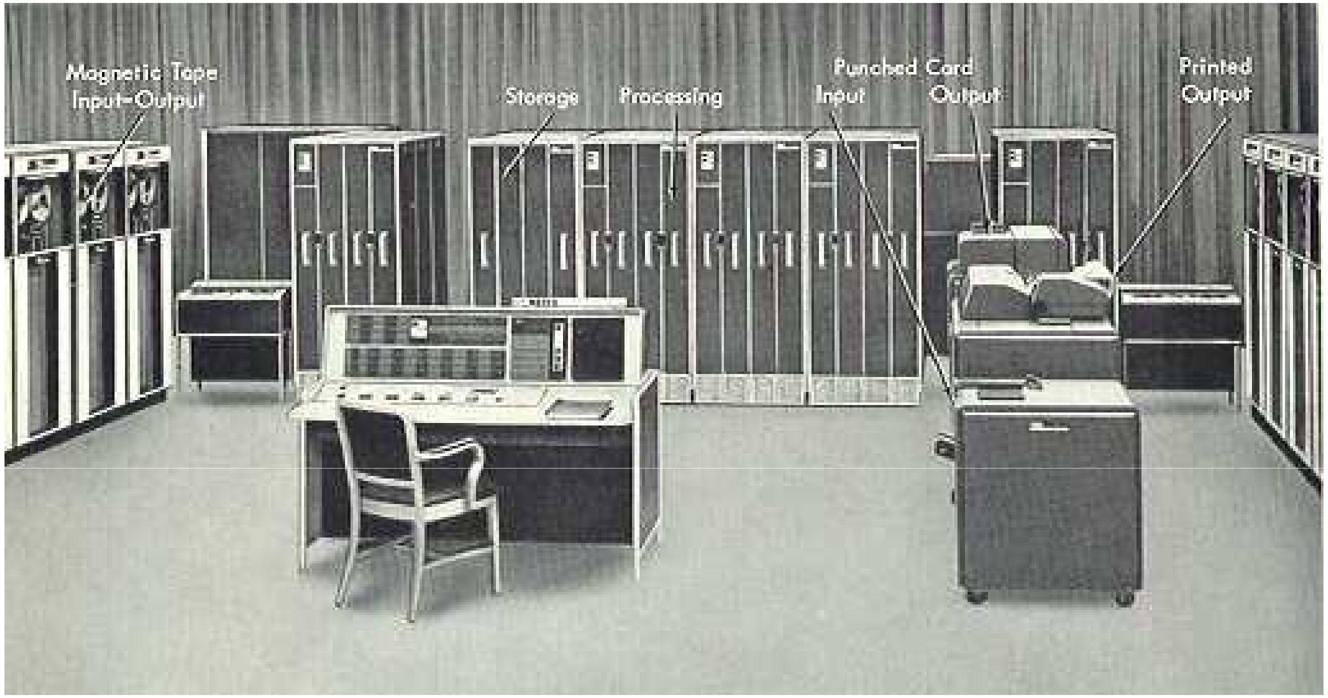
UNIVERSITAT DE BARCELONA

Grau d'Enginyeria Informàtica
Facultat de Matemàtiques



Algorísmica

Algorismes Numèrics

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.

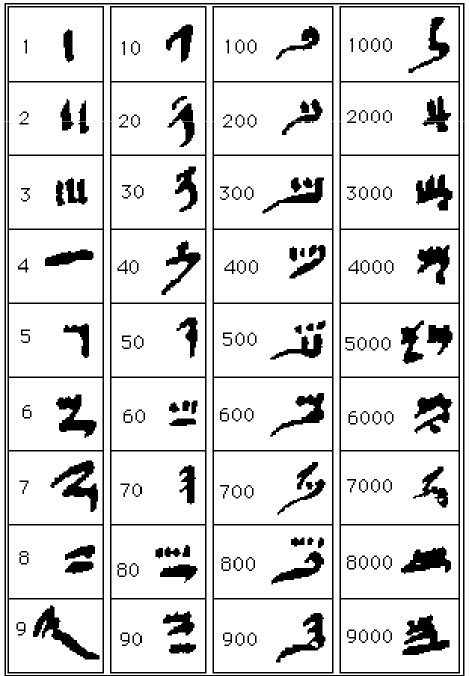
Un sistema de numeració ve definit doncs per:

• 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).



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 dígits i el valor de les posicions dels dígits.

