In [2]:
```python
import random
```

[Preparación] 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".

No os preocupéis de entender ningún algoritmo de esta sección, simplemente ejecutadlos porque los necesitaremos para el resto de ejercicios.

Máximo común divisor

Algoritmo de Euclides para determinar el máximo común divisor (*gcd* por sus siglas en inglés) de dos enteros a y b

In [3]:
```python
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
```

Inverso multiplicativos de un número en un anillo cíclico $\mathbb{Z}_\phi$

Un inverso multiplicativo en un anillo cíclico en ϕ de un número a es un número a^{-1} tal que $a * a^{-1} \mod \phi = 1$

Recuerda que `mod` es la operación módulo, es decir, divide un número entre otro y quédate solo con el resto.

Por ejemplo: $(3 * 5 \mod 7) = (15 \mod 7) = 1$ porque 15 entre 7 son 2 (que esto no nos importa) y **de resto** 1. Por tanto 5 es el inverso de 3 en $\mathbb{Z}_7$

In [4]:
```python
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 7 = ', multiplicative_inverse(3, 7))
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 7 =  5
3^{-1} mod 10 =  7
2^{-1} mod 10 =  None
25^{-1} mod 119 =  100
```

Test de si un número es primo

Algoritmo de testeo muy ineficiente de si un número es primo, pero que nos servirá en nuestros ejercicios porque usaramos números pequeñose.

In [5]:
```python
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

Bien, ya tenemos todo lo necesario para poder hacer los ejercicios.

RSA es un algoritmo de cifrado asimétrico. Es decir, tiene dos claves: una para cifrar y otra diferente para descifrar. Puede usarse tanto para cifrar una información como para firmar digitalmente un documento.

RSA está compuestos por dos funciones sencillas:

- Una función que genera el par de claves necesario. Da como resultado dos claves, una se podrá hacer pública y la otra tiene que permanecer siempre privada.
- Una función que cifra, que es la misma función que para descifrar pero usando la otra clave.

Aquí vemos los dos algoritmos. Fíjate qué sencilla es la función para cifrar o descifrar.

In [6]:
```python
import random

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

# The decrypt function is exactly the same than the encrypt function
decrypt = encrypt
```

En los ejemplos usaremos como parámetros de configuración 17 y 23, dos números primos que simplemente sirven para configurar el algoritmo inicialmente. Cuanto más grandes sean, mayor será el tamaño en bits de la clave. Como estamos usando algoritmos poco eficientes para aprender, no uses números demasiado altos.

In [7]:
```python
pk, sk = generate_keypair(17, 23)
print(f'Clave pública pk=(e, n): {pk}')
print(f'Clave privada o secreta sk=(d, n): {sk}')
```
```
Clave pública pk=(e, n): (271, 391)
Clave privada o secreta sk=(d, n): (239, 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 [8]:
```python
pk, sk = generate_keypair(17, 23)
print(f'Clave pública pk=(e, n): {pk}')
print(f'Clave privada o secreta sk=(d, n): {sk}')
```
```
Clave pública pk=(e, n): (201, 391)
Clave privada o secreta sk=(d, n): (345, 391)
```

Vamos a intentar cifrar un texto sencillo:

In [9]:
```python
print(encrypt(pk, 'hola'))
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 print(encrypt(pk, 'hola'))

Cell In[6], line 33, in encrypt(pk, number)
     30 def encrypt(pk, number):
     31     # Unpack the key into it's components
     32     key, n = pk
---> 33     return (number ** key) % n

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

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

In [10]:
```python
msg = [ord(c) for c in 'hola']
print(f'mensaje = {msg}')

