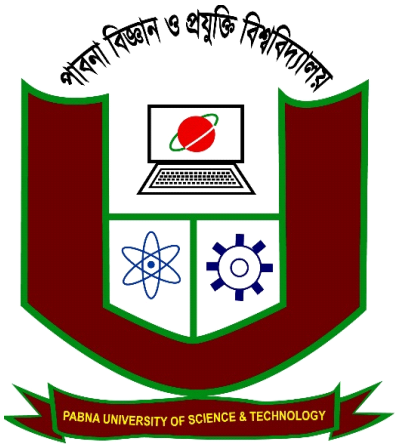
Pabna University of Science and Technology



Faculty of **Engineering and Technology**

Department of **Information and Communication Engineering**

Lab Report

Course Code: ICE-4108

Course Title: Cryptography and Computer Security Sessional

Submitted By:

Name: MD. Mujahid Hossain

ID: 170621

Session: 2016-2017

4th Year 1st Semester

Department of ICE

PUST

Submitted To:

Tarun Debnath

Lecturer

Department of ICE

PUST

Signature: 22/03/2022

INDEX

Problem Name: 01

Problem Name: To implement Caesar cipher algorithm to encrypt and decrypt messages.

Theory:

Caesar cipher is simplest form of substitution cipher. The Caesar cipher involves replacing each letter of the alphabet with the letter standing three places further down the alphabet. For example:

plain: meet me after the toga party

cipher: PHHW PH DIWHU WKH WRJD SDUWB

The transformation can be defined by listing all possibilities as follows:

plain: a b c d e f g h i j k l m n o p q r s t u v w x y z

cipher: D E F G H I J K L M N O P Q R S T U V W X Y Z A B C

Then the algorithm can be expressed as follows. For each plaintext letter, substitute the ciphertext letter:

$C = E(3, p) = (p + 3) \bmod 26$

A shift may be of any amount, so that the general Caesar algorithm is

$C=E(k, p) =(p+k) \bmod 26$

Where k takes on a value in the range 1 to 25. The decryption algorithm is simply

$p = D(k, C) = (C - k) \bmod 26$

Example: Let us consider an example:

Plaintext: cryptography and computer network

a	b	c	d	e	f	g	h	i	j	k	l	m
D	E	F	G	H	I	J	K	L	M	N	O	P
n	o	p	q	r	s	t	u	v	w	x	y	z
Q	R	S	T	U	V	W	X	Y	Z	A	B	C

In this example, the shift k=3

So, the encrypted ciphertext will be: FUBSFWJUDSKB DQG FRPSXWHU QHWZRUN

Ciphertext: FUBSFWJUDSKBDQGFRPSXWHUQHWZRUN

Algorithm for Caesar Cipher:
Input:

- A String of lower-case letters, called Text.
- An Integer between 0-25 denoting the required shift.

Procedure:

- Traverse the given text one character at a time.
- For each character, transform the given character as per the rule, depending on whether we're encrypting or decrypting the text.
- Return the new string generated.

Source Code:

```
def encrypt(string, shift):
    cipher = ' '
    for char in string:
        if char == ' ':
            cipher = cipher + char
        elif char.isupper():
            cipher = cipher + chr((ord(char) + shift - 65) % 26 +65)
        else:
        cipher = cipher + chr((ord(char) + shift - 97) % 26 + 97)
    return cipher
def decrypt(cipher, shift):
    plain = ''
    for char in cipher:
        if char == ' ':
            plain = plain + char
        elif char.isupper():
            plain = plain + chr((ord(char) - shift - 65) % 26 + 65)
        else plain = plain + chr((ord(char) - shift - 97) % 26 + 97)
  return plain
text = input("Enter string: ")
s = int(input("Enter shift number: "))
print("Original string: ", text)
plain=encrypt(text,s)
print("After encryption: ", encrypt(text, s))
print("After decryption :",decrypt(plain,s))
```

Output:

Enter string: SHIHAB
Enter shift number: 4
Original string: SHIHAB
After encryption: WLMLEF
After decryption : SHIHAB

Problem No: 02

Problem Name: To implement Monoalphabetic Cipher

Theory:

Monoalphabetic cipher is a substitution cipher in which for a given key, the cipher alphabet for each plain alphabet is fixed throughout the encryption process. For example, if 'A' is encrypted as 'D', for any number of occurrences in that plaintext, 'A' will always get encrypted to 'D'.

