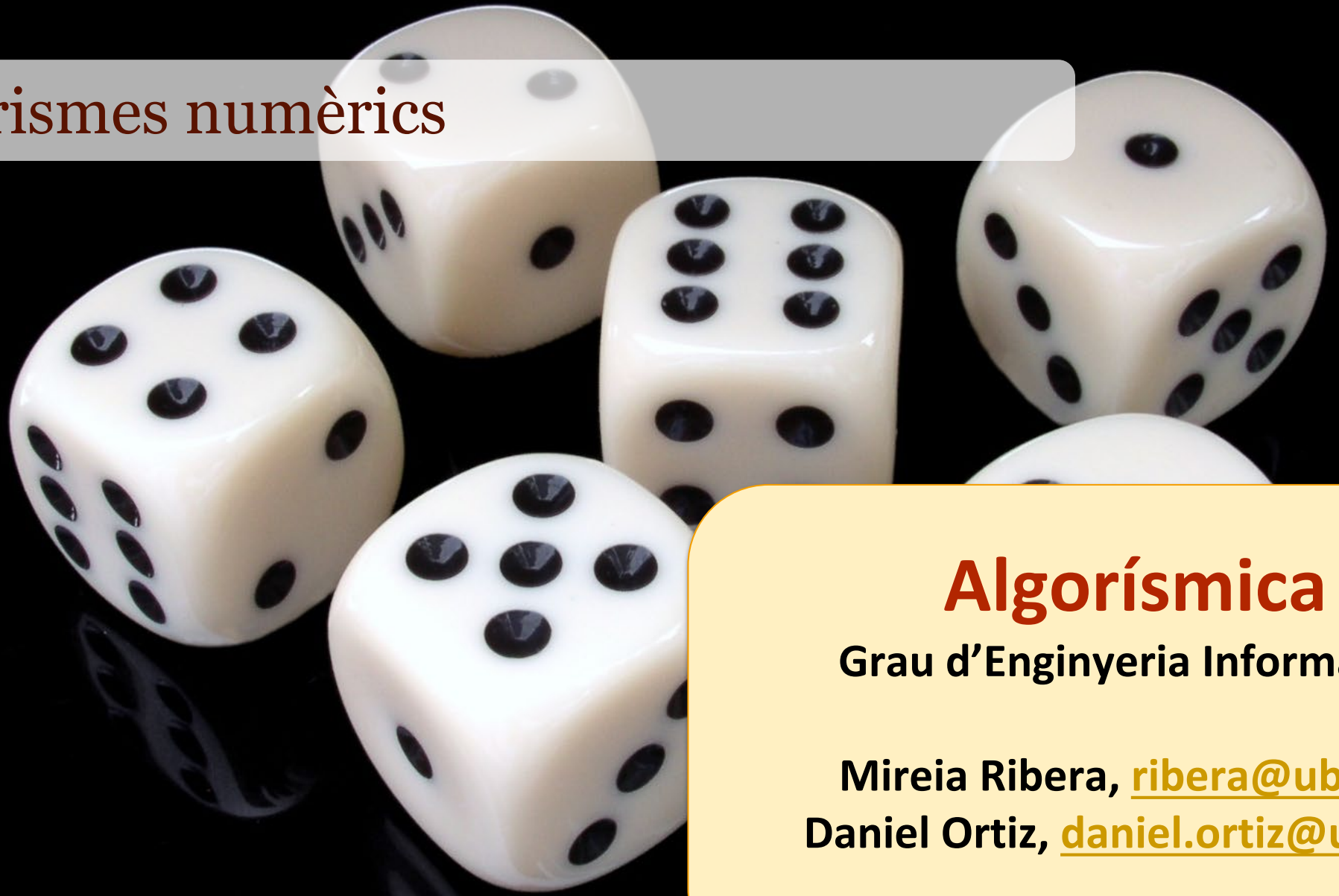
Algorismes numèrics



Algorísmica

Grau d'Enginyeria Informàtica

Mireia Ribera, ribera@ub.edu
Daniel Ortiz, daniel.ortiz@ub.edu

Algorismes numèrics: Conceptes clau

1. Complexitat: Passos computacionals i Gran O
2. Sistemes de numeració
3. Bases decimal, binària i hexadecimal
4. Nombres primers
5. Teoremes de Fermat, Lagrange i sedàs d'Eratóstenes

Complexitat, passos computacionals i O gran

A la introducció de l'assignatura hem dit que un algorisme ha de ser eficient. Però com ho podem mesurar?

Parlarem de complexitat dels algorismes i buscarem algorismes de complexitat baixa.

Recordem el cas dels sensor de soroll treballa a problemes...

Si l'ordinador ha de fer 6 operacions per cada registre d'un sensor... quantes operacions ha de fer per TOTS els sensors en UN DIA COMPLET?

- els passos que ha de fer un algorisme fins arribar a la solució, i
- com augmenten aquests passos quan augmenta la grandària de les dades d'entrada.

Passos computacionals

Considerarem que l'ordinador fa un pas computacional cada vegada que fa una instrucció simple: emmagatzema un enter o un caràcter a memòria, fa una comparació o una suma simples... Per exemple:

```
a:int = 5                   # 1 pas
for i in range(3):          # El de dintre x 3 (0, 1, 2)
    a = a + 2                   # 2 passos, suma + emmagatzematge
print(a)                    # 1 pas

>11
```

Aquest codi fa 8 passos.

L'única excepció a aquesta regla és si les dades que treballem són molt grans i ocupen més de 64 bits, perquè llavors l'ordinador ja no les pot operar en un sol pas.

Passos computacionals: nombres grans

Les operacions amb nombres grans ja no són tan barates!

```
import math

a:int = 12345851275275752352349823745982345
b:int = 81123875127592875128758512857891 27
for i in range(3):
    a += math.sqrt(a+b)
print(a)
```

En aquest tema (algorismes numèrics) veurem quin cost tenen les operacions aritmètiques amb nombres grans.

