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): (63, 391) Private-key (d, n): (447, 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: (227, 391) Private-key: (459, 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
---> 31     return (number ** key) % n

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'in
t'
```

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'])
```

```
[246, 15, 386, 385]
```

¿Qué pasa si intentamos cifrar varias veces lo mismo?

In [10]:

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

```
[385, 385, 385, 385]
```

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) = 40
decrypt(sk, 40) = 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) = 59
decrypt(sk, 59) = 2
```

In [13]:

```
cm = c1 * c2
print(f"c1 = {c1}; c2 = {c2}; cm = {cm}")
```

```
c1 = 40; c2 = 59; cm = 2360
```

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 números involucrados en estos cifrados. Ojo: ¡mide cuánto tiempo necesitamos para generar las claves!

In [15]:

```
# Clave de 2048 bits
from Crypto.PublicKey import RSA
key2048 = RSA.generate(2048)
key2048
```

Out[15]:

```
RsaKey(n=21301174681354891827849850037366935626306992880593318438355531
9300695523272508265796258717680445185113412255004713013170156504374412
2605439309145682380900730271962225634702348875122279822989334430167194
2142496786664583764020935491503628505127020246013573716889528773491095
4081839800782126875073916085770947663096141233856894560381737822820310
8499816551039787785443135346575117485781116539596475228271677457803199
6590586233189158250026014767642438821024480460199048170919629779657456
9558145481145038905426778463128250156160734543980678996479139582851296
494636004268115273206554655575516278668776025602610939926125766307, e=
65537, d=5736531768902652903178905476822371032188004804095881315558420
5562907310068836632310155826128906921992030394236184495162530665724123
6541907790525462133101521115552634594149733199791061192280735771432304
5431435225328680156718723920425605581768976888172569589123966322494329
6773949305568480138184091727697993618056725865749957447394122679898174
8077389373464940504936631687508587828005394159140655193012721133380059
1157121439466952531313542612553003891258561161727676384824163982031330
9298061979951611388099311843256829090173014993021563279280564158979856
57774170602147861247122079375424833160471069157791061110344799433, p=1
3958648473160132115658706581939101858376097468204840141253992999943598
5506542083741118176165570429543815541500594991436268367886896563589647
3504432073703241095283406281182204332257012457896495835302409627524291
9213233577390667007386386156965467468766584302885541697537380222949238
0405489060103462222857103611, q=15260198523025393935904339944129104658
9334471078760301244273061600915394956726931516410948928629404245690993
5995182341405068181429748869343728939464901669590823036559196738150504
8657793835089871208529816468859309699743981302636966876571653894168376
6247218368698729399689375359519746100644511903054235823855737, u=13259
6977193673356870655188941157393961190802059640576894598162437298350963
8341649499907464423652173593926089400723547449926441192490622933933561
3492271296168710806757728787919566079045788204046171548033503912611488
2107690982496157297415029487939264635693277691582431088696729751861450
374546061844353993977862)
```

In [16]:

```
key4096 = RSA.generate(4096)
key4096
```

Out[16]:

