

Carlos Martín Sanz y Alejandro Pulido Sánchez

Entrega 2 - Criptografía 2022

Grado en Ingeniería Informática - Mención de Computación (Universidad de Valladolid)

Imports

In [1]:

```
import math
import random
import numpy as np
from sage.crypto.util import ascii_integer
import time
import matplotlib.pyplot as plt
```

Ejercicio 1 - Código de Huffman

a) Implementa una función que tenga como input el alfabeto fuente y la frecuencia de cada símbolo y tenga como output una codificación trivial para este canal. ¿Cuáles es el número medio de bits usados para transmitir un símbolo?

In [2]:

```
# Recibe un alfabeto con los caracteres así como un array de probabilidades de igual longitud
def codificacionTrivial(alf,prob):
    # Sacamos el tamaño del array, que es el número de caracteres a representar
    tam = len(alfabeto)
    exp = 0
    dicc = {}

    while(2^exp < tam):
        exp = exp + 1

        print("\nEl número de bits necesarios para la representación del código son: ",exp)
        k = 0
        for i in range(len(alf)):
            binario = ""
            if(len(k.binary()) < exp):
                binario = "0" + str(k.binary())
                while(len(binario) < exp):
                    binario = "0" + binario
            dicc[alf[i]] = binario
            else:
                dicc[alf[i]] = k.binary()

            k = k + 1
    return dicc
```

In [3]:

```
# Probamos con un alfabeto de 5 caracteres que necesita 3 bits para su representación
alfabeto = ["a","b","c","d","e"]
probabilidad = [0.2,0.05,0.5,0.15,0.1]
codificacionTrivial(alfabeto,probabilidad)
```

El número de bits necesarios para la representación del código son: 3

Out[3]:

```
{'a': '000', 'b': '001', 'c': '010', 'd': '011', 'e': '100'}
```

In [4]:

```
# Ahora con un alfabeto de 18 caracteres que necesita 5 bits para su representacion
alfabeto = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "ñ", "o", "p", "q"]
codificacionTrivial(alfabeto,probabilidad)
```

El número de bits necesarios para la representación del código son: 5

Out[4]:

```
{'a': '00000',
 'b': '00001',
 'c': '00010',
 'd': '00011',
 'e': '00100',
 'f': '00101',
 'g': '00110',
 'h': '00111',
 'i': '01000',
 'j': '01001',
 'k': '01010',
 'l': '01011',
 'm': '01100',
 'n': '01101',
 'ñ': '01110',
 'o': '01111',
 'p': '10000',
 'q': '10001'}
```

b) Implementa una función que tenga como input el alfabeto fuente y la frecuencia de cada símbolo y tenga como output un código compresor óptimo (código Huffman) para este canal. ¿Cuáles el número medio de bits usados para transmitir un símbolo?

In [5]:

```
# Primero he cogido de internet la clase Nodo, lo cual facilita mucho el acceder a los subarboles
class NodeTree(object):

    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right

    def children(self):
        return self.left, self.right

    def __str__(self):
        return self.left, self.right
```

In [6]:

```
# Función que crea el código de Huffman a partir del árbol
def Codigo_Huffman(node, binString=''):
    if type(node) is str:
        return {node: binString}

    (l, r) = node.children()
    d = dict()

    d.update(Codigo_Huffman(l, binString + '1'))
    d.update(Codigo_Huffman(r, binString + '0'))
    return d

# Función que crea el árbol binario
def creaArbol(nodes):
    while len(nodes) > 1:
        (key1, c1) = nodes[-1]
        (key2, c2) = nodes[-2]
        nodes = nodes[:-2]
        node = NodeTree(key1, key2)
        nodes.append((node, c1 + c2))
        nodes = sorted(nodes, key=lambda x: x[1], reverse=True)
    return nodes[0][0]
```

In [7]:

```

freq = {}

# Defino el alfabeto y las frecuencias de cada símbolo
alfabeto = ["a", "b", "c", "d", "e", "f"]
frecuencias = [0.05, 0.1, 0.08, 0.33, 0.32, 0.12]

# Relleno el diccionario creado, de forma que la clave sea el carácter y el valor la frecuencia
# a : 0.1 , b : 0.2 etc
for i in range(len(alfabeto)):
    freq[alfabeto[i]] = round(frecuencias[i],2)

print("Diccionario de caracteres y frecuencias: \n")
print(freq)

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)

# Creamos el árbol binario
arbol = creaArbol(freq)

# Convertimos el árbol a el código de Huffman
codificación = Código_Huffman(arbol)
print("\n***** SOLUCIÓN *****")
# Impresión del código de Huffman
print("SÍMBOLO | CODIFICACIÓN")

# Variable que contiene el número medio de bits por símbolo
nmedio_bits = 0

for símbolo, código in codificación.items():
    # Nº de bits para un símbolo
    longitud = 0

    # Print de la codificación
    print(" ", símbolo, " ----> ", código)

    # Contamos los bits para representar el símbolo
    for i in código:
        longitud = longitud + 1

    # Halla el índice del símbolo, ya que ese índice coincide con el de su probabilidad en el array de probabilidades
    índice = alfabeto.index(símbolo)

    # Hallamos el número medio de bits
    nmedio_bits = round(nmedio_bits + (frecuencias[indice] * longitud), 3)

print("\n El número medio de bits usados para transmitir un símbolo es : ", nmedio_bits)

