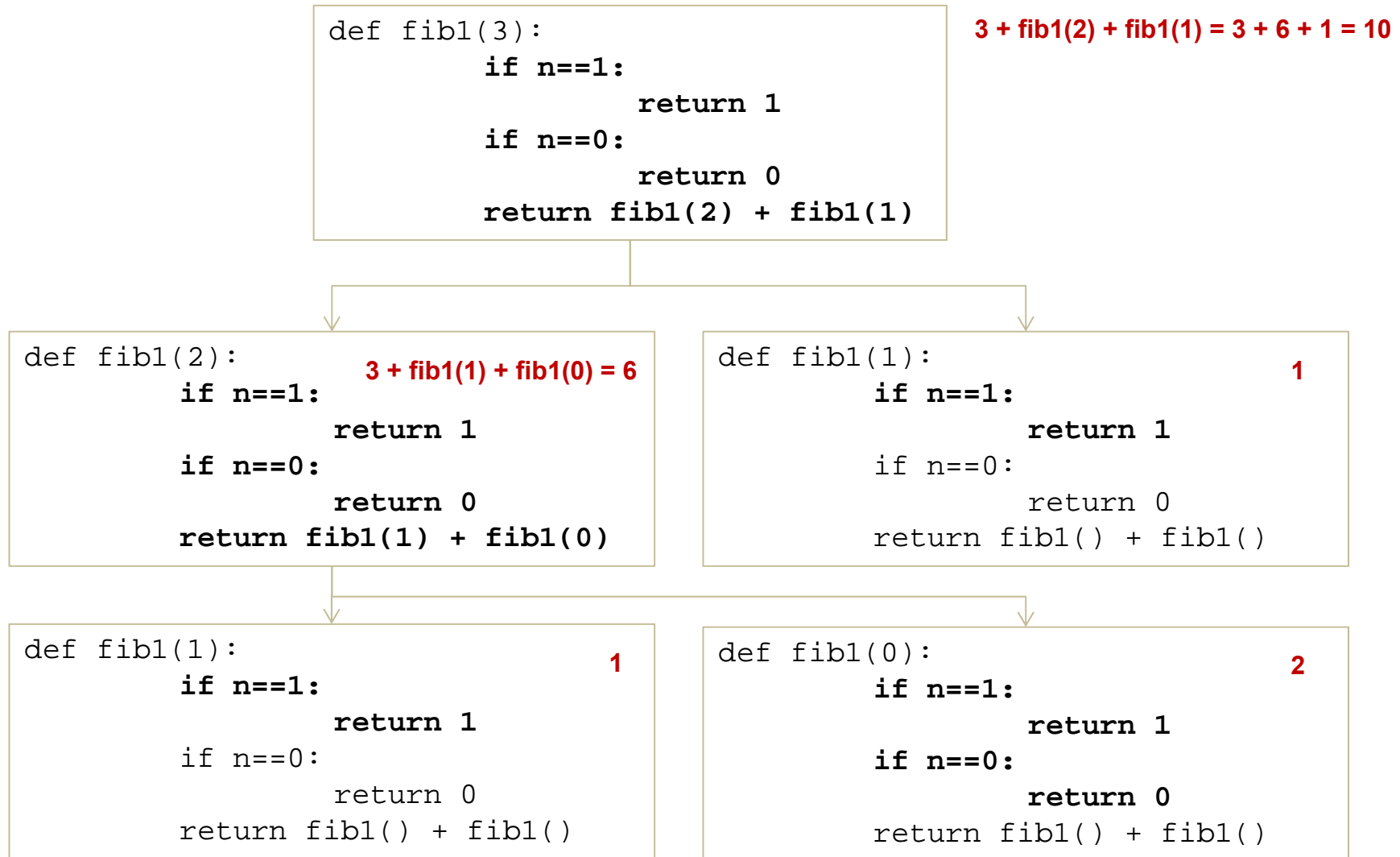
1) És correcte?

En aquest cas és evident,
atès que segueix
exactament la definició!

2) Quant trigarà, en funció de n ?

3) Hi ha alguna manera millor de fer-ho?

La seqüència de Fibonacci: quants passos fa?



La seqüència de Fibonacci

Per exemple, per calcular F_{200} es pot veure que l'algorisme executa al voltant de 2^{138} passos.

