In [1]:
```
import random
```

Algoritmos de Euclides

Antes de empezar necesitaremos un sistema para cálculo del máximo común divisor (mcd en castellano, gcd en inglés) de dos números, y el inverso de un número en un anillo cíclico. Ambas cosas se conocen desde hace tiempo: son dos "algoritmos de Euclides"

Algoritmo de Euclides para determinar el máximo común divisor (gcd) de dos enteros a yb

In [2]:
```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

print('gcd(2, 3) = ', gcd(2, 3))
print('gcd(20, 30) = ', gcd(20, 30))
print('gcd(50720, 48184) = ', gcd(50720, 48184))
```

```
gcd(2, 3) =  1
gcd(20, 30) =  10
gcd(50720, 48184) =  2536
```

Algoritmo generalizado de Euclides para encontrar el inverso multiplicativos de un número en un anillo cíclico $\mathbb{Z}_\phi$

In [3]:
```
def multiplicative_inverse(e, phi):
    d = 0
    x1 = 0
    x2 = 1
    y1 = 1
    temp_phi = phi

    while e > 0:
        temp1 = temp_phi // e
        temp2 = temp_phi - temp1 * e
        temp_phi = e
        e = temp2

        x = x2 - temp1 * x1
        y = d - temp1 * y1

        x2 = x1
        x1 = x
        d = y1
        y1 = y

    if temp_phi == 1:
        return d + phi
    # no inverse: return None
    return None

print('3^{-1} mod 10 = ', multiplicative_inverse(3, 10))
print('2^{-1} mod 10 = ', multiplicative_inverse(2, 10))
print('25^{-1} mod 119 = ', multiplicative_inverse(25, 119))
```

```
3^{-1} mod 10 =  7
2^{-1} mod 10 =  None
25^{-1} mod 119 =  100
```

In [4]:
```
def is_prime(num):
    if num == 2:
        return True
    if num < 2 or num % 2 == 0:
        return False
    for n in range(3, int(num**0.5)+2, 2):
        if num % n == 0:
            return False
    return True
```

```
for i in [2, 5, 19, 25, 222, 314, 317]:
    print(f'{i}: ' is prime(i))
```

```
2:  True
5:  True
19:  True
25:  False
222:  False
314:  False
317:  True
```

RSA

RSA son unas pocas funciones sencillas:

- Generación de claves
- Cifrado y descifrado son iguales (y simplemente es una potencia)

In [5]:
```
def generate_keypair(p, q):
    if not (is_prime(p) and is_prime(q)):
        raise ValueError('Both numbers must be prime.')
    elif p == q:
        raise ValueError('p and q cannot be equal')
    #n = pq
    n = p * q

    #Phi is the totient of n
    phi = (p - 1) * (q - 1)

    # Choose an integer e such that e and phi(n) are coprime
    e = random.randrange(1, phi)

    # Use Euclid's Algorithm to verify that e and phi(n) are coprime
    g = gcd(e, phi)
    while g != 1:
        e = random.randrange(1, phi)
        g = gcd(e, phi)

    #Use Extended Euclid's Algorithm to generate the private key
    d = multiplicative_inverse(e, phi)

    #Return public and private keypair
    #Public key is (e, n) and private key is (d, n)
    return ((e, n), (d, n))

def encrypt(pk, number):
    # Unpack the key into it's components
    key, n = pk
    return (number ** key) % n

decrypt = encrypt
```

In [6]:
```
pk, sk = generate_keypair(17, 23)
print(f'Publickey (e, n): {pk} Private-key (d, n): {sk}')
```

```
Publickey (e, n): (109, 391) Private-key (d, n): (197, 391)
```

Fíjate: si generamos otro par de claves, aunque usemos los mismos primos, obtendremos unas claves diferentes. Eso es porque el parámetro e se escoge al azar

In [7]:
```
pk, sk = generate_keypair(17, 23)
print(f'Publickey: {pk} Private-key: {sk}')
```

```
Publickey: (317, 391) Private-key: (181, 391)
```

Vamos a intentar cifrar un texto sencillo:

In [8]:
```
print(encrypt(pk, 'hola'))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [8], in <module>
----> 1 print(encrypt(pk, 'hola'))

Input In [5], in encrypt(pk, number)
     28 def encrypt(pk, number):
     29     # Unpack the key into it's components
     30     key, n = pk
```

No podemos: RSA solo puede cifrar enteros. Una posibilidad es codificar el mensaje como un conjunto de enteros

In [9]:
```
print([encrypt(pk, ord(c)) for c in 'hola'])
```

```
[372, 263, 146, 218]
```

¿Qué pasa si intentamos cifrar varias veces lo mismo?