```

Diccionario de caracteres y frecuencias:

```
{'a': 0.05, 'b': 0.1, 'c': 0.08, 'd': 0.33, 'e': 0.32, 'f': 0.12}
```

***** SOLUCIÓN *****

SÍMBOLO		CODIFICACIÓN
a	---->	111
c	---->	110
b	---->	101
f	---->	100
e	---->	01
d	---->	00

El número medio de bits usados para transmitir un símbolo es : 2.35

Ejercicio 2 - RSA

a) Implementa una función que, dados dos primos p y q , calcule la clave pública y la clave privada del criptosistema RSA.

In [8]:

```
def RSA_pub_priv(p,q):
    # Primero hallamos n
    n = p*q

    # Ahora hallamos phi
    phi=(p-1)*(q-1)

    # Elegimos el número e, que es un numero entre 1 y phi
    e = random.randint(1,phi)

    # Y ese numero e tiene que cumplir que el mcd(e,phi) = 1
    while(gcd(e,phi) != 1):
        e = random.randint(1,phi)

    # Ya tenemos n y e, y por tanto La clave pública
    c_publica = (n,e)

    # Ahora calculamos la d
    d = inverse_mod(e,phi)

    # Y con la d calculada tenemos la clave privada
    c_privada = (d,p,q,phi)

    return c_publica,c_privada
```

b) Implementa una función que, dada la clave pública (n, e) de un criptosistema RSA y un mensaje en Z_n , cifre dicho mensaje.

In [9]:

```
def cifraRSA(c_publica,m):

    # c_publica contiene dos posiciones (X,Y) donde X es n y donde Y es e
    n = c_publica[0]
    e = c_publica[1]

    # m que recibimos como parametro es el mensaje, en este caso un numero entre 0 y n-
    1
    # y generamos el criptograma, elevando utilizando la funcion powermod (que optimiza
    esta cuenta, sobretodo es útil para
    # números grandes, y evita que el ordenador se quede sin memoria ejecutando dicha o
    peración)

    # power_mod(numero,elevas,mod) ----> similar a ----> m^e % n
    C = power_mod(m,e,n)

    return C
```

c) Implementa una función que, dada la clave privada d de un criptosistema RSA y un mensaje cifrado con la función anterior, descifre dicho mensaje.

In [10]:

```
def descifraRSA(c_privada,C):

    # La clave privada obtenida en la función del apartado a) esta formada por (d,p,q,p
    hi)
    # para descifrar necesitamos d y n = p*q, por tanto accedemos a las posiciones corr
    espondientes
    d = c_privada[0]
    n = c_privada[1] * c_privada[2]

    # desciframos
    M_descifrado = power_mod(C,d,n)

    return M_descifrado
```

Pruebas funcionamiento de las funciones anteriores - Números pequeños

In [11]:

```
# Pruebo con Los números del código Sage del Campus Virtual
# Números primos p y q, así como el mensaje a cifrar (número entre 0 y n-1)
p = 11
q = 23
m = 68

# Llamadas a funciones y resultados
clave_publica, clave_privada = RSA_pub_priv(p,q)
print("La clave pública es: ", clave_publica, "\nLa clave privada es: ", clave_privada)
cifrado = cifraRSA(clave_publica,m)
print("El mensaje: ",m," se ha cifrado con RSA como: ",cifrado)
m_original = descifraRSA(clave_privada, cifrado)
print("Y descifrando el mensaje cifrado: ", cifrado, " obtenemos el mensaje original: ", m_original)
```

La clave pública es: (253, 133)
 La clave privada es: (177, 11, 23, 220)
 El mensaje: 68 se ha cifrado con RSA como: 206
 Y descifrando el mensaje cifrado: 206 obtenemos el mensaje original: 68