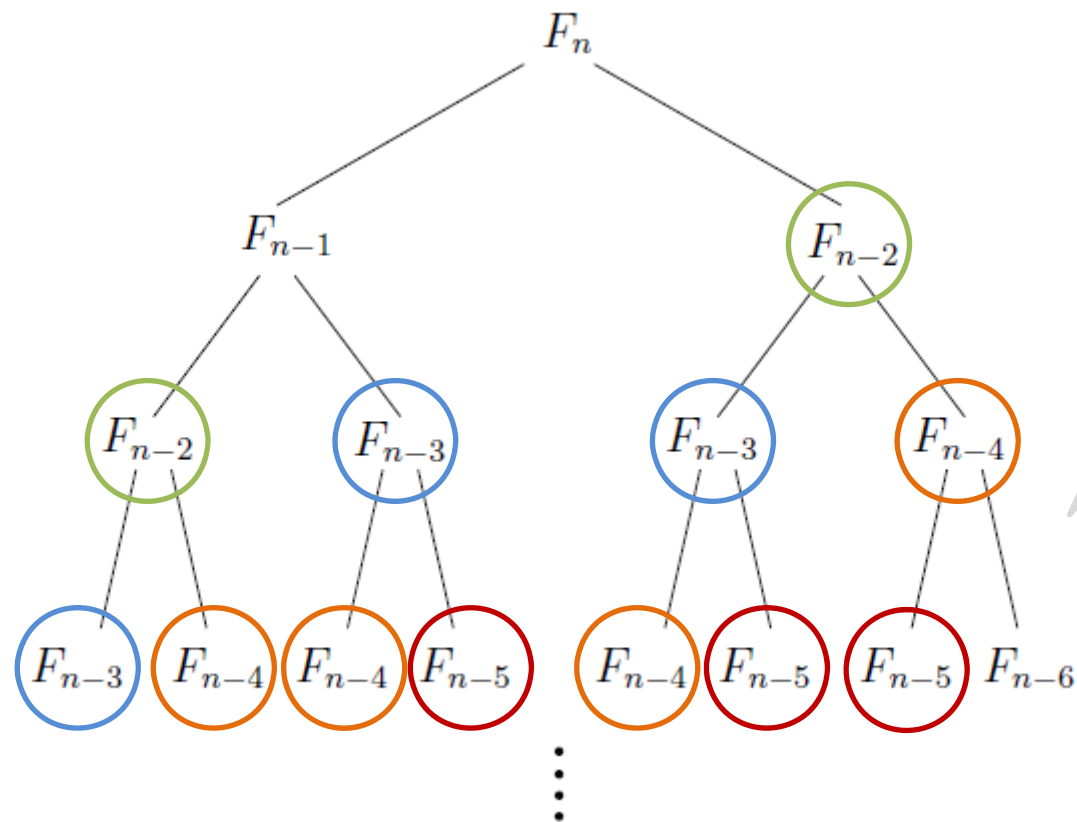
A l'ordinador més ràpid del món, que pot executar al voltant de 40.000.000.000.000 passos per segon, necessitaríem més temps que el necessari pel col·lapse del Sol!

A la velocitat que els ordinadors augmenten la seva capacitat de càlcul, cada any que passa podríem calcular un nombre de Fibonacci més que l'any anterior!

Aquesta dada ens fa adonar de la importància de la tercera pregunta: **es pot fer millor?**

La seqüència de Fibonacci

Per què l'algorisme `fib1(n)` és tant lent?



Hi ha molts
càlculs que es
repeteixen!

**Perquè no
guardar-los?**

La seqüència de Fibonacci

```
def fib2(n):  
    if n==0:  
        return 0  
    ls = [0,1]  
    for i in range(2,n+1):  
        ls.append(ls[i-1]+ls[i-2])  
    return ls[n]
```

```
fib2(10)
```

55

1. És evident que és correcte.
2. Només executa $(n-1)$ vegades la iteració.

fib2(n) és **lineal (o polinòmic)** respecte n.
Ara podem calcular fins i tot F(100.000.000)!

La seqüència de Fibonacci

Inici →
 Assignacions →
 Iteració {

```
def fib2(n):
    if n==0:
        return 0
    ls = [0,1]
    for i in range(2,n+1):
        ls.append(ls[i-1]+ls[i-2])
    return ls[n]
```

Inici						
Assignacions	0	1				
Iteració 1	0	1	1			
Iteració 2	0	1	1	2		
Iteració 3	0	1	1	2	3	
Iteració 4	0	1	1	2	3	5

La seqüència de Fibonacci

```
def fib2(n):  
    if n==0:  
        return 0  
    ls = [0,1]  
    for i in range(2,n+1):  
        ls.append(ls[i-1]+ls[i-2])  
    return ls[n]
```

```
fib2(1000)
```