Bases i representació de nombres

R és la base del nombre

Fòrmula:

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

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

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 dígits:

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

Binari és base 2 i té 2 dígits:

0,1

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

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

Bases i representació de nombres

Quina és la notació decimal equivalent del nombre octal 642?

$$
\begin{aligned}
& 6 \times 8^2 = 6 \times 64 = 384 \\
+ & 4 \times 8^1 = 4 \times 8 = 32 \\
+ & 2 \times 8^0 = 2 \times 1 = 2 \\
& = 418 \text{ en base } 10
\end{aligned}
$$

Bases i representació de nombres

Quina és la notació decimal equivalent del nombre hexadecimal DEF?

$$
\begin{aligned}
& D \times 16^2 = 13 \times 256 = 3328 \\
+ \; & E \times 16^1 = 14 \times 16 = 224 \\
+ \; & F \times 16^0 = 15 \times 1 = 15 \\
& = 3567 \text{ en base } 10
\end{aligned}
$$

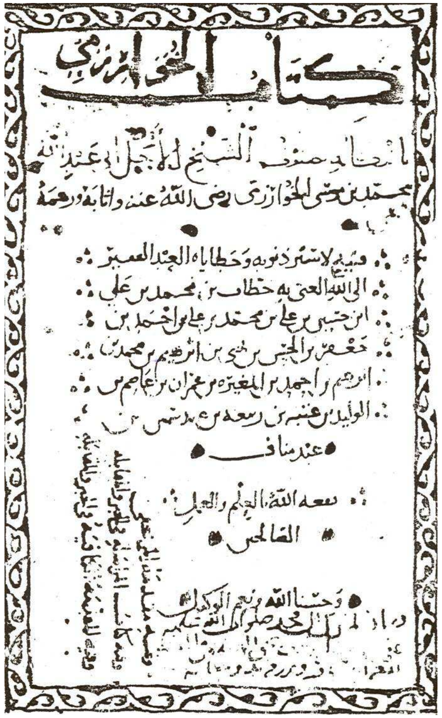
Bases i representació de nombres

Quin és el decimal equivalent del binari 1101110?

$$
\begin{aligned}
& 1 \times 2^6 &= \; & 1 \times 64 &= 64 \\
+\; & 1 \times 2^5 &= \; & 1 \times 32 &= 32 \\
+\; & 0 \times 2^4 &= \; & 0 \times 16 &= 0 \\
+\; & 1 \times 2^3 &= \; & 1 \times 8 &= 8 \\
+\; & 1 \times 2^2 &= \; & 1 \times 4 &= 4 \\
+\; & 1 \times 2^1 &= \; & 1 \times 2 &= 2 \\
+\; & 0 \times 2^o &= \; & 0 \times 1 &= 0 \\
& & & &= 110 \text{ in base } 10
\end{aligned}
$$

Una mica d’història...

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


كتاب الخوارزمي

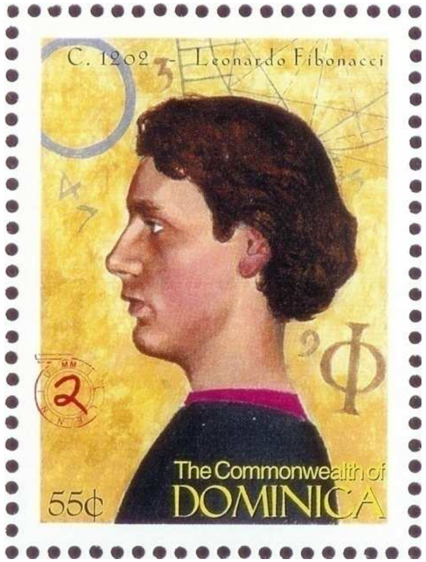
El medi de transmissió més important va ser un manual, escrit en àrab durant el segle IX a Bagdad, obra de **Al Khwarizmi**, en el que **especificava els procediments per sumar, multiplicar i dividir nombres escrits en base deu**.

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**!

