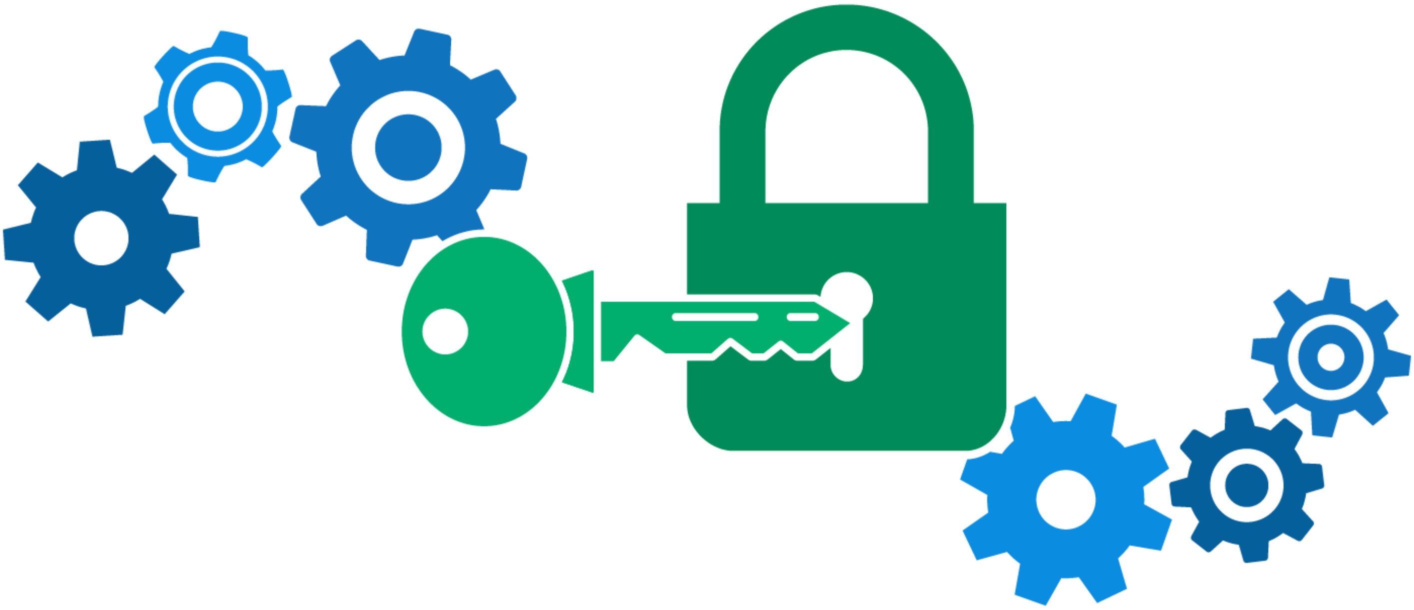
PROGRAM: The RSA Cryptosystem



Rudi Clotet Muntada

Mohammed El Marnissi Kouliss

January 13th 2025

Security and privacy in information encoding

Industrial Electronics and Automation Engineering

INDEX

1. Introduction

In this project, the aim is to implement a version of the RSA cryptosystem using the Python programming language. This is done by following the steps on the project description.

In some cases, we have implemented doctests to test its functionality. In other cases where it was not possible, we have provided examples to ensure it works correctly.

Finally, all the subroutines have been implemented in one final script.

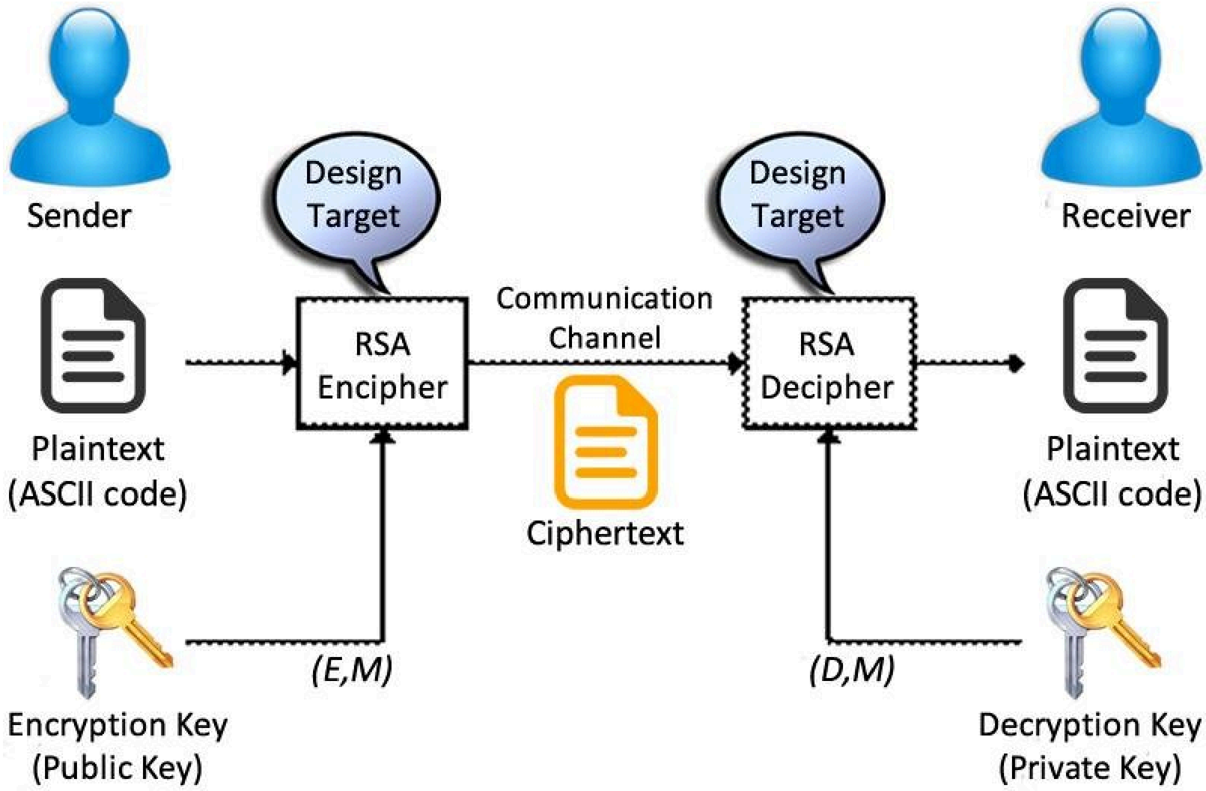
2. Activities

2.1. The RSA Cryptosystem

The RSA Cryptosystem is a public key asymmetric cryptosystem used in modern cryptography. It was developed in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman, from whom it takes its name.

Imagine that Alice wants to send an encrypted message to Bob. In this case, only Bob's keys are involved. Specifically, Alice will use Bob's public key to encrypt the message and then transmit the encrypted message over a public channel. With the RSA system, only Bob, who holds the corresponding private key, will be able to decrypt the message.

