Assignment – 2 K.P. SHRIRAM
CSE455 Cryptography E0219007

1. Hill Cipher

Program :

```
import numpy as np


def encrypt(msg):
    # Replace spaces with nothing
    msg = msg.replace(" ", "")
    # Ask for keyword and get encryption matrix
    C = make_key()
    # Append zero if the messsage isn't divisble by 2
    len_check = len(msg) % 2 == 0
    if not len_check:
        msg += "0"
    # Populate message matrix
    P = create_matrix_of_integers_from_string(msg)
    # Calculate length of the message
    msg_len = int(len(msg) / 2)
    # Calculate P * C
    encrypted_msg = ""
    for i in range(msg_len):
        # Dot product
        row_0 = P[0][i] * C[0][0] + P[1][i] * C[0][1]
        # Modulate and add 65 to get back to the A-Z range in ascii
        integer = int(row_0 % 26 + 65)
        # Change back to chr type and add to text
        encrypted_msg += chr(integer)
        # Repeat for the second column
        row_1 = P[0][i] * C[1][0] + P[1][i] * C[1][1]
        integer = int(row_1 % 26 + 65)
        encrypted_msg += chr(integer)
    return encrypted_msg

def decrypt(encrypted_msg):
    # Ask for keyword and get encryption matrix
    C = make_key()
    # Inverse matrix
    determinant = C[0][0] * C[1][1] - C[0][1] * C[1][0]
    determinant = determinant % 26
    multiplicative_inverse = find_multiplicative_inverse(determinant)
    C_inverse = C
```

```
    # Swap a <-> d
    C_inverse[0][0], C_inverse[1][1] = C_inverse[1, 1], C_inverse[0, 0]
    # Replace
    C[0][1] *= -1
    C[1][0] *= -1
    for row in range(2):
        for column in range(2):
            C_inverse[row][column] *= multiplicative_inverse
            C_inverse[row][column] = C_inverse[row][column] % 26

    P = create_matrix_of_integers_from_string(encrypted_msg)
    msg_len = int(len(encrypted_msg) / 2)
    decrypted_msg = ""
    for i in range(msg_len):
        # Dot product
        column_0 = P[0][i] * C_inverse[0][0] + P[1][i] * C_inverse[0][1]
        # Modulate and add 65 to get back to the A-Z range in ascii
        integer = int(column_0 % 26 + 65)
        # Change back to chr type and add to text
        decrypted_msg += chr(integer)
        # Repeat for the second column
        column_1 = P[0][i] * C_inverse[1][0] + P[1][i] * C_inverse[1][1]
        integer = int(column_1 % 26 + 65)
        decrypted_msg += chr(integer)
    if decrypted_msg[-1] == "0":
        decrypted_msg = decrypted_msg[:-1]
    return decrypted_msg

def find_multiplicative_inverse(determinant):
    multiplicative_inverse = -1
    for i in range(26):
        inverse = determinant * i
        if inverse % 26 == 1:
            multiplicative_inverse = i
            break
    return multiplicative_inverse


def make_key():
     # Make sure cipher determinant is relatively prime to 26 and only a/A -
z/Z are given
    determinant = 0
    C = None
    while True:
        cipher = input("Input 4 letter cipher: ")
        C = create_matrix_of_integers_from_string(cipher)
        determinant = C[0][0] * C[1][1] - C[0][1] * C[1][0]
        determinant = determinant % 26
        inverse_element = find_multiplicative_inverse(determinant)
```

```
        if inverse_element == -1:
            print("Determinant is not relatively prime to 26, uninvertible
key")
        elif np.amax(C) > 26 and np.amin(C) < 0:
            print("Only a-z characters are accepted")
            print(np.amax(C), np.amin(C))
        else:
            break
    return C

def create_matrix_of_integers_from_string(string):
    # Map string to a list of integers a/A <-> 0, b/B <-> 1 ... z/Z <-> 25
    integers = [chr_to_int(c) for c in string]
    length = len(integers)
    M = np.zeros((2, int(length / 2)), dtype=np.int32)
    iterator = 0
    for column in range(int(length / 2)):
        for row in range(2):
            M[row][column] = integers[iterator]
            iterator += 1
    return M

def chr_to_int(char):
    # Uppercase the char to get into range 65-90 in ascii table
    char = char.upper()
    # Cast chr to int and subtract 65 to get 0-25
    integer = ord(char) - 65
    return integer

if __name__ == "__main__":
    msg = input("Message: ")
    encrypted_msg = encrypt(msg)
    print(encrypted_msg)
    decrypted_msg = decrypt(encrypted_msg)
    print(decrypted_msg)
```

Output :

```
Message: shriram
Input 4 letter cipher: hill
VNZQPGBN
Input 4 letter cipher: hill
SHRIRAMJ
```

