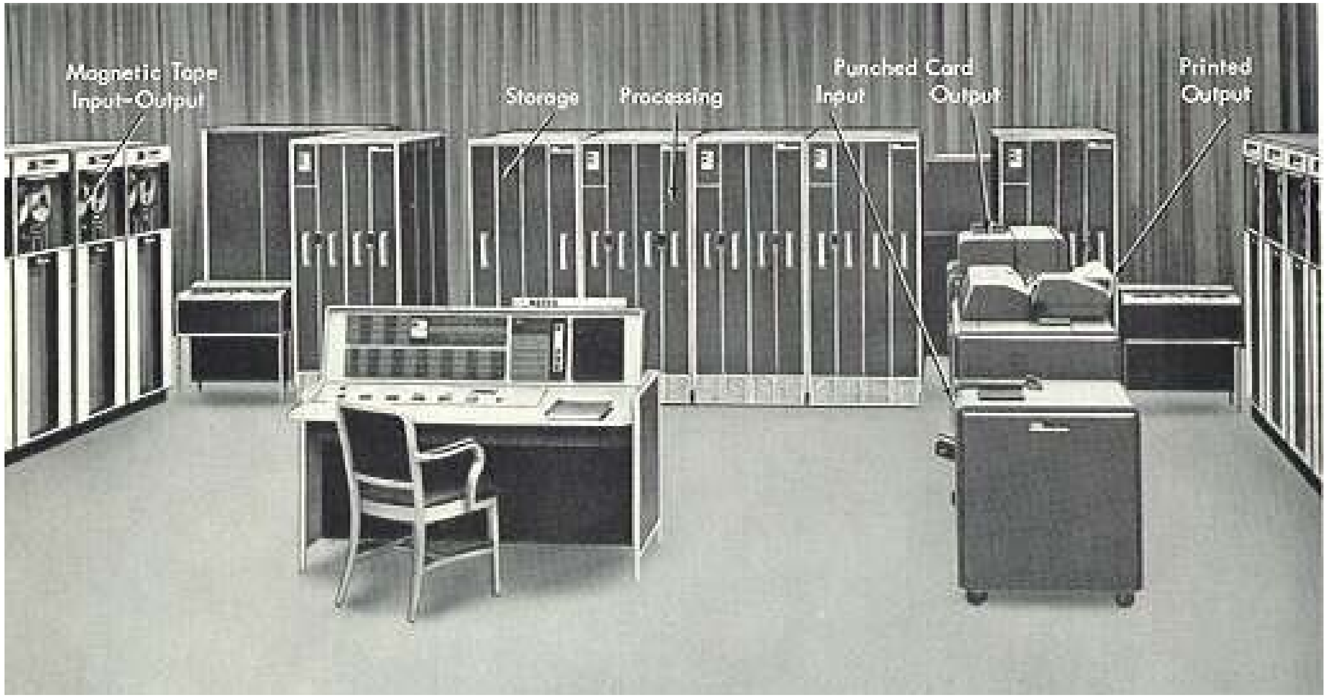
UNIVERSITAT DE BARCELONA

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.

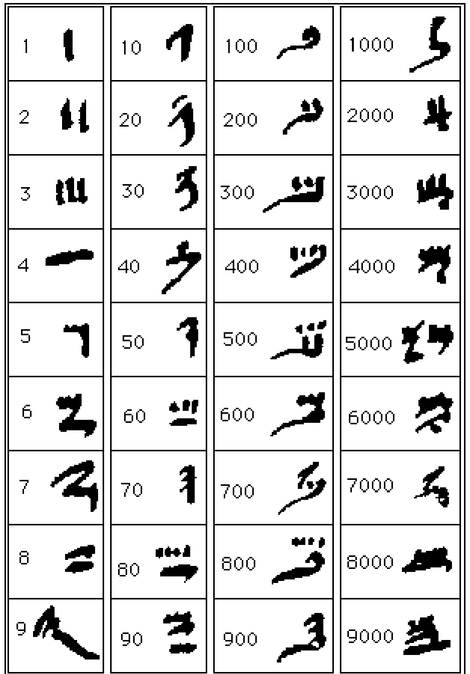
Un sistema de numeració ve definit doncs per:

• 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

Simple!

Quantes "unitats" hi ha a 642?

600 + 40 + 2

642 és 600 + 40 + 2 en **BASE 10**

La **base** d'un nombre determina el nombre de dígits diferents 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?

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

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?

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

Una mica d’història...

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

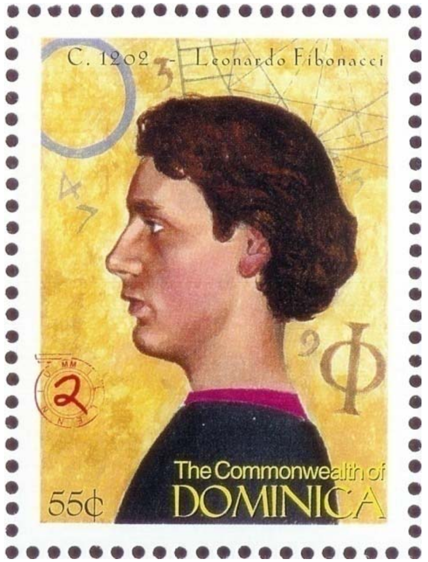
El medi de transmissió més important va ser un manual, escrit en àrab durant el segle IX a Bagdad, obra de **Al Khwarizmi**, en el que **especificava els procediments per sumar, multiplicar i dividir nombres escrits en base deu**.

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**!

