DES REPORT

(SARADHI RAMAKRISHNA – 2017H1030081H – M.E Computer Science)

DES Encryption :

The Data Encryption Standard (DES) is a symmetric-key block cipher published by the National Institute of Standards and Technology (NIST).

DES is an implementation of a Feistel Cipher. It uses 16 round Feistel structure. The block size is 64-bit. Though, key length is 64-bit, DES has an effective key length of 56 bits, since 8 of the 64 bits of the key are not used by the encryption algorithm (function as check bits only).

DES Decryption :

Decryption is same as encryption but keys are given in reverse order.

Below is the DES Encryption and Decryption code combined together in single file.

Code :

```
'''
@Author : Saradhi Ramakrishna
@ID     : 2017H1030081H
@Description : Below code is practical implementation of DES Encryption and Decryption
Procedure.
@Inputs : message and key are read from a file.
@Outputs : Writes output to a binary file for Encryption and prints Plain text to the console
for the user for Decryption.
'''



from BitVector import *
from Tables import *



'''
        S-Boxes used in f method
'''
def sBox(MessageBitVector_48_Bits):
    iteration = 0
    message_32_Bits = ""
    for i in range(0,len(MessageBitVector_48_Bits),6):
        rowBinary = str(MessageBitVector_48_Bits[i])+str(MessageBitVector_48_Bits[i+5])
        rowDecimal = int(rowBinary,2)
        colBinary = str(MessageBitVector_48_Bits[i+1:i+5])
        colDecimal = int(colBinary,2)
        message_32_Bits += str("{0:04b}".format(S_BOX[iteration][rowDecimal][colDecimal]))
        iteration += 1
    return BitVector(bitstring = message_32_Bits).permute(S_BOX_PERMUTE_TABLE_ORIGINAL_MODIFIED)
```

```
'''
        Generating 16 rounds of Keys used for encrypting and decrypting
'''
def generateKeysAndStore():
    keyBitVector_56_Bit = keyBitVector_64_Bits.permute(PC_1_ORIGINAL_MODIFIED)
    [leftKeyBitVector_28_Bits, rightKeyBitVector_28_Bits] = keyBitVector_56_Bit.divide_into_two()
    for currentIteration in range(0,16):
        leftKeyBitVector_28_Bits << ITERATION_AND_SHIFTS[currentIteration]
        rightKeyBitVector_28_Bits << ITERATION_AND_SHIFTS[currentIteration]
        newKeyBitVector_56_Bits = leftKeyBitVector_28_Bits + rightKeyBitVector_28_Bits
        Keys[currentIteration] = newKeyBitVector_56_Bits.permute(PC_2_ORIGINAL_MODIFIED)

'''
        Padding function used in encryption procedure if bit vector is having bits less than 64
'''
def returnPadding():
    binString = ""
    for i in range(0,48):
        binString += "0"
    padding = str("0000110100001010") + binString;
    return padding


'''
        function f used for encrypting and decrypting
'''
def f(rightMessageBitVector_32_Bits,keyBitVector_48_Bits):
    rightMessageBitVector_48_Bits = rightMessageBitVector_32_Bits.permute(E_BOX_PERMUTATION_ORIGINAL)
    newrightMessageBitVector_48_Bits = rightMessageBitVector_48_Bits ^ keyBitVector_48_Bits
    result = sBox(newrightMessageBitVector_48_Bits)
    return result



'''
        A single DES Algorithm procedure for encrypting and decrypting with key interchanging
'''
def DES():
    finalCipherText = ""
    while fullMessageBitVectors.more_to_read:
        singleMessageBitVector_64_Bits = fullMessageBitVectors.read_bits_from_file(64)
        if singleMessageBitVector_64_Bits.length() != 64:
            padding = returnPadding()
            Message = str(singleMessageBitVector_64_Bits)+str(padding[0:64-
```

```
singleMessageBitVector_64_Bits.length()])
        singleMessageBitVector_64_Bits = BitVector(bitstring=Message)

            # Applying IP table
                                                permutedSingleMessageBitVector_64_Bits =
singleMessageBitVector_64_Bits.permute(INITIAL_PERMUTATION_64_BITS_MODIFIED)
                    [oldLeftMessageBitVector_32_Bits,oldRightMessageBitVector_32_Bits] =
permutedSingleMessageBitVector_64_Bits.divide_into_two()
      for currentIteration in range(0, 16):
        if user_choice == 1:
          keyBitVector_48_Bits = Keys[currentIteration] # For Encryption
        else:
          keyBitVector_48_Bits = Keys[15-currentIteration] # For Decryption
        newLeftMessageBitVector_32_Bits = oldRightMessageBitVector_32_Bits
                  newRightMessageBitVector_32_Bits = oldLeftMessageBitVector_32_Bits ^
f(oldRightMessageBitVector_32_Bits,keyBitVector_48_Bits)
        oldLeftMessageBitVector_32_Bits = newLeftMessageBitVector_32_Bits
        oldRightMessageBitVector_32_Bits = newRightMessageBitVector_32_Bits

            reversedMessageBitVector_64_Bits = newRightMessageBitVector_32_Bits +
newLeftMessageBitVector_32_Bits  # Reversing
            # Applyting Inverse Intial Permutation Table
                                                finalMessageBitVector_64_Bits =
reversedMessageBitVector_64_Bits.permute(INVERSE_INITIAL_PERMUTATION_64_BITS
_MODIFIED)
      if user_choice == 1:
        finalMessageBitVector_64_Bits.write_to_file(FILEOUT)
        cipherTextInHex = finalMessageBitVector_64_Bits.get_bitvector_in_hex()
        finalCipherText += cipherTextInHex
      if user_choice == 2:
        cipherTextInAscii = finalMessageBitVector_64_Bits.get_bitvector_in_ascii()
        finalCipherText += cipherTextInAscii
  return finalCipherText

while True:
        keyBitVector_64_Bits = BitVector(filename='key.txt').read_bits_from_file(64)
        #keyBitVector_64_Bits = BitVector(hexstring = "0E329232EA6D0D73")
        finalCipherTextInASCII = ""
        finalCipherTextInHEX = ""
        FILEOUT = ""
        Keys = {i: None for i in range(16)}
        fullMessageBitVectors = ""
        generateKeysAndStore()
        print "+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
        print "\t1.Encryption\n\t2.Decryption\n\t3.Quit"
        user_choice = input("Enter Your Choice : ")
        if user_choice == 1:
                FILEOUT = open('output.bits', 'wb')
                fullMessageBitVectors = BitVector(filename='message.txt')
```

```
		print "++++++++++++++++++++ Final Full Cipher Text In HexaDecimal Representation +++++++++++++++++++++++++++++++++++++++++++++"
	elif user_choice == 2:
		fullMessageBitVectors = BitVector(filename='output.bits')
		print "++++++++++++++++++++++++++++++++ Final Full Plain Text ASCII Representation +++++++++++++++++++++++++++++++++++++++++++++"
	else:
		exit()

	print DES()
```