It can be seen on the following diagram:



Below are the steps of RSA explained (Screenshot taken from the course notes):

Algorithm 1.8 (RSA). *1. Bob choses two large primes p and q and computes $n = pq$ and $\phi(n) = (p-1)(q-1)$.*

2. *Bob choses an exponent e with $2 < e < \phi(n)$ and coprime with $\phi(n)$. e can be hardcoded and it does not need to be large.*

3. *Bob computes the inverse d of e modulo $\phi(n)$ using the extended Euclid's algorithm. That is, $d \cdot e \equiv 1 \mod \phi(n)$*

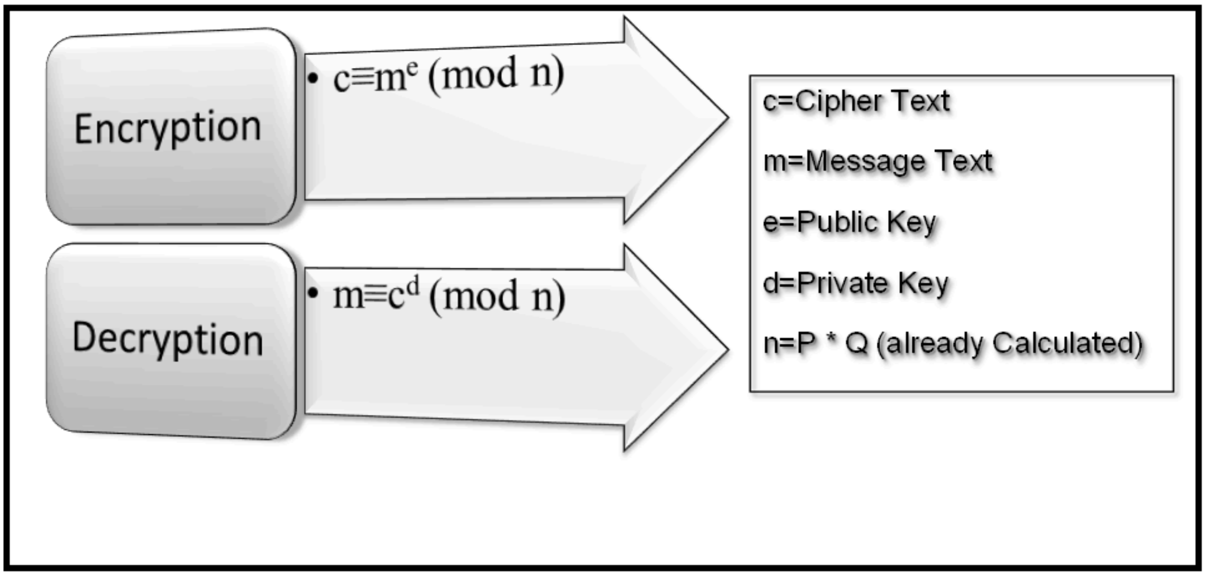
4. *Bob's public key is the pair (n, e) and his private key is d. p, q and $\phi(n)$ must be deleted or otherwise kept secret.*

5. *Alice takes the plain text message $m \in \mathbb{Z}/n\mathbb{Z}$ and encrypts it by computing $c \equiv m^e \mod n$ using n and e from Bob's public key.*

6. *c is sent through a public channel.*

7. *Bob receives c and computes $c^d = (m^e)^d = m^{1+k\phi(n)} \equiv m \mod n$ which is the plain text again thanks to the Euler theorem.*

RSA is based on the difficulty of factoring large numbers into prime factors. Its security relies on the practical impossibility of efficiently decomposing a composite number into its prime factors. This principle is used to ensure the confidentiality, authenticity, and integrity of data.



Despite its strength, RSA's security is contingent on careful parameter selection, such as using sufficiently large primes, ensuring these primes are not too close in value, and watching out for choosing weak private keys, as these factors could otherwise expose vulnerabilities to sophisticated factoring attacks.

But why must they be large and not close?

if p and q are close to each other then an attacker can use the identity $(x + y)(x − y) = x^2 − x^2$ and test x from $\lfloor \sqrt{n} \rfloor$ downwards until $n − x^2$ is a square.

For example: if we use p=53,q=59 (very close to each other, then:
n=p*q= 3127
$x = \sqrt{3127} = 55.93 \approx 56$
We compute $y^2 = x^2 - n$, where $x^2 = 56^2 = 3136$, $y^2 = 3136 - 3127 = 9$
Is y^2 a perfect square? YES → $y^2 = \sqrt{9} = 3$

Now we can recover p $p = x + y = 56 + 3 = 59$

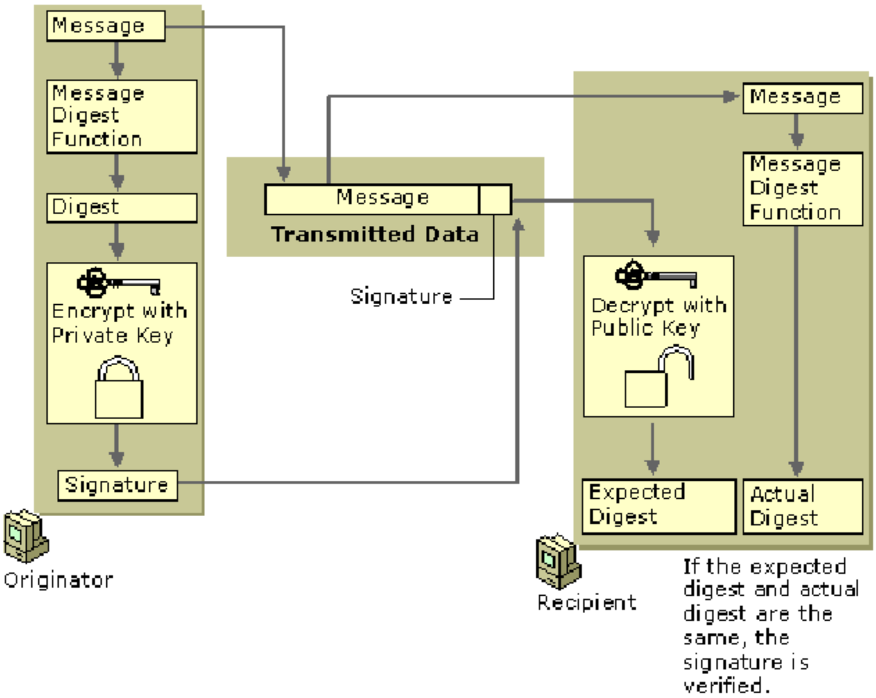
Another poor choice occurs when either p−1 or q−1 consist only of small prime factors. In such cases, n can be efficiently factored using Pollard's p−1 algorithm.

So to recapitulate, in order to have a strong RSA encryption we will have to take into account the following:

Avoid Small Prime Factors: Ensure p − 1 and q − 1 do not have only small factors.
Size of Primes: p and q should be large and separated from each other, to prevent Fermat's factorization attack.
A large enough d: There are also known attacks if the private exponent d is not large enough (d <$n^{1/4}/3$).

In practice the most efficient algorithm for factoring a general number n n is the General Number Field Sieve (GNFS), which operates in subexponential time.

There are other applications, for example, RSA Digital Key Signatures:



2.2. Extended Euclidean Algorithm

This subroutine can be found in the *Extended_Euclidean_Algorithm.py* file.

```
def natural(x):
    """
    Retorna True si un nombre és natural i False si no és natural.

    >>> natural(8)
    True
```

```
    >>> natural(0)
    False

    >>> natural(-8)
    False

    >>> natural(8.5)
    False

    >>> natural(-8.5)
    False
    """
    if isinstance(x, int)==True and x>0: #isinstance ha sigut idea del chat. La primera
    part és per saber si és enter. Segons el chat és la manera més òptima. Després mirem
    si és un natural
        return True
    else:
        return False


def euclides(n,a):
    """
    Retorna x,y (coeficients de la identitat de Bézout que fan que n*x+a*y=MCD(n,a)) i
    l'MCD.
    Si n o a no són naturals o bé n=a, retorna False.

    >>> euclides(15,15)
    False

    >>> euclides(15,8)
    (-1, 2, 1)

    >>> euclides(8,15)
    (2, -1, 1)

    >>> euclides(4023,325)
    (37, -458, 1)

    >>> euclides(325,4023)
    (-458, 37, 1)

    >>> euclides(8,1)
```

```
    (0, 1, 1)

    >>> euclides(1,8)
    (1, 0, 1)

    >>> euclides(8,15.5)
    False
    """
    if natural(n)==True and natural(a)==True: # Comprovem n i a són naturals
        if n<a: # Canviem l'ordre si n < a
            nbona=a
            abona=n
        elif n>a: # Deixem igual si n > a
            nbona=n
            abona=a
        elif n==a:
            return False
    else:
        return False

    iteracions=0; # Iniciem les iteracions a 0
    valorsq=[]; # Creem la llista buida on es guardaran els valors dels quocients

    while abona!=0: # Calculem tots els r i q
        r=nbona%abona
        q=nbona//abona
        nbona=abona
        abona=r

        iteracions=iteracions+1
        valorsq.append(-q)

    del valorsq[-1] #Elimino l'últim coeficient, no interessa
    mcd=nbona

    if iteracions==1: # Tenim en compte si introduim euclides(x, 1)
        if n<a:
            x=1
            y=0

        elif n>a:
            x=0
            y=1
```

```
    elif iteracions>1:
        if n<a:
            nbona2=a
            abona2=n
        elif n>a:
            nbona2=n
            abona2=a

        x=1
        y=valorsq[-1]
        index=-2

        while (nbona2 * x + abona2 * y) != mcd and abs(index) <= len(valorsq):
 #abs(index) <= len(valorsq) ha sigut idea del chat perquè em petava
            x,y=y,x+y*valorsq[index]; # Calculem x i y
            index=index-1
        if n<a:
            x,y=y,x

    return x,y,mcd
```

