Pabna University Of Science and Technology

Faculty of Engineering and Technology.

Department of Information and Communication Engineering.

Lab Report

- **Course Title:** Cryptography and Computer Security Sessional.
- **Course Code:** ICE-4108

Submitted by:	**Submitted To:**
Md. Sariful Islam Roll : 200627 Session: 2019-2020 4th year 1st Semester Dept. of Information and Communication Engineering, PUST.	Md. Anwar Hossain Professor Dept. of Information and Communication Engineering, PUST.

Date of Submission: 13-**11-24**

Signature

INDEX

Experiment No: 01
Experiment Name: Write a program to implement encryption and decryption using Caesar cipher.
Caesar cipher: The Caesar cipher, one of the earliest known encryption methods, is a substitution cipher in which each letter in the plaintext is moved a fixed number of positions along the alphabet. This cipher is named after Julius Caesar, who allegedly used it to protect military communications. The Caesar cipher's simplicity makes it an excellent starting point for learning the fundamentals of encryption and decryption.
In Caesar's cipher:

- A fixed shift key, usually an integer value, specifies how many positions each letter in the message is shifted.
- This key is required for both encryption (moving letters forward) and decryption (shifting them backward).
- For example, using the shift key three times changes the letters "A" to "D," "B" to "E," and so on.

This fundamental principle enables a plaintext message to be converted into ciphertext, hiding its content until deciphered.

Procedure:
Procedure for Encryption Using Caesar Cipher:

1. Select the Shift Key:
Determine the number of positions (n) in which each letter will be relocated.
For example, if n is three, "A" becomes "D", "B" becomes "E", and so on.

2. Convert Plaintext into Ciphertext:
For each letter in plain text:
Locate its place in the alphabet.
Shift it by n, returning to the beginning of the alphabet if necessary.
Non-alphabetic characters are frequently kept unmodified.

Form the encrypted message:
Combine the shifted characters to form the ciphertext.

For example, using the shift key three times would transform the plaintext "HELLO" into “KHOOR”

Procedure for Decryption Using Caesar Cipher:

1. In reverse, use the Shift Key:

Each letter should be shifted left rather than right using the same shift key (n).
Ciphertext to Plaintext Conversion:

In the ciphertext, for every letter:
Determine where it is in the alphabet.
Wrap it around if needed, and move it back by the number n.

Create the Initial Message:
The original plaintext can be seen by combining the shifted characters.

For instance, the ciphertext "KHOOR" would decrypt back to "HELLO" if the shift key was set to 3.

Source Code:

```
def cipher_encrypt(message,shift):
    encrypted_message=""
    for char in message:
        if char.isupper():
            encrypted_message += chr((ord(char)+shift-65)%26+65)
        elif char.islower():
            encrypted_message += chr((ord(char)+shift-97)%26+97)
        else:
            #if it’s not a letter
            encrypted_message += char
    return encrypted_message

def cipher_decrypt(message,shift):
    decrypted_message=""
    for char in message:
        #decrypt uppercase characters
        if char.isupper():
            decrypted_message += chr((ord(char)-shift-65)%26+65)
        elif char.islower():
            decrypted_message += chr((ord(char)-shift-97)%26+97)
        else:
            #if it’s not a letter
            decrypted_message += char
    return decrypted_message
```

```
message=input("Enter the message: ")

shift=int(input("Enter the shift value: " ))

encrypted_message=cipher_encrypt(message,shift)
print("The Encrypted message is: ", encrypted_message)

decrypted_message=cipher_decrypt(encrypted_message,shift)
print("The Decrypted message: ",decrypted_message)
```

Input:

```
Enter the message: Hello
Enter the shift value: 3
```

Output:

```
The Encrypted message is: Khoor
The Decrypted message is: Hello
```

Experiment No: 02
Experiment Name: Write a program to implement encryption and decryption using Mono-Alphabetic cipher.

Mono-Alphabetic cipher: A substitution cipher known as the Mono-Alphabetic Cipher encrypts a message by substituting a fixed, distinct letter from the alphabet for each letter in the plaintext. A Mono-Alphabetic Cipher permits any arbitrary substitution for every letter, in contrast to the Caesar cipher, which shifts letters by a predetermined number. This leads to a significantly greater number of possible keys and a more secure encryption than Caesar.