Output :

```
saradhi@saradhi-Lenovo-ideapad-310-15ISK:~/Desktop/krishna$ python DES.py
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++
	1.Encryption
	2.Decryption
	3.Quit
Enter Your Choice : 1
++++++++++++++++++++++ Final Full Cipher Text In HexaDecimal Representation +++++++
++++++++++++++++++++++++++++++++++++++++
8cc93bab92f24c24832aa5a2da09b775c0f7807c17a8cd7d35774ffdaf5aa5f7e9f68e45ee3ca361ae
82fd61c7d10db248cdcf09cad684bfa006006a85a7a258dad8efb9b6eca6e132c88f51c56c031d8a4
66dd1a24661a867001b515226ae1740a7f73fd11f5b22205e5ad862dc0333f70533f7c8a01d81297
fd7d8473a4a9035439667cd56e844428416cac9147992aeb85bea5cccf74c2ad4e5b05e5ea7e1d67
c09819cf103ded958da40e51463eecda2bb70804b47c84fd24cd78072f9c593eebb221619f2f16c96
b5639355828433b5cb9b0b77b0b4799879bbbd670e7fe2313f9370240453817f0e420da7cd26a9b
61354d23b7a46bba51172aa28048e57017fe8d40ef3d0f31ee803ccdc33c59ac4328526cae87a217
1c0becfd79914ed261f018af89d35100137267d3a3b05561eb51c97899488e20156cc7860c2fc603b
5feabb5d0f5a81b19da3fc48cc826840f7a2c7fb140041a97df721fc79d41a0f994aa2bd2de1deeb9a
3afc9f6d6f61b2e1b787c73fa4e625107412bbbdeb08d048144739623a0568df43bb3f7093884503c
6c9e070692c6f28afeaa27ace6d6b8440efd50bef2f2707dc6aa439b0fc017769de7432712da82c30
113cdda01a84c8f471d96c04f0572675b1305da26ec84be33d5080a259a419fda9392cfc5d6d6972
6fff02c41538cc4e3fd6b5e7d2bb65b17e42ab6e6bffa9960fdf836c941cadc25047f7207db18ce3a59
2edfca96f6e37ac38fd03dca6c0498c64b51fa174edc989b92dcc
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++
	1.Encryption
	2.Decryption
	3.Quit
Enter Your Choice : 2
++++++++++++++++++++++++++++++++ Final Full Plain Text ASCII Representation +++++
++++++++++++++++++++++++++++++++++++++++
We prove that the set of DES permutations (encryption and decryption for each DES key) is not
closed under functional composition.
This implies that, in general, multiple DES-encryption is
not equivalent to single DES-encryption, and that DES is not susceptible to a particular
```

```
known-
plaintext attack which requires, on average, 2
∧ 28 steps.
We also show that the size of the subgroup
generated by the set of DES permutations is greater than 10
∧ 2499 , which is too large for potential
attacks on DES which would exploit a small subgroup.

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Confusion and Diffusion :

In cryptography, **confusion** and **diffusion** are two properties of the operation of a secure cipher identified by Claude Shannonin his 1945

Confusion means that each binary digit (bit) of the ciphertext should depend on several parts of the key, obscuring the connections between the two.

Diffusion means that if we change a single bit of the plaintext, then (statistically) half of the bits in the ciphertext should change, and similarly, if we change one bit of the ciphertext, then approximately one half of the plaintext bits should change

Problem-2_Part-1 :

In order to observe the effects of Diffusion, we need to change one bit in plaintext and determine the number of bits changed in the ciphertext. We run this for universal S-Boxes

On an Average, we are getting 31 to 33 bit changes in one block of cipher text for 1 bit change in one block of plain text

Code :