Complexitat segons la mida de l'entrada

Per poder comparar uns algoritmes amb els altres i tenir una idea dels passos computacionals que executaran quan les dades creixin, mesurem els passos en funció de la grandària de l'entrada.

```
list = [1, 2, 3, 4, 5]
for i in list:
    print(i)
```

Quants passos fa? 1 per cada element de la llista. Si al nombre d'elements de la llista li diem n, aquest codi fa n passos.

```
list = [1, 2, 3, 4, 5]
for i in list:
    for j in list:
        print(i, j, end=",")
```

Quants passos fa? n per cada element Si al nombre d'elements de la llista li diem n, aquest codi fa n x n passos, n^2.

I aquest codi?

```
list = [1, 2, 3, 4, 5]
for i in list:
    print(2 * i)
```

Complexitat: Gran O

En alguns algorismes el cost depèn dels valors de l'entrada, i podríem considerar el millor cas, el cas mig o el pitjor cas. A Algorísmica considerarem només el pitjor cas, la Gran O.

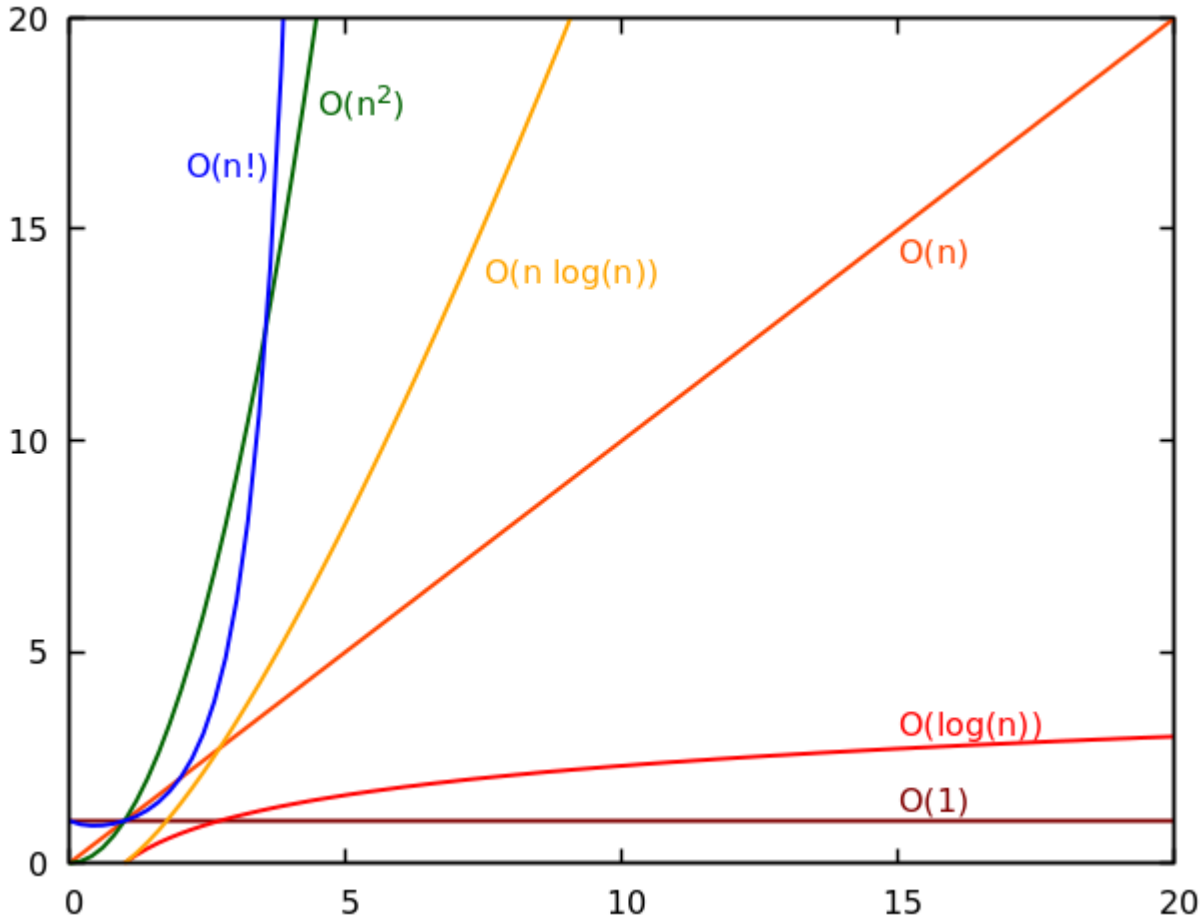
I per poder comparar fàcilment molts algorismes diferents farem una aproximació dels passos computacionals, i només ens fixarem en els termes que aporten més complexitat a l'algorisme:

- Ometrem constants multiplicatives: $14n^2$ serà n^2
- Si el cost està expressat com a $\log(n^a) + n^b$ descartarem el logaritme i només considerarem el polinomi n^b
- Si el cost està expressat com a $n^a + n^b$ només considerarem l'índex major, si $a > b$, n^a
- Si el cost està expressat com a $3^n + 2^n + n^b$ només considerarem l'exponencial més gran, 3^n

Vegeu el notebook Teoria2-Complexitat.ipynb per entendre com calcular els passos computacionals.

Complexitat: Gran O

La Gran O ens permetrà agrupar els algoritmes en famílies i veure com evolucionen quan la n creix.



Complexitat: Gran O, aproximació quantitativa

Si ho mirem en termes absoluts, podem fer algunes afirmacions:

- A partir de n = 20, els algorismes factorials O(n!) ja no tenen sentit
- Els algoritmes exponencials $O(C^n)$ són inútils a partir de n = 40
- Els algoritmes quadràtics $O(n^2)$ comencen a ser costosos a partir de n = 10.000 i a ser inútils a partir de n = 1.000.000
- Els algoritmes lineals O(n) o super-lineals O(n log n) poden arribar fins a n = 1.000.000.000
- Els algoritmes logarítmics O(log n) i els constants O(1), són útils per qualsevol n.