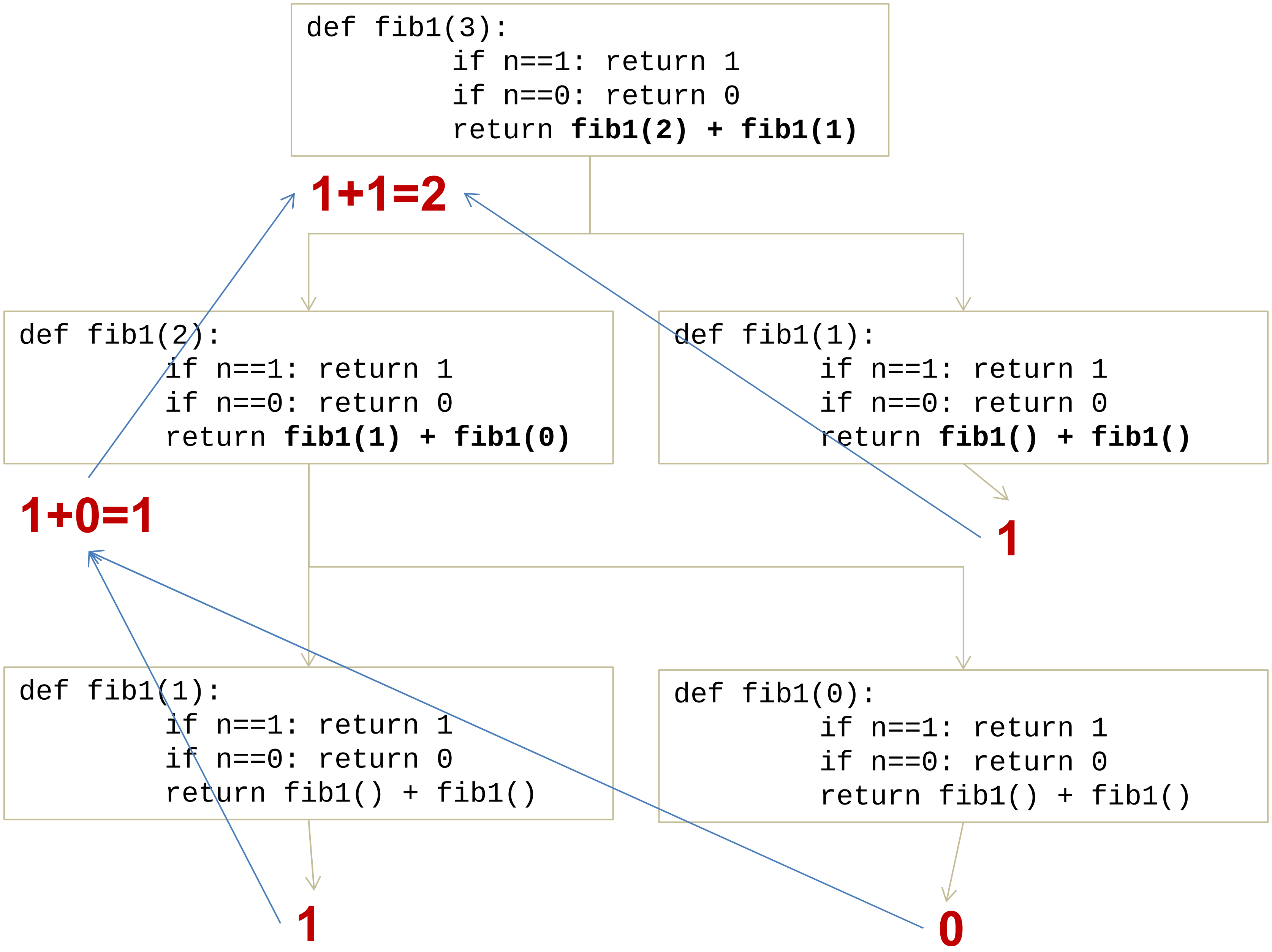
Una primera possibilitat és aquesta (**algorisme recursiu**):

```
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.

```python
>>> 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 $\mathbf{F_{200}}$, l’algorisme executa $T(200) >= F_{200} >= \mathbf{2^{138}}$ **passos**.

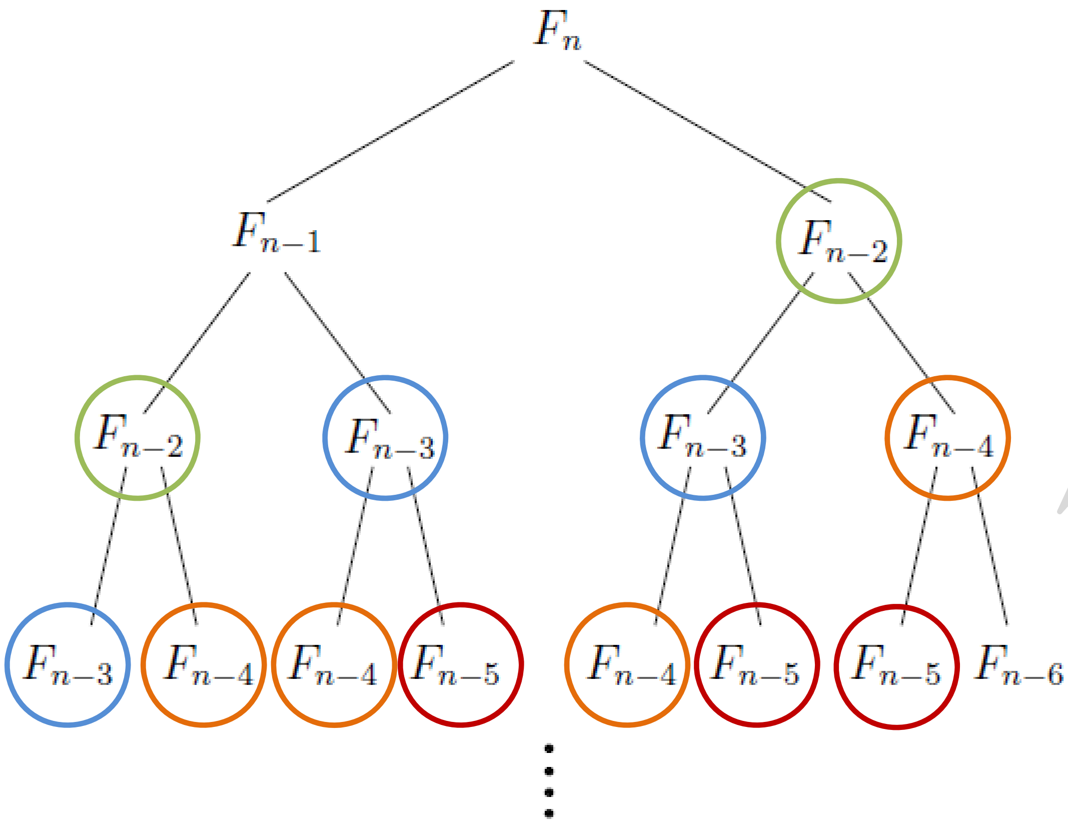
A l’ordinador més ràpid del món, que pot executar al voltant de 40.000.000.000.000 passos per segon, necessitaríem més temps que el necessari pel col·lapse del Sol!