In [10]:
```
print([encrypt(pk, ord(c)) for c in 'aaaa'])
```

```
[218, 218, 218, 218]
```

Pocas veces querremos eso. RSA debe usarse siguiendo recomendaciones como PKCS#1

(semi) Homorfismo

RSA es semihomomórfico con la multiplicación: se pueden hacer cálculos con los números cifrados, aunque no sepas lo que son ni qué resultado tienes. Al descifrar, el resultado es correcto.

Por ejemplo, vamos a multiplicar los mensajes cifrados c1 y c2, que son los cifrados de 5 y 2 respectivamente

In [11]:
```
m1 = 5
c1 = encrypt(pk, m1)
print(f'encrypt(pk, {m1}) = {c1}')
print(f'decrypt(sk, {c1}) = {decrypt(sk, c1)}')
```

```
encrypt(pk, 5) = 241
decrypt(sk, 241) = 5
```

In [12]:
```
m2 = 2
c2 = encrypt(pk, m2)
print(f'encrypt(pk, {m2}) = {c2}')
print(f'decrypt(sk, {c2}) = {decrypt(sk, c2)}')
```

```
encrypt(pk, 2) = 236
decrypt(sk, 236) = 2
```

In [13]:
```
cm = c1 * c2
print(f"c1 = {c1}; c2 = {c2}; cm = {cm}")
```

```
c1 = 241; c2 = 236; cm = 56876
```

Un atacante no sabe cuánto vale c1 ni c2, ni sabe qué valor tiene cm, pero sabe que, sea lo que sea, ha multiplicado c1 y c2 y cuando se descifre el resultado va a ser correcto

In [14]:
```
print(f'decrypt(sk, c1 * c2) = m1 * m2 = {m1} * {m2} = {decrypt(sk, cm)}')
```

```
decrypt(sk, c1 * c2) = m1 * m2 = 5 * 2 = 10
```

Según la utilidad, el semihomorfismo puede ser útil o no:

- Sistemas PET (private enhanced technologies) necesitas calcular sin descifrar. Por ejemplo, voto electrónico
- Pero en general no querremos que un atacante pueda multiplicar una orden de pago por otro número y que el resultado sea válido: recomendaciones PKCS#1

PyCryptoDome

La función de arriba solo sirve para ver cómo funciona RSA a alto nivel. Veamos ahora cómo de grandes son los

In [15]:
```
# Clave de 2048 bits
from Crypto.PublicKey import RSA
key2048 = RSA.generate(2048)
key2048
```

Out[15]:
```
RsaKey(n=2354222026222113561647910027731331363039844722801153552435526124955605032035744
652619998667162331708301507929743539510531840631979828504320261573284460466901277288822 4
996385954876260616814458993540889067041712057508582987594294192257619868896218760956265 3
051867733359386338378229558788617482653213772240123407667041860616482922207540315336794 2
987196325420471904030799019076187642726949254527300329856773564000818980681826514787270 4
346458892248805289560824583055798955352206016125425960454101290377403869198950673992163 5
789045889829692950286540890460226308794495997598050354808447568554254215170411473053794 5
9069078967, e=65537, d=357675644522843351592661247022611141519870209270047239293842159790
392592031675381939016536138903776791096853021292138888956424197371750576267520411269555
654607595172278696026599787794622307350349705711949790693740349666306783019168270016989 7
914921385500307364461328805727712943881649844003043743304591596973830465386208881148575 5
548057406204439815486315327885646116282073497207774222356657071747800972228367159504091 5
309025816780730139459092408538923809495239144155541998156465771180711883828709458853420 4
438553657162844312273870433732544200891184217240912794890424868763888730369292987393449 5
638481497458215859989 13, p=132969006767568351351789567204534270997231423246725187278171 8
292036259599955049714988300562588386916940806957581498209478904262606150001402450686485 4
949888685309847550653566047885313399266905992999073858865990531544308357211077788681474 2
481085253403741045924382262591521375651373139446164148808183013057229537, q=177050433289
114204784186388219569781028116116911156034514976760164391816081755091756385398256301482 8
271911674359236778795606198570795594228877685616020134848432842284705245391380087904551 5
355346354724096039206730959051086966985291071613620639109669818131332376020595392896397 4
513786087742930385326213082090391, u=965621210364236328093180558840249857909869528065165
167003472761637382842696838654028038316283919634624095217458499742472608430232856901749 8
125452683486376216792845205556718216384729509855488727289480358685794605942753064690543 9
363334332271451039668153198501979784625593176628366718788904665468999314762168 18)
```

In [16]:
```
key4096 = RSA.generate(4096)
key4096
```

Out[16]:

```
RsaKey(n=9811142028917447223373588614482178642992005682365223413635573155208949776709842
5943107988079360615619176139668405303060076571996489284415199122347445775333153200575709
7965034946231231682195836226649168665022843504339586399007620448630408167296615201750712
9171328027415977507427753957038997484963513690092115207099049896864942997073997928970307
8523412157033374058488711138593025125280893691222104055949157020311998146914504148000020
2444350144958740506209538276019432506022562990372376608630267756302010928416715418801275
4539772772049142963829429659804371616704441610765929792605316042510705275804272569028837
5310684691361339846205183808021665850244153246322341673257917511116214998629364932229533
0062642303226331475638993032345117798772128607700603697012413220000757704254937119295858
8165130514634012941181652290339470326434119039782428146610481325412286472367441254652546
```

Ejercicios

Hemos visto cómo crear claves con PyCryptoDrome, pero no cómo usarlo para cifrar o descifrar.

