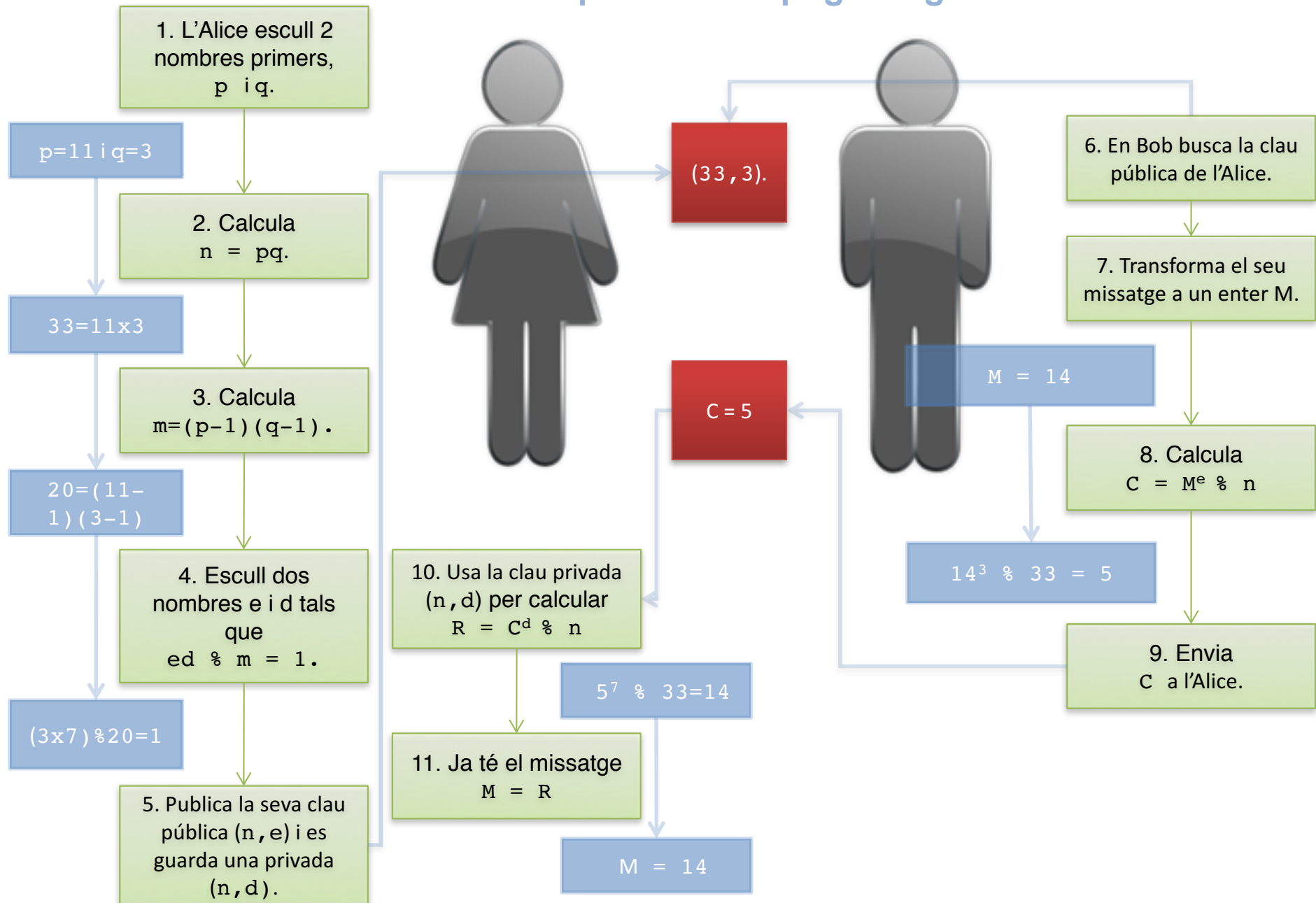


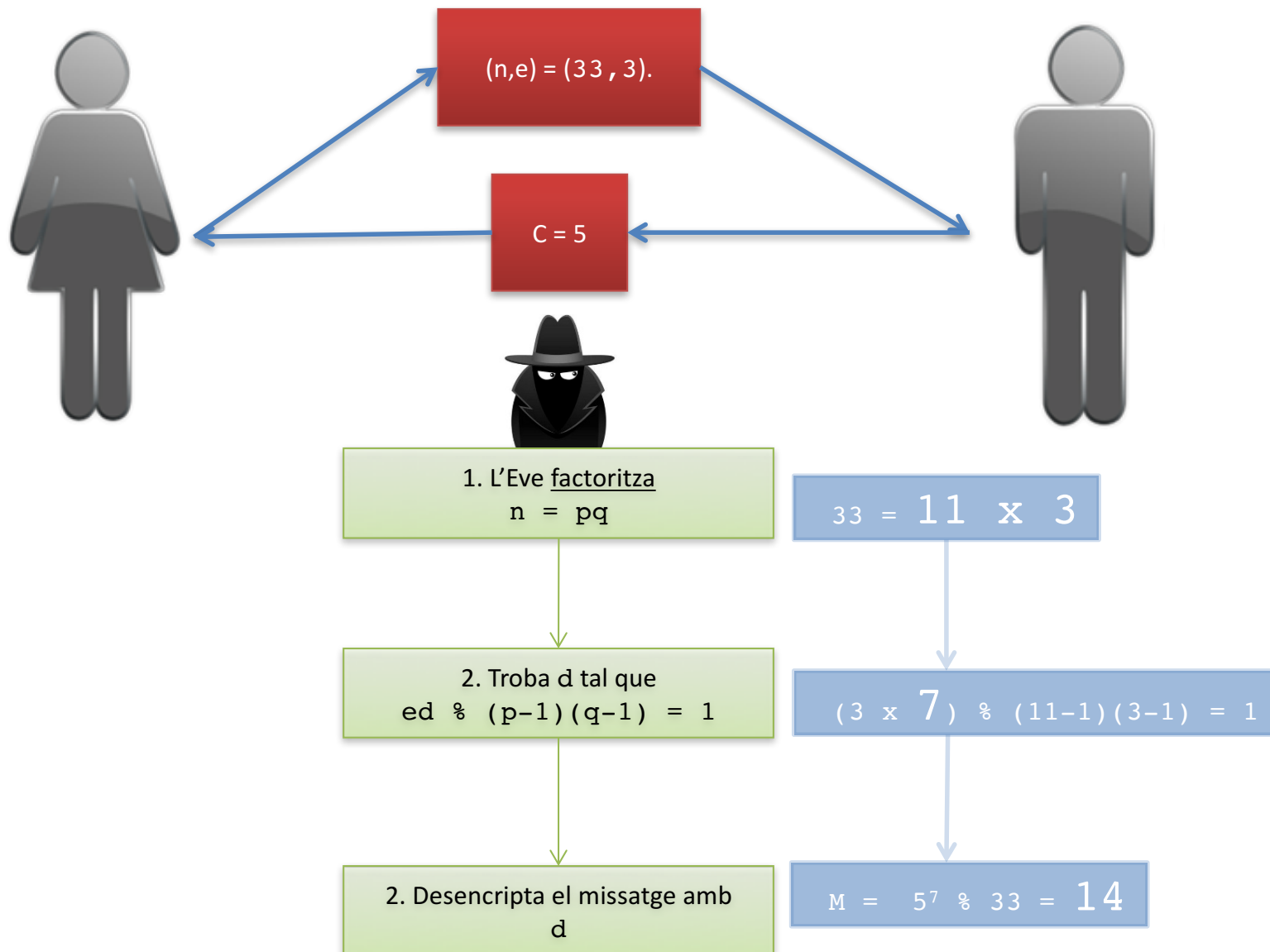
Aritmètica Modular

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

En Bob envia un missatge secret M a l'Alice sense que l'Eve ho pugui llegir



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 $ed \% (p-1)(q-1) = 1$ 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** , **$x \% N$** , com la resta de dividir x per N , és a dir, si $x = qN + r$ amb $0 \leq r < N$, llavors el x mòdul N és r .

La complexitat és $O(n^2)$

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

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

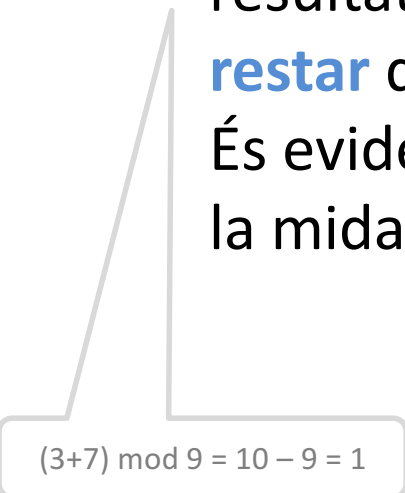
Com que 10 divideix $(133 - 3)$, 133 és congruent amb 3 mòdul 10

