Pabna University of Science and Technology

Faculty of Engineering and Technology

Department of Information and Communication Engineering

Practical Lab Report

Course Title: **Cryptography and Computer Security Sessional**

Course Code: **ICE-4108**

Submitted By:	**Submitted To:**
Syed Shahanur Mahmud Pranto **Roll: 200633** **Session**: 2019-2020 4th Year 1st semester Department of Information and Communication Engineering, PUST.	**Md. Anwar Hossain** **Professor** Department of Information and Communication Engineering, PUST.

Date of Submission: **12/11/2024**

Signature

INDEX

Experiment No: 01

Experiment Name: Write a program to implement encryption and decryption using Caesar cipher.

Objective: To implement and analyze the Caesar cipher for encrypting and decrypting textual data. It aims to explain the cipher's mechanics and significance in cryptography, develop Python functions for accurate encryption and decryption, and evaluate their functionality through testing.

Theory:

The Caesar cipher is the first and most fundamental encryption method. The Caesar cipher technique is based on a mono-alphabetic encryption and is frequently referred to as a shift cipher or additive cipher. Julius Caesar used the shift cipher, also referred to as the additive cipher, to communicate with his officers. Thus, the Caesar cipher is the name given to the shift cipher technique. Every letter in the plain text is changed to a different letter in the Caesar cipher, a kind of substitution cipher.

Caesar ciphers are a weak form of cryptography, making them easy to hack. Messages encrypted using this method can be easily decrypted.

The example below helps illustrate the Caesar cipher: if we shift by 1, then A will be changed to B, B to C, C to D, D to C, and so on until the plain text is finished.

Algorithm:

Caesar Cipher Encryption Algorithm:

1. Input: Plaintext message and a shift value (an integer that represents the number of positions to shift each character).

2. Initialize: An empty string to store the encrypted text.

3. Process Each Character in the Plaintext:

- For each character in the plaintext:
- If the character is an uppercase letter (A-Z):
- Convert the character to its ASCII code using `ord(char)`.
- Apply the shift: new_char = (ord(char) + shift - 65) % 26 + 65.
- Convert the result back to a character using `chr(new_char)`.

- Append this shifted character to the encrypted text.
  - If the character is a lowercase letter (a-z):
  - Convert the character to its ASCII code using `ord(char)`.
  - Apply the shift: new_char = (ord(char) + shift - 97) % 26 + 97.
  - Convert the result back to a character using `chr(new_char)`.
  - Append this shifted character to the encrypted text.
  - If the character is not a letter:
  - Append the character as is to the encrypted text (e.g., punctuation and spaces).

4. Output: The encrypted text, which is the result of shifting each letter by the specified number of positions.

Suppose we have:

- Plaintext: "Hello, Pranto!"
- Shift: 3

Each letter is shifted by 3 positions:

- "H" becomes "K"
- "e" becomes "h"
- and so on.

Final output: "Khoor, Sudqwr!"

Caesar Cipher decryption Algorithm:

To decrypt the message, shift each letter back by the same number of positions used during encryption. For example, shifting each letter in "Khoor, Sudqwr!" back by 3 positions will recover the original message, "Hello, Pranto!".

The decryption formula is: $P=(C-K) \bmod 26$ where C represents the encrypted character,K is the shift value, and P is the decrypted (original) character.

Source code in python:

```
def encrypt(text, shift):
    result = ""
    for char in text:
        if char.isupper():
            result += chr((ord(char) + shift - 65) % 26 + 65)
        elif char.islower():
            result += chr((ord(char) + shift - 97) % 26 + 97)
        else:
            result += char
    return result
def decrypt(text, shift):
    result = ""
    for char in text:
        if char.isupper():
            result += chr((ord(char) - shift - 65) % 26 + 65)
        elif char.islower():
            result += chr((ord(char) - shift - 97) % 26 + 97)
        else:
            result += char
    return result
def main():
    text = input("Enter the plaintext message: ")
    shift = int(input("Enter the shift value: "))
    encrypted_text = encrypt(text, shift)
    print("Encrypted Text:", encrypted_text)
    decrypted_text = decrypt(encrypted_text, shift)
    print("Decrypted Text:", decrypted_text)
if __name__ == "__main__":
    main()
```

Input:

```
Enter the plaintext message: Hello, Pranto!
Enter the shift value: 3
```

Output:

```
Encrypted Text: Khoor, Sudqwr!
Decrypted Text: Hello, Pranto!
```

Experiment No: 02

Experiment Name: Write a program to implement encryption and decryption using Mono-Alphabetic cipher.