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**

```
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**.

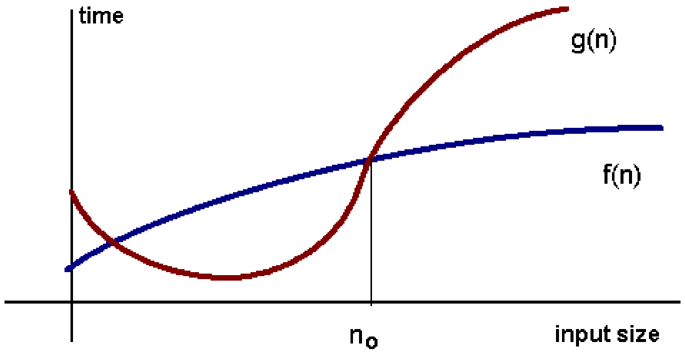
Fem la següent aproximació: enlloc de dir que pren $5n^3+4n+3$ direm que pren $O(n^3)$

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 O

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

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

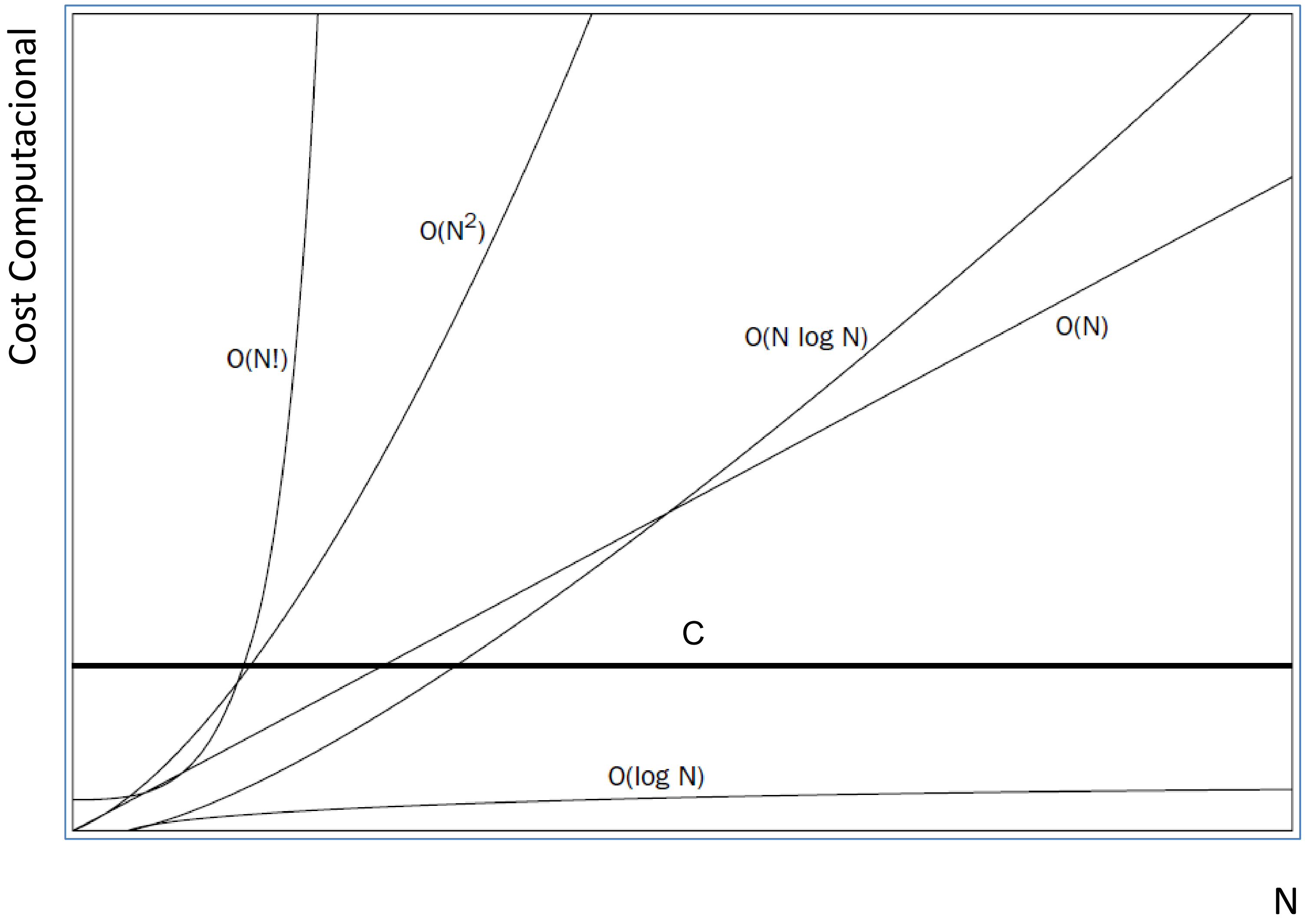
$$f = \Theta(g) \text{ si } f = O(g) \text{ i } f = \Omega(g)$$

La notació Gran $\mathbf{O}$

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*

Per tant, 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!

$$
\begin{array}{lccccccccl}
\text{Carry:} & 1 & & & 1 & 1 & 1 & & \\
 & & 1 & 1 & 0 & 1 & 0 & 1 & (53) \\
 & & 1 & 0 & 0 & 0 & 1 & 1 & (35) \\
\cline{2-8}
 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & (88)
\end{array}
$$

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)
			$\times$	1	0	1	1	(binary 11)
				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)
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
	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:

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

```
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 x/y consisteix en trobar un quocient q i una resta r de manera que x=y x q + r i r<y.