In this cipher:

1.A fixed substitution key, which is a scrambled alphabet with distinct replacement letters for each letter, is employed.

2. It is regarded as a symmetric cipher since the same key is utilized for both encryption and decryption.

Procedure:
Procedure for Encryption Using Mono-Alphabetic cipher:

1. **Create or Get a Cipher Alphabet Key:** Make a substitution alphabet or use one that already exists, in which every letter in the standard alphabet has a distinct cipher alphabet equivalent.
   Using the reverse alphabet key, for instance, "A" would map to "Z," "B" to "Y," and so on

2. Transform Ciphertext from Plaintext:

In the plaintext, for every letter:
Locate the letter in the cipher alphabet that corresponds to it.
Put the cipher letter in place of the plaintext letter.
Usually, non-alphabetic characters don't change.

3. Create the Encrypted Communication:
The entire ciphertext is created by combining the replaced characters.

Procedure for Decryption Using Mono-Alphabetic cipher:

Using the same cipher alphabet key, follow these steps to decrypt using a mono-alphabetic cipher:

The same replacement alphabet that was used for encryption must be used for the decryption procedure.

1. Ciphertext to Plaintext Conversion:

In the ciphertext, for every letter:
Locate the letter in the cipher alphabet that corresponds to it.
Substitute the original letter from the normal alphabet for the cipher letter.

2. Create the Initial Message:

The plaintext can be seen by combining the replaced characters.

Source Code:

```
import string
import random

def gen_key():
    alpha=list(string.ascii_lowercase)
    shuffled_alpha= alpha[:]
    random.shuffle(shuffled_alpha)
    return dict(zip(alpha,shuffled_alpha))


def encrypt(plaintext,key):
    plaintext=plaintext.lower()
    ciphertext = ""
    for ch in plaintext:
        if ch in key:
            ciphertext += key[ch]
        else:
            ciphertext += ch
    return ciphertext


def decrypt(ciphertext,key):
    reverse_key = {y:x for x,y in key.items()}
    plaintext= ""
    for ch in ciphertext:
        if ch in reverse_key:
            plaintext +=reverse_key[ch]
        else:

            plaintext += ch
    return plaintext



#generate key
key=gen_key()

#encrypting message
plaintext = "Sharif"
ciphertext= encrypt(plaintext,key)
print("Plaintext is : ",plaintext)
print("Ciphertext is: ",ciphertext)

#Decrypting message
decrypted_text=decrypt(ciphertext,key)
print("Decrypted Text is: ",decrypted_text)
```

Input:

```
Plaintext is :  Sharif
```

Output:

```
Ciphertext is:  sqrlki
Decrypted Text is:  sharif
```

Experiment No: 03
Experiment Name: Write a program to implement encryption and decryption using Brute force attack cipher.

Brute force attack cipher: In cryptography, a brute force attack is a technique that breaks encryption by methodically attempting every key or combination until the right one is discovered. The Caesar cipher, which has just 25 potential shifts (ignoring a shift of zero, which leaves the text intact), is one of the simple ciphers with constrained keyspaces that this attack works well against. Without requiring any more knowledge about the key or technique, a brute force attack can discover the original plaintext by testing every potential key.

Because it takes a lot of time and processing resources, the brute force method is typically not feasible for ciphers with big keyspaces. Nonetheless, it can work well for less complex ciphers and illustrates why more sophisticated encryption techniques are required for secure communication.

Procedure:
Brute Force Attack Procedure for Ciphers:

1. **Determine the Type of Cipher:**
   Assess whether the target cipher (such as the Caesar cipher) is straightforward enough for a brute force assault.
   For example, there are only 25 shift values that can be tried when employing a Caesar cipher.

2. **Produce Potential Plaintexts:**
   To create possible plaintexts, apply every key that could be used to the ciphertext.
   This would entail changing the ciphertext by every conceivable value between 1 and 25 for a Caesar cipher and recording the outcome each time.

3. **Examine the findings:**
   Examine each generated plaintext to determine which one makes sense given the language or the information that is known about the original message.
   Algorithms that select results that resemble legible text can automate this procedure.

