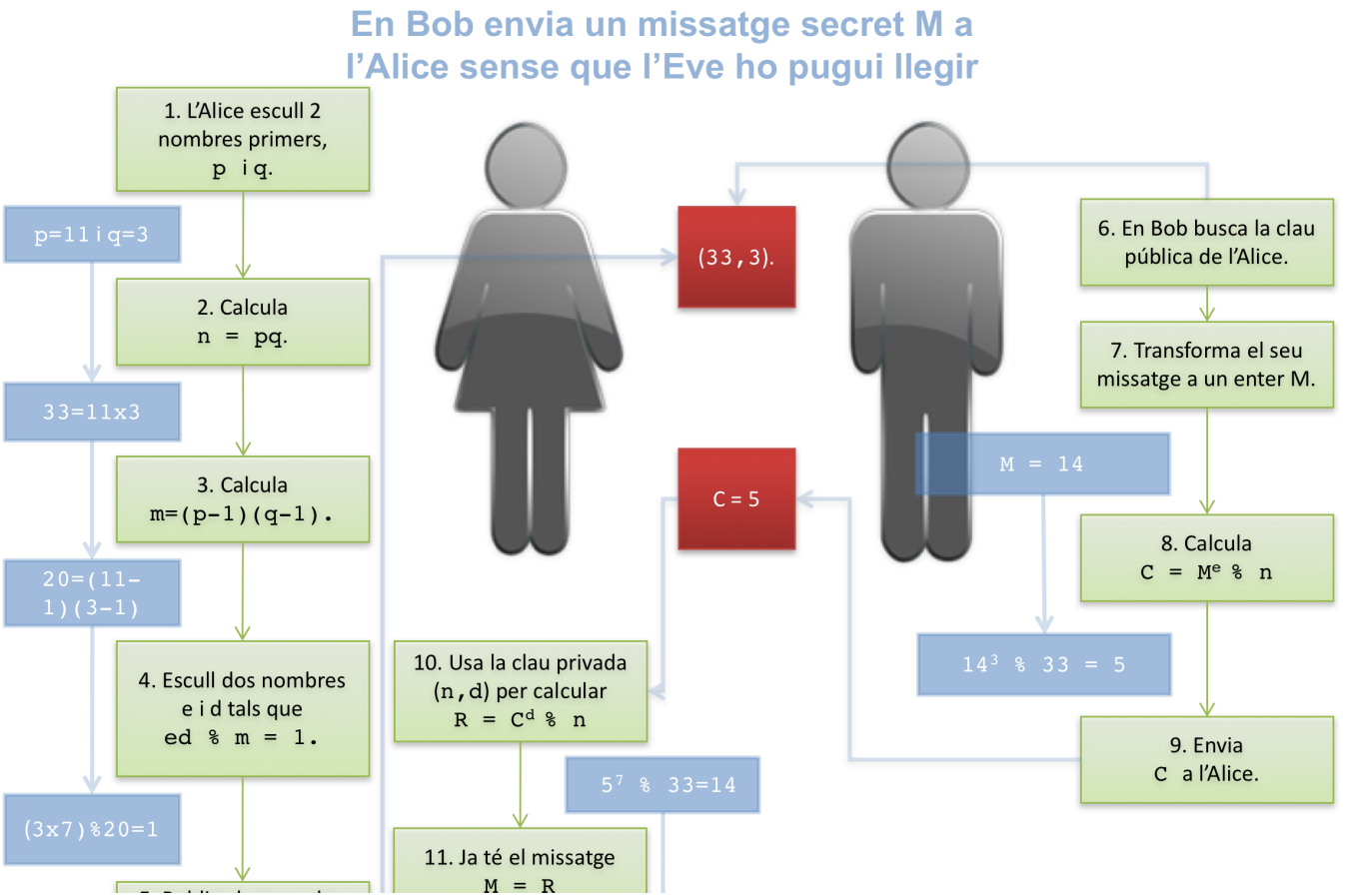
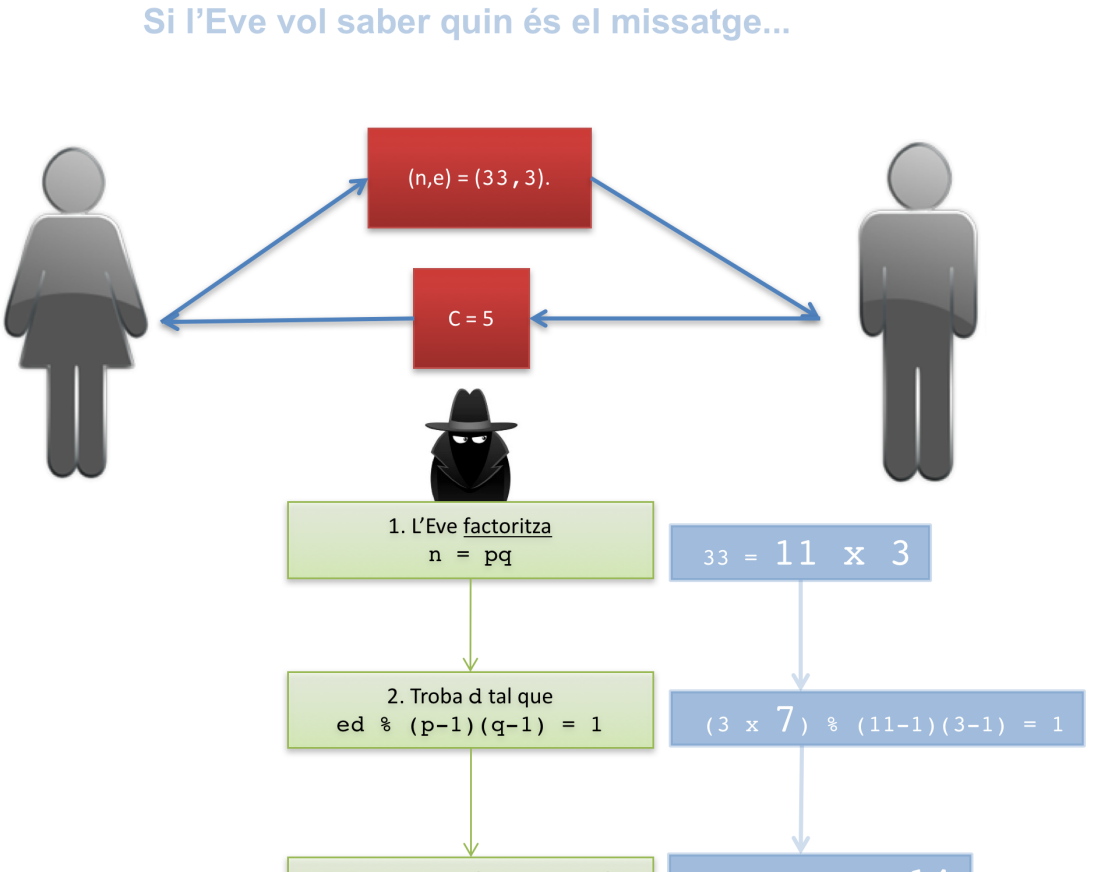
Aritmètica Modular

