Práctica 2: Seguridad Perfecta y Criptografía Simétrica

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.

```
import numpy as np
import math
import random
```

1. Seguridad Perfecta

Podemos definir la seguridad perfecta de un criptosistema, como: Pp (x|y) = Pp (x), ∀ x que pertenecen a P, y ∀ y que pertenecen a C.

Se asume que la clave es usada en un único cifrado, y que cada vez que se cifra se cambia a clave de acuerdo a una distribución de probabilidad de claves, es decir, no se puede obtener infromación del texto plano observando el texto cifrado.

a. Comprobación empírica de la Seguridad Perfecta del cifrado por desplazamiento:

1. Consigue Seguridad Perfecta si se eligen las claves con igual probabilidad.
2. No consigue Seguridad Perfecta si las claves no son equiprobables.

seg-perf {-P | -I} [-i f ilein] [-o f ileout]

-P se utiliza el m´etodo equiprobable.

-I se utiliza el m´etodo no equiprobable.

La salida consistie en las probabilidades Pp(x) de los elementos de texto plano, y las probabilidades condicionadas Pp(x|y) para cada elemento de texto plano y de texto cifrado, con

el siguiente formato:

Pp(A) = %lf

Pp(B) = %lf

... Pp(Z) = %lf

Pp(A|A) = %lf Pp(A|B) = %lf ... Pp(A|Z) = %lf Pp(B|A) = %lf Pp(B|B) = %lf ... Pp(B|Z) = %lf ...

Pp(Z|A) = %lf Pp(Z|B) = %lf ... Pp(Z|Z) = %lf

```
def seg_perf(modo, cad=0,i=0, o=0):
  if cad==0: #esta parte solo esta implementada para que ejecutar los ejemplo sea más fáci
    if i==0:
      cadena=input()

    else:
      file=open(i, "r")
      cadena=file.read()
      file.close()

  else:
    cadena=cad

  print("cadena: ", cadena)

  p_x=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
  p_y=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
  #p_k=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
  p_yx=[[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*2
  p_xy=[[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*26,[0]*2

  alfabeto="abcdefghijklmnopqrstuvwxyz"
  alfabetoM="ABCDEFGHIJKLMNOPQRSTUVWXYZ"

  cadena_numerica=[]
  for i in range(len(cadena)):
    if cadena[i] in alfabeto:
      cadena_numerica.append(alfabeto.index(cadena[i]))
    elif cadena[i] in alfabetoM:
      cadena_numerica.append(alfabetoM.index(cadena[i]))

  if modo=="-P": #el método equiprobable
    #en este método las probabilidades de cada letra del alfabeto son iguales
    cifrado,p_k=cifrado_equiprobable(cadena_numerica)

  elif modo=="-I": #el método no equiprobable
    cifrado,p_k=cifrado_no_equiprobable(cadena_numerica)

  print("cifrado: ", cifrado)

  #calculamos P(x)
  for i in range(len(cadena)):
      for j in range(len(alfabeto)):
        #if cadena[i] not in caracteres_plano:
```

```
        #caracteres_plano.append(i)
      if cadena[i]==alfabeto[j]:
        p_x[j]=p_x[j]+1

for i in range(len(p_x)):
    p_x[i]=p_x[i]/len(cadena)



#calculamos P(y), la probabilidad
#p_k=1/len(cadena) #p_k es igual para todos los elementos
for y in range(len(cifrado)):
  for j in range(len(alfabeto)):
      if cifrado[y]==alfabeto[j]:
        p_y[j]=p_x[j]*p_k[j]
          #p_y[j]=p_y[j]+1

for i in range(len(p_y)):
    p_y[i]=p_y[i]/len(cifrado)



#calculamos P(y|x)
for x in range(len(cadena_numerica)):
  for y in range(len(cifrado)):

    clave=alfabeto.index(cifrado[y])-(cadena_numerica[x])

    if clave<0:
      clave=clave+25

    p_yx[alfabeto.index(cifrado[y])][cadena_numerica[x]]=p_yx[alfabeto.index(cifrado[y])



#calculamos P(x|y)
for i in range(len(p_xy)):
  for j in range(len(p_xy[i])):
      if p_yx[j][i]==p_y[j]:
          p_xy[i][j]=p_x[i]

      elif p_y[j]!=0.0:
        p_xy[i][j]=(p_x[i]*p_yx[j][i])/p_y[j]

      else:
          p_xy[i][j]=0.0


if o==0:
  imprimir_datos(p_xy,p_x, p_yx, p_y, p_k)
  imprimir_seguridad(p_xy,p_x)

else:
  file=open(o, "w")
  #cadenaToStr = ' '.join([str(elem) for elem in cadena])
```

```
s="Soluciones de Pk(k)"+"\n"
file.write(s)
stri=""
for k in range(len(p_k)):
  sub="Pk("+str(alfabeto[k])+") = "+ str(p_k[k])
stri=stri+ " "+sub
file.write(stri)

s="Soluciones de P(x)"+"\n"
file.write(s)
stri=""
for x in range(len(p_x)):
  sub="Px("+str(alfabeto[x])+") = "+ str(p_x[x])
stri=stri + " " + sub
file.write(stri)

s="Soluciones de P(y)"+"\n"
file.write(s)
stri=""
for y in range(len(p_y)):
  sub="Py("+str(alfabeto[y])+") = "+ str(p_y[y])
stri=stri+ " "+sub
file.write(stri)

#print P(y|x)
s="Soluciones de P(y|x)" + "\n"
file.write(s)
for k in range(len(p_yx)):
  stri=""
  for j in range(len(p_yx[k])):
    sub="Pyx("+ str(alfabeto[k])+"|"+ str(alfabeto[j])+") = "+ str(p_yx[k][j])
  stri=stri+" "+sub
  file.write(stri)

#print P(x|y)
s="Soluciones de P(x|y)" + "\n"
file.write(s)
for k in range(len(p_xy)):
  stri=""
  for j in range(len(p_xy[k])):
      sub="Pxy("+ str(alfabeto[k])+"|"+ str(alfabeto[j])+") = "+ str(p_xy[k][j])
      stri=stri+" "+sub
  file.write(stri)



file.close()
```

```
  #salida: probabilidade Pp(x) de los elementos de texto plano y las probabilidades
  #condicionadas Pp(x|y) para cada elemento de texto plano y de texto cifrado
```

imprimir_seguridad {p_xy} {p_x}

Función que imprime por pantalla si el cifrado tiene seguridad perfecta. Para ello imprime cada p_xy con su correspondiete p_x y analiza si son iguales. Si no lo son, imprime: "no seguridad perfecta" y si lo son, imprime: "seguridad perfecta".

```
def imprimir_seguridad(p_xy,p_x):
  print("Comprobacion de seguridad perfecta"+"\n")
  for i in range(len(p_xy)):
      for j in range(len(p_xy[i])):
        if p_xy[i][j]<p_x[i]+0.05:
          if p_xy[i][j]>p_x[i]-0.05:
            print(p_xy[i][j] , p_x[i],"seguridad perfecta")
          else:
            print(p_xy[i][j], p_x[i], "no seguridad perfecta")
        else:
          print(p_xy[i][j], p_x[i], "no seguridad perfecta")
```

imprimir_datos{p_xy} {p_x} {p_yx} {p_y} {p_k}

Función que imprime por pantalla las probabilidades con el formato que hemos indicado antes

```
def imprimir_datos(p_xy,p_x, p_yx, p_y, p_k):
  alfabeto="abcdefghijklmnopqrstuvwxyz"
  #imprimimos por pantalla las probabilidades de Pk(k)
  print("Soluciones de Pk(k)"+"\n")
  stri=""
  for k in range(len(p_k)):
    sub="Pk("+str(alfabeto[k])+") = "+ str(p_k[k])
    stri=stri+ " "+sub
  print(stri+ "\n")

  #print P(x)
  print("Soluciones de P(x)"+"\n")
  stri=""
  for x in range(len(p_x)):
    sub="Px("+str(alfabeto[x])+") = "+ str(p_x[x])
    stri=stri + " " + sub
  print(stri+ "\n")

  #print P(y)
  stri=""
  for y in range(len(p_y)):
    sub="Py("+str(alfabeto[y])+") = "+ str(p_y[y])
    stri=stri+ " "+sub

  print("Soluciones de P(y)" + "\n")
  print(stri+"\n")
```

```
  #print P(y|x)
  print("Soluciones de P(y|x)" + "\n")
  for k in range(len(p_yx)):
    stri=""
    for j in range(len(p_yx[k])):
        sub="Pyx("+ str(alfabeto[k])+"|"+ str(alfabeto[j])+") = "+ str(p_yx[k][j])
        stri=stri+" "+sub
    print(stri+"\n")

  #print P(x|y)
  print("Soluciones de P(x|y)" + "\n")
  for k in range(len(p_xy)):
    stri=""
    for j in range(len(p_xy[k])):
        sub="Pxy("+ str(alfabeto[k])+"|"+ str(alfabeto[j])+") = "+ str(p_xy[k][j])
        stri=stri+" "+sub
    print(stri+"\n")
```

Mediante está función se cifra y descifra con claves de probabilidad equiprobable. Cuanto más largo sea el texto, más debería acercarse a p=0.038.

def cifrado_equiprobable {cadena_numerica}

Esta función devuelve la cadena cifrada y vector que contirne la probabilidad de las llaves, p_k.