Objective:

The objective of this program is to implement a Mono-Alphabetic cipher for encrypting and decrypting textual messages. It aims to demonstrate the consistent substitution of letters using a predefined key, while providing functions that accurately perform both encryption and decryption.

Theory:

The Mono-Alphabetic cipher is a simple substitution cipher where each letter in the plaintext is replaced with a corresponding letter from a fixed substitution alphabet. This method is one of the oldest forms of cryptography.

Mechanism:

In this cipher, each letter in the plaintext consistently maps to one specific letter in the ciphertext. For example, if 'A' is substituted with 'D', every 'A' will be encrypted as 'D'.

Key Characteristics:

- One-to-One Mapping: Each plaintext character corresponds to exactly one ciphertext character.
- Fixed Substitution: The substitution remains constant throughout the encryption process.

While the Mono-Alphabetic cipher demonstrates fundamental principles of substitution encryption, its simplicity and vulnerabilities underscore the need for more advanced cryptographic methods in modern applications. Understanding this cipher provides a foundation for exploring more complex encryption techniques.

Algorithm:

Mono-Alphabetic Cipher Algorithm:

1. **Encryption:**
   - **Generate Cipher Key:** Create a shuffled version of the alphabet to act as the cipher key.
   - **Encrypt:** For each letter in the plaintext, replace it with the corresponding letter from the cipher key. Non-alphabet characters remain unchanged.

2. Decryption:

- **Generate Inverse Key:** Reverse the cipher key (mapping of cipher letters back to original).
- **Decrypt:** For each letter in the ciphertext, replace it with the corresponding letter from the inverse cipher key. Non-alphabet characters remain unchanged.

Example:

Plaintext: "Hello"

Cipher Key: {'a': 'q', 'b': 'd', 'c': 'j', ...}

Encrypted Text: "kbppq"

Decrypted Text: "hello"

Encrypted Text: "kbppq, vlfqna!"

Decrypted Text: "hello, world!"

Source code in Python:

```
import string
import random
def generate_cipher_key():
    alphabet = list(string.ascii_lowercase)
    shuffled_alphabet = alphabet.copy()

    random.shuffle(shuffled_alphabet)
    cipher_key = dict(zip(alphabet, shuffled_alphabet))
    return cipher_key
def encrypt(plaintext, cipher_key):
    ciphertext = ""
    for char in plaintext.lower():
        if char in cipher_key:
            ciphertext += cipher_key[char]
        else:
            ciphertext += char
    return ciphertext
def decrypt(ciphertext, cipher_key):
    reverse_key = {v: k for k, v in cipher_key.items()}
    plaintext = ""
    for char in ciphertext.lower():
        if char in reverse_key:
            plaintext += reverse_key[char]
        else:
            plaintext += char
    return plaintext
def main():
    plaintext = input("Enter the plaintext message: ")
    cipher_key = generate_cipher_key()
    encrypted_text = encrypt(plaintext, cipher_key)
    decrypted_text = decrypt(encrypted_text, cipher_key)
    print("Encrypted Message:", encrypted_text)
    print("Decrypted Message:", decrypted_text)

if __name__ == "__main__":
    main()
```

Input:

```
Enter the plaintext message: Good Morning!
```

Output:

```
Encrypted Message: xggi fgtdsdx!

Decrypted Message: good morning!
```

Experiment No: 03

Experiment Name: Write a program to implement encryption and decryption using Brute force attack cipher.

Objective:

The objective of this program is to implement encryption and decryption using a brute force attack to demonstrate the effectiveness and limitations of this cryptographic technique. It aims to illustrate how a brute force attack systematically tests all possible keys to decrypt an encrypted message, thereby showcasing the trial-and-error nature of the method.

Theory:

A brute force attack is a method used to decrypt encrypted data by systematically trying every possible key until the correct one is found. This approach relies on computational power rather than sophisticated techniques, making it straightforward but often time-consuming.

