PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF INFORMATION AND COMMUNICATION ENGINEERING

LAB REPORT

COURSE CODE: ICE-4108

COURSE TITTLE: CRYPTOGRAPHY AND COMPUTER SECURITY SESSIONAL

SUBMITTED BY:

Sajeeb Kumar Ray
Roll: 200622
Session: 2019-2020
4th Year 1st Semester
Department of Information and Communication Engineering, Pabna University of Science and Technology.

SUBMITTED TO:

Md. Anwar Hossain
Professor,
Department of Information and Communication Engineering, Pabna University of Science and Technology.

DATE OF SUBMISSION: 12/11/2024

SIGNATURE

INDEX

Experiment Number: 01
Experiment Name: Write a program to implement encryption and decryption using Caesar cipher.

Objectives:

- To move each character by a certain number of places in order to encrypt text.
- To reverse the shift used during encryption in order to decode text.
- To use the Caesar Cipher approach to comprehend fundamental encryption and decryption concepts.

Theory: One of the first known encryption methods is the Caesar Cipher, which is a kind of substitution cipher. This cipher, which bears Julius Caesar's name because he employed it in his military communications, moves each letter in a message by a predetermined number of points. With a shift of 3, for instance, "A" becomes "D," "B" becomes "E," and so on. By shifting the characters back, the recipient can decode the message if they know the shift amount (key). Despite its simplicity, the Caesar Cipher introduces more sophisticated encryption methods and provides a basic knowledge of cryptography topics like keys and encryption.

For encryption,

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

For decryption,

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

Where,
C = Cipher text
K = Key
P = Plain text

Code:

```
import string

alphabet = string.ascii_uppercase
mp = dict(zip(alphabet, range(26)))
mp2 = dict(zip(range(26), alphabet))

def caesar_encrypt(message, shift):
    encrypted_message = ""
    x = " "
    for char in message:
        if char == " ":
            encrypted_message += x
            continue
        encrypted_message += mp2[(mp[char] + shift) % 26]
    return encrypted_message

def caesar_decrypt(caesar_message, shift):
    decrypted_message = ""
    x = " "
    for char in caesar_message:
        if char == " ":
            decrypted_message += x
            continue

        decrypted_message += mp2[(mp[char] - shift + 26) % 26]
    return decrypted_message

message = input("Enter Message to encrypt: ").upper()
shift = int(input("Enter Shift To Encrypt: "))
encrypted_message = caesar_encrypt(message, shift)
print("Encrypted Message:", encrypted_message)
print("Decrypted Message:", caesar_decrypt(encrypted_message, shift))
```

Input:

```
Enter Message to Encrypt: Information
Enter Shift to Encrypt: 3
```

Output:

```
Encrypted Message: LQIRUPDWLRQ
Decrypted Message: INFORMATION
```

Experiment Number: 02
Experiment Name: Write a program to implement encryption and decryption using Mono-Alphabetic cipher.

Objectives:

- To implement a monoalphabetic substitution cipher.
- To understand how monoalphabetic substitution ciphers encrypt and decrypt messages using a fixed key mapping.
- To demonstrate both encryption and decryption processes to verify the reversibility of the cipher with a specific key.

Theory:
A monoalphabetic substitution cipher represents a traditional cryptographic methodology wherein each individual letter in the plaintext is exchanged for a corresponding letter from a predetermined and static cipher alphabet. In contrast to polyalphabetic ciphers, which exhibit varying mappings throughout the encryption process, a monoalphabetic cipher adheres to a singular substitution principle applicable to all letters within the message. This fixed correspondence can be established by permuting the alphabet or employing a specified permutation as a key. For instance, in the code presented, a bespoke cipher alphabet (encoded_character) is utilized, wherein each letter from the original alphabet (normal_character) is assigned to a distinct letter. During the encryption phase, each character extracted from the input message is substituted by its corresponding cipher character in accordance with this mapping. The identical key is subsequently employed to invert the substitution during the decryption process, thereby restoring the original text. Although relatively straightforward, monoalphabetic ciphers are susceptible to frequency analysis, as each character in the plaintext possesses a direct equivalent in the ciphertext. Notwithstanding this drawback, the monoalphabetic cipher affords a fundamental comprehension of substitution techniques and their implications on both readability and security within cryptographic frameworks.