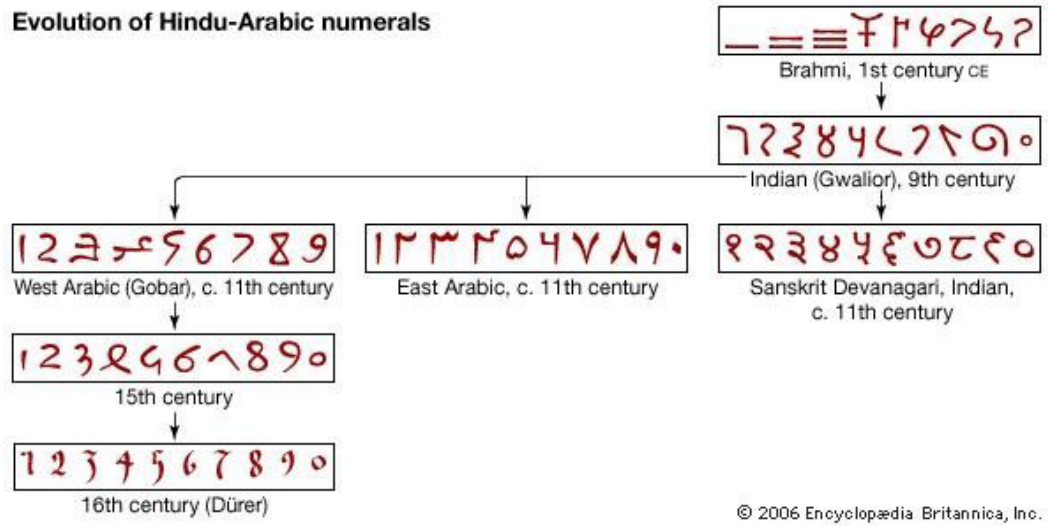
n	n^2	$n!$
5	25	120
6	36	720
7	49	5.040
8	64	40.320
9	81	362.880
10	100	3.628.800

Ara et toca...

Copia el codi de les solucions del tema 1 demanar_números i factorial, i calcula'n els passos computacionals i la complexitat O gran.

Sistemes de numeració

Un sistema de numeració ve definit pel conjunt S de símbols que el formen i el conjunt R de les regles de generació. Comparem el sistema decimal (inventat a l'Índia cap a l'any 600) amb el sistema romà. Quins símbols tenen? Quines regles de generació?



I II III IV V
VI VII VIII
IX O

El sistema decimal té una notació posicional, en la que cada dígit té un valor depenent de la seva posició relativa. Això en simplifica les operacions i permet fer algorismes més eficients.

Molt difícil...

Escriu una funció que sumi nombres romans amb valors entre 1 al 10, i una que sumi nombres decimals amb valors entre 1 i 10. Pots fer servir operacions i funcions bàsiques de Python.

L'arribada del sistema decimal a Europa

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

Ho va fer en un manual, escrit en àrab al segle IX a Bagdad, obra d'Al Khwarizmi en el que especificava els procediments per sumar, multiplicar i dividir de manera precisa, no ambigua, mecànica i eficient. Eren els primers algorismes implementats en paper.

Font: Viquipèdia

Al Khwarizmi

Font: Viquipèdia

Leonardo Fibonacci

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

Bases decimal, binària i hexadecimal

En els sistemes numèrics posicionals definim una base per la qual es multipliquen els nombres segons la posició. Per ex. 642 en sistema decimal és 6 x 10 x 10 + 4 x10 + 2

Podem generalitzar aquesta fórmula per a qualsevol base b:

$$d_n * b^{n-1} + \ldots + d_2 * b + d_1$$

Per tant:

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

A informàtica són importants les bases:

- decimal: la base és 10, el conjunt de símbols és {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- binària: la base és 2, el conjunt de símbols és {0,1}
- hexadecimal: la base és 16, el conjunt de símbols és {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

Bases decimal, binària i hexadecimal

Què vol dir 11 en decimal? i en binari? i en hexadecimal?

Pots escriure de 1 a 20 en binari i en hexadecimal?

Suma 3 + 4 en binari

Bases decimal, binària i hexadecimal

Quants dígits té el nombre N en base b?

Quin és el número més gran que puc expressar amb N bits?

Quin és el número més petit que puc expressar amb N bits (sense zeros al davant)?

Ara et toca...

Escriu una funció per llegeixi un nombre en una base donada. L'entrada serà el nombre i la base, i la sortida el seu valor en decimal.

Nombres primers

Els nombres primers tenen propietats que els fan ideals per criptografia, per a hashing o accés ràpid a dades, i per a detectar errors en el codi... En aquesta assignatura en veurem dues aplicacions: la criptografia i el hashing.

Com determinen que un nombre és primer?

Els següents nombres són primers? 10007, 127, 27, 23, 33, 10000169

Nombres primers

Els nombres primers tenen propietats que els fan ideals per criptografia, per a hashing o accés ràpid a dades, i per a detectar errors en el codi... En aquesta assignatura en veurem dues aplicacions: la criptografia i el hashing.

Com determinen que un nombre és primer?

Típicament ho resolem factoritzant, és a dir fent un recorregut per tots els nombres més petits per validar que no és divisible per cap nombre excepte 1.