```
RsaKey(n=7353792484092541419383854579152041018577967484355412519023339611283654372659227155491889456732652561269654770051584525465750317325230078168184106065703746354399028442775596136265958507252220718440459938405385207054058167887024425225403677849732709019513789449078812044437318532454032548344377127031454901533018220510314184872589405268587466065261547641021568521701803091250357822788937885868175727662718436292598974444156835701072362323267915337180954563626183260620700934719055546024376135502373720421865857832192949655102874260358574720293126697737826996052113737629263968946076393468471025765533947880748716512801512212745086797852425069718793144924676411961969141727609105856093896705417437436194756500175473795199337148749496320450165636472143681965600642391164825049527706808073406312527068478605020791518693652946536976459957510122295221432485561834946953668659694107220783468970365328304835681194619938356784642169034291441050928415175767185380960134231864196677578848949164538595325800874516291920073534963236233832742911629857489613050403475156866955886781626832722460108033822282632699003104199730758052724856590257300040468871650796843009934009972820961287013642647019164253712457516978882122197066117778561966523009701 1, e=65537, d=2724584771998917321128050795179521247426658490186855502703285484098815228797855015885321480135708121617851325190465685873577007341388439248680323655864405101001571770652541885999387696932187884322016997946051252042100087027919855671004766829551013535214063483342343632711365334784665018579184173978275969171215983662987226757156715199493515987360548530531840274344299125077369477913996601548867188477383733018580984756180389796230962139744417447545040942317914595461518680232246627718554238585850549955346381844643378657100270707430867022762676898715550699675593935032805484870594533591782950274284946754755328440508060060838256638024281508958262466171560918292111115879927110976464071935016278118324237939120330268500998390440208343784962512854390561302581752841739413021063427913747865529728080365732614462884496244573848834316180401798005122408130277125725730257227158819663069262314975112134099292412259072864220438553437317297878767865992498970744687858259743317856887762938070745906121161813253512221150582627349786679916303571362864225325930126909688030363276862422934853406707069923522637479149332088067915913826261565516121321122266896669969928396032522405109201284039167053458769116310997331362191384162271027761350073 03961, p=2576182754684028774917269996090566332976869656816391312599125272887026354052131725223131044563140533127968144752493941249366512154091748134886972420320035359957956432861135908422395104938489625499823184358602460584196622171057770921303192285115631256196602029934419797520971419554807257426388222470967487870008172330133579999772385470208938485764278715043518112148298545458704782889881348603316130766390235622471044091725320713586744613183387740277048240833541542436086482016954509578681532928403869472141747066160363645502201222720092603032253182249993906488241475448479465609464399881684004593771808806684992432 1689, q=2854530592102535422649633934567960942085377298465413121293933961599339993756565458372362702891375106920123391223850468435211745803679640601571509235258404307555639381623369643473826435835127447251007511185332856151978362652223209228713980338724441077746546004669242891138049561563081779949779302149773161328264044738333075140783175704402669253700833682971339907779134384736275285136958802054476759113229080386866196846427841695695173412779275416219384162159217935931338106323075055469962434168643121856307315847333814894751482946908524457434510251354951791764279458041749378350556476349506898487625997673966 75013
```

```
86199, u=226370571467851746346265311814882282708149101716628867717236
818252259091893642334773007493115957928386900938017709778214417630879
196129457058948100755811191081945919095421218007702391007080215758232
598655948042835046455426248595997588155123452644636581809050909033606
828704983776303536128384420590272515588268547163900526196399146463586
566655634395698191090402954124576516479082795362885331905046732055608
337727277852078483874704761793184398031945300253755110313463901621241
569081326053698293111588608238485956901976954108909978045726619109805
286051106023119858726774443465743996132354677591910704547811541190379
51170)
```

# 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)](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 [17]:

```python
key2048.encrypt(b'hola', None)
```

```
---------------------------------------------------------------------------
-----
NotImplementedError                       Traceback (most recent call
 last)
Input In [17], in <module>
----> 1 key2048.encrypt(b'hola', None)

File /opt/homebrew/lib/python3.9/site-packages/Crypto/PublicKey/RSA.p
y:379, in RsaKey.encrypt(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)](https://github.com/Legrandin/pycryptodome/blob/master/lib/Crypto/PublicKey/RSA.py#L147)

In [18]:

```
c = key2048._encrypt(15)
d = key2048._decrypt(c)
print(f"Cifrado: {c}")
print(f"Descifrado: {d}")
```

```
Cifrado: 1592020934965144610436389636944855369438010549661230821645051
4401455489441197881232392503415459900325275672223618332771276766737972
5308543888123428601162941010265438595605022639829798978331224161839070
3546017957225847285947073012262727106914328143103967962167902165140938
2566675291993493666258090058182147964919202366677700458361892824929495
5212117829673207407134812264562311059156258811069580988010758989965281
6078862890784125966188416578602099498870143841096400645303570018240871
2374893842717599141660213034534990436322076087225969519479181664155495
5351020994797706989576059779922410561123146534842052698639924203 42
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)](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)](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 [ ]:

```

```