Encryption Algorithm:

```
A. Input: A plaintext message (message) and a substitution key (encoded_character), which is a shuffled version of the alphabet.
B. Create a mapping:
    a. Define the normal_character as the standard alphabet.
    b. Use the encoded_character to create a dictionary (mp) that maps each character in normal_character to the corresponding character in encoded_character.
C. Initialize the encrypted message: Set encrypted_message as an empty string.
D. Encrypt each character:
    a. For each character char in message:
        i. If char is in mp, append mp[char] to encrypted_message.
```

```
            ii. If char is not in the alphabet (e.g., spaces or punctuation), append char
                directly to encrypted_message.
  E. Output: The encrypted_message now contains the ciphertext.
```

Decryption Algorithm:

```
  A. Input: An encrypted message (encrypted_message) and the same substitution key
     (encoded_character).
  B. Create a reverse mapping:
       a. Define normal_character as the standard alphabet.
       b. Use encoded_character to create a reverse dictionary (mp2) that maps each
          character in encoded_character back to the corresponding character in
          normal_character.
  C. Initialize the decrypted message: Set decrypted_message as an empty string.
  D. Decrypt each character:
       a. For each character char in encrypted_message:
             i. If char is in mp2, append mp2[char] to decrypted_message.
            ii. If char is not in the alphabet (e.g., spaces or punctuation), append char
                directly to decrypted_message.
  E. Output: The decrypted_message now contains the original plaintext.
```

Code:

```python
normal_character = "abcdefghijklmnopqrstuvwxyz".upper()
encoded_character = "qwertyuiopasdfghjklzxcvbnm".upper()

mp = dict(zip(normal_character, encoded_character))
mp2 = dict(zip(encoded_character, normal_character))

def mono_alphabetic_encrypt(message):
    encrypted_message = ""
    for char in message:
        encrypted_message += mp[char]
    return encrypted_message

def mono_alphabetic_decrypt(encrypted_message):
    decrypted_message = ""
    for char in encrypted_message:
        decrypted_message += mp2[char]
    return decrypted_message

message = input("Enter message: ").upper()
encrypted_message = mono_alphabetic_encrypt(message)
decrypted_message = mono_alphabetic_decrypt(encrypted_message)
print("Encrypted message: ", encrypted_message)
print("Decrypted message: ", decrypted_message)
```

Input:

```
Enter message: Communication
```

Output:

```
Encrypted message:  EGDDXFOEQZOGF
Decrypted message:  COMMUNICATION
```

Experiment Number: 03
Experiment Name: Write a program to implement encryption and decryption using Brute force attack cipher.

Objectives:

- To decrypt a message that has been encrypted by trying every conceivable key value without knowing the encryption key.
- To learn about brute-force assaults and how they may be used to crack basic ciphers, such as the Caesar Cipher.

Theory: A brute-force attack is a technique that involves methodically trying every key until the right one is found in order to decode encrypted data. This approach works with straightforward ciphers with a small and controllable key space, such the Caesar Cipher. For example, a Caesar Cipher adjusts each character by a predetermined number of locations, and the original message may be revealed by testing only 25 potential shifts (1–25). The limits of poor encryption systems are brought to light by brute-force assaults, which rely on testing every key instead of utilizing patterns or other information.