```
'''

@Author : Saradhi Ramakrishna
@ID     : 2017H1030081H
@Description : Below code is practical implementation of Diffusion which is finding out the average number of bits change when one bit in plain
                                text is changed for universal S-Boxes.
@Inputs : message and key are read from a file.
@Outputs : Writes average to console.

'''

from BitVector import *
```

```
from Tables import *

'''
        S-Boxes used in f method
'''
def sBox(MessageBitVector_48_Bits):
   iteration = 0
   message_32_Bits = ""
   for i in range(0,len(MessageBitVector_48_Bits),6):
      rowBinary = str(MessageBitVector_48_Bits[i])+str(MessageBitVector_48_Bits[i+5])
      rowDecimal = int(rowBinary,2)
      colBinary = str(MessageBitVector_48_Bits[i+1:i+5])
      colDecimal = int(colBinary,2)
      message_32_Bits += str("{0:04b}".format(S_BOX[iteration][rowDecimal][colDecimal]))
      iteration += 1
                                                        return                  BitVector(bitstring                 =
message_32_Bits).permute(S_BOX_PERMUTE_TABLE_ORIGINAL_MODIFIED)

'''
        Padding function used in encryption procedure if bit vector is having bits less than 64
'''
def returnPadding():
   binString = ""
   for i in range(0,48):
      binString += "0"
   padding = str("0000110100001010") + binString;
   return padding

'''
        function f used for encrypting and decrypting
'''
def f(rightMessageBitVector_32_Bits,keyBitVector_48_Bits):
                                                             rightMessageBitVector_48_Bits                  =
rightMessageBitVector_32_Bits.permute(E_BOX_PERMUTATION_ORIGINAL)
                 newrightMessageBitVector_48_Bits      =      rightMessageBitVector_48_Bits      ^
keyBitVector_48_Bits
   result = sBox(newrightMessageBitVector_48_Bits)
   return result