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

Com enviar un missatge secret?



Com enviar un missatge secret?



Com enviar un missatge secret?

Aquest esquema té sentit si:

- Factoritzar `n = p*q` és impossible.
- Trobar `(p,q)` “grans” es basa en un mètode eficient.
- Calcular `x^y % n` es es basa en un mètode eficient.
- Calcular `e*d % (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**.

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

Per exemple 12 % 7 és 5, i 100 % 12 és 4.

Això permet definir una equivalència (congruència) entre nombres (inclosos els negatius!). Direm que `x` és congruent amb `y`, `mod N`, si i només si `N` divideix `(x-y)`.

Per exemple 16, 28, 40 i 100 són congruents entre sí mòdul 12.

*La complexitat és `O(n^2)`

Aritmètica Modular

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

Suma modular `(a + b) % N`:

Sabem que `(a + b) % N = (a % N + b % N) % N`

A més, sabem que si dos nombres estan el rang `[0, N-1]` (`a % N` i `b % N` ho estan) la seva suma està en el rang `[0, 2(N-1)]` (que només és un bit més).

Com que els operands ja estan en mòdul N, fem la suma bàsica i l'únic que hem de tenir en compte és que si el resultat passa de `N-1` li hem de restar `N`. Altrament, no hem de fer res.

Exemple: `(11 + 8) % 12 = 19` => com que passa de `11 => 19 - 12 => 7`

Per tant, **la complexitat de la suma modular és lineal `O(n)`**, on `n` és el nombre de dígits de `N`*.

* Recordem que per escriure `N` en base `b` necessitem `n = log_b(N)` dígits.

Aritmètica Modular

Multiplicació modular (a * b) % N:

Sabem que `(a * b) % N = (a % N * b % N) % N`

A més, sabem que el producte pot ser fins `(N-1)^2` i que això es pot representar amb `2n` bits. Per transformar el resultat hem de dividir per `N` i calcular el mòdul (amb complexitat `O(n^2)`).