In this technique, any plaintext letter can be substituted with any of the ciphertext letter that means no ciphertext letter can not be repeated. Such an approach is referred to as monoalphabetic substitution cipher because a single cipher alpabet is used per message.

Example: Let us consider an example:

Plaintext: cryptography and computer network

a	b	c	d	e	f	g	h	i	j	k	l	m
D	**J**	**U**	**B**	**N**	**G**	**S**	**K**	**P**	**F**	**W**	**Q**	**L**
n	o	p	q	r	s	t	u	v	w	x	y	z
I	**C**	**O**	**H**	**V**	**T**	**A**	**Y**	**X**	**M**	**Z**	**R**	**E**

So, the encrypted ciphertext will be: UVROASVDOKR DIB UCLOYANV INAMCVW

Ciphertext: UVROASVDOKRDIBUCLOYANVINAMCVW

Monoalphabetic cipher is easy to break because it reflects the frequency data of the original alphabet. A countermeasure is to provide multiple substitutes, known as homophones, for a single letter.

For example, the letter e could be assigned a number of different cipher symbols, such as 16, 74, 35, and 21, with each homophone assigned to a letter in rotation or randomly. If the number of symbols assigned to each letter is proportional to the relative frequency of that letter, then single-letter frequency information is completely obliterated

However, even with homophones, each element of plaintext affects only one element of ciphertext, and multiple-letter patterns (e.g., digram frequencies) still survive in the ciphertext, making cryptanalysis relatively straightforward.

Source Code:

```
import  random
plain_text = []
key =[]
for i in range(65, 65+26):
    plain_text.append(chr(i))
    key.append(chr(i))
message = input("Enter message: ")
random.shuffle(key)
print("Plain Text: ",plain_text)
print("Key:        ",key)
cipher = ''
for ch in message:
    try:
        index = plain_text.index(ch.upper())
        cipher = cipher + key[index]
    except:
        cipher = cipher + ch
print("Cipher: ", cipher)
decrypted_mess = ''
for ch in cipher:
    try:
        index = key.index(ch.upper())
        decrypted_mess = decrypted_mess + plain_text[index]
    except:
        decrypted_mess = decrypted_mess + ch
print("Decrypted Message: ", decrypted_mess)
```

Output:

```
Enter message: my name is shihab

Plain Text:  ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
'Y', 'Z']

Key:         ['D', 'S', 'K', 'O', 'I', 'J', 'F', 'P', 'U', 'M', 'Z',
'N', 'A', 'W', 'Q', 'X', 'R', 'G', 'Y', 'C', 'T', 'B', 'H', 'L',
'E', 'V']

Cipher:  AE WDAI UY YBHBDS

Decrypted Message:  MY NAME IS SHIHAB
```

Problem No: 03

Problem Name: To implement Playfair cipher

Theory:

The Playfair algorithm is based on the use of a 5 × 5 matrix of letters constructed using a keyword. Here is an example, solved by Lord Peter Wimsey in Dorothy Sayers's *Have His Carcase*:

M	**O**	**N**	**A**	**R**
C	**H**	**Y**	B	D
E	F	G	I/J	K
L	P	Q	S	T
U	V	V	X	Z

In this case, the keyword is monarchy. The matrix is constructed by filling in the letters of the keyword (minus duplicates) from left to right and from top to bottom, and then filling in the remainder of the matrix with the remaining letters in alphabetic order. The letters I and J count as one letter. Plaintext is encrypted two letters at a time, according to the following rules:

1. Repeating plaintext letters that are in the same pair are separated with a filler letter, such as *x*, so that balloon would be treated as *ba lx lo on*.
2. Two plaintext letters that fall in the same row of the matrix are each replaced by the letter to the right, with the first element of the row circularly following the last. For example, *ar* is encrypted as *RM*.
3. Two plaintext letters that fall in the same column are each replaced by the letter beneath, with the top element of the column circularly following the last. For example, *mu* is encrypted as *CM*.
4. Otherwise, each plaintext letter in a pair is replaced by the letter that lies in its own row and the column occupied by the other plaintext letter. Thus, *hs* becomes *BP* and *ea* becomes *IM* (or *JM*, as the encipherer wishes).