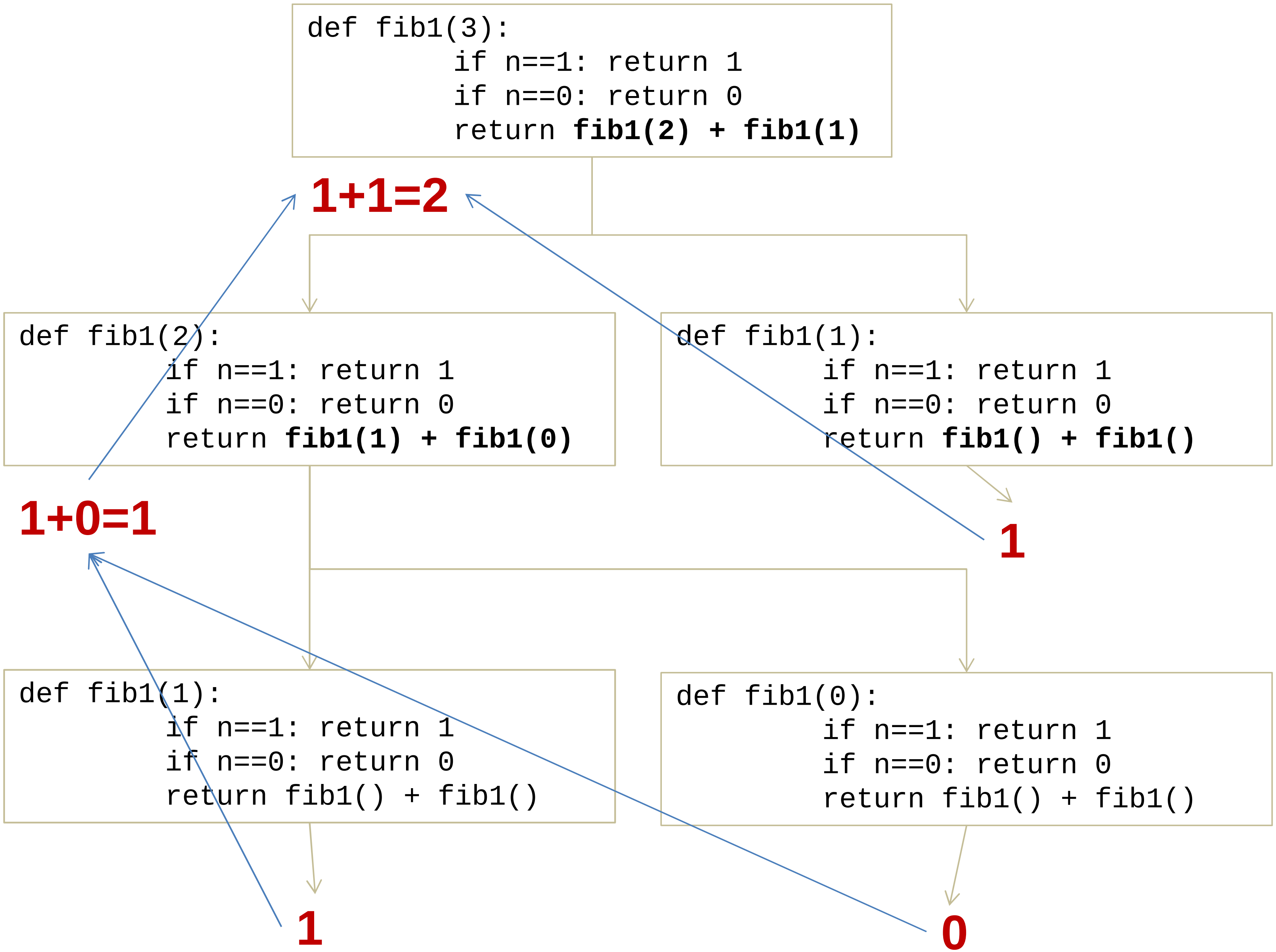
Una primera possibilitat és aquesta (**algorisme recursiu**):

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

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

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

>>> fib1(10)
55
```


def fib1(3):
if n==1: return 1
if n==0: return 0
return **fib1(2) + fib1(1)**
1+1=2
def fib1(2):
if n==1: return 1
if n==0: return 0
return **fib1(1) + fib1(0)**
def fib1(1):
if n==1: return 1
if n==0: return 0
return **fib1() + fib1()**
1+0=1
1
def fib1(1):
if n==1: return 1
if n==0: return 0
return fib1() + fib1()
def fib1(0):
if n==1: return 1
if n==0: return 0
return fib1() + fib1()
1
0

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

Sigui $T(n)$ el nombre de "**passos computacionals**" que ha de fer l'algorisme `fib1(n)`.

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

1. És evident que T(0)=1 i T(1)=2.
2. També ho és que si $n>1$, $T(n) = T(n-1) + T(n-2) + 3$

I per tant, $T(n) >= F_n$ i **sabem que F_n creix molt ràpid!**

El cost creix segons la fórmula de la seqüència de Fibonacci!

$T(n)$ és **exponencial** respecte n

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

La seqüència de Fibonacci

