Práctica 1: Criptografía clásica

Fundamentos de Criptografía y Seguridad Informática

UAM, 2022/2023

Maitane Gómez González

Ana Martínez Sabiote

El lenguaje elegido para la implementación de esta práctica ha sido Python. La memoria de la práctica y el código en Python desarrollado están contenidos en este notebook. Se entrega tanto en formato .pdf para su lectura, como el fichero .ipynb, que permite la ejecución del código en Jupyter Notebook.

1. Sustitución monoalfabeto

1.a Método afín

Para implementar el método afín, al igual que para la mayoría de los demás cifrados de la práctica, utilizamos un alfabeto formato por todos los caracteres en minúsculas y todos los caracteres en mayúsculas. Por tanto, un alfabeto de 52 elementos. Si una cadena contiene además símbolos que no están contemplados en nuestro alfabeto, éstos no se tienen en cuenta.

Para el desarrollo de este apartado hemos implementado las siguientes funciones

- La función *algoritmo_euclides(a,b)* que calcula el máximo común divisor de los dos números pasados como parámetros de manera recursiva.
- La función *algoritmo_euclides_extendido(a,b)* que aplica el algoritmo de extendido de Euclides que hemos estudiado. Esta función recibe dos números *a* y *b* como parámetros y devuelve el máximo común divisor de ellos, *u* y *v*, los coeficientes de la Identidad de Bézout: **1=a*u + b*v**. Por tanto con esta función obtenemos el inverso de *a* módulo *b*, que es *u* y recíprocamente, el inverso de *b* mod *a*, que es *v*.
- La función *inverso(a,m)*, que calcula el inverso multiplicativo de *a* módulo *m* si *a* y *m* son primos relativos.

In [1]:

```
import gmpy2
from gmpy2 import mpz
import sympy
import numpy as np
```

In [2]:

```
def algoritmo_euclides(a,b):
    if gmpy2.t_mod(a,b) == 0:
        return b
    else:
        return algoritmo_euclides(b, gmpy2.t_mod(a,b))
```

In [3]:

```
mcd=algoritmo_euclides(39,150)
print(mcd)
```

```
3
```

In [4]:

```
algoritmo_euclides(7,15)
```

Out[4]:

```
mpz(1)
```

In [5]:

```
def algoritmo_euclides_extendido(a,b):
    """
    # Condición a>b, sino las cambiamos
    if b>a:
        aux=a
        a=b
        b=aux
    """
    # Identidad de Bézout 1=u*a + v*b
    # El inverso de a módulo b es u. Recíprocamente, el inverso de b mod a es v
    if a==0:
        mcd=b
        u=0
        v=1
    else:
        mcd, x, y = algoritmo_euclides_extendido(gmpy2.c_mod(b,a), a)
        u=gmpy2.sub(y,(gmpy2.mul(gmpy2.c_div(b,a),x)))
        v=x

    return mcd, u, v
```

In [6]:

```
def inverso(a,m):
    result = algoritmo_euclides_extendido(a,m)
    # Comprobamos que el mcd es 1 para que exista inverso multiplicativo
    # En consecuencia, a y m determinan una función afín inyectiva
    if result[0] == 1:
        # Entonces devolvemos el coeficiente u (que acompaña a) de la Id. de Bézout
        inv=result[1]
        return inv
    else:
        print("Error")
```

In [7]:

```
inverso(51,23)
```

Out[7]:

```
mpz(-9)
```

Las siguientes dos funciones: *read_input* y *read_output* se utilizarán a lo largo de todos los ejercicios de esta práctica para gestionar el comportamiento de entrada y salida requerido.

- *read_input(i)* lee por la entrada estándar si el parámetro *i* es nulo. De lo contrario, abre el fichero pasado como parámetro.
- *read_output(o, cadena)* imprime el parámetro *cadena* por la entrada estándar si el parámetro *o* es nulo. De lo contrario, escribe *cadena* en el fichero *o*, y si no existe, lo crea.

In [8]:

```
def read_input(i):
    # Primero tomamos el input de i o de la entrada estándar
    if i==0:
        cadena=input()
    else:
        file=open(i, "r")
        cadena=file.read()
        file.close()

    if len(cadena)<50:
        print("Cadena: {}".format(cadena))
    return cadena
```

In [9]:

```
def print_output(o,cadena):
    if o==0:
        print("Cadena: {}".format(cadena))
    else:
        file=open(o, "w")
        cadenaToStr = ' '.join([str(elem) for elem in cadena])
        file.write(cadenaToStr)
        file.close()
```

La siguiente función implementa el método afín.

La llamada a la función:

afin {-C|-D} {-m |Zm|} {-a N×} {-b N+} [-i filein] [-o fileout]

- -C el programa cifra
- -D el programa descifra
- -m tamaño del espacio de texto cifrado
- -a coeficiente multiplicativo de la función afín
- -b término constante de la función afín
- -i fichero de entrada
- -o fichero de salida

Como estamos trabajando con funciones de Python en celdas de Jupyter notebook, no con scripts, la llamada de la función se realiza en una celda así: afin(modo,m,a,b,i,o), indicando como modo *-C* o *-D*, *m*, *a*, *b* como enteros y *i*, *o* se introducen opcionalmente: si no se especifican, por defecto realiza la operación (entrada o salida) con la estándar y si se especifican, se trabaja con los ficheros proporcionados.

Los parámetros *a* y *m* deben ser primos relativos, es la primera condición que verifica nuestra función. Si lo son, continúa con el algoritmo de cifrado o descifrado, según se haya especificado en la llamada. Además, *m* debe se la longitud de nuestro alfabeto.

El esquema de funcionamiento que sigue este método y los demás métodos de la práctica es:

1. Traducimos el input de caracteres a números enteros utilizando nuestro alfabeto de 52 elementos.
2. Aplicamos el mecanismo de cifrado o descifrado sobre la cadena numérica. En este caso el del cifrado afín.
3. Obtenemos una cadena cifrada numérica, que pasamos de números a caracteres usando nuestro alfabeto.

In [10]:

```
def afin(modo,m,a,b,i=0,o=0):
    alfabeto='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
    if algoritmo_euclides(a,m) == 1:
        if modo=="-C":
            cadena=read_input(i)
            #Traducimos los caracteres a números
            cadena_numerica=[]
            for k in cadena:
                if k in alfabeto:
                    cadena_numerica.append(alfabeto.index(k))
            cadena_cifrada=[]
            for k in cadena_numerica:
                cadena_cifrada.append(((a*k)+b)%m)

            #Traducimos la cadena_cifrada numérica a caracteres
            resul=""
            for i in range(len(cadena_cifrada)):
                   resul=resul+alfabeto[cadena_cifrada[i]]

            print_output(o,resul)

        elif modo=="-D":

            cadena_cifrada=read_input(i)

            #Traducimos los caracteres a números
            cadena_numerica=[]
            for k in cadena_cifrada:
                if k in alfabeto:
                    cadena_numerica.append(alfabeto.index(k))

            cadena_descifrada=[]
            cadena_texto=""
            #inv=inverso(m,a)
            inv=pow(a, -1, m)
            print(inv)
            for i in range(len(cadena_numerica)):
                cadena_numerica[i]=int(cadena_numerica[i])

            for k in cadena_numerica:
                k_descifrado=gmpy2.c_mod(gmpy2.mul((k-b),inv),m)
                if k_descifrado<0:
                    k_descifrado=m+k_descifrado
                cadena_descifrada.append(k_descifrado)
                cadena_texto=cadena_texto+alfabeto[k_descifrado]

            print_output(o, cadena_texto)
    else:
        print("{} y {} no son primos relativos. Error".format(a,m))
```

In [11]:

```
afin("-C",51,23,3,"cadena.txt","cadena_cifrada.txt")
```

```
Cadena: Hola Nueva York!
```

In [12]:

```
1 afin("-D",51,23,3, "cadena_cifrada.txt")
```

```
Cadena: W t b d H e S B d F t L D
20
Cadena: HolaNuevaYork
```

In [13]:

```
1 afin("-C",130,16,27)
```

```
16 y 130 no son primos relativos. Error
```

In [14]:

```
1 afin("-C",51,13,0)
```

```
Universidad Autonoma de Madrid
Cadena: Universidad Autonoma de Madrid
Cadena: LqcsbrEcNaNGfRDqDdaNbJaNrcN
```

In [16]:

```
1 afin("-D",51,13,0)
```

```
LqcsbrEcNaNGfRDqDdaNbJaNrcN
Cadena: LqcsbrEcNaNGfRDqDdaNbJaNrcN
4
Cadena: UniversidadAutonomadeMadrid
```

1.b Criptoanálisis del cifrado afín

Nuestro afin no trivial se basa en aumentar el tamaño de la clave. Hemos decidido cambiar el tamaño de la base y cifrarlo en bigramas. Lo que nos daría un espacio de 676.Con el afín normal, en nuestro caso, obteniamos uno de 52.

Como se puede observar en la función fortaleza y en su ejecución el normal nos daría 128 claves y el no trivial 115454445638713287613492120981354885654453486093934770480152773664000000000.

In [17]:

```
1 def fortaleza(m):
2     z_m_inv=sympy.totient(m) #calculamos la funcion phi
3     return gmpy2.mul((m),z_m_inv) #calculamos la fortaleza multiplicando Zm y Zm*
4
```

In [18]:

```
print(fortaleza(26**26+26))
#fortaleza(26) este seria el resultado si no aceptaramos mayúsculas
print(fortaleza(52))

```

```
1154544456387132876134921209813548856544534860939347704801527736640000000
1248
```

El siguiente programa implementa el método afín no trivial.