The Playfair Cipher Algorithm: The Algorithm mainly consist of three steps:

- Convert plaintext into digraphs (i.e., into pair of two letters)
- Generate a Cipher Key Matrix
- Encrypt plaintext using Cipher Key Matrix and get ciphertext.

Source Code:

#source code for playfair cipher

```
key = input("Enter key")
key = key.replace(" ", "")
key = key.upper()
def matrix(x, y, initial):
    return [[initial for i in range(x)] for j in range(y)]
result = list()
for c in key:  # storing key
    if c not in result:
        if c == 'J':
            result.append('I')
        else:
            result.append(c)
flag = 0
for i in range(65, 91):  # storing other character
    if chr(i) not in result:
        if i == 73 and chr(74) not in result:
            result.append("I")
            flag = 1
        elif flag == 0 and i == 73 or i == 74:
            pass
        else:
            result.append(chr(i))
k = 0
my_matrix = matrix(5, 5, 0)  # initialize matrix
for i in range(0, 5):  # making matrix
    for j in range(0, 5):
        my_matrix[i][j] = result[k]
        k += 1
def locindex(c):  # get location of each character
    loc = list()
    if c == 'J':
        c = 'I'
    for i, j in enumerate(my_matrix):
        for k, l in enumerate(j):
            if c == l:
                loc.append(i)
                loc.append(k)
                return loc
def encrypt():  # Encryption
```

```
    msg = str(input("ENTER MSG:"))
    msg = msg.upper()
    msg = msg.replace(" ", "")
    i = 0
    for s in range(0, len(msg) + 1, 2):
        if s < len(msg) - 1:
            if msg[s] == msg[s + 1]:
                msg = msg[:s + 1] + 'X' + msg[s + 1:]
    if len(msg) % 2 != 0:
        msg = msg[:] + 'X'
    print("CIPHER TEXT:", end=' ')
    while i < len(msg):
        loc = list()
        loc = locindex(msg[i])
        loc1 = list()
        loc1 = locindex(msg[i + 1])
        if loc[1] == loc1[1]:
            print("{}{}".format(my_matrix[(loc[0] + 1) %
5][loc[1]], my_matrix[(loc1[0] + 1) % 5][loc1[1]]), end=' ')
        elif loc[0] == loc1[0]:
            print("{}{}".format(my_matrix[loc[0]][(loc[1] + 1)
% 5], my_matrix[loc1[0]][(loc1[1] + 1) % 5]), end=' ')
        else:
            print("{}{}".format(my_matrix[loc[0]][loc1[1]],
my_matrix[loc1[0]][loc[1]]), end=' ')
        i = i + 2
def decrypt():  # decryption
    msg = str(input("ENTER CIPHER TEXT:"))
    msg = msg.upper()
    msg = msg.replace(" ", "")
    print("PLAIN TEXT:", end=' ')
    i = 0
    while i < len(msg):
        loc = list()
        loc = locindex(msg[i])
        loc1 = list()
        loc1 = locindex(msg[i + 1])
        if loc[1] == loc1[1]:
            print("{}{}".format(my_matrix[(loc[0] - 1) %
5][loc[1]], my_matrix[(loc1[0] - 1) % 5][loc1[1]]), end=' ')
        elif loc[0] == loc1[0]:
```

```
            print("{}{}".format(my_matrix[loc[0]][(loc[1] - 1)
% 5], my_matrix[loc1[0]][(loc1[1] - 1) % 5]), end=' ')
        else:
            print("{}{}".format(my_matrix[loc[0]][loc1[1]],
my_matrix[loc1[0]][loc[1]]), end=' ')
        i = i + 2

while (1):
    choice = int(input("\n 1.Encryption \n 2.Decryption: \n
3.EXIT"))
    if choice == 1:
        encrypt()
    elif choice == 2:
        decrypt()
    elif choice == 3:
        exit()
    else:
        print("Choose correct choice")
```