```
def fib1(3):
        if n==1:
                return 1
        if n==0:
                return 0
        return fib1(2) + fib1(1)
```

3 + fib1(2) + fib1(1) = 3 + 6 + 1 = 10

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

3 + fib1(1) + fib1(0) = 6

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

1

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

1

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

2

La seqüència de Fibonacci

Per exemple, per calcular **F_{200}**, l'algorisme executa T(200) >= F_{200} >= **2^{138} passos**.

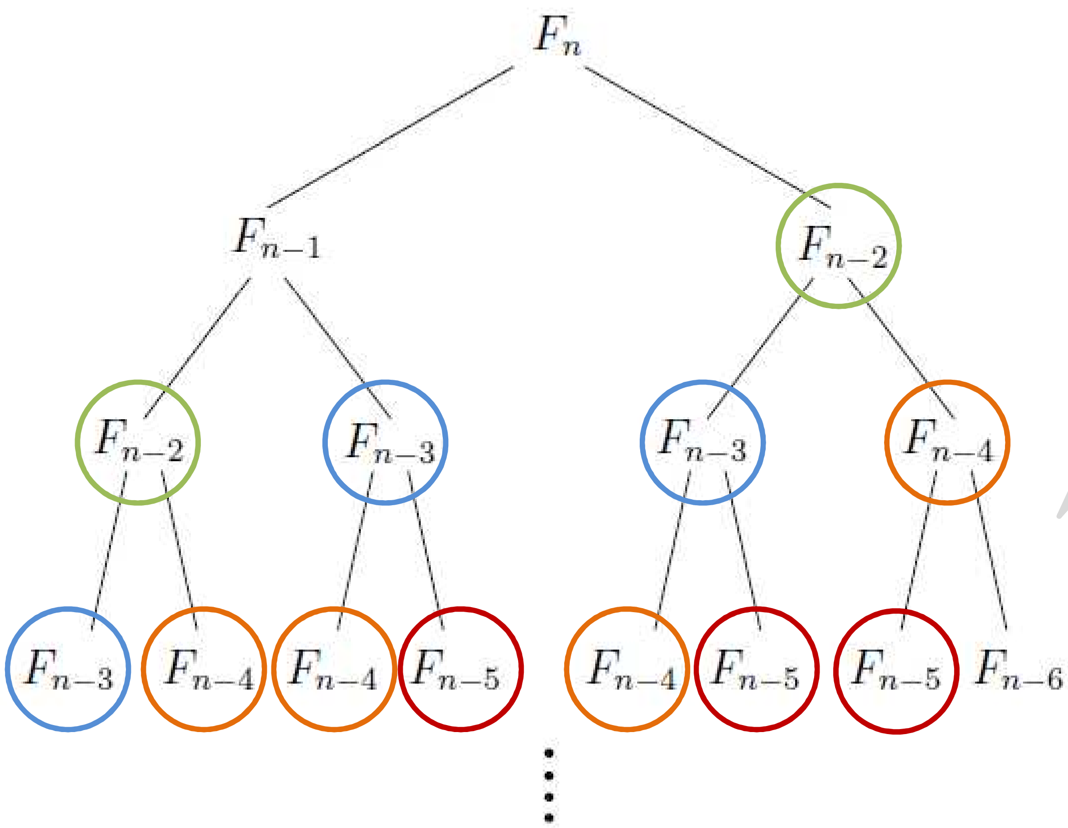
A l'ordinador més ràpid del món, que pot executar al voltant de 40.000.000.000.000 passos per segon, necessitaríem més temps que el necessari pel col·lapse del Sol!

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

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i-1] + f[i-2]
return f[n]
```

