**AMRITA VISHWA VIDYAPTEETHAM**MCA – S4  
Abhilash Saj George - AM.SC.P2CSC19002

**18CA314-Cryptography and Network Security**

**ASSIGNMENT-1**

**PART A**

2. Find the multiplicative inverse of all the elements in Z5 and Z11

Z5 =>

A = 1 2 3 4  
A-1 = 1 3 2 4

Z11 =>

A = 1 2 3 4 5 6 7 8 9 10  
A-1 = 1 6 4 3 9 2 8 7 5 10

3. Determine the gcd of 56245 and 43159

56245 = 43159 \* 1 + 13086  
43159 = 13086 \* 3 + 3901  
13086 = 3901 \* 3 + 1383  
3901 = 1383 \* 2 + 1135  
1383 = 1135 \* 1 + 248  
1135 = 248 \* 4 + 143  
248 = 143 \* 1 + 105  
143 = 105 \* 1 + 38  
105 = 38 \* 2 + 29  
38 = 29 \* 1 + 9   
29 = 9 \* 3 + 2  
9 = 2 \* 4 + 1  
2 =1 \* 2 + 0

GCD (56245, 43159) = 1

4. Compute phi(n) for 34 and 210

Phi (34 ) =>

= 34 \* (1 - 1/3)  
= 81 \* 2/3  
=162 / 3  
= 54

Phi (210) =>

=210 \* (1 - 1/2)

= 1024 \* 1/2

= 512

5. Compute 3100 mod (31319)

Given, e=100

Therefore, 26+25+22

30 mod 31319 = 3  
32 mod 31319 = 9  
34 mod 31319 = 81  
38 mod 31319 = 6561  
316 mod 31319 = 14418  
332 mod 31319 = 21979  
364 mod 31319 = 12185

3100 mod (31319) = 12185 \* 21979 \* 81 mod 31319  
 = 5346 \* 81 mod 31319  
 = 25879

**Part B - Programming Assignment 1**

1. Write a program to implement Extended Euclidean Algorithm and find multiplicative inverse for following values.

**Source code**

#include<iostream>

using namespace std;

int gcdExtended(int a, int b, int \*x, int \*y)

{

if (a == 0)

{

\*x = 0, \*y = 1;

return b;

}

int x1, y1;

int gcd = gcdExtended(b%a, a, &x1, &y1);

\*x = y1 - (b/a) \* x1;

\*y = x1;

return gcd;

}

int main()

{

int a, m;

cin>>a>>m;

int x, y;

int g = gcdExtended(a, m, &x, &y);

if (g != 1)

cout << "\nNo Inverse. ";

else

{

int res = (x%m + m) % m;

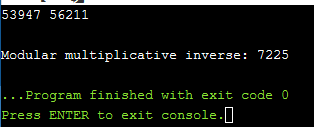
cout << "\nModular multiplicative inverse: " << res;

}

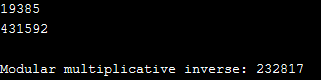
return 0;

}

(a) 53947-1 mod 56211



(b) 19385-1 mod 431592.



2. In cryptography, a brute-force attack consists of an attacker submitting many passwords or passphrases with the hope of eventually guessing correctly. Implement a DES algorithm in any programming language (You are free to use language libraries) and decrypt the following cipher text using brute force attack. Convert the hexadecimal value to string in the final stage.

Cipher text: 0x4B518774A408E3E53.

**Source Code**

**DES implementation**

subKeyList = 16\*[[None]\*8]

IPtable = (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)

EPtable = (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)

PFtable = (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)

FPtable = (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)

sBox = 8\*[64\*[0]]

sBox[0] = (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)

sBox[1] = (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)

sBox[2] = (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)

sBox[3] = ( 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)

sBox[4] = ( 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)

sBox[5] = (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)

sBox[6] = ( 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)

sBox[7] = (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)

def bit2Byte(bitList):

    """Convert bit list into a byte list"""

    return [int("".join(map(str,bitList[i\*8:i\*8+8])),2) for i in range(len(bitList)//8)]

def byte2Bit(byteList):