De forma semblant a la suma, fem la multiplicació bàsica i transformem al rang `[0,N-1]`, si és que ens hem passat.

Exemple: `(11 * 8) % 12 = 88 % 12` => com que passa de `11 => 88 % 12 => 4`

Per tant, **la complexitat de la multiplicació modular 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):

$$x^y \bmod N$$

- El resultat intermig d'aquesta operació pot necessitar molts bits per ser representat, tot i que el resultat final necessita només `n = log(N)` bits. Si els operadors tenen 20 bits, necessitem 10 milions de bits!

```
>>> (155 ** 245)
427824894297936883715416303810559372169915876329527628346848602611690253546055884268710170564791547920936871133273639806030034911510400847598847975741337732142515770352938597064824736760699623574403668342124231048856816364528995889978321141331260938990745511620992751010056642232660973228260184874531641949264591590378009465446342528981341951429642759537873516202755230036214044808536307148121446731580433384923331951608734536640900405562488098895844580440209740364076761507446284998849068230137963897430353199524688534438610076904296875
```

Aritmètica Modular

Una solució és fer totes les operacions intermèdies mòdul `N`.

O sigui, calcular `x^y mod N` fent `y` multiplicacions successives per `x mòdul N`.

$$x \bmod N \rightarrow x^2 \bmod N \rightarrow x^3 \bmod N \rightarrow \ldots x^N \bmod N$$

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 `2^(y-1)` multiplicacions (o sigui, anem a una complexitat d'ordre `2^500`) i l'algorisme és exponencial sobre `n`, la mida de `y`.

Aritmètica Modular

Però una petita modificació pot ser un gran canvi!

- Si l'exponent fos `2`, `x^2 mod N` seria equivalent a una simple multiplicació `(x * x) mod N`, d'ordre `O(n^2)`.
- Generalitzant, observem que si `y` és una potència de 2, es pot calcular `x` elevat a `y` elevant al quadrat, mòdul `N`, successivament:

$$x \bmod N \to x^2 \bmod N \to x^4 \bmod N \to \ldots x^{2^{\log y}} \bmod N$$

Exemple:

```
5^8 mod 12 = (((((5 mod N)^2)mod N)^2 mod N)^2 mod N)

390625 mod 12 = (((25 mod 12)^2 mod 12)^2 mod 12) = 1
```

Aritmètica Modular

Segons el que hem vist de la multiplicació, cada potència pren un temps proporcional a `O(log^2 N)` (o el que és el mateix `O(n^2)`) i hi ha `m = log_y` multiplicacions: l'algorisme és polinòmic, `O(n^2)`, respecte la mida de `N` i lineal `O(m)` respecte la mida de `y`.

El més interessant d'aquest resultat és que es pot generalitzar a valors de `y` que no són potències de dos amb un augment de cost molt reduït:

> Per un valor qualsevol de `y` (que no sigui potència de 2) només hem de reduir primer el nombre a la multiplicació de les potències de 2 que corresponen a la representació binaria de `y`, i per cadascuna aplicar l'algorisme anterior:

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

Aritmètica Modular

Aquestes operacions es poden expressar recursivament fent operacions mòdul `N`:

```
def modexp(x,y,N):
    import math
    if y == 0:
        return 1    # cas base x^0 dona 1

    z = modexp(x, math.floor(y/2), N)
                    # dividim la potència per 2 fins arribar al cas base
    if y%2 == 0:
        return (z**2)%N
                    # anem fent les potències de 2 mòdul N
    else:
        return (x*(z**2))%N
                    # la y inicial no és una potència de 2,
                    # cal multiplicar per x, també mòdul N
```

La **complexitat és `O(n^3)`**: `n` crides recursives i per cada una d'elles una multiplicació mòdul `N` d'ordre `n^2`.

Algorisme d'Euclides

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. Per exemple, el mcd de `1035` i `759`:

> `1035 = 32*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! No hi ha cap algorisme publicat per poder factoritzar-lo en temps polinòmic, és a dir, no existeix cap algorisme publicat que pugui factoritzar-lo en temps `O(n^k)` independentment de quina sigui la constant `k`.

Curiositat: El millor algorísme que es coneix té aquesta complexitat:

$$O\left(\exp\left(\left(\tfrac{64}{9}\ n\right)^{\frac{1}{3}}(\log n)^{\frac{2}{3}}\right)\right)$$

per factoritzar un nombre de `n` bits.

Algorisme d'Euclides

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

```
def gcd(a,b):
    while a:            # aquí es fa un truc, si 'a!=0', 'a' s'avalua com a True
        a,b = b%a, a
    return b

gcd(1071, 462)
> 21
```

Quina complexitat té per nombres grans?