Now we are going to explain what does each part of the code:

- **1st function: natural(x)**

```
def natural(x):
    if isinstance(x, int)==True and x>0: #isinstance ha sigut idea del chat. La primera
     part és per saber si és enter. Segons el chat és la manera més òptima. Després mirem
     si és un natural
        return True
    else:
        return False
```

In this function basically we check if a number is natural (as a requirement of a project statement). It is done by a function called isinstance(a, type) in Python that checks if a is part of the given type. Therefore, we check if x is integer and if it's greater than 0 (then x will be a natural number).

- **2nd function: euclides(n,a)**

```
def euclides(n,a):
    if natural(n)==True and natural(a)==True: # Comprovem n i a són naturals
        if n<a: # Canviem l'ordre si n < a
            nbona=a
```

```
            abona=n
        elif n>a: # Deixem igual si n > a
            nbona=n
            abona=a
        elif n==a:
            return False
    else:
        return False

    iteracions=0; # Iniciem les iteracions a 0
    valorsq=[]; # Creem la llista buida on es guardaran els valors dels quocients

    while abona!=0: # Calculem tots els r i q
        r=nbona%abona
        q=nbona//abona
        nbona=abona
        abona=r

        iteracions=iteracions+1
        valorsq.append(-q)

    del valorsq[-1] #Elimino l'últim coeficient, no interessa
    mcd=nbona

    if iteracions==1: # Tenim en compte si introduïm euclides(x, 1)
        if n<a:
            x=1
            y=0

        elif n>a:
            x=0
            y=1

    elif iteracions>1:
        if n<a:
            nbona2=a
            abona2=n
        elif n>a:
            nbona2=n
            abona2=a

        x=1
        y=valorsq[-1]
```

```
    index=-2

    while (nbona2 * x + abona2 * y) != mcd and abs(index) <= len(valorsq):
#abs(index) <= len(valorsq) ha sigut idea del chat perquè em petava
        x,y=y,x+y*valorsq[index]; # Calculem x i y
        index=index-1
    if n<a:
        x,y=y,x

return x,y,mcd
```

This function returns x, y (coefficients of Bézout's identity such that n*x* + a*y* = GCD(n, a)) and the GCD between n and a. If n or a are not natural numbers or if n = a, return False.

The function is divided into the following parts:

```
if natural(n)==True and natural(a)==True: # Comprovem n i a són naturals
    if n<a: # Canviem l'ordre si n < a
        nbona=a
        abona=n
    elif n>a: # Deixem igual si n > a
        nbona=n
        abona=a
    elif n==a:
        return False
else:
    return False
```

First we check if n and a are natural. Then, in order to avoid problems with the code, we impose that no matter the input order of n and a, the calculations are always done using the larger number as nbona.

```
iteracions=0; # Iniciem les iteracions a 0
valorsq=[]; # Creem la llista buida on es guardaran els valors dels quocients
```

We initialize the variable "iteracions" to count the number of iterations (starting at 0), and we initialize the list where we are going to add all the computed quotients for the euclidean division.

```
while abona!=0: # Calculem tots els r i q
```

```
        r=nbona%abona
        q=nbona//abona
        nbona=abona
        abona=r

        iteracions=iteracions+1
        valorsq.append(-q)
```

In this part, we compute the gcd(nbona, abona) using the procedure for the euclidean division. For each loop, we have:

- r is the remainder for the division between nbona and abona
- q is the quotient for the division between nbona and abona
- In the next loop, nbona must be the previous abona and abona must be the previous remainder
- Then we add 1 to the number of iterations
- The list valorsq is updated by adding the computed quotient. Note that we need to change the sign (the same as when in the paper we isolate it)

Let's take for example nbona = 4023 and abona = 325:

Iteration	nbona	abona	q	r
0	4023	325	-	-
1	4023	325	12	123
2	325	123	2	79
3	123	79	1	44
4	79	44	1	35
5	44	35	1	9
6	35	9	3	8
7	9	8	1	1
8	8	1	1	0
9	1	0	-	-

```
    del valorsq[-1] #Elimino l'últim coeficient, no interessa
    mcd=nbona
```

Finally, we eliminate the last quotient (it is not our interest). Note that gcd(nbona, abona) is the last nbona that we have on the loop (1 in the example made above).

Once we have computed all the quotients, we can move to the next part of the code:

```
if iteracions==1: # Tenim en compte si introduïm euclides(x, 1)
      if n<a:
          x=1
          y=0

      elif n>a:
          x=0
          y=1
```

We considered if one of the given input numbers is 1. In this case, the loop will have 1 iteration. Let's see it taking an example for nbona = 7 and abona = 1:

Iteration	nbona	abona	q	r
0	7	1	-	-
1	7	1	7	0
2	1	0	-	-

This means that the x and y such that n*x* + *a*y = GCD(nbona, abona) will be always the same, x = 0 and y = 1, 7*0+1*1 =GCD(7,1)=1.

As you can see, we are taking into account which of the input numbers, n and a, is the larger. This is done to ensure the correct order for x and y as output.

Now, finally let's take a look at the part where we compute x and y for the other cases:

```
 elif iteracions>1:
      if n<a:
          nbona2=a
          abona2=n
      elif n>a:
          nbona2=n
          abona2=a
```

First we make sure the order of the inputs (nbona2 will be always the greatest number).

```
    x=1
    y=valorsq[-1]
    index=-2

    while (nbona2 * x + abona2 * y) != mcd and abs(index) <= len(valorsq):
#abs(index) <= len(valorsq) ha sigut idea del chat perquè em petava
        x,y=y,x+y*valorsq[index]; # Calculem x i y
        index=index-1
```

In this part, we compute the x and y explained previously using the list of the quotients of the euclidean algorithm.
Note that the initial values for x and y will always be x = 1 and y = the last number in the list of quotients.
Since we initially use the last number in the list of quotients, we then need to proceed with the second-to-last number.

- x updates to the previous y value
- y updates to the previous value of x + the quotient * previous y value. We discovered this by calculating some examples. Then we realized that we could extrapolate it to all cases without making any errors.
- Index is updated to take the next quotient in the next loop

The loop ends when nbona2 * x + abona2 * y = mcd(nbona, abona).

Let's take for example nbona2 = 4023, abona2 = 325 and valorsq = [-1, -3, -1, -1, -1, -2, -12]:

Iteration	x	y	índex
0	1	-1	–2
1	-1	4	-3
2	4	-5	-4
3	-5	9	-5
4	9	-14	-6

5	-14	37	-7
6	37	-458	-8

As you can see, we stop at the 6th iteration, since 4023*37 + 325*(-458) = gcd(4023,325) = 1.

```
        if n<a:
            x,y=y,x

    return x,y,mcd
```

Finally, we make sure the correct order for x and y depending on whether n < a. Then, we return x, y and mcd(n,a), as it will be useful for the next subroutines.

In order to check it's functionality, we made this doctests:

```
    """
    Retorna True si un nombre és natural i False si no és natural.

    >>> natural(8)
    True

    >>> natural(0)
    False

    >>> natural(-8)
    False

    >>> natural(8.5)
    False

    >>> natural(-8.5)
    False
    """
```

And for the other function:

```
    """
     Retorna x,y (coeficients de la identitat de Bézout que fan que n*x+a*y=MCD(n,a)) i
l'MCD.
    Si n o a no són naturals o bé n=a, retorna False.