Mechanism:
In a brute force attack, an attacker needs the encrypted message and knowledge of the encryption system. The attacker generates and tests all possible keys against the encrypted data. For simple ciphers like the Caesar cipher, which has only 26 possible keys, brute force attacks can be executed quickly. However, for more complex systems like AES with 128-bit or 256-bit keys, the number of potential combinations becomes astronomically high, making brute force impractical.

Algorithm:

Algorithm for Encryption and Decryption Using Brute Force Attack on a Cipher:

Encryption Algorithm:

Input: Plaintext message, Shift value (key)
Output: Encrypted ciphertext
Steps:

1. Define the alphabet (e.g., \`a-z\`).
2. Initialize an empty string for the ciphertext.

3. For each character in the plaintext:
    - If it is alphabetic, find its position in the alphabet.

- Calculate the new position by adding the shift value and using modulo to wrap around.
- Replace the character with the one at the new position.
- If it is non-alphabetic, append it unchanged.

4. Return the resulting ciphertext.

Decryption Algorithm Using Brute Force Attack:

Input: Encrypted ciphertext.
Output: Possible plaintext messages for each key.
Steps:

1. For each possible key value from 0 to 25:
   - Initialize an empty string for the decrypted message.
   - For each character in the ciphertext:
   - If it is alphabetic, find its position and calculate the new position by subtracting the key.
   - Replace the character accordingly.
   - If it is non-alphabetic, append it unchanged.
2. Output the decrypted message along with the key used.
3. Repeat for all keys to generate potential plaintext messages.

Source code in python:

```
import string

def caesar_encrypt(plaintext, shift):
    encrypted_text = ""
    for char in plaintext:
        if char.isalpha():
            start = 65 if char.isupper() else 97
            encrypted_text += chr((ord(char) - start + shift) % 26 + start)
        else:
            encrypted_text += char
    return encrypted_text

def caesar_decrypt(ciphertext, shift):
    decrypted_text = ""
    for char in ciphertext:
        if char.isalpha():
            start = 65 if char.isupper() else 97
            decrypted_text += chr((ord(char) - start - shift) % 26 + start)
        else:
            decrypted_text += char
    return decrypted_text

def brute_force_attack(ciphertext):
    print("\nBrute Force Decryption Attempts:")
    for shift in range(26):
        decrypted_text = caesar_decrypt(ciphertext, shift)
        print(f"Shift {shift}: {decrypted_text}")
def main():
    plaintext = input("Enter the plaintext message: ")
    shift = int(input("Enter the shift (key) for encryption: "))
    encrypted_text = caesar_encrypt(plaintext, shift)
    print("\nEncrypted Message:", encrypted_text)
    brute_force_attack(encrypted_text)

if __name__ == "__main__":
    main()
```

Input:

```
Enter the plaintext message: Pranto Mahmud

Enter the shift (key) for encryption: 3
```

Output:

```
Encrypted Message: Sudqwr Pdkpxg

Brute Force Decryption Attempts:

Shift 0: Sudqwr Pdkpxg

Shift 1: Rtcpvq Ocjowf

Shift 2: Qsboup Nbinve

Shift 3: Pranto Mahmud

Shift 4: Oqzmsn Lzgltc

Shift 5: Npylrm Kyfksb

Shift 6: Moxkql Jxejra

Shift 7: Lnwjpk Iwdiqz

Shift 8: Kmvioj Hvchpy

………
```

Experiment No: 04

Experiment Name: Write a program to implement encryption and decryption using Hill cipher.

Objectives:

The objective of this program is to implement the Hill cipher encryption and decryption techniques using a square key matrix. This demonstrates the use of linear algebra in classical cryptography to securely encode and decode messages.

Theory:

The Hill Cipher is a polygraphic substitution cipher that encrypts blocks of text using a square key matrix. It converts the plaintext into numerical form, multiplies it by the key matrix, and takes the result modulo 26 to produce ciphertext. Decryption is achieved by multiplying the ciphertext by the inverse of the key matrix. The cipher's strength comes from using matrix operations, making it more secure than simple substitution ciphers. However, the key matrix must be invertible modulo 26 for decryption to work.