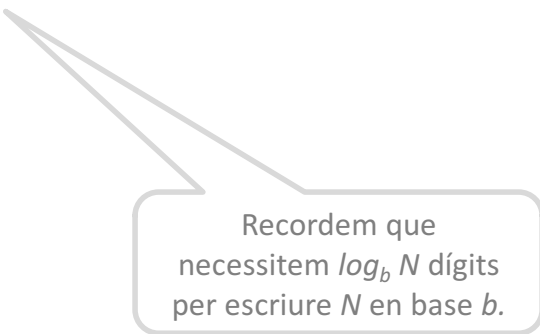
Aritmètica Modular

La suma i la multiplicació no són gaire complexes d'analitzar.

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 .


$$(3+7) \bmod 9 = 10 - 9 = 1$$



Recordem que
necessitem $\log_b N$ dígit
per escriure N en base b .

Aritmètica Modular

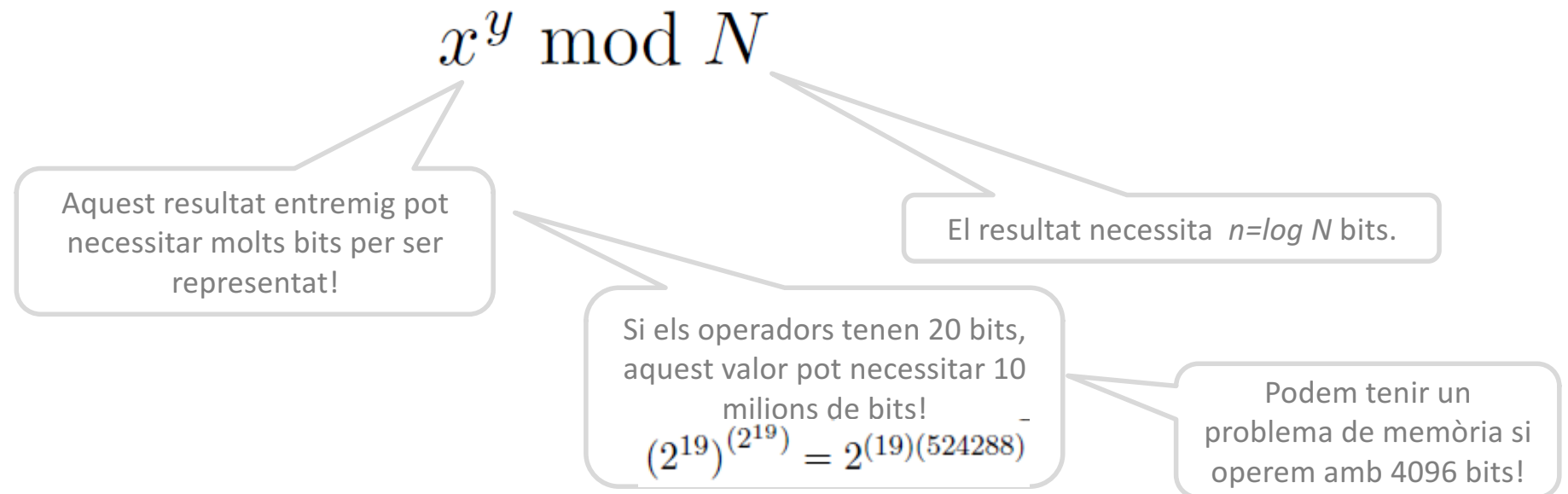
La suma i la multiplicació no són gaire complexes.

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 simple (no està definida per tots els nombres) i té **una complexitat $O(n^3)$** .

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

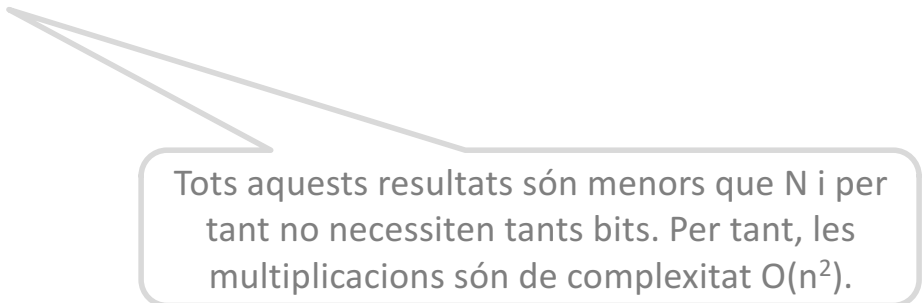


Aritmètica Modular

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 .