Llamada a la función:

afin_no_trivial {-C|-D} {-m |Zm|} {-a N×} {-b N+} [-i filein] [-o fileout]

- -C el programa cifra
- -D el programa descifra
- -m tamaño del espacio de texto cifrado
- -a coeficiente multiplicativo de la función afín
- -b término constante de la función afín
- -i fichero de entrada
- -o fichero de salida

En este programa utilizamos un alfabeto de 26 elementos, las letras del abecedario en minúsculas

In [19]:

```
def afin_no_trivial(modo,m,a,b,i=0,o=0):
    alfabeto='abcdefghijklmnopqrstuvwxyz'
    digrama=([]) #generamos un vetor de di-gramas del alfabeto que hemos declarado arr
    for k in alfabeto:
        for j in alfabeto:
            digrama.append(k+j)

    if algoritmo_euclides(a,m) == 1:
        if modo=="-C":
            #obtenemos el texto claro, si no se ha pasado por parámetro, se obtine de
            cadena=read_input(i)


            #ponemos la cadena de entrada solo en caracteres de nuestro alfabeto
            cadena_formateada=""
            for k in range(len(cadena)):
                if cadena[k] in alfabeto:
                    cadena_formateada=cadena_formateada+cadena[k]

            #Traducimos los caracteres a números para poder operar
            cadena_numerica=[]
            j=0
            if len(cadena_formateada)%2==0:
                    while j <(len(cadena_formateada)-1):
                        cadena_numerica.append(digrama.index(cadena_formateada[j]+cade
                        j=j+2
            else:
                    while j <(len(cadena_formateada)-2):
                        cadena_numerica.append(digrama.index(cadena_formateada[j]+cade
                        j=j+2
                    cadena_numerica.append(alfabeto.index(cadena_formateada[len(cadena

            #utilizamos la función de cifrado
            cadena_cifrada=[]
            for k in cadena_numerica:
                cadena_cifrada.append(((a*k)+b)%m)

            #pasamos el resultado a caracteres y lo guardamos como string
            resul=""
            for k in cadena_cifrada:
                resul=resul+digrama[k]

            #si no se ha pasado un fichero de salida por parámetro, imprimimos el resu
            print_output(o,resul)

        elif modo=="-D":

            cadena_cifrada=read_input(i)
            #cadena_cifrada=input()

            cadena_descifrada=[]
            cadena_texto=""

             #obtenemos el texto claro, si no se ha pasado por parámetro, se obtine de
            #if i==0:
             #    cadena_cifrada=input()
            #cadena=read_input(i)
            #else:
             #   file=open(i, "r")
```

```
            #   cadena_cifrada=file.read()
            #  file.close()

            #obtenemos el inverso en el modulo para poder utilizar la función de desci
            #ya hemos comprobado al principio que el inverso existe.

            #inv=inverso(a,m)
            inv=pow(a, -1, m)

            #pasamos el texto a un formato numérico para poder operar
            cadena_numerica=[]
            j=0
            if len(cadena_cifrada)%2==0:
                while j <(len(cadena_cifrada)-1):
                    cadena_numerica.append(digrama.index(cadena_cifrada[j]+cadena_cifr
                    j=j+2
            else:
                while j <(len(cadena_cifrada)-2):
                    cadena_numerica.append(digrama.index(cadena_cifrada[j]+cadena_cifr
                    j=j+2
                cadena_numerica.append(alfabeto.index(cadena_cifrada[len(cadena_cifrad

            #desciframos el texto con la función de descifrado:
            for k in cadena_numerica:
                k_descifrado=gmpy2.c_mod(gmpy2.mul((k-b),inv),m)
                if k_descifrado<0: #ajustamos el modulo
                    k_descifrado=m+k_descifrado
                cadena_descifrada.append(k_descifrado)

            #pasamos el texto a caracteres
            if len(cadena_cifrada)%2==0:
                for k in cadena_descifrada:
                    cadena_texto=cadena_texto+digrama[k]
            else:
                for k in range(len(cadena_descifrada)-1):
                    cadena_texto=cadena_texto+digrama[cadena_descifrada[k]]

                cadena_texto=cadena_texto+alfabeto[cadena_descifrada[(len(cadena_desci

            #si no se ha pasado un archivo para guardar el resultado, se imprime por p
            print_output(o, cadena_texto)
    else:
        print("{} y {} no son primos relativos. Error".format(a,m))
```

In [97]:

```
afin_no_trivial("-C",701,23,3)
```

```
Universidad Autonoma de Madrid
Cadena: Universidad Autonoma de Madrid
Cadena: jqbjcqzacuspkcjfculgdhza
```

In [98]:

```
1 afin_no_trivial("-D",701,23,3, "cadena_trivial.txt")
```

```
Cadena: ltkmalzdzefiyuzhwn
Cadena: holasupernuevayork
```

In [ ]:

```
1 afin_no_trivial("-D",701,23,3, 0, "resultado_trivial.txt")
```

In [100]:

```
1 afin_no_trivial("-D",701,23,3)
```

```
holasupernuevayork
Cadena: holasupernuevayork
Cadena: vlqxfnarizjcgugyby
```

El cifrado afin muy vulnerable a los ataques. Se rompe imediatamente con B,C,D y E. Con A hace falta un analisis de frecuencias (El análisis de frecuencia es el estudio de la frecuencia de letras o grupos de letras en un texto cifrado).

Ejemplo de criptoánalisis afín

Hemos cifrado "antiaereo" con afin (modulo 51, a=13 y b=0), lo que nos ha dado la cadena "aqRcabrbD".

En el ejemplo de abajo hemos guardado las tablas de frecuencia del castellano y el ingles y luego las hemos ordenado por mayor a menor. En este caso solo hemos utilizado la del castellano.

Hemos conseguido descifrarlo con la segunda hipotesís: c1: la posición del elemento más utilizado de la cadena en el alfabeto. c2: la posición del segundo elemento más utilizado de la cadena en el alfabeto. t1: la posición del elemento más utilizado en el alfabeto. t2: la posición del segundo elemento más utilizado en el alfabeto.

pos(c1)=pos(t1)* a+b pos(c2)=pos(t1)* a+b

Se puede resolver de dos formas: 1- restando las ecuaciones, lo que nos daría el resultado de: pos(c1)-pos(c2)=(pos(t1)-pos(t2))* a -> a=pos(c1)-pos(c2) inv((pos(t1)-pos(t2)) b=pos(c1)-pos(t1)* a

2- Al introducir los datos conocidos, nos damos cuenta de que se puede simplificar y resolver casi directamente: pos(c1)=pos(t1)* a+b -> 1=4* a+b->a=1* inv(4)=13 pos(c2)=pos(t1)* a+b -> 0=0* a+b ->b=0

En ambos casos el resultado da a=13, b=0. El cifrado esta roto.

In [20]:

```
alfabeto='abcdefghijklmnopqrstuvwxyz'
cadena = "aqRcabrbD"

castellano=(['a', 11.96],['b', 0.92],['c', 2.92],['d', 6.87],['e', 16.78],['f', 0.52],[
           ['h', 0.89],['i', 4.15],['j', 0.3],['k', 0.0],['l', 8.37],['m', 2.12],['n',
           ['o', 8.69],['p', 2.77],['q', 1.53],['r', 4.94],['s', 7.88],['t', 3.31],['u
           ['v', 0.39],['w', 0.0],['x', 0.06],['y', 1.54],['z', 0.15])

ingles=(['a', 11.96],['b', 1.54],['c', 3.06],['d', 3.99],['e', 12.51],['f', 2.30],['g',
        ['h', 0.89],['i', 7.26],['j', 0.16],['k', 0.67],['l', 4.14],['m', 2.53],['n', 7
       ['o', 7.60],['p', 2.0],['q', 0.11],['r', 6.12],['s', 6,54],['t', 9.25],['u', 2.7
       ['v', 0.99],['w', 1.92],['x', 1.92],['y', 1.73],['z', 0.19])


castellano_ord=sorted(castellano, key=lambda letra: letra[1], reverse=True)
ingles_ord=sorted(ingles, key=lambda letra: letra[1], reverse=True)

print("Primera hipotesis:")
c1=alfabeto.index("a")
c2=alfabeto.index("b")
t1=alfabeto.index(castellano_ord[0][0])
t2=alfabeto.index(castellano_ord[1][0])

#pos(c1)=pos(t1)*a+b -> 0=4*a+b->a=-1*inv(4)
#-
#pos(c2)=pos(t1)*a+b -> 1=0*a+b ->b=1
a=1*pow(-1,-1,51)

#comprobamos que hemos resuelto bien la ecuación
#comprobamos que sean co-primos
if algoritmo_euclides(a,51)==1:
        b=c1-int(a)*t1
        if a==13 and b%51==0:
            print("es correcto, antiaereo se cifro con 13 y 0")
        else:
            print("No es correcto")
else:
    print("no son coprimos")


print("Segunda hipotesis:")
c1=alfabeto.index("b")
c2=alfabeto.index("a")
t1=alfabeto.index(castellano_ord[0][0])
t2=alfabeto.index(castellano_ord[1][0])

#pos(c1)=pos(t1)*a+b -> 1=4*a+b->a=1*inv(4)
#-
#pos(c2)=pos(t1)*a+b -> 0=0*a+b ->b=0

#pos(c1)-pos(c2)=(pos(t1)-pos(t2))*a
#0-1=(4-0)*a
#a=1*inv(4)

a=1*pow(4,-1,51)

#comprobamos que hemos resuelto bien la ecuación
#comprobamos que sean co-primos
```

```
if algoritmo_euclides(a,51)==1:
        b=c1-int(a)*t1
        if a==13 and b%51==0:
            print("Es correcto, antiaereo se cifro con 13 y 0")
        else:
            print("No es correcto")

else:
    print("no son coprimos")
```

```
Primera hipotesis:
No es correcto
Segunda hipotesis:
Es correcto, antiaereo se cifro con 13 y 0
```

2. Sustitución polialfabeto

2.a Método de Hill

El siguiente programa implementa el método hill. El cifrado Hill es de sustitución poligráfica basado en álgebra lineal.

En este cifrado se utiliza una matriz cuadrada como clave de dimesiones n*n, tenemos que divir el texto claro (ahora numerico) en bloques de n elementos. Si la división no es exacta, se hace padding.

El requisito principal para poder cifrar y descifrar, es que la matriz tenga una función biyectiva. Si no fuera así habría que cambiar los datos ya que si lo cifráramos no podríamos descifrarlo.

Primero se asocia cada letra del alfabeto con un número. La forma más sencilla es hacerlo con la asociación natural ordenada pero se podría hacer mediante otras asociaciones.

Después, aplicamos la función de cifrado y volvemos a pasar el resultado a letras. Para descifrar el proceso es muy parecido, simplemente hay que cambiar la función de cifrado por la de descifrado. Y para ello tenemos que tener la inversa de la matriz calculada.