Pruebas funcionamiento de las funciones anteriores - Números grandes

In [12]:

```
# Números primos p y q, así como el mensaje a cifrar (número entre 0 y n-1)
p = next_prime(11000000)
q = next_prime(23000000)

m = random.randint(1,(p*q)-1)

# Llamadas a funciones y resultados
clave_publica, clave_privada = RSA_pub_priv(p,q)
print("La clave pública es: ", clave_publica, "\nLa clave privada es: ", clave_privada)
cifrado = cifraRSA(clave_publica,m)
print("\nEl mensaje: ",m," se ha cifrado con RSA como: ",cifrado)
m_original = descifraRSA(clave_privada, cifrado)
print("\nY descifrando el mensaje cifrado: ", cifrado, " obtenemos el mensaje original: ", m_original)
```

La clave pública es: (253000720000243, 164690444155835)
 La clave privada es: (95320617200883, 11000027, 23000009, 25300068600020
 8)

El mensaje: 169439518170211 se ha cifrado con RSA como: 46703607530727

Y descifrando el mensaje cifrado: 46703607530727 obtenemos el mensaje original: 169439518170211

Ejercicio 3 - Cifrado en bloque de texto

a) Implementa una función que dada la clave pública de un criptosistema RSA y un texto M escrito con el alfabeto ZN , cifre dicho mensaje. Por ejemplo N puede ser igual a 256 y considerar el código ASCII. Nota: el tamaño del bloque será k, con $Nk \leq n < Nk+1$

In [13]:

```
# Antes de empezar con el cifrado en bloque tenemos que asegurarnos la condición del enunciado es decir  $Nk \leq n < Nk+1$ 
# Debido a esto no nos sirve cualquiera dos números primos, sino que han de cumplir la condición
```

In [14]:

```
# Defino los límites de n
def previaBloques(N,k):
    lim_inf = N^k
    lim_sup = N^(k+1)
    p = 0
    q = 0

    # Ahora no puedo empezar el primo en 2, por que no acaba por tanto
    primo = (sqrt(lim_inf)).floor()

    # Y para que no tarde tanto en encontrar los valores defino una cte, a la que luego multiplico a un número en este caso 50
    suma = 100

    while(((p*q) < lim_inf) or ((p*q) > lim_sup) or (p==q)):
        # para p resto, ya que es mejor que q, y para q lo sumo
        p = next_prime(primo-250*suma)
        q = next_prime(primo+1000*suma)
        suma = suma + 1

    # Hallamos n
    n = p*q

    # Comprobamos que se cumple
    if((n < lim_sup) and (n > lim_inf)):
        print("\nEl valor de n es: ", n, " y se encuentra dentro del intervalo")

    return p,q
```

In [15]:

```
# Con los pasos previos que he realizado ya podemos enviar al cifrado en bloque tanto el mensaje en ASCII (M) como la clave
# publica generada a partir de los numeros primos que cumplen la condición
def cifradoBloqueRSA(M,c_pub,k,N):

    # El mensaje M lo tenemos que dividir en bloques el tamaño k
    bloques = dividirBloques(M,k)

    resultadoC = ""
    # Ahora que ya tengo los bloques, he de cifrar cada uno de ellos
    for i in bloques:
        palabra = cifradoBloque(c_pub,i,k,N)
        for j in palabra:
            resultadoC = resultadoC + chr(j)
    return resultadoC, N
```

In [16]:

```

def dividirBloques(M,k):

    # Pasamos el mensaje a una Lista para iterarla
    lista = list(M)
    bloques = []

    # Contador
    cont = 0

    # Contenido de cada bloque
    palabra = ""

    # Defino el tamaño de la Lista como una variable porque la utilizo mucho
    tam = len(lista)

    # Calcula el número de bloques en los que ha de dividirse el mensaje M
    if((tam % k) == 0):
        numbloques = tam // k
    else:
        numbloques = (tam // k)+1

    print("\nEl número de bloques en los que se divide el mensaje M es: ", numbloques)

    # Siempre que se pueda formar un bloque completo ...
    while(k < tam-cont):

        palabra = ""
        for i in range(k):
            palabra += lista[i+cont]

        cont += k

        # Añadimos la palabra
        bloques.append(palabra)

        if cont != tam:
            palabra = ""

        # Añadimos los caracteres del bloque incompleto
        for i in range(cont,tam):
            palabra += lista[i]

        # Lo que falta entre lo que llevamos de palabra y longitud del bloque se rellena con " "
        for j in range(len(palabra),k):
            palabra += " "

        bloques.append(palabra)

    print("\nMensaje dividido en bloques: \n",bloques)
    return bloques