La seva versió recursiva és:

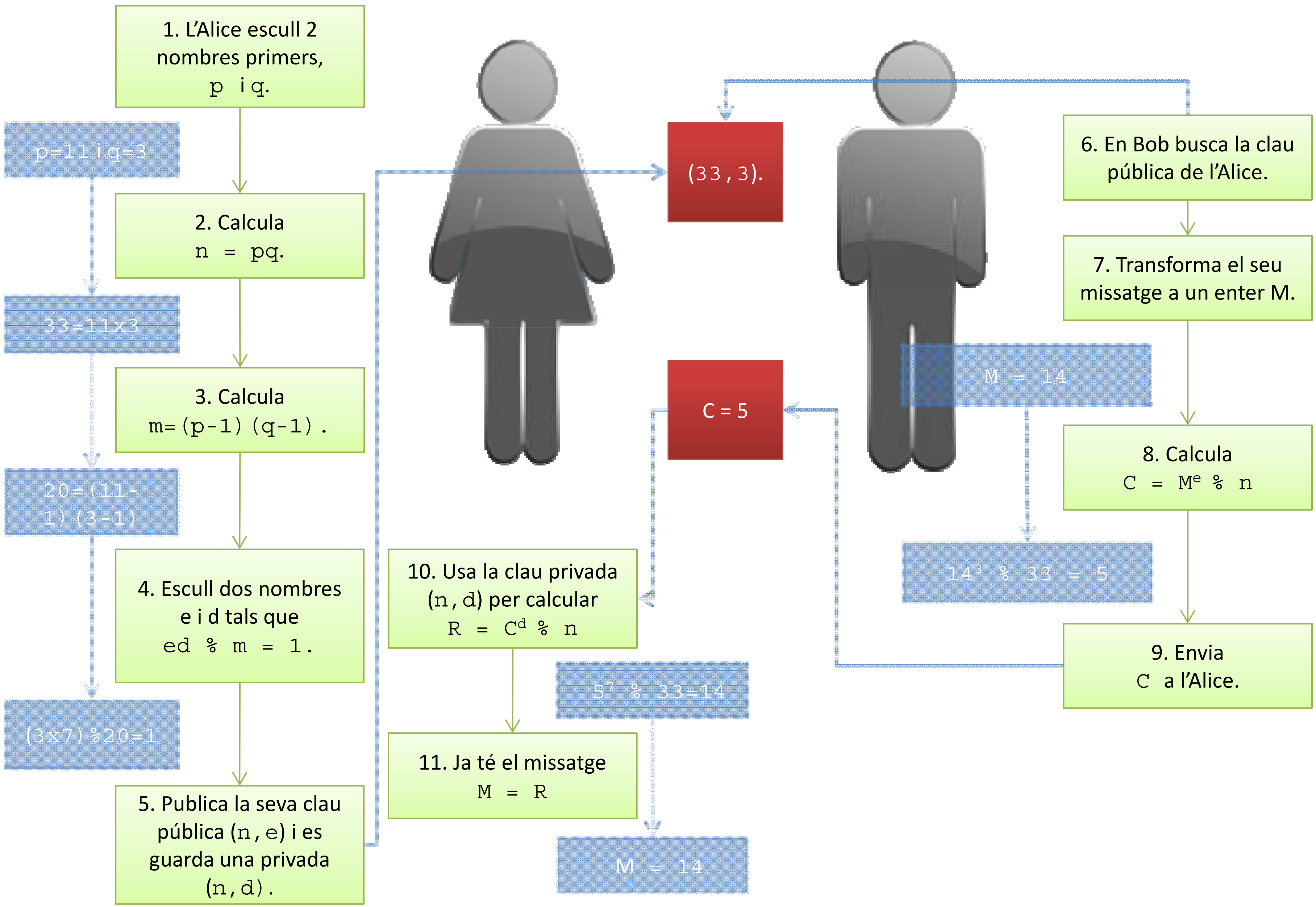
```
function divide(x, y)
Input:  Two n-bit integers x and y, where y ≥ 1
Output:  The quotient and remainder of x divided by y

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)
```