In [21]:

```
import numpy as np
import os
import math
import copy
```

Funcion que cálcula el determinante de una matriz.

determinante{matriz}

In [22]:

```python
def determinante(matriz):

    if len(matriz)==2 and len(matriz[0])==2:
        #calculamos el determinante
        det=matriz[0][0]*matriz[1][1]-(matriz[1][0]*matriz[0][1])

        return det
    else:
        suma=0
        for i in range(len(matriz)): #calculamos el determinante por cofactores
            maux=copy.deepcopy(matriz)
            maux.remove(matriz[0]) #eliminamos la primera fila
            for j in range(len(maux)):
                maux[j]=maux[j][0:i]+maux[j][i+1:]


            suma= suma+ (-1)**((i+j)%2)*matriz[0][i]*determinante(maux)

        return suma

```

In [23]:

```python
#comprobación de la función
matriz = [[11,8], [3,7]]
print(determinante(matriz))
```

```
53
```

Función que cálcula el adjunto de una matriz.

adjunto{matriz}

In [24]:

```
def adjunto(matriz):
    adjunto=np.zeros(np.shape(matriz))
    if len(matriz)==2 and len(matriz[0])==2:
         #calculamos el adjunto
        adjunto[0][0]=matriz[1][1]
        adjunto[0][1]=-matriz[0][1]
        adjunto[1][0]=-matriz[1][0]
        adjunto[1][1]=matriz[0][0]

        return adjunto
    else:

        for i in range(len(matriz)):
            maux=copy.deepcopy(matriz)
            for j in range(len(matriz)):

                maux=np.delete(matriz,i,0)
                aux=np.delete(maux,j,1)
                auxi=aux.tolist()
                #la matriz de cofactores transpuesta es el djunto
                adjunto[j][i]=(-1)**((i+j)%2)*determinante(auxi)


        return adjunto
```

In [25]:

```
#comprobación de la función
matriz = [[11,8], [3,7]]
print(adjunto(matriz))
```

```
[[ 7. -8.]
 [-3. 11.]]
```

Función que cálcula la inversa de una matriz.

inversa{matriz}{modulo}

In [26]:

```
def inversa(matriz,modulo):
    inversa=np.zeros(np.shape(matriz))
    det=determinante(matriz)%modulo
    if det !=0:
        adj=adjunto(matriz)%modulo
        for i in range(len(matriz)):
            for j in range(len(matriz[i])):
                inversa[i][j]=(adj[i][j]/det)#%modulo

    return inversa%modulo #esto puede que no sea necesario porque ya estamos en matemát
```

In [27]:

```
#comprobación de la función
matriz = [[11,8], [3,7]]
print(inversa(matriz,26))
```

```
[[ 7. 18.]
 [23. 11.]]
```

Función de cifrado del algoritmo.

cifrar{matriz_numerica}{matriz}{mod}{n}

Parámetros:

- matriz_numerica: el texto a cifrar en formato matriz de numeros
- matriz: matriz de transformación
- mod: modulo en el que trabajamos
- n: dimensión

In [28]:

```
def cifrar(matriz_numerica, matriz,mod,n):

    matriz_cifrada=[]
    for i in range(len(matriz_numerica)):
        cadena_cifrada= (np.dot(matriz_numerica[i],matriz))%mod #utilizamos la función
        matriz_cifrada.append(cadena_cifrada)

    return matriz_cifrada
```

Función de descifrado del algoritmo.

descifrar{matriz_cifrada}{matriz}{mod}{n}

Parámetros:

- matriz_cifrada: el cifrado a descifrar en formato matriz de numeros
- matriz: matriz de transformación
- mod: modulo en el que trabajamos
- n: dimensión

In [29]:

```
def descifrar(matriz_cifrada, matriz,mod,n):

    inv=inversa(matriz,mod)
    matriz_descifrada=[]
    for i in range(len(matriz_cifrada)):
        cadena_descifrada= (np.dot(matriz_cifrada[i],inv))%mod #utilizamos la funcion d
        matriz_descifrada.append(cadena_descifrada)

    return matriz_descifrada
```

El siguiente programa implementa el método hill.

Llamada a la función:

hill {-C|-D} {-m |Zm|} {-n NK} {-k f ileK} [-i f ilein] [-o f ileout]

Los parámetros introducidos en este caso son:

- m cardinalidad de Zm
- n dimensión de la matriz de transformación
- k fichero que contiene la matriz de transformación

In [30]:

```
def hill(modo,mod,n,k,i=0,o=0):
    alfabeto='abcdefghijklmnopqrstuvwxyz'

    #leemos la matriz de transformación del archivo y la guardamos
    with open(k,'r') as f:
        datos = ''.join(f.readlines()).replace('\n',';')
    matriz = np.matrix(datos).tolist()
    f.close()

    #cálculamos el determinante de la matriz
    det=np.linalg.det(matriz)
    #comprobamos que la matriz K tiene una función biyectiva
    if algoritmo_euclides(int(det),mod)==1:

        if modo=="-C":

            cadena=read_input(i)
            #Traducimos los caracteres a números
            cadena_numerica=[]
            for k in cadena:
                if k in alfabeto:
                    cadena_numerica.append(alfabeto.index(k))

            # Dividimos en bloques de n elementos el texto
            # Si m no es múltiplo de n se añade padding
            m=len(cadena_numerica)/n
            maxi=len(cadena_numerica)

            matriz_numerica=np.zeros((math.ceil(m),n))

            pos=0
            for i in range(math.ceil(m)):
                for j in range(n):
                    if pos<maxi:
                        matriz_numerica[i][j]=cadena_numerica[pos]
                        pos=pos+1

            #ciframos cadena a cadena y lo guardamos en un matriz
            matriz_cifrada=cifrar(matriz_numerica,matriz,mod,n)

            #lo volvemos a pasar a caracteres
            resul=""
            for i in range(len(matriz_cifrada)):
                for j in range(len(matriz_cifrada[i])):
                    resul=resul+alfabeto[int(matriz_cifrada[i][j])]

            print_output(o,resul)


        elif modo=="-D":
            if i==0:
                cadena_cifrada=input()
                datos=[]
                for i in range(len(cadena_cifrada)):
                    if cadena_cifrada[i] in alfabeto:
                        datos.append(alfabeto.index(cadena_cifrada[i]))


            else:
```

```
                file=open(i, "r")
                cadena_cifrada=file.read()
                file.close()


                datos=[]
                for i in range(len(cadena_cifrada)):
                    if cadena_cifrada[i] in alfabeto:
                        datos.append(alfabeto.index(cadena_cifrada[i]))



            # Dividimos en bloques de n elementos el texto
            # Si m no es múltiplo de n se añade padding
            m=len(datos)/n
            maxi=len(datos)

            matriz_cifrada=np.zeros((math.ceil(m),n))

            pos=0
            for i in range(math.ceil(m)):
                for j in range(n):
                    if pos<maxi:
                        matriz_cifrada[i][j]=datos[pos]
                        pos=pos+1

            #ciframos cadena a cadena y lo guardamos en un matriz
            matriz_descifrada=descifrar(matriz_cifrada, matriz,mod,n)

            resul=""
            for i in range(len(matriz_descifrada)):
                for j in range(len(matriz_descifrada[i])):

                    if matriz_descifrada[i][j]<0:
                        matriz_descifrada[i][j]=mod+matriz_descifrada[i][j]

                    resul=resul+alfabeto[int(matriz_descifrada[i][j])]


            print_output(o,resul)

    else:
        print("{} y {} no son primos relativos. Error".format(det,mod))
```

In [112]:

```
k = [[11,8], [3,7]]
hill("-D",26, 2,"matriz_k.txt","matriz_cifrada.txt" ,0)
```

```
Cadena: holaquetal
```

In [ ]:

```
hill("-C",26,2, "matriz_k.txt", 0,"resulta_hill.txt" )
```

In [114]:

```
1 hill("-D",26, 2,"matriz_k.txt","resulta_hill.txt",0)
```

```
Cadena: niversidadutonomadeadrid
```

In [115]:

```
1 hill("-D",26,2, "matriz_k.txt" )
```

```
Cadena:
```

2.b Método de Vigenere

El siguiente programa implementa el método de Vigenere.

Llamada a la función:

vigenere {-C|-D} {-k clave} [-i filein] [-o fileout]

El parámetro *k* es cadena de caracteres usada como clave. Consideramos que la clave está formada por caracteres de nuestro alfabeto. Puede ser una frase, ya que se eliminarán los espacios. Al igual que hacemos con el input, la clave se traducirá de caracteres a números.

Como el cifrado de Vigenere es un cifrado de bloques, entonces dividimos el input en bloques de n (longitud de la clave) elementos. Si la longitud del input no es múltiplo de la longitud de la clave, añadimos padding de ceros al final del último bloque para que todos los bloques tengan n elementos. Tras este proceso hemos obtenido una matriz de n columnas, en la que cada fila es un bloque.

Realizamos el cifrado de Vigenere a la matriz. Ciframos bloque a bloque recorriendo las filas de la matriz: al elemento i-ésimo del bloque se le suma el elemento i-ésimo de la clave y al resultado se le aplica el módulo de la longitud del alfabeto. Esta operación se realiza para cada elemento del bloque (i de 1 a n). Como resultado, obtenemos una matriz de bloques cifrados. A continuación concatenamos esta matriz para obtener la cadena cifrada resultante.

A la hora de descifrar, el procesamiento es el mismo pero la operación que se realiza a la hora de descifrar bloque a bloque es diferente, naturalmente. Para cada elemento i-ésimo del bloque, se resta el elemento i-ésimo de la clave y al resultado se le aplica el módulo de la longitud del alfabeto.

In [31]:

```
def vigenere(modo,k,i=0,o=0):
    alfabeto='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
    base=len(alfabeto)

    #Traducimos la clave de caracteres a números
    k_numerica=[]
    for j in k:
        if j in alfabeto:
            k_numerica.append(alfabeto.index(j))
    n=len(k_numerica)
    if modo=="-C":
        cadena=read_input(i)
        # Traducimos los caracteres a números
        cadena_numerica=[]
        for k in cadena:
            if k in alfabeto:
                cadena_numerica.append(alfabeto.index(k))
        # Dividimos en bloques de n elementos el input
        # Si m no es múltiplo de n se añade padding
        m=len(cadena_numerica)/n
        maxi=len(cadena_numerica)
        matriz_numerica=np.zeros((math.ceil(m),n))
        pos=0
        for i in range(math.ceil(m)):
            for j in range(n):
                if pos<maxi:
                    matriz_numerica[i][j]=cadena_numerica[pos]
                    pos=pos+1
        # Tenemos una matriz que tenemos que cifrar.
        # Cada bloque es una fila de la matriz
        filas=matriz_numerica.shape[0]
        elementos=matriz_numerica.shape[1]
        matriz_cifrada=np.zeros((filas,elementos))
        for i in range(filas):
            for j in range(elementos):
                matriz_cifrada[i][j]=(matriz_numerica[i][j]+k_numerica[j])%base

        cadena_cifrada=np.concatenate(matriz_cifrada)
        resul=""
        for i in cadena_cifrada:
            resul=resul+alfabeto[int(i)]
        print_output(o,resul)
    elif modo=="-D":
        cadena_cifrada_texto=read_input(i)
        cadena_cifrada=[]
        for k in cadena_cifrada_texto:
            if k in alfabeto:
                cadena_cifrada.append(alfabeto.index(k))
        # Dividimos en bloques de n elementos el texto cifrado
        # Si m no es múltiplo de n se añade padding
        m=len(cadena_cifrada)/n
        maxi=len(cadena_cifrada)
        matriz_cifrada=np.zeros((math.ceil(m),n))
        pos=0
        for i in range(math.ceil(m)):
            for j in range(n):
                if pos<maxi:
                    matriz_cifrada[i][j]=cadena_cifrada[pos]
                    pos=pos+1
```

```
        # Tenemos una matriz que tenemos que descifrar.
        # Cada bloque es una fila de la matriz
        filas=matriz_cifrada.shape[0]
        elementos=matriz_cifrada.shape[1]
        matriz_descifrada=np.zeros((filas,elementos))
        for i in range(filas):
            for j in range(elementos):
                matriz_descifrada[i][j]=(matriz_cifrada[i][j]-k_numerica[j])%base
        cadena_descifrada=np.concatenate(matriz_descifrada)
        cadena_texto=[]
        for i in range(len(cadena_descifrada)):
            cadena_texto.append(alfabeto[int(cadena_descifrada[i])])
        print_output(o,cadena_texto)
```