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 mod a, a)`: 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.

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 mod a, a)`: 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.

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)`.

Nombres primers

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

Nombres primers

Test de primeritat: 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 quadrada 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 primeritat d'un nombre és un problema dur**, però això no és veritat: **només és dur si ho intentem pel camí de la factorització**!

Nombres primers

Una de les activitats bàsiques de la informàtica, la criptografia, es basa en el següent fet: **la factorització és dura, però la primeritat 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 l'any 1640...

Nombres primers

Una de les activitats bàsiques de la informàtica, la criptografia, es basa en el següent fet: **la factorització és dura, però la primeritat 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 l'any 1640...

Teorema petit de Fermat:

> Si `p` és primer, llavors per a qualsevol enter `a`, `1 <= a < p`, es compleix que `a^(p-1) % p = 1`.

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

Nombres primers

```
def fermat(num, test_count):
    if num == 1:
        return False
    for x in range(test_count):
        val = randint(1, num-1)              # genera nombre aleatori
        if pow(val, num-1, num) != 1:         # la potència a Python es
            return False                     # basa en l'algorisme modular
    return True

fermat(41651,10)
> True
```

Nombres primers

```
def fermat(num, test_count):
    if num == 1:
        return False
    for x in range(test_count):
        val = randint(1, num-1)              # genera nombre aleatori
        if pow(val, num-1, num) != 1:         # la potència a Python es
            return False                     # basa en l'algorisme modular
    return True

fermat(41651,10)
> True
```

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 i no són primers... però són pocs i és poc probable que en trobem un de forma aleatòria. Per altra banda existeixen algorismes modificats de Fermat que els eviten.

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

Nombres primers

Lema

> Si `a^(N-1)` no és congruent amb 1 mòdul `N` per algun `a` que sigui nombre compost però no de Carmichael, llavors com a mínim en la meitat dels casos en que `a < N` el teorema petit de Fermat fallarà.

Test de primeritat

Si ignorem els nombres de Carmichael, podem dir que:

- Si `N` és primer, llavors `a^(N-1)` és congruent amb 1 mòdul `N` per tots el `a < N`.
- Si `N` no és primer, llavors `a^(N-1)` no serà congruent amb 1 mòdul `N` per com a mínim la meitat dels valors `a < N`.

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

- 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.

Nombres primers: Algorisme de test de primeritat

Si repetim l'algorisme `k` vegades per nombres `a` escollits aleatòriament, llavors **la probabilitat de que retorni sempre `True` quan `N` no és primer és menor que `1/(2^k)`**.

Si `k=100`, la probabilitat és menor que `2^(-100)`.

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

Nombres primers: Algorisme de test de primeritat

```
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)
```

```
1 loop, best of 3: 2.85 s per loop
```

```
13344803736336051021810447750596416197709968368917104278447009456089567365038775197574289256044886865710306124092220160479184590986686464012079658080272098609817765459565243558950507944345133209107880252558574566897128124166511190105802
```

Nombres primers grans

Com és que l'algorisme anterior no ha tardat en trobar un nombre primer format per uns quants centenars de bits?

És difícil trobar nombres primers grans?

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:

$$\frac{1}{ln2^2} \approx \frac{1.44}{n}$$

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

Recapitulació

Abans hem dit que l'esquema de comunicació secreta té sentit si:

- Factoritzar `n = p*q` 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 `e*d % ((p-1)(q-1)) == 1` es basa en un mètode eficient.

Només ens falta solucionar el darrer punt!

Recapitulació

Abans hem dit que l'esquema de comunicació secreta té sentit si:

- Factoritzar `n = p*q` 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 `e*d % ((p-1)(q-1)) == 1` es basa en un mètode eficient.

Només ens falta solucionar el darrer punt!

La solució del darrer punt és:

- Definim `e=3`.
- 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 algorísmes de més alta complexitat en un procés criptogràfic tenen `O(n^3)`.

Possibles preguntes d'exàmen relacionades amb el tema 3

1. Tenim una n=30, té sentit un algorisme d'ordre n^2? i un d'ordre n!? perquè?
2. Suma 12 i 25 amb l'algorisme d'Al Khwarizmi
3. Quin ordre de complexitat tenen les següents operacions de l'aritmètica bàsica? i de l'aritmètica modular? suma, multiplicació, divisió
4. Quina probabilitat hi ha que un nombre de 2 bits sigui primer?
5. Perquè és segur enviar un missatge encriptat? quina dificultat troba qui el vulgui desxifrar?