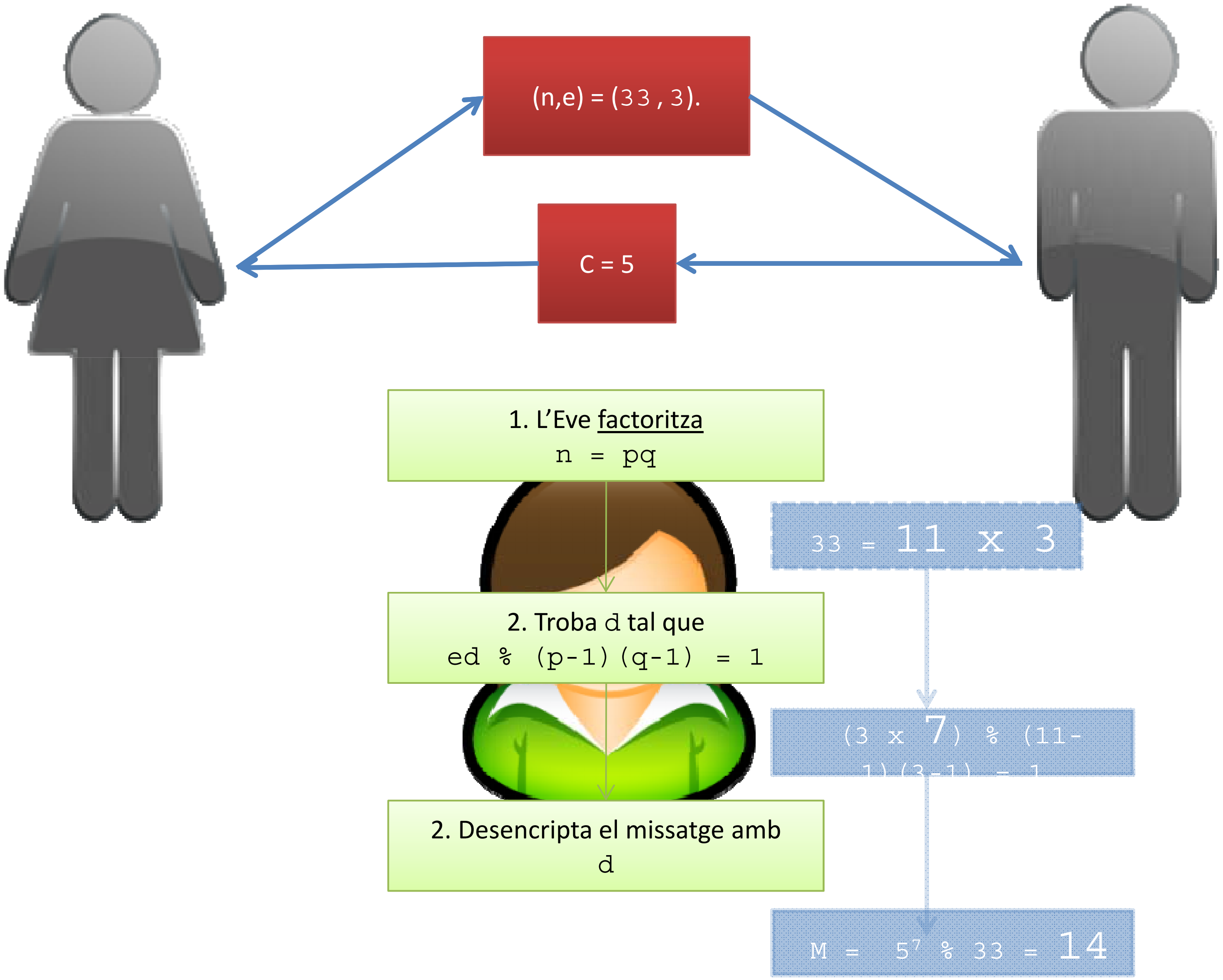
La seva complexitat és $O(n^2)$

Aritmètica Modular

(o l'Alice envia un missatge secret M a en Bob sense que l'Eve ho pugui llegir)


1. L'Alice escull 2 nombres primers, p i q.
p=11 i q=3
2. Calcula n = pq.
33=11x3
3. Calcula m=(p-1)(q-1).
20=(11-1)(3-1)
4. Escull dos nombres e i d tals que ed % m = 1.
(3x7)%20=1
5. Publica la seva clau pública (n,e) i es guarda una privada (n,d).
(33,3).
6. En Bob busca la clau pública de l'Alice.
7. Transforma el seu missatge a un enter M.
M = 14
8. Calcula C = M^e % n
14^3 % 33 = 5
9. Envia C a l'Alice.
C = 5
10. Usa la clau privada (n,d) per calcular R = C^d % n
5^7 % 33=14
11. Ja té el missatge M = R
M = 14

Si l'Eve vol saber quin és el missatge...



Aquest esquema té sentit si:

- Factoritzar n = pq és impossible.
- Trobar `(p,q)` "grans" es basa en un mètode eficient.
- Calcular $x^y \% n$ es es basa en un mètode eficient.
- Calcular `ed%(p-1)(q-1)` es basa en un mètode eficient.

Aritmètica Modular

En certs aspectes de la informàtica (per exemple, la criptografia) és important una variació de l'aritmètica sobre els nombres enters: **l'aritmètica modular**.

Serveix per operar amb rangs restringits d'enters.

Definim x mòdul N com la resta de dividir x per N, és a dir, si $x=qN+r$ amb $0<=r<N$, llavors el x mòdul N és r.

La complexitat és $O(n^2)$

Això permet definir una equivalència (**congruència**) entre nombres (inclosos els negatius!):

$$x \equiv y \pmod N \text{ si i només si } N \text{ divideix } (x\text{-}y)$$

Com que 10 divideix (133-3), 133 és congruent amb 3 mòdul 10

Aritmètica Modular

Això afecta a les operacions aritmètiques:

Substitution rule *If* $x \equiv x' \pmod{N}$ *and* $y \equiv y' \pmod{N}$, *then:*

$$x + y \equiv x' + y' \pmod{N} \quad \textit{and} \quad xy \equiv x'y' \pmod{N}.$$

Però les seves propietats es conserven:

$$x + (y + z) \equiv (x + y) + z \pmod{N} \qquad \text{Associativity}$$
$$xy \equiv yx \pmod{N} \qquad \text{Commutativity}$$
$$x(y + z) \equiv xy + yz \pmod{N} \qquad \text{Distributivity}$$

Si ajuntem les dues coses, tenim que és legal reduir els resultats intermedis al seu mòdul N en qualsevol moment.

$$2^{345} \equiv (2^5)^{69} \equiv 32^{69} \equiv 1^{69} \equiv 1 \pmod{31}.$$

Aritmètica Modular

La suma i la multiplicació no són gaire complexes.

Suma: Si dos nombres estan el rang $0..(N-1)$ la seva **suma** ho està en el $0..2(N-1)$ (que només és un bit més). Si el resultat passa de $N-1$ el que hem de fer és simplement **restar** del resultat N. És evident que **la complexitat és lineal $O(n)$**, on n = log N, la mida de N.

(3+7) mod 9 = 10 – 9 = 1

Recordem que necessitem log_b *N* dígits per escriure *N* en base *b*.

Multiplicació: De forma semblant, fem la multiplicació normal i transformem al rang $0..(N-1)$ si és que ens hem passat. El **producte** pot ser fins $(N-1)^2$ però això es pot representar amb $2n$ bits. Per transformar el resultat hem de **dividir** per N (amb complexitat $O(n^2)$). Per tant, **la complexitat és $O(n^2)$**

Aritmètica Modular

Divisió: Aquesta operació no és tant trivial (no està definida per tots els nombres) i té **una complexitat $O(n^3)$.** (*veure més endavant*)

Exponenciació: Ara imaginem que volem calcular expressions com aquesta amb nombres molt grans (centenars de bits):

$$x^y \bmod N$$

Aquest resultat entremig pot necessitar molts bits per ser representat!

El resultat necessita *n=log N* bits.

Si els operadors tenen 20 bits, aquest valor pot necessitar 10 milions de bits!

$$(2^{19})^{(2^{19})} = 2^{(19)(524288)}$$

La complexitat és molt alta!

Aritmètica Modular

Una solució és fer totes les operacions intermèdies mòdul N!

O sigui, calcular $x^y \bmod N$ fent y multiplicacions successives per x mòdul N.

$$x \bmod N \to x^2 \bmod N \to x^3 \bmod N \to \cdots \to x^y \bmod N,$$

Tots aquests resultats són menors que N i per tant no necessiten tants bits. Per tant, les multiplicacions són de complexitat $O(n^2)$.