El esquema utilizado para la creación de claves es:

1. elegir una clave de forma aleatoria
2. añadir en orden la clave al vector de claves (cada elemento del texto plano esta cifrado con una clave distinta)

```
 for k in range(len(cadena_numerica)):
   n=random.randint(0,len(alfabeto)-1)
   clave.append(n)
```

Un lenguaje aleatorio es equiprobable, por lo que al elegir las claves de forma aleatoria, conseguimos el objetivo.

```
def cifrado_equiprobable(cadena_numerica):
  import random
  alfabeto="abcdefghijklmnopqrstuvwxyz"
  probabilidades_clave=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
  cadena_cifrada=[]
  clave=[]

#generamos las claves
  #random.seed(0)
  for k in range(len(cadena_numerica)):
    random.seed(1)
```

```
    n=random.randint(0,len(alfabeto)-1) #un lenguaje aleatorio es equiprobable
    clave.append(n)

#ciframos el mensaje
  cadena_cifrada=[]
  for i in range(len(cadena_numerica)):
    indice=cadena_numerica[i]+clave[i]
    if indice>25:
      indice=26-indice
    cadena_cifrada.append(indice)

 #pasamos la cadena_numerica a texto
  cadena_texto=[]
  for i in range(len(cadena_cifrada)):
    cadena_texto.append(alfabeto[cadena_cifrada[i]])

  stri=""
  for i in range(len(cadena_texto)):
    stri=stri+cadena_texto[i]

#calculamos las probabilidades de las claves

  for k in range(len(clave)):
    for j in range(len(alfabeto)):
      if alfabeto[clave[k]]==alfabeto[j]:
        probabilidades_clave[j]=probabilidades_clave[j]+1

  for k in range(len(probabilidades_clave)):
     probabilidades_clave[k]=probabilidades_clave[k]/len(probabilidades_clave)

  return stri, probabilidades_clave
```

Mediante está función se cifra y descifra con claves de probabilidad no equiprobable.

def cifrado_no_equiprobable {cadena_numerica}

Esta función devuelve la cadena cifrada y vector que contirne la probabilidad de las llaves, p_k.

El esquema utilizado para la creación de claves es:

1. elegir una clave de forma aleatoria
2. si esa clave está entre 0 y 9, cambiamos el resultado por la clave de b.
3. añadir en orden la clave al vector de claves (cada elemento del texto plano esta cifrado con una clave distinta)

```
   for k in range(len(cadena_numerica)):
     if(random.randint(0, 9) == 1):
         clave.append(1)#b tiene mas probabilidad de aparecer que las demas
     else:
         clave.append(random.randint(0,25))
```

De esta forma, la letra b tiene una porbabilidad de 10/26 de ser elegida como clave, eliminando así la posibilidad de que este cifrado sea equiprobable y en consecuencia, no tenga seguridad perfecta.

```
def cifrado_no_equiprobable(cadena_numerica):
  import random
  alfabeto="abcdefghijklmnopqrstuvwxyz"
  probabilidades_clave=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
  cadena_cifrada=[]
  clave=[]

#generamos las claves
  for k in range(len(cadena_numerica)):
    if(random.randint(0, 9) == 1):
        clave.append(1)#b tiene mas probabilidad de aparecer que las demas
    else:
        clave.append(random.randint(0,25))

#ciframos el mensaje
  cadena_cifrada=[]
  for i in range(len(cadena_numerica)):
    indice=cadena_numerica[i]+clave[i]
    if indice>25:
      indice=26-indice
    cadena_cifrada.append(indice)

 #pasamos la cadena_numerica a texto
  cadena_texto=[]
  for i in range(len(cadena_cifrada)):
    cadena_texto.append(alfabeto[cadena_cifrada[i]])

  stri=""
  for i in range(len(cadena_texto)):
    stri=stri+cadena_texto[i]

#calculamos las probabilidades de las claves
  for k in range(len(clave)):
    for j in range(len(alfabeto)):
      if alfabeto[clave[k]]==alfabeto[j]:
        probabilidades_clave[j]=probabilidades_clave[j]+1



  for k in range(len(probabilidades_clave)):
     probabilidades_clave[k]=probabilidades_clave[k]/len(probabilidades_clave)

  return stri, probabilidades_clave
```

▾ Pruebas y ejecuciones del ejercicio 1

Podemos observar que cuanta más grande sea el texto, mejor funciona el programa. Esto se debe a que la estimación mejora cuando la cantidad de datos aumenta. En estos ejemplos para no ocupar mucho espacio hay diferentes tamaños pero sería mejor hacer las pruebas con tamaños mayores

```
cadena="hola nueva york"
seg_perf("-P",cadena,0,"resultado1")
```

```
     cadena:  hola nueva york
     cifrado:  lsperyizeysvo
```

```
cadena="hola nueva york"
seg_perf("-P", cadena)
```

```
     0.0 0.0 seguridad perfecta
     0.0 0.0 seguridad perfecta
     0.0 0.0 seguridad perfecta
     0.0 0.0 seguridad perfecta
     0.0 0.0 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.0 0.06666666666666667 no seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.06666666666666667 0.06666666666666667 seguridad perfecta
     0.0 0.0 seguridad perfecta
     0.0 0.0 seguridad perfecta
     0.0 0.0 seguridad perfecta
     0.0 0.0 seguridad perfecta
     0.0 0.0 seguridad perfecta
     0.0 0.0 seguridad perfecta
     0.0 0.0 seguridad perfecta
     0.0 0.0 seguridad perfecta
     0.0 0.0 seguridad perfecta
     0.0 0.0 seguridad perfecta
```

```
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
```

```
cadena="Hola nueva york"
seg_perf("-I", cadena)
```

```
cadena:  Hola nueva york
cifrado:  upoyyhhvneqzy
Soluciones de Pk(k)

 Pk(a) = 0.038461538461538464 Pk(b) = 0.038461538461538464 Pk(c) = 0.0 Pk(d) = 0.0

Soluciones de P(x)

 Px(a) = 0.13333333333333333 Px(b) = 0.0 Px(c) = 0.0 Px(d) = 0.0 Px(e) = 0.0666666

Soluciones de P(y)

 Py(a) = 0.0 Py(b) = 0.0 Py(c) = 0.0 Py(d) = 0.0 Py(e) = 0.0 Py(f) = 0.0 Py(g) = 0

Soluciones de P(y|x)

 Pyx(a|a) = 0 Pyx(a|b) = 0 Pyx(a|c) = 0 Pyx(a|d) = 0 Pyx(a|e) = 0 Pyx(a|f) = 0 Pyx

 Pyx(b|a) = 0 Pyx(b|b) = 0 Pyx(b|c) = 0 Pyx(b|d) = 0 Pyx(b|e) = 0 Pyx(b|f) = 0 Pyx

 Pyx(c|a) = 0 Pyx(c|b) = 0 Pyx(c|c) = 0 Pyx(c|d) = 0 Pyx(c|e) = 0 Pyx(c|f) = 0 Pyx

 Pyx(d|a) = 0 Pyx(d|b) = 0 Pyx(d|c) = 0 Pyx(d|d) = 0 Pyx(d|e) = 0 Pyx(d|f) = 0 Pyx

 Pyx(e|a) = 0.0 Pyx(e|b) = 0 Pyx(e|c) = 0 Pyx(e|d) = 0 Pyx(e|e) = 0.038461538461538

 Pyx(f|a) = 0 Pyx(f|b) = 0 Pyx(f|c) = 0 Pyx(f|d) = 0 Pyx(f|e) = 0 Pyx(f|f) = 0 Pyx

 Pyx(g|a) = 0 Pyx(g|b) = 0 Pyx(g|c) = 0 Pyx(g|d) = 0 Pyx(g|e) = 0 Pyx(g|f) = 0 Pyx

 Pyx(h|a) = 0.0 Pyx(h|b) = 0 Pyx(h|c) = 0 Pyx(h|d) = 0 Pyx(h|e) = 0.15384615384615

 Pyx(i|a) = 0 Pyx(i|b) = 0 Pyx(i|c) = 0 Pyx(i|d) = 0 Pyx(i|e) = 0 Pyx(i|f) = 0 Pyx

 Pyx(j|a) = 0 Pyx(j|b) = 0 Pyx(j|c) = 0 Pyx(j|d) = 0 Pyx(j|e) = 0 Pyx(j|f) = 0 Pyx

 Pyx(k|a) = 0 Pyx(k|b) = 0 Pyx(k|c) = 0 Pyx(k|d) = 0 Pyx(k|e) = 0 Pyx(k|f) = 0 Pyx
```

```
Pyx(l|a) = 0 Pyx(l|b) = 0 Pyx(l|c) = 0 Pyx(l|d) = 0 Pyx(l|e) = 0 Pyx(l|f) = 0 Pyx

Pyx(m|a) = 0 Pyx(m|b) = 0 Pyx(m|c) = 0 Pyx(m|d) = 0 Pyx(m|e) = 0 Pyx(m|f) = 0 Pyx

Pyx(n|a) = 0.07692307692307693 Pyx(n|b) = 0 Pyx(n|c) = 0 Pyx(n|d) = 0 Pyx(n|e) = (

Pyx(o|a) = 0.0 Pyx(o|b) = 0 Pyx(o|c) = 0 Pyx(o|d) = 0 Pyx(o|e) = 0.0 Pyx(o|f) = 0

Pyx(p|a) = 0.07692307692307693 Pyx(p|b) = 0 Pyx(p|c) = 0 Pyx(p|d) = 0 Pyx(p|e) = (

Pyx(q|a) = 0.0 Pyx(q|b) = 0 Pyx(q|c) = 0 Pyx(q|d) = 0 Pyx(q|e) = 0.0 Pyx(q|f) = 0

Pyx(r|a) = 0 Pyx(r|b) = 0 Pyx(r|c) = 0 Pyx(r|d) = 0 Pyx(r|e) = 0 Pyx(r|f) = 0 Pyx

Pyx(s|a) = 0 Pyx(s|b) = 0 Pyx(s|c) = 0 Pyx(s|d) = 0 Pyx(s|e) = 0 Pyx(s|f) = 0 Pyx

Pyx(t|a) = 0 Pyx(t|b) = 0 Pyx(t|c) = 0 Pyx(t|d) = 0 Pyx(t|e) = 0 Pyx(t|f) = 0 Pyx
```

```
cadena="universidad autonoma de madrid"
seg_perf("-P", cadena)
```

```
cadena="universidad autonoma de madrid"
seg_perf("-I", cadena)
```

```
cadena="Tom told us about his new job, and Jim told us about his new girlfriend. After a w
seg_perf("-P", cadena)
```

```
cadena="Tom told us about his new job, and Jim told us about his new girlfriend. After a w
seg_perf("-I", cadena)
```

```
cadena="conestasrazonesperdiaelpobrecaballeroeljuicioydesvelabaseporentenderlasydesentrana
seg_perf("-P",cadena)
```

```
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
```

```
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.0 0.0 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.0 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
0.01764705882352941 0.01764705882352941 seguridad perfecta
```

```
cadena="conestasrazonesperdiaelpobrecaballeroeljuicioydesvelabaseporentenderlasydesentrana
seg_perf("-I",cadena)
```

```
cadena:  conestasrazonesperdiaelpobrecaballeroeljuicioydesvelabaseporentenderlasyd
cifrado:  zpqytidswyfwvfspeoqyytswokwyduexyunrwwauttykrztplqnxmghkvwwrssoqotkswrsn
Soluciones de Pk(k)

 Pk(a) = 0.4230769230769231 Pk(b) = 1.7307692307692308 Pk(c) = 0.46153846153846156

Soluciones de P(x)

 Px(a) = 0.11176470588235295 Px(b) = 0.029411764705882353 Px(c) = 0.035294117647058

Soluciones de P(y)

 Py(a) = 0.00013907372903912695 Py(b) = 0.00014972052169284002 Py(c) = 4.791056694

Soluciones de P(y|x)

 Pyx(a|a) = 128.6153846153839 Pyx(a|b) = 40.0 Pyx(a|c) = 33.23076923076926 Pyx(a|d
```

```
Pyx(b|a) = 394.61538461538385 Pyx(b|b) = 25.384615384615397 Pyx(b|c) = 36.0 Pyx(b
Pyx(c|a) = 70.15384615384606 Pyx(c|b) = 69.23076923076925 Pyx(c|c) = 20.307692307
Pyx(d|a) = 76.0 Pyx(d|b) = 18.461538461538456 Pyx(d|c) = 83.07692307692307 Pyx(d|
Pyx(e|a) = 102.30769230769266 Pyx(e|b) = 35.0 Pyx(e|c) = 38.769230769230695 Pyx(e
Pyx(f|a) = 284.999999999999 Pyx(f|b) = 50.00000000000006 Pyx(f|c) = 78.0 Pyx(f|d)
Pyx(g|a) = 51.15384615384603 Pyx(g|b) = 28.846153846153836 Pyx(g|c) = 23.07692307
Pyx(h|a) = 102.30769230769207 Pyx(h|b) = 13.461538461538478 Pyx(h|c) = 34.6153846
Pyx(i|a) = 70.15384615384606 Pyx(i|b) = 21.538461538461554 Pyx(i|c) = 12.92307692
Pyx(j|a) = 175.38461538461502 Pyx(j|b) = 36.923076923076856 Pyx(j|c) = 51.6923076
Pyx(k|a) = 184.15384615384656 Pyx(k|b) = 51.923076923077026 Pyx(k|c) = 49.8461538
Pyx(l|a) = 73.07692307692338 Pyx(l|b) = 53.84615384615396 Pyx(l|c) = 69.230769230
Pyx(m|a) = 143.23076923076897 Pyx(m|b) = 13.461538461538447 Pyx(m|c) = 45.2307692
Pyx(n|a) = 227.9999999999987 Pyx(n|b) = 70.00000000000009 Pyx(n|c) = 30.000000000
Pyx(o|a) = 157.84615384615236 Pyx(o|b) = 83.07692307692314 Pyx(o|c) = 116.3076923
Pyx(p|a) = 306.9230769230793 Pyx(p|b) = 34.61538461538456 Pyx(p|c) = 83.076923076
Pyx(q|a) = 305.461538461539 Pyx(q|b) = 102.30769230769207 Pyx(q|c) = 52.615384615
Pyx(r|a) = 350.76923076923174 Pyx(r|b) = 67.69230769230744 Pyx(r|c) = 103.3846153
Pyx(s|a) = 416.5384615384645 Pyx(s|b) = 109.61538461538504 Pyx(s|c) = 96.46153846
Pyx(t|a) = 305.461538461539 Pyx(t|b) = 109.61538461538504 Pyx(t|c) = 131.538461538
```