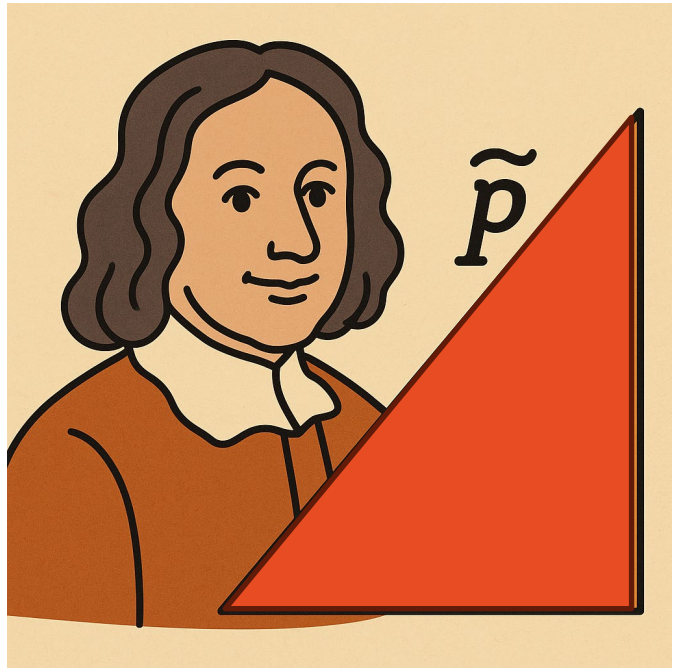
Els següents nombres són primers? 10007, 127, 27, 23, 33, 10000169

Teoremes de Fermat, Lagrange. Sedàs d'Eratóstenes

Per poder tractar amb els nombres primers a l'ordinador veurem tres algorismes, basats en teoremes matemàtics:

- Un test de primeritat, basat en el petit teorema de Fermat – que ens permetrà determinar de manera eficient si un nombre és o no primer.
- Una tècnica per generar nombres primers grans i saber que en trobarem, basada en el teorema dels nombres primers de Lagrange.
- Un mètode per generar tots els nombres primers menors que *N*, a partir de l'algorisme ideat per Eratòstenes.

Test de primeritat: petit teorema de Fermat



Petit teorema de Fermat

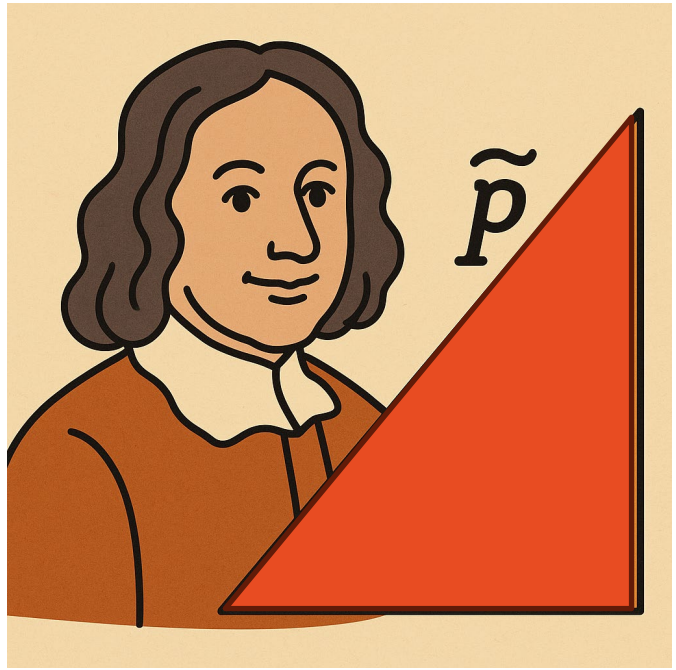
Si p és primer, llavors per a qualsevol enter a, $1 \leq a < p$, es compleix que $a^{(p-1)} \% p = 1$

Pierre de Fermat, un gran matemàtic francès, l'any 1640 va establir **un teorema per comprovar si un nombre és primer sense haver-lo de factoritzar**, amb el qual va assentar la base per escriure un algorisme correcte i eficient.

Aquest teorema és necessari però no suficient. És a dir **alguns nombres t el compleixen i NO són primers**.

Test de primeritat: petit teorema de Fermat

El teorema petit de Fermat ens diu que per a tot nombre a més petit que p, es compleix $a^{(p-1)} \% p = 1$. Per tant, si trobem un nombre que no compleix ja podem dir que p no és primer. Ho hem de provar amb tots? Idealment sí però no és gaire eficient...




```python
import random
def fermat(num, test_count): #volem saber si num és primer
    if num == 1:
        return False
    for x in range(test_count):
        val = random.randint(1, num-1)
        if pow(val, num-1) % num != 1:
            return False
    return True
```

Test de primeritat: petit teorema de Fermat

```
import random
def fermat(num, test_count): #volem saber si num és primer
    if num == 1:
        return False
    for x in range(test_count):
        val = random.randint(1, num-1)
        if pow(val, num-1) % num != 1:
            return False
    return True
```

Aquest algorisme prova el teorema amb test_count nombres, només uns quants.

Aquest algorisme té dues limitacions:

- Hi ha alguns nombres, els nombres de Carmichael, que compleixen el teorema de Fermat i no són primers. (hi ha variants de l'algorisme que els eviten).
- No ho podem provar amb tots els números, però sabem que en els nombres que no són de Carmichael, si a < N el petit teorema de Fermat fallarà la meitat de les vegades.

Test de primeritat: petit teorema de Fermat

Com es comportarà el test (si ignorem els nombres de Carmichael)?

- El test retornarà True en tots els casos si N és primer.
- El test retornarà True per la meitat o menys dels casos en que N no és primer.
- Si repetim el test k vegades amb nombres a escollits aleatòriament, la probabilitat que l'algorisme retorni True quan N no és primer és menor que $(1/2)^{\mathrm{k}}$
  - Si $\mathrm{k} = 100$ la probabilitat és menor que $(1/2)^{100}$
- Amb un nombre moderat de tests podem determinar si un nombre és primer.

Aquesta tasca acaba tenint una complexitat O(n^3), on n és el nombre de dígits del nombre que volem testejar.

Com generar nombres primers grans

Teorema de Lagrange

La probabilitat que un nombre de n bits sigui primer és aproximadament:

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

Joseph-Louis Lagrange, un gran matemàtic italià, l'any 1770 va quantificar el que costa trobar un nombre primer.