In [117]:

```
vigenere("-C", "clave")
```

```
Universidad Autonoma de Madrid
Cadena: Universidad Autonoma de Madrid
Cadena: WyiQitDiyefLuOspzmvhgXayvkoave
```

In [118]:

```
vigenere("-D", "clave")
```

```
WyiQitDiyefLuOspzmvhgXayvkoave
Cadena: WyiQitDiyefLuOspzmvhgXayvkoave
Cadena: ['U', 'n', 'i', 'v', 'e', 'r', 's', 'i', 'd', 'a', 'd', 'A', 'u',
't', 'o', 'n', 'o', 'm', 'a', 'd', 'e', 'M', 'a', 'd', 'r', 'i', 'd', 'a',
'a', 'a']
```

In [119]:

```
vigenere("-C", "probamos con otra clave mas larga")
```

```
Universidad Autonoma de Madrid
Cadena: Universidad Autonoma de Madrid
Cadena: jEwweDGAfoqONKopzmvhqMsorzja
```

In [120]:

```
vigenere("-D", "probamos con otra clave mas larga")
```

```
jEwweDGAfoqONKopzmvhqMsorzja
Cadena: jEwweDGAfoqONKopzmvhqMsorzja
Cadena: ['U', 'n', 'i', 'v', 'e', 'r', 's', 'i', 'd', 'a', 'd', 'A', 'u',
't', 'o', 'n', 'o', 'm', 'a', 'd', 'e', 'M', 'a', 'd', 'r', 'i', 'd', 'a']
```

In [127]:

```
vigenere("-C", "clave", "cadena.txt", "resultado_vigenere.txt")
```

```
Cadena: Hola Nueva York!
```

In [129]:

```
vigenere("-D", "clave", "resultado_vigenere.txt", "cadena_descifradaVigenere.txt")
```

```
Cadena: J z l v R w p v v c q C k v e
```

También probamos el cifrado de Vigenere para cifrar el Quijote con la clave "En un lugar de la Mancha, de cuyo nombre no quiero acordarme".

In [123]:

```
vigenere("-C","En un lugar de la Mancha, de cuyo nombre no quiero acordarme", "quijote.txt
```

In [124]:

```
vigenere("-D", "En un lugar de la Mancha, de cuyo nombre no quiero acordarme", "quijote
```

2.c Criptoanálisis del cifrado de Vigenere

Este apartado ha sido completado y corregido

```
El primer paso para criptoanalizar el cifrado de vigenere es calcular el tamaño de la clave. Para ello hay dos opciones:

1. Test de Kasiski: se trata de buscar repeticiones de conjuntos de caracteres en el texto y despues medir la distancia entre ellas.

2. Indice de coincidencia: este índice determina la probabilidad de que dos caracteres en una cadena o en un texto sean el mismo.

Lo más interesante a destacar del uso de este índice es que si una cadena o texto descifrado posee un IC parecido al de su idioma, significa que la clave propuesta es la original del cifrado.

```

El siguiente programa implementa el indice de coincidencia.

Llamada a la función: **IC {-l Ngrama} [-i filein] [-o fileout]**

- l longitud de n-grama buscado

Test de Kasiski

El siguiente programa implementa el test de kasiski.

Llamada a la función: **kasiski {lista}{-tam conjunto_caracteres} [-i f ilein] [-o f ileout]**

- tam longitud del conjunto, 3 por defecto

Consiste en buscar conjuntos del tamaño especificado y guardamos la distancia a la repetición, es decir, la diferencia entre la posición del primer caracter del conjunto y la primera posición del primer caracter de la repetición. Si obtenemos repeticiones de un conjunto, calculamos el máximo común divisor de las distancias de las repeticiones y éste valor es muy probablemente la longitud de la clave con la que se ha cifrado el texto por el cifrado de Vigenere.

In [104]:

```
def test_kasiski(lista, tam=3, i=0, o=0):
    from collections import Counter

    res=0
    conj_dist=[]
    conjuntos=[]
    #buscamos conjuntos que se repitan en el texto y la distancia entre ellos
    i = 0
    while i < len(lista):
        conj= lista[i:i+tam] # Cogemos al menos 3 caracteres como tamaño de la tupla
        t = len(conj)
        if t == tam: #tiene que ser t, si no estamos al final de la lista
            print("Conjunto a buscar es: {}".format(conj))
            flag_repes=0
            distancias=[]
            for j in range(i+1,len(lista)):

                if lista[i:i+t] == lista[j:j+t]: #Si coinciden, seguimos comprobando
                    flag_repes=1
                    while lista[i:i+t] == lista[j:j+t]:
                        t = t + 1
                    t = t -1
                    conj = lista[i:i+t] #Ahora tenemos un conjunto que sabemos que se r
                    conjuntos.append(conj)
                    dist = j - i #calculamos la distancia
                    print("Distancia: {}".format(dist))
                    conj_dist.append([conj,dist])
                    distancias.append(dist)
                    j = j + t + 1
            if flag_repes==1 and len(distancias)>1:
                # Calculamos mcd de distancias
                print("Distancias {}".format(distancias))
                res=mcd_lista(distancias)
                if res != 1:
                    print("Posible tamaño de la clave: {}".format(res))
                    return res
            i = i + t -tam +1
        else:
            i = i + 1

    if res==0:
        print("No se ha encontrado n tamaño de la clave")
        return 0

    return res

```

Función que cálcula el mcd de todos los elementos de una lista.

mcd_lista {lista}

In [33]:

```
def mcd_lista(lista):
    mcd=lista[0]
    for i in range(1,len(lista)):
        mcd=math.gcd(mcd,lista[i])
    return mcd
```

In [34]:

```
def calcular_divisores(n):
    lista = []
    for i in range(2,n):
        if n % i == 0:
            lista.append(i)
    return lista
```

In [127]:

```
vigenere("-C", "be", "pruebaKasiski.txt", "kasiski.txt")
```

In [128]:

```
cifrado=read_input("kasiski.txt")
cifrado=cifrado.replace(" ", "")
print(cifrado)

print(test_kasiski(cifrado,2))
```

```
UiyxppbvhsqesedsntsscesimgjjseeseiLetmtojCfrdsoxsespbpprhmuyehfpbgmewiMssinm
qwvqesmsswjxbqfxdsowfguiuyseemqmtgjrhimmuRvrdtpvueuimpvwwmuefxvvqmtgvvtyteui
ntpvsmtyttiesiuvbRvpmenzfpmiptiesiuvbhjenzbvjytjbydmcytzfpfkfxnmEsoidwppmmdm
uyemojbydmcytzfpjxjhumogjhvruWfhuvjwumryfvjwvwwmuefiseuwphbpfwbxwymtvxbxfeox
fgprtidxfxvvQlbwfpmytuvmtiomneoxfRvpmenediomnrfgniuytimijjfrejbydmcytiviujfp
jw
Conjunto a buscar es: Ui
Conjunto a buscar es: iy
Conjunto a buscar es: yx
Conjunto a buscar es: xp
Conjunto a buscar es: pp
Distancia: 49
Distancia: 218
Distancias [49, 218]
Conjunto a buscar es: pb
Distancia: 46
Distancia: 58
Distancias [46, 58]
Posible tamaño de la clave: 2
2
```

In [129]:

```
vigenere("-C", "cac", "pruebaKasiski.txt", "kasiski.txt")
```

In [130]:

```
cifrado=read_input("kasiski.txt")
cifrado=cifrado.replace(" ", "")
print(cifrado)

print(test_kasiski(cifrado,2))
```

```
VezvoncriqpctaeqmrtodcrgnckhrcfofgKcuiumiAgneqnvtatnanqniktwfdgnaenaxgLqteok
puwmfqlqtskvaogteqnugcvgtwtafkpkuckpggnivPupepqttcvennuuxivcevwrrksewruwscve
orottiuwsrjatgttcNwnlcovgnlgqpjcrgvrcficovctiwufcwckduuxengggvmkFopgcuqlnkck
vufknhcuekbwuvgnivkdvknekdwptUgdvtiuviswetkswuvkvaggrcvsqfangscvvwnpwvavgapv
eeqnugcvgtwtPjcsgnlwuqwksgpiocnvgNwnlcoaegnkongemgvuuglgkfgpdhcuekbwuewgthgl
kuac
Conjunto a buscar es: Ve
Conjunto a buscar es: ez
Conjunto a buscar es: zv
Conjunto a buscar es: vo
Conjunto a buscar es: on
Distancia: 344
Conjunto a buscar es: nc
Distancia: 19
Conjunto a buscar es: cr
Distancia: 15
Distancia: 177
Distancias [15, 177]
Posible tamaño de la clave: 3
3
```

In [131]:

```
vigenere("-C", "clave", "pruebaKasiski.txt", "kasiski.txt")
```

In [132]:

```
cifrado=read_input("kasiski.txt")
cifrado=cifrado.replace(" ", "")
print(cifrado)

print(test_kasiski(cifrado,2))
```

```
VpxOsnlrBsrlrvgqxpMsdlrzpetfMefzdzOcDiNokJeIgqytMetwaGspriOyfoeGeewaQiNzrzqk
AsPqfzlJvuttvqgEcJrupcOivFrvhkAiNgkygzpkENPreAoMxcEeGpwDvDxcptPvrtsxytDuNevp
mKstCiNyuAhvvgErvRwwlvqxplGiqAhvvgErvhklmQettuNjcFcDfwDvzpgreOqkOoIieDoGpkni
OyftnAewniwyuGeGmvtdOmpniyypESzhvCiNxkBuzvkDuNzkEazitltNsfllzwcEvPprFtvxglnO
ieznNieEeOytahvwgwlPwsFiNiptmvrvpNPpnlmvggyiHrgnmzxwDeGikqeIhhluxmdFszygEfzp
kDave
Conjunto a buscar es: Vp
Conjunto a buscar es: px
Conjunto a buscar es: xO
Conjunto a buscar es: Os
Conjunto a buscar es: sn
Conjunto a buscar es: nl
Distancia: 335
Conjunto a buscar es: lr
Distancia: 5
Distancia: 15
Distancias [5, 15]
Posible tamaño de la clave: 5
5
```

In [133]:

```
vigenere("-C", "Mancha", "quijote.txt", "quijote_cifradoVigenere_kasiski.txt")
```

In [138]:

```
cifrado=read_input("quijote_cifradoVigenere_kasiski.txt")
cifrado=cifrado.replace(" ", "")
print(test_kasiski(cifrado,3))
```

```
Conjunto a buscar es: pOa
Distancia: 779022
Conjunto a buscar es: tAc
Distancia: 229441
Distancia: 307633
Distancia: 339361
Distancias [229441, 307633, 339361]
Conjunto a buscar es: AcT
Conjunto a buscar es: cTP
Conjunto a buscar es: TPM
Conjunto a buscar es: PMq
Conjunto a buscar es: MqR
Conjunto a buscar es: qRN
Conjunto a buscar es: RNR
Conjunto a buscar es: NRH
Conjunto a buscar es: RHR
Conjunto a buscar es: HRF
Conjunto a buscar es: RFE
Conjunto a buscar es: FEZ
```

In [139]:

```
cifrado=read_input("quijote_cifradoVigenere_kasiski.txt")
cifrado=cifrado.replace(" ", "")
print(test_kasiski(cifrado,6))
```

```
Conjunto a buscar es: pOaSbI
Distancia: 779022
Conjunto a buscar es: CUCtAc
Conjunto a buscar es: UCtAcT
Conjunto a buscar es: CtAcTP
Conjunto a buscar es: tAcTPM
Conjunto a buscar es: AcTPMq
Conjunto a buscar es: cTPMqR
Conjunto a buscar es: TPMqRN
Conjunto a buscar es: PMqRNR
Conjunto a buscar es: MqRNRH
Conjunto a buscar es: qRNRHR
Conjunto a buscar es: RNRHRF
Conjunto a buscar es: NRHRFE
Conjunto a buscar es: RHRFEZ
Conjunto a buscar es: HRFEZk
Conjunto a buscar es: RFEZkn
Conjunto a buscar es: FEZknu
Conjunto a buscar es: EZknuO
```

Podemos observar que obtenemos el tamaño de clave con el que se ha cifrado el Quijote y tomando diferentes tamaños de agrupaciones (3 y 6) obtenemos que el tamaño de clave es el correcto: 6 (Mancha).