Haz doble clic (o pulsa Intro) para editar

2. Implementación del DES

2.1. Programación del DES

Pequeño resumen y esquema simplificado del DES:

Tenemos el bloque de texto plano de 64 elementos. Lo dividimos en dos bloques de 32: L y R.

Li+1=R

Ri+1:

1. R se mete en la función F:
    1. R se espande con la permutación E para llegar a 48 bits.
    2. XOR de R_expand y la K (clave)
    3. El resultado se divide en 8 trozos y a cada uno se le aplica la caja de substitución corresponiente.
    4. Se concantenan todos los resultados (obtenemos 32 bits)
    5. Se aplica la permutación P
2. Ri+1= XOR de L con el resultado de F

Esto se repite 16 veces

Esquema:

vector XOR bloque

IP (PC1 es para la clave)

16 rondas

IP-1 (PC2 es para la clave

*vector_0=VECTOR_IV Los siguientes vectores son el resultado del bloque anterior cifrado

desCBC {-C | -D -k clave -iv vectorincializacion} [-i f ilein] [-o f ileout]

-**C** el programa cifra

-**D** el programa descifra

-**k** clave de 64 bits: 56 bits + 8 bits de paridad

-**iv** vector de incialización

-**i** fichero de entrada

-**o** fichero de salida

funciones y archivos auxiliares

```
#BITS_IN_PC1=56
#BITS_IN_PC2=48
#ROUNDS=16
#BITS_IN_IP=64
#BITS_IN_E=48
#BITS_IN_P=32
NUM_S_BOXES=8
#ROWS_PER_SBOX=4
#COLUMNS_PER_SBOX=16
```

```
#Permutación PC1
PC1 =np.array([57, 49, 41, 33, 25, 17, 9,
    1, 58, 50, 42, 34, 26, 18,
    10, 2, 59, 51, 43, 35, 27,
    19, 11, 3, 60, 52, 44, 36,
    63, 55, 47, 39, 31, 23, 15,
    7, 62, 54, 46, 38, 30, 22,
    14, 6, 61, 53, 45, 37, 29,
    21, 13, 5, 28, 20, 12, 4])

#Permutación PC2
PC2= np.array([14, 17, 11, 24, 1, 5,
    3, 28, 15, 6, 21, 10,
    23, 19, 12, 4, 26, 8,
    16, 7, 27, 20, 13, 2,
    41, 52, 31, 37, 47, 55,
    30, 40, 51, 45, 33, 48,
    44, 49, 39, 56, 34, 53,
    46, 42, 50, 36, 29, 32])

#número de bits que hay que rotar cada semiclave según el número de ronda
ROUND_SHIFTS= np.array([1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1])

#Permutación IP
IP= np.array([58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17, 9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7])

#Inversa de IP
IP_INV=np.array([40, 8, 48, 16, 56, 24, 64, 32,
    39, 7, 47, 15, 55, 23, 63, 31,
    38, 6, 46, 14, 54, 22, 62, 30,
    37, 5, 45, 13, 53, 21, 61, 29,
    36, 4, 44, 12, 52, 20, 60, 28,
    35, 3, 43, 11, 51, 19, 59, 27,
    34, 2, 42, 10, 50, 18, 58, 26,
    33, 1, 41, 9, 49, 17, 57, 25])

#Expansión de E
E=np.array([32, 1, 2, 3, 4, 5,
    4, 5, 6, 7, 8, 9,
    8, 9, 10, 11, 12, 13,
    12, 13, 14, 15, 16, 17,
    16, 17, 18, 19, 20, 21,
    20, 21, 22, 23, 24, 25,
    24, 25, 26, 27, 28, 29,
    28, 29, 30, 31, 32, 1])

#Permutación P
```

```
P=np.array([16, 7, 20, 21,
    29, 12, 28, 17,
    1, 15, 23, 26,
    5, 18, 31, 10,
    2, 8, 24, 14,
    32, 27, 3, 9,
    19, 13, 30, 6,
    22, 11, 4, 25])

#Cajas S
s0=np.array([[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
            [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8 ],
            [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
            [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]])

s1=np.array([[ 15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10 ],
            [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5 ],
            [ 0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15 ],
            [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9 ]])

s2=np.array([[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
            [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
            [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7 ],
            [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]])

s3=np.array([[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
            [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9 ],
            [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4 ],
            [ 3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]])

s4=np.array([[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
            [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
            [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14 ],
            [ 11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3 ]])

s5=np.array([[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
            [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
            [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
            [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]])

s6=np.array([[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1 ],
            [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
            [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
            [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]])

s7=np.array([[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
            [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
            [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
            [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]])

S_BOXES=np.array([s0,s1,s2,s3,s4,s5,s6,s7])
```

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

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


def print_output(o,cadena):
    if o==0:
        str=""
        for i in range(len(cadena)):
            str=str+cadena[i]
        print("Cadena: {}".format(str))
    else:
        str=""
        for i in range(len(cadena)):
          str=str+cadena[i]
        file=open(o, "w")
        file.write(str)
        file.close()
```

** permutacion{ P, bloque}**

-P: la permutación actual

-bloque: el bloque de datos sobre el que vamos a operar.

En esta función realizamos la permutación P al bloque de datos y devolvemos el bloque permutado.