Source Code:

```
import string
def encrypt(plaintext,shift):
    alpha=string.ascii_lowercase
    encrypted= ""
    for ch in plaintext:
        if ch.isalpha():
            is_upper=ch.isupper()
            ch=ch.lower()
            shifted=(alpha.index(ch)+shift)%26
            encrypted +=alpha[shifted].upper() if is_upper else alpha[shifted]
        else:
            encrypted += ch
    return encrypted

def decrypt(ciphertext,shift):
    alpha=string.ascii_lowercase
    decrypted= ""
    for ch in ciphertext:
        if ch.isalpha():
            is_upper=ch.isupper()
            ch=ch.lower()
            shifted=(alpha.index(ch)-shift)%26
            decrypted +=alpha[shifted].upper() if is_upper else alpha[shifted]
        else:
            decrypted += ch
    return decrypted


#Brute force decryption function

def brute_force_dec(ciphertext):
    alpha=string.ascii_lowercase
    print(f"Attempting to decrypt the ciphertext with all possible shifts....\n")
    for shift in range(26):
        decrypted_text=decrypt(ciphertext,shift)
        print(f"Shift{shift}: {decrypted_text}")


#test the encryption and decryption with bruth force
if __name__ == "__main__":
    plaintext= "Hello, world!"
    shift=3
```

```
ciphertext=encrypt(plaintext,shift)
print(f"Ciphertext: {ciphertext}")

#brute force attack on ciphertext

brute_force_dec(ciphertext)
```

Input:

```
plaintext= "Hello, world!"
shift=3
```

Output:

```
Ciphertext: Khoor, zruog!
Attempting to decrypt the ciphertext with all possible shifts....

Shift0: Khoor, zruog!
Shift1: Jgnnq, yqtnf!
Shift2: Ifmmp, xpsme!
Shift3: Hello, world!
Shift4: Gdkkn, vnqkc!
Shift5: Fcjjm, umpjb!
Shift6: Ebiil, tloia!
Shift7: Dahhk, sknhz!
Shift8: Czggj, rjmgy!
Shift9: Byffi, qilfx!
Shift10: Axeeh, phkew!
Shift11: Zwddg, ogjdv!
Shift12: Yvccf, nficu!
Shift13: Xubbe, mehbt!
Shift14: Wtaad, ldgas!
Shift15: Vszzc, kcfzr!
Shift16: Uryyb, jbeyq!
Shift17: Tqxxa, iadxp!
Shift18: Spwwz, hzcwo!
Shift19: Rovvy, gybvn!
Shift20: Qnuux, fxaum!
Shift21: Pmttw, ewztl!
Shift22: Olssv, dvysk!
Shift23: Nkrru, cuxrj!
Shift24: Mjqqt, btwqi!
Shift25: Lipps, asvph!
```

Experiment No: 04
Experiment Name: Write a program to implement encryption and decryption using Hill cipher.

Hill cipher: The polygraphic substitution cipher known as the Hill cipher was created in 1929 by mathematician Lester Hill. Instead of encrypting individual letters, it encrypts blocks of plaintext characters at a time. This technique encodes and decodes messages using matrix multiplication in linear algebra. Because it processes several letters at once, it creates more intricate patterns that are more difficult to decipher by frequency analysis, making it more secure than straightforward monoalphabetic or polyalphabetic ciphers.

Procedure:
Procedure for Encryption Using Hill cipher:

1. **Convert Plaintext into Numerical Form:**
   - Assign each letter in the plaintext a numerical equivalent from 0 to 25.
   - If necessary, pad the plaintext so that its length matches a multiple of the matrix size.
2. **Choose an Invertible Key Matrix:**
   - Select a matrix (e.g., 2x2 or 3x3) that is invertible in modulo 26 arithmetic.
3. **Divide Plaintext into Blocks:**
   - Break the plaintext into blocks that match the dimensions of the key matrix. For a 2x2 matrix, divide it into pairs of letters.
4. **Encrypt Each Block:**
   - Multiply each block (vector) by the key matrix and take the result modulo 26.
5. **Form the Encrypted Message:**
   - Repeat this process for each block of plaintext to produce the entire ciphertext.

Procedure for Decryption Using the Hill Cipher

1. **Determine the Key Matrix's Inverse:**
   - Determine the inverse of the key matrix modulo 26 in order to decrypt.
   - This inverse matrix will allow the decryption of the ciphertext.
2. **Segment the Ciphertext into Blocks:**
   - Dividing the ciphertext into blocks that correspond to the key matrix's dimensions.
3. **Decrypt Each Block**:

- Multiply each ciphertext block by the inverse of the key matrix and take the result modulo 26.