```

```
    >>> euclides(15,15)
    False

    >>> euclides(15,8)
    (-1, 2, 1)

    >>> euclides(8,15)
    (2, -1, 1)

    >>> euclides(4023,325)
    (37, -458, 1)

    >>> euclides(325,4023)
    (-458, 37, 1)

    >>> euclides(8,1)
    (0, 1, 1)

    >>> euclides(1,8)
    (1, 0, 1)

    >>> euclides(8,15.5)
    False
    """
```

As you can see, all the items passed the tests:

```
2 items passed all tests:
   8 tests in Extended_Euclidean_Algorithm.euclides
   5 tests in Extended_Euclidean_Algorithm.natural
13 tests in 3 items.
13 passed.
Test passed.
```

2.3. Fast Modular Exponentiation

This subroutine can be found in the *Fast_Modular_Exponentiation.py* file.

```
from Extended_Euclidean_Algorithm import natural


def fme(M,e,n):
    """
```

```
    Retorna el resultat del càlcul de la potència M^e mod n. Si M, n o e no són naturals,
retorna False.

    >>> fme(11,8,27)
    22

    >>> fme(3,13,7)
    3

    >>> fme(3,0,7)
    False

    >>> fme(3,-4,7)
    False

    >>> fme(3,4.5,7)
    False
    """

    if natural(M)==True and natural(e)==True and natural(n)==True:

        a=M%n # 1r pas: Reduir la base a mòdul n

        q=e//2 # 2n pas: Convertim l'exponent a binari
        breves=[]
        r=e%2
        breves.append(r)

        while q != 0:
            r=q%2
            breves.append(r)
            q=q//2

        p=1
        s=a

        for element in breves: # 3r pas: A partir de p i s calculem el resultat
            if element == 1:
                p=(p*s)%n
            s=(s**2)%n

        return p
    else:
```

```
        return False
```

Now we are going to explain what does each part of the code:

```
from Extended_Euclidean_Algorithm import natural
```

First of all, we need the function natural(x) from the previous file.

- **1st function: fme(M,e,n)**

```
    if natural(M)==True and natural(e)==True and natural(n)==True:

        a=M%n # 1r pas: Reduir la base a mòdul n

        q=e//2 # 2n pas: Convertim l'exponent a binari
        breves=[]
        r=e%2
        breves.append(r)

        while q != 0:
            r=q%2
            breves.append(r)
            q=q//2

        p=1
        s=a

        for element in breves: # 3r pas: A partir de p i s calculem el resultat
            if element == 1:
                p=(p*s)%n
            s=(s**2)%n

        return p
    else:
        return False
```

This function returns the result of the calculation M^e mod n. If M, n, or e are not natural numbers, it returns False.

The function is divided into the following parts:

```
    if natural(M)==True and natural(e)==True and natural(n)==True:
```

First, we make sure that M, n and e are natural numbers. It uses the function explained in the previous subroutine.

```
a=M%n # 1r pas: Reduir la base a mòdul n
```

The first step is to reduce M mod n (the new number is called a).

For example, we want to compute 430^13 mod 7. M = 430, e = 13, n = 7.

Then a = 430 % 7 = 3. Therefore, 430 ≡ 3 mod 7.

```
q=e//2 # 2n pas: Convertim l'exponent a binari
breves=[]
r=e%2
breves.append(r)

while q != 0:
    r=q%2
    breves.append(r)
    q=q//2
```

The second step is to express e in binary. We know that dividing by 2 the number and looking for the rest, we get the conversion in binary. If we want it correctly, we'll need to flip it upside down, but for this algorithm, we're interested in going from the LSB to the MSB, so for that reason, we'll leave it just as it comes out of the loop.

Before the loop, we compute the first quotient q, initialize the list breves, compute the first rest r and append this first rest to the list.

Inside the loop:

- First we compute the next rest by dividing q by 2
- We append r to the list
- Finally, we update q by dividing q by 2 (note that q is computed with the integer division)

The loop ends when q = 0.

Let's take for example e = 13:

Iteration	q	r
0	6	1
1	3	0

2	1	1
3	0	1

Therefore, the final breves = [1, 0, 1, 1]. As we said, the first one number is equivalent to the LSB of the number (13 in binary is 1101).

Once we have converted the exponent, let's move on to the last part. The order is from the LSB to the MSB.

```
    p=1
    s=a

    for element in breves: # 3r pas: A partir de p i s calculem el resultat
        if element == 1:
            p=(p*s)%n
        s=(s**2)%n

    return p
```

First, we initialize p = 1, and s = a, where a is the result of the reduction M mod n.

In the for loop, for every bit in breves, if this bit is 1, we update p (p*s mod n). Then, we update s (s = s^2 mod n). The final p will be the result of M^e mod n.

Let's take for example breves = [1, 0, 1,1] (from the previous example):

Iteration	p	s
0	1	3
1	3	2
2	3	4
3	5	2
4	3	4

The final p is 3, therefore, 3^13 mod 7 is 3.

```
else:
    return False
```

These last 2 lines ensure that the function returns False if M, e, and n are not natural numbers.

In order to check it’s functionality, we made this doctests:

```
    """
    Retorna el resultat del càlcul de la potència M^e mod n. Si M, n o e no són naturals,
retorna False.

    >>> fme(11,8,27)
    22

    >>> fme(3,13,7)
    3

    >>> fme(3,0,7)
    False

    >>> fme(3,-4,7)
    False

    >>> fme(3,4.5,7)
    False
    """
```

As you can see, all the items passed the tests:

```
1 item passed all tests:
   5 tests in Fast_Modular_Exponentiation.fme
5 tests in 2 items.
5 passed.
Test passed.
```

2.4. RSA Generation Key Algorithm

This subroutine can be found in the *RSA_Generation_Key_Algorithm.py* file.

```
from Extended_Euclidean_Algorithm import euclides, natural
from Fast_Modular_Exponentiation import fme

import random

def primer(x):
```

```
    """
    Retorna True si un nombre és primer i False si no ho és o bé si no és natural.

    >>> primer(4.5)
    False

    >>> primer(-8)
    False

    >>> primer(1)
    False

    >>> primer(2)
    True

    >>> primer(3)
    True

    >>> primer(14939)
    True

    >>> primer(14933)
    False
    """

    if x == 1 or natural(x)==False: # 1 per conveni no és primer
        return False

    elif x == 2: # 2 sempre és primer
        return True

    divisor=2
    r=1

    while divisor<=(int(x**0.5)+1) and r!=0: # Comprovem tots els divisors fins a sqrt(x)
        r=x%divisor
        divisor=divisor+1

    if r==0:
        return False

    else:
        return True
```

```
def Generar_e(phieuler):
    """
    En el context de l'RSA, genera una e aleatòria entre 2 i phi(n) i que sigui coprimer
amb phi(n).

    En aquest cas, no podem fer un doctest ja que al generar nombres aleatoris no podem
saber quin generarà.
    Tantmateix, posarem algun exemple del seu funcionament per veure que genera e
correctament.
    """

    trobat=False

    while trobat == False:

        e=random.randint(3,phieuler-1) #Ambdós estan inclosos, per això posem aquests
límits.
        _,_,mcd=euclides(e,phieuler)

        if mcd == 1:
            trobat = True

    return e

def GenKey(p,q):
    """
    En el context de l'RSA, genera una clau pública n,e i una clau privada d donats uns
primers p i q.
    Si p o q no són primers o no són naturals, retorna False.

    En aquest cas, no podem fer un doctest ja que al generar e aleatòria no podem saber
quina e generarà.
    Tanmateix, posarem algun exemple del seu funcionament per veure que genera n,e i d
correctament.

    >>> GenKey(8,-14)
    False

    >>> GenKey(8,1.4)
    False

    >>> GenKey(8,0)
```

```
    False
    """

    if natural(p)==True and natural(q)==True and primer(p)==True and primer(q) == True:

        n=p*q # Seguim els passos de l'RSA
        phieuler=(p-1)*(q-1)

        e=Generar_e(phieuler)

        d,_,_=euclides(e,phieuler)

        if d<0:
            while d<0:
                d=d+phieuler

        elif d>0:
            d=d%phieuler

        return(n,e,d)

    else:
        return False
```

Now we are going to explain what does each part of the code:

```
from Extended_Euclidean_Algorithm import euclides, natural
from Fast_Modular_Exponentiation import fme

import random
```

First of all, we need the functions natural(x), euclides(n, a), fme(M, e, n) from the previous files. We also need the random library, in order to generate the e (a part of the public key).