Output:

```
Enter key:SHIHAB
1.Encryption
 2.Decryption:
 3.EXIT
 choice : 1
ENTER MSG:BANGLADESH
CIPHER TEXT: SB OF NH EF HI
 1.Encryption
 2.Decryption:
 3.EXIT
 choice : 2
ENTER CIPHER TEXT: SB OF NH EF HI
PLAIN TEXT: BA NG LA DE SH
```

Problem No: 04

Problem Name: To implement polyalphabetic cipher

Theory:

One way to improve on the simple monoalphabetic technique is to use different monoalphabetic substitutions as one proceeds through the plaintext message. The general name for this approach is polyalphabetic substitution cipher. All these techniques have the following features in common:

1. A set of related monoalphabetic substitution rules is used.
2. A key determines which particular rule is chosen for a given transformation.

Vigenère Cipher: The best known, and one of the simplest, polyalphabetic ciphers is the Vigenère cipher. In this scheme, the set of related monoalphabetic substitution rules consists of the 26 Caesar ciphers with shifts of 0 through 25. Each cipher is denoted by a key letter, which is the ciphertext letter that substitutes for the plaintext letter a. Thus, a Caesar cipher with a shift of 3 is denoted by the key value.

A general equation of the encryption process is

$C_i = (p_i + k_{i \bmod m}) \bmod 26$

decryption is a generalization of Equation

$p_i = (C_i - k_{i \bmod m}) \bmod 26$

Example: To encrypt a message, a key is needed that is as long as the message. Usually, the key is a repeating keyword. For example,

if the keyword is *deceptive*, the message *"we are discovered save yourself"* is encrypted as

key: deceptivedeceptivedeceptive

plaintext: wearediscoveredsaveyourself

ciphertext: ZICVTWQNGRZGVTWAVZHCQYGLMGJ

Expressed numerically, we have the following result

a	b	c	d	e	f	g	h	i	j	k	l	m
0	1	2	3	4	5	6	7	8	9	10	11	12
n	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**	**x**	**y**	**z**
13	14	15	16	17	18	19	20	21	22	23	24	25

key	3	4	2	4	15	19	8	21	4	3	4	2	4	15
Plaintext	22	4	0	17	4	3	8	18	2	14	21	4	17	4
Ciphertext	25	8	2	21	19	22	16	13	6	17	25	6	21	19

Key	19	8	21	4	3	4	2	4	15	19	8	21	4
Plaintext	3	18	0	21	4	14	20	20	17	18	4	11	5
Ciphertext	22	0	21	25	7	2	16	24	6	11	13	6	9

Source code:

```
key_word = input("Enter key Word: ")
#key_word = 'deceptive'

message = input("Enter Message: ")
#message = "wearediscoveredsaveyourself"
print("Message = ", message)
key_word_list = list(key_word)
key = ''
mess_len = len(message)
key_word_len = len(key_word)
for i in range(mess_len):
    key = key + key_word_list[(i%key_word_len)]
print("Key =     ", key)

key_list = list(key)
mess_list = list(message)

cipher_list = []
for i in range(mess_len):
    sum = (ord(key_list[i].upper()) + ord(mess_list[i].upper()) -
(65*2))%26
    cipher_list.append(chr(sum + 65))

cipher = ''
for ch in cipher_list:
    cipher = cipher + ch

print("Cipher Text:", cipher )

dec_mess_list = []
for i in range(mess_len):
    sum = (ord(cipher_list[i].upper()) - ord(key_list[i].upper()) -
(65*2))%26
```

```
    dec_mess_list.append(chr(sum + 65))

dec_mess = ''
for ch in dec_mess_list:
    dec_mess = dec_mess + ch

print("Decrypted Text:", dec_mess )
```

Output:

```
Enter key Word: SHIHAB
Enter Message: BANGLADESH
Message =  BANGLADESH
Key =      THVNLBVLAO
Cipher Text: NUWGSIGQMQ
Decrypted Text: BANGLADESH
```

Problem No: 05

Problem Name: To implement an algorithm to calculate the GCD of two numbers.