Indice de coincidencia

En general, el algoritmo consiste en iterar varias veces en busca de un tamaño n de clave correcto para un texto que nos dan cifrado.

Este texto se va a dividir en bloques iguales al de clave propuesta, en este caso (n). En cada iteración, y luego, se van a coger de cada bloque los caracteres cuyas posiciones puedan coincidir con la subclave i de esta clave de tamaño n (como se ha visto en teoría).

Cuabdo tenemos los n vectores, cálculamos sus IC. Si utilizando el conteo de frecuencias de sus caracteres, se aproximan al IC del idioma utilizado, esto implica que el tamaño de clave n propuesto es el correcto para la clave original del cifrado.

Esto ocurre porque se trata de un cifrado por desplazamiento, por lo que las frecuencias de los caracteres permanece igual en el estado de cifrado y de descifrado.

Función que divide una lista siguiendo el esquema dado en clase para poder criptoanalizar vigenere.

dividir_lista {n}{lista}

In [140]:

```
def dividir_lista(n,lista): #ahora que sabemos la cardinalidad de la llave, dividimos l
    dic = {}
    for elem in range(n): #hacemos sublistas de n elementos
        dic[elem] = []

    i = 0
    for j in range (len(lista)):
        if i == n: #si el inidice es igual a n hemos llegado al final de la sublista
            i = 0
        dic[i].append(lista[j])
        i = i + 1
    return dic
```

Función que calcula la probabilidad de que dos caracteres de una cadena sean el mismo.

calcular_IC {lista}

In [141]:

```
def calcular_IC(lista):
    #print(lista)
    alfabeto='abcdefghijklmnopqrstuvwxyz'
     #esta funcion esta mal
    castellano=(['a', 11.96],['b', 0.92],['c', 2.92],['d', 6.87],['e', 16.78],['f', 0.5
           ['h', 0.89],['i', 4.15],['j', 0.3],['k', 0.0],['l', 8.37],['m', 2.12],['n',
           ['o', 8.69],['p', 2.77],['q', 1.53],['r', 4.94],['s', 7.88],['t', 3.31],['u'
           ['v', 0.39],['w', 0.0],['x', 0.06],['y', 1.54],['z', 0.15])

    from collections import Counter
    resul_contador = Counter(lista)

    frecuencias=[]
    for i in range(len(alfabeto)):
        frecuencias.append(resul_contador[alfabeto[i]])

    n_pares_iguales=0
    for i in range(len(frecuencias)):
        n_pares_iguales=n_pares_iguales+(frecuencias[i]*frecuencias[i]-1)
         #n_pares_iguales=castellano[i][1]*castellano[i][1]
    #n_pares_letras=(len(lista)*len(lista)-1)/2
    n_pares_letras=len(lista)*(len(lista)-1)
    #frecuencias=[]
    #el número de casos posibles en los que podemos elegir dos caracteres
    #iguales entre un total de m caracteres del alfabeto
    #for j in range(len(resul_contador)): #vamos a calcular la frecuencia de cada carac
            #frecuencias.append(resul_contador[j])

    #print(resul_contador)
    #print(frecuencias)
    #castellano_ord=sorted(castellano, key=lambda letra: letra[1], reverse=True)
    IC=n_pares_iguales/n_pares_letras
    #for i in range(len(lista)):
       # for j in range(len(castellano)):
           # if lista[i]==resul_contador[j][0]:
           #     IC= IC+(resul_contador[j][1]*resul_contador[j][1])
    return IC
```

El siguiente programa implementa el indice de coincidencia.

Llamada a la función: **IC {lista}{-maxi maximo de iteraciones} [-i filein] [-o fileout]**

In [201]:

```
def IC(cadena,maxi, i=0, o=0):
    from collections import Counter
    #INFORMACIÓN SOBRE EL IC DEL CASTELLANO Y EL ÍNGLES
    castellano=(['a', 11.96],['b', 0.92],['c', 2.92],['d', 6.87],['e', 16.78],['f', 0.5
           ['h', 0.89],['i', 4.15],['j', 0.3],['k', 0.0],['l', 8.37],['m', 2.12],['n',
           ['o', 8.69],['p', 2.77],['q', 1.53],['r', 4.94],['s', 7.88],['t', 3.31],['u'
           ['v', 0.39],['w', 0.0],['x', 0.06],['y', 1.54],['z', 0.15])

    castellano_ord=sorted(castellano, key=lambda letra: letra[1], reverse=True)
    IC_castellano=0
    for i in range(len(castellano_ord)):
        IC_castellano= IC_castellano+castellano_ord[i][1]*castellano_ord[i][1]
    IC_castellano=0.073
    #print("IC castellano {}".format(IC_castellano))

    ingles=(['a', 8.04],['b', 1.54],['c', 3.06],['d', 3.99],['e', 12.51],['f', 2.30],['
        ['h', 5.49],['i', 7.26],['j', 0.16],['k', 0.67],['l', 4.14],['m', 2.53],['n', 7
       ['o', 7.60],['p', 2.0],['q', 0.11],['r', 6.12],['s', 6,54],['t', 9.25],['u', 2.7
       ['v', 0.99],['w', 1.92],['x', 0.19],['y', 1.73],['z', 0.19])

    ingles_ord=sorted(ingles, key=lambda letra: letra[1], reverse=True)
    IC_ingles=0
    for i in range(len(ingles)):
        IC_ingles= IC_ingles+(ingles[i][1]*ingles[i][1])
    IC_ingles=IC_ingles/10000
    # print("IC inglés {}".format(IC_ingles))

    IC_aleatorio=0.038
    #probamos con 18 tamaños diferentes (ponemos un máximo para que sea mas eficiente/p
    sublista=[]
    tamaño_clave=[]
    #n es el tamaño de la clave propuesta
    for n in range(1,maxi):
        print("n = {}".format(n))
        dic=dividir_lista(n,cadena) #dividimos el texto en sublistas del tamaño de la c
        suma=0
        for j in range(len(dic)):
            ic=calcular_IC(dic[j])
            print("Vector {} IC= {}".format(j,ic))
            suma=suma+ic
        media=suma/n
        print("n = {}, media de IC = {}".format(n,media))
        #Si n es el tamaño de bloque con el que se ha cifrado, todos estos vectores tie
        #y por tanto su coincidencia es diferente al de un lenguaje aleatorio y casi la
        #en este caso
        if media<=IC_castellano+0.005 and media>=IC_castellano-0.005:
            print("Posible tamaño de clave encontrado: n = {} con IC = {}".format(n, me
            sublista.append(dic)
            tamaño_clave.append(n)


    return sublista, tamaño_clave

```

In [170]:

```
lista="jkceizwpsovcdmzvepdzwvlmwmpnmjinpmkitldjvxfegzgwaniuaaghcdyyilldzwrpccefzayspli
lista,n=IC(lista,18)
```

```
n = 17
Vector 0 IC= 0.055379746835443035
Vector 1 IC= 0.05411392405063291
Vector 2 IC= 0.0670886075949367
Vector 3 IC= 0.04852320675105485
Vector 4 IC= 0.05306718597857838
Vector 5 IC= 0.052742616033755275
Vector 6 IC= 0.061506004543979226
Vector 7 IC= 0.04787406686140863
Vector 8 IC= 0.05663745537163259
Vector 9 IC= 0.05533917559234015
Vector 10 IC= 0.049172346640701074
Vector 11 IC= 0.049172346640701074
Vector 12 IC= 0.04982148653034729
Vector 13 IC= 0.06312885426809478
Vector 14 IC= 0.05111976630963973
Vector 15 IC= 0.052742616033755275
Vector 16 IC= 0.05111976630963973
n = 17, media de IC = 0.05403230425568475
```