```
def permutacion(P,bloque):
  m=len(P)

  bloque_cifrado=[]
  for i in range(m):

    bloque_cifrado.append(bloque[P[i]-1])
  return bloque_cifrado
```

La clave es de 64 bits y tiene 8 bits de paridad en las posiciones 8,16,24,32,40,48,56,64 que se descartan. Luego el tamaño de la clave es 56.

Las permutaciones PC1 y PC2 contienen valores de 1 a 64 y no tienen los valores de los bits de paridad. El tamaño de las permutaciones es 56.

permutacion_clave_inicial{ P, bloque}

-P: la permutación PC1, el resto de permutaciones se utilizan en la función permutacion().

-bloque: el bloque actual de 64 bits sobre el que vamos a operar, en este caso se trata de la clave k en su estado inicial.

En esta función se eliminan los bits de paridad que estan en las posiciones: 8, 16, 24, 32, 40, 48, 56 y 64. Como vamos eliminado elemento mientra recorremos el vector estas posiciones varían ligeramente.

```
def permutacion_clave_inicial(P,bloque):
  paridad=[7,15,23,31,39,47,55,63]
  bloque_cifrado=[]

  i=0
  j=0
  while i <len(bloque):
    if i not in paridad: #no permutamos las posiciones de los bits de paridad
      bloque_cifrado.append(bloque[P[j]-1])
      j=j+1

    i=i+1

  return bloque_cifrado


def dividir(v):
  l=v[:32]
  r=v[32:]
  return r,l
```

sustitucion{s_box, v}

-s_box: la caja de sustitución que vamos a utilizar.

-v: uno de los 8 vectores de 6 bits sobre el que vamos a operar.

Esta función se encarga de las cajas de sustitución del DES.

Siendo v: (a0, a1, a2, a3, a4, a5).

Para obtener la fila se cogen los valores a0 y a5, para obtener la columna se obtienen los valores a1,a2,a3 y a4.

Ya que los valores de v estan en bits, se pasan a enteros y se busca la posicion en la caja de sustitución.

Fila=int(a0,a5)

Columna=int(a1,a2,a3,a4)

s_box[Fila, Columna]=x, siendo x un numero entero. Este se pasa a bits y de este forma obtenemos una solución de 4 bits que devolvemos como resultado de la función.

```
def sustitucion(s_box, v):
  fila_b="0b"+str(v[0])+str(v[len(v)-1])
  #print(fila_b)
  fila=int(fila_b,2)
  #print(fila)
  columna_b="0b"+str(v[1])+str(v[2])+str(v[3])+str(v[4])
  columna=int(columna_b,2)
  v_sub=bin(s_box[fila][columna])[2:].zfill(4)
  v_vec=list(np.array(list(str(v_sub).zfill(4))).astype(int))

  #v_sub_vec=[]
  #for i in v_sub:
   # v_sub_vec.append(int(i))

  #v_vec=[0,0,0,0]
  #i=len(v_sub_vec)-1
  #j=len(v_vec)-1
  #while i>=0:
    #v_vec[j]=v_vec[j]+v_sub_vec[i]
    #j=j-1
    #i=i-1
  return v_vec


def vec_to_int(vec):
  vec_join=""
  for i in vec:
    vec_join=vec_join+str(int(i))
  vec_join=int(vec_join)
  return vec_join


def f(r,k):
  # Permutación expansión E
  r_expand=permutacion(E,r)

  # r_expand XOR k
  y=int(str(vec_to_int(r_expand)),2)^int(str(vec_to_int(k)),2)
  y_result=bin(y)[2:].zfill(len(r_expand))
  rk=list(np.array(list(str(y_result).zfill(len(r_expand)))).astype(int))

  #rk=[]
  #for i in range(len(k)):
    #rk.append(int(r_expand[i])^k[i])
    #rk.append(int((r_expand[i]+k[i])%2))

  # s-Boxes
  i=6
  j=0
  rk_divided=[]
  boxes=[]
```

```
  while i<=48:
    #print(rk_divided.shape)
    #print(len(rk))
    rk_divided.append(rk[i-6:i])
    #print(rk_divided)
    boxes.append(sustitucion(S_BOXES[j],rk_divided[j]))
    j+=1
    i+=6
  boxes_concat=np.concatenate(boxes)
  #print("RESULTADO SUSTITUCION")
  #print(boxes_concat)

  return permutacion(P,boxes_concat)


def clave_k(num_ronda, c0, d0):

  LCS=ROUND_SHIFTS[num_ronda]
  for i in range(LCS):
    new_c0=c0[1:len(c0)]
    new_c0.append(c0[0])
    c0=new_c0
    new_d0=d0[1:len(d0)]
    new_d0.append(d0[0])
    d0=new_d0

  k_reunida=c0+d0

  #PC-2
  return permutacion(PC2,k_reunida), c0, d0




k=[]
iv=[]
for i in range(64):
  k.append(random.randint(0,1))
  iv.append(random.randint(0,1))
print(k)
print(iv)
```

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

```
def desCBC(modo, iv=0, k=0, i=0, o=0):

  if modo=="-C":
    #inputt=read_input(i)
    inputt=[0,0,0,0,0,0,0,1,0,0,1,0,0,0,1,1,0,1,0,0,0,1,0,1,0,1,1,0,0,1,1,1,1,0,0,0,1,0,0,
    #inicializacion de k
    if k==0: #la clave no se ha inicializado
      k=[]
```

```
  for i in range(64):
    k.append(random.randint(0,1))
print("K: {}".format(k))

#PC-1
k_permutada=permutacion_clave_inicial(PC1,k)


#C0 D0
c0=k_permutada[:((int(len(k_permutada)/2)))]
d0=k_permutada[int((len(k_permutada)/2)):]
#

#inicializacion de IV
if iv==0:
  iv=[]
  for i in range(64):
    iv.append(random.randint(0,1))
print("IV: {}".format(iv))


#Traducimos el inputt a binario
#cadena_numerica=[]
#for k in inputt:
#  cadena_numerica.append(list(np.binary_repr(ord(str(k)),8)))
#cadena_numerica=list(np.concatenate(cadena_numerica))
cadena_numerica=inputt
print("Vector en binario a cifrar de longitud {}bits".format(len(cadena_numerica)))

# Dividimos en bloques de 64 elementos el inputt
# Si m no es múltiplo de 64 se añade padding
n=64
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]=int(cadena_numerica[pos])
            pos=pos+1
print("Matriz de bloques a cifrar: {}".format(matriz_numerica))
# Tenemos una matriz que tenemos que cifrar.
# Cada bloque es una fila de la matriz
matriz_cifrada=[]
for v in matriz_numerica:
  print("//////////////Cifrado bloque////////////////")
  # CBC

  y=int(str(vec_to_int(v.tolist())),2)^int(str(vec_to_int(iv)),2)
  y_result=bin(y)[2:].zfill(len(iv))
  v=list(np.array(list(str(y_result).zfill(len(iv)))).astype(int))
  print("bloque XOR IV: {}".format(v))
  print(len(v))
```

```
  #Bloque de cifrado
  # IP
  v=permutacion(IP,v)

  r,l=dividir(v)
  R_rondas=[]
  R_rondas.append(r)
  L_rondas=[]
  L_rondas.append(l)
  # Empiezan 16 rondas
  for i in range(16):
    print("RONDA {}".format(i))
    #l_next=r
    L_rondas.append(R_rondas[i])
    print("L_i+1 = R_i: {}".format(R_rondas[i]))

    # r_next=l^f(r)
    k,c0,d0=clave_k(i, c0, d0)
    print("Clave K {}: {}".format(i,k))

    f_rk=f(R_rondas[i],k)
    #r_result=L_rondas[i]^f_rk
    #R_rondas.append(L_rondas[i]^f(R_rondas[i],k))

    y=int(str(vec_to_int(L_rondas[i])),2)^int(str(vec_to_int(f_rk)),2)
    y_result=bin(y)[2:].zfill(len(f_rk))
    r_result=list(np.array(list(str(y_result).zfill(len(f_rk)))).astype(int))
    R_rondas.append(r_result)
    print("R_i+1 = L_i XOR f(R_i,k): {}".format(r_result))

  rl_concat=R_rondas[len(R_rondas)-1]+L_rondas[len(L_rondas)-1]
  #IP_INV y actualizacion iv=bloque cifrado
  iv=permutacion(IP_INV,rl_concat)
  print("Bloque cifrado: {}".format(iv))
  # Inserción bloque cifrado
  matriz_cifrada.append(iv)
print("Matriz cifrada: {}".format(matriz_cifrada))

texto_cifrado=np.concatenate(matriz_cifrada)
print("Texto cifrado binario vector: {}".format(texto_cifrado.tolist()))
print("Texto cifrado binario {}".format(vec_to_int(texto_cifrado.tolist())))

texto_cifrado_letras=[]
i=0
while i<=len(texto_cifrado)-7:
  texto_cifrado_letras.append(chr(int(str(vec_to_int(texto_cifrado[i:i+8])),2)))
  i=i+8