Per ex. per a generar un nombre primer de 1000 dígits ens caldrà generar 1000 nombres aleatoris.

Trobar un nombre primer té una complexitat O(n), on n és el nombre de dígits del nombre que volem generar.

Tot junt

```python
import random

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

```python
def generar_primer(n):
    trobat = False
    v_minim = 2**(n-1)
    v_maxim = 2**n
    while not trobat:
        p = random.randint(v_minim, v_maxim)
        trobat = fermat(p, 10)
    return p
```

Com generar molts nombres primers

El sedàs d'Eratòstenes

1. S'escriu una llista amb els nombres des del 2 fins a N
2. El primer nombre de la llista és un nombre primer i es guarda a la llista de nombres primers
3. S'esborra de la llista A el primer nombre i els seus múltiples
4. Si el primer nombre de la llista A és més petit que $\sqrt{N}$ es torna al punt 2.
5. Els nombres de la llista B i els que queden a la llista A són els nombres primers cercats.

Eratòstenes, un matemàtic, astrònom i geògraf grec, va viure el segle II aC. i se li atribueix un algorisme per trobar tots els nombres primers fins a un determinat enter.

Aquesta tasca té una complexitat O (n log n).

El sedàs d'Eratòstenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Vegeu el codi al notebook Tema2Numerics-Codi.ipynb

Gràcies i recapitulació

MIREIA RIBERA | ACCESSIBILITAT DIGITAL
| EXPERIÈNCIA D'USUARI
| VISUALITZACIÓ DE DADES

UNIVERSITAT DE BARCELONA

Recapitulació

⏱️ Per saber si un algorisme és eficient en calculem la complexitat amb l'O gran.

🔢 Els sistemes de numeració posicionals faciliten la definició d'algorismes numèrics

🖥️ A informàtica cal familiaritzar-se amb el sistema de numeració binari i hexadecimal

🔐 Els nombres primers són molt rellevants per criptografia.

⏳ Saber si un nombre és primer per factorització és molt costós, el teorema petit de Fermat ens facilita identificar nombres primers, amb certes limitacions.

🛠️ Gràcies a Lagrange i a Eratòstenes, sabem com crear nombres primers fins a un determinat nombre, i que trobar-ne un de molts dígits no serà impossible.

Algorismes numèrics: Aritmètica bàsica

1. Suma binària, de complexitat O(n)
2. Multiplicació binària, de complexitat O(n^2)
3. Divisió binària, de complexitat O(n^2)

Aritmètica bàsica, preliminar

Les operacions aritmètiques bàsiques, com sumar, multiplicar... amb nombres petits signifiquen un pas computacional.

Ara bé, quan els nombres són molt grans l'ordinador els tracta per parts.

- En aquesta secció veurem el cost de les operacions bàsiques però per a nombres molt grans amb la complexitat O(). La n en aquest cas serà la quantitat de dígits del nombre.

  (per ser rigorosos hauríem de comptar cada 64 bits, que és la mida dels operands dins l'ordinador, però simplificarem i comptarem cada bit)

- Per altra banda treballarem amb nombres binaris, perquè són més simples d'operar.

  El cost de les operacions és equivalent en complexitat O() que en nombres decimals (la diferència és un factor multiplicatiu, que obviem).

Aritmètica bàsica, suma

Tots els nombres compleixen la següent propietat:

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

Això ens permet definir l'algorisme general de la suma, vist a l'escola, del que veurem la complexitat a continuació:

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)

Algorisme bàsic de la suma, en binari

Suma (x,y), complexitat O(n)

Considerem que els dos operands, x, y tenen n bits.

Com hem vist abans, la suma té com a màxim n+1 bits.

La complexitat és O(n) perquè es fan tantes sumes unàries com bits tenen els operands.

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)

Es pot pensar un algorisme més eficient?

NO! Perquè només llegir i escriure els valors ja és ordre O(n).

Tenim n+1 operacions de complexitat 1, per tant la complexitat és $O(n)$

Aritmètica bàsica, multiplicació

L'algorisme que coneixem de la multiplicació és el següent:

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

Tenim n multiplicacions de complexitat n (un bit per n bits) + aproximadament $2 * n$ sumes de complexitat n, per tant $n^2 + 2n^2 = 3n^2$, per tant la complexitat és $O(n^2)$

Tenim n operacions de complexitat n per tant la complexitat és $O(n^2)$

Multiplicació (x,y), complexitat $O(n^2)$

L'algorisme rus que vam veure al primer tema, consistia en multiplicar i dividir per 2 fins arribar al resultat

11	13
5	26
2	52
1	104
	143

Tenim 3 x n operacions de complexitat n (desplaçar n bits), per tant $3n^2$, i la complexitat també és d'$O(n^2)$

Es pot pensar un algorisme més eficient?

Sí! Només una mica. Ho veurem més endavant.

Aritmètica bàsica, divisió

La divisió x/y consisteix en trobar un quocient q i una resta r, tal que $r < y$ i que compleixin:

$$x = y * q + r$$

Podem fer-la amb un algorisme en dues passades: primer anem reduint la x per la meitat, després segons el mòdul de la divisió per 2 anem augmentant r i q. Quan r és més gran que y, li restem i incrementem q. Veiem un exemple amb x = 22 i y = 3

y	x
3	22
3	11
3	5
3	2
3	1
3	0

Dividim per 2 la x

X, %2	q	r
0,0	0	0
1, 1	x2 = 0	x2 +1 = 1
2, 0	x2 = 0	x2 = 2
5, 1	x2 +1 = 1	x2 +1 = 5 – y = 2
11, 1	x2 +1 = 3	x2 +1 = 5 – y = 2
22, 0	x2 +1 = 7	x2 = 4 – y = 1

x2

x2

Aritmètica bàsica, divisió, codi