4. **Convert Numerical Values Back to Letters**:

    - Translate each number back to its corresponding letter to reveal the plaintext.

Source code:

```
import numpy as np
from sympy import Matrix

#Function to calculate modular inverse of matrix of an n*n
def mat_mod_inv(matrix,mod):
    sympy_matrix = Matrix(matrix)
    if sympy_matrix.det() == 0:
        raise ValueError("Matrix is singular, cannot find inverse")
    inv_matrix= sympy_matrix.inv_mod(mod)
    return np.array(inv_matrix).astype(int)


#process converting letter to number
def text_to_mat(text,n):
    text = text.replace(" ","").upper()
    while len(text)%n !=0:
        text +='X'  #padding with x
    matrix = [ord(char)-ord('A') for char in text]
    return np.array(matrix).reshape(-1,n)

#convert matrix back to text
def mat_to_text(matrix):
    text=''.join(chr(int(round(num))+ord('A')) for num in matrix.flatten())
    return text;

#Hill cipher Encryption
def encrypt(plain_text,key_matrix):
    n=key_matrix.shape[0]
    text_matrix=text_to_mat(plain_text,n)
    encrypted_matrix=(text_matrix @ key_matrix) % 26
    cipher_text=mat_to_text(encrypted_matrix)
    return cipher_text

#decryption
def decrypt(cipher_text,key_matrix):
    n=key_matrix.shape[0]
    inv_key_matrix=mat_mod_inv(key_matrix,26)
    text_matrix=text_to_mat(cipher_text,n)
    decrypted_matrix=(text_matrix @ inv_key_matrix) % 26
    plain_text=mat_to_text(decrypted_matrix)
    return plain_text

#example uses
key_matrix = np.array([[5, 8, 17], [1, 19, 8], [9, 4, 14]])
plain_text="HELLOHILL"

#encryption
cipher_text=encrypt(plain_text,key_matrix)
print("Encrypted Message: ",cipher_text)

#Decryption
decrypted_text=decrypt(cipher_text,key_matrix)
print("Decrypted text: ",decrypted_text)
```

Input:
Plain_text="HELLOHILL"

Output:
```
Encrypted Message:  IUTCSHUFO
Decrypted text:  HELLOHILL
```

Experiment No: 05
Experiment Name: Write a program to implement encryption using Playfair cipher.

Playfair cipher: As a digraph substitution cipher, the Playfair cipher encrypts letters in pairs as opposed to one at a time. Charles Wheatstone created it in 1854, but Lord Playfair popularized it. It was utilized for secure communications during both World Wars I and II. By defining letter pairs using a 5x5 matrix and implementing a more intricate substitution method, it offers greater security than straightforward monoalphabetic ciphers.
The cipher used in Playfair:

A 5x5 grid of letters with a single appearance for each letter is created using a keyword. To accommodate all 25 letters, the alphabet usually leaves out the letter "J" .The plaintext is divided into digraphs, or letter pairs. A filler letter (often "X") is used to separate pairs that share the same letter (such as "LL").Depending on where they are in the grid, each pair of letters gets changed.

Procedure:

Procedure for Encryption Using the Playfair Cipher:

Create the matrix of keys: Select a keyword (such as "KEYWORD") and eliminate any extraneous letters. Add the remaining letters of the alphabet (apart from "J") after filling in the 5x5 grid with the letters of the keyword.

Get the plaintext ready: Divide the plaintext into letter pairs. Put a filler letter like "X" in between a pair of letters that share the same letter (like "LL").Add a filler letter to finish the final pair if it only contains one letter.
The plaintext "HELLO" → "HE LX LO" is an example.

Use the Key Matrix to Encrypt Every Pair:
For every pair of letters:

- **Same Row:** Swap out each letter with the one immediately to its right if they are in the same row (looping around to the start if necessary).
- **Same Column:** Swap out each letter with the one immediately below it if both are in the same column (wrapping to the top if necessary).
- **Rectangle:** If the letters make a rectangle, swap out each letter for the one in the other letter's column but on the same row.