In [202]:

```
vigenere("-C", "be", "pruebaKasiski.txt", "ic.txt")
```

In [203]:

```
cifrado=read_input("ic.txt")
cifrado=cifrado.replace(" ", "")
cifrado=cifrado.lower()
print("Texto cifrado a analizar por índice de coincidencia: {}".format(cifrado))
sublista, n=IC(cifrado,10)
```

```
Texto cifrado a analizar por índice de coincidencia: uiyxppbvhsqesedsntssces
imgjjseeseiletmtojcfrdsoxsespbpprhmuyehfpbgmewimssinmqwvqesmsswjxbqfxdsowfgu
iuyseemqmtgjrhimmurvrdtpvueuimpvwwmuefxvvqmtgvvtyteuintpvsmtyttiesiuvbrvpmen
zfpmiptiesiuvbhjenzbvjytjbydmcytzfpfkfxnmesoidwppmmdmuyemojbydmcytzfpjxjhumo
gjhvruwfhuvjwumryfvjwvwwmuefiseuwphbpfwbxwymtvxbxfeoxfgprtidxfxvvqlbwfpmytuv
mtiomneoxfrvpmenediomnrfgniuytimijjfrejbydmcytiviujfpjw
n = 1
Vector 0 IC= 0.05256214700911077
n = 1, media de IC = 0.05256214700911077
n = 2
Vector 0 IC= 0.06858638743455497
Vector 1 IC= 0.07547533755855608
n = 2, media de IC = 0.07203086249655552
Posible tamaño de clave encontrado: n = 2 con IC = 0.07203086249655552
n = 3
Vector 0 IC= 0.06532972440944881
Vector 1 IC= 0.04918135233095863
Vector 2 IC= 0.05580552430946132
n = 3, media de IC = 0.05677220034995625
n = 4
Vector 0 IC= 0.07543859649122807
Vector 1 IC= 0.07609649122807018
Vector 2 IC= 0.06998880179171332
Vector 3 IC= 0.07849944008958566
n = 4, media de IC = 0.07500583240014931
Posible tamaño de clave encontrado: n = 4 con IC = 0.07500583240014931
n = 5
Vector 0 IC= 0.05827067669172932
Vector 1 IC= 0.05109364319890636
Vector 2 IC= 0.06070175438596491
Vector 3 IC= 0.06
Vector 4 IC= 0.05754385964912281
n = 5, media de IC = 0.057521986785144676
n = 6
Vector 0 IC= 0.0873015873015873
Vector 1 IC= 0.06646825396825397
Vector 2 IC= 0.06994047619047619
Vector 3 IC= 0.09325396825396826
Vector 4 IC= 0.0647721454173067
Vector 5 IC= 0.08269329237071173
n = 6, media de IC = 0.07740495391705068
Posible tamaño de clave encontrado: n = 6 con IC = 0.07740495391705068
n = 7
Vector 0 IC= 0.0643097643097643
Vector 1 IC= 0.05218855218855219
Vector 2 IC= 0.06228956228956229
Vector 3 IC= 0.054208754208754206
Vector 4 IC= 0.05660377358490566
Vector 5 IC= 0.06079664570230608
Vector 6 IC= 0.06079664570230608
n = 7, media de IC = 0.0587419568551644
n = 8
```

```
Vector 0 IC= 0.08599290780141844
Vector 1 IC= 0.07712765957446809
Vector 2 IC= 0.07092198581560284
Vector 3 IC= 0.09308510638297872
Vector 4 IC= 0.07624113475177305
Vector 5 IC= 0.09131205673758866
Vector 6 IC= 0.07076780758556891
Vector 7 IC= 0.06706753006475485
n = 8, media de IC = 0.0790645235892692
n = 9
Vector 0 IC= 0.07032115171650055
Vector 1 IC= 0.044850498338870434
Vector 2 IC= 0.05370985603543743
Vector 3 IC= 0.06478405315614617
Vector 4 IC= 0.059233449477351915
Vector 5 IC= 0.06039488966318235
Vector 6 IC= 0.0743321718931475
Vector 7 IC= 0.0627177700348432
Vector 8 IC= 0.056910569105691054
n = 9, media de IC = 0.06080604549124118
```

In [204]:

```
1 vigenere("-C", "cac", "pruebaKasiski.txt", "ic.txt")
2
3 cifrado=read_input("ic.txt")
4 cifrado=cifrado.replace(" ", "")
5 cifrado=cifrado.lower()
6 sublista, n=IC(cifrado,10)
```

```
Vector 2 IC= 0.07092198581560284
Vector 3 IC= 0.06826241134751773
Vector 4 IC= 0.06382978723404255
Vector 5 IC= 0.0673758865248227
Vector 6 IC= 0.051418439716312055
Vector 7 IC= 0.05673758865248227
n = 8, media de IC = 0.06471631205673758
n = 9
Vector 0 IC= 0.0725359911406423
Vector 1 IC= 0.0592469545957918
Vector 2 IC= 0.0769656699889258
Vector 3 IC= 0.08693244739756367
Vector 4 IC= 0.0681063122923588
Vector 5 IC= 0.08361018826135105
Vector 6 IC= 0.10104529616724739
Vector 7 IC= 0.06968641114982578
Vector 8 IC= 0.07665505226480836
n = 9, media de IC = 0.077198258139835
Posible tamaño de clave encontrado: n = 9 con IC = 0.077198258139835
```

In [206]:

```
vigenere("-C", "Mancha", "quijote.txt", "quijote_cifradoVigenere_IC.txt")
cifrado=read_input("quijote_cifradoVigenere_IC.txt")
cifrado=cifrado.replace(" ", "")
cifrado=cifrado.lower()
sublista,n=IC(cifrado,20)
```

```
n = 1
Vector 0 IC= 0.04539523511253747
n = 1, media de IC = 0.04539523511253747
n = 2
Vector 0 IC= 0.046452277328584826
Vector 1 IC= 0.056951620829083574
n = 2, media de IC = 0.0517019490788342
n = 3
Vector 0 IC= 0.05836630171699936
Vector 1 IC= 0.052497294815661295
Vector 2 IC= 0.06060676209095659
n = 3, media de IC = 0.05715678620787241
n = 4
Vector 0 IC= 0.04646505491077915
Vector 1 IC= 0.05698753026647876
Vector 2 IC= 0.046443459317309685
Vector 3 IC= 0.05691913894924689
n = 4, media de IC = 0.051703795860953625
n = 5
```

En los ejemplos de ejecución anteriores, podemos observar que el índice de coincidencia averigua el tamaño de clave correcto en cada caso

Esta función cálcula la clave de vigenere. Habiendo cálculado antes el tamaño de la clave con el test de kasiski o e IC (ambas funciones programadas al principio de este apartado).

En esta función unificamos todas las funciones implementadas anteriormente en este apartado. Primero hallamos la longitud de la clave con el test de Kasiski y el índice de coincidencia y comprobamos que coinciden para tener más seguridad de que esa es la longitud de la clave a buscar, denotada n. Después dividimos el texto cifrado en n vectores y para cada vector calculamos la frecuencia de cada caracter del alfabeto.

Calculamos M(k_i)=n/l * sum (P_j * f_{j-k_i}) donde

- n/l=1/(longitud del vector)
- P_j es la probabilidad del símbolo j-ésimo de la lengua española
- f_{j-k_i} es la frecuencia en el vector del caracter j-k_i - ésimo del alfabeto

Aquel símbolo k del vector cuyo valor M(k) es más próximo al índice de coincidencia del español es un posible caracter de la clave.

La llamada a la función es la siguiente: **criptoanalisis_vigenere {lista}**

- lista es la cadena cifrada por Vigenere de la cuál queremos encontrar la clave de cifrado

In [336]:

```
def criptoanalisis_vigenere(lista):
    from collections import Counter
    alfabeto='abcdefghijklmnopqrstuvwxyz'
    maxi=18 #la cantidad de intentos que queramos que haga como maximo
    conj=4 #la longitud de los conjuntos que queremos que compruebe
    castellano=(['a', 11.96],['b', 0.92],['c', 2.92],['d', 6.87],['e', 16.78],['f', 0.5
           ['h', 0.89],['i', 4.15],['j', 0.3],['k', 0.0],['l', 8.37],['m', 2.12],['n',
           ['o', 8.69],['p', 2.77],['q', 1.53],['r', 4.94],['s', 7.88],['t', 3.31],['u'
           ['v', 0.39],['w', 0.0],['x', 0.06],['y', 1.54],['z', 0.15])
    cadena_cifrada=lista
    print("Test de Kasiski")
    n_kasiski=test_kasiski(cadena_cifrada,conj)
    lista_ic=lista.lower()
    print("\n Índice de coincidencia")
    lista,n= IC(lista_ic,maxi)
    sublista=lista[0]
    n_ic=n[0]


    print("\nTamaño de clave según IC: {}".format(n_ic))
    print("Tamaño de clave según test de Kasiski: {}".format(n_kasiski))
    if n_ic==n_kasiski:
        print("el tamaño esta comprobado\n")
    else:
        print("puede que el tamaño escogido sea incorrecto\n")

    n=n_ic


    #ahora que ya tenemos el tamaño de la clave, vamos a esimar la clave
    #de cifrado
    print("Hallamos la clave de cifrado\n")

    IC_castellano=0.073

    dic=dividir_lista(n,cadena_cifrada.lower())

    clave=[]

    for i in range(len(dic)):
        print("Vector {}".format(i))
        resul_contador = Counter(dic[i])
        frecuencias=[]

        # Hallamos las frecuencias de los caracteres del vector
        for s in range(len(alfabeto)):
            frecuencias.append(resul_contador[alfabeto[s]])


        for k in range(len(castellano)):
            ic=0
            for m in range(len(castellano)):
                ic=ic+((castellano[m][1]*0.01)*(frecuencias[m-k]))
            ic=ic*(1/len(dic[i]))

            if ic<=IC_castellano+0.005 and ic>=IC_castellano-0.005:
                print("Posible k_{} = {} con IC = {}".format(i,alfabeto[k],ic))
                clave.append(alfabeto[k])

```

```
60      print("Clave: {}".format(clave))
61
```

Vamos a probar el algoritmo para hallar la clave del Quijote que hemos realizado antes con la clave **mancha**