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

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

La seqüència de Fibonacci

Inici

Assignacions

Iteració

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i-1] + f[i-2]
return f[n]
```

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

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

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!

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!

F_n té aproximadament $0{,}694n$ bits.

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

Més endavant veurem que sumar dos nombres de n bits té una complexitat lineal respecte n. Per tant, el cost computacional de fib1(n) és de nF_n i el de fib2(n) és de n^2

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**.

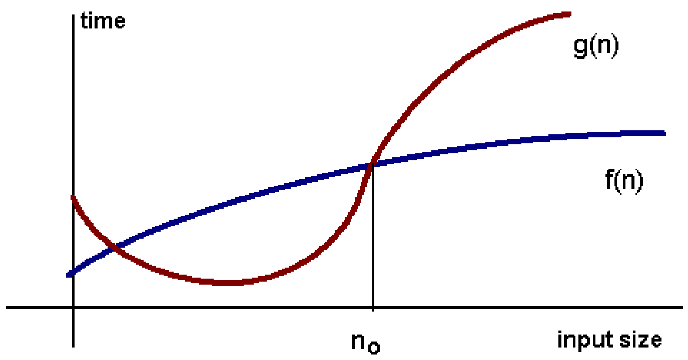
Fem la següent aproximació: enlloc de dir que pren $5n^3+4n+3$ direm que pren $O(n^3)$

La notació Gran O

Per tant
$10n=O(n)$

Més concretament:

Siguin $f(n)$ i $g(n)$ dues funcions dels enters positius als reals positius.
Direm que $f=O(g)$ (que vol dir que "f no creix més ràpid que g") si existeix una constant $c>0$ i un valor n_0 tals que $f(n) <= c \cdot g(n)$ per tot $n>n0$.



La notació Gran $\mathbf{O}$

A partir d'aquí podem definir els conceptes complementaris (>= i =):

$$f=\Omega(g) \text{ si } g=O(f)$$

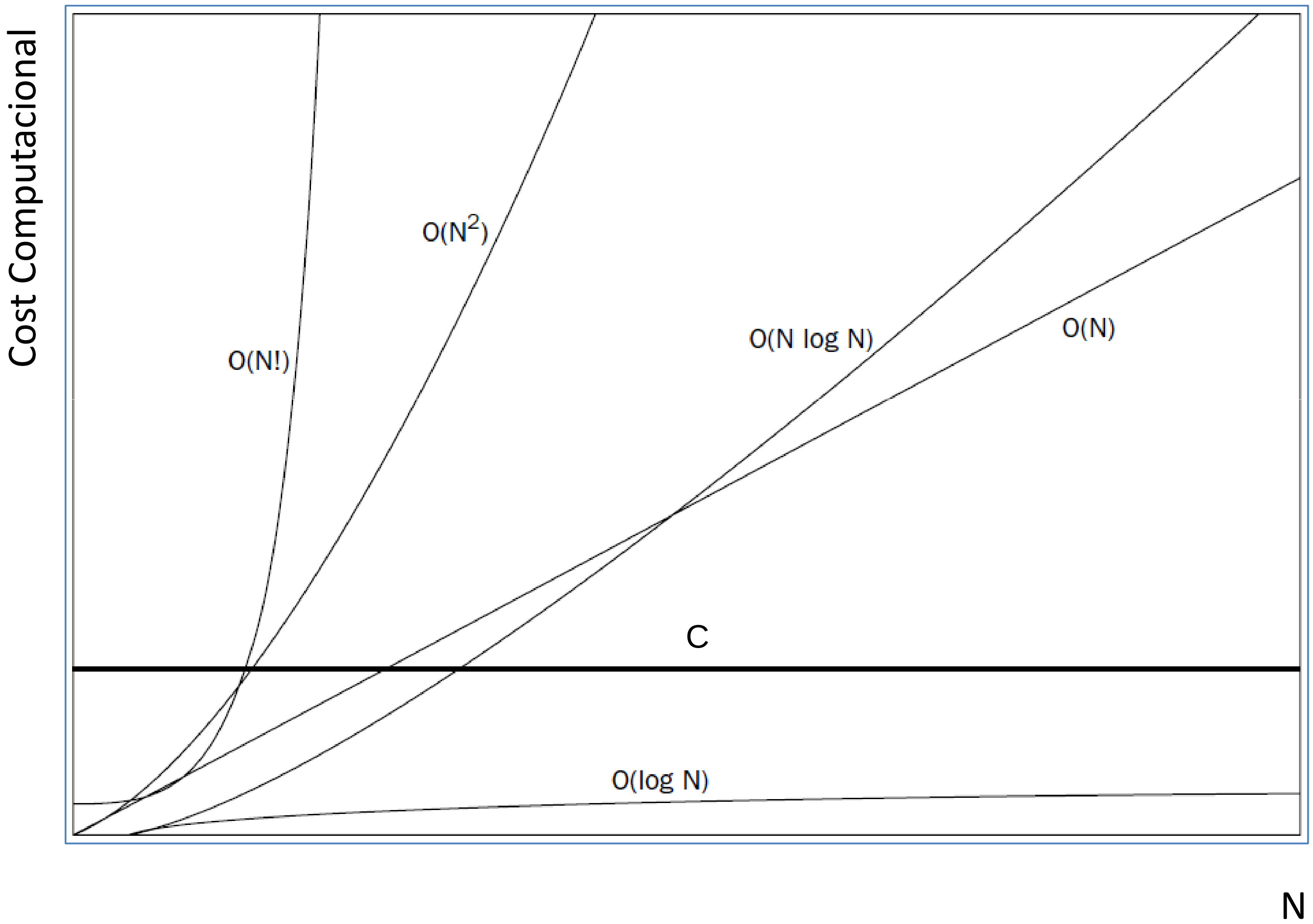
$$f=\Theta(g) \text{ si } f=O(g) \text{ i } f=\Omega(g)$$

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 nlogn poden arribar fins a n=1.000.000.000
- Els algorismes sublineals, logn, 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.

Polinòmics

- **Exponencials**, O(n) = c^n per c>1.
- **Factorials**, O(n) = n!

Aritmètica Bàsica

Aritmètica Bàsica

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.*

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

És evident que quan fem un canvi de base la mida del nombre només es veu afectada per un factor multiplicatiu, i per tant considerem que no canvia!

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

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

Aritmètica Bàsica

Aquesta propietat es compleix per totes les bases b>=2

Hi ha una propietat útil dels nombres decimals:

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