Hill Cipher Algorithm:

Encryption Algorithm:

- **Input**: Plaintext message and key matrix (e.g., 2x2 matrix).
- **Convert**: Convert the plaintext into numerical form (A=0, B=1, ..., Z=25).
- **Divide**: Divide the plaintext into blocks of size equal to the key matrix.
- **Matrix Multiplication**: For each block, multiply the key matrix by the block vector (modulo 26).
- **Output**: Convert the resulting ciphertext vectors back to letters to get the ciphertext.

Decryption Algorithm:

- **Input**: Ciphertext message and key matrix.
- **Inverse Key Matrix**: Compute the inverse of the key matrix modulo 26.
- **Matrix Multiplication**: For each block of the ciphertext, multiply the inverse key matrix by the block vector (modulo 26).
- **Output**: Convert the resulting plaintext vectors back to letters to get the decrypted message.

Summary:

- **Encryption**: C=K×P mod 26
    - C: Ciphertext vector
    - K: Key matrix
    - P: Plaintext vector
- **Decryption**: = K^{-1}×C mod 26
    - K^{-1} Inverse of the key matrix

The Hill cipher uses matrix multiplication to encrypt and decrypt messages by treating text as vectors and applying modular arithmetic.

Source code in python:

```
import numpy as np

def mod_inverse(matrix, mod):
   det = int(np.round(np.linalg.det(matrix)))
   det = det % mod
   if np.gcd(det, mod) != 1:
      raise ValueError("The key matrix is not invertible under modulo 26.")
   det_inv = pow(det, -1, mod)
   matrix_inv = det_inv * np.round(np.linalg.inv(matrix) * det) % mod
   return matrix_inv.astype(int)

def hill_encrypt(plaintext, key_matrix):
   mod = 26
   plaintext = [ord(char) - 97 for char in plaintext.lower() if char.isalpha()]
   while len(plaintext) % len(key_matrix) != 0:
      plaintext.append(0)
   ciphertext = []
   for i in range(0, len(plaintext), len(key_matrix)):
      block = np.array(plaintext[i:i+len(key_matrix)])
      encrypted_block = np.dot(key_matrix, block) % mod
      ciphertext.extend(encrypted_block)

   ciphertext = ''.join(chr(i + 97) for i in ciphertext)
   return ciphertext

def hill_decrypt(ciphertext, key_matrix):
   mod = 26
   inverse_key_matrix = mod_inverse(key_matrix, mod)

   ciphertext = [ord(char) - 97 for char in ciphertext.lower() if char.isalpha()]

   plaintext = []
```

```
    for i in range(0, len(ciphertext), len(key_matrix)):
        block = np.array(ciphertext[i:i+len(key_matrix)])
        decrypted_block = np.dot(inverse_key_matrix, block) % mod
        plaintext.extend(decrypted_block)
    plaintext = ''.join(chr(i + 97) for i in plaintext)
    return plaintext
def take_matrix_input(n):
    matrix = []
    print(f"Enter the {n}x{n} key matrix:")
    for i in range(n):
        row = list(map(int, input(f"Enter row {i+1} (space-separated values): ").split()))
        matrix.append(row)
    return np.array(matrix)

def main():
    n = int(input("Enter the size of the key matrix (e.g., 2 for 2x2): "))
    key_matrix = take_matrix_input(n)
    try:
        plaintext = input("Enter plaintext: ")
        print("\nEncrypted Message: ")
        encrypted_text = hill_encrypt(plaintext, key_matrix)
        print(encrypted_text)

        print("\nDecrypted Message: ")
        decrypted_text = hill_decrypt(encrypted_text, key_matrix)
        print(decrypted_text)
    except ValueError as e:
        print(f"Error: {e}")

if __name__ == "__main__":
    main()
```

Input:

```
Enter the size of the key matrix (e.g., 2 for 2x2): 2

Enter the 2x2 key matrix:

Enter row 1 (space-separated values): 3 3

Enter row 2 (space-separated values): 2 5

Enter plaintext: pranto
```

Output:

```
Encrypted Text:   slnnve
Decrypted Text:   pranto
```

Experiment No: 05

Experiment Name: Write a program to implement encryption using Playfair cipher.