Theory: GCD (Greatest Common Divisor) of two numbers is the largest number that divides both of them. Therefore, it is also known as the Highest Common Factor (HCF) of two numbers.

Example: Let's say 2 numbers are 36 and 60. Then

*36 = 2*2*3*3*
*60 = 2*2*3*5*
*GCD=2*2*3 i.e., GCD=12*

A simple solution is to find all prime factors of both numbers, then find intersection of all factors present in both numbers. Finally return product of elements in the intersection.
An efficient solution is to use Euclidean algorithm which is the main algorithm used for this purpose. The idea is, GCD of two numbers doesn't change if smaller number is subtracted from a bigger number.

Euclidean Algorithm for calculating GCD of two numbers A and B can be given as follows:

- If A=0 then GCD (A, B) =B since the Greatest Common Divisor of 0 and B is B.
- If B=0 then GCD (a, b) =a since the Greatest Common Divisor of 0 and a is a.
- Let R be the remainder of dividing A by B assuming A > B. (R = A % B)
- Find GCD (B, R) because GCD (A, B) = GCD (B, R).

Algorithm for finding the GCD of 2 numbers:

Step 1: Start

Step 2: Declare variable n1, n2, gcd=1, i=1

Step 3: Input n1 and n2

Step 4: Repeat until i<=n1 and i<=n2

Step 4.1: If n1%i==0 && n2%i==0:

Step 4.2: gcd = i

Step 5: Print gcd

Step 6: Stop

Source Code:

#Source code for GCD

```
def mygcd(a, b):
    if (b == 0):
        return a
    else:
        return mygcd(b, a % b)

a = int(input("Enter a: "))
b = int(input("Enter b: "))

result = mygcd(a,b)
print('gcd (%s , %s ) = %s ' %(a, b, result))
```

Output:

```
Enter the first Number :8

Enter the second number:10

GCD of given number is : 2
```

Problem No:

Problem Name: To implement RSA algorithm

Theory: RSA algorithm is asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e., Public Key and Private Key. As the name describes that the Public Key is given to everyone and Private key is kept private.

An example of asymmetric cryptography:

- A client (for example browser) sends its public key to the server and requests for some data.
- The server encrypts the data using client's public key and sends the encrypted data.
- Client receives this data and decrypts it.

Since this is asymmetric, nobody else except browser can decrypt the data even if a third party has public key of browser.

The RSA algorithm holds the following features −

- RSA algorithm is a popular exponentiation in a finite field over integers including prime numbers.
- The integers used by this method are sufficiently large making it difficult to solve.
- There are two sets of keys in this algorithm: private key and public key.

The idea of RSA is based on the fact that it is difficult to factorize a large integer. The public key consists of two numbers where one number is multiplication of two large prime numbers. And private key is also derived from the same two prime numbers. So, if somebody can factorize the large number, the private key is compromised. Therefore, encryption strength totally lies on the key size and if we double or triple the key size, the strength of encryption increases exponentially. RSA keys can be typically 1024 or 2048 bits long.

RSA algorithm uses the following procedure to generate public and private keys:

- Select two large prime numbers, p and q.
- Multiply these numbers to find $n = p \times q$, where n is called the modulus for encryption and decryption.
- Choose a number e less than n, such that n is relatively prime to $(p - 1) \times (q - 1)$. It means that e and $(p - 1) \times (q - 1)$ have no common factor except 1. Choose "e" such that $1 < e < \varphi(n)$, e is prime to $\varphi(n)$, $gcd(e, d(n)) = 1$
- If $n = p \times q$, then the public key is $<e, n>$. A plaintext message m is encrypted using public key $<e, n>$. To find ciphertext from the plain text

following formula is used to get ciphertext *C*.
$C = m^e \bmod n$
Here, *m* must be less than *n*. A larger message *(>n)* is treated as a concatenation of messages, each of which is encrypted separately.

