Tema 3: Algorísmes Numèrics

Resum del tema 3

- Sistema numeració
    - base
    - nombre de dígits
- Seqüència de Fibonacci
- Notació Gran O
- Aritmètica bàsica
    - suma
    - multiplicació
        - versió d'Al Khwarizmi
    - divisió
- Aritmètica Modular
    - suma
    - multiplicació, versió d'Al Khwarizmi
    - divisió
    - potència
- Algorisme d'Euclides
- Test de primeritat
    - Teorema petit de Fermat
    - Teorema de Lagrange

Una mica d'història

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

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

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.

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).

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.

Bases i representació numèrica

Quantes “unitats” hi ha a 642? Depèn de la base en que està escrit! La **base d’un nombre** determina el nombre de dígits diferents i el valor de les posicions dels dígits.

642 és 600 + 40 + 2 en BASE 10.

La fòrmula que ens permet entendre una base és:

$$d_n \times R^{n-1} + \cdots + d_2 \times R + d_1$$

on `R` és la base del nombre i `d_i` és el dígit a la posició i-èssima del nombre.

$$642 = 6_3 \times 10^2 + 4_2 \times 10 + 2_1$$

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`

HEXADECIMAL és base 16 i té 16 dígits: `0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F`

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

Una mica d'història

Fibonacci és avui conegut sobretot per la seva seqüència:

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

La seqüència es pot definir 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 computacionalment aquesta definició.

Una mica d'història

La seqüència creix molt ràpid i es pot demostrar que el terme `n`-èssim de la seqüència té aproximadament aquest valor:

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

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

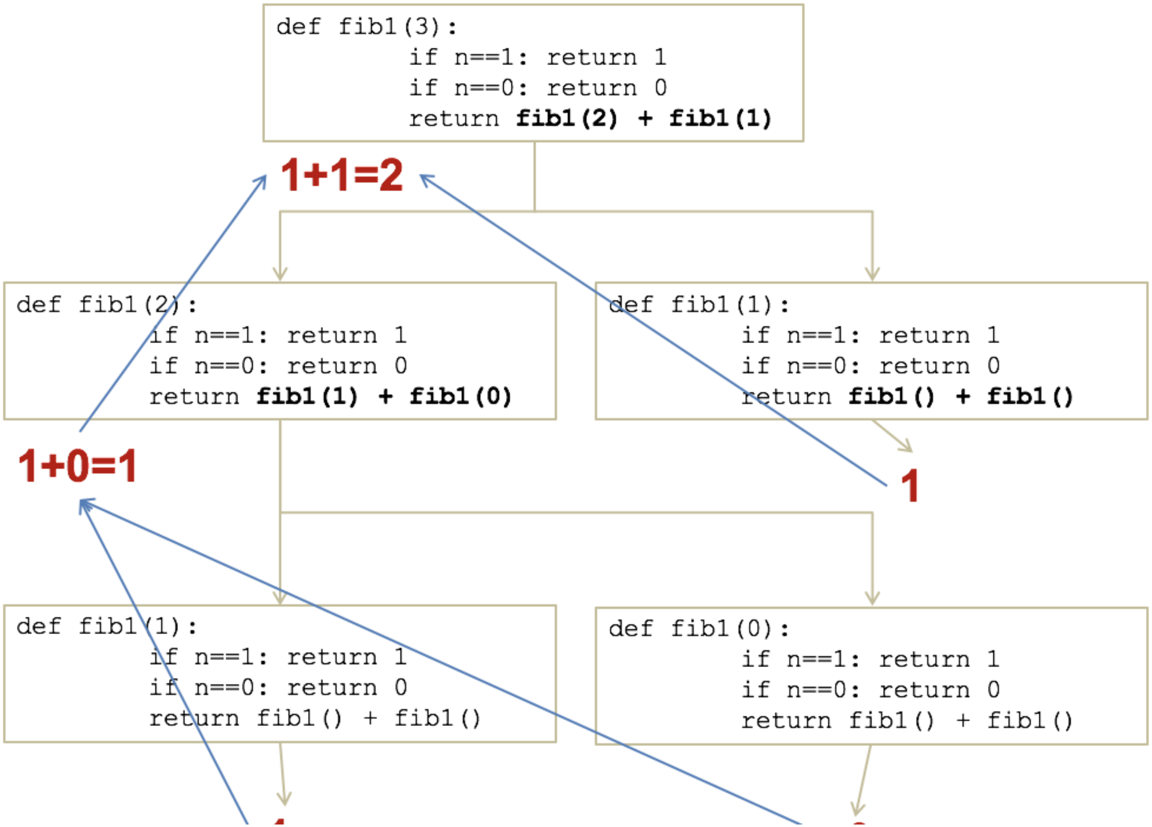
Una primera possibilitat és aquesta (*algorisme recursiu*)*:

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

fib1(10)
> 55
```

*Un algorisme recursiu és un algorisme que es crida a si mateix.

Algorisme recursiu de Fibonacci



Algorisme recursiu de Fibonacci

Com per a qualsevol algorisme, ens podem fer tres preguntes (**les tres preguntes bàsiques de l'algorísmica**):

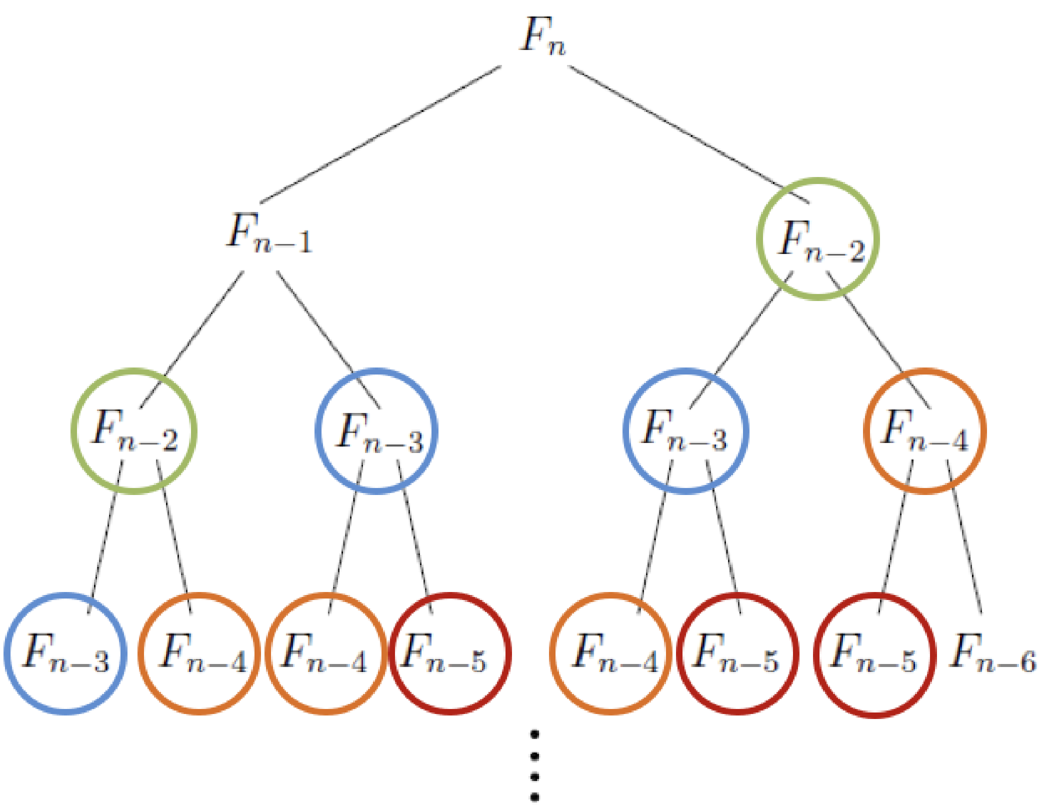
- És correcte?
- Quant trigarà, en funció de `n`?
- Hi ha alguna manera millor de fer-ho?

I les respostes són:

- En aquest cas és evident que si, atès que segueix exactament la definició!
- Es pot demostrar que el nombre de passos computacionals que fa és de l'ordre de `F_n`. Per calcular el terme 200 hauria de fer de l'ordre 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!
- Sí.

Per què és tant lent?

Algorisme recursiu de Fibonacci



Algorisme de Fibonacci

Anem a fer-ne una versió basada en llistes:

```
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]
```

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

Direm que `fib2(n)` és lineal (o polinòmic) respecte `n`. Ara podem calcular fins i tot `fib(100.000.000)`.

Però encara ho podem fer millor!

Algorisme de Fibonacci

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

fib3(10)
> 55
```

Com hem de comptar els passos computacionals?