Form the Encrypted Message:

- Combine the encrypted pairs to form the final ciphertext.

Source Code:

```
import string
def gen_table(key):
    key=key.lower().replace("j","i")
    table=[]
    seen=set()

       for ch in key:
        if ch not in seen and ch.isalpha():
            seen.add(ch)
            table.append(ch)

       for ch in string.ascii_lowercase:
        if ch not in seen and ch !='j':
            seen.add(ch)
            table.append(ch)

    return [table[i:i+5] for i in range(0,25,5)] #5x5 matrix

def prepare_mes(message):
    message=message.lower().replace("j","i")
    prepared_mes=[]

    i=0
    while i<len(message):
        if i+1 <len(message) and message[i] != message[i+1]:
            prepared_mes.append(message[i]+message[i+1])
            i+=2
        else:
            prepared_mes.append(message[i]+'x') # adding 'x' to duplicate letters
            i+=1
    return prepared_mes

def find_coordinates(letter,table):
    for i in range(5):
        for j in range(5):
            if table[i][j] == letter:
                return i,j
    return None, None


def playfair_encrypt(message,key):
    table=gen_table(key)
    prepared_message = prepare_mes(message)
    encrypted_message = []
    for digraph in prepared_message:
        row1, col1 = find_coordinates(digraph[0],table)
        row2, col2 =find_coordinates(digraph[1],table)

        if row1==row2: #same row
            encrypted_message.append(table[row1][(col1+1)%5])
            encrypted_message.append(table[row2][(col1+2)%5])

        elif col1==col2:
            encrypted_message.append(table[(row1 + 1) % 5][col1])
            encrypted_message.append(table[(row2 + 1) % 5][col2])
        else:
            encrypted_message.append(table[row1][col2])
            encrypted_message.append(table[row2][col1])
    return ''.join(encrypted_message)


if __name__ == "__main__":
    message=input("Enter the message to encrypt : ")
    key = input("Enter the key: ")

    encrypted_message= playfair_encrypt(message,key)
    print(f"Encrypted message: {encrypted_message}")
```

Input:

```
Enter the message to encrypt : Sharif
Enter the key: Keyword
```

Output:

```
Encrypted message: plbclf
```

Experiment No: 06
Experiment Name: Write a program to implement decryption using Playfair cipher.

Playair cipher: As a digraph substitution cipher, the Playfair cipher encrypts letters in pairs as opposed to one at a time. Charles Wheatstone created it in 1854, but Lord Playfair popularized it. It was utilized for secure communications during both World Wars I and II. By defining letter pairs using a 5x5 matrix and implementing a more intricate substitution method, it offers greater security than straightforward monoalphabetic ciphers.
In essence, the Playfair cipher decryption procedure is the opposite of the encryption procedure. Every digraph (letter pair) in the ciphertext can be decoded back to plaintext once the key matrix and encrypted message (ciphertext) are accessible. Similar to encryption, decryption uses the 5x5 key matrix and adheres to particular guidelines determined by the letter positions.

Procedure:
Procedure for Decryption Using the Playfair Cipher:

- **Create the matrix of keys:** The identical 5x5 key matrix that was used for encryption is needed for the decryption procedure.

- **Split the Ciphertext into Two Sets:** The ciphertext should be divided into digraphs, or letter pairs.

- **Decrypt Each Pair Using the Key Matrix**:

- Apply the following decryption rules based on the positions of the letters in the key matrix:
  - **Same Row**: If both letters are in the same row, replace each letter with the letter immediately to its left (wrap around to the right side of the row if needed).
  - **Same Column**: If both letters are in the same column, replace each letter with the letter immediately above it (wrap around to the bottom of the column if needed).
  - **Rectangle**: If the letters form a rectangle, replace each letter with the letter on the same row as the original letter but in the column of the other letter in the pair.

- **Form the Message after Decryption:** Recreate the original plaintext message by combining the decrypted pairings.

- **Eliminate any filler letters or padding:** Remove any filler letters (such as "X") that were added during encryption if they are present in the decrypted text and don't make sense in the context of the message.