    """Convert byte list into a bit list"""

    return [(byteList[i//8]>>(7-(i%8)))&0x01 for i in range(8\*len(byteList))]

def permBitList(inputBitList,permTable):

    """Permute input bit list according to input permutation table"""

    return [inputBitList[e - 1] for e in permTable]

def permByteList(inByteList,permTable):

    """Permute input byte list according to input permutation table"""

    outByteList = (len(permTable)>>3)\*[0]

    for index,elem in enumerate(permTable):

        i = index%8

        e = (elem-1)%8

        if i>=e:

            outByteList[index>>3] |= \

                (inByteList[(elem-1)>>3]&(128>>e))>>(i-e)

        else:

            outByteList[index>>3] |= \

                (inByteList[(elem-1)>>3]&(128>>e))<<(e-i)

    return outByteList

def getIndex(inBitList):

    """Permute bits to properly index the S-boxes"""

    return (inBitList[0]<<5)+(inBitList[1]<<3)+ \

           (inBitList[2]<<2)+(inBitList[3]<<1)+ \

           (inBitList[4]<<0)+(inBitList[5]<<4)

def padData(string):

    """Add PKCS5 padding to plaintext"""

    padLength = 8-(len(string)%8)

    return [ord(s) for s in string]+padLength\*[padLength]

def unpadData(byteList):

    """Remove PKCS5 padding from plaintext"""

    return "".join(chr(e) for e in byteList[:-byteList[-1]])

def setKey(keyByteList):

    """Generate all sixteen round subkeys"""

    PC1table = (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)

    PC2table= (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)

    def leftShift(inKeyBitList,round):

        """Perform one (or two) circular left shift(s) on key"""

        LStable = (1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1)

        outKeyBitList = 56\*[0]

        if LStable[round] == 2:

            outKeyBitList[:26] = inKeyBitList[2:28]

            outKeyBitList[26] = inKeyBitList[0]

            outKeyBitList[27] = inKeyBitList[1]

            outKeyBitList[28:54] = inKeyBitList[30:]

            outKeyBitList[54] = inKeyBitList[28]

            outKeyBitList[55] = inKeyBitList[29]

        else:

            outKeyBitList[:27] = inKeyBitList[1:28]

            outKeyBitList[27] = inKeyBitList[0]

            outKeyBitList[28:55] = inKeyBitList[29:]

            outKeyBitList[55] = inKeyBitList[28]

        return outKeyBitList

    permKeyBitList = permBitList(byte2Bit(keyByteList),PC1table)

    for round in range(16):

        auxBitList = leftShift(permKeyBitList,round)

        subKeyList[round] = bit2Byte(permBitList(auxBitList,PC2table))

        permKeyBitList = auxBitList

def encryptBlock(inputBlock):

    """Encrypt an 8-byte block with already defined key"""

    inputData = permByteList(inputBlock,IPtable)

    leftPart,rightPart = inputData[:4],inputData[4:]

    for round in range(16):

        expRightPart = permByteList(rightPart,EPtable)

        key = subKeyList[round]

        indexList = byte2Bit([i^j for i,j in zip(key,expRightPart)])

        sBoxOutput = 4\*[0]

        for nBox in range(4):

            nBox12 = 12\*nBox

            leftIndex = getIndex(indexList[nBox12:nBox12+6])

            rightIndex = getIndex(indexList[nBox12+6:nBox12+12])

            sBoxOutput[nBox] = (sBox[nBox<<1][leftIndex]<<4)+ \

                                sBox[(nBox<<1)+1][rightIndex]

        aux = permByteList(sBoxOutput,PFtable)

        newRightPart = [i^j for i,j in zip(aux,leftPart)]

        leftPart = rightPart

        rightPart = newRightPart

    return permByteList(rightPart+leftPart,FPtable)

def decryptBlock(inputBlock):