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

```python
import math
a = 5
b = 4
for i in range(3):
    a += math.sqrt(a+b)
```

Però si manipulem nombres molt grans (que ocupen més de 64 bits), aquestes operacions no són tan barates!

```python
import math
a = 1234585127527575235234982374598245
b = 8112387512759287512875851285789127
for i in range(327864287686868676876876876887986):
    a += math.sqrt(a+b)
```

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).

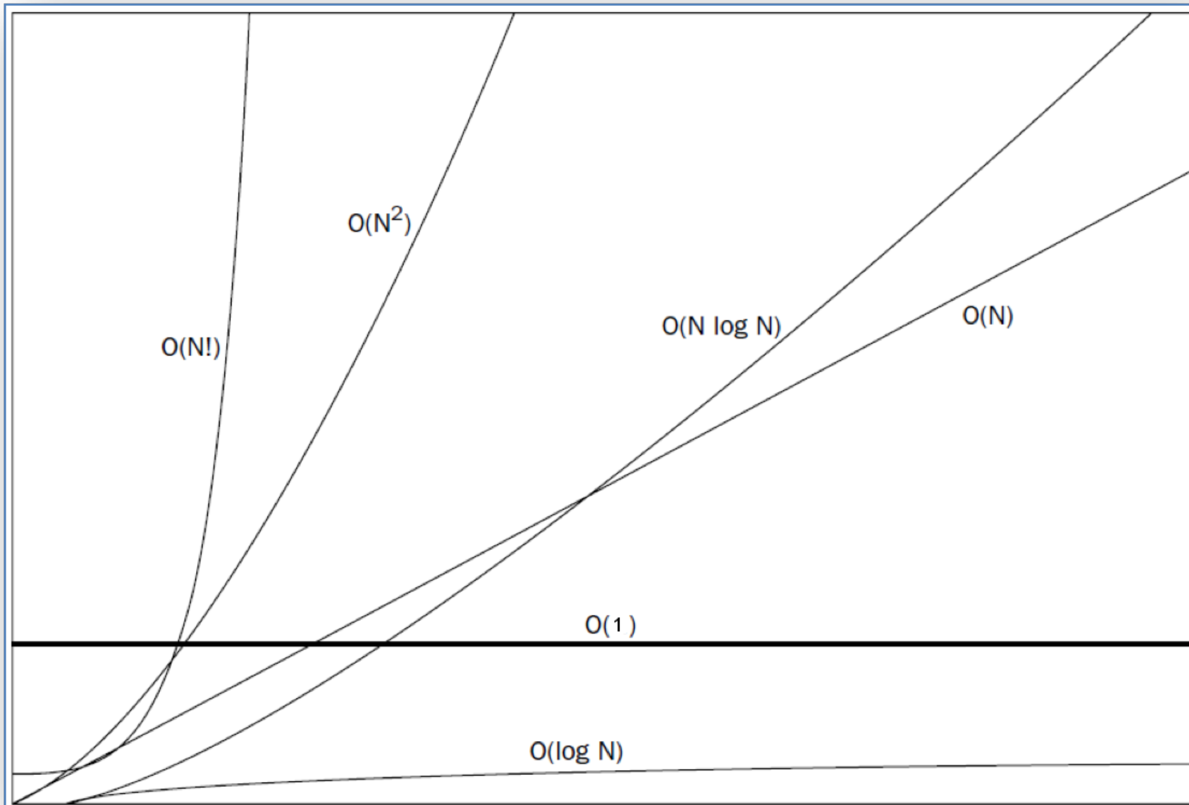
La regla principal és **comptar el nombre de passos computacionals aproximats en funció de la mida de l'entrada**.

