Formative Assignment

Exercise 1:

Answer:

Decrypted, the result is ALICELOVESBOBBUTBOBDOESNOTLOVEALICE and the key number is 5

Code Implemented (Python):

```
import numpy as np
import math as m
import binascii as biasc
import cryptography as crypto

class RailFence:
    def initialise():
        print("Hello World")
        usr_inp = "hello world"
        usr_inp = "To be or not to be"
        usr_inp = "tobeornottobe"
        exerone="AVUEVLETSEISBNACBOOLEOBTILBDLCOBOOE"
        key = 5
        keytest = 5
        inp = "ALICELOVESBOBBUTBOBDOESNOTLOVEALICE"
        cipher = RailFence.Encrypt(key, usr_inp)
        cipher2 = RailFence.Encrypt(keytest, inp)
        print(cipher2)
        #print (cipher)
        for i in range(2,len(exerone)):
            orig = RailFence.Decrypt(i, exerone)
            print(orig)
        for j in range(2, len(usr_inp)):
            orig = RailFence.Decrypt(j, cipher)
            print(orig)


    def Encrypt(k, inp):
        print("Encrypting...")
        print("Key is " + str(k))
        colsize = k
        colcount = 0
        rowsize = m.ceil(len(inp) / colsize)
        #print(rowsize)
        rowcount = 0
        store = np.empty([colsize, rowsize], dtype = str)
        #print(store.shape)
        for chars in inp:
            #print(chars)
            if colcount < colsize:
```

```
            store[colcount, rowcount] = chars
            colcount+=1
        else :
            colcount = 0
            rowcount+=1
            store[colcount, rowcount] = chars
            colcount+=1

    ciphertext = ""
    colcount = 0
    rowcount = 0
    for char2 in inp:
        if rowcount < rowsize:
            temp = str(store[colcount, rowcount])
            #print (temp)
            ciphertext += temp
            rowcount+=1
        else :
            rowcount = 0
            colcount+=1
            ciphertext += str(store[colcount, rowcount])
            rowcount+=1
    print(store)
    print("Done")
    return ciphertext


def Decrypt(k, inp):
    print("Decrypting...")
    print("Key is " + str(k))
    origplain = ""
    colsize = m.ceil(len(inp) / k)
    colcount = 0
    rowsize = k
    rowcount = 0
    store = np.empty([rowsize, colsize], dtype = str)
    for chars in inp:
        if rowcount < rowsize:
            store[rowcount, colcount] = chars
            rowcount+=1
        else :
            rowcount = 0
            colcount+=1
            store[rowcount, colcount] = chars
            rowcount+=1
    #print(store)
    colcount = 0
    rowcount = 0
```

```
    for chars2 in inp:
        #print(str(rowcount) + " " + str(colcount))
        if colcount < colsize:
            origplain += store[rowcount, colcount]
            colcount+=1
        else :
            colcount = 0
            rowcount+=1
            #print(rowcount)
            origplain += store[rowcount, colcount]

            colcount+=1
    return origplain
```

Further Explanation:

The code has both an encryption and decryption function so I could check that the process could be reversed. As stated in the algorithm the plaintext is split into columns of size key and seperately written in rows, then the ciphertext is acquired by reading down the rows, Decryption is simply the reversal of this procedure as shown above.

Exercise 2:

Answer:

The output of the feistel encryption was (9,2)

Code Implemented (Python):

```
class feistel:
    def initialise():
        print("initialising")
        Keyprime = 89
        Keyzero = ((Keyprime + 75 * 0)%256)
        Keyone = ((Keyprime + 75 * 1)%256)
        print(Keyone)
        Leftinp = 86
        Rightinp = 83
        feistel.encr(Leftinp, Rightinp, Keyzero, Keyone)
        Leftinp = 9
        Rightinp = 2
        feistel.decr(Leftinp, Rightinp, Keyzero, Keyone)

    def encr (left, right, kz, ko):
        leftout = right
        feistfunczero = ((127 * (kz + right))% 256)
        rightout = left ^ feistfunczero
        left = leftout
        right = rightout
        rightout = right
        feistfuncone= ((127 * (ko + right))% 256)
        leftout = left ^ feistfuncone
        ciphertext = str(leftout) + " " + str(rightout)
        print(ciphertext)

    def decr (left, right, kz, ko):
        leftout = right
        feistfunczero = ((127 * (ko + right))% 256)
        rightout = left ^ feistfunczero
        left = leftout
        right = rightout
        rightout = right
        feistfuncone= ((127 * (kz + right))% 256)
        leftout = left ^ feistfuncone
        plaintext = str(leftout) + " " + str(rightout)
        print(plaintext)
```

Further Explanation:

This task involved simply following the instructions given and then creating a decryption function to test the outputted results were correct.