434665576869374564356885276750406258025646605173717804024817290895365554
179490518904038798400792551692959225930803226347752096896232398733224711
61642996440906533187938298969649928516003704476137795166849228875

```
def fib3(n):
    a,b = 0,1
    for i in range(1,n+1):
        a,b = b, a+b
    return a

fib3(10)
```

55

a	b
0	1
1	1
1	2
2	3
3	5

En aquest cas no només hem minimitzat el cost computacional sinó també l'espai necessari per calcular-ho!

```
In [5]: def fib3(n):  
        a,b = 0,1  
        for i in range(1,n+1):  
            a,b = b, a+b  
        return a  
  
fib3(100000)
```

```
Out[5]: 2597406934722172416615503402127591541488048538651769658472477070395253  
4543511273686265556772836716744754637587223074432111638399473875091030  
9656973821883044930522876385313349213530267927895670105127657827163560  
8073050532200243233114383986516137827238124777453778337299916214634050  
0546698603908627509966393664092118901252719601721050603003505868940285  
5810367511765825136837743868493641345733883436515877542537191241050033  
2195991330062204363035213756525421823998690848556374080179251761629391  
7549634585586163007628199160811098365263529954406942842065710460449038  
0564713634603300052085227770755444679472370903097901901486043284681985  
7961015951001850608264919234587313399150133919932363102301864172536477  
1362664750801339824312317034314529641817900511879573167668349799016820  
1184990775668645684506628739248560391404760519955006628882634587718941  
0680370091879365001733011710028310473947456256091444932821374855573864  
0805798130282666402703542944121049199958031318768058991865134251759599  
1152056315533770399694103551827527491995980225750790203779810308992298  
4996304496255814045517000250299764322193462165366210841876745428298261  
3982344783665815880408190033073829395000821320093747154851310272208173  
0543226486694963098791471436292555425262404399961532697987680751064681  
9068792118299167964409178271868561702918102212679267401362650499784968  
8436809752547001310045741864064482994858725517447466956518791269169932
```

Com hem de contar els *passos computacionals*?

Considerarem de la **mateixa categoria** les instruccions simples com emmagatzemar a memòria, *branching*, comparacions, operacions aritmètiques, etc.

Que ocupen més de 32/64 bits

Però si manipulem **nombres molt grans**, aquestes operacions no són tant barates!

Caldrà tenir en compte quina complexitat computacional té operar dos nombres d'aquestes característiques.

La notació Gran O

Aquesta notació és una convenció per no ser ni massa ni massa poc precisos a l'hora d'escriure la complexitat computacional d'un algorisme (=nombre de passos).

La regla principal és contar el **nombre de passos computacionals aproximats en funció de la mida de la entrada.**