```

In [17]:

```
# Ahora consultamos como se realiza el cifrado en bloque mirando en Los apuntes del día
# 09/11, diapositiva 6
def cifradoBloque(c_pub,M,k,N):

    # Primero hallamos m
    m = 0
    ind = 1

    # Hallamos m
    for j in M:
        m = m +(ord(j)* N^(k-ind))
        ind = ind + 1

    # Ahora desglose de la clave publica
    n = c_pub[0]
    e = c_pub[1]

    # Ahora cifro el mensaje
    C = power_mod(m,e,n)

    # Hallar C en base N
    # Primero duplico el criptograma y hago la listaFinal
    repetido = C
    listaFinal = []

    while(repetido>=N):

        resto = int(repetido%N)

        #Insertamos al principio de la lista cada resto
        listaFinal.insert(0,resto)

        #Actualizamos el valor de nuestra variable auxiliar
        repetido = int(repetido/N)

        if(repetido<N):
            listaFinal.insert(0,repetido)
    return listaFinal
```

b) Implementa una función que dada la clave privada de un criptosistema RSA y un texto crificado con la función anterior, descifre dicho texto.

In [18]:

```
def descifrado_bloque_RSA(privada,cifrado,n,k,N):
    lista = dividirBloques(cifrado,k)
    descifrado = ""

    for i in lista:
        palabra = descifrado_bloque(privada,i,k,N,n)
        for j in palabra:
            descifrado += chr(j)

    return descifrado
```

In [19]:

```
def descifrado_bloque(privada,cifrado,k,N,n):
    rep = 0
    c = 0

    # Desglose de la clave privada
    d = privada[0]

    for i in range(len(cifrado)):
        c += ord(cifrado[-i-1])*pow(N,0+rep)
        rep += 1

    #Calculamos b
    b = power_mod(c,d,n)

    # Duplicamos b en repetido que es una variable auxiliar
    repetido = b
    listaFinal = []

    #Escribimos b en base N
    while(repetido>=N):

        resto = int(repetido%N)
        #Insertamos al principio de la lista cada resto
        listaFinal.insert(0,resto)
        #Actualizamos el valor de nuestra variable auxiliar
        repetido = int(repetido/N)

        if(repetido<N):
            listaFinal.insert(0,repetido)

    return listaFinal
```

Aunque el funcionamiento de cada parte anterior ya ha sido probada una por una que funciona, he eliminado esas pruebas realizadas. La mejor forma de ver el funcionamiento es juntando todo y viendo si cifra un mensaje bien y como lo hace y como lo descifra printeando cada parte

In [20]:

```
# Prueba de que el cifrado y descifrado en bloque funciona
# Defino n y el tamaño de bloque que es k
N = 256
k = 5

# Con previa bloques hallo tanto p como q
p,q = previaBloques(N,k)

# Y ahora la clave publica y privada RSA, usando la funcion del ejercicio 2
c_pub,c_priv = RSA_pub_priv(p,q)

print("\nValores obtenidos: \n-p: ",p," \n-q: ",q," \n-clave publica: ",c_pub," \n-clave privada: ",c_priv)

# Defino el mensaje que va a ser cifrado y descifrado
mensaje = "Diego ponme un 10 en la practica"

# Realizo el cifrado por bloques
mcifrado, N = cifradoBloqueRSA(mensaje,c_pub,k,N)

# mcifrado contiene el mensaje cifrado
print("\nEl mensaje ORIGINAL: ",mensaje)

print("\nMediante cifrado de bloque RSA obtenemos el mensaje CIFRADO: ", mcifrado)

print("\n*****\nYA HEMOS CIFRADO AHORA TOCA DESCIFRAR\n*****")

# n es el primero campo de la clave publica (n,e)
n = c_pub[0]
descifrado = descifrado_bloque_RSA(c_priv,mcifrado,n,k,N)
print("\nEl mensaje descifrado es: ", descifrado)
```

El valor de n es: 1175673377161 y se encuentra dentro del intervalo

Valores obtenidos:

```
-p: 1023577
-q: 1148593
-clave publica: (1175673377161, 556242469243)
-clave privada: (121839117619, 1023577, 1148593, 1175671204992)
```

El número de bloques en los que se divide el mensaje M es: 7

Mensaje dividido en bloques:

```
['Diego', ' ponm', 'e un ', '10 en', ' la p', 'racti', 'ca ']
```

El mensaje ORIGINAL: Diego ponme un 10 en la practica

Mediante cifrado de bloque RSA obtenemos el mensaje CIFRADO: $\text{^a} \text{pM } \text{^U\$1Ö}$
 $\text{^o } \text{on+^a-d} \text{^N } 2$
 $\text{D\$% } 4 \text{ } \emptyset \#$