Exercise 3:

Answer:

The output has changed as the Sbox and Pbox are implimented, every other part of the implimentation does not change the zeros - 00000000000000000000000000000000110110001101100011011011101111100

Code Implemented:

None

Further Explanation:

All values are zero so Initial permutation and the key schedule/rotations will do nothing to change the items. This is the same within the feistel function for expansion and round key addition, all of which will just output a Zeros. The SBox and PBox are where changes occur, as they both have distinctive arrays that are used to replace the inputted bits.

Sbox Output –

(Denary) 14,15,10,7,2,12,4,13
1234,5678,9012,3456,7890,1234,5678,9012

(Binary) 1110,1111,1010,0111,0010,1100,0100,1101

Pbox Output -

(Binary) 1101,1000,1101,1000,1101,1011,1011,1100

This output will then be XORd with the right input but based on how XOR works none of the values will be changed, finally the left output (which is all zeros) will have the right output appended to it, giving the answer shown above.

Exercise 4:

Answer:

nitially the part of the question about bits changing was not clear enough and so both outputs are included in the answer, I believe the comparison of rounds between X and Y to be the correct implimentation as it shows the algorithm working more succinctly but I believed it best to have both to be sure.

In the first round of both implimentations the change in bits is significantly lower than all other rounds, this is mostly due to the number of zeros in both originals, they both rise to above fifty percent and stay in it's close proximity in later rounds, this shows that the algorithm is sound in it's attempts at randomness.

In terms of difference between rounds, the difference seems to increase as the rounds go on, this shows that there is an increase of diffusion as the rounds continue which is to be expected

Code Implemented (Python + Java):

Taken from http://www.pracspedia.com/INS/DES-java.html and changed to increase efficiency of test and get quicker result,

Notable changes include removing user input in place of hardcoding and replacing the hex outputs with binary. A small piece of Pyhton code shown below was also written to facilitate the checking of changed bits. The output of each round is shown after this and so they were removed from the code for increased readability.

```
Xtab = [Xoutzero, Xoutone, Xouttwo, Xoutthr, Xoutfou]

Ytab = [Youtzero, Youtone, Youttwo, Youtthr, Youtfou]

check = 0

for i in range((len(Xtab) - 1)):
    check = 0
    for num, bina in enumerate(Xtab[i], start=0):
        indexer = Xtab[i + 1]
        if bina != indexer[num]:
            check +=1
    print("X:" + str(check) + " Percent:" + str((check/64) * 100))

print("")

for i in range((len(Ytab) - 1)):
    check = 0
    for num, bina in enumerate(Ytab[i], start=0):
        indexer = Ytab[i + 1]
        if bina != indexer[num]:
            check +=1
    print("Y:" + str(check) + " Percent:" + str((check/64) * 100))

print("")
```

```
for j in range(len(Xtab)):
    check = 0
    for num, bina in enumerate(Xtab[i], start=0):
        indexx = Xtab[j]
        indexy = Ytab[j]
        if indexx[num] != indexy[num]:
            check+=1
    print(str(j)+":" + str(check) + " Percent:" + str((check/64) * 100))
```

Comparing Rounds with same input , Output when input is x:
Original:
0000000000000000000000000000000000000000000000000000000000000000
Round 1:
0000000000000000000000000000000110110001101100011011011101111100
Number of bits changed = 19 = 29.7%
Round 2:
1101100011011000110110111011110011100111001110101110101001111
Number of bits changed = 39= 60.9%
Round 3:
1110011100111010111011010100111101011011111110100110101110100110
Number of bits changed = 35 = 54.6%
Round 4:
0101101111111010011010111010011001011110010001011110001001010111
Number of bits changed = 32 = 50.0%
Comparing Rounds with same input, Output when input is y:
Original:
0000000000001000000000000000000000000000000000000000000000000000
Round 1:
0000000000000000000001000000000011011000111110001101101110110100
Number of bits changed = 21= 32.8%
Round 2:
1101100011111000110111011101101001110010100101000111000100110101 1
Number of bits changed = 37 = 57.8%
Round 3:
1110010100101000111000100110101110000111110101110000001000110110
Number of bits changed = 38 = 59.3%
Round 4:
1000011111010111000000100011011000111101000101001111010100111100
Number of bits changed = 37 = 57.8%
Comparing Rounds with different input, X ouput vs Y output:
Original:
Number of bits difference = 1 = 1.5%
Round 1:
Number of bits difference = 3 = 4.7%
Round 2:
Number of bits difference = 11 = 17.2%
Round 3:

Number of bits difference = 24 = 37.5%
Round 4:
Number of bits difference = 31 = 48.4%