El problema és que si y té 500 bits, hem de fer $y - 1 \approx 2^{500}$ multiplicacions i **l'algorisme és exponencial sobre n**, la mida de y.

Aritmètica Modular

Però una petita modificació pot ser un gran canvi!

Observem que es pot calcular x elevat a y, **si y és una potència de 2**, elevant al quadrat, mòdul N, successivament:

$$x \bmod N \to x^2 \bmod N \to x^4 \bmod N \to x^8 \bmod N \to \cdots \to x^{2^{\lfloor \log y \rfloor}} \bmod N.$$

Cada potència pren un temps proporcional a $O(\log^2 N)$ i hi ha $\log y$ multiplicacions: **l'algorisme és polinòmic O(n²) respecte la mida de N i lineal O(m) respecte la mida de y**!

m= log y és la mida de y: $\log_2 8 = 3$

n=log N és la mida en bits de N i a cada potència el nombre que multipliquem és < N.

Aritmètica Modular

Per un **valor qualsevol** de y (que no sigui potència de 2) només hem multiplicar les potències de 2 que corresponen a la representació binaria de y:

$$x^{25} = x^{11001_2} = x^{10000_2} \cdot x^{1000_2} \cdot x^{1_2} = x^{16} \cdot x^{8} \cdot x^{1}.$$

Aritmètica Modular

Aquesta operació es pot expressar **recursivament** fent aquestes operacions mòdul N:

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

$$x^{25} \bmod N = x \cdot (x^{12})^2 \bmod N$$

$$(x^6)^2 \bmod N$$

$$(x^3)^2 \bmod N$$

$$x \cdot (x)^2 \bmod N$$

$$3^{25} \bmod 3 = ((((3 \cdot 3^2 \bmod 3)^2 \bmod 3)^2 \bmod 3)^2 \cdot 3) \bmod 3 = 0$$

Aritmètica Modular

L'algorisme queda:

```
function modexp(x, y, N)
Input:  Two n-bit integers x and N, an integer exponent y
Output:  x^y mod N

if y = 0:  return 1
z = modexp(x, ⌊y/2⌋, N)
if y is even:
   return z^2 mod N
else:
   return x · z^2 mod N
```

La complexitat és $O(n^3)$
n crides recursives en les que fa una multiplicació mòdul N.

L'algorisme d'Euclides per trobar el mcd.

La forma més obvia de buscar-ho és **trobar els factors dels dos nombres i multiplicar llavors els seus factors comuns**.

Exemple pel *mcd* de 1035 i 759:

$1035=3^2*5*23$ i $759 = 3*11*23$, per tant $mcd=3*23=69$

El problema és que **no es coneix** cap algorisme **eficient** per **factoritzar els nombres**!

Fa més de 2000 anys que Euclides va enunciar un **algorisme** per trobar el **màxim comú divisor** de dos nombres *a* i *b*.

L'algorisme d'Euclides per trobar el mcd.

Euclides va utilitzar aquesta **regla simple**: Si x i y són enters positius amb x>=y, llavors mcd(x,y)=mcd(x mod y,y).

Proof. It is enough to show the slightly simpler rule $\gcd(x,y) = \gcd(x-y,y)$ from which the one stated can be derived by repeatedly subtracting y from x.

Here it goes. Any integer that divides both x and y must also divide $x-y$, so $\gcd(x,y) \leq \gcd(x-y,y)$. Likewise, any integer that divides both $x-y$ and y must also divide both x and y, so $\gcd(x,y) \geq \gcd(x-y,y)$. ∎

Això ens permet escriure aquest algorisme:

```
function Euclid(a, b)
Input:  Two integers a and b with a ≥ b ≥ 0
Output:  gcd(a, b)

if b = 0:  return a
return Euclid(b, a mod b)
```

Quan triga?....

```
def gcd(a,b):
# Algorisme d’euclides
        while a:
                a, b = b%a, a
        return b

>>> gcd(1071,462)
21
```

Quina complexitat té l’algorisme d’Euclides?

La primera cosa que hem de veure és com es van reduint els nombres a mesura que anem calculant.

Cal fixar-se que a cada iteració els arguments (a,b) es converteixen a (b, a mod b): canviem l’ordre i el més gran queda reduït al mòdul del petit.

És fàcil demostrar que si a>=b, llavors (a mod b)<a/2.

Proof. Witness that either $b \leq a/2$ or $b > a/2$. These two cases are shown in the following figure. If $b \leq a/2$, then we have $a \bmod b < b \leq a/2$; and if $b > a/2$, then $a \bmod b = a - b < a/2$.

L’algorisme d’Euclides

Això vol dir que **en dos iteracions successives** els dos arguments decreixen al menys a la meitat, és a dir, **perden un bit en la seva representació.**

Si inicialment eren enters de n bits, en 2n crides recursives arribarem al final de l’algorisme. Com que cada crida implica una divisió d’ordre quadràtic, (a mod b) , **el temps total serà *$O(n^3)$.***

Una extensió de l'algorisme d'Euclides

Suposem que algú ens diu que d és el mcd(a,b). **Com podem comprovar-ho?**

No és simple: no n'hi ha prou amb dir que d divideix a i b, perquè això només vol dir que és factor comú, no el més gran!

Però podem usar aquest lema:

Si d divideix a i b, i d=ax+by per alguns enters x i y, llavors necessariament d=mcd(a,b).

Proof. By the first two conditions, d is a common divisor of a and b and so it cannot exceed the greatest common divisor; that is, $d \leq \gcd(a,b)$. On the other hand, since $\gcd(a,b)$ is a common divisor of a and b, it must also divide $ax + by = d$, which implies $\gcd(a,b) \leq d$. Putting these together, $d = \gcd(a,b)$. ∎

Una extensió de l'algorisme d'Euclides

Però per usar el lema, hem de trobar els nombres....