Brute-Force Attack Algorithm:

1. Input: Get the encrypted_text that has been encrypted with a Caesar Cipher (or a similar substitution cipher).
2. Define the Key Range:
    - ➢ For a Caesar Cipher, there are 25 possible keys (1 to 25), as shifting by 26 would simply return the original text.
3. Loop through All Keys:
    - ➢ For each possible shift key from 1 to 25:
        - Initialize an empty string decrypted_text to hold the attempted decryption.
4. Attempt Decryption for Each Key:
    - ➢ For each character char in encrypted_text:
        - If char is an uppercase letter:
            - ▪ Shift char backward by the current key number of positions.
            - ▪ Wrap around to the end of the alphabet if the shift moves before "A."
        - If char is a lowercase letter:
            - ▪ Shift char backward by the current key number of positions.
            - ▪ Wrap around to the end of the alphabet if the shift moves before "a."
        - If char is a non-alphabet character:
            - ▪ Leave it unchanged.
        - Append the resulting character to decrypted_text.

5. Store the Decrypted Result:
   - After decrypting the entire encrypted_text with the current key, store or display the resulting decrypted_text as a possible solution.
6. Identify the Correct Decryption:
   - Review each decrypted_text output to find the correct message, which is typically recognizable by its readable content in the intended language.
7. Output: Display each possible decrypted_text for manual review or use additional language processing to identify readable text.

Code:

```
import string

alphabet = string.ascii_lowercase
mp = dict(zip(alphabet, range(26)))
mp2 = dict(zip(range(26), alphabet))

def caesar_encrypt(message, shift):
    encrypted_message = ""
    for char in message:
        encrypted_message += mp2[(mp[char] + shift) % 26]
    return encrypted_message

def bruteforce_decrypt(encrypted_message):
    decrypted_message = []
    for i in range(26):
        decryp = ""
        for j in range(len(encrypted_message)):
            decryp += mp2[(mp[encrypted_message[j]] - i + 26) % 26]
        decrypted_message.append(decryp)
    return decrypted_message

message = "hello"
shift = 3
encrypted_message = caesar_encrypt(message, shift)
decrypted = bruteforce_decrypt(encrypted_message)

print("Encrypted Message:", encrypted_message)
print(
    "Decrypted Messages",
)
ind = 0
for msg in decrypted:
    print(f"For key {ind}: ", msg)
    ind += 1
```

Output:

```
Message: hello

Encrypted Message: khoor

Decrypted Messages
For key 0:  khoor
For key 1:  jgnnq
For key 2:  ifmmp
For key 3:  hello
For key 4:  gdkkn
For key 5:  fcjjm
For key 6:  ebiil
For key 7:  dahhk
For key 8:  czggj
For key 9:  byffi
For key 10:  axeeh
For key 11:  zwddg
For key 12:  yvccf
For key 13:  xubbe
For key 14:  wtaad
For key 15:  vszzc
For key 16:  uryyb
For key 17:  tqxxa
For key 18:  spwwz
For key 19:  rovvy
For key 20:  qnuux
For key 21:  pmttw
For key 22:  olssv
For key 23:  nkrru
For key 24:  mjqqt
For key 25:  lipps
```

Experiment Number: 04
Experiment Name: Write a program to implement encryption and decryption using Hill cipher.

Objectives:

- To utilize matrix multiplication with a predetermined key matrix to encrypt plaintext using the Hill Cipher.
- To recover the original plaintext, decrypt the ciphertext using the key matrix's inverse.
- To Learn how the Hill Cipher encrypts data using linear algebra, specifically matrix operations, which makes it more robust than monoalphabetic substitution ciphers.

Theory: The Hill Cypher is a polygraphic substitution cipher, which uses ideas from linear algebra to encrypt many letters at once. This cipher was created by Lester S. Hill in 1929 and converts a block of plaintext letters into ciphertext by multiplying it by an invertible matrix, also known as a key matrix, over a specified modulo (often modulo 26 for the English alphabet).