2. SDES Key Generation

Program :

```
import numpy as np

def table_shift(array, table_array):
    array_shifted = np.zeros(table_array.shape[0], dtype='int')
    for index, value in enumerate(table_array): array_shifted[index] =
array[value - 1]
    return array_shifted

def array_split(array):
    left_split = array[:int(len(array) / 2)]
    right_split = array[int(len(array) / 2):]
    return left_split, right_split

def shifting_LtoR(array):
    temp = array[0]
    for index in range(1, len(array)): array[index - 1] = array[index]
    array[len(array) - 1] = temp
    return array

table_p_10 = np.array([3, 5, 2, 7, 4, 10, 1, 9, 8, 6])
table_p_08 = np.array([6, 3, 7, 4, 8, 5, 10, 9])

key = list('0001101101')

def split_and_merge(key):
    left_split, right_split = array_split(key)
    return np.concatenate((shifting_LtoR(left_split),
shifting_LtoR(right_split)))

def key_generation_1(key, table):
    k = table_shift(key, table)
    key_merge = split_and_merge(k)
    return table_shift(key_merge, table)

def key_generation_2(key, table): return split_and_merge(key)

key_1 = key_generation_1(key, table_p_10)
print("".join([str(elem) for elem in key_1]))  #1000111010

key_2 = key_generation_2(key_1, table_p_08)
print("".join([str(elem) for elem in key_2]))  #0001110101
```

Output :

```
1000111010
0001110101
```

3. Elgamel key generation

a. key generation program :

```
import random


def gcd(a: int, b: int):
    """ euclid algorithm for finding the greatest common divisor """
    while a != 0:
        a, b = b % a, a
    return b


def euler(n):
    """ euler function """
    #  if n is a prime number, it is returned directly. n-1
    if (n, 1) == 1:
        return n - 1
    m = 0
    for i in range(n):
        if gcd(i, n) == 1:
            m += 1
    return m


def getFirstPrimitiveRoot(p):
    """ calculate the first primitive root """
    #  the value m of euler function is obtained.
    euler_n = euler(p)
    #  double cycle
    for a in range(2, p):
        for m in range(2, p):
            #  the first m satisfies a^m = 1 mod p  at the same time m =
euler_n then an is the first primitive root.
            if pow(a, m, p) == 1:
                #  if the smallest positive power an is not an euler function
,  proceed to the next cycle
                if m == euler_n:
                    return a
                else:
                    break
    return False
```

```python
def getAllPrimitiveRoot(p, first):
    primitiveRoot = []
    for i in range(p):
        #  if i interprime with p ,  that is, i is  p-1 a member of the simplified residual department of
        if gcd(i, p - 1) == 1:
            #  change the original root   add to the list
            primitiveRoot.append(pow(first, i, p))
    return primitiveRoot


if __name__ == '__main__':
    p = 41
    firstp = getFirstPrimitiveRoot(p)
    pR = getAllPrimitiveRoot(p, firstp)
    print(pR)

    #  randomly select a primitive root g
    g = pR[random.randint(0, len(pR) - 1)]
    #  randomly generate a x
    x = random.randint(1, p - 2)
    #  calculate out y = g^x mod p
    y = pow(g, x, p)

    print(f" open to the public (p,g,y)：{(p, g, y)}")
    print(f" secret preservation x ：{x}")
```

Output :

```
[6, 11, 29, 19, 28, 24, 26, 34, 35, 30, 12, 22, 13, 17, 15, 7]
 open to the public (p,g,y): (41, 24, 16)
 secret preservation x : 8
```

b. Signing :

```python
import random

#  the first step is to disclose the information.
p, g = 521, 186
#  secret kept x
x = 401
#  signed message m
m = 1914168


def gcd(a: int, b: int):
    """ euclid algorithm for finding the greatest common divisor """
    while a != 0:
```

```
        a, b = b % a, a
    return b


#  select k to make  gcd(k,p-1)=1
while True:
    k = random.randint(0, p - 1)
    if gcd(k, p - 1) == 1:
        break

#  calculate r = g^k mod p
r = pow(g, k, p)
#  beg k^-1

#  inversion of extended euclidean algorithm ki that is, the inverse of the 
final need.
ai, bi = k, p - 1
ki, ti, xi, yi = 1, 0, 0, 1  #  initialize s, t,x2,y2
while bi:
    qi, ri = divmod(ai, bi)
    ai, bi = bi, ri  #  find the greatest common divisor
    ki, ti, xi, yi = xi, yi, ki - qi * xi, ti - qi * yi  #  toss and turn and 
divide each other

# s = k^{-1} * (m-xr) mod (p-1)
s = ki * (m - x * r) % (p - 1)

print(f"Alice signed message (m,r,s)：{(m, r, s)}")
```

Output :

```
Alice signed message (m,r,s): (1914168, 343, 355)
```

c. verification program :

```
#  Alice  the public key of
p, g, y = 41, 11, 10
# Alice signed message
m, r, s = 168, 13, 0
#  calculate v1 v2  and compare
v1 = pow(y, r, p) * pow(r, s, p) % p
v2 = pow(g, m, p)

print(f"v1:{v1},v2:{v2}")
if v1 == v2:
    print(" the verification is successful and the signature is valid ")
```

Output :

```
v1:16,v2:16
 the verification is successful and the signature is valid
```