Doncs resulta que aquests nombres es podem trobar amb una petita extensió de l'algorisme d'Euclides:

```
function extended-Euclid(a, b)
Input:  Two positive integers a and b with a ≥ b ≥ 0
Output:  Integers x, y, d such that d = gcd(a, b) and ax + by = d

if b = 0:  return (1, 0, a)
(x', y', d) = Extended-Euclid(b, a mod b)
return (y', x' − ⌊a/b⌋y', d)
```

La complexitat és $O(n^3)$, no canvia respecte a l'algorisme original.

```python
def egcd(a, b):
# Algorisme extes d'Euclides
# retorna (g,x,y) tals que ax + by = g = gcd(a, b).

    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

>>> egcd(25,11)
(1, 4, -9)
>>> 1 == 4*25 + 11*-9
True
```

Divisió Modular: el problema.

A l'aritmètica real, cada nombre *a* diferent de zero té un invers 1/*a*, i **dividir per *a* és el mateix que multiplicar pel seu invers.**

A l'aritmètica modular podem fer una definició semblant: Direm que *x* és **l'invers multiplicatiu** de *a (mod N)* si

$$ax \equiv 1 \pmod{N}$$

Aquest invers, però, no sempre existeix: 2 no és invertible mòdul 6 perquè no hi ha cap *x* que faci

$$2x \equiv 1 \bmod 6$$

Divisió Modular: Quins nombres no poden tenir invers?

De forma més general, podem estar segurs que *mcd(a,N)* divideix *a·x mod N*, perquè aquesta quantitat es pot escriure com *ax+kn*. Per tant, **si *mcd(a,N)*>1** llavors

$$ax \not\equiv 1 \pmod{N}$$

independentment del valor x, i per tant *a* **no pot tenir un invers multiplicatiu** mòdul N!

Quan *mcd(a,N)=1* diem que *a* i *N* són **relativament primers**. **Només els nombres relativament primers tenen invers multiplicatiu mòdul N.**

Divisió Modular

Per nombres **relativament primers** l'algorisme modificat d'Euclides ens retorna dos enters, x i y tals que ax+Ny=1, lo que significa que ax és congruent mòdul N, i per tant **x és l'invers de a!**

Example. Continuing with our previous example, suppose we wish to compute $11^{-1} \bmod 25$. Using the extended Euclid algorithm, we find that $15 \cdot 25 - 34 \cdot 11 = 1$. Reducing both sides modulo 25, we have $-34 \cdot 11 \equiv 1 \bmod 25$. So $-34 \equiv 16 \bmod 25$ is the inverse of $11 \bmod 25$.

Teorema de la divisió modular

Per qualsevol *a mod N*, *a* té un invers multiplicatiu mòdul *N* si i només si és relativament primer per *N*. Quan aquest invers existeix, es pot trobar amb un temps $O(n^3)$ amb l'algorisme extès d'Euclides.

Amb això resolem el problema de la divisió modular: quan treballem mòdul N, podem dividir pels nombres relativament primers per N, i només per aquests. I per dividir, el que hem de fer és multiplicar per l'invers

Test de primalitat: És un nombre primer el vostre DNI?

Comprovar si un nombre més o menys gran és primer per la via de la factorització és una tasca a priori dura, perquè **hi ha molts factors per provar**. Però hi ha alguns fets que ens poden estalviar feina:

- No cal considerar com a factor cap nombre parell excepte el 2. De fet, podem obviar tots els factors que no són primers.
- Podem dir que un nombre és primer si no hem trobat cap candidat a factor menor que arrel de N, atès que N=K*L, i per tant és impossible que els dos nombres siguin més grans que arrel de N.

Fins aquí, bé, però no trobarem més maneres d'eliminar més candidats!

Això podria fer dir que provar la primalitat d'un nombre és un problema dur, però això no és veritat: només és dur si ho intentem pel camí de la factorització!

Test de primalitat

Una de les activitats bàsiques de la informàtica, la **criptografia**, es basa en el següent fet: **la factorització és dura, però la primalitat és fàcil.**

O el que és el mateix, **no podem factoritzar grans nombres, però podem mirar fàcilment si grans nombres són primers** (evidentment, sense buscar els factors!).

Per fer-ho, ens basarem en un teorema de 1640...

Test de primalitat

Teorema petit de Fermat

Si p és primer, llavors per a qualsevol enter a, 1<= a <p, es compleix que,

$$a^{p-1} \equiv 1 \pmod{p}$$

Això ens suggereix un test directe per comprovar si un nombre és primer:

```
function primality(N)
Input:  Positive integer N
Output:  yes/no

Pick a positive integer a < N at random
if a^(N-1) ≡ 1 (mod N):
   return yes
else:
   return no
```

Podríem repetir aquest test un munt de vegades i decidir

Test de primalitat

Els problema és que aquest teorema és **necessari però no suficient**: no diu què passa quan N no és primer!

D'entrada, es coneixen uns certs nombres compostos, anomenats nombres de Carmichael, que passen el test per tots els enters a relativament primers a N. Per tant, és perillós passar la funció tal i com l'hem escrit...

Test de primalitat

Per exemple, el nombre 561 és un nombre de Carmichael:

a^{560} **és congruent amb 1 mòdul 561**

per tots els nombres *a* relativament primers a 561, i 561=3x11x17.

I per tant no és congruent amb 1 per *a* igual a 3, 11, 17 i múltiples d'aquests nombres. Per tots els altres valors compleix el test de Fermat.

Test de primalitat

Suposem de moment que no existeixen els nombres de Carmichael. Què passa amb els altres nombres compostos?

Lema

Si $a^{N-1} \not\equiv 1 \pmod{N}$ per algun *a* relativament primer a N (*o sigui, els nombres compostos que no són de Carmichael*), llavors com a mínim en **la meitat dels casos** en que *a*<N el teorema petit de Fermat fallarà.

Test de primalitat

Una petita variació de l'algorisme que veurem, coneguda com l'algorisme de *Rabin & Miller* soluciona aquest problema.

Si ignorem els nombres de Carmichael, podem dir que:

- Si N és primer, llavors $a^{N-1} \equiv 1 \pmod{N}$ per tots *a*<N
- Si N no és primer, llavors $a^{N-1} \equiv 1 \pmod{N}$ fallarà per almenys la meitat dels valors *a*<N

I per tant el comportament de l'algorisme proposat és:

- El test retornarà `yes` en tots els casos si N és primer.
- El test retornarà `yes` per la meitat o menys dels casos en que N no és primer.