```python
import math

def divisio(x:int,y:int) -> list[int]:
    if x <= 0:
        return 0, 0
    if y == 1:
        return x, 0
    q, r = divisio( 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]
```

Tenim n operacions de complexitat n per tant la complexitat és $O(n^2)$

Ara et toca...

Escriu una funció calculadora, en la que entrem 2 enters i una operació (multiplicació o divisió) i crida l'algorisme rus o el codi de la divisió i en dóna el resultat.

Gràcies i recapitulació

MIREIA RIBERA | ACCESSIBILITAT DIGITAL
| EXPERIÈNCIA D'USUARI
| VISUALITZACIÓ DE DADES

UNIVERSITAT DE BARCELONA

Recapitulació

🔢 Les operacions amb nombres grans no tenen cost unari

🧮 Per a calcular l'ordre de complexitat és equivalent treballar amb nombres binaris, decimals o hexadecimals.

➕ L'algorisme de suma que coneixem no es pot millorar i té un cost $O(n)$

✖️ L'algorisme de multiplicació que coneixem o el de multiplicació russa tenen un cost $O(n^2)$.

En alguns casos es poden millorar una mica però ho veurem més endavant.

➗ L'algorisme de divisió té un cost d'$O(n^2)$

Algorismes numèrics: Aritmètica modular

1. Aritmètica modular i congruència. Operació mòdul, de complexitat O(n^2)
2. Suma modular, de complexitat O(n)
3. Multiplicació modular, de complexitat O(n^2)
4. Divisió modular, de complexitat O(n^3)
5. Exponenciació modular, de complexitat O(n^3). Funció pow de Phyton

Aritmètica modular

L'aritmètica modular ens dóna eines que milloraran l'eficiència d'alguns algorismes coneguts i en permet d'altres de criptografia.

Definim l'operació mòdul, o $x \% N$, com la resta de dividir x per N.

Si tenim que $x = y * q + r$, amb $0 \leq r < N$, el resultat de l'operació mòdul és r.

Per exemple $12 \% 7 = 5$,

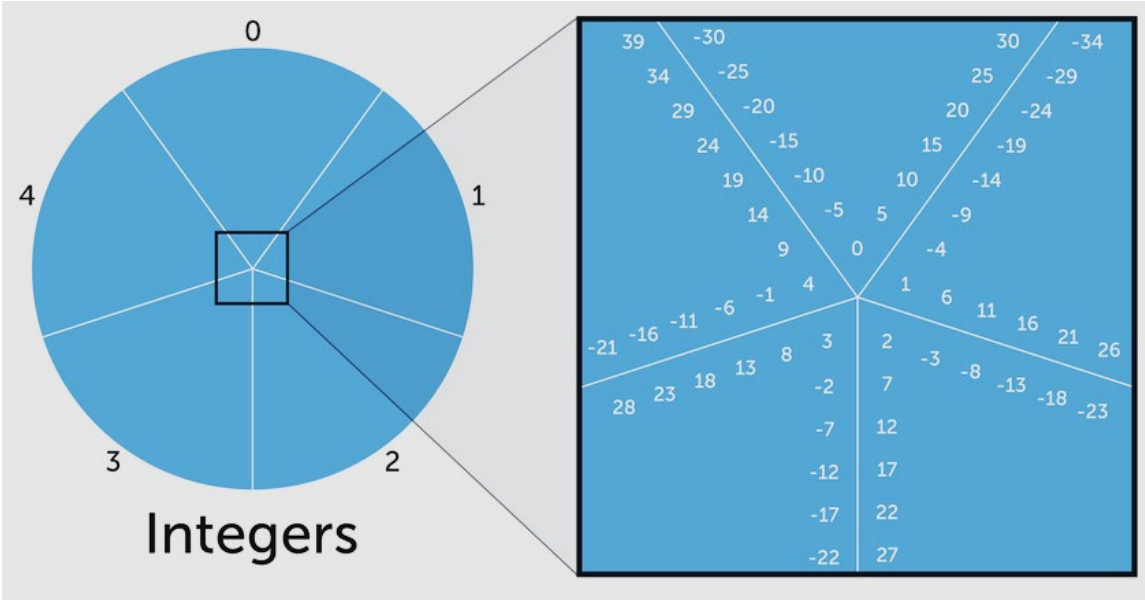
i $100 \% 12$ és 4.

Quan treballem en aritmètica modular tots els operands i els resultats han d'estar **en mòdul N**

Donat un N, tenim N -1 famílies de nombres que tenen el mateix mòdul. Són nombres congruents.

Donat un N, x és congruent amb y, si $(x - y) \% N = 0$

A continuació es dibuixen les famílies de nombres congruents de 5.



Famílies de congruències mòdul 5. Font: Acadèmia Khan

Aritmètica modular, suma (a + b) % N

Si a i b tenen un valor entre [0, N-1], la seva suma està en el rang [0,2(N-1)]

La suma modular consisteix en

- fer la suma bàsica i si resultat > N – 1, resultat = resultat – N
- Exemples
  - (11 + 8) % 12 = 19 ... = 19 – 12 = 7
  - (4 + 3) % 5 = 7... = 7 – 5 = 2

L'ordre de complexitat és el mateix que la suma de l'aritmètica bàsica, però cal tenir en compte que el nombre de bits serà menor.

Tenim n+1 operacions de complexitat 1, per tant la complexitat és $O(n)$

Aritmètica modular, multiplicació (a * b) % N

Si a i b tenen un valor entre $[0, \mathrm{N}-1]$, la seva multiplicació està en el rang $[0, (\mathrm{N}-1)^2]$, que es pot representar amb $2\mathrm{n}$ bits.

- Fem la multiplicació bàsica
- Calculem el mòdul N del resultat. Això representa una divisió, amb complexitat $\mathrm{O}(\mathrm{n}^2)$
  - (11 * 8) % 12 = 88 ... = 88 % 12 = 4
  - (4 * 3) % 5 = 12... = 12 % 5 = 2

Tenim dues operacions d'ordre $\mathrm{O}(\mathrm{n}^2)$, la complexitat final és $\mathrm{O}(\mathrm{n}^2)$

Aritmètica modular, divisió (a / b) % N

Aquesta operació en aritmètica modular no és tan simple. No està definida per tots els nombres.

No la veurem, però sabem que la seva complexitat és $O(n^3)$

La complexitat final és $O(n^3)$

Aritmètica modular, exponenciació (x^y) % N

Si x i y tenen un valor entre $[0, N-1]$, per fer aquesta operació de manera eficient veurem un petit truc:

- x^2 equival a x * x
- x^4 equival a $x^2 * x^2$
- x^8 equival a $x^4 * x^4$, i així successivament
- Qualsevol exponent el podem expressar com un producte d'exponents potències de 2

Per ex, $x^{25} = x^{11001} = x^{10000} * x^{1000} * x^1 = x^{16} * x^8 * x^1$

Cadascun d'aquests operands tindrà un valor entre $[0, N-1]$, i per tant tenim n multiplicacions modulars, de complexitat $O(n^2)$, on n és el nombre de dígits de l'exponent y.

La complexitat final és $O(n^3)$

Exponenciació modular

Quantes multiplicacions em calen per calcular x^8

I per calcular x^{16}

I per calcular x^{31}?

Funció pow de Python

Python aprofita l'eficiència de l'exponenciació modular amb la funció pow (de power):