#for elem in texto_cifrado:
# b="0b"+str(elem)
  #num=int(b,2)
  #texto_cifrado_letras.append(alfabeto[num])

print_output(o,texto_cifrado_letras)
```

```
elif modo=="-D":
  if k==0 or iv==0:
    print("K e iv deben estar inicializados para el descifrado")
    exit()
  print("K: {}".format(k))
  print("IV: {}".format(iv))

  #inputt=read_input(i)
  inputt=[1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1,

  #PC-1
  k_permutada=permutacion_clave_inicial(PC1,k)


  #C0 D0
  c0=k_permutada[:((int(len(k_permutada)/2)))]
  d0=k_permutada[int((len(k_permutada)/2)):]

  # Generamos todas las claves
  claves=[]
  for i in range(16):
    k,c0,d0=clave_k(i, c0, d0)
    claves.append(k)

  #Traducimos el inputt a binario
  #cadena_numerica=[]
  #for k in inputt:
   # cadena_numerica.append(list(np.binary_repr(ord(str(k)),8)))
  #cadena_numerica=list(np.concatenate(cadena_numerica))

  cadena_numerica=inputt
  print("Vector en binario a descifrar de longitud {}bits".format(len(cadena_numerica)))

  # Dividimos en bloques de 64 elementos el inputt
  # Si m no es múltiplo de 64 se añade padding
  n=64
  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]=int(cadena_numerica[pos])
              pos=pos+1
  print("Matriz de bloques a descifrar: {}".format(matriz_numerica))
  # Tenemos una matriz que tenemos que descifrar.
  # Cada bloque es una fila de la matriz
  matriz_cifrada=[]
  for v in matriz_numerica:
    print("//////////////Cifrado bloque////////////////")
    # CBC

    #Bloque de cifrado
```

```
  # IP
  v_ip=permutacion(IP,v)

  r,l=dividir(v_ip)
  R_rondas=[]
  R_rondas.append(r)
  L_rondas=[]
  L_rondas.append(l)
  # Empiezan 16 rondas
  for i in range(16):
    print("RONDA {}".format(i))
    #l_next=r
    L_rondas.append(R_rondas[i])
    print("L_i+1 = R_i: {}".format(R_rondas[i]))


    #Aplicamos las claves en el orden inverso
    print("Clave K {}: {}".format((15-i),claves[15-i]))

    # r_next=l^f(r)
    f_rk=f(R_rondas[i],claves[15-i])
    #r_result=L_rondas[i]^f_rk
    #R_rondas.append(L_rondas[i]^f(R_rondas[i],k))

    y=int(str(vec_to_int(L_rondas[i])),2)^int(str(vec_to_int(f_rk)),2)
    y_result=bin(y)[2:].zfill(len(f_rk))
    r_result=list(np.array(list(str(y_result).zfill(len(f_rk)))).astype(int))
    R_rondas.append(r_result)
    print("R_i+1 = L_i XOR f(R_i,k): {}".format(r_result))

  rl_concat=R_rondas[len(R_rondas)-1]+L_rondas[len(L_rondas)-1]
  #IP_INV
  output=permutacion(IP_INV,rl_concat)

  # CBC: bloque descifrado = iv XOR output
  y=int(str(vec_to_int(output)),2)^int(str(vec_to_int(iv)),2)
  y_result=bin(y)[2:].zfill(len(iv))
  v_descifrado=list(np.array(list(str(y_result).zfill(len(iv)))).astype(int))
  #Actualizacion de iv
  iv=v

  print("Bloque descifrado: {}".format(v_descifrado))
  # Inserción bloque descifrado
  matriz_cifrada.append(v_descifrado)
print("Matriz descifrada: {}".format(matriz_cifrada))

texto_cifrado=np.concatenate(matriz_cifrada)
print("Texto cifrado binario {}".format(vec_to_int(texto_cifrado.tolist())))

texto_cifrado_letras=[]
i=0
while i<=len(texto_cifrado)-7:
  texto_cifrado_letras.append(chr(int(str(vec_to_int(texto_cifrado[i:i+8])),2)))
  i=i+8
```

```
    #for elem in texto_cifrado:
    # b="0b"+str(elem)
      #num=int(b,2)
      #texto_cifrado_letras.append(alfabeto[num])

    print_output(o,texto_cifrado_letras)
```

Comprobamos con el ejemplo que se generan correctamente las claves

```
# Comprobamos que se generan bien las claves con el ejemplo
K_ejemplo=[0,0,0,1,0,0,1,1,0,0,1,1,0,1,0,0,0,1,0,1,0,1,1,1,0,1,1,1,1,0,0,1,1,0,0,1,1,0,1,1
print("K= {}".format(vec_to_int(K_ejemplo)))
#PC-1
k_permutada=permutacion_clave_inicial(PC1,K_ejemplo)
print("k+= {}".format(vec_to_int(k_permutada)))
#C0 D0
c0=k_permutada[:((int(len(k_permutada)/2)))]
d0=k_permutada[int((len(k_permutada)/2)):]
print("c0= {}".format(vec_to_int(c0)))
print("d0= {}".format(vec_to_int(d0)))
# Generamos todas las claves
claves=[]
for i in range(16):
  k,c0,d0=clave_k(i, c0, d0)
  claves.append(k)
  print("////////////////////////////////////////////////////")
  print("c{}= {}".format(i,vec_to_int(c0)))
  print("d{}= {}".format(i,vec_to_int(d0)))
  print("k{}= {}".format(i, vec_to_int(k)))
```

```
    d1= 10101011001100111100011101
    k1= 111100110101110110110011101101111001001111001O1
    ////////////////////////////////////////////////////
    c2= 110011001010101011111111
    d2= 101011001100111100011110101
    k2= 101010111111100100010100100001011001111100110O1
    ////////////////////////////////////////////////////
    c3= 110011001010101011111111100
    d3= 101100110011110001111010101
    k3= 111001010101101110101101101101100110101010011101
    ////////////////////////////////////////////////////
    c4= 110011001010101011111111110000
    d4= 110011001111000111101010101
    k4= 111110011101100000001111110101101010011101010O0O
    ////////////////////////////////////////////////////
    c5= 110010101010111111111000011
    d5= 100110011110001111010101010101
    k5= 110001110100101001111100101000001111011001011111
    ////////////////////////////////////////////////////
    c6= 110010101010111111111000011OO
    d6= 110011110001111010101010110
    k6= 111011001000010010110111111101100001100010111100
    ////////////////////////////////////////////////////
    c7= 101010101111111110000110011
    d7= 100111100011110101010110011
    k7= 111101111000101000111010110000010011101111111011
    ////////////////////////////////////////////////////
```

```
.....................................................
c8= 10101010111111100001100110
d8= 11110001111010101010110011
k8= 11100000110110111110101111101101111001111000001
////////////////////////////////////////////////////
c9= 10101011111111000011001001
d9= 11110001111010101011001100
k9= 10110001111100110100011110111010010001100100111
////////////////////////////////////////////////////
c10= 10101111111100001100100101
d10= 11000111101010101100110011
k10= 10000101011111110100111011110110100111000011O
////////////////////////////////////////////////////
c11= 10111111110000110011001010l
d11= 11110101010110011001111
k11= 11101010111000111110101100101000110011111101001
////////////////////////////////////////////////////
c12= 11111111000011001100101010l
d12= 11110101010110011001111100
k12= 10010111110001011101000111111010101110100100000l
////////////////////////////////////////////////////
c13= 11111110000110011001010101O1
d13= 11101010101011001100111100O1
k13= 10111110100001110110111111100101110011100111010
////////////////////////////////////////////////////
c14= 11111000011001100101010101l1
d14= 10101010101100110011110001l1
k14= 10111111100100011000110100111101001111110000l010
////////////////////////////////////////////////////
c15= 11110000110011001010101011l1
d15= 10101010110011001111000l1l1
k15= 11001011001111011000101100001110000101111111010l
```

Comprobamos el cifrado siguiendo paso a paso el ejemplo

```
M=[0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,1,1,0,1,0,0,0,1,0,1,0,1,1,0,0,1,1,1,1,0,0,0,1,0,0,1,1,0,1,0
print("M = {}".format(vec_to_int(M)))

# Permutación IP
M_IP=permutacion(IP,M)
print("IP = {}".format(vec_to_int(M_IP)))

#
r,l=dividir(M_IP)
R_rondas=[]
R_rondas.append(r)
L_rondas=[]
L_rondas.append(l)
print("L0 = {}".format(vec_to_int(l)))
print("R0 = {}".format(vec_to_int(r)))
# Empiezan 16 rondas
for i in range(16):
  print("RONDA {}".format(i+1))
  #l_next=r
  L_rondas.append(R_rondas[i])

  f_rk=f(R_rondas[i],claves[i])
```

```
  y=int(str(vec_to_int(L_rondas[i])),2)^int(str(vec_to_int(f_rk)),2)
  y_result=bin(y)[2:].zfill(len(f_rk))
  r_result=list(np.array(list(str(y_result).zfill(len(f_rk)))).astype(int))
  R_rondas.append(r_result)
  print("L{} = {}".format((i+1),vec_to_int(L_rondas[i+1])))
  print("R{} = {}".format((i+1),vec_to_int(R_rondas[i+1])))

rl_concat=R_rondas[len(R_rondas)-1]+L_rondas[len(L_rondas)-1]
print("R16 L16 = {}".format(vec_to_int(rl_concat)))
IPINV=permutacion(IP_INV,rl_concat)
print("IP inversa = {}".format(vec_to_int(IPINV)))
```