- To determine the private key, we use the following formula to calculate the *d* such that:
  $D_e \bmod \{(p - 1) \times (q - 1)\} = 1$
  Or $D_e \bmod \varphi(n) = 1$
- The private key is *<d, n>*. A ciphertext message *c* is decrypted using private key *<d, n>*. To calculate plain text m from the ciphertext *c* following formula is used to get plain text *m*.
  $m = c^d \bmod n$

Source Code:

```
from decimal import Decimal
def gcd(a, b):
    if b == 0  return a
    else:
        return gcd(b, a % b)
p = int(input('Enter the value of p = '))
q = int(input('Enter the value of q = '))
n = p * q
Qn = (p - 1) * (q - 1)
for e in range(2, Qn):
    if gcd(e, Qn) == 1:
        break
print('n = ', n)
print('Qn = ', Qn)
print('e = ', e)
d = 1
while 1:
    if (d * e) % Qn == 1:
        break
    d += 1
print('d = ', d)
print('Public Key = ' + '(' + str(e) + ',' + str(n) + ')')
print('Private Key = ' + '(' + str(d) + ',' + str(n) + ')')
M = int(input('Enter the value of text = '))
C = (pow(M, e) % n)
print('Cipher Text is = ', C)
PT = (pow(C, d) % n)
print('Decrypted Plaintext =', PT)
```

Output:

```
Enter the value of p = 11
Enter the value of q = 13
n =  143
Qn =  120
e =  7
d =  103
Public Key = (7,143)
Private Key = (103,143)
Enter the value of text = 141
Cipher Text is =  15
Decrypted Plaintext = 141
```

Problem No:

Problem Name: To implement Diffie-Hellman key exchange to share keys.

Theory: The Diffie–Hellman (DH) Algorithm is a key-exchange protocol that enables two parties communicating over public channel to establish a mutual secret without it being transmitted over the Internet. DH enables the two to use a public key to encrypt and decrypt their conversation or data using symmetric cryptography.

The simple idea of understanding to the DH Algorithm is the following.

1. The first party picks two prime numbers, g and p and tells them to the second party.
2. The second party then picks a secret number (let's call it a), and then it computes g^a mod p and sends the result back to the first party; let's call the result A. Keep in mind that the secret number is not sent to anyone, only the result is.
3. Then the first party does the same; it selects a secret number b and calculates the result B similar to the
4. step 2. Then, this result is sent to the second party.
5. The second party takes the received number B and calculates B^a mod p
6. The first party takes the received number A and calculates A^b mod p

The answer in step 5 is the same as the answer in step 4. This means both parties will get the same answer no matter the order of exponentiation

$$(g^a \bmod p)^b \bmod p = g^{ab} \bmod p$$
$$(g^b \bmod p)^a \bmod p = g^{ba} \bmod p$$

The number we came within steps 4 and 5 will be taken as the shared secret key. This key can be used to do any encryption of data that will be transmitted

Diffie Hellman Algorithm

1. key $=(Y_A)^{XB}$ mod q -> this is the same as calculated by B

2. Global Public Elements

   q: q is a prime number

   a: a < q and α is the primitive root of q

3. Key generation for user A

   Select a Private key X_A Here, $X_A <$ q

   Now, Calculate Public key Y_A $Y_A = a^{XA}$ mod q

4. Key generation for user B

Select a Private key X_B Here, $X_B < q$

Now, Calculate Public key Y_B $Y_B = a^{Xb} \bmod q$

5. Calculation of Secret Key by A

$key = (Y_B)^{XA} \bmod q$

6. Calculation of Secret Key by B

$key = (Y_A)^{XB} \bmod q$

Source Code:

#source code for Diffie-Hellman key exchange

```
p = int(input("Enter Prime number, p: "))
g = int(input("Enter primitive root of p, g: "))
a = int(input("select Alies Privet Key: "))
b = int(input("select Bob Privet Key: "))

def power(a, b, c):
    if b == 1:
        return int(a)
    else:
        return int((a**b)%c)

x = power(g, a, p)
y = power(g, b, p)

secret_alis = power(y, a, p)
secret_bob = power(x, b, p)
print("Alies Secret key = ", secret_alis)
print("Bob Secret key = ", secret_bob)
```

Output:

```
Enter Prime number, q: 7
Primitive root of q is:
2
select User A Privet Key: 2
select User B Privet Key: 3
USER A Secret key =  1
USER B Secret key =  1
```