YA HEMOS CIFRADO AHORA TOCA DESCIFRAR

El número de bloques en los que se divide el mensaje M es: 8

Mensaje dividido en bloques:

```
['^a' \x91^A', '\xa0\x91Ö', '\x01\n°\x9eò', 'n+^a-d', '^N\x112', '\n\x98D\x%', '\x144\x81Ø#', '\x10 ']
```

El mensaje descifrado es: Diego ponmy ^o;ï - Ü\&Zmç "Q< NÂ ã4 d

Ejercicio 4 - Ataques de fuerza bruta RSA y ElGamal

a) Implementa una función por fuerza bruta que dado $n = p \cdot q$ producto de dos primos, factorice n. Es decir, encuentre p y q.

In [21]:

```
def factorizaFB(n):
    # next_prime esta función nos devuelve el siguiente número primo, el primero de ellos es 2 por tanto
    primo = 2

    # Y ahora Los numeros p y q, almacenados en una lista, donde p ocupa la posición [0] y q la posición [1]
    pyq = []

    while(n > 1):
        # Si el resto es 0, hemos encontrado uno de los primos que lo divide
        if((n % primo) == 0):
            # Actualizo el valor de n con el cociente de la división
            n = n // primo
            pyq.append(primo)
        else:
            primo = next_prime(primo)

    return pyq
```

In [22]:

```
# Tomo dos números primos de la tabla de primos en este caso 43 y 47, los multiplico y
da 2021, para comprobar si el método
# programado funciona voy a pasarle a la función 2021, y ver que dos primos nos devuelv
e
n = 256
pyq= factorizaFB(n)
print("Los dos primos p y q son respectivamente: ",pyq[0],pyq[1])
```

Los dos primos p y q son respectivamente: 2 2

b) Implementa una función por fuerza bruta que dada la clave pública (n, e) del criptosistema RSA, calcule $\phi(n)$.

In [23]:

```
def hallaphi(c_publica):
    n = c_publica[0]
    # Con n podemos obtener p y q, utilizando la función del apartado anterior
    primos = factorizaFB(n)
    p = primos[0]
    q = primos[1]

    #  $\phi = (p-1)(q-1)$ 
    phi = (p-1)*(q-1)

    return phi
```

In [24]:

```
# Pruebas para hallar el valor phi, con los datos del ejemplo Campus Virtual SAGE
c_pub,c_priv = RSA_pub_priv(pyq[0],pyq[1])
phi = hallaphi(c_pub)
print("El valor de  $\phi(n)$ : ",phi)
```

El valor de $\phi(n)$: 1

c) Implementa una función por fuerza que dada la clave pública (n, e) del criptosistema RSA, calcule la clave privada $d = e^{-1} \bmod n$.

In [25]:

```
def hallarD(c_publica):
    n = c_publica[0]
    e = c_publica[1]

    # Ahora hallamos p y q
    primos = factorizaFB(n)
    p = primos[0]
    q = primos[1]

    phi = (p-1)*(q-1)

    # Pero ahora necesitamos d, no phi por tanto...
    d = power_mod(e,-1,phi)

    return d
```

In [26]:

```
d = hallarD(c_pub)
print(c_pub)
print("El valor de d de la clave privada es: ",d)
```

(4, 1)
El valor de d de la clave privada es: 0

d) Implementa una función por fuerza bruta que dados g, p y A , con g un generador de \mathbb{Z}_p^* , calcule el logaritmo discreto de A en base g modulo p : $\text{logg}(A) \bmod p$

In [27]:

```
def log_discreto(g, p, A):
    # Para cada valor de i desde 0 hasta p-1, calculamos  $g^i \bmod p$  y lo comparamos con A
    for i in range(p):

        # Calculamos  $g^i \bmod p$  utilizando la función power_mod() de SageMath
        valor = power_mod(g, i, p)

        # Si  $g^i \bmod p$  es igual a A, entonces hemos encontrado el logaritmo discreto de A en base g modulo p
        if valor == A:

            # Devolvemos el valor de i como el Logaritmo discreto de A en base g modulo p
            return i

    # Si llegamos hasta aquí, entonces no se ha encontrado un logaritmo discreto de A en base g modulo p, por lo que devolvemos false como un valor de error
    return False
```