```
M = 1001000110100010101100111100010011010101111001101111011111
IP = 1100110000000000110011001111111111110000101010101111000010101010
L0 = 11001100000000000110011001111111
R0 = 11110000101010101111000010101010
RONDA 1
L1 = 11110000101010101111000010101010
R1 = 11101111010010100110010101000100
RONDA 2
L2 = 11101111010010100110010101000100
R2 = 11001100000000010111011100001001
RONDA 3
L3 = 11001100000000010111011100001001
R3 = 10100010010111000000101111110100
RONDA 4
L4 = 10100010010111000000101111110100
R4 = 11101110010001000000000001000101
RONDA 5
L5 = 11101110010001000000000001000101
R5 = 10001010010011111010011000110111
RONDA 6
L6 = 10001010010011111010011000110111
R6 = 11101001011001111100110101101001
RONDA 7
L7 = 11101001011001111100110101101001
R7 = 11001001010101110100010000
RONDA 8
L8 = 11001001010101110100010000
R8 = 11010101011010010100101110010000
RONDA 9
L9 = 11010101011010010100101110010000
R9 = 10010001111100110001100111010
RONDA 10
L10 = 10010001111100110001100111010
R10 = 10110111110101011101011110110010
RONDA 11
L11 = 10110111110101011101011110110010
R11 = 11000101011110000011110001111000
RONDA 12
L12 = 11000101011110000011110001111000
R12 = 11101011011110100011000010110000
RONDA 13
L13 = 11101011011110100011000010110000
R13 = 11000110000110001010101011010
RONDA 14
L14 = 11000110000110001010101011010
```

```
R14 = 11000010100011001001011000001101
RONDA 15
L15 = 11000010100011001001011000001101
R15 = 10000110100001000110010001101 00
RONDA 16
L16 = 10000110100001000110010001101 00
R16 = 10100100110011011001100101010100001101000010001100100011 0100
IP inversa = 10000101111010000001001101010100000011110000101010110100000000101
```

Comparando con el ejemplo, concluimos que cada uno de los valores obtenidos coincide con el ejemplo y el DES implementado funciona correctamente.

Ahora fijamos un vector inicial y probamos a cifrar y descifrar un bloque: 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

```
iv=K_ejemplo
```

```
desCBC("-C",K_ejemplo,iv)
```

```
RONDA 3
L_i+1 = R_i: [0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0,
Clave K 3: [0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0
R_i+1 = L_i XOR f(R_i,k): [0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1
RONDA 4
L_i+1 = R_i: [0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0,
Clave K 4: [0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1
R_i+1 = L_i XOR f(R_i,k): [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1
RONDA 5
L_i+1 = R_i: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0,
Clave K 5: [0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0
R_i+1 = L_i XOR f(R_i,k): [1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0
RONDA 6
L_i+1 = R_i: [1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1,
Clave K 6: [1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1
R_i+1 = L_i XOR f(R_i,k): [0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0
RONDA 7
L_i+1 = R_i: [0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0,
Clave K 7: [1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0
R_i+1 = L_i XOR f(R_i,k): [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1
RONDA 8
L_i+1 = R_i: [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0,
Clave K 8: [1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1
R_i+1 = L_i XOR f(R_i,k): [0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1
RONDA 9
L_i+1 = R_i: [0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0,
Clave K 9: [1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1
R_i+1 = L_i XOR f(R_i,k): [1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1
RONDA 10
L_i+1 = R_i: [1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0,
Clave K 10: [0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1,
R_i+1 = L_i XOR f(R_i,k): [0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1
RONDA 11
L_i+1 = R_i: [0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0,
Clave K 11: [0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0,
R_i+1 = L_i XOR f(R_i,k): [1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1
RONDA 12
```

```
RONDA 12
L_i+1 = R_i: [1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0,
Clave K 12: [1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0,
R_i+1 = L_i XOR f(R_i,k): [0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0
RONDA 13
L_i+1 = R_i: [0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0,
Clave K 13: [0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1,
R_i+1 = L_i XOR f(R_i,k): [0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1
RONDA 14
L_i+1 = R_i: [0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1,
Clave K 14: [1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0,
R_i+1 = L_i XOR f(R_i,k): [1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1
RONDA 15
L_i+1 = R_i: [1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0,
Clave K 15: [1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1,
R_i+1 = L_i XOR f(R_i,k): [1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0
Bloque cifrado: [1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0,
Matriz cifrada: [[1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0
Texto cifrado binario vector: [1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, (
Texto cifrado binario 1000111111010000101010110100100110110010000010000000001011010
Cadena:  Đ«I Ó
```

desCBC("-D",K_ejemplo,iv)

```
RONDA 3
L_i+1 = R_i: [1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0,
Clave K 12: [1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0,
R_i+1 = L_i XOR f(R_i,k): [0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1
RONDA 4
L_i+1 = R_i: [0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0,
Clave K 11: [0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0,
R_i+1 = L_i XOR f(R_i,k): [1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1
RONDA 5
L_i+1 = R_i: [1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0,
Clave K 10: [0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1,
R_i+1 = L_i XOR f(R_i,k): [0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1
RONDA 6
L_i+1 = R_i: [0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0,
Clave K 9: [1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1

R_i+1 = L_i XOR f(R_i,k): [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1
RONDA 7
L_i+1 = R_i: [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0,
Clave K 8: [1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1
R_i+1 = L_i XOR f(R_i,k): [0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0
RONDA 8
L_i+1 = R_i: [0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0,
Clave K 7: [1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0
R_i+1 = L_i XOR f(R_i,k): [1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0
RONDA 9
L_i+1 = R_i: [1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1,
Clave K 6: [1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1
R_i+1 = L_i XOR f(R_i,k): [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1
RONDA 10
L_i+1 = R_i: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0,
Clave K 5: [0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0
R_i+1 = L_i XOR f(R_i,k): [0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1
RONDA 11
L_i+1 = R_i: [0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0,
```

```
Clave K 4: [0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1
R_i+1 = L_i XOR f(R_i,k): [0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1
RONDA 12
L_i+1 = R_i: [0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0,
Clave K 3: [0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0
R_i+1 = L_i XOR f(R_i,k): [0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0
RONDA 13
L_i+1 = R_i: [0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0,
Clave K 2: [0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0
R_i+1 = L_i XOR f(R_i,k): [1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1
RONDA 14
L_i+1 = R_i: [1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1,
Clave K 1: [0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1
R_i+1 = L_i XOR f(R_i,k): [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0
RONDA 15
L_i+1 = R_i: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
Clave K 0: [0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1
R_i+1 = L_i XOR f(R_i,k): [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1
Bloque descifrado: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0,
Matriz descifrada: [[0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0
Texto cifrado binario 100100011010001010110011110001001101010111100110111101111
Cadena:  #Eg «Íï
```

Observamos que se verifica la inyectividad del criptosistema, es decir, al cifrar obtenemos la cadena cifrada Y. Al descifrar Y obtenemos la cadena original 000000010010001101000101011001111000100110101011110011011110111 1

En el modo de operación CBC cada bloque de texto cifrado depende de todo el texto plano procesado hasta ese punto. Además el vector de inicialización hace que el mensaje cifrado sea único. Por otro lado,el modo de operación ECB cifra los bloques de forma independiente. ésto no supone problema especial mente si el mensaje es corto y tiende a la longitud del bloque, pero para mensajes largos y estructurados, no es una buena opción porque éstos pueden tener patrones grandes como una imagen que tiene un área grande de un color uniforme. En estos casos, el cifrado en el modo ECB encripta el color de cada píxel pero la imagen general aún se puede discernir porque el patron de los píxeles del mismo color del área de color uniforme permanece en la imagen cifrada.

3. Principios de diseño del DES

3.1 Estudio de la no linealidad de las S-Boxes del DES

La no linealidad es la propiedad F(x XOR y) != F(x) XOR F(y)

Para comprobarla, probamos todas las cajas de sustitución sobre todos los valores de 6 bits (64) y comprobamos la condición de no linealidad. En total probamos un total de 4096 posibilidades.

```
frecuencia=[]
for i in range(NUM_S_BOXES):
  frecuencia.append(0)
  x=0
  for j in range(64):
    y=0
    for k in range(64):
      z=(x^y)
      x_vec=str(np.binary_repr(x,6))
      y_vec=str(np.binary_repr(y,6))
      z_vec=str(np.binary_repr(z,6))

      s_x=sustitucion(S_BOXES[i],x_vec)
      s_y=sustitucion(S_BOXES[i],y_vec)
      s_z=sustitucion(S_BOXES[i],z_vec)

      if vec_to_int(s_z)==(vec_to_int(s_x)^vec_to_int(s_y)):
        print("Se cumple linealidad para x={}, y={}".format(x,y))
        frecuencia[i]=frecuencia[i]+1

      y=y+1
    x=x+1
```

```
Se cumple linealidad para x=27, y=55
Se cumple linealidad para x=27, y=56
Se cumple linealidad para x=29, y=35
Se cumple linealidad para x=29, y=62
Se cumple linealidad para x=30, y=35
Se cumple linealidad para x=30, y=36
Se cumple linealidad para x=30, y=58
Se cumple linealidad para x=30, y=61
Se cumple linealidad para x=31, y=38
Se cumple linealidad para x=31, y=57
Se cumple linealidad para x=33, y=8
Se cumple linealidad para x=33, y=41
Se cumple linealidad para x=35, y=23
Se cumple linealidad para x=35, y=27
Se cumple linealidad para x=35, y=29
Se cumple linealidad para x=35, y=30
Se cumple linealidad para x=35, y=52
Se cumple linealidad para x=35, y=56
Se cumple linealidad para x=35, y=61
Se cumple linealidad para x=35, y=62
Se cumple linealidad para x=36, y=30
Se cumple linealidad para x=36, y=58
Se cumple linealidad para x=37, y=3
Se cumple linealidad para x=37, y=38
Se cumple linealidad para x=38, y=3
Se cumple linealidad para x=38, y=10
Se cumple linealidad para x=38, y=31
Se cumple linealidad para x=38, y=37
Se cumple linealidad para x=38, y=44
Se cumple linealidad para x=38, y=57
Se cumple linealidad para x=41, y=8
Se cumple linealidad para x=41, y=33
Se cumple linealidad para x=42, y=25
```