Objectives:

The objective of this program is to implement the Playfair cipher for encrypting textual messages. Specifically, the program aims to demonstrate how to create a 5x5 key square matrix from a given keyword, prepare the plaintext by forming digraphs, and apply the encryption rules of the Playfair cipher to produce ciphertext

Theory:

The Playfair cipher, created by Charles Wheatstone in 1854, is a digraph substitution cipher that encrypts pairs of letters (digraphs) using a 5x5 matrix derived from a keyword. This method enhances security compared to traditional substitution ciphers.

Key Square Generation

To create the key square, a keyword is processed to remove duplicate letters and convert 'J' to 'I'. The grid is filled with the unique letters of the keyword followed by the remaining letters of the alphabet. For example, using "PLAYFAIR" as the keyword results in a matrix that starts with P, L, A, Y, F, I, R, followed by B, C, D, E, G, H, K, M, N, O, Q, S, T, U, V, W, X, Z.

The Playfair cipher represents a significant advancement in cryptography by introducing digraph substitution. While not secure against modern attacks, it provides foundational concepts for understanding more complex encryption methods.

Playfair Cipher Encryption Algorithm:

Step 1: Generate the Key Square
Input: Keyword (e.g., "PLAYFAIR").
Process:

- Remove duplicates and convert to uppercase.
- Replace 'J' with 'I'.
- Create a 5x5 grid starting with unique letters from the keyword, followed by remaining letters of the alphabet (excluding 'J').

Output: A 5x5 key square matrix.

Step 2: Prepare the Plaintext

Input: Plaintext message (e.g., "HELLO WORLD").
Process:

- Convert to uppercase and remove spaces.
- Split into digraphs (pairs of letters).
- Replace identical letters in a pair with 'X' (e.g., "LL" becomes "LX").
- Append 'X' if there's an odd number of characters.

Output: List of digraphs.

Step 3: Apply Encryption Rules
For each digraph:

- Same Row: Replace each letter with the one to its right (wrap around).
- Same Column: Replace each letter with the one below it (wrap around).
- Rectangle Formation: Swap letters based on their rectangle positions.

Step 4: Construct Encrypted Message

- Combine: Join all encrypted digraphs to form the final ciphertext.
- Output: The resulting encrypted message.

Source code in python:

```
def generate_key_matrix(key):
      key = ''.join(sorted(set(key.lower().replace('j', 'i')), key=lambda x:
key.index(x))) + 'abcdefghiklmnopqrstuvwxyz'
   key = ''.join(sorted(set(key), key=key.index))[:25]
   return [key[i:i+5] for i in range(0, len(key), 5)]

def find_position(matrix, char):
   for i, row in enumerate(matrix):
      if char in row:
         return i, row.index(char)

def preprocess_text(text):
   text = text.lower().replace('j', 'i')
   text = ''.join([char for char in text if char.isalpha()])
   if len(text) % 2 != 0: text += 'x'
   return [text[i:i+2] for i in range(0, len(text), 2)]

def encrypt_digraph(digraph, matrix):
   r1, c1 = find_position(matrix, digraph[0])
   r2, c2 = find_position(matrix, digraph[1])
```

```
    if r1 == r2: return matrix[r1][(c1 + 1) % 5] + matrix[r2][(c2 + 1) % 5]
    if c1 == c2: return matrix[(r1 + 1) % 5][c1] + matrix[(r2 + 1) % 5][c2]
    return matrix[r1][c2] + matrix[r2][c1]

def playfair_encrypt(plaintext, key):
    matrix = generate_key_matrix(key)
    digraphs = preprocess_text(plaintext)
    return ''.join([encrypt_digraph(d, matrix) for d in digraphs])
key = input("Enter keyword: ")
plaintext = input("Enter plaintext: ")
ciphertext = playfair_encrypt(plaintext, key)
print("Encrypted Message:", ciphertext)
```

Input:

```
Enter keyword: playfair

Enter plaintext: pranto
```

Output:

```
Ciphertext Message: lipqnq
```

Experiment No: 06

Experiment Name: Write a program to implement decryption using Playfair cipher.

Objectives:

The goal is to understand and implement the decryption process of the Playfair cipher by applying the reverse of the encryption rules to recover the original plaintext from the ciphertext.