In [28]:

```
def generadorMultiplicativo(q):
    # Hallamos  $2^q + 1$ 
    p = 2*q + 1

    # Vemos si es primo o no
    while not is_prime(p):

        # Si no lo es, obtenemos el siguiente primo
        q = next_prime(q)
        p = 2*q + 1

    # Ahora buscamos al azar g tal que,  $1 < g < p-1$  y que ademas, cumpla con lo siguiente:
    g^2 != 1 mod p y que  $g^g \neq 1 \bmod p$ 
    g = random.randint(1,p-1)

    while True:
        if power_mod(g,2,p) != 1 and power_mod(g,g,p) != 1:
            break

        else:
            g = random.randint(1,p-1)

    return p,g
```

In [29]:

```
Cifras = []
Tiempo = []
```

In [30]:

```
#Numero de digitos de q con los que vamos a probar para hallar los tiempos
numDigitos = 6
q = 1
for i in range(numDigitos):
    q = q*10
    q = q + random.randint(1,9)

p,g = generadorMultiplicativo(q)

A = random.randint(1,p-1)
print("Tenemos p, g y A con valores: ",p," , ",g," y ",A," respectivamente")

inicio = time.time()
salida = log_discreto(g,p,A)
fin = time.time()

tiempoTotal = fin-inicio

Cifras.append(len(str(q)))
Tiempo.append(tiempoTotal)
# En funcion de la salida obtenida imprimimos un mensaje u otro
if(salida == False):
    print("\nNo podemos calcular el valor del logaritmo discreto")
else:
    print("\nEl valor del logaritmo discreto es: ",salida)
```

Tenemos p, g y A con valores: 37 , 28 y 36 respectivamente

El valor del logaritmo discreto es: 9

Tenemos p, g y A con valores: 383 , 236 y 194 respectivamente

El valor del logaritmo discreto es: 45

Tenemos p, g y A con valores: 3779 , 1043 y 1184 respectivamente

El valor del logaritmo discreto es: 1752

Tenemos p, g y A con valores: 37799 , 5986 y 27610 respectivamente

El valor del logaritmo discreto es: 13536

Tenemos p, g y A con valores: 377717 , 157689 y 80221 respectivamente

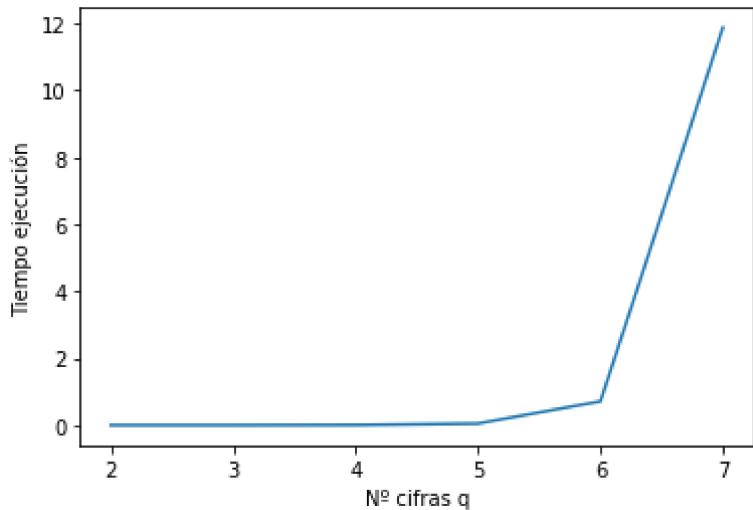
El valor del logaritmo discreto es: 151271

Tenemos p, g y A con valores: 3777539 , 2771440 y 1649497 respectivamente

El valor del logaritmo discreto es: 2084557

In [31]:

```
#Ahora que ya tenemos los tiempos y el número de cifras vamos a crear un gráfico donde se muestre
#que a medida que aumentamos el numero de cifras de q el tiempo se ve aumentado exponencialmente.
plt.plot(Cifras, Tiempo)
plt.xlabel('Nº cifras q')
plt.ylabel('Tiempo ejecución')
plt.show()
```



e) ¿Hasta qué tamaño del input las funciones anteriores son capaces de terminar?, ¿Hemos atacado con éxito los criptosistema RSA y ElGamal?

Con números exageradamente grandes las funciones terminan, en principio terminan siempre. Pero para que terminen el número n, que proporcionamos ha de ser generado a través del producto de dos números primos, si no lo son las funciones no terminan.

Pero en el ejercicio de fuerza bruta podemos ver que al duplicar el número de cifras el tiempo no se duplica sino que se multiplica.