c = [encrypt(pk, m) for m in msg]
print(f'cifrado = {c}')
```
```
mensaje = [104, 111, 108, 97]
cifrado = [325, 281, 232, 56]
```

¿Qué pasa si intentamos cifrar varias veces lo mismo?

In [11]:
```python
print([encrypt(pk, ord(c)) for c in 'aaaa'])
```
```
[56, 56, 56, 56]
```

El texto cifrado es siempre igual. Pocas veces querremos eso. RSA debe usarse siguiendo recomendaciones como PKCS#1. Lo veremos un poco más abajo.

(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. Más detalles: https://ciberseguridad.com/guias/prevencion-proteccion/criptografia/cifrado-homomorfico/

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

In [12]:
```python
m1 = 5
c1 = encrypt(pk, m1)
print(f'encrypt(pk, {m1}) = {c1}')
print(f'decrypt(sk, {c1}) = {decrypt(sk, c1)}')
```
```
encrypt(pk, 5) = 148
decrypt(sk, 148) = 5
```

In [13]:
```python
m2 = 2
c2 = encrypt(pk, m2)
print(f'encrypt(pk, {m2}) = {c2}')
print(f'decrypt(sk, {c2}) = {decrypt(sk, c2)}')
```
```
encrypt(pk, 2) = 376
decrypt(sk, 376) = 2
```

In [14]:
```python
cm = c1 * c2
print(f"c1 = {c1}; c2 = {c2}; cm = {cm}")
```
```
c1 = 148; c2 = 376; cm = 55648
```

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 [15]:
```python
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 es para jugar y solo sirve para ver cómo funciona RSA a alto nivel.

Vamos a usar PyCryptoDome, que incluye una librería RSA real.

Mide cuánto tiempo necesitamos para generar las claves ya más útiles de 2048 y 4096 bits. Prueba a generar claves mayores, de 16384 bits, por ejemplo, que es similar en seguridad al cifrado AES de 256 bits.

In [16]:
```python
!python3 -m pip install pycryptodome

# Clave de 2048 bits
import timeit

start2048 = timeit.default_timer()

from Crypto.PublicKey import RSA
key2048 = RSA.generate(2048)
key2048

stop2048 = timeit.default_timer()

print('Tiempo para generar una clave de 2048 bits: ', stop2048 - start2048)
```
```
Requirement already satisfied: pycryptodome in /home/gattes/.cache/pypoetry/virtualenvs/crypto-83WUx_V0-py3.10/lib/python3.10/site-packages (3.17)

[notice] A new release of pip is available: 23.0 -> 23.1
[notice] To update, run: pip install --upgrade pip
Tiempo para generar una clave de 2048 bits:  0.6157128210179508
```

In [17]:
```python
# Clave de 4096 bits
start4096 = timeit.default_timer()

key4096 = RSA.generate(4096)

stop4096 = timeit.default_timer()

print('Tiempo para generar una clave de 4096 bits: ', stop4096 - start4096)
```
```
Tiempo para generar una clave de 4096 bits:  2.715338264941238
```

In [18]:
```python
start = timeit.default_timer()
key = RSA.generate(8192)
stop = timeit.default_timer()
print(f'Tiempo para generar una clave de 8192 bits: ', stop - start)
```
```
Tiempo para generar una clave de 8192 bits:  20.330803863005713
```