Theory:

The Playfair cipher is a digraph substitution cipher that encrypts pairs of letters using a 5x5 matrix derived from a keyword. Decryption reverses the encryption process using the same key and key square.
Decryption in the Playfair cipher effectively recovers plaintext by applying specific rules based on letter positions in a key square. While more secure than simple substitution ciphers, it is still vulnerable to certain cryptanalysis techniques, emphasizing the importance of understanding both its mechanics and limitations in cryptography.

Playfair Cipher Decryption Algorithm:

Steps:

1. Create the 5x5 key square: Same as encryption.

2. Break ciphertext into digraphs: Pair up letters.

3.Decryption rules:

- Different row, different column: Form a rectangle with the letters as opposite corners. The other two corners are the decrypted letters.
- Same row: Move one letter left for each letter (wrap around).
- Same column: Move one letter up for each letter (wrap around).

Example:

Key: MONARCHY

Ciphertext: ARRLQDARXM

Decrypted Text: ATTA CKATDAWN

Source code in python:

```
def generate_key_matrix(key):
    key = ''.join(sorted(set(key.lower().replace('j', 'i')), key=lambda x: key.index(x))) +
'abcdefghiklmnopqrstuvwxyz'
    key = ''.join(sorted(set(key), key=key.index))[:25]
    return [key[i:i+5] for i in range(0, len(key), 5)]

def find_position(matrix, char):
    for i, row in enumerate(matrix):
        if char in row:
            return i, row.index(char)

def preprocess_text(text):
    text = text.lower().replace('j', 'i')
    text = ''.join([char for char in text if char.isalpha()])
    if len(text) % 2 != 0: text += 'x'
    return [text[i:i+2] for i in range(0, len(text), 2)]

def decrypt_digraph(digraph, matrix):
    r1, c1 = find_position(matrix, digraph[0])
    r2, c2 = find_position(matrix, digraph[1])
    if r1 == r2: return matrix[r1][(c1 - 1) % 5] + matrix[r2][(c2 - 1) % 5]
    if c1 == c2: return matrix[(r1 - 1) % 5][c1] + matrix[(r2 - 1) % 5][c2]
    return matrix[r1][c2] + matrix[r2][c1]

def playfair_decrypt(ciphertext, key):
    matrix = generate_key_matrix(key)
    digraphs = preprocess_text(ciphertext)
    return ''.join([decrypt_digraph(d, matrix) for d in digraphs])

key = input("Enter keyword: ")
ciphertext = input("Enter ciphertext: ")
decrypted_text = playfair_decrypt(ciphertext, key)
print("Plaintext Message:", decrypted_text)
```

Input:

```
Enter keyword: playfair

Enter ciphertext: lipqnq
```

Output:

```
Plaintext Message: pranto
```

Experiment No: 07

Experiment Name: Write a program to implement encryption using Poly-Alphabetic cipher.

Objective:

To get it and actualize encryption employing a poly-alphabetic cipher, such as the Vigenère cipher, which employments different letter sets to cloud the recurrence of person letters.

Theory:

An encryption technique called a poly-alphabetic cipher uses many substitution letter sets to jumble a message. Poly-alphabetic ciphers change the substitution periodically based on a slogan or grouping, as contrast to mono-alphabetic ciphers, which use a single fixed move or substitution throughout.
One of the most well-known poly-alphabetic ciphers is the Vigenère Cipher. It determines the shift for every letter in the plaintext using a catchphrase. A move esteem (A=0, B=1, …, Z=25) corresponds to each letter in the watchword, and these move esteems are connected continuously throughout the message.

Poly-Alphabetic Cipher (Vigenère Cipher) Encryption Algorithm:

Steps:-

1. Generate Vigenère Table:

- Create a 26x26 table of shifted alphabets.

2. Preprocess Plaintext:

- Convert plaintext to lowercase and remove non-alphabetic characters.

3. Preprocess Key:

- Convert the key to lowercase and repeat it to match the length of the plaintext.

4. Encryption Process:

- For each character in the plaintext and corresponding key character:
- Find the row for the key character and column for the plaintext character in the Vigenère table.
- The intersection of the row and column gives the encrypted character.

5. Output Ciphertext:

- Combine the encrypted characters to form the ciphertext.