Tanto p como q, que son los primos encargados de generar ese número n, han de ser grandes (aumenta la seguridad). y por conseciente, cuanto más grande sea esos p y q, más grande n, y la factorización va a ser más costosa

Con respecto a la pregunta que nos dice si hemos sido capaces de atacar con éxito el criptosistema RSA y ElGamal, la respuesta es SÍ. Puesto que hemos conseguido hallar el valor de la clave privada por fuerza bruta, así como los primos p y q, con los que hemos hallado n, como hemos conseguido atacar RSA, también ElGamal

Ejercicio 5 - Firma digital

a) Implementa una función que firme digitalmente un mensaje usando la firma digital con ElGamal y una función hash. Se puede usar cualquier función hash que el alumno quiera, en SageMath y Python está por ejemplo implementada la función hash(). Se puede truncar o reducir por un módulo el output de función hash. El input debe ser la clave privada de ElGamal (también son necesarios p y g) y el mensaje a firmar. El output debe ser la firma.

In [32]:

```
def firmaDigitalElGamal(p,g,priv_gamal,M):
    # Elegimos un K de forma que  $1 < K < p-1$  y que  $\text{mcd}(k,p-1)=1$ 
    k = 1

    while not((gcd(k,p-1) == 1) and k != 1):
        k = random.randint(1,p-1)

    # Ahora para el mensaje M calculamos --> tanto r como s
    r = power_mod(g,k,p)

    # Hay que poner k pero no va no da Lo mismo idk why
    s = ZZ(k)^-1*(hash(M) - priv_gamal*r) % (p-1)

    # La firma es (r,s)
    return (r,s)
```

Firma digital con El Gamal números pequeños

In [33]:

```
p = 23
# Como uso el ejemplo de los apuntes, tengo el valor de g, pero si no lo usase recurriria al método anterior
g = 7
# La clave privada es  $2 < \text{priv} < p-2$ 
priv = 13
mensaje = "8"
firma = firmaDigitalElGamal(p,g,priv,mensaje)
print("La firma está constituida por (r,s): ",firma)
```

La firma está constituida por (r,s): (5, 2)

Firma digital con El Gamal números grandes

In [34]:

```
p2 = next_prime(2345820)
# Como uso el ejemplo de los apuntes, tengo el valor de g, pero si no lo usase recurriria al método anterior
g2 = 13
# La clave privada es  $2 < \text{priv} < p-2$ 
priv2 = 121212
mensaje2 = "estoesunaprueba"
firma2 = firmaDigitalElGamal(p2,g2,priv2,mensaje2)
print("La firma está constituida por (r,s): ",firma2)
```

La firma está constituida por (r,s): (1308696, 1484705)

b) Implementa una función que dado un mensaje, una firma y una clave pública de ElGamal, compruebe si la firma del mensaje es válida o no, de acuerdo a la firma digital implementada con la función anterior.

In [35]:

```

def compruebaFirmaElGamal(M,firma,cpub_gamal):
    es_valida = False
    p,g,A = cpub_gamal

    r,s = firma

    # Lo primero es mirar que r se ajusta al valor que tiene que tener
    if(1 <= r and r <=p-1):

        # Tenemos que comprobar que ( $A^{r^s} \mod p$ ) y que  $g^h(M) \mod p$  son IGUALES
        #  $(A^{r^s}) \mod p$ 
        condicion1 = (power_mod(A,r,p)*power_mod(r,s,p) % p) % p
        print("El resultado de la primera condicion para que sea correcta: ",condicion
1)

        #  $g^h(M) \mod p$ 
        condicion2 = (power_mod(g,hash(M),p))
        print("El resultado de la segunda condicion para que sea correcta: ",condicion
2)

        # SI SON IGUALES ES VALIDA LA FIRMA PARA ESE MENSAJE
        if(condicion1 == condicion2):
            es_valida = True
        else:
            es_valida = False

    return es_valida

```

Comprobación de la firma El Gamal números pequeños

In [36]:

```

A = power_mod(g,priv,p)
cpub = (p,g,A)
correcta = compruebaFirmaElGamal(mensaje,firma,cpub)

if(correcta == True):
    print("\nLa firma para ese mensaje M es correcta")
else:
    print("\nLa firma para ese mensaje M es incorrecta")

```

El resultado de la primera condicion para que sea correcta: 20
 El resultado de la segunda condicion para que sea correcta: 20

La firma para ese mensaje M es correcta

Comprobación de la firma El Gamal números grandes

In [37]:

```
A2 = power_mod(g2,priv2,p2)
cpub2 = (p2,g2,A2)
correcta = compruebaFirmaElGamal(mensaje2,firma2,cpub2)

if(correcta == True):
    print("\nLa firma para ese mensaje M es correcta")
else:
    print("\nLa firma para ese mensaje M es incorrecta")
```

El resultado de la primera condicion para que sea correcta: 1601751
 El resultado de la segunda condicion para que sea correcta: 1601751

La firma para ese mensaje M es correcta

Ejercicio 6 - Comunicación de Alice y Bob

Alice quiere enviar un mensaje a Bob por un canal inseguro y además quiere firmar dicho mensaje. Para ello supondremos que Alice sabe firmar un mensaje como en el Ejercicio 5 y que Bob tiene implementado un criptosistema de clave pública para recibir mensajes como en el Ejercicio 2.

a) Implementa una función que firme digitalmente un mensaje usando la firma digital con ElGamal y una función hash. Y que además también cifre el mensaje usando RSA. El input debe ser la clave privada de Alice (de ElGamal), la clave pública de Bob (de RSA) y el mensaje a firmar. El output debe ser el mensaje cifrado y su firma.