```
    Se cumple linealidad para x=42, y=51
    Se cumple linealidad para x=44, y=10
    Se cumple linealidad para x=44, y=27
    Se cumple linealidad para x=44, y=38
    Se cumple linealidad para x=44, y=55
    Se cumple linealidad para x=47, y=25
    Se cumple linealidad para x=47, y=54
    Se cumple linealidad para x=51, y=25
    Se cumple linealidad para x=51, y=42
    Se cumple linealidad para x=52, y=23
    Se cumple linealidad para x=52, y=35
    Se cumple linealidad para x=54, y=25
    Se cumple linealidad para x=54, y=47
    Se cumple linealidad para x=55, y=27
    Se cumple linealidad para x=55, y=44
    Se cumple linealidad para x=56, y=27
    Se cumple linealidad para x=56, y=35
    Se cumple linealidad para x=57, y=31
    Se cumple linealidad para x=57, y=38
    Se cumple linealidad para x=58, y=30
    Se cumple linealidad para x=58, y=36
    Se cumple linealidad para x=61, y=30
    Se cumple linealidad para x=61, y=35
    Se cumple linealidad para x=62, y=29
    Se cumple linealidad para x=62, y=35
```

```
for i in range(NUM_S_BOXES):
  frecuencia[i]=(1-(frecuencia[i]/(64*64)))*100
  print("S-Box {} tiene {}% de no linealidad".format(i,frecuencia[i]))
```

```
    S-Box 0 tiene 98.828125% de no linealidad
    S-Box 1 tiene 98.2421875% de no linealidad
    S-Box 2 tiene 97.94921875% de no linealidad
    S-Box 3 tiene 97.94921875% de no linealidad
    S-Box 4 tiene 98.681640625% de no linealidad
    S-Box 5 tiene 97.216796875% de no linealidad
    S-Box 6 tiene 98.53515625% de no linealidad
    S-Box 7 tiene 97.94921875% de no linealidad
```

Observamos un porcentaje de no linealidad elevado para todas las cajas (entre 97 y 98%), por lo que podemos concluir que son lo suficientemente no lineales.

3.2 Estudio del Efecto Avalancha

Para probar el efecto avalancha con las claves realizaremos el DES a un bloque con dos claves (diferentes en un bit) y compararemos la diferencia los resultados en cada paso del algoritmo. Análogamente, para probar el efecto avalancha con los bloques, realizaremos el DES a dos bloques (diferentes en un bit) con una clave fija y compararemos la diferencia de los resultados en cada paso del algoritmo.

```
def cambiar_bit(v):
```

```
  pos=random.randint(0,len(v)-1)
  v_mod=[]
  for i in range(len(v)):
    if i==pos:
      if v[pos]==0:
        v_mod.append(1)
      elif v[pos]==1:
        v_mod.append(0)
    else:
      v_mod.append(v[i])

  return v_mod


def contador_bits_diferentes(v1,v2):
  count=0
  if len(v1)==len(v2):
    for i in range(len(v1)):
      if v1[i] != v2[i]:
        count=count+1
  print("Tienen {} diferentes".format(count))


def cifrado_des_claves(bloque, k_input, k_cambiado):
  # Generación de claves k_input
  #PC-1
  k_permutada=permutacion_clave_inicial(PC1,k_input)
  #C0 D0
  c0=k_permutada[:((int(len(k_permutada)/2)))]
  d0=k_permutada[int((len(k_permutada)/2)):]
  # Generamos todas las claves
  claves=[]
  for i in range(16):
    k,c0,d0=clave_k(i, c0, d0)
    claves.append(k)

  # Generación de claves k_cambiado
  #PC-1
  k_permutada=permutacion_clave_inicial(PC1,k_cambiado)
  #C0 D0
  c0=k_permutada[:((int(len(k_permutada)/2)))]
  d0=k_permutada[int((len(k_permutada)/2)):]
  # Generamos todas las claves
  claves_mod=[]
  for i in range(16):
    k,c0,d0=clave_k(i, c0, d0)
    claves_mod.append(k)

  # Cifrado

  # Permutación IP
  M_IP=permutacion(IP,bloque)
  print("IP = {}".format(vec_to_int(M_IP)))

  #
```

```
r,l=dividir(M_IP)
R_rondas=[]
R_rondas_mod=[]
R_rondas.append(r)
R_rondas_mod.append(r)
L_rondas=[]
L_rondas_mod=[]
L_rondas.append(l)
L_rondas_mod.append(l)

print("Bloque 0: {}".format(vec_to_int(l+r)))
print("Bloque modificado 0: {}".format(vec_to_int(l+r)))
contador_bits_diferentes(l+r, l+r)
# Empiezan 16 rondas
for i in range(16):
  print("RONDA {}".format(i+1))
  # claves
  L_rondas.append(R_rondas[i])

  f_rk=f(R_rondas[i],claves[i])
  y=int(str(vec_to_int(L_rondas[i])),2)^int(str(vec_to_int(f_rk)),2)
  y_result=bin(y)[2:].zfill(len(f_rk))
  r_result=list(np.array(list(str(y_result).zfill(len(f_rk)))).astype(int))
  R_rondas.append(r_result)

  print("Bloque {}: {}".format((i+1),vec_to_int(L_rondas[i+1]+R_rondas[i+1])))

  # claves_mod
  L_rondas_mod.append(R_rondas_mod[i])

  f_rk_mod=f(R_rondas_mod[i],claves_mod[i])
  y=int(str(vec_to_int(L_rondas_mod[i])),2)^int(str(vec_to_int(f_rk_mod)),2)
  y_result=bin(y)[2:].zfill(len(f_rk_mod))
  r_result=list(np.array(list(str(y_result).zfill(len(f_rk_mod)))).astype(int))
  R_rondas_mod.append(r_result)

  print("Bloque modificado {}: {}".format((i+1),vec_to_int(L_rondas_mod[i+1]+R_rondas_mo

  contador_bits_diferentes(L_rondas[i+1]+R_rondas[i+1],L_rondas_mod[i+1]+R_rondas_mod[i+

rl_concat=R_rondas[len(R_rondas)-1]+L_rondas[len(L_rondas)-1]
print("Bloque 16 swap: {}".format(vec_to_int(rl_concat)))

rl_concat_mod=R_rondas_mod[len(R_rondas_mod)-1]+L_rondas_mod[len(L_rondas_mod)-1]
print("Bloque modificado 16 swap: {}".format(vec_to_int(rl_concat_mod)))

contador_bits_diferentes(rl_concat,rl_concat_mod)

IPINV=permutacion(IP_INV,rl_concat)
print("IP inversa = {}".format(vec_to_int(IPINV)))

IPINV_mod=permutacion(IP_INV,rl_concat_mod)
print("IP inversa = {}".format(vec_to_int(IPINV_mod)))
```

```
  contador_bits_diferentes(IPINV,IPINV_mod)


def cifrado_des_bloques(bloque, bloque_cambiado, k_input):
  # Generación de claves k_input
  #PC-1
  k_permutada=permutacion_clave_inicial(PC1,k_input)
  #C0 D0
  c0=k_permutada[:((int(len(k_permutada)/2)))]
  d0=k_permutada[int((len(k_permutada)/2)):]
  # Generamos todas las claves
  claves=[]
  for i in range(16):
    k,c0,d0=clave_k(i, c0, d0)
    claves.append(k)

  # Cifrado

  # Permutación IP
  M_IP=permutacion(IP,bloque)
  print("IP = {}".format(vec_to_int(M_IP)))

  M_IP_mod=permutacion(IP,bloque_cambiado)
  print("IP modificado= {}".format(vec_to_int(M_IP_mod)))
  contador_bits_diferentes(M_IP,M_IP_mod)

  #
  r,l=dividir(M_IP)
  r_mod,l_mod=dividir(M_IP_mod)
  R_rondas=[]
  R_rondas_mod=[]
  R_rondas.append(r)
  R_rondas_mod.append(r_mod)
  L_rondas=[]
  L_rondas_mod=[]
  L_rondas.append(l)
  L_rondas_mod.append(l_mod)

  print("Bloque 0: {}".format(vec_to_int(l+r)))
  print("Bloque modificado 0: {}".format(vec_to_int(l_mod+r_mod)))
  contador_bits_diferentes(l+r, l_mod+r_mod)