In [338]:

```
cifrado=read_input("quijote_criptoanalisisVigenere.txt")
cifrado=cifrado.replace(" ", "")
criptoanalisis_vigenere(cifrado)
```

```
Test de Kasiski
Conjunto a buscar es: ROao
Distancia: 779022
Conjunto a buscar es: CVAc
Conjunto a buscar es: VAcp
Conjunto a buscar es: Acpb
Conjunto a buscar es: cpbM
Conjunto a buscar es: pbMS
Conjunto a buscar es: bMSR
Conjunto a buscar es: MSRN
Conjunto a buscar es: SRNn
Conjunto a buscar es: RNnT
Conjunto a buscar es: NnTR
Conjunto a buscar es: nTRh
Conjunto a buscar es: TRhE
Conjunto a buscar es: RhEZ
Conjunto a buscar es: hEZG
Conjunto a buscar es: EZGz
Conjunto a buscar es: ZGzu
Conjunto a buscar es: Gzus
Distancia: 46932
Distancia: 52116
Distancia: 157386
Distancia: 184140
Distancia: 206772
Distancia: 247626
Distancia: 480852
Distancia: 540648
Distancias [46932, 52116, 157386, 184140, 206772, 247626, 480852, 540648]
Posible tamaño de la clave: 6

 Índice de coincidencia
n = 1
Vector 0 IC= 0.04540763982455335
n = 1, media de IC = 0.04540763982455335
n = 2
Vector 0 IC= 0.04652155304287341
Vector 1 IC= 0.056912403414830205
n = 2, media de IC = 0.05171697822885181
n = 3
Vector 0 IC= 0.058326836417098606
Vector 1 IC= 0.052565588800220574
Vector 2 IC= 0.06060676209095659
n = 3, media de IC = 0.057166395769425254
n = 4
Vector 0 IC= 0.04644416329036276
Vector 1 IC= 0.05695319199624351
Vector 2 IC= 0.04660250931482768
Vector 3 IC= 0.056874712929480105
n = 4, media de IC = 0.05171864438272851
n = 5
Vector 0 IC= 0.04547160943194117
Vector 1 IC= 0.04534120133666053
Vector 2 IC= 0.04538471101068617
```

```
Vector 3 IC= 0.045334443181469605
Vector 4 IC= 0.045531252033311875
n = 5, media de IC = 0.045412643398813866
n = 6
Vector 0 IC= 0.07421536941738713
Vector 1 IC= 0.07435898963451414
Vector 2 IC= 0.07454324028607684
Vector 3 IC= 0.07465037054211479
Vector 4 IC= 0.0741558446492256
Vector 5 IC= 0.07468091058303544
n = 6, media de IC = 0.07443412085205899
Posible tamaño de clave encontrado: n = 6 con IC = 0.07443412085205899
n = 7
Vector 0 IC= 0.045407931944058434
Vector 1 IC= 0.04535896461299036
Vector 2 IC= 0.04544735603082865
Vector 3 IC= 0.04550524400404284
Vector 4 IC= 0.04527972435683036
Vector 5 IC= 0.04551455937827802
Vector 6 IC= 0.04539056173890152
n = 7, media de IC = 0.0454149060094186
n = 8
Vector 0 IC= 0.046565214257596216
Vector 1 IC= 0.057095184697877076
Vector 2 IC= 0.04652195186839465
Vector 3 IC= 0.05696934570764565
Vector 4 IC= 0.04633394492208657
Vector 5 IC= 0.05681908387664309
Vector 6 IC= 0.04669108403997907
Vector 7 IC= 0.05679052529785521
n = 8, media de IC = 0.05172329183350969
n = 9
Vector 0 IC= 0.058403075851317196
Vector 1 IC= 0.05248212558157402
Vector 2 IC= 0.06039753869481548
Vector 3 IC= 0.058318177206679034
Vector 4 IC= 0.05268188878247862
Vector 5 IC= 0.06083399442444769
Vector 6 IC= 0.058272256514820216
Vector 7 IC= 0.05254773035696121
Vector 8 IC= 0.060607113186902616
n = 9, media de IC = 0.05717154451111068
n = 10
Vector 0 IC= 0.04650670865853126
Vector 1 IC= 0.05673135067971477
Vector 2 IC= 0.04656832510077257
Vector 3 IC= 0.056649329714631935
Vector 4 IC= 0.04657970440144782
Vector 5 IC= 0.057048183203321515
Vector 6 IC= 0.04642718020866104
Vector 7 IC= 0.056949334815091734
Vector 8 IC= 0.04657665977578829
Vector 9 IC= 0.057236542865939696
n = 10, media de IC = 0.05172733194239006
n = 11
Vector 0 IC= 0.045385974341883344
Vector 1 IC= 0.04561505334091338
Vector 2 IC= 0.04543933365707059
Vector 3 IC= 0.045478229908795596
Vector 4 IC= 0.045410932504902336
Vector 5 IC= 0.04543790516467236
```

```
Vector 6 IC= 0.04526775548718899
Vector 7 IC= 0.045523467484976425
Vector 8 IC= 0.045432430798207406
Vector 9 IC= 0.04531501902248691
Vector 10 IC= 0.04529402603094756
n = 11, media de IC = 0.04541819343109499
n = 12
Vector 0 IC= 0.07453680586351419
Vector 1 IC= 0.07439405616733817
Vector 2 IC= 0.07457480995808413
Vector 3 IC= 0.074856943032465
Vector 4 IC= 0.07378979054527347
Vector 5 IC= 0.07503816244300034
Vector 6 IC= 0.07390519733197598
Vector 7 IC= 0.07433705356004644
Vector 8 IC= 0.07452205024043183
Vector 9 IC= 0.07446074065008836
Vector 10 IC= 0.0745312759216027
Vector 11 IC= 0.0743354837427201
n = 12, media de IC = 0.07444019745471173
Posible tamaño de clave encontrado: n = 12 con IC = 0.07444019745471173
n = 13
Vector 0 IC= 0.0455485628490752
Vector 1 IC= 0.04525070257962584
Vector 2 IC= 0.045766014667531316
Vector 3 IC= 0.045478420330527174
Vector 4 IC= 0.04518729726937021
Vector 5 IC= 0.04554124987246884
Vector 6 IC= 0.04532310115864712
Vector 7 IC= 0.0455233993954613
Vector 8 IC= 0.04536878320295401
Vector 9 IC= 0.045355438209794256
Vector 10 IC= 0.045517119040797464
Vector 11 IC= 0.045447001612998056
Vector 12 IC= 0.045209082471680116
n = 13, media de IC = 0.04542432097391776
n = 14
Vector 0 IC= 0.04659636374395455
Vector 1 IC= 0.05682173806554764
Vector 2 IC= 0.0467797759814526
Vector 3 IC= 0.05684772446998195
Vector 4 IC= 0.04641820241252629
Vector 5 IC= 0.057069331179874276
Vector 6 IC= 0.04636443675214715
Vector 7 IC= 0.05704289640219994
Vector 8 IC= 0.04642546477938836
Vector 9 IC= 0.05695185840311342
Vector 10 IC= 0.04659962906222175
Vector 11 IC= 0.056815020123990825
Vector 12 IC= 0.04655878394019863
Vector 13 IC= 0.056946473882677096
n = 14, media de IC = 0.051731264228519605
n = 15
Vector 0 IC= 0.05823650231481272
Vector 1 IC= 0.0526574799924363
Vector 2 IC= 0.060624753964924345
Vector 3 IC= 0.05805976537759401
Vector 4 IC= 0.05281166264818428
Vector 5 IC= 0.06080291062380147
Vector 6 IC= 0.05828399671705567
Vector 7 IC= 0.05243797614936833
```

```
Vector 8 IC= 0.06022179454334546
Vector 9 IC= 0.058679173442468695
Vector 10 IC= 0.05271444853309064
Vector 11 IC= 0.06069137926378846
Vector 12 IC= 0.0584373251834241
Vector 13 IC= 0.05231278070476323
Vector 14 IC= 0.06077555741803204
n = 15, media de IC = 0.05718316712513931
n = 16
Vector 0 IC= 0.04656865646043197
Vector 1 IC= 0.05721415918514167
Vector 2 IC= 0.04646847436159038
Vector 3 IC= 0.05701265220357541
Vector 4 IC= 0.046416414858552794
Vector 5 IC= 0.0567416373193021
Vector 6 IC= 0.04660014752921022
Vector 7 IC= 0.056355882408949315
Vector 8 IC= 0.04657975000984248
Vector 9 IC= 0.05700580220282167
Vector 10 IC= 0.046598368003574094
Vector 11 IC= 0.056941869141726095
Vector 12 IC= 0.04626827695294296
Vector 13 IC= 0.05690832116815646
Vector 14 IC= 0.046803280057726486
Vector 15 IC= 0.0572584169526569
n = 16, media de IC = 0.051733881801012564
n = 17
Vector 0 IC= 0.04544017002685136
Vector 1 IC= 0.04534438904353626
Vector 2 IC= 0.04536333186391223
Vector 3 IC= 0.0454455630746736
Vector 4 IC= 0.04559594039129125
Vector 5 IC= 0.045448504565155395
Vector 6 IC= 0.04520229235366075
Vector 7 IC= 0.045417321928597786
Vector 8 IC= 0.04563977616600493
Vector 9 IC= 0.04559032696813709
Vector 10 IC= 0.045443724406668266
Vector 11 IC= 0.045411891993917416
Vector 12 IC= 0.04520503541965851
Vector 13 IC= 0.04544353635517262
Vector 14 IC= 0.045560459539289055
Vector 15 IC= 0.04533233088274386
Vector 16 IC= 0.04538259566926272
n = 17, media de IC = 0.04542748180285489

Tamaño de clave según IC: 6
Tamaño de clave según test de Kasiski: 6
el tamaño esta comprobado

Hallamos la clave de cifrado

Vector 0
Posible k_0 = m con IC = 0.077511257195487
Vector 1
Posible k_1 = a con IC = 0.07759426279837286
Vector 2
Posible k_2 = n con IC = 0.07767395195333489
Vector 3
Posible k_3 = c con IC = 0.07772645329649244
Vector 4
```

```
Posible k_4 = h con IC = 0.07748807736587614
Vector 5
Posible k_5 = a con IC = 0.07777413615780182
Clave: ['m', 'a', 'n', 'c', 'h', 'a']
```

Podemos observar que hemos obtenido satisfactoriamente la clave con la que hemos cifrado el quijote.

- **¿Qué sucede si el tamaño de clave se aproxima al tamaño del texto a cifrar?**

Si la clave se aproxima al tamaño a cifrar no podremos encontrar en el texto cifrado fragmentos que hayan sido cifrados con el mismo trozo de la clave, ya que no se repite la clave a lo largo del mensaje. Cada elemento se cifra con un elemento único de la clave. Al realizar un cifrado de vigenere con una clave del tamaño del mensaje a cifrar estamos haciendo un cifrado de flujo.

3. Cifrado de flujo

El siguiente programa implementa el cifrado de flujo.

Llamada a la función:

flujo {-C|-D} {-m clave} {-n tamaño de la secuencia de claves} [-i filein] [-o fileout]

- m es la clave del cifrado de flujo. En este caso asumimos que es una clave numérica, un número entero.

Este cifrado es parecido al cifrado de Vigenere, de hecho, hemos visto que el cifrado de Vigenere se puede entender como un caso particular del cifrado de flujo. Su parecido reside en que el cifrado se realiza elemento a elemento operando con el elemento de la clave correspondiente. La diferencia es que las claves del cifrado de flujo pertenecen a una secuencia cifrante generada de manera aleatoria. Por lo tanto, para la implementación del cifrado de flujo es necesario programar un generador de una secuencia de números aleatorios, es decir, la secuencia cifrante. Hemos realizado un cifrado síncrono de flujo ya que el flujo de claves se codifica a partir de una clave que es independiente del texto original. Nuestro funcion *generador_aleatorio* genera la secuencia cifrante y se basa en el registro de desplazamiento LFSR: Linear Feedback Shift Register. Inicializamos el registro LFSR con la función *secuencia_LFSR*, que crea un vector de tamaño m (clave del cifrado de flujo), inicializa la semilla de la generación de números aleatorios y llena el vector aleatoriamente de 0 y 1. La función *generador_aleatorio* calcula el output del LFSR, es decir el número aleatorio de la secuencia de claves, ejecutando la operación XOR con las dos últimas celdas del LFSR. Después, realiza el desplazamiento: desplaza una posición a la derecha los elementos de todas las celdas, inserta el output generado en la primera posición y "elimina la última celda". De esta forma se realiza el desplazamiento y el vector LFSR sigue teniendo el mismo tamaño.

Para cifrar, este método toma para cada elemento, un número aleatorio de la secuencia y los opera con la aplicación XOR lógica. Obtenemos una la cadena cifrada y expresada en binario, ya que actualmente los cifrados de flujo normalmente se expresan en alfabeto binario.

La fortaleza de este cifrado depende directamente del generador de la secuencia aleatoria. En nuestro caso se basa en LFSR, que es un método correcto para generar números aleatorios, pero tal y como se indica en su nombre, los LFSR son lineales y las dependencias lineales son generalmente más sencillas de analizar. Además, en nuestro caso, si el criptoanalista conoce la longitud del LFSR, conoce también la clave del cifrado de flujo ya que hemos utilizado esta misma para determinar la semilla y la longitud del LFSR. Por tanto, la combinación de LFSR o la implementación de NLFSR son mecanismos más sofisticados de generación de secuencias cifrantes y darán más robustez al criptosistema.