- **1st function: primer(x)**

```
def primer(x):

    if x == 1 or natural(x)==False: # 1 per conveni no és primer
        return False
```

```
    elif x == 2: # 2 sempre és primer
        return True

    divisor=2
    r=1

    while divisor<=(int(x**0.5)+1) and r!=0: # Comprovem tots els divisors fins a sqrt(x)
        r=x%divisor
        divisor=divisor+1

    if r==0:
        return False

    else:
        return True
```

This function returns True if a number is prime and False if it is not, or if it is not a natural number.

The function is divided into the following parts:

```
    if x == 1 or natural(x)==False: # 1 per conveni no és primer
        return False
```

First, we make sure that x is a natural number. It uses the function explained in the previous subroutine. It also makes sure that the number is not 1 (for convenience, 1 is not a prime number).

```
    elif x == 2: # 2 sempre és primer
        return True
```

For the method we chose, we need to take into account that 2 is a prime number. All the other cases will be computed in the next part.

```
    divisor=2
    r=1

    while divisor<=(int(x**0.5)+1) and r!=0: # Comprovem tots els divisors fins a sqrt(x)
        r=x%divisor
        divisor=divisor+1
```

Our idea is based on checking all the divisors one by one up to sqrt(x). If any of these divisions has a remainder of 0, it means that our number is not prime.

We start from the divisor 2 to sqrt(x). The initial value is 1 because we needed to create the variable initially different to 0.

Let's take for example primer(17):

Iteration	divisor	r
0	2	1
1	3	2
2	4	1
3	5	2

As you can see, all the remainders are different to 0. This means that 17 is a prime number.

```
    if r==0:
        return False

    else:
        return True
```

As we said, if r = 0, x is not a prime number. If r ≠ 0, x is a prime number.

- **2nd function: Generar_e(phieuler)**

```
def Generar_e(phieuler):

    trobat=False

    while trobat == False:

        e=random.randint(3,phieuler-1) #Ambdós estan inclosos, per això posem aquests límits.
        _,_,mcd=euclides(e,phieuler)

        if mcd == 1:
            trobat = True

    return e
```

This function generates a random e between 2 and phi(n) that is coprime with phi(n).

We made this function because if e is a fixed number (ex: e = 5), could happen following situation:

- p = 11, q = 101
- n = p * q = 1111
- phi(n) = 10 * 100 = 1000
- MCD(5, 1000) ≠ 1, so we need this function

Basically, we choose a random number between 2 and phi(n). Then we check with our previous function euclides, if the gcd(e, phi(n)) is 1. If it is, returns e. If it is not, it generates another random number.

- **3rd function: GenKey(p,q)**

```
if natural(p)==True and natural(q)==True and primer(p)==True and primer(q) == True:

    n=p*q # Seguim els passos de l'RSA
    phieuler=(p-1)*(q-1)

    e=Generar_e(phieuler)

    d,_,_=euclides(e,phieuler)

    if d<0:
        while d<0:
            d=d+phieuler

    elif d>0:
        d=d%phieuler

    return(n,e,d)

else:
    return False
```

This function follows all the steps from the RSA algorithm to generate public keys (n,e) and private keys (d).

```
if natural(p)==True and natural(q)==True and primer(p)==True and primer(q) == True:
```

First we check if p, q are naturals and also prime numbers.

```
        n=p*q # Seguim els passos de l'RSA
        phieuler=(p-1)*(q-1)

        e=Generar_e(phieuler)
```

Then we calculate n, phi(n) and with the previous function we generate e.

```
        d,_,_=euclides(e,phieuler)
```

With our function euclides, we need to calculate d (d*e is congruent to 1 mod phi(n)). So, we need to compute it with the euclidean algorithm, to compute the inverse of e mod phi(n).

```
        if d<0:
            while d<0:
                d=d+phieuler

        elif d>0:
            d=d%phieuler
```

If d < 0, we will fix d so that it is written in Z modulo phi(n)Z, adding phi(n) each time until it is greater than 0. If d > 0, we will calculate the remainder to write it in Z modulo phi(n)Z.

```
        return(n,e,d)
```

Finally, we return n, e and d.

```
    else:
        return False
```

These last 2 lines ensure that the function returns False if p and q are not natural or prime numbers.

In order to check it's functionality, we made this doctests:

- For the first function:

```
    """
    Retorna True si un nombre és primer i False si no ho és o bé si no és natural.

    >>> primer(4.5)
    False
```

```
    >>> primer(-8)
    False

    >>> primer(1)
    False

    >>> primer(2)
    True

    >>> primer(3)
    True

    >>> primer(14939)
    True

    >>> primer(14933)
    False
    """
```

For the third function:

```
    """
    En el context de l'RSA, genera una clau pública n,e i una clau privada d donats uns
primers p i q.
    Si p o q no són primers o no són naturals, retorna False.

    En aquest cas, no podem fer un doctest ja que al generar e aleatòria no podem saber
quina e generarà.
    Tanmateix, posarem algun exemple del seu funcionament per veure que genera n,e i d
correctament.

    >>> GenKey(8,-14)
    False

    >>> GenKey(8,1.4)
    False

    >>> GenKey(8,0)
    False
    """
```

As you can see, all the items passed the tests:

```
2 items passed all tests:
   3 tests in RSA_Generation_Key_Algorithm.GenKey
   7 tests in RSA_Generation_Key_Algorithm.primer
10 tests in 4 items.
10 passed.
Test passed.
```

Anyway, as we mentioned in the code, we use a random generation of e. This means we cannot perform tests on this part, but let's look at an example of how it works:

- p = 5 and q = 13:

```
n: 65
phi(n): 48
e: 31
d: -17
n final: 65
e final: 31
d final: 31
```

As you can see, the code works as expected.

2.5. The Encrypting Module

This we could call the "heart" of our RSA program, although encryption in RSA might seem simple enough as we only need to apply the formula c ≡ m^e mod n, but, in our case the complication arises when we have to determine m, meaning the message to encrypt, because a long message will need to be sent in "parts", let's look at the code to see how we did it:

```
from Fast_Modular_Exponentiation import fme # Importem funcions necessàries d'altres fitxers i llibreries de Python
import math


def calculate_block_size(n):
    """
    Podem treballar en bits o bytes, decidim treballar en bytes,aquesta funcio retorna el numero maxim de bytes que
   podem encriptar en un bloc.
    En un principi el nostre codi no tenia la part de restar 1, pero, una vegada acabat, el vam donar a una
```

```
    inteligencia artificial per tal de que sugerir millores, i ens va indicar que per
        seguretat era millor treballar en
    bits - 1.
    """
    bits = math.floor(math.log2(n)); #Indicat per el professor, ens permet obtenir el
        numero de bits a n.
    bytes_size = (bits - 1) // 8; #Convertim de bits a bites, tenint en compte el factor
        de seguretat
    return max(1, bytes_size) #Retornem


def text_to_blocks(text, block_size):
    """
    Converteix el text introduit a format UTF 8 en blocs de la mesura especificada.
    Afegeix padding si es necessari.
    """
    byte_data = text.encode('utf-8') #Converteix el text introduit per nosaltres en format
        UTF-8 a través de la funció encode de Python.

    #Utilizem encode perquè suposem que l'input serà un string.

    # Aquesta funció l'hem creat amb ajuda de inteligencia artificial
    blocks = [] # Creem la llista buida blocs
    for i in range(0, len(byte_data), block_size): # Fem servir la funció range. Crea una
        seqüència entre 0, la longitud del byte agafant steps del tamany del bloc.
        block = byte_data[i:i + block_size] # Comencem des d'i fins a i + block size
        (tamany del bloc)

        # En aquesta part afegim el padding necessari en cas que la longitud no sigui
        block size

        if len(block) < block_size:
            block = block + b'\x00' * (block_size - len(block)) # El nou bloc serà el bloc
        que teníem més el padding necesari fins a obtenir la longitud block size

        # Convertim el bloc a un enter
        block_int = int.from_bytes(block, 'big')
        blocks.append(block_int) # Ho afegim a la llista

    return blocks


def encrypt(text, n, e):
    """
    Funció d'encriptació final.

    Paràmetres:
    text (str): El text a encriptar (el qual convertim a UTF8)
    n (int): Public key
    e (int): Public key
```

```
    Retorna una llista dels blocs ja encriptats

    >>> encrypt("Hello!", 3233, 17)
    [3000, 1313, 745, 745, 2185, 1853]
    """
    # Calcula el tamany del bloc
    block_size = calculate_block_size(n)

    # Converteix text a blocs
    blocks = text_to_blocks(text, block_size)

    # Encripta cada bloc
    encrypted_blocks = [] # Creem la llista buida dels blocs encriptats
    for block in blocks:
        # Simplement encriptem utilzant c=m^e fent servir fme
        encrypted_block = fme(block, e, n)
        encrypted_blocks.append(encrypted_block)

    return encrypted_blocks
```