Recuerda de las transparencias que no es recomendable utilizar RSA "de forma pura", es decir, sin tener en cuenta muchas consideraciones sobre padding, conversiones, longitudes... que se recogen en PKCS#1 (https://en.wikipedia.org/wiki/PKCS_1). De hecho, PyCryptoDome no nos va a dejar utilizar el cifrado y descifrado directamente.

Observa que la línea siguiente da un error, avisando que uses el módulo `Crypto.Cipher.PKCS1_OAEP`

In [19]:
```python
key2048.encrypt(b'hola', None)
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
Input In [19], in <module>
----> 1 key2048.encrypt(b'hola', None)

File /opt/homebrew/lib/python3.9/site-packages/Crypto/PublicKey/RSA.py:379, in RsaKey.en
crypt(self, plaintext, K)
    378 def encrypt(self, plaintext, K):
--> 379     raise NotImplementedError("Use module Crypto.Cipher.PKCS1_OAEP instead")

NotImplementedError: Use module Crypto.Cipher.PKCS1_OAEP instead
```

Aunque no se debe, vamos a utilizar la función `_encrypt()`, que no está documentada pero la puedes encontrar en el código: https://github.com/Legrandin/pycryptodome/blob/master/lib/Crypto/PublicKey/RSA.py#L147 (https://github.com/Legrandin/pycryptodome/blob/master/lib/Crypto/PublicKey/RSA.py#L147)

In [18]:
```python
c = key2048._encrypt(15)
d = key2048._decrypt(c)
print(f"Cifrado: {c}")
print(f"Descifrado: {d}")
```

```
Cifrado: 1028364403538740680809009303937933617023681140957930551999089997594349283795324
0810693478045731816113599744677338249835608819177402646578683657880000245032029824733824
0533797062252214497490893519922020682370172689824317800825297014974929401128027345705939
7788983013630220236786256704015597073388365166790199533225078844998799577189409125372506
9719180239209062109690441973737726864948350166705667163379780216433298345886352896831909
8039400882731596465441629976784751403301295083359909674684670487309431249353208656131353
9386190637454823403255180688814794949576754826554913151831788133588030714886633658894339
1188471731
Descifrado: 15
```

Usando estas funciones `_encrypt() y _decrypt()` para cifrar cadenas:

1. Una posibilidad es cifrar cada caracter por separado y cifrarlos también por separado, como hemos hecho antes. ¿Cuándo ocupa el cifrado, en bytes?
2. Otra posibilidad es codificar la cadena como un enorme entero, es decir, cada caracter representa un byte de un número entero: `msg = int.from_bytes(b"hola mundo", "big")` ¿Cuánto ocupa el cifrado, en bytes?
3. ¿Puedes probar el método anterior para cifrar una cadena realmente larga, como `msg = int.from_bytes(b"hola mundo" * 1000, "big")` ? ¿Por qué crees que no funciona? ¿Cómo lo harías?

Vamos a hacer las cosas bien: cifra `"hola mundo"` y `"hola mundo" * 1000` usando PKCS1. Encontrarás en ejemplo en la documentación de pyCryptoDome: https://pycryptodome.readthedocs.io/en/latest/src/cipher/oaep.html (https://pycryptodome.readthedocs.io/en/latest/src/cipher/oaep.html)

Cifrado híbrido

En el tema de TLS veremos un cifrado híbrido: ciframos con RSA la clave AES que usamos para cifrar el texto.

1. Bob: Crea par de claves RSA
2. Alice: Crea clave simétrica AES. Cifra la clave AES con la clave pública de Bob. Envía mensaje
3. Alice: cifra "hola mundo" con clave AES. Envía mensaje
4. Bob: descifra clave AES con clave privada. Descifra mensaje de Alice

Entre los ejemplos de RSA precisamente verás algo así: https://pycryptodome.readthedocs.io/en/latest/src/examples.html#encrypt-data-with-rsa (https://pycryptodome.readthedocs.io/en/latest/src/examples.html#encrypt-data-with-rsa)

- ¿Puedes hacer cifrado híbrido del mensaje "hola mundo"?
- ¿Se te ocurre por qué es necesario el cifrado híbrido?

In [ ]: