Primeiramente vamos realizar os imports:

In [1]:
```
# import's para implementar o RSA
import numpy as np
import random as rd
from glob import glob
from math import log

# import's para trabalhar com o e-mail
import email
import smtplib
import imaplib
from getpass import getpass
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
```

Feito isso, vamos criar uma função para calcular o Algoritmo de Euclides Estendido, isso é, uma função que recebe dois inteiros a e b e retorna $mdc(a,b)$ e valores x e y tais que $a \cdot x + b \cdot y = mdc(a,b)$.

In [2]:
```
def euclid_extended(a, b):
    inverted = False
    if b > a:
        a, b = b, a
        inverted = True

    table = np.array([[a, b], [1, 0], [0, 1]])
    iteration = 0
    while table[0, (iteration + 1) % 2] != 0:
        a, b = table[0, iteration % 2], table[0, (iteration + 1) % 2]
        q = a // b
        table[:, iteration % 2] -= table[:, (iteration + 1) % 2] * q
        iteration += 1

    lcd, x, y = table[:, iteration % 2]
    if inverted:
        return lcd, y, x
    else:
        return lcd, x, y
```

Elaborada tal função, já temos o ferramental para, dados dois primos, gerar os parâmetros para a implementação do RSA, isso é, as duas chaves: pública e privada.

A geração da chave se dará do seguinte modo:

- dados dois primos p e q, calculamos $n = p \cdot q$ e $\phi(n) = (p-1) \cdot (q-1)$;
- feito isso, escolhemos e de modo que $mdc(e, \phi(n)) = 1$ e $2 < e < \phi(n)$;
- agora, encontramos d de modo que $2 < d < \phi(n)$ e $d \cdot e \equiv 1 \pmod{\phi(n)}$.

Note que os dois últimos passos podem ser realizados simultaneamente via Algoritmo de Euclides Estendido. Dessa forma, a chave pública será dada pelo par (n, e) enquanto a chave privada será dada pelo par (n, d).

In [3]:
```
def generate_keys(p, q):
    n = p * q
    phi_n = (p - 1) * (q - 1)
    e = rd.randint(3, phi_n)
```

```
    lcd, _, d = euclid_extended(phi_n, e)
    while lcd != 1 or d<10**20:
        e = rd.randint(3, phi_n)
        lcd, _, d = euclid_extended(phi_n, e)

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

Tendo as duas chaves, devemos ter funções que vão criptografar e descriptografar uma mensagem. Entretanto, a mensagem é um texto (string), e o RSA trabalha com números, então vamos primeiro criar funções que transformem strings para números e números para strings. A ideia para essas funções será transformar a mensagem para um inteiro em "base" 256 (quantidade de caracteres da tabela ASCII), bem como o caminho inverso. Para isso, estamos usando funções como `ord` e `chr`.

In [4]:
```
def str2int(message):
    exp = 1
    number = 0
    for i in range(len(message)):
        number += ord(message[i]) * exp
        exp *= 256

    return number

def int2str(number):
    message = ''
    while number != 0:
        temp = number % 256
        message += chr(temp)
        number -= temp
        number = number // 256

    return message
```

Tendo essas funções podemos, finalmente, elaborar funções que vão criptografar e descriptografar mensagens:

In [5]:
```
def encrypt(message, public_key):
    n, e = public_key
    m = str2int(message)
    m = pow(m, e, n)
    encrypted = int2str(m)

    return encrypted

def decrypt(encrypted, private_key):
    n, d = private_key
    m = str2int(encrypted)
    m = pow(m, d, n)
    message = int2str(m)

    return message
```

Agora vamos testar as funções. Para isso, criamos uma lista de primos, alguns desses primos foram pegos da Wikipédia.

In [6]:
```
primes = [100000000000000003, 100000000000000013, 170141183460469231731687303
p, q = rd.choice(primes), rd.choice(primes)
```

```
print(p)
print(q)
```

```
209889366574405864861512642566102225938639 21
100000000000000013
```

Sabemos que o algoritmo pode não funcionar corretamente caso os primos sejam muito grandes para comportar a mensagem, veja:

In [7]:

```
text = '''Você sabia? Resolver equações na Roma antiga era bem mais fácil. Af
Estatísticas comprovam: água causa morte. Segundo os cientistas, 100% das mor
p, q = primes[-3], primes[-2]
public_key, private_key = generate_keys(p, q)
t = encrypt(text, public_key)
e = decrypt(t, private_key)
m = str2int(text)
print(e)
print()

p, q = primes[-1], primes[-2]
public_key, private_key = generate_keys(p, q)
t = encrypt(text, public_key)
e = decrypt(t, private_key)
print(e)
```

```
u    CÂæù@Ð¬1

þ üþ« þHA¶Å5N
1 þî«ûúSú òòôÿü÷L ï õkþH
É½  ¦ 90W       éMaÔ¸X'ðøØ ±  ß J  ø ugg Èª yBt×ZÅ}ë Þþc  à > ï z aS ^ßÉå çîïÁ
ÅbÜ#§

Você sabia? Resolver equações na Roma antiga era bem mais fácil. Afinal, o va
lor de x era sempre 10.
Estatísticas comprovam: água causa morte. Segundo os cientistas, 100% das mor
tes ocorrem em seres que bebem água.
```

Para isso, vamos analisar o maior tamanho de mensagem que é suportada em função de n. Para tanto, note que as mensagens são transformadas em um número na base 256, assim, o maior número pode ter, no máximo, $\lfloor \log_{256} n \rfloor$ dígitos. Pensando nisso, vamos separar as mensagem em trechos que vão possuir entre $\lfloor \frac{\log_{256} n}{2} \rfloor$ e $\lfloor \log_{256} n \rfloor$ caracteres, possibilitando que tenhamos mensagens arbitrariamente grandes.

Dito isso, podemos fazer uma reimplementação das funções para criptografar e descriptografar.

In [8]:

```
def encrypt(message, public_key):
    n, e = public_key
    m = message
    L = int(log(n, 256))
    l = int(log(n, 256) / 2)
    parts = []
    while len(m) > L:
        r = rd.randint(l, L)
        aux, m = m[:r], m[r:]
        parts.append(aux)
    else:
        parts.append(m)

    encrypted = ''
    for part in parts:
        part = str2int(part)
```

```
        part = pow(part, e, n)
        encrypted += chr(256) + int2str(part)

    return encrypted

def decrypt(encrypted, private_key):
    encrypted = encrypted.split(chr(256))
    n, d = private_key
    message = ''
    encrypted.remove('')
    for part in encrypted:
        m = str2int(part)
        m = pow(m, d, n)
        message += int2str(m)

    return message
```

Agora, podemos criar uma rotina para utilizarmos o algoritmo implementado acima. A ideia será trocar e-mails criptografados. Para tanto, criamos uma função que recebe um usuário e senha, além de um endereço de e-mail de destinatário, assunto, corpo do e-mail e a chave pública. Para simplificar na localização do e-mail que vamos descriptografar para ler, essa função também recebe como parâmetro uma tag para marcar esse e-mail.

In [9]:

```
def send_mail(username, password, mail_address, subject, body, public_key, ta
    body = encrypt(body, public_key)
    subject = tag + subject
    msg = MIMEMultipart()
    msg['From'] = username
    msg['To'] = mail_address
    msg['Subject'] = subject
    msg.attach(MIMEText(body, 'plain'))
    server = smtplib.SMTP('smtp.gmail.com', 587)
    server.ehlo()
    server.starttls()
    server.ehlo()
    server.login(username, password)
    text = msg.as_string()
    server.sendmail(username, mail_address, text)
    server.quit()
```

Tendo enviado o e-mail, criamos uma função que recebe o usuário e a senha de um e-mail, além da chave privada e lê os e-mails criptografados (marcados com a tag no assunto).

In [10]:

```
def read_email(username, password, private_key, tag = '[Encrypted]', qtd = 1)
    printed = 0
    crip = False
    server = 'imap.gmail.com'
    mail = imaplib.IMAP4_SSL(server)
    mail.login(username, password)
    mail.select('inbox')
    data = mail.search(None, 'ALL')
    mail_ids = data[1]
    id_list = mail_ids[0].split()
    first_email_id = int(id_list[0])
    latest_email_id = int(id_list[-1])
    for i in range(latest_email_id, first_email_id, -1):
        if printed == qtd:
            break

        data = mail.fetch(str(i), '(RFC822)')
        for response_part in data:
```

```
            arr = response_part[0]
            if isinstance(arr, tuple):
                msg = email.message_from_string(str(arr[1], 'utf-8'))
                email_subject = msg['subject']
                email_from = msg['from']
                if tag not in email_subject:
                    break
                else:
                    printed += 1
                    crip = True

                print('From: ' + email_from)
                print('Subject: ' + email_subject)
                print()
                for part in msg.walk():
                    content_type = part.get_content_type()
                    content_disposition = str(part.get('Content-Disposition'))
                    try:
                        body = part.get_payload(decode = True).decode()
                    except:
                        pass

                    if content_type == 'text/plain':
                        body = decrypt(body, private_key)
                        print(body)

        if printed != qtd and crip:
            print()
            crip = False
```

Agora, temos um código para gerar e salvar as chaves. A ideia de salvar as chaves é para possibilitar que carreguemos as mesmas, podendo ler os e-mails mesmo após fechar esse notebook, caso contrário perderíamos as duas chaves.

In [11]:

```
if 'public.csv' not in glob('*.csv'):
    p, q = rd.choice(primes), rd.choice(primes)
    while 1/5 < p/q < 5:
        p, q = rd.choice(primes), rd.choice(primes)
    public_key, private_key = generate_keys(p, q)
    with open('public.csv', 'w') as file:
        file.write(str(public_key[0]) + '\n')
        file.write(str(public_key[1]) + '\n')

    with open('private.csv', 'w') as file:
        file.write(str(private_key[0]) + '\n')
        file.write(str(private_key[1]) + '\n')
else:
    file = open('public.csv')
    public_key = file.readlines()
    file.close()
    public_key = tuple([int(i) for i in public_key])

    file = open('private.csv')
    private_key = file.readlines()
    file.close()
    private_key = tuple([int(i) for i in private_key])
```

Agora, vamos pegar as credenciais do usuário:

In [12]:

```
username = input('Username: ')
password = getpass(prompt = 'Password: ')
```

```
Username: igorpmichels@gmail.com
Password: ········
```

E, finalmente, vamos enviar um e-mail utilizando o RSA:

In [13]:
```
body = '''Boa noite professor,

segue o link para o repositório do GitHub com o nosso trabalho: https://githu

Abraço,
Igor'''

mail_address = username # 'luca.escopelli@gmail.com' # destinatário (igual o
subject = 'Entrega do Trabalho'

send_mail(username, password, mail_address, subject, body, public_key)
```

Agora, vamos ler o e-mail que acabamos de enviar e que foi criptografado:

In [14]:
```
read_email(username, password, private_key)
```

```
From: igorpmichels@gmail.com
Subject: [Encrypted] Entrega do Trabalho

Boa noite professor,

segue o link para o repositório do GitHub com o nosso trabalho: https://githu
b.com/IgorMichels/RSA

Abraço,
Igor
```