Source Code:

```
def gen_key_matrix(key):

    matrix = []
    seen = set()
    for char in key.upper():
        if char not in seen and char != 'J':
            seen.add(char)
            matrix.append(char)
    for char in "ABCDEFGHIKLMNOPQRSTUVWXYZ":
        if char not in seen:
            seen.add(char)
            matrix.append(char)
    return [matrix[i:i + 5] for i in range(0, 25, 5)]

def find_position(matrix, char):
        for row in range(5):
        for col in range(5):
            if matrix[row][col] == char:
                return row, col
    return None

def decrypt_pair(matrix, pair):
    row1, col1 = find_position(matrix, pair[0])
    row2, col2 = find_position(matrix, pair[1])

    if row1 == row2:  # Same row, shift left
        col1 = (col1 - 1) % 5
        col2 = (col2 - 1) % 5
    elif col1 == col2:  # Same column, shift up
        row1 = (row1 - 1) % 5
        row2 = (row2 - 1) % 5
    else:  # Rectangle swap
        col1, col2 = col2, col1

    return matrix[row1][col1] + matrix[row2][col2]

def playfair_decrypt(key, ciphertext):
    matrix = gen_key_matrix(key)
    plaintext = ""

    for i in range(0, len(ciphertext), 2):
        pair = ciphertext[i:i + 2]
        plaintext += decrypt_pair(matrix, pair)
    return plaintext.replace("X", "")

# Example usage
key = "KEYWORD"
ciphertext = "GYISSC"
plaintext = playfair_decrypt(key, ciphertext)
print("Decrypted Text is :", plaintext)
```

Input:
key = "KEYWORD"
ciphertext = "GYISSC"

Output:
```
Decrypted Text is : HELQLO
```

Experiment No: 07
Experiment Name: Write a program to implement encryption using Poly-Alphabetic cipher.

Poly-Alphabetic cipher: Unlike mono-alphabetic ciphers, which only employ one alphabet for substitution, poly-alphabetic ciphers use many substitution alphabets to encrypt the plaintext. This multiple-alphabet method decreases the predictability of character substitution, making it much more difficult to break. Other variants can also be categorized as Poly-Alphabetic ciphers, however the Vigenère Cipher is the most well-known.

The shifting pattern of the alphabet for every letter in the plaintext is determined by a keyword or key phrase in this cipher. To match the plaintext's length, the key is either truncated or repeated.

Procedure:

Procedure for Encryption Using Poly-Alphabetic Cipher (Vigenère Cipher):

Select Keyword:
A word or phrase should serve as the keyword. The alphabet's shifting pattern for encryption will depend on the length of the keyword.

Repeat or truncate the keyword to match the length of the plaintext message: To match the length of the plaintext, extend the keyword and to make the keyword fit the length of the plaintext message, repeat or truncate it.

Transform the Keyword and Plaintext into Numerical Values:
In the plaintext and the keyword, map each letter to a number (A = 0, B = 1, C = 2, ... ,Z = 25).

Encrypt Each Letter Using the Vigenère Formula:
For each letter in the plaintext, shift it forward by the number of positions indicated by the corresponding letter in the keyword. The formula for encryption is: Ci=(Pi+Ki)mod26

- Ci is the ciphertext letter.
- Pi is the plaintext letter.
- Ki is the corresponding letter of the keyword.
- mod 26 ensures the result wraps around the alphabet if it exceeds 25.

Transform the Numerical Outcome into Letters:
The ciphertext can be obtained by converting the generated numbers back to letters after performing the encryption procedure.

Source Code:

```
#polyalphabetic cipher

def gen_key(message,key):
    key=list(key)
    if len(message) == len(key):
        return "".join(key)
    else:
        for i in range(len(message)-len(key)):
            key.append(key[i%len(key)])
    return "".join(key)

def encrypt_vigenere(message,key):
    encrypted_text=[]
    for i in range(len(message)):
        x=(ord(message[i])+ord(key[i])) % 26
        x+=ord('A')
        encrypted_text.append(chr(x))
    return "".join(encrypted_text)



#User Input
message=input("Enter the plain text (A-Z) is: ").upper()
key=input("Enter the key (A-Z) is : ").upper()

#generate key
key=gen_key(message,key)

#encryption
encrypted_text=encrypt_vigenere(message,key)
print(f"Encrypted Text: {encrypted_text}")
```

Input:

```
Enter the plain text (A-Z) is: Sharif
Enter the key (A-Z) is : Key
```

Output:

```
Encrypted Text: CLYBMD
```

Experiment No: 08
Experiment Name: Write a program to implement decryption using Poly-Alphabetic cipher.