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

Carry:	1			1	1	1		
		1	1	0	1	0	1	(53)
		1	0	0	0	1	1	(35)
	1	0	1	1	0	0	0	(88)

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 $\mathrm{n}+1$ bits. **La seva complexitat és per tant, $\mathbf{O(n)}$.**

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 $2\mathrm{n}$ passos!

Aritmètica Bàsica

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

$$
\begin{array}{rrrrrrrrl}
 & & & & 1 & 1 & 0 & 1 & \text{(binary 13)} \\
 & & & \times & 1 & 0 & 1 & 1 & \text{(binary 11)} \\
\hline
 & & & & 1 & 1 & 0 & 1 & \text{(1101 times 1) (binary 13)} \\
 & & & 1 & 1 & 0 & 1 & & \text{(1101 times 1, shifted once) (binary 26)} \\
 & & 0 & 0 & 0 & 0 & & & \text{(1101 times 0, shifted twice) (binary 52)} \\
 + & 1 & 1 & 0 & 1 & & & & \text{(1101 times 1, shifted thrice) (binary 104)} \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{(binary 143)}
\end{array}
$$

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), i per cada fila (i en tenim n) **una suma de complexitat 2n**: **la complexitat total és $O(n^2)$**

Aritmètica Bàsica

Però 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
	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:

$$x \cdot y = \begin{cases} 2(x \cdot \lfloor y/2 \rfloor) \\ x + 2(x \cdot \lfloor y/2 \rfloor) \end{cases}$$

Si y és parell

Si y és senar

```
function multiply(x,y)
Input:  Two n-bit integers x and y, where y ≥ 0
Output:  Their product

if y = 0:  return 0
z = multiply(x, ⌊y/2⌋)
if y is even:
   return 2z
else:
   return x + 2z
```

Multiplicar/dividir per 2 i sumar té complexitat n

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

```
def mul(x,y):
    import math
    if y == 0:
            return 0
    z = mul(x,math.floor(y/2))
    if y%2 == 0:
            return 2*z
    else:
            return x+2*z
```

Aritmètica Bàsica

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

La seva versió recursiva és:

```
function divide(x, y)
Input:  Two n-bit integers x and y, where y ≥ 1
Output:  The quotient and remainder of x divided by y

if x = 0:  return (q, r) = (0, 0)
(q, r) = divide(⌊x/2⌋, y)
q = 2 · q,  r = 2 · r
if x is odd:  r = r + 1
if r ≥ y:  r = r − y,  q = q + 1
return (q, r)
```

La seva complexitat és $O(n^2)$