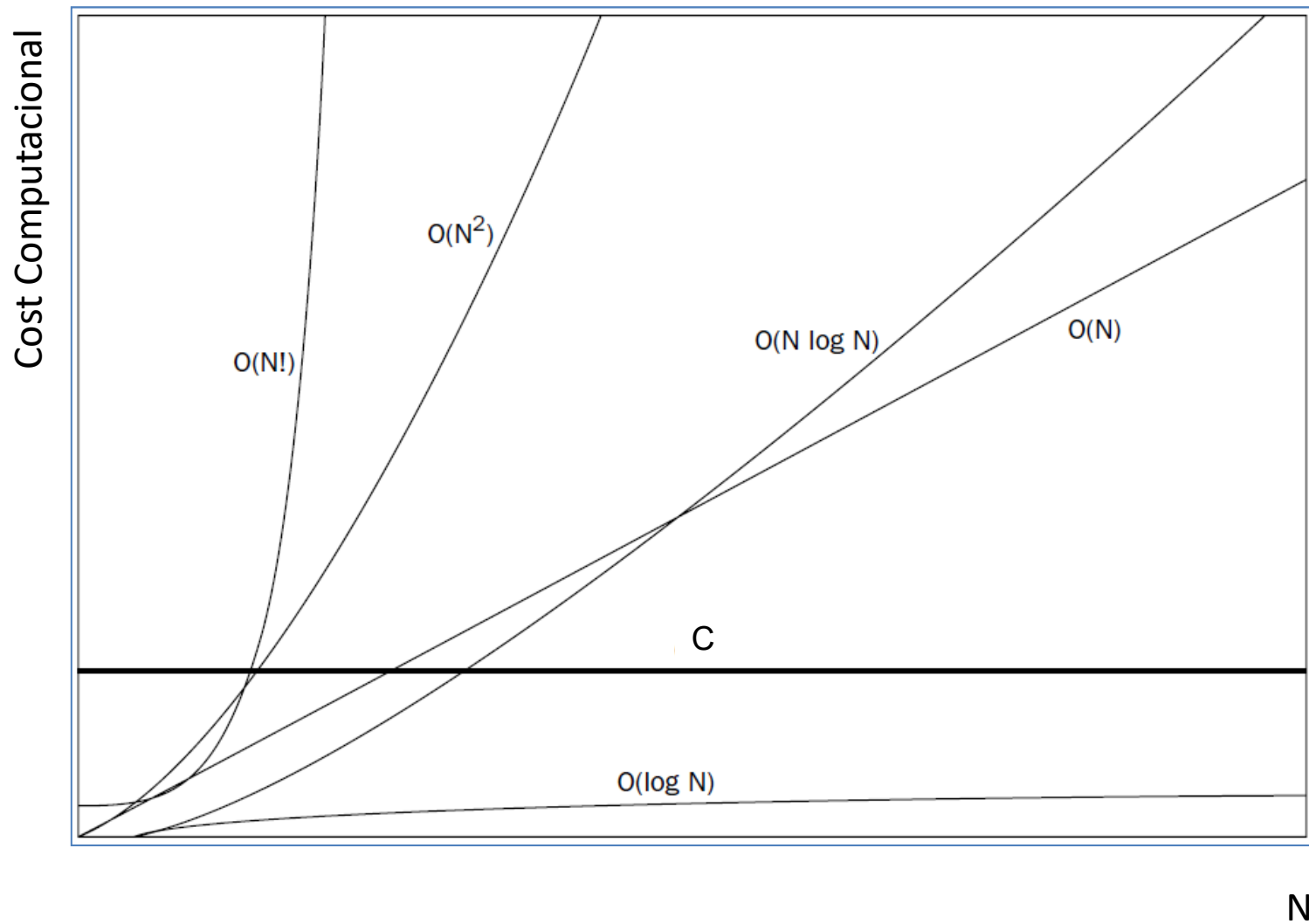
Fem la següent aproximació: enlloc de dir que pren $5n^3+4n+3$ direm que pren $O(n^3)$

La notació Gran O

En general utilitzarem aquestes convencions:

1. Ometrem les constants multiplicatives: $14n^2$ és n^2
2. n^a domina sobre n^b si $a > b$: n^2 domina sobre n
3. Qualsevol exponencial domina sobre un polinomi: 3^n domina sobre n^5 (i també sobre 2^n)!
4. Qualsevol polinomi domina sobre un logaritme: n domina sobre $(\log n)^3$ i n^2 domina sobre $(n \log n)$

La notació Gran O



La notació Gran O

N	N^2	$N!$
5	25	120
6	36	720
7	49	5,040
8	64	40,320
9	81	362,880
10	100	3,628,800

Observacions:

- Qualsevol algorisme amb $n!$ és inútil a partir de $n=20$
- Els algorismes amb 2^n són inútils a partir de $n=40$
- Els algorismes quadràtics, n^2 , comencen a ser costosos a partir de $n=10.000$ i a ser inútils a partir de $n=1.000.000$
- Els algorismes lineals i els $n \log n$ poden arribar fins a $n=1.000.000.000$
- Els algorismes sublineals, $\log n$, són útils per qualsevol n .

La notació Gran O

Les famílies més importants d'algorismes són les que tenen un ordre:

- **Constant**, $O(n) = 1$, com $f(n) = \min(n,1)$, que no depenen de n .
- **Logarítmic**, $O(n) = \log n$.
- **Lineals**, $O(n) = n$.
- **Super-lineals**, $O(n) = n \log n$.
- **Quadràtics**, $O(n) = n^2$.
- **Cúbics**, $O(n) = n^3$.
- **Exponencials**, $O(n) = c^n$ per $c > 1$.
- **Factorials**, $O(n) = n!$

} **Polinòmics**

Aritmètica Bàsica

Aritmètica Bàsica

Quan definim una operació aritmètica, dependrà el seu cost de la base en que està escrit el nombre?

Abans hem vist que necessitem $\lceil \log_b(N+1) \rceil \approx \lceil \log_b N \rceil$ dígit per escriure un nombre N en base b .