Test de primalitat

Si repetim l'algorisme *k* vegades per nombres *a* escollits aleatòriament, llavors:

$$\Pr\,(\text{"}yes\text{"}\ \text{quan}\ N\ \text{no és primer}) \leq \frac{1}{2^k}$$

Si k=100, la probabilitat d'error és menor que 2^{-100}

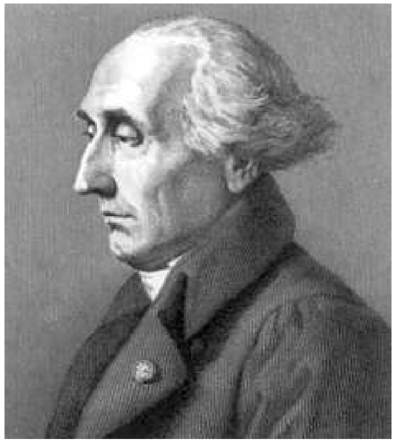
Amb un nombre moderat de tests podem determinar si un nombre és primer.

Primalitat i grans nombres

Com ho fem per trobar primers formats per uns quants centenars de bits?

Si n'hi ha pocs tenim un problema amb l'algorisme anterior, doncs l'haurem de repetir moltes vegades per poder trobar-ne!

El teorema dels nombres primers de Lagrange ens assegura que no tindrem problemes: la probabilitat de que un nombre de *n* bits sigui primer és aproximadament:



$$\frac{1}{\ln 2^n} \approx \frac{1.44}{n}$$

Primalitat i grans nombres

Com és de ràpid aquest mètode per trobar primers de n bits?

A cada iteració té una probabilitat 1/n d'aturar-se, per tant s'aturarà quan hagi fet O(n) iteracions.

Com l'apliquem?

N'hi ha prou amb provar la funció de primalitat de Fermat per a=2, 3 i 5! (De fet, la probabilitat de fallar és molt menor que ½!)

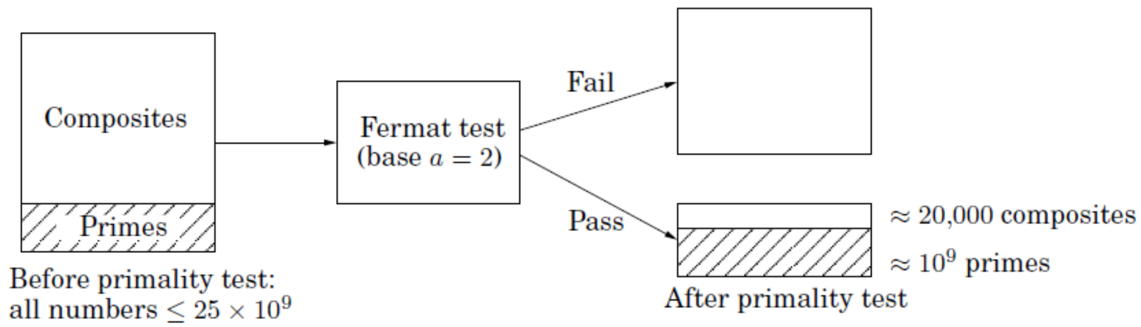
Quina és la probabilitat de que el nombre seleccionat sigui primer?

Suposem que volem passar el test per a=2 a tots els nombres < 25×10^9

Primalitat i grans nombres

Quina és la probabilitat de que el nombre seleccionat sigui primer?

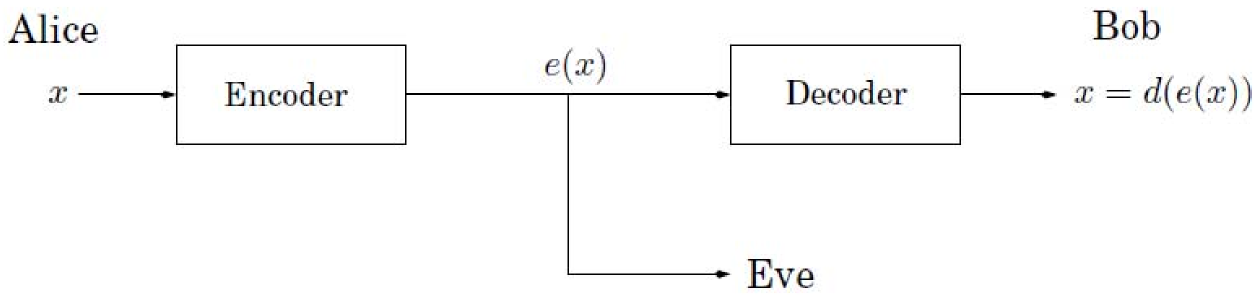
En aquest rang hi ha al voltant de 10^9 primers i 20.000 compostos que passen el test.



Per tant, la probabilitat d'error és $20.000/10^9 = 2 \times 10^{-5}$
Aquesta probabilitat va baixant a mesura que creix N.

Criptografia

El problema és que l'Alice pugui codificar amb una clau secreta x, generant e(x), i que en Bob sigui l'únic que pugui descodificar-ho:



Anem a veure com implementar un sistema segur amb clau pública!

Criptografia

$\boxed{\text{Bob chooses his public and secret keys.}}$

- He starts by picking two large (n-bit) random primes p and q.
- His public key is (N, e) where $N = pq$ and e is a $2n$-bit number relatively prime to $(p-1)(q-1)$. A common choice is $e = 3$ because it permits fast encoding.
- His secret key is d, the inverse of e modulo $(p-1)(q-1)$, computed using the extended Euclid algorithm.

$\boxed{\text{Alice wishes to send message } x \text{ to Bob.}}$

- She looks up his public key (N, e) and sends him $y = (x^e \bmod N)$, computed using an efficient modular exponentiation algorithm.
- He decodes the message by computing $y^d \bmod N$.

Criptografia

És molt segur?

És segur si donats N, e, i $y=x^e \bmod N$, és computacionalment intractable trobar x.

Com podria l'Eve trobar x?

Podria provar per tots els valors de x si $y=x^e \bmod N$ però això té una complexitat exponencial!

També podria factoritzar N per trobar p i q, i llavors determinar d invertint $e \bmod (p-1)(q-1)$, però això és també intractable!