Poly-Alphabetic cipher: Poly-Alphabetic ciphers, however the Vigenère Cipher is the most well-known. Unlike mono-alphabetic ciphers, which only employ one alphabet for substitution, poly-alphabetic ciphers use many substitution alphabets to decrypt the plaintext. For the Vigenère cipher, which is a kind of Poly-Alphabetic cipher, the decryption procedure is the opposite of the encryption process. The same keyword that was used for encryption is utilized to convert the encrypted ciphertext back into plaintext. Here, the secret is to reverse the encryption process by subtracting the shift rather than adding it.

Procedure:

Procedure for Decryption Using Poly-Alphabetic Cipher (Vigenère Cipher):

Select Keyword:
A word or phrase should serve as the keyword. The alphabet's shifting pattern for decryption will depend on the length of the keyword.

Repeat or truncate the keyword to match the length of the plaintext message:
To match the length of the plaintext, extend the keyword and to make the keyword fit the length of the plaintext message, repeat or truncate it.

Transform the Keyword and Plaintext into Numerical Values:
In the plaintext and the keyword, map each letter to a number (A = 0, B = 1, C = 2, ... ,Z = 25).

Decrypt Each Letter Using the Vigenère Formula:
For each letter in the plaintext, shift it forward by the number of positions indicated by the corresponding letter in the keyword. The formula for decryption is: $P_i=(C_i-K_i+26) \bmod 26$

- C_i is the ciphertext letter.
- P_i is the plaintext letter.
- K_i is the corresponding letter of the keyword.
- mod 26 ensures the result wraps around the alphabet if it exceeds 25.

Transform the Numerical Outcome into Letters:
The ciphertext can be obtained by converting the generated numbers back to letters after performing the decryption procedure.

Source Code:

```
def gen_key(text,key):
    key=list(key)
    if len(text) == len(key):
        return "".join(key)
    else:
        for i in range(len(text)-len(key)):
            key.append(key[i%len(key)])
    return "".join(key)

def dec_vigenere(encrypted_text,key):
    decrypted_text=[]
    for i in range(len(encrypted_text)):
        x=(ord(encrypted_text[i]) - ord(key[i]) +26 ) %26
        x+=ord('A')
        decrypted_text.append(chr(x))
    return "".join(decrypted_text)

#User Input
ciphertext=input("Enter the cipher text (A-Z) is : ").upper()
key=input("Enter the key (A-Z) is : ").upper()

#generate key
key=gen_key(ciphertext,key)

decrypted_text=dec_vigenere(ciphertext,key)
print(f"Decrypted Text is : {decrypted_text}")
```

Input:

```
Enter the cipher text (A-Z) is : CLYBMD
Enter the key (A-Z) is : Key
```

Output:

```
Decrypted Text is : SHARIF
```

Experiment No: 09
Experiment Name: Write a program to implement encryption using Vernam cipher.

Vernam cipher: Every bit or character in the plaintext is XORed with a matching bit or character from a secret key (the key is as lengthy as the plaintext) in the Vernam cipher, sometimes called the one-time pad. The key needs to be kept secret, used just once, and genuinely random. The Vernam cipher is thought to be unbreakable when the key is random and only used once, as the ciphertext conceals no information about the plaintext.Nevertheless, the Vernam cipher's primary drawback is that the key may only be used once and must be as long as the message. The cipher is compromised if the key is not random or is reused.

Procedure:

Procedure for Encryption using Vernam cipher:
1. Formation of Keys
The encryption key requires to be random and the same length as the plaintext. The key needs to be kept secret and used just once. The key length should be increased if it is less than the plaintext, or a new key that is the same length as the plaintext must be created.

2. Get the key and plaintext ready.
Make that the key and the plaintext have the same length. To facilitate processing, convert the plaintext and the key into binary format or ASCII values (for each character).

3. XOR Operation

- For each character in the plaintext, perform an **XOR** operation with the corresponding character from the key.
    - Convert the characters (from both plaintext and key) to their respective ASCII values.
    - Apply the XOR operation to the ASCII values.
    - Convert the resulting value back to a character.