  # Empiezan 16 rondas
  for i in range(16):
    print("RONDA {}".format(i+1))
    # bloque
    L_rondas.append(R_rondas[i])

    f_rk=f(R_rondas[i],claves[i])
    y=int(str(vec_to_int(L_rondas[i])),2)^int(str(vec_to_int(f_rk)),2)
    y_result=bin(y)[2:].zfill(len(f_rk))
    r_result=list(np.array(list(str(y_result).zfill(len(f_rk)))).astype(int))
    R_rondas.append(r_result)

    print("Bloque {}: {}".format((i+1),vec_to_int(L_rondas[i+1]+R_rondas[i+1])))
```

```
    # bloque mod
    L_rondas_mod.append(R_rondas_mod[i])

    f_rk_mod=f(R_rondas_mod[i],claves[i])
    y=int(str(vec_to_int(L_rondas_mod[i])),2)^int(str(vec_to_int(f_rk_mod)),2)
    y_result=bin(y)[2:].zfill(len(f_rk_mod))
    r_result=list(np.array(list(str(y_result).zfill(len(f_rk_mod)))).astype(int))
    R_rondas_mod.append(r_result)

    print("Bloque modificado {}: {}".format((i+1),vec_to_int(L_rondas_mod[i+1]+R_rondas_mo

    contador_bits_diferentes(L_rondas[i+1]+R_rondas[i+1],L_rondas_mod[i+1]+R_rondas_mod[i+

  rl_concat=R_rondas[len(R_rondas)-1]+L_rondas[len(L_rondas)-1]
  print("Bloque 16 swap: {}".format(vec_to_int(rl_concat)))

  rl_concat_mod=R_rondas_mod[len(R_rondas_mod)-1]+L_rondas_mod[len(L_rondas_mod)-1]
  print("Bloque modificado 16 swap: {}".format(vec_to_int(rl_concat_mod)))

  contador_bits_diferentes(rl_concat,rl_concat_mod)

  IPINV=permutacion(IP_INV,rl_concat)
  print("IP inversa = {}".format(vec_to_int(IPINV)))

  IPINV_mod=permutacion(IP_INV,rl_concat_mod)
  print("IP inversa = {}".format(vec_to_int(IPINV_mod)))

  contador_bits_diferentes(IPINV,IPINV_mod)



def avalancha(modo):
  # Generamos una clave aleatoria y un bloque aleatorios
  k=[]
  b=[]
  for i in range(64):
    k.append(random.randint(0,1))
    b.append(random.randint(0,1))

  # Efecto avalancha de la clave
  if modo=="-K":
    k_cambiado=cambiar_bit(k)
    print("Clave: {}".format(k))
    print("clave modificada: {}".format(k_cambiado))
    contador_bits_diferentes(k,k_cambiado)
    cifrado_des_claves(b,k,k_cambiado)

  elif modo=="-B":
    b_cambiado=cambiar_bit(b)
    print("Bloque: {}".format(b))
    print("Bloque modificado: {}".format(b_cambiado))
    contador_bits_diferentes(b,b_cambiado)
    cifrado_des_bloques(b,b_cambiado,k)
  else:
```

```
    print("Parámetros erróneos. -K para clave y -B para bloque")



avalancha("-K")
```

```
Bloque 4: 10100110100111001101000100000001100000100011010001010011110 01
Bloque modificado 4: 11011100011010101001011101010111001111110101001010010001100 01
Tienen 37 diferentes
RONDA 5
Bloque 5: 11000001000110100010100111100100001001111100111001110111011 10001
Bloque modificado 5: 11001111110101001010010001100011000101111011001010011100111 10
Tienen 29 diferentes
RONDA 6
Bloque 6: 10011111001111001110111100010111000111100101000110101101 11110
Bloque modificado 6: 10111101100101001110011110100010011101011001010110110011011 01010
Tienen 30 diferentes
RONDA 7
Bloque 7: 11100011110010100011010110111100001010011010111011101 01000111
Bloque modificado 7: 10011101011001010110110011010100011011000000000110001011 11000
Tienen 36 diferentes
RONDA 8
Bloque 8: 10100110101110111101011000111000100110110011001100100 10110000011
Bloque modificado 8: 11011000000000110001011110001111011100110001000110000010000 11(
Tienen 30 diferentes
RONDA 9
Bloque 9: 10011011001100110010110000011010110110001000101010011 00011100
Bloque modificado 9: 11011100110001000110000010000110111110110001110101010000000 111
Tienen 21 diferentes
RONDA 10
Bloque 10: 10110110001000101010011000111000110001000111100000111 0100000101
Bloque modificado 10: 11111011000111010101000000111110101001011110110110110111001 00(
Tienen 24 diferentes
RONDA 11
Bloque 11: 11000100011110000011101000001010111010010001111010101 01101001
Bloque modificado 11: 10100101111011011011011100100011101011111100110110101110111 11
Tienen 30 diferentes
RONDA 12
Bloque 12: 11101001000111101010101011010011101010111001011110010110001 1011
Bloque modificado 12: 10101111111001101101011101111000110001111010001110000000111 11
Tienen 27 diferentes
RONDA 13
Bloque 13: 11010101110010111100101100011011111000001100111000101000011 11000
Bloque modificado 13: 11000111101000111000000001111000001110111100110010011111010 101
Tienen 28 diferentes
RONDA 14
Bloque 14: 11100000110011100010100001111000111001011100010100011001 11001011
Bloque modificado 14: 11101111001100100111110101100111011010000100000000110001111 10
Tienen 32 diferentes
RONDA 15
Bloque 15: 11100101110001010001100111001011110111000010101110011111011000101
Bloque modificado 15: 11011010000100000000110001111011010001010111111111110001011 00(
Tienen 35 diferentes
RONDA 16
Bloque 16: 11011100001010111001111011000101100000110011001111011101100 10000
Bloque modificado 16: 10001010111111111111000101100010110111100101011010110101011 11
Tienen 35 diferentes
Bloque 16 swap: 10000011001100111101110110010000110111000010101110011111011000101
Bloque modificado 16 swap: 11011110010101101011010101011101010001010111111111111000 11
Tienen 35 diferentes
```

```
Tienen 39 diferentes
IP inversa = 1110110011110001000111010101100100111010011000010000011011001111
IP inversa = 1010110101110010111101010110000101111101001011101111101101001100
Tienen 35 diferentes
```

Observamos el efecto avalancha al aplicar el des sobre el mismo bloque con dos claves distintas, ya que va incrementando el número de bits diferentes entre el bloque cifrado con la clave k y el bloque cifrado con la clave modificada 1 bit. A medida que van pasando las rondas hay un mayor número de bits diferentes hasta que en el bloque cifrado final se observan en torno a 39 bits diferentes, teniendo en cuenta que el bloque es de 64 bits, se cambian más del 60% de los bits, lo que quiere decir que se propaga por todo el bloque.

```
avalancha("-B")
```

```
Bloque 4: 1010000111101101010101010101011011101111101011010111011001011101
Bloque modificado 4: 1011100111100101011110011101110000011110010001100111101011111
Tienen 24 diferentes
RONDA 5
Bloque 5: 1110111110101101011101100101110110000110011010010001001100101100
Bloque modificado 5: 1111001000110011110101111110110111110010010100001111000000 1
Tienen 36 diferentes
RONDA 6
Bloque 6: 1000011001101001000100110010110010000010000101111110001010111110
Bloque modificado 6: 1101111100100101000011110000001000111101111010101010011010011
Tienen 37 diferentes
RONDA 7
Bloque 7: 1000001000010111111000101011111010111110100001110001010110111101
Bloque modificado 7: 1111011110101010011010011110111100100110100000101010111101 0
Tienen 32 diferentes
RONDA 8
Bloque 8: 1011111010000111000101011011110110111111000000000011110000011010
Bloque modificado 8: 1111001001101000001010101111010000001001000110000110100010110
Tienen 27 diferentes
RONDA 9
Bloque 9: 1011111100000000001111000001101001100111111111110000011000100 0100
Bloque modificado 9: 1001000110000110100010110111101011001100010110110101101 1001
Tienen 30 diferentes
RONDA 10
Bloque 10: 1100111111111100000110001000100011101011011011101111100010 1010
Bloque modificado 10: 1101011001100010110110101101100101001010100011111011001111100
Tienen 32 diferentes
RONDA 11
Bloque 11: 1110101101101110110111110001010010000110001010110010100110 10000
Bloque modificado 11: 1001010100011110110011111000111100111011100011101101010001100
Tienen 35 diferentes
RONDA 12
Bloque 12: 1000011000101011001010011010000010010101100100101000110000 11100
Bloque modificado 12: 1001110111000111011010100011001010100010011000010000000100010
Tienen 35 diferentes
RONDA 13
Bloque 13: 1001010110010010100011000011100110111001001110001101000100 00101
Bloque modificado 13: 1010001001100001000000010001011111100010011000000001011000000
Tienen 33 diferentes
RONDA 14
Bloque 14: 1101110010011100011010001000010111000000001100110001011010010100
Bloque modificado 14: 1110001001100000000101100000110100110110110100001001001010100
```

```
Tienen 37 diferentes
RONDA 15
Bloque 15: 1100000000110011000101101001010001110011000001110000010010111011
Bloque modificado 15: 1101101101000010010010101000111010110111000111111010110100100
Tienen 38 diferentes
RONDA 16
Bloque 16: 1110011000001110000010010111011111000001110111011110001000100010
Bloque modificado 16: 1010110111000111111010110100100100010111111001110011011001010
Tienen 37 diferentes
Bloque 16 swap: 1110000011101110111100010001001001110011000001110000010010111011
Bloque modificado 16 swap: 1011111100111001101100101000110101101110001111110101101010
Tienen 37 diferentes
IP inversa = 1010011010110011001110000001001010000111110101101101010001010110
IP inversa = 1111101101111100111101001000101001000101100111000011101110111000
Tienen 37 diferentes
```

También observamos el efecto avalancha cuando ciframos dos bloques que se diferencian en 1 bit. A medida que van pasando las rondas primeras rondas se van incrementando el número de bits diferentes, hasta la tercera ronda, a partir de la cual se mantienen entre 26 y 35 bits diferentes en cada ronda.