Aquesta fórmula ens indica que quan fem un canvi de base la mida del nombre només canvia lleugerament (un factor multiplicatiu) i per tant, des del punt de vista de la notació $O()$, considerarem que les operacions aritmètiques tenen el mateix cost en totes les bases!

```
import math
for i in range(2,16):
    print(i,":", math.ceil(math.log(23463453452562346,i)))
```

```
2 : 55
3 : 35
4 : 28
5 : 24
6 : 22
7 : 20
8 : 19
9 : 18
10 : 17
11 : 16
12 : 16
13 : 15
14 : 15
15 : 14
```

Aritmètica Bàsica

Aquesta propietat es compleix
per totes les bases $b \geq 2$

Hi ha una propietat útil dels nombres decimals:

**La suma de tres nombres d'un sol dígit qualsevol
té com a màxim dos dígit.**

Aquesta regla ens permet definir una regla general per **sumar**
dos nombres en qualsevol base: la que hem après a
l'escola!

$$\begin{array}{rcccccc} \text{Carry:} & 1 & & & 1 & 1 & 1 & \\ & & 1 & 1 & 0 & 1 & 0 & 1 & (53) \\ & & 1 & 0 & 0 & 0 & 1 & 1 & (35) \\ \hline & 1 & 0 & 1 & 1 & 0 & 0 & 0 & (88) \end{array}$$

Aritmètica Bàsica

És funció de n : el nombre de bits de x i y

Quina complexitat té aquest algorisme?

Suposem que tant x com y tenen n bits. La seva suma té com a màxim $n+1$ bits. **La seva complexitat és per tant, $O(n)$.**

Per un nombre petit de bits, l'ordinador ho pot fer en un sol pas, però això no és veritat per a nombres molt grans.

Es pot fer millor? No! Per sumar n bits com a mínim s'han de poder llegir i escriure, i això ja són $2n$ passos!

Aritmètica Bàsica

La multiplicació o producte que ens han ensenyat a l'escola:

				1	1	0	1	(binary 13)		
				×	1	0	1	1 (binary 11)		
<hr/>										
					1	1	0	1 (1101 times 1) (binary 13)		
					1	1	0	1 (1101 times 1, shifted once) (binary 26)		
			0	0	0	0		(1101 times 0, shifted twice) (binary 52)		
		+	1	1	0	1		(1101 times 1, shifted thrice) (binary 104)		
<hr/>										
			1	0	0	0	1	1	1	1 (binary 143)

L'algorisme és la suma (amb desplaçament) d'una sèrie de multiplicacions d'un bit.

Tenim (**n multiplicacions de complexitat n** (un bit per n bits) + **una suma de complexitat 2n**) = $n^2 + 2n$ = **la complexitat total és $O(n^2)$**

Aritmètica Bàsica

Al Khwarizmi ens va donar un segon algorisme (i que avui encara s'utilitza en uns quants països!)

Escrivim els nombres un al costat de l'altre

Repetim "Dividim el primer per dos i l'arrodonim. Doblem el segon fins que el primer nombre és 1".

Sumem els nombres de la segona columna que corresponen a totes les files on el nombre de la primera columna és senar i obtenim el resultat.

11	13
5	26
2	52
1	104
<hr/>	
	143

Això no és diferent del cas anterior: els nombres de la segona columna que sumem corresponen als nombre binaris que sumàvem abans!

Aritmètica Bàsica

Aquest algorisme es pot escriure de varies maneres. Una d'elles és recursiva:

$$x \cdot y = \begin{cases} 2(x \cdot \lfloor y/2 \rfloor) & \text{Si } y \text{ és parell} \\ x + 2(x \cdot \lfloor y/2 \rfloor) & \text{Si } y \text{ és senar} \end{cases}$$

L'algorisme s'acaba després de n crides recursives (anem dividint per 2) i a cada crida fem $O(n)$ operacions. **Per tant és $O(n^2)$**

```
1 def mult(x,y):
2     import math
3     if y == 0:
4         return 0
5     z = mult(x,math.floor(y/2))
6     if y%2 == 0:
7         return 2*z
8     else:
9         return x+2*z
10
```

Multiplicar/dividir
per 2 i sumar té
complexitat n

```
mult(100,10)
```

1000

Aritmètica Bàsica

La divisió x/y consisteix en trobar un quocient q i una resta r de manera que $x=y \times q + r$ i $r < y$.

La seva versió recursiva és:

```
def div(x,y):  
    import math  
    if x==0:  
        return 0,0  
    q,r = div(math.floor(x/2),y)  
    q = 2 * q  
    r = 2 * r  
    if x%2 != 0:  
        r += 1  
    if r >= y:  
        r = r-y  
        q = q+1  
    return q,r
```

La seva complexitat és $O(n^2)$

```
div(100,10)
```

```
(10, 0)
```