'''
        A single DES Algorithm procedure for encrypting
'''
def Encryption(val):
   bitString = ""
   while fullMessageBitVectors.more_to_read:
      keyBitVector_56_Bit = keyBitVector_64_Bits.permute(PC_1_ORIGINAL_MODIFIED)
                                      [leftKeyBitVector_28_Bits,      rightKeyBitVector_28_Bits]      =
keyBitVector_56_Bit.divide_into_two()
      singleMessageBitVector_64_Bits = fullMessageBitVectors.read_bits_from_file(64)
```

```
        if val == 1:
            singleMessageBitVector_64_Bits[bitToChange] = not(singleMessageBitVector_64_Bits[bitToChange])
        if singleMessageBitVector_64_Bits.length() != 64:
            padding = returnPadding()
            Message = str(singleMessageBitVector_64_Bits)+str(padding[0:64-singleMessageBitVector_64_Bits.length()])
            singleMessageBitVector_64_Bits = BitVector(bitstring=Message)
        permutedSingleMessageBitVector_64_Bits = singleMessageBitVector_64_Bits.permute(INITIAL_PERMUTATION_64_BITS_MODIFIED)
        [oldLeftMessageBitVector_32_Bits,oldRightMessageBitVector_32_Bits] = permutedSingleMessageBitVector_64_Bits.divide_into_two()
        for currentIteration in range(0, 16):
            leftKeyBitVector_28_Bits << ITERATION_AND_SHIFTS[currentIteration]
            rightKeyBitVector_28_Bits << ITERATION_AND_SHIFTS[currentIteration]
            newKeyBitVector_56_Bits = leftKeyBitVector_28_Bits + rightKeyBitVector_28_Bits
            keyBitVector_48_Bits = newKeyBitVector_56_Bits.permute(PC_2_ORIGINAL_MODIFIED)
            newLeftMessageBitVector_32_Bits = oldRightMessageBitVector_32_Bits
            newRightMessageBitVector_32_Bits = oldLeftMessageBitVector_32_Bits ^ f(oldRightMessageBitVector_32_Bits,keyBitVector_48_Bits)
            oldLeftMessageBitVector_32_Bits = newLeftMessageBitVector_32_Bits
            oldRightMessageBitVector_32_Bits = newRightMessageBitVector_32_Bits

        reversedMessageBitVector_64_Bits = newRightMessageBitVector_32_Bits + newLeftMessageBitVector_32_Bits  #Reversing
        finalMessageBitVector_64_Bits = reversedMessageBitVector_64_Bits.permute(INVERSE_INITIAL_PERMUTATION_64_BITS_MODIFIED)
        bitString += str(finalMessageBitVector_64_Bits)

    return bitString


bitToChange = 62 # Bit to be changed in one plain text block

fullMessageBitVectors = BitVector(filename='message.txt')
keyBitVector_64_Bits = BitVector(filename='key.txt').read_bits_from_file(64)
oldBitString = Encryption(0)

fullMessageBitVectors = BitVector(filename='message.txt')
keyBitVector_64_Bits = BitVector(filename='key.txt').read_bits_from_file(64)
newBitString = Encryption(1)


noOfCipherBlocksOf64Bits = len(newBitString)/64
Total = 0
for i in range(0,len(oldBitString),64):
    changedBits = 0
    oldSlice = oldBitString[i:i+64]
```

```
    newSlice = newBitString[i:i+64]
    for j in range(0,len(oldSlice)):
        if oldSlice[j]!=newSlice[j]:
            changedBits += 1
    Total += changedBits

Average = Total / noOfCipherBlocksOf64Bits
print "Average No Of Cipher Bits Changed For Plain Text Blocks = "+ str(Average)
```

Output :

```
saradhi@saradhi-Lenovo-ideapad-310-15ISK:/media/saradhi/Academics/Academics/Sem-2/NS/Homework_Assignmets/Assg2/Problem_2$ python Average_1.py
Average No Of Cipher Bits Changed For Plain Text Blocks = 32
```

Problem-2_Part-2 :

In order to observe the effects of Diffusion, we need to change one bit in plaintext and determine the number of bits changed in the ciphertext. We run this for randomly generated S-Boxes

On an Average, we are getting 31 to 33 bit changes in one block of cipher text for 1 bit change in one block of plain text

code :

```
'''

@Author : Saradhi Ramakrishna
@ID     : 2017H1030081H
@Description : Below code is practical implementation of Diffusion which is finding out the average number of bits change when one bit in plain
                                    text is changed for randomly generated S-Boxes (2 times).
@Inputs : message and key are read from a file.
@Outputs : Writes average to console.