Now let's look at it chunk by chunk:

```
def calculate_block_size(n):
    """
    Podem treballar en bits o bytes, decidim treballar en bytes,aquesta funcio retorna el
numero maxim de bytes que
    podem encriptar en un bloc.
    En un principi el nostre codi no tenia la part de restar 1, pero, una vegada acabat,
el vam donar a una
    inteligencia artificial per tal de que sugerir millores, i ens va indicar que per
seguretat era millor treballar en
    bits - 1.
    """
     bits = math.floor(math.log2(n)); #Indicat per el professor, ens permet obtenir el
numero de bits a n.
    bytes_size = (bits - 1) // 8; #Convertim de bits a bites, tenint en compte el factor
de seguretat
    return max(1, bytes_size) #Retornem
```

We first create a function to compute the block size. This funciton does the following:

- First, we compute the number of bits required to "represent" n, this is done (indicated by the teacher) by computing the logarithm of n in base 2.
- Then we compute the "size" of our bytes (how many bytes), here as we say in the code, the artificial intelligence recommended us adding a factor of security of -1.
- Finally we return either 1 or the byte size to ensure that we have at least one byte to work with.

```python
def text_to_blocks(text, block_size):
    """
    Converteix el text introduit a format UTF 8 en blocs de la mesura especificada.
    Afegeix padding si es necessari.
    """
     byte_data = text.encode('utf-8') #Converteix el text introduit per nosaltres en format UTF-8 a través de la funció encode de Python.

    #Utilizem encode perquè suposem que l'input serà un string.

    # Aquesta funció l'hem creat amb ajuda de inteligencia artificial
    blocks = [] # Creem la llista buida blocs
    for i in range(0, len(byte_data), block_size): # Fem servir la funció range. Crea una seqüència entre 0, la longitud del byte agafant steps del tamany del bloc.
        block = byte_data[i:i + block_size] # Comencem des d'i fins a i + block size (tamany del bloc)

        # En aquesta part afegim el padding necessari en cas que la longitud no sigui block size

        if len(block) < block_size:
            block = block + b'\x00' * (block_size - len(block)) # El nou bloc serà el bloc que teníem més el padding necesari fins a obtenir la longitud block size

        # Convertim el bloc a un enter
        block_int = int.from_bytes(block, 'big')
        blocks.append(block_int) # Ho afegim a la llista

    return blocks
```

This was the function that gave us the most trouble, so we had to use the help of artificial intelligence to debug the code built by us, let's look at what this code does:

- First we turn the introduced text to utf-8, although the input should already be utf-8 we are more comfortable working with strings (alphanumeric characters) and then turning it to utf-8, as in python, unlike other languages, going from one to the other is extremely easy.

- Then we create a list where we will save all the data of the given text, in chunks of the desired size (block_size), we do this for all the byte_date, sepating it in chunks of size block_size, and in case we are at the end and we don't have enough data left in byte_data to create a chunk of block_size, then we add padding using **if len(block) < block_size:**
  block = block + b'\x00' * (block_size - len(block))
  This was one of our bigger problems, how to add the padding, but it seems that we overcomplicated things and the solution was far easier.

- Finally, we turn the block into an integer (in order to do calculations in the module), and then append it to the list previously created.

```
def encrypt(text, n, e):
    """
    Funció d'encriptació final.

    Paràmetres:
    text (str): El text a encriptar (el qual convertim a UTF8)
    n (int): Public key
    e (int): Public key

    Retorna una llista dels blocs ja encriptats

    >>> encrypt("Hello!", 3233, 17)
    [3000, 1313, 745, 745, 2185, 1853]
    """
    # Calcula el tamany del bloc
    block_size = calculate_block_size(n)

    # Converteix text a blocs
    blocks = text_to_blocks(text, block_size)

    # Encripta cada bloc
    encrypted_blocks = [] # Creem la llista buida dels blocs encriptats
    for block in blocks:
        # Simplement encriptem utilzant c=m^e fent servir fme
        encrypted_block = fme(block, e, n)
        encrypted_blocks.append(encrypted_block)

    return encrypted_blocks
```

This part is the “heart of the heart” of our code, what it does is pretty simple:

- First it uses the previously created functions, to first determine the size of the blocks, and then it uses the text_to_block function to obtain the blocks to be encrypted.
- After, it simply encrypts the code using the function previously created (Fast Modular Exponentiation algorithm) to compute $c \equiv m^e \bmod n$

In this part of the code we added a test because it will help us test all other functions, the test is the following:
>>> encrypt("Hello!", 3233, 17)
Answer: [3000, 1313, 745, 745, 2185, 1853]

But how do we know that with those inputs, the output should be the one we provided? By doing the computation by hand, so let’s do it step by step:

We are provided with the message Hello!, and the keys n and e. First we need to obtain the size of the blocks meaning that we will do the work that **calculate_block_size(n)** does:

$$bits = log_2(3233) = 11.659 \approx 11(nomes\ part\ entera\ (es\ el\ que\ fa\ math.floor))$$

$$block\ size = \frac{11-1}{8} \approx 1byte$$

Then we need to turn the text into blocks, meaning that we will do the work that the function **text_to_blocks** does, first we need to turn the message Hello! into UTF-8 and then to its integer value using a conversion table:

Letter	UTF-8	Integer
H	x48	72
e	x65	101
l	x6c	108
l	x6c	108
o	x6f	111
!	x21	33

In this case, as we saw above, the byte size is 1 so we convert each letter to a block. We apply the encryption formula c ≡ m^e mod n

Letter	UTF-8	Integer	c ≡ m^e mod n
H	x48	72	72^17 mod(3233)=3000
e	x65	101	101^17 mod(3233)=1313
l	x6c	108	108^17 mod(3233)=745
l	x6c	108	108^17 mod(3233)=745
o	x6f	111	111^17 mod(3233)=2185
!	x21	33	33^17 mod(3233)=1853

Finally, the blocks will be:
[3000, 1313, 745, 745, 2185, 1853]

```
1 item passed all tests:
   1 test in ENCRYPT_SYST.encrypt
1 test in 4 items.
1 passed.
Test passed.
```

As we can see, the test was successful.