Source code in python:

```
def encrypt_polyalphabetic(plaintext, keyword):
  ciphertext = ""
  keyword_index = 0

  for char in plaintext:
    if char.isalpha():
      shift = ord(keyword[keyword_index % len(keyword)].upper()) - ord('A')
      encrypted_char = chr((ord(char.upper()) + shift - 65) % 26 + 65)
      ciphertext += encrypted_char
      keyword_index += 1
    else:
      ciphertext += char

  return ciphertext
plaintext=input("Enter the plaintext: " )
keyword=input("Enter the keyword:" )
ciphertext = encrypt_polyalphabetic(plaintext, keyword)
print("Ciphertext:",ciphertext)
```

Input:

```
Enter the plaintext: pranto
Enter the keyword: Polyalpha
```

Output:

```
Ciphertext: eflltz
```

Experiment No: 08

Experiment Name: Write a program to implement decryption using Poly-Alphabetic cipher.

Objective:

To implement and understand the decryption process of the Poly-Alphabetic (Vigenère) Cipher, enabling the reversal of encrypted messages back to their original plaintext form using a keyword-based shifting technique. This program will decrypt a ciphertext by applying shifts opposite to those used in encryption, guided by the provided keyword, thus demonstrating how to decode securely transmitted information.

Theory:

The Poly-Alphabetic Cipher, specifically the Vigenère Cipher, is a method of encrypting alphabetic text by using a keyword to dictate multiple shifts in the alphabet for each letter. Unlike mono-alphabetic ciphers, where each letter in the plaintext is shifted by the same fixed amount, the Vigenère Cipher applies a shifting pattern that varies for each character based on the keyword.

For decryption, the reverse shift is applied using the same keyword, which brings the characters back to their original state, revealing the plaintext. Each letter in the keyword represents a different shift value, which repeats to match the length of the ciphertext. This multi-layered approach of shifting makes it more secure than simpler ciphers, as it creates complex patterns that are difficult to break without knowing the keyword.

Algorithm for Decryption using the Poly-Alphabetic Cipher:

Steps:-

1. Initialize:

- Set a variable `plaintext` to an empty string.
- Set a variable `keyword_index` to 0.

2. Iterate over the ciphertext:

- For each character `char` in the ciphertext:
- If `char` is an alphabet:
- Calculate the shift amount using the current character in the keyword.
- Decrypt the character by subtracting the shift amount from its ASCII value, modulo 26, and converting it back to a character.

- Append the decrypted character to the `plaintext` string.
- Increment the `keyword_index`.
- If `char` is not an alphabet, append it directly to the `plaintext` string.

3. Return the plaintext: Return the final `plaintext` string.

Source code in python:

```
def decrypt_polyalphabetic(ciphertext, keyword):
  """Decrypts a ciphertext message using a Vigenère cipher.

  Args:
    ciphertext: The ciphertext message to decrypt.
    keyword: The keyword used for encryption.
  Returns:
    The decrypted plaintext.
  """
  plaintext = ""
  keyword_index = 0
  for char in ciphertext:
    if char.isalpha():
      shift = ord(keyword[keyword_index % len(keyword)].upper()) - ord('A')
      decrypted_char = chr((ord(char.upper()) - shift - 65) % 26 + 65)
      plaintext += decrypted_char
      keyword_index += 1
    else:
      plaintext += char
  return plaintext
ciphertext=input("The ciphertext message to decrypt: " )
keyword=input("The keyword used for encryption:" )
plaintext = decrypt_polyalphabetic(ciphertext, keyword)
print("Plaintext:",plaintext)
```

Input:

```
The ciphertext message to decrypt: eflltz
The keyword used for encryption:polyalpha
```

Output:

```
Plaintext: PRANTO
```

Experiment No: 09

Experiment Name: Write a program to implement encryption using Vernam cipher.

Objective:
To understand and implement the encryption process of the Vernam Cipher, a symmetric-key cipher where the plaintext is XORed with a key of the same length.

Theory:

The Vernam cipher, also known as the one-time pad, is a theoretically unbreakable encryption technique. It involves combining plaintext with a random key of equal length using a bitwise XOR operation.