```
pow (x, y)
pow (x, y, m ) #calcula eficientment (x ** y) % m (només amb enters).
```

Amb només dos paràmetres calcula x^y però amb tres, calcula $x^y \% m$, és a dir, mòdul m, usant l'exponenciació per quadrats.

Ara et toca...

Fes una implementació pròpia de la funció pow de Python.

Gràcies i recapitulació

MIREIA RIBERA | ACCESSIBILITAT DIGITAL
| EXPERIÈNCIA D'USUARI
| VISUALITZACIÓ DE DADES

UNIVERSITAT DE BARCELONA

Recapitulació

➕ ✖️ La suma i multiplicació modular tenen el mateix ordre de complexitat que la suma i multiplicació de l'aritmètica bàsica, però n sempre serà molt més petita.

➗ La divisió té un cost superior , d'$O(n^3)$

✖️✖️ L'exponenciació té un cost $O(n^3)$. És molt eficient!

🐍 La funció pow de Python aprofita l'aritmètica modular per calcular els exponents de manera eficient.

Algorismes numèrics: Una aplicació, la criptografia

1. Enviament i lectura de missatges secrets
2. Operacions involucrades i cost

Criptografia com a usuaris

Alice

Clau pública: (n=638479, e=65537)

Clau privada: (n=638479, d=222753)

Bob

Missatge: M = 324