2.6. The Decrypting Module

Obviously if we encrypt we eventually will need to decrypt, in RSA in order to decrypt we will need to compute the following c^d = (m^e)^ d = m^(1+kϕ(n)) ≡ m mod n, so in a way it is similar to encrypting from a coding perspective, let's look at the code we created:

```
from Fast_Modular_Exponentiation import fme # Importem funcions necessàries d'altres fitxers i llibreries de Python
from ENCRYPT_SYST import calculate_block_size


def blocks_to_text(blocks, block_size):
    """
    Converteix els blocs en text de nou
    """
    byte_data = b'' # Creem una nova variable on guardarem els bytes
    for block in blocks: # Per cada bloc a la llista blocks
         block_bytes = block.to_bytes(block_size, 'big') # Converteix els blocs en bytes, llegint de dreta a esquerra és a dir el byte més singificant primer
        byte_data += block_bytes #Guardem el resultat
    byte_data = byte_data.rstrip(b'\x00') #Vam preguntar al IA com eliminar el padding i ens va donar aquesta solució, utilizant la funció rstrip

    # Convertim de tornada a UTF8
    text = byte_data.decode('utf-8')
    return text



def decrypt(encrypted_blocks, n, d):
    """
    Desencripta el missatge, utilitzant els paràmetres n i d de l'RSA.
    Retorna un string amb el missatge desencriptat
    Un seguit de testos, el ultim ha de fallar.
    >>> decrypt([3000, 3179, 1853], 3233, 2753)
    'Hi!'
```

```
    >>> decrypt([669, 1307, 1307, 1759, 524, 99, 28], 3233, 2753)
    'GOODBYE'
    >>> decrypt([3000, 1313, 745, 745, 2185, 1853], 3233, 2753)
    'Hello!'
    >>> decrypt([3000, 1313, 745, 745, 2185, 1853], 3233, 2752)
    'Hello!'
    """
    # Calcula el tamany del bloc utilizant la funció que hem fet servir a l'encriptació
    block_size = calculate_block_size(n)

    # Desencriptem cada bloc
    decrypted_blocks = [] # Creem la llista on tindrem els blocs desencriptats
    for block in encrypted_blocks:
        # Desencriptem la funció fme anteriorment creada
        decrypted_block = fme(block, d, n)
        decrypted_blocks.append(decrypted_block) # Afegim els blocs a la llista


    return blocks_to_text(decrypted_blocks, block_size)
```

Now let’s look at it chunk by chunk:

```
def blocks_to_text(blocks, block_size):
    """
    Converteix els blocs en text de nou
    """
    byte_data = b'' # Creem una nova variable on guardarem els bytes
    for block in blocks: # Per cada bloc a la llista blocks
         block_bytes = block.to_bytes(block_size, 'big') # Converteix els blocs en bytes,
llegint de dreta a esquerra és a dir el byte més singificant primer
        byte_data += block_bytes #Guardem el resultat
     byte_data = byte_data.rstrip(b'\x00') #Vam preguntar al IA com eliminar el padding i
ens va donar aquesta solució, utilizant la funció rstrip

    # Convertim de tornada a UTF8
    text = byte_data.decode('utf-8')
    return text
```

This part of the code is the inverse of the previously created **text_to_blocks** function, therefore, it works in a similar manner.

- First we create a variable called byte_data.
- Then we separate the given blocks in chunks of the given size block_size, if it is smaller it will add padding at the start, the “big” at the end means that we are using Big-Endian representation, the most significant byte at the start. This solution was one we found on the internet.
- Then we add the extracted information to the previously created variable.
- Then we remove the padding, by using rstrip, a function we found by asking AI about it, it removes any trailing characters, like padding in our case.

Finally, we decode byte_data using UTF-8 and return the text.

```
def decrypt(encrypted_blocks, n, d):
    """
    Desencripta el missatge, utilitzant els paràmetres n i d de l'RSA.
    Retorna un string amb el missatge desencriptat
    Un seguit de testos, el ultim ha de fallar.
    >>> decrypt([3000, 3179, 1853], 3233, 2753)
    'Hi!'
    >>> decrypt([669, 1307, 1307, 1759, 524, 99, 28], 3233, 2753)
    'GOODBYE'
    >>> decrypt([3000, 1313, 745, 745, 2185, 1853], 3233, 2753)
    'Hello!'
    >>> decrypt([3000, 1313, 745, 745, 2185, 1853], 3233, 2752)
    'Hello!'
    """
    # Calcula el tamany del bloc utilizant la funció que hem fet servir a l'encriptació
    block_size = calculate_block_size(n)

    # Desencriptem cada bloc
    decrypted_blocks = [] # Creem la llista on tindrem els blocs desencriptats
    for block in encrypted_blocks:
        # Desencriptem la funció fme anteriorment creada
        decrypted_block = fme(block, d, n)
        decrypted_blocks.append(decrypted_block) # Afegim els blocs a la llista


    return blocks_to_text(decrypted_blocks, block_size)
```

This is the main function of decryption, what we do is the following:

- First we compute the block size using the same function as in the encryption.
- We create a list where we will save our decrypted blocks.
- Then, for each bloc in the list, we first apply c^d=m using Fast Modular Exponentiation algorithm
- And then we append it to the decrypted_blocks list.
- Finally, we use the previously created function **blocks_to_text** to return the text already decrypted.

In this part of the code, we added some tests because we are using various functions and it allows us to check them all, the tests are the following:

>>> decrypt([3000, 3179, 1853], 3233, 2753)
'Hi!'
>>> decrypt([669, 1307, 1307, 1759, 524, 99, 28], 3233, 2753)
'GOODBYE'
>>> decrypt([3000, 1313, 745, 745, 2185, 1853], 3233, 2753)

'Hello!'
>>> decrypt([3000, 1313, 745, 745, 2185, 1853], 3233, 2752)
'Hello!'

In order to know the blocks, and it's text we did the same as in encrypting but putting the result as a variable and not as a result. For example, using exactly the same as in encryption, the only variable we don't have is d, which we can compute because $d \cdot e \equiv 1 \bmod \phi(n)$. Where e= 17

First let's compute $\phi(n)$:

$$\phi(n) = \phi(3233) = \phi(53 * 61) = (53 - 1) * (61 - 1) = 3120$$

Then let's compute using euclides $d \cdot 17 \equiv 1 \bmod \phi(n)$
3120= 17*183+9 → 9=3120+17(-183)
17=9*1+8 → 8=17+9(-1)
9=8*1+1 → 1= 9+8(-1)

1=9+8(-1)=9+(17+9(-1))(-1)=9(2)+17(-1)=(3120+17(-183))(2)+17(-1)=3120(2)+17(-367)

So d= -367 mod 3120= 2753 mod 3120

Now all other tests.
In the last test we added a test that had to fail, by changing the d, in order to ensure that not any key could unlock the message.

As we can see all test were successful, with the exception of the one that had to fail

```
**********************************************************************
1 item had failures:
   1 of   4 in Decryp_SYSTM.decrypt
4 tests in 3 items.
3 passed and 1 failed.
***Test Failed*** 1 failure.
```

3. Final script

This script is basically the final test, here we put together all the previously created functions to reproduce RSA encryption of a message, so, ideally we should have:

Input: message, p, q
output: encrypted message and decrypted message (original message)

Then we should check that the original message and the decrypted one coincide.