In [147]:

```
import random
```

In [148]:

```
def secuencia_LFSR(m):
    random.seed(m)
    LFSR=[]
    for i in range(m):
        LFSR.append(random.randint(0,1))
    return LFSR
```

In [149]:

```
def generador_aleatorio(LFSR):
    n=len(LFSR)
    output=int(bin(LFSR[n-1]^LFSR[n-2])[2:])
    LFSR.insert(0,output)
    LFSR.pop()
    return LFSR, output
```

In [150]:

```
# Ejemplo de secuencia cifrante
m=5
lfsr=secuencia_LFSR(m)
for i in range(10):
    lfsr,k=generador_aleatorio(lfsr)
    print(lfsr)
    print(k)
```

```
[1, 1, 1, 0, 1]
1
[1, 1, 1, 1, 0]
1
[1, 1, 1, 1, 1]
1
[0, 1, 1, 1, 1]
0
[0, 0, 1, 1, 1]
0
[0, 0, 0, 1, 1]
0
[0, 0, 0, 0, 1]
0
[1, 0, 0, 0, 0]
1
[0, 1, 0, 0, 0]
0
[0, 0, 1, 0, 0]
0
```

In [151]:

```python
# m es la clave
def flujo(modo,m,i=0,o=0):
    alfabeto='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
    if modo=="-C":
        cadena=read_input(i)
        # Traducimos los caracteres a números
        cadena_numerica=[]
        for k in cadena:
            if k in alfabeto:
                cadena_numerica.append(alfabeto.index(k))
        # Inicializamos LFSR
        lfsr=secuencia_LFSR(m)
        # Ciframos carácter a carácter
        cadena_cifrada=[]
        for i in cadena_numerica:
            lfsr,k=generador_aleatorio(lfsr)
            cadena_cifrada.append(int(bin(i^k)[2:]))
        print_output(o,cadena_cifrada)
    elif modo=="-D":
        cadena_cifrada=read_input(i)
        cadena_cifrada=cadena_cifrada.split(" ")
        cadena_descifrada=[]
        cadena_texto=[]
        for i in range(len(cadena_cifrada)):
            cadena_cifrada[i]=int(cadena_cifrada[i],2)

        # Inicializamos LFSR
        lfsr=secuencia_LFSR(m)
        #Desciframos carácter a carácter
        for i in cadena_cifrada:
            lfsr,k=generador_aleatorio(lfsr)
            cadena_descifrada.append((i^k))

        for i in range(len(cadena_descifrada)):
            cadena_texto.append(alfabeto[int(cadena_descifrada[i])])
        print_output(o,cadena_texto)
```

In [152]:

```python
flujo("-C", 5)
```

```
Universidad Autonoma de Madrid
Cadena: Universidad Autonoma de Madrid
Cadena: [101111, 1100, 1001, 10101, 100, 10001, 10010, 1001, 11, 0, 11, 1101
1, 10101, 10011, 1110, 1100, 1110, 1101, 0, 10, 101, 100111, 1, 10, 10001, 1
000, 11]
```

In [153]:

```python
flujo("-D", 5)
```

```
101111 1100 1001 10101 100 10001 10010 1001 11 0 11 11011 10101 10011 1110 1
100 1110 1101 0 10 101 100111 1 10 10001 1000 11
Cadena: ['U', 'n', 'i', 'v', 'e', 'r', 's', 'i', 'd', 'a', 'd', 'A', 'u',
't', 'o', 'n', 'o', 'm', 'a', 'd', 'e', 'M', 'a', 'd', 'r', 'i', 'd']
```

In [154]:

```
flujo("-C", 10, "cadena.txt", "cadena_cifradaFlujo.txt")
```

```
Cadena: Hola Nueva York!
```

In [155]:

```
flujo("-D", 10, "cadena_cifradaFlujo.txt", "cadena_descifradaFlujo.txt")
```

A continuación, aplicamos el cifrado de flujo al libro del Quijote.

In [156]:

```
flujo("-C", 35, "quijote.txt", "quijote_cifradoFlujo.txt")
```

In [157]:

```
flujo("-D", 35, "quijote_cifradoFlujo.txt", "quijote_descifradoFlujo.txt")
```

4. Producto de criptosistemas de permutación

El siguiente programa implementa el cifrado producto de criptosistemas de permutación.

Llamada a la función:

permutacion {-C|-D} {-k1} {-k2} [-i filein] [-o fileout]

- k1 es el vector de m elementos que constitiye la clave para el cifrado de permutación por filas.
- k2: vector de n elementos que constituye la clave para el cifrado de permutación por columnas.

Primero hacemos un procesamiento del input: formamos una matriz de dimensiones m x n, y la rellenamos fila a fila con el input. A esta matriz resultante la hemos denominado *matriz_numerica*.

A continuación realizamos el producto de criptosistemas de permutación:

1. Realizamos la permutación de filas con k1 y obtenemos *matriz_cifrada1* resultado de esta operación. Para ello, hemos ido recorriendo *matriz_cifrada1* (matriz m x n inicializada a 0) y en el elemento i,j hemos introducido el elemento de *matriz_numerica* k1[i],j (i de 1 a m, j de 1 a n) De esta manera se consigue permutar las filas según la clave k1.
2. Realizamos la permutación de columnas sobre la matriz que hemos obtenido en el paso anterior, ahora con k2 como clave. Para este paso, recorremos *matriz_cifrada2* (matriz m x n inicializada a 0) y en el elemento i,j introducimos el elemento de *matriz_cifrada1* i,k2[j] (i de 1 a m, j de 1 a n).

Por último, concatenamos la matriz *matriz_cifrada2*, resultante de aplicar ambas permutaciones.

Para facilitar el seguimiento del algoritmo, se han dejado habilitados los prints de la matriz obtenida tras cada paso.

In [158]:

```
# k1: vector de m elementos que constituye la clave para el cifrado de permutación por
# k2: vector de n elementos que constituye la clave para el cifrado de permutación por
def permutacion(modo,k1,k2,i=0,o=0):
    alfabeto='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
    if modo=="-C":
        cadena=read_input(i)
        # Traducimos los caracteres a números
        cadena_numerica=[]
        for k in cadena:
            if k in alfabeto:
                cadena_numerica.append(alfabeto.index(k))

        # Matriz numerica es la matriz m x n
        m=len(k1)
        n=len(k2)
        maxi=len(cadena_numerica)
        matriz_numerica=np.zeros((m,n))

        pos=0
        for i in range(m):
            for j in range(n):
                if pos<maxi:
                    matriz_numerica[i][j]=cadena_numerica[pos]
                    pos=pos+1
        print(matriz_numerica)

        # Cifrado de permutación por filas
        matriz_cifrada1=np.zeros((m,n))
        for i in range(m):
            for j in range(n):
                matriz_cifrada1[i][j]=matriz_numerica[k1[i]][j]
        print(matriz_cifrada1)

        # Cifrado de permutación por columnas
        matriz_cifrada2=np.zeros((m,n))
        for i in range(m):
            for j in range(n):
                matriz_cifrada2[i][j]=matriz_cifrada1[i][k2[j]]

        print(matriz_cifrada2)
        cadena_cifrada=np.concatenate(matriz_cifrada2)
        resul=""
        for i in cadena_cifrada:
            resul=resul+alfabeto[int(i)]

        print_output(o, resul)
    elif modo=="-D":
        cadena_cifrada_texto=read_input(i)
        cadena_cifrada=[]
        for k in cadena_cifrada_texto:
            if k in alfabeto:
                cadena_cifrada.append(alfabeto.index(k))


         # Matriz numerica es la matriz m x n
        m=len(k1)
        n=len(k2)
        maxi=len(cadena_cifrada)
        matriz_cifrada=np.zeros((m,n))
```

```
60
61          pos=0
62          for i in range(m):
63              for j in range(n):
64                  if pos<maxi:
65                      matriz_cifrada[i][j]=cadena_cifrada[pos]
66                      pos=pos+1
67          print(matriz_cifrada)
68
69          # Desciframos cifrado de permutación por columnas
70          matriz_descifrada2=np.zeros((m,n))
71          for i in range(m):
72              for j in range(n):
73                  matriz_descifrada2[i][k2[j]]=matriz_cifrada[i][j]
74
75          print(matriz_descifrada2)
76
77          # Desciframos cifrado de permutación por filas
78          matriz_descifrada1=np.zeros((m,n))
79          for i in range(m):
80              for j in range(n):
81                  matriz_descifrada1[k1[i]][j]=matriz_descifrada2[i][j]
82          print(matriz_descifrada1)
83
84          cadena_descifrada=np.concatenate(matriz_descifrada1)
85          cadena_texto=[]
86          for i in range(len(cadena_descifrada)):
87              cadena_texto.append(alfabeto[int(cadena_descifrada[i])])
88
89          print_output(o, cadena_texto)
```

In [159]:

```
1 k1=[3,2,4,1,0]
2 k2=[1,3,2,0]
3 permutacion("-C", k1, k2, "cadena.txt", "cadena_cifradaPermutacion.txt")
```

```
Cadena: Hola Nueva York!
[[33. 14. 11.  0.]
 [39. 20.  4. 21.]
 [ 0. 50. 14. 17.]
 [10.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
[[10.  0.  0.  0.]
 [ 0. 50. 14. 17.]
 [ 0.  0.  0.  0.]
 [39. 20.  4. 21.]
 [33. 14. 11.  0.]]
[[ 0.  0.  0. 10.]
 [50. 17. 14.  0.]
 [ 0.  0.  0.  0.]
 [20. 21.  4. 39.]
 [14.  0. 11. 33.]]
```

In [160]:

```
1 permutacion("-D", k1, k2, "cadena_cifradaPermutacion.txt", "cadena_descifradaPermutacio
```

```
Cadena: a a a k Y r o a a a a a a u v e N o a l H
[[ 0.  0.  0. 10.]
 [50. 17. 14.  0.]
 [ 0.  0.  0.  0.]
 [20. 21.  4. 39.]
 [14.  0. 11. 33.]]
[[10.  0.  0.  0.]
 [ 0. 50. 14. 17.]
 [ 0.  0.  0.  0.]
 [39. 20.  4. 21.]
 [33. 14. 11.  0.]]
[[33. 14. 11.  0.]
 [39. 20.  4. 21.]
 [ 0. 50. 14. 17.]
 [10.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

Posible criptoanálisis del doble cifrado de permutación implementado

Bajo el modelo de seguridad que nos permita utilizar la máquina de cifrado infinitas veces para cifrar los textos que queramos, podemos romper el criptosistema mediante el ataque tipo E. con texto claro elegido.

Primero es necesario hallar el tamaño de las claves k1 y k2. Para ello, pensamos que con una cadena de texto plano formada por bloques e ir cambiando el número de elementos del bloque

- K=1 ABCDABCDABCD
- K=2 AABBCCDD
- K=3 AAABBBCCCDDD

Ésto nos puede ayudar a entender el comportamiento del criptosistema y deducir el tamaño de las claves, ya que obtendremos una cadena cifrada con secuencias que se repiten según las claves desconocida. Una vez deducida la longitud de las claves, deberíamos interpretar el texto cifrado y deducir la permutación por columnas y despues la permutación por filas, que nos da la cadena original.