Missatge encriptat: C = $\mathrm{M}^{\mathrm{e}} mod\, n \;=\; 324^{65537} mod\, 638479 \;=\; 30547$

Eve

30547???

Alice

Missatge encriptat: C = 30547

Missatge desencriptat: R = $\mathrm{C}^{\mathrm{d}} mod\, n \;=\; 30547^{222753} mod\, 638479 \;=\; 324$

Usa les funcions generar_claus(), encriptar_missatge i desencriptar_missatge del notebook Tema2Numerics-Codi.ipynb.

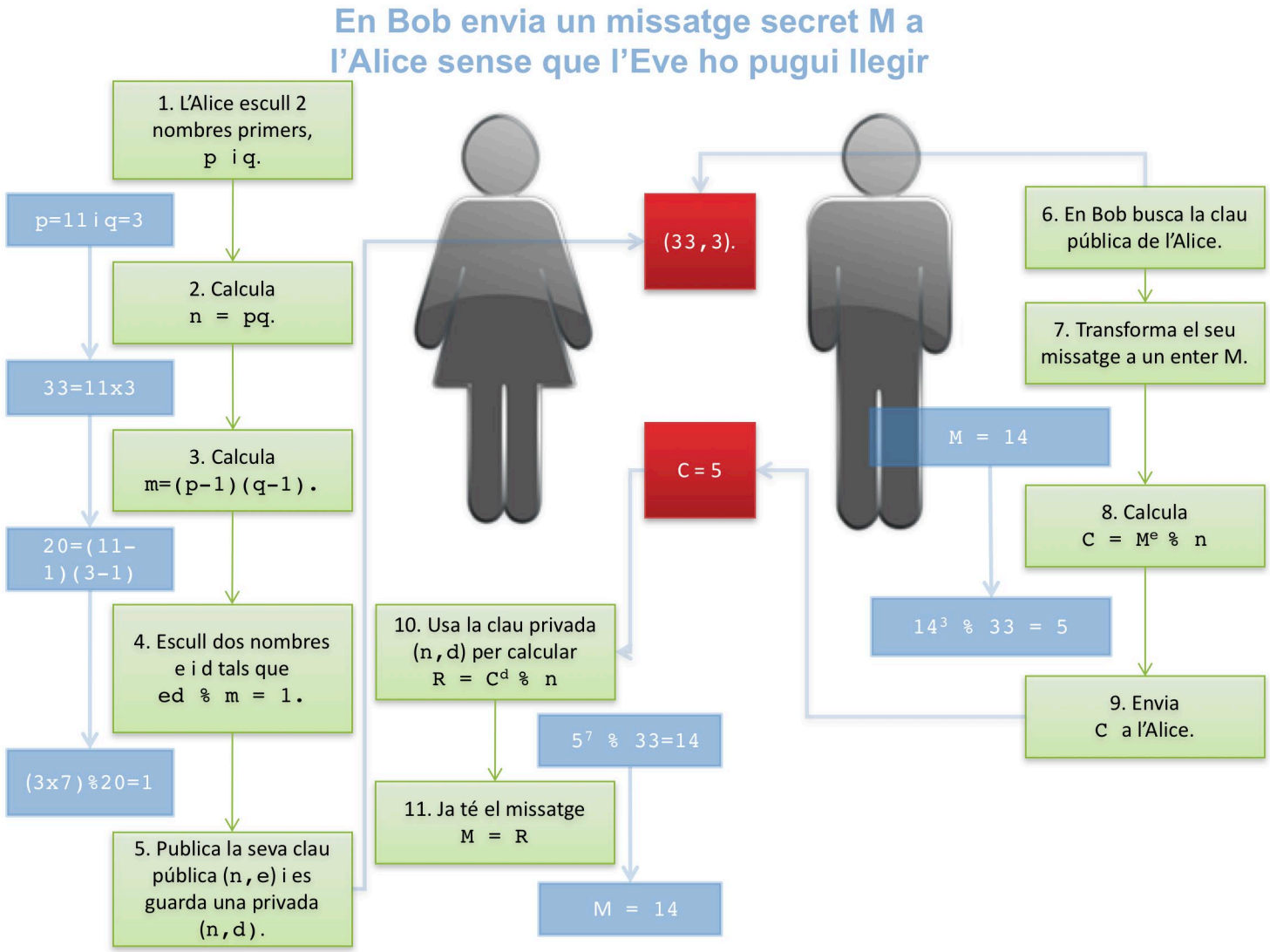
1) comparteix la clau privada amb la teva/el teu company/a. Ell/a farà el mateix.

2) Encripta un missatge per enviar-li.

3) Desencripta el missatge que t'ha enviat.

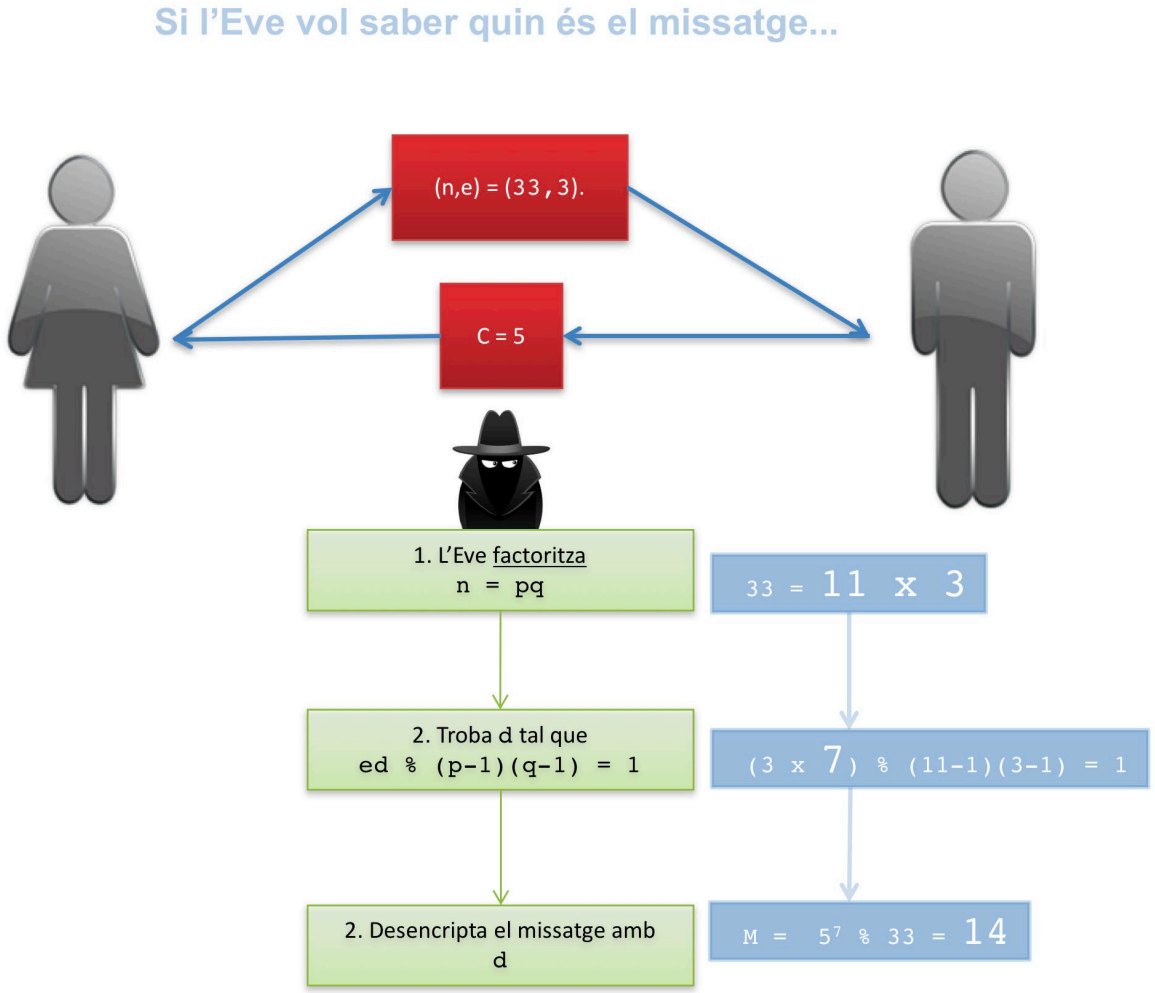
Encriptació, explicació matemàtica

Visió general del procés.



Encriptació, explicació matemàtica

Visió general del procés (II).



Per què funciona l'encriptació?

Quines operacions han de fer Bob i Alice?

Són eficients?

Quines operacions ha de fer l'Eve?

És eficient?

Gràcies i recapitulació

MIREIA RIBERA | ACCESSIBILITAT DIGITAL
| EXPERIÈNCIA D'USUARI
| VISUALITZACIÓ DE DADES

UNIVERSITAT DE BARCELONA

Recapitulació

- 🔐 Els nombres primers i els algoritmes estudiats ens permeten encriptar amb seguretat.