  The XOR operation is performed using the following rule: Ci=Pi⊕Ki

  Where:

    - Ci is the ciphertext character at position iii.
    - Pi is the plaintext character at position iii.
    - Ki is the key character at position iii.
    - ⊕ is the XOR operation.

4. Produce a Ciphertext
The ciphertext is created by combining the characters that emerge from executing the XOR function on each character. Depending on the XOR results, the ciphertext will probably contain characters that cannot be printed.

5. Encryption Transmission
Together with the key, the ciphertext is securely transmitted to the recipient. The recipient will use the same key to carry out the same XOR operation in order to decrypt the message.

6. Destruction of Keys:
To maintain security, the key must be destroyed after use or kept a secret. You should never use it again.

Source Code:

```
def vernam_encrypt(text, key):
    if len(text) != len(key):
        raise ValueError("Message and key must have the same length.")

    encrypted_message = ''.join(chr(ord(m) ^ ord(k)) for m, k in zip(text, key))
    return encrypted_message

if __name__ == "__main__":
    text = "HelloWorld"
    key = "XMCKLQFTWP"
    encrypted = vernam_encrypt(text, key)
    print(f"Encrypted Message is: {encrypted}")
```

Input:

text = "HelloWorld"
key = "XMCKLQFTWP"

Output:
Encrypted Message is: (/'#)&;4

Experiment No: 10
Experiment Name: Write a program to implement decryption using Vernam cipher.

Vernam cipher: Every bit or character in the plaintext is XORed with a matching bit or character from a secret key (the key is as lengthy as the plaintext) in the Vernam cipher, sometimes called the one-time pad. The key needs to be kept secret, used just once, and genuinely random. The Vernam cipher is thought to be unbreakable when the key is random and only used once, as the ciphertext conceals no information about the plaintext.Nevertheless, the Vernam cipher's primary drawback is that the key may only be used once and must be as long as the message. The cipher is compromised if the key is not random or is reused.

Procedure:

Procedure for Decryption using Vernam cipher:

1. Retrieval of the Key and Ciphertext

- Retrieve the ciphertext and the original encryption key.
- Ensure that the key is the **same length** as the ciphertext and was used only once, as the Vernam cipher relies on a **one-time pad** principle for security.

2. Prepare the Key and Ciphertext

- Confirm that both the key and ciphertext have the same length.
- Convert each character in the ciphertext and the key into **binary format** or **ASCII values** to facilitate processing.

3. XOR Operation

- Perform the **XOR** operation on each character in the ciphertext using the corresponding character from the key:
    - Convert each character in the ciphertext and key to its ASCII value.
    - Apply the XOR operation to each ASCII value pair (from ciphertext and key).
    - Convert the XOR result back into a character, forming part of the original plaintext.

The XOR operation follows this rule:

Pi=Ci⊕Ki

Where:

- Pi is the **plaintext** character at position iii.
- Ci is the **ciphertext** character at position iii.
- Ki is the **key** character at position iii.
- ⊕ is the XOR operation.

4. Produce the Plaintext

- Combine the resulting characters from the XOR operation to reconstruct the original plaintext message.
- Since XORing with the same key used in encryption restores the original characters, the plaintext will now be fully retrieved.

5. Secure Plaintext Retrieval

- After decryption, the plaintext should be securely stored or used as needed.
- The ciphertext and key should be securely handled to prevent unauthorized access.

6. Key Destruction or Secure Storage

- To maintain security, the key should be **destroyed** after use or securely stored if further communication will occur.
- **Never reuse the key** for any other message, as reusing a key in Vernam cipher encryption can compromise security.

Source Code:

```
def vernam_decrypt(ciphertext, key):

    if len(ciphertext) != len(key):
        raise ValueError("Ciphertext and key must have the same length.")

    decrypted_message = ''.join(chr(ord(c) ^ ord(k)) for c, k in zip(ciphertext, key))
    return decrypted_message

# Example usage
if __name__ == "__main__":
    ciphertext = "0(/'#&)&;4"
    key = "XMCKLQFTWP"

    decrypted_message = vernam_decrypt(ciphertext, key)
    print(f"Decrypted Message is: {decrypted_message}")
```

Input:

```
    ciphertext = "0(/'#&)&;4"
    key = "XMCKLQFTWP"
```

Output:

```
Decrypted Message is: helloworld
```