'''

from BitVector import *
from Tables import *
import random

S_BOX_RANDOM = [[[0 for k in xrange(16)] for j in xrange(4)] for i in xrange(8)]

'''
        Randomly Generating S-Boxes
'''
def generateS_Boxes():
```

```
    for i in range(0,8):
        for j in range(0,4):
            S_BOX_RANDOM[i][j] = random.sample(range(0,16),16)


'''
        S-Boxes used in f method
'''
def sBox(MessageBitVector_48_Bits):
    iteration = 0
    message_32_Bits = ""
    for i in range(0,len(MessageBitVector_48_Bits),6):
        rowBinary = str(MessageBitVector_48_Bits[i])+str(MessageBitVector_48_Bits[i+5])
        rowDecimal = int(rowBinary,2)
        colBinary = str(MessageBitVector_48_Bits[i+1:i+5])
        colDecimal = int(colBinary,2)
            message_32_Bits  +=  str("{0:04b}".format(S_BOX_RANDOM[iteration][rowDecimal]
[colDecimal]))
        iteration += 1
                                        return                  BitVector(bitstring                 =
message_32_Bits).permute(S_BOX_PERMUTE_TABLE_ORIGINAL_MODIFIED)


'''
        Padding function used in encryption procedure if bit vector is having bits less than 64
'''
def returnPadding():
    binString = ""
    for i in range(0,48):
        binString += "0"
    padding = str("0000110100001010") + binString;
    return padding


'''
        function f used for encrypting and decrypting
'''
def f(rightMessageBitVector_32_Bits,keyBitVector_48_Bits):
                                        rightMessageBitVector_48_Bits                   =
rightMessageBitVector_32_Bits.permute(E_BOX_PERMUTATION_ORIGINAL)
            newrightMessageBitVector_48_Bits    =    rightMessageBitVector_48_Bits    ^
keyBitVector_48_Bits
    result = sBox(newrightMessageBitVector_48_Bits)
    return result


'''
        A single DES Algorithm procedure for encrypting
'''
def Encryption(val):
```

```
    bitString = ""
    while fullMessageBitVectors.more_to_read:
        keyBitVector_56_Bit = keyBitVector_64_Bits.permute(PC_1_ORIGINAL_MODIFIED)
        [leftKeyBitVector_28_Bits, rightKeyBitVector_28_Bits] = keyBitVector_56_Bit.divide_into_two()
        singleMessageBitVector_64_Bits = fullMessageBitVectors.read_bits_from_file(64)
        if val == 1:
            singleMessageBitVector_64_Bits[bitToChange] = not(singleMessageBitVector_64_Bits[bitToChange])
        if singleMessageBitVector_64_Bits.length() != 64:
            padding = returnPadding()
            Message = str(singleMessageBitVector_64_Bits)+str(padding[0:64-singleMessageBitVector_64_Bits.length()])
            singleMessageBitVector_64_Bits = BitVector(bitstring=Message)
        permutedSingleMessageBitVector_64_Bits = singleMessageBitVector_64_Bits.permute(INITIAL_PERMUTATION_64_BITS_MODIFIED)
        [oldLeftMessageBitVector_32_Bits,oldRightMessageBitVector_32_Bits] = permutedSingleMessageBitVector_64_Bits.divide_into_two()
        for currentIteration in range(0, 16):
            leftKeyBitVector_28_Bits << ITERATION_AND_SHIFTS[currentIteration]
            rightKeyBitVector_28_Bits << ITERATION_AND_SHIFTS[currentIteration]
            newKeyBitVector_56_Bits = leftKeyBitVector_28_Bits + rightKeyBitVector_28_Bits
            keyBitVector_48_Bits = newKeyBitVector_56_Bits.permute(PC_2_ORIGINAL_MODIFIED)
            newLeftMessageBitVector_32_Bits = oldRightMessageBitVector_32_Bits
            newRightMessageBitVector_32_Bits = oldLeftMessageBitVector_32_Bits ^ f(oldRightMessageBitVector_32_Bits,keyBitVector_48_Bits)
            oldLeftMessageBitVector_32_Bits = newLeftMessageBitVector_32_Bits
            oldRightMessageBitVector_32_Bits = newRightMessageBitVector_32_Bits