In [38]:

```
def firmaElGamalHash(priv_Alice, pub_Bob, M,p,g):
    a = priv_Alice

    # Alice firma usando su clave privada
    firma = firmaDigitalElGamal(p,g,a,M)

    # Bob cifra ahora usando su clave publica
    cifrado = cifraRSA(pub_Bob,M)

    return firma,cifrado
```

In [39]:

```
def ElGamal_pub_priv(p,q):
    a = random.randint(2,p-2)
    A = power_mod(g,a,p)

    publica = (p,g,A)
    privada = a

    return publica,privada
```

In [40]:

```
# Para no coger los datos tan pequeños como en ejemplos anteriores
# Uso next_prime para obtener el primo q grande
q = next_prime(1234)

# Tenemos p y q, por lo tanto ahora si que utilizo el metodo generadorMultiplicativo para hallar la g
p,g = generadorMultiplicativo(q)

# Como la clave privada de Alice es de ElGamal entonces:
pub_Alice, priv_Alice = ElGamal_pub_priv(p,g)
print("Clave pública Alice: ", pub_Alice, "\nClave privada Alice: ", priv_Alice)
print("\n-----\n")

# La clave publica de Bob es de RSA entonces:
pub_Bob, priv_Bob = RSA_pub_priv(p,q)
print("Clave pública Bob: ", pub_Bob, "\nClave privada Bob: ", priv_Bob)
print("\n-----\n")

n = p*q

# Mensaje --> igual que antes el mensaje tiene que ser (número entre 0 y n-1)
mensaje = random.randint(0,n-1)
print("\nEl mensaje es: ", mensaje)

# Llamamos a la función que realiza la firma digital con el Gamal y cifrado del mensaje con RSA
firma, cifrado = firmaElGamalHash(priv_Alice, pub_Bob, mensaje, p, g)
print("La firma es: ", firma, " el mensaje: ", mensaje, " cifrado es: ", cifrado)
```

Clave pública Alice: (2579, 679, 2396)
 Clave privada Alice: 2232

Clave pública Bob: (3190223, 1313621)
 Clave privada Bob: (2719205, 2579, 1237, 3186408)

El mensaje es: 2649893
 La firma es: (1429, 2333) el mensaje: 2649893 cifrado es: 233995

b) Implementa una función que dado un mensaje cifrado y una firma, descifre el mensaje y compruebe que la firma es válida, de acuerdo al método de la función anterior. El input debe ser la clave pública de Alice (de ElGamal), la clave privada de Bob (de RSA), el mensaje cifrado y la firma. El output debe ser el mensaje descifrado y un mensaje que nos diga si la firma era válida o no.

In [41]:

```
def descifrarElGamalHash(cpub_Alice,cpriv_Bob,cifrado,firma):
    n = cpub_Alice[0]
    e = cpub_Alice[1]

    # Primero desciframos el mensaje que ha sido cifrado con RSA
    mensaje_original = descifraRSA(cpriv_Bob,cifrado)

    # Ahora comprobamos la firma que hemos realizado con el gamal
    valida = compruebaFirmaElGamal(mensaje_original,firma,cpub_Alice)

    return mensaje_original,valida
```

In [42]:

```
mensaje_des, valida = descifrarElGamalHash(pub_Alice,priv_Bob,cifrado,firma)

# Vemos que el mensaje descifrado coincide con el original
print("\nEl mensaje original es: ", mensaje_des)

print("\n++++++")
if(mensaje_des == mensaje):
    print("El mensaje descifrado coincide con el original")
else:
    print("El mensaje descifrado no coincide con el original")
print("++++++")

if(valida == True):
    print("\nLa firma es válida")
else:
    print("\nLa firma NO es válida")
```

El resultado de la primera condicion para que sea correcta: 1897
 El resultado de la segunda condicion para que sea correcta: 1897

El mensaje original es: 2649893

++++++
 El mensaje descifrado coincide con el original
 +++++++

La firma es válida