In [23]:
```python
start = timeit.default_timer()
key = RSA.generate(16384)
stop = timeit.default_timer()
print(f'Tiempo para generar una clave de 16384 bits: ', stop - start)
```
```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[23], line 2
      1 start = timeit.default_timer()
----> 2 key = RSA.generate(16384)
      3 stop = timeit.default_timer()
      4 print(f'Tiempo para generar una clave de 16384 bits: ', stop - start)

File ~/.cache/pypoetry/virtualenvs/crypto-83WUx_V0-py3.10/lib/python3.10/site-packages/Crypto/PublicKey/RSA.py:493, in generate(bits, randfunc, e)
    488 def filter_q(candidate):
    489     return (candidate > min_q and
    490             (candidate - 1).gcd(e) == 1 and
    491             abs(candidate - p) > min_distance)
--> 493 q = generate_probable_prime(exact_bits=size_q,
    494                             randfunc=randfunc,
    495                             prime_filter=filter_q)
    497 n = p * q
    498 lcm = (p - 1).lcm(q - 1)

File ~/.cache/pypoetry/virtualenvs/crypto-83WUx_V0-py3.10/lib/python3.10/site-packages/Crypto/Math/Primality.py:334, in generate_probable_prime(**kwargs)
    332     if not prime_filter(candidate):
    333         continue
--> 334     result = test_probable_prime(candidate, randfunc)
    335 return candidate

File ~/.cache/pypoetry/virtualenvs/crypto-83WUx_V0-py3.10/lib/python3.10/site-packages/Crypto/Math/Primality.py:272, in test_probable_prime(candidate, randfunc)
    269 except IndexError:
    270     mr_iterations = 1
--> 272 if miller_rabin_test(candidate, mr_iterations,
    273                      randfunc=randfunc) == COMPOSITE:
    274     return COMPOSITE
    275 if lucas_test(candidate) == COMPOSITE:

File ~/.cache/pypoetry/virtualenvs/crypto-83WUx_V0-py3.10/lib/python3.10/site-packages/Crypto/Math/Primality.py:101, in miller_rabin_test(candidate, iterations, randfunc)
     98 assert(2 <= base <= candidate - 2)
    100 # Step 4.3-4.4
--> 101 z = pow(base, m, candidate)
    102 if z in (one, minus_one):
    103     continue

File ~/.cache/pypoetry/virtualenvs/crypto-83WUx_V0-py3.10/lib/python3.10/site-packages/Crypto/Math/_IntegerGMP.py:450, in IntegerGMP.__pow__(self, exponent, modulus)
    448 def __pow__(self, exponent, modulus=None):
    449     result = IntegerGMP(self)
--> 450     return result.inplace_pow(exponent, modulus)

File ~/.cache/pypoetry/virtualenvs/crypto-83WUx_V0-py3.10/lib/python3.10/site-packages/Crypto/Math/_IntegerGMP.py:442, in IntegerGMP.inplace_pow(self, exponent, modulus)
    440     elif exponent.is_negative():
    441         raise ValueError("Exponent must not be negative")
--> 442     _gmp.mpz_powm(self._mpz_p,
    443                   self._mpz_p,
    444                   exponent._mpz_p,
    445                   modulus._mpz_p)
    446 return self

KeyboardInterrupt: 
```

Nota del alumno: La ejecucion nunca termino luego de 2 minutos. Por tanto se paro la misma

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. 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)
Cell In[19], line 1
----> 1 key2048.encrypt(b'hola', None)

File ~/.cache/pypoetry/virtualenvs/crypto-83WUx_V0-py3.10/lib/python3.10/site-packages/Crypto/PublicKey/RSA.py:413, in RsaKey.encrypt(self, plaintext, K)
    412 def encrypt(self, plaintext, K):
--> 413     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

In [20]:
```python
c = key2048._encrypt(15)
d = key2048._decrypt(c)
print(f"Cifrado: {c}")
print(f"Descifrado: {d}")
```
```
Cifrado: 2854068270792073198806104223823687997633539987531954537795237187510097475925075042208153296314602677661242420576521514660006410815080668878153371248749460676324469221646325141915918193696459846537750402027785420775285386601515833952842085235890201135404822530504089996333131022853385794745790
3427972143412195809462737497539704032399471779954567194569209181874515406040215958861037579681053253956588415553560006097607683473904041492661118860214042212219439203354301394363502905409366935743994916782187205362601686244957652835708707177851217296645570474039788674364719236241182495865778018903274991396484067918078977 41
Descifrado: 15
```

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

1. Una posibilidad es cifrar cada caracter por separado y descifrarlos también por separado, como hemos hecho antes. ¿Cuánto ocupa el cifrado, en bytes?

Respuesta: El cifrado deberia ocupar no mas de 256 bytes, dado que 2048 bits / 8 bits = 256 bytes. Tambien habria que restarle 11 bytes de Padding por utilizar PKCS#1. Entonces 256 - 11 = 244 bytes por cifrado

In [21]:
```python
def byte_length(i: int):
    """
    Calcula los bytes ocupados por un entero
    """
    return (i.bit_length() + 7) // 8

# Utilizamos bit_length en vez de getsizeof para verificar cuanto realmente ocupan los enteros
# Ver: https://python-reference.readthedocs.io/en/latest/docs/ints/bit_length.html
encrypted = [byte_length(key2048._encrypt(ord(c))) for c in 'hola']
print(f"Tamaño del texto `hola` encriptado por partes: {encrypted}")
```
```
Tamaño del texto `hola` encriptado por partes: [256, 256, 256, 256]
```

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?