Fem la següent aproximació: enlloc de dir que té una complexitat de `5 n^3 + 4 n + 3` direm que té una complexitat de `O(n^3)`

En general utilitzarem aquestes convencions:

- Ometrem les constants multiplicatives: `14n^2 és n^2`.
- `n^a` domina sobre `n^b` si `a>b`: `n^2` domina sobre `n`.
- Qualsevol exponencial domina sobre un polinomi: `3^n` domina sobre `n^5` (i també sobre `2^n`).
- Qualsevol polinomi domina sobre un logaritme: `n` domina sobre `log(n)^3` i `n^2` domina sobre `nlog(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 `nlog(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) = nlog(n)`.
- Quadràtics, `O(n) = n^2`.
- Cúbics, `O(n) = n^3`.
- Exponencials, `O(n) = c^n` per `c`>1.
- Factorials, `O(n) = n!`

Aritmètica Bàsica: Preliminar

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 `log_b (N+1)` dígits per escriure `N` en base `b` (això surt d'aillar `k` a l'equació `b^k -1 = N`).

Veiem un exemple:

> `k=5, b=2`
>
> Si tenim 5 dígits en base 2, podem representar fins a `2^5 -1 = 32 - 1 = 31`, efectivament `1111 = 16 + 8 + 4 + 2 + 1 = 31`
>
> Per altra banda, necessitarem `log_2 (31+1)` dígits per escriure `31` en base `b =>` `5` dígits

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!

Aritmètica Bàsica: Suma

Hi ha una propietat, que ens serà molt útil, dels nombres decimals:

- La suma de tres nombres d'un sol dígit qualsevol té com a màxim dos dígits. Aquesta regla és compleix per totes les bases `b >= 2`.

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)

Però, **quina complexitat té aquest algorisme**?

- Aquesta pregunta la farem sempre en relació a la mida (nombre de bits) dels elements de l'entrada.
- Per un nombre petit de bits (64), l'ordinador ho pot fer en un sol pas, però això no és veritat per a nombres molt grans.

Aritmètica Bàsica: Suma

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

Es pot fer millor?

Aritmètica Bàsica: Suma

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

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: Multiplicació

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

$$
\begin{array}{cccccccc l}
 & & & & 1 & 1 & 0 & 1 & \\
 & & & \times & 1 & 0 & 1 & 1 & \\
\hline
 & & & & 1 & 1 & 0 & 1 & \text{(1101 times 1)} \\
 & & & 1 & 1 & 0 & 1 & & \text{(1101 times 1, shifted once)} \\
 & & 0 & 0 & 0 & 0 & & & \text{(1101 times 0, shifted twice)} \\
+ & 1 & 1 & 0 & 1 & & & & \text{(1101 times 1, shifted thrice)} \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{(binary 143)}
\end{array}
$$

Tenim `n` multiplicacions de complexitat `n` (un bit per `n` bits) + aproximadament `2n` sumes de complexitat `2n`, que és un total de `(n^2 + 4n^2) = 5n^2` i per tant la **complexitat total de la multiplicació és `O(n^2)`**.

Aritmètica Bàsica: Multiplicació

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 aquesta operació "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.

Exemple: 11 x 13:

11	13
5	26
2	52
1	104
	143

Aritmètica Bàsica: Multiplicació

L'algorisme d'Al Khwarizmi es pot escriure així:

```
def mult(x,y):
    import math
    if y==0 or x==0:                    # en aquest cas arribem a 0
        return 0
    z = mult(math.floor(x/2),y)         # fem les crides reduint x
    if x%2 == 0:
        return 2*z                      # en el retorn és quan doblem y
    else:
        return y+2*z                    # només si és senar el sumem

print(mult(11,13))
```

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

*Si cada vegada que cridem la funció recursivament anem dividint per `2` el paràmetre `x` al cap de `n` crides el paràmetre ja valdrà `0`. Per exemple, si `x=16`, que necessita 5 bits (`n`= 5) per representar-se, llavors arribem a `0` en `5` crides: `16, 8, 4, 2, 0`.

Aritmètica Bàsica: Divisió

La divisió `x/y` consisteix en trobar un quocient `q` i una resta `r` de manera que:

$$x = y \times q + r$$

amb `r < y`. La seva versió recursiva és:

```
def div(x,y):
    import math
    if x<=0:
        return 0,0
    if y==1:
        return x,0
    q,r = div(math.floor(x/2),y)
    q = 2*q                  #desfem la divisió per 2
    r = 2*r                  #desfem la divisió per 2
    if x%2 != 0:
        r += 1               #recuperem el que hem perdut amb el floor
    if r >= y:
        r = r-y
        q = q+1              #aquí és on anem augmentant el quocient
    return q,r
```

La seva complexitat és O(n^2)