How it works:
Key Generation: A truly random key, the same length as the plaintext, is generated.
Encryption: Each bit of the plaintext is XORed with the corresponding bit of the key. The result is the ciphertext.
Decryption: The same key is used to XOR the ciphertext, resulting in the original plaintext.

Vernam Cipher Encryption Algorithm:

Steps:-

1. **Input:**
    - plaintext: The message to be encrypted.
    - key: A random key of the same length as the plaintext.
2. **Check Length:**
    - Ensure that the plaintext and key are of equal length.
3. **Iterate and Encrypt:**
    - For each character in the plaintext and key:
        - Convert both characters to their ASCII values.
        - Perform a bitwise XOR operation on the ASCII values.
        - Convert the result back to a character and append it to the ciphertext.
4. **Return Ciphertext:**
    - Return the encrypted ciphertext.

Key Points:

- **One-Time Pad:** The key must be truly random and used only once.
- **Key Length:** The key must be as long as the message to be encrypted.
- **Unbreakable:** If these conditions are met, the Vernam cipher is theoretically unbreakable.

Source code in python:

```
def vernam_encrypt(plaintext, key):
    """Encrypt the plaintext using the Vernam cipher."""
    ciphertext = ''.join(chr((ord(p) ^ ord(k)) % 256) for p, k in zip(plaintext, key))
    return ciphertext

def main():
    plaintext = input("Enter the plaintext: ")

    key = input("Enter the key (must be the same length as plaintext): ")

    if len(key) != len(plaintext):
        print("Error: Key must be of the same length as plaintext.")
        return

    ciphertext = vernam_encrypt(plaintext, key)
    print(f" Ciphertext: {ciphertext}")

if __name__ == "__main__":
    main()
```

Input:

```
Enter the plaintext: hello

Enter the key (must be the same length as plaintext): ABCDE
```

Output:

```
Ciphertext: )'/(*
```

Experiment No: 10

Experiment Name: Write a program to implement decryption using Vernam cipher.

Objective:

The primary objective of this program is to demonstrate the implementation of the Vernam cipher, a theoretically unbreakable encryption technique. The goal is to reverse the encryption process using the XOR operation between the ciphertext and the key to recover the original plaintext message.

Theory:

The Vernam Cipher, also known as the One-Time Pad, is a symmetric key cipher that encrypts and decrypts data using an XOR (exclusive OR) operation. It was invented by Gilbert Vernam in 1917 and is considered one of the most secure encryption methods when used correctly. The key to its security lies in the use of a truly random key that is as long as the plaintext message and is never reused.

Key Features:

1. **Symmetric Key Encryption**:
    - The same key is used for both encryption and decryption. This means that both the sender and receiver must have access to the same secret key.
2. **XOR Operation**:
    - The encryption and decryption process relies on the XOR operation, which is performed bit by bit between the plaintext and the key. XOR outputs 1 if the inputs are different and 0 if they are the same.
3. **Key Characteristics**:
    - The key used in the Vernam cipher must be random and as long as the plaintext message.
    - Each key is used only once, hence the name One-Time Pad.

Vernam Cipher Decryption Algorithm:

1.Input:

- ciphertext: The encrypted message.
- key: The key used for encryption.

2. Check Length:

- Ensure that the ciphertext and key are of the same length.

3. Iterate and Decrypt:

- For each character in the ciphertext and key:
  - Convert both characters to their ASCII values.
  - Perform a bitwise XOR operation on the ASCII values.
  - Convert the result back to a character and append it to the plaintext.

4. Return Plaintext:

- Return the decrypted plaintext.

Source code in :

```
def vernam_decrypt(ciphertext, key):
    plaintext = ''.join(chr((ord(c) ^ ord(k)) % 256) for c, k in zip(ciphertext, key))
    return plaintext

def main():
    ciphertext = input("Enter the ciphertext: ")
    key = input("Enter the key (must be the same length as ciphertext): ")

    if len(key) != len(ciphertext):
        print("Error: Key must be of the same length as ciphertext.")
        return

    decrypted_text = vernam_decrypt(ciphertext, key)
    print(f"Plaintext message: {decrypted_text}")

if __name__ == "__main__":
    main()
```

Input:

```
Enter the ciphertext: )'/(*
Enter the key (must be the same length as ciphertext): ABCDE
```

Output:

```
Plaintext message: hello
```