Respuesta: Efectivamente, la misma cantidad (256 bytes) dado que nuestra key es de 2048 y RSA encripta los datos hasta el tamaño maximo de nuestro key size.

In [22]:
```python
msg = int.from_bytes(b"hola mundo", "big")
encrypted = byte_length(key2048._encrypt(msg))
print(f"Tamaño del texto encriptado: {encrypted}")
```
```
Tamaño del texto encriptado: 256
```

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?

In [23]:
```python
# Demostracion
msg = int.from_bytes(b"hola mundo" * 1000, "big")
print(f"Tamaño del texto desencriptado: {byte_length(msg)}")
```
```
Tamaño del texto desencriptado: 10000
```

In [24]:
```python
encrypted = key2048._encrypt(msg)
print(f"Tamaño del texto encriptado: {byte_length(encrypted)}")
```
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[24], line 1
----> 1 encrypted = key2048._encrypt(msg)
      2 print(f"Tamaño del texto encriptado: {byte_length(encrypted)}")

File ~/.cache/pypoetry/virtualenvs/crypto-83WUx_V0-py3.10/lib/python3.10/site-packages/Crypto/PublicKey/RSA.py:180, in RsaKey._encrypt(self, plaintext)
    178 def _encrypt(self, plaintext):
    179     if not 0 <= plaintext < self._n:
--> 180         raise ValueError("Plaintext too large")
    181     return int(pow(Integer(plaintext), self._e, self._n))

ValueError: Plaintext too large
```

- El texto no puede ser encriptado directamente utilizando RSA dado que no fue diseñado para esto. Con RSA solo podemos encriptar data tan grande como lo es nuestra key (2048 bits o 256 bytes, en nuestro caso), el texto aqui es de 10000 bytes de largo.
- Una opcion **no recomendable** podria ser dividir el texto en bloques de <256 bytes (es menor, pero tomandolo como convencion), encriptarlo, enviarlo y desencriptarlo. Finalmente concatenar los datos de la misma manera en que llegaron. Esta opcion no es recomendada dado que habria que saber el tamaño en que se ha dividido y sincronizar el orden de envio.
- Para encriptar mas datos de nuestro limite, podriamos utilizar un **cifrado mixto**:
  - Generar una random key `K` de 256 bits
  - Encriptar nuestros datos (el texto plano) con esa `K`, utilizando AES (CBC u OFB) o ChaCha20
  - Encriptar `K` con RSA
  - Enviar `K` y nuestros datos encriptados previamente al receptor

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

In [25]:
```python
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

msg = b"hola mundo"

# Generamos la key inicial en formato privado y publico
private_key = RSA.generate(2048)
public_key = private_key.public_key()

cipher = PKCS1_OAEP.new(public_key)
ciphertext = cipher.encrypt(msg)
print(f"Mensaje encriptado: {ciphertext}")

priv = PKCS1_OAEP.new(private_key)
message = priv.decrypt(ciphertext)
print(f"Mensaje desencriptado: {message}")
```
```
Mensaje encriptado: b'0u\x13\xd5B\xe1\xf8\xe5\xb4\xc3an\x99\xe1D\xb8\xe1\xb4\xb5%\xb4\xa7\xd7\xec(pvqh:\xb7 \xab\xe3\xfe\xd5\xd5\xae#/\xfe?\xaf\xe9\xb1\xdb\xbe\xadX\xfcKi\x1dt\xda\xd3\xde.\xa6\x1b\x7f\xee\xe7\xf9c\x8f\xef\x81\xab\xf1y\xdd\xfe|\xc8\x07\nF\x92?\x97t\xf06j\x19\x0f\x1c~\xbf\xd1!h\x07\xc8
\xab\x927\x03\x0c6e\xed$\xdf`;W,9\xbb\xb2\xf9\x85\xe7tTb\xe9\xb8kD{`\xc0\xe3u\xb6i\xa06\xe5Q\x1b\xd7\x8f\xed\xc6\xac\x96\x1bkhF\x96\xa568\xfd\x0erx\x1f\xa7\x81qY\xbc\x1d`\xc2\xd1E+\xfd\x08\xebH\x11a\xc9\xb8\xd7\xc7\x91fXy\x1aRx\xce\xa5\xec\xfa\xf1\x12\xbe\x91i\xb4\xba\x16\t|\x9e\xb4\xfa\xb2)\x9e\xa0
\xeaQ\x1e\x13\xd3\n\x1f\x98\x95\xb5oD\x8a!\x82\xd0H\x05\xfe\xf2ciOr\xee\x16\xe6[b&\xbaU\xcd\xd6,\xd6g\x18")o$\xe7\x91t=\xee\x11Oq\x11\xef\xc7K'
Mensaje desencriptado: b'hola mundo'
```

In [26]:
```python
# Generamos la key inicial en formato privado y publico
private_key = RSA.generate(2048)
public_key = private_key.public_key()