$$x \bmod N \rightarrow x^2 \bmod N \rightarrow x^3 \bmod N \rightarrow \dots \rightarrow x^y \bmod N,$$



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**, elevat al quadrat, mòdul N , successivament:

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

Cada potència pren un temps proporcional a $O(\log^2 N)$ i hi ha $\log y$ multiplicacions: **l'algorisme és polinòmic $O(n^2)$ 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 binària 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:

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

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

$$\xrightarrow{\quad} (x^6)^2 \bmod N$$

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

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

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

Aritmètica Modular

L'algorisme queda:

```
def modexp(x,y,N):  
    import math  
    if y==0:  
        return 1  
    z = modexp(x,math.floor(y/2),N)  
    if y%2 == 0:  
        return (z**2)%N  
    else:  
        return (x*z**2)%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 trobar el **màxim comú divisor** de dos nombres és **trobar els factors dels dos nombres i multiplicar llavors els seus factors comuns**.

Exemple pel *mcd* de 1035 i 759:

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

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** alternatiu per trobar el **màxim comú divisor** de dos nombres a i b .

L'algorisme d'Euclides per trobar el mcd.

```
def gcd(a,b):  
    """  
    Algorisme d'euclides  
    """  
    while a:  
        a, b = b%a, a  
    return b  
  
gcd(1071,462)
```

21

Quan triga?....

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 $(b, a \bmod b)$: canviem l'ordre i el més gran queda reduït al mòdul del petit.

Es pot demostrar que 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**.

L'algorisme d'Euclides

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, $(a \bmod b)$, **el temps total serà $O(n^3)$.**

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 ,
 $1 \leq a < p$, es compleix que,

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



Això ens suggereix un test directe per comprovar si un nombre és primer... Però cal anar en compte....

Test de primalitat

```
def fermat(num, test_count):  
    if num == 1:  
        return False  
    for x in range(test_count):  
        val = randint(1, num-1)  
        if pow(val, num-1, num) != 1:  
            return False  
    return True
```

fermat(41651,10)

True



41651 és primer

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... però són pocs i és poc probable que en trobem un de forma aleatòria.

Test de primalitat

Què passa amb els nombres compostos que no són nombres de Carmichael?

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:

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

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.

```

from random import randint

def fermat(num, test_count):
    if num == 1:
        return False
    for x in range(test_count):
        val = randint(1, num-1)
        if pow(val, num-1, num) != 1:
            return False
    return True

def generate_prime(n):
    found_prime = False
    while not found_prime:
        p = randint(2**(n-1), 2**n)
        if fermat(p, 20):
            return p

%timeit generate_prime(1024)
generate_prime(1024)

```

n: Nombre de bits del
nombre primer

1 loop, best of 3: 2.85 s per loop

```

1334480373633605102181044775059641619770996836891710427844700945608956736503877
5197574289256044886865710306124092220160479184590986686464012079658080272098609
8177654595652435589505079443451332091078802525585745668971281241665111901058028
070835983690342206608283359724533098000683954245986013813260117840291349

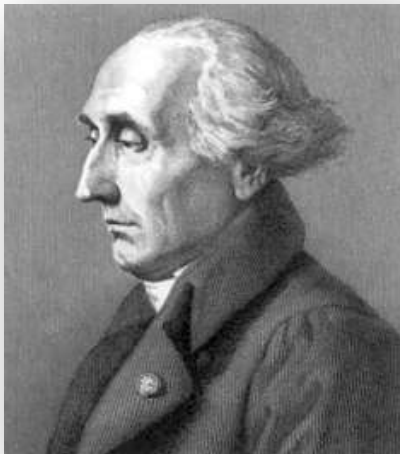
```

Primalitat i grans nombres

Com és que no triga en 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:



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

Pel cas $n=1000$, generarem al voltant de 1000 nombres aleatoris per trobar un primer.

Abans hem dit que aquest esquema té sentit si:

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

Com solucionem el darrer punt?

La solució del darrer punt és:

- 1) Definim $e=3$.
- 2) Llavors d és el que s'anomena invers de e mòdul $(p-1)(q-1)$ i aquest nombre es pot calcular amb una petita variació de l'algorisme d'Euclides!

Per tant, els algorismes de més alta complexitat en un procés criptogràfic tenen $O(N^3)$.

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