        reversedMessageBitVector_64_Bits = newRightMessageBitVector_32_Bits + newLeftMessageBitVector_32_Bits  #Reversing
        finalMessageBitVector_64_Bits = reversedMessageBitVector_64_Bits.permute(INVERSE_INITIAL_PERMUTATION_64_BITS_MODIFIED)
        bitString += str(finalMessageBitVector_64_Bits)

    return bitString


'''
        Running average for 2 times
'''
for iteration in range(0,2):
    generateS_Boxes()
    bitToChange = 63
    fullMessageBitVectors = BitVector(filename='message.txt')
    keyBitVector_64_Bits = BitVector(filename='key.txt').read_bits_from_file(64)
    oldBitString = Encryption(0)
```

```
    fullMessageBitVectors = BitVector(filename='message.txt')
    keyBitVector_64_Bits = BitVector(filename='key.txt').read_bits_from_file(64)
    newBitString = Encryption(1)

    noOfCipherBlocksOf64Bits = len(newBitString)/64
    Total = 0
    for i in range(0,len(oldBitString),64):
        changedBits = 0
        oldSlice = oldBitString[i:i+64]
        newSlice = newBitString[i:i+64]
        for j in range(0,len(oldSlice)):
            if oldSlice[j]!=newSlice[j]:
                changedBits += 1
        Total += changedBits

    Average = Total / noOfCipherBlocksOf64Bits
      print "Average No Of Cipher Bits Changed For Plain Text Blocks For Random S-Box
Generation - " +str(iteration)+" = "+ str(Average)
```

Output :

```
saradhi@saradhi-Lenovo-ideapad-310-15ISK:/media/saradhi/Academics/Academics/Sem-
2/NS/Homework_Assignmets/Assg2/Problem_2$ python Average_2.py
Average No Of Cipher Bits Changed For Plain Text Blocks For Random S-Box Generation - 0
= 32
Average No Of Cipher Bits Changed For Plain Text Blocks For Random S-Box Generation - 1
= 31
```

Problem-2_Part-3 :

For Confusion, change one bit in encryption key and observe the number of bits changed in the ciphertext.

On an average , we are getting 2100 to 2200 bits change in cipher text when 1 bit is changed in key

code :

```
'''

@Author : Saradhi Ramakrishna
@ID     : 2017H1030081H
@Description : Below code is practical implementation of Confusion which is finding out the
average number of bits change in cipher text when
                                   one bit in key is changed.
@Inputs : message and key are read from a file.
@Outputs : Writes average to console.
```

```
'''

from BitVector import *
from Tables import *

'''
        S-Boxes used in f method
'''
def sBox(MessageBitVector_48_Bits):
   iteration = 0
   message_32_Bits = ""
   for i in range(0,len(MessageBitVector_48_Bits),6):
      rowBinary = str(MessageBitVector_48_Bits[i])+str(MessageBitVector_48_Bits[i+5])
      rowDecimal = int(rowBinary,2)
      colBinary = str(MessageBitVector_48_Bits[i+1:i+5])
      colDecimal = int(colBinary,2)
      message_32_Bits += str("{0:04b}".format(S_BOX[iteration][rowDecimal][colDecimal]))
      iteration += 1
                                        return              BitVector(bitstring              =
message_32_Bits).permute(S_BOX_PERMUTE_TABLE_ORIGINAL_MODIFIED)


'''
        Padding function used in encryption procedure if bit vector is having bits less than 64
'''
def returnPadding():
   binString = ""
   for i in range(0,48):
      binString += "0"
   padding = str("0000110100001010") + binString;
   return padding