# Mensaje original
orig_message = b"hola mundo" * 1000

# Cortamos el mensaje en partes y encriptamos cada una
cipher = PKCS1_OAEP.new(public_key)
chunked_encrypted_msg = [cipher.encrypt(bytes([msg])) for msg in orig_message]

# Desencriptar usando la private key, luego unimos cada una de las partes
priv = PKCS1_OAEP.new(private_key)
decrypted = bytes([int.from_bytes(priv.decrypt(chunk), 'big') for chunk in chunked_encrypted_msg])

print(f"Mensaje desencriptado: {decrypted.decode('utf-8')}")
```
```
Mensaje desencriptado: hola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundohola mundo
```

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

- ¿Puedes hacer cifrado híbrido del mensaje "hola mundo"?

In [4]:
```python
from Crypto.PublicKey import RSA
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES, PKCS1_OAEP


msg = b"hola mundo"


# Generamos la key inicial en formato privado y publico
private_key = RSA.generate(2048)
public_key = private_key.public_key()

#### ENVIO ####

# Generamos la session key (simetrica)
session_key = get_random_bytes(16)

# Encriptamos la clave simetrica `session_key` con la RSA publica
cipher_rsa = PKCS1_OAEP.new(public_key)
enc_session_key = cipher_rsa.encrypt(session_key)

# Encriptamos los datos a enviar ("hola mundo") con la clave simetrica en AES CBC
# Y un IV generado que enviaremos en texto plano
iv = get_random_bytes(16)
cipher = AES.new(session_key, AES.MODE_CBC, iv=iv)
ciphertext = cipher.encrypt(pad(msg, AES.block_size))

# Armamos una tupla con los datos a enviar: la clave simetrica encriptada, el IV (nonce) en plano y el texto cifrado
data = (enc_session_key, iv, ciphertext)

print(f"Texto encriptado: {ciphertext}")
```
```
Texto encriptado: b'0\x8fm\xc8\x04\xc2<d\xff=jZ\xcb\xc0hb'
```

In [5]:
```python
#### RECEPTOR ####

# Se supone que recibimos los datos por un canal inseguro
enc_session_key, iv, ciphertext = data

# Desencriptamos la clave simetrica de AES con nuestra clave privada definida en el paso anterior
cipher_rsa = PKCS1_OAEP.new(private_key)
decrypted_session_key = cipher_rsa.decrypt(enc_session_key)

# Desencriptamos los datos con AES CBC
decipher = AES.new(session_key, AES.MODE_CBC, iv=iv)
data = decipher.decrypt(ciphertext)
print(f"Mensaje desencriptado: {data.decode('utf-8')}")
```
```
Mensaje desencriptado: hola mundo
```

- ¿Se te ocurre por qué es necesario el cifrado híbrido?

La razon por la cual es necesario es lo expuesto antes, RSA no fue diseñado para encriptar grandes porciones de datos. De hacerlo, habria que crear claves realmente grandes. En cambio, ChaCha20, AES y la criptografia simetrica es ideal para esto, haciendo uso de pequeñas claves para encriptar grandes cantidades de datos.

Pero dado que es necesario compartir las claves para tener encriptacion simetrica, y nuevamente, que las mismas son pequeñas, la criptografia asimetrica es el complemento ideal para este caso de uso.