    """Decrypt an 8-byte block with already defined key"""

    inputData = permByteList(inputBlock,IPtable)

    leftPart,rightPart = inputData[:4],inputData[4:]

    for round in range(16):

        expRightPart = permByteList(rightPart,EPtable)

        key = subKeyList[15-round]

        indexList = byte2Bit([i^j for i,j in zip(key,expRightPart)])

        sBoxOutput = 4\*[0]

        for nBox in range(4):

            nBox12 = 12\*nBox

            leftIndex = getIndex(indexList[nBox12:nBox12+6])

            rightIndex = getIndex(indexList[nBox12+6:nBox12+12])

            sBoxOutput[nBox] = (sBox[nBox\*2][leftIndex]<<4)+ \

                                sBox[nBox\*2+1][rightIndex]

        aux = permByteList(sBoxOutput,PFtable)

        newRightPart = [i^j for i,j in zip(aux,leftPart)]

        leftPart = rightPart

        rightPart = newRightPart

    return permByteList(rightPart+leftPart,FPtable)

def encrypt(key, inString):

    """Encrypt plaintext with given key"""

    setKey(key)

    inByteList,outByteList = padData(inString),[]

    for i in range(0,len(inByteList),8):

        outByteList += encryptBlock(inByteList[i:i+8])

    return outByteList

def decrypt(key, inByteList):

    """Decrypt ciphertext with given key"""

    setKey(key)

    outByteList = []

    for i in range(0,len(inByteList),8):

        outByteList += decryptBlock(inByteList[i:i+8])

    return unpadData(outByteList)

import sys

from pyDES import setKey, encryptBlock, decryptBlock, encrypt, decrypt

def sanityCheck1():

    x0 =  [0x94, 0x74, 0xb8, 0xe8, 0xc7, 0x3b, 0xca, 0x7d]

    x16 = [0x1b, 0x1a, 0x2d, 0xdb, 0x4c, 0x64, 0x24, 0x38]

    x = x0

    for i in range(16):

        setKey(x)

        if i % 2 == 0:

            x = encryptBlock(x) # if i is even, x[i+1] = E(x[i], x[i)

        else:

            x = decryptBlock(x) # if i is odd, x[i+1] = D(x[i], x[i)

    try:

        assert x == x16

    except AssertionError:

        return False

    return True

def sanityCheck2():

    """Tests multi-block DES encryption and decryption"""

    try:

        key = [0x0f, 0x15, 0x71, 0xc9, 0x47, 0xd9, 0xe8, 0x59]

        plaintext = "The quick brown fox jumps over the lazy dog"

        ciphertext = encrypt(key, plaintext)

        assert decrypt(key, ciphertext) == plaintext

    except AssertionError:

        return False

    return True

def main():

    if sanityCheck1() and sanityCheck2():

        print("All DES tests ok!")

    else:

        sys.exit(1)

    sys.exit()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

Key Breaking

from des import DesKey

import codecs

import string

decode\_hex = codecs.getdecoder("hex\_codec")

i = 1000000

while i < 10:

  sample = str(i)

  key0 = DesKey(bytes(sample.zfill(8), encoding='utf8'))

  x = key0.decrypt(b"4B518774A408E3E5")

  # print(x.hex())

  print(decode\_hex(x.hex()))

  i += 1

# decode\_hex("4B518774A408E3E5")

# all(c in string.hexdigits for c in s)

3. In real world, the commonly used RSA key size if 1024 bits, which is hard for cryptanalysis with limited resources. Implement a RSA algorithm with integer data type and show that you are able to decrypt the cipher text without knowing the private key.

**Source Code**

def p\_and\_q(n):

   data = []

   for i in range(2, n):

      if n % i == 0:

         data.append(i)

   return tuple(data)

def euler(p, q):

   return (p - 1) \* (q - 1)

def private\_index(e, euler\_v):

   for i in range(2, euler\_v):

      if i \* e % euler\_v == 1:

         return i

def decipher(d, n, c):

   return c \*\* d % n

def main():

    e = int(input("input e: "))

    n = int(input("input n: "))

    c = int(input("input c: "))

    p\_and\_q\_v = p\_and\_q(n)

    euler\_v = euler(p\_and\_q\_v[0], p\_and\_q\_v[1])

    d = private\_index(e, euler\_v)

    plain = decipher(d, n, c)

    print("plain: ", plain)

if \_\_name\_\_ == "\_\_main\_\_":

   main()

Output