'''
        function f used for encrypting and decrypting
'''
def f(rightMessageBitVector_32_Bits,keyBitVector_48_Bits):
                                            rightMessageBitVector_48_Bits                =
rightMessageBitVector_32_Bits.permute(E_BOX_PERMUTATION_ORIGINAL)
            newrightMessageBitVector_48_Bits    =    rightMessageBitVector_48_Bits    ^
keyBitVector_48_Bits
   result = sBox(newrightMessageBitVector_48_Bits)
   return result

'''
        Calculating number of bits changed in new cipher text compared to old cipher text
'''
def calculateChanges(oldBitString,newBitString):
   bitsChanged = 0
```

```
    for i in range(0,len(oldBitString)):
        if oldBitString[i] != newBitString[i]:
            bitsChanged += 1
    return bitsChanged


'''
        A single DES Algorithm procedure for encrypting
'''
def Encryption():
    bitString = ""
    while fullMessageBitVectors.more_to_read:
        keyBitVector_56_Bit = keyBitVector_64_Bits.permute(PC_1_ORIGINAL_MODIFIED)
        [leftKeyBitVector_28_Bits, rightKeyBitVector_28_Bits] = keyBitVector_56_Bit.divide_into_two()
        singleMessageBitVector_64_Bits = fullMessageBitVectors.read_bits_from_file(64)
        if singleMessageBitVector_64_Bits.length() != 64:
            padding = returnPadding()
            Message = str(singleMessageBitVector_64_Bits)+str(padding[0:64-singleMessageBitVector_64_Bits.length()])
            singleMessageBitVector_64_Bits = BitVector(bitstring=Message)
        permutedSingleMessageBitVector_64_Bits = singleMessageBitVector_64_Bits.permute(INITIAL_PERMUTATION_64_BITS_MODIFIED)
        [oldLeftMessageBitVector_32_Bits,oldRightMessageBitVector_32_Bits] = permutedSingleMessageBitVector_64_Bits.divide_into_two()
        for currentIteration in range(0, 16):
            leftKeyBitVector_28_Bits << ITERATION_AND_SHIFTS[currentIteration]
            rightKeyBitVector_28_Bits << ITERATION_AND_SHIFTS[currentIteration]
            newKeyBitVector_56_Bits = leftKeyBitVector_28_Bits + rightKeyBitVector_28_Bits
            keyBitVector_48_Bits = newKeyBitVector_56_Bits.permute(PC_2_ORIGINAL_MODIFIED)
            newLeftMessageBitVector_32_Bits = oldRightMessageBitVector_32_Bits
            newRightMessageBitVector_32_Bits = oldLeftMessageBitVector_32_Bits ^ f(oldRightMessageBitVector_32_Bits,keyBitVector_48_Bits)
            oldLeftMessageBitVector_32_Bits = newLeftMessageBitVector_32_Bits
            oldRightMessageBitVector_32_Bits = newRightMessageBitVector_32_Bits

        reversedMessageBitVector_64_Bits = newRightMessageBitVector_32_Bits + newLeftMessageBitVector_32_Bits  #Reversing
        finalMessageBitVector_64_Bits = reversedMessageBitVector_64_Bits.permute(INVERSE_INITIAL_PERMUTATION_64_BITS_MODIFIED)
        bitString += str(finalMessageBitVector_64_Bits)

    return bitString




fullMessageBitVectors = BitVector(filename='message.txt')
```

```
keyBitVector_64_Bits = BitVector(filename='key.txt').read_bits_from_file(64)
oldBitString = Encryption()
Total = 0

'''
        Running for first 7 bits change in key sequentially
'''
for i in range(0,8):
    fullMessageBitVectors = BitVector(filename='message.txt')
    keyBitVector_64_Bits = BitVector(filename='key.txt').read_bits_from_file(64)
    keyBitVector_64_Bits[i] = not(keyBitVector_64_Bits[i])
    newBitString = Encryption()
    Total += calculateChanges(oldBitString,newBitString)

Average = Total / 7
print "Average No Of Cipher Bits Changed For 7 Choice of Keys = "+ str(Average)
```

Output :

```
saradhi@saradhi-Lenovo-ideapad-310-15ISK:/media/saradhi/Academics/Academics/Sem-2/NS/Homework_Assignmets/Assg2/Problem_2$ python Average_3.py
Average No Of Cipher Bits Changed For 7 Choice of Keys = 2177
```