The code is the following:

```
from ENCRYPT_SYST import encrypt
from Decryp_SYSTM import decrypt
from RSA_Generation_Key_Algorithm import GenKey  # Importem funcions necessàries
d'altres fitxers i llibreries de Python


def test_rsa_with_message(message, p, q):
    """
    Test RSA d'encriptació i desencriptació amb un missatge donat i uns primers p i q.
    """
    print(f"\nComprovant amb el següent missatge: '{message}'")
    print(" ")

    # Generem clau
    keys = GenKey(p, q)
    if keys == False:
        print("Error! Primers no vàlids")
        return

    n, e, d = keys
    print(f"Claus generades:")
    print(f"n = {n}")
    print(f"e = {e}")
    print(f"d = {d}")

    try:
        # Encriptem
        encrypted_blocks = encrypt(message, n, e)
        print(f"\nBlocs encriptats: {encrypted_blocks}")

        # Desencriptem
        decrypted_text = decrypt(encrypted_blocks, n, d)
        print(f"Text desencriptat: '{decrypted_text}'")

        # Verifiquem
        if decrypted_text == message:
            print("\nCorrecte! El missatge coincideix amb l'original")
```

```
        else:
            print("\nError! El missatge NO coincideix amb l'original")

    except ValueError as e:
        print(f"Error durant l'encriptació o desencriptació. Consell: Comprova que els
caràcters siguin vàlids!: {e}")


if __name__ == "__main__":
    # Provem amb un missatge llarg. Caràcters, nombres i caràcters especials
    test_message = "Hola! Això és un test d'RSA llarg. Haviam si funciona...
123%%%%====####¥¥¥"

    # Utilitzem primers grans
    p = 911
    q = 619

    test_rsa_with_message(test_message, p, q)
```

Let's look at it chunk by chunk:

```
    print(f"\nComprovant amb el següent missatge: '{message}'")
    print(" ")
```

- First we print (in a new line) a message to indicate to the user that message is being tested

```
 # Generem clau
    keys = GenKey(p, q)
    if keys == False:
        print("Error! Primers no vàlids")
        return

    n, e, d = keys
    print(f"Claus generades:")
    print(f"n = {n}")
    print(f"e = {e}")
    print(f"d = {d}")
```

- Then, using the GenKey function, we generate all the keys using the provided primes, if the numbers are not primes, then we will raise an error.
- Then we display the generated keys.

```
  try:
        # Encriptem
        encrypted_blocks = encrypt(message, n, e)
        print(f"\nBlocs encriptats: {encrypted_blocks}")
```

- Now we encrypt the message using the encrypt function, and save the result in a list
- Then we display the message already encrypted in blocks.

```
# Desencriptem
        decrypted_text = decrypt(encrypted_blocks, n, d)
        print(f"Text desencriptat: '{decrypted_text}'")
```

- After encrypting, we then decrypt the message using the decrypt function.
- Then we display the decrypted message.

```
        if decrypted_text == message:
            print("\nCorrecte! El missatge coincideix amb l'original")
        else:
            print("\nError! El missatge NO coincideix amb l'original")
```

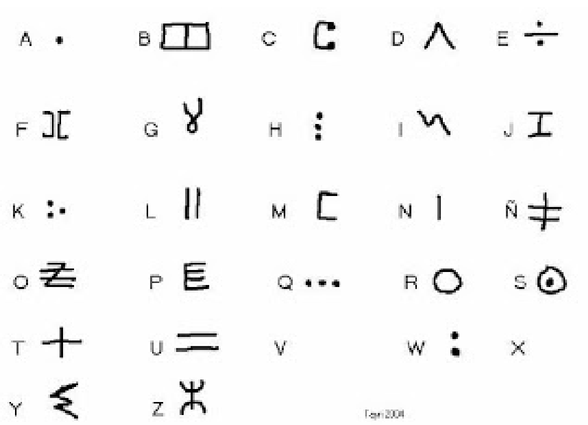
- Finally, we check if the original message and the decrypted message coincide, and display the consequent message.

```
    except ValueError as e:
        print(f"Error durant l'encriptació o desencriptació. Consell: Comprova que els caràcters siguin vàlids!: {e}")
```

As you can see, the whole process is within a try and except clause, that is because the message might be impossible to work with, or some other error.
(e → The object of exception.)

We added advice to the user because we believe that the majority of errors will come from characters that can not be translated to UTF-8, for example:

Some dead languages like Numidian are proposed to be included in Unicode but are not (as of unicode 16.0)



Numdian abecedary

Although one must wonder how they wrote them in the first place…

Any other possible error will be displayed after the message (saved as variable e).

Finally, we wanted to add some tests, but that was not possible because we use the random function, but, we can simply add an execution of the program at the end, so that was what we did.

```
if __name__ == "__main__":
   # Provem amb un missatge llarg. Caràcters, nombres i caràcters especials
      test_message  =  "Hola!  Això  és  un  test  d'RSA  llarg.  Haviam  si  funciona...
123%%%====####¥¥¥"

   # Utilitzem primers grans
   p = 911
   q = 619

   test_rsa_with_message(test_message, p, q)
```

if __name__ == "__main__": According to the internet, it is good practice to use this in case we want to reuse some function in the future.

As we can see we test with a very long message, using letters, numbers and special characters and using large primes to obtain a large n.

Once we ran the test we saw the following;

```
Comprovant amb el següent missatge: 'Hola! Això és un test d'RSA llarg. Haviam si funciona... 123%%%====####¥¥¥'

Claus generades:
n = 563909
e = 397637
d = 140333

Blocs encriptats: [380023, 133093, 379499, 550339, 19450, 48812, 153131, 176330, 353467, 561016, 362691, 360701, 292181, 82873, 142824, 102244, 35834
9, 375483, 148621, 234498, 497035, 43100, 226381, 323554, 353467, 133325, 37726, 503395, 105139, 226938, 296185, 356015, 356015, 115470, 115470, 4878
31, 487831, 59634, 59634, 59634]
Text desencriptat: 'Hola! Això és un test d'RSA llarg. Haviam si funciona... 123%%%====####¥¥¥'

Correcte! El missatge coincideix amb l'original
```

It displayed the keys generated, the encrypted blocks and it confirmed to us that the original message and decrypted message were the same.

We can compute
n=911*619= 563909

Using wolframalpha (very large numbers that's why we use wolfram) we can confirm that:

Input interpretation

solve	$d \times 397\,637 \equiv 1 \pmod{\phi(563\,909)}$

Solution in the least residue system

$d \equiv 140\,333 \pmod{562\,380}$

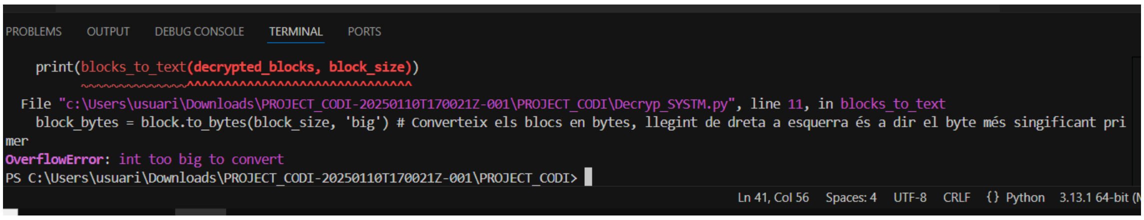
4. Conclusions

During this project, we were able to delve deeper into the concept of RSA. We had to learn how it works behind the scenes and familiarize ourselves with key concepts in cryptography (public key, private key, etc.), as well as the advantages of each. We also came to understand how, by simply using numbers and "simple" procedures, it is possible to ensure that a message remains secret between the people communicating.

On the other hand, we had to program, which meant choosing a programming language and comparing how difficult it would be to implement the project using different languages. We concluded that Python was the best choice, not only because we were already familiar with it from coursework in our degree, but also because it is a high-level language with many built-in functions (e.g., encode and decode utf-8).

During this project, we encountered many problems while programming. We learned to use all the resources available to solve them, including programming forums, general forums, drawing inspiration from other programs on the internet, programming tutorials, notes from previous courses, and artificial intelligence. With AI, we realized that it is essential to be cautious about what it generates and to thoroughly review it, as it may lack full context or produce incorrect outputs (a lesson we learned the hard way).

For example, here we have a problem that we found in a part of the decrypt code.



We were also able to see the importance of mathematics and its application in fields such as cybersecurity.

5. Webgraphy

1- **W3Schools**
URL: https://www.w3schools.com/

2- **Unicode Unsupported Characters**

URL: https://unicode.org/standard/unsupported.html

3-**ChatGPT by OpenAI**

URL: https://chat.openai.com/

4-**GeeksforGeeks**

URL: https://www.geeksforgeeks.org/

5-**Reddit**

URL: https://www.reddit.com/

6-**TutorialsPoint**

URL: https://www.tutorialspoint.com/index.htm

7-**UTF-8 on Wikipedia (Spanish)**

URL: https://es.wikipedia.org/wiki/UTF-8

8-**Numidian Language**

URL: https://en.wikipedia.org/wiki/Numidian_language

9-**UTF-8 Table**
URL: https://www.utf8-chartable.de/