Key elements of the hill cipher:

- Matrix as key: An n × n matrix of integers, where n is the block size, serves as the Hill Cipher's key. A 2x2 key matrix, for instance, encrypts two letters simultaneously.
- Vectorization of Plaintext: Numerical vectors are created by dividing plaintext into blocks of size n.
- Matrix Multiplication: To create an encrypted vector, each block is multiplied by the key matrix.
- Modular Arithmetic: To maintain the characters inside the alphabet range, the resultant values are taken modulo 26.
- Decryption Inverse Matrix: To decrypt, multiply the ciphertext blocks by the inverse of the key matrix, if one exists.

Code:

```
from sympy import Matrix

def get_key_matrix(key):
   return Matrix(3, 3, [ord(char) % 65 for char in key])

def hill_encrypt(message, key):
   message_vector = Matrix([ord(char) % 65 for char in message])
   cipher_vector = (key * message_vector) % 26
   ciphertext = "".join(chr(int(num) + 65) for num in cipher_vector)
   return ciphertext

def hill_decrypt(ciphertext, key):
    key_inv = key.inv_mod(26)  # Get modular inverse of the key matrix
    cipher_vector = Matrix([ord(char) % 65 for char in ciphertext])
    message_vector = (key_inv * cipher_vector) % 26
    decrypted_message = "".join(chr(int(num) + 65) for num in message_vector)
    return decrypted_message

# Example usage
message = "ONI"
key = "ABCDEGFGH"
key_matrix = get_key_matrix(key)

ciphertext = hill_encrypt(message, key_matrix)
print("Ciphertext:", ciphertext)

decrypted_message = hill_decrypt(ciphertext, key_matrix)
print("Decrypted Message:", decrypted_message)
```

Output:

```
Message: ONI
Key: ABCDEGFGH

Ciphertext: DMW
Decrypted Message: ONI
```

Experiment Number: 05
Experiment Name: Write a program to implement encryption using Playfair cipher.

Objectives:

- To utilize the Playfair Cipher to encrypt a plaintext message by arranging the letters in a 5x5 grid key matrix.
- To recognize the idea behind digraph encryption, which encrypts letter pairs rather than single characters.

Theory: The first workable digraph substitution cipher was the Playfair cipher. Charles Wheatstone created the method in 1854, but it was called for Lord Playfair, who encouraged the adoption of the cipher. In contrast to a typical cipher, the Playfair cipher encrypts two alphabets (digraphs) rather than just one.

Key Elements of the Playfair Cipher

1. 5x5 Key Matrix: The Playfair Cipher uses a 5x5 grid as its key matrix, which includes the letters of the alphabet. Typically, the letter "J" is omitted, and "I" and "J" are treated as the same letter.
2. Key Word: A keyword or phrase is chosen and used to fill the 5x5 matrix, with remaining letters added in alphabetical order. Duplicate letters in the keyword are excluded.
3. Digraph Pairing: The plaintext message is divided into pairs of letters. If a pair contains duplicate letters, an "X" is inserted between them, and if there's an odd number of letters, a filler "X" is added at the end.
4. Encryption Rules:
   - Same Row: If the two letters of a pair appear in the same row of the matrix, each letter is replaced by the letter immediately to its right (wrapping around to the beginning if needed).
   - Same Column: If the two letters of a pair appear in the same column, each letter is replaced by the letter immediately below it (wrapping to the top if needed).
   - Rectangle: If the letters form the corners of a rectangle, each letter is replaced by the letter in its row at the other corner of the rectangle.

By encrypting in pairs and using a key matrix, the Playfair Cipher creates more complex substitutions, improving security compared to monoalphabetic ciphers. However, it is still vulnerable to certain cryptanalysis techniques, especially if a substantial amount of ciphertext is available for analysis.

Code:

```
import string

def playfair_encrypt(message, key):
    alphabet = string.ascii_lowercase.replace("j", "")
```

```
message = message.lower().replace(" ", "").replace("j", "i")
key = key.lower().replace(" ", "").replace("j", "i")

# Generate key square
key_square = ""
for char in key + alphabet:
   if char not in key_square:
      key_square += char

# Prepare message by handling double letters and padding with 'x' if needed
prepared_message = ""
i = 0
while i < len(message) - 1:
   if message[i] == message[i + 1]:  # Insert 'x' between duplicate letters
      prepared_message += message[i] + "x"
      i += 1
   else:
      prepared_message += message[i] + message[i + 1]
      i += 2
if i < len(message):
   prepared_message += message[i]
if len(prepared_message) % 2 == 1:
   prepared_message += "x"  # Pad with 'x' if message length is odd

# Split message into digraphs (pairs of two characters)
digraphs = [prepared_message[i : i + 2] for i in range(0, len(prepared_message), 2)]

# Encrypt each digraph
def encrypt_digraph(digraph):
   a, b = digraph
   row_a, col_a = divmod(key_square.index(a), 5)
   row_b, col_b = divmod(key_square.index(b), 5)

   if row_a == row_b:  # Same row
      col_a = (col_a + 1) % 5
      col_b = (col_b + 1) % 5
   elif col_a == col_b:  # Same column
      row_a = (row_a + 1) % 5
      row_b = (row_b + 1) % 5
   else:  # Rectangle swap
      col_a, col_b = col_b, col_a

   return key_square[row_a * 5 + col_a] + key_square[row_b * 5 + col_b]

# Generate encrypted message
```

```
    encrypted_message = "".join(encrypt_digraph(digraph) for digraph in digraphs)
    return encrypted_message

# Example usage
message = "communication"
key = "computer"
ciphertext = playfair_encrypt(message, key)
print("Cipher Text:", ciphertext)
```

Output:

```
Message: communication
Key: computer
Cipher Text: omrmpcsgptbdml
```

Experiment Number: 06
Experiment Name: Write a program to implement decryption using Playfair cipher.

Objectives:

- Decrypt a ciphertext encrypted with the Playfair Cipher by reversing the encryption rules with a 5x5 key matrix.
- Understand the concept of digraph decryption, where pairs of letters are decrypted simultaneously.

Theory: The Playfair encryption is a polygraphic encryption that uses a 5x5 key matrix to encrypt and decrypt digraphs, or letter pairings. Reversing the Playfair encryption rules, each pair of letters in the ciphertext is decrypted according to where they are in the key matrix. Compared to straightforward monoalphabetic ciphers, the Playfair Cipher is more difficult to crack since it was one of the first encryption methods to encrypt several characters simultaneously.

Key Elements of Playfair Cipher Decryption

1. 5x5 Key Matrix: The same 5x5 grid of letters used in encryption is essential for decryption. This matrix is filled with a keyword or phrase, with duplicates removed and remaining letters filled in alphabetically (usually omitting "J" and treating "I" and "J" as the same letter).
2. Ciphertext Digraphs: The ciphertext is divided into digraphs (pairs of letters), similar to how the plaintext was handled during encryption.
3. Decryption Rules:
    - Same Row: If the two letters in a pair are in the same row, each letter is replaced by the letter immediately to its left (wrapping to the end of the row if necessary).
    - Same Column: If the two letters are in the same column, each letter is replaced by the letter immediately above it (wrapping to the bottom if necessary).
    - Rectangle: If the letters form the corners of a rectangle, each letter is replaced by the letter in its row at the opposite corner of the rectangle.

This cipher provides greater security than simple monoalphabetic substitutions but is still susceptible to cryptanalysis, especially if the key matrix or a large amount of ciphertext is available.

Code:

```
import string

def playfair_decrypt(ciphertext, key):
    alphabet = string.ascii_lowercase.replace("j", "")
    ciphertext = ciphertext.lower().replace(" ", "")
    key = key.lower().replace(" ", "").replace("j", "i")
    key_square = ""
    for char in key + alphabet:
        if char not in key_square:
            key_square += char

    # Split ciphertext into digraphs (pairs of two characters)
    digraphs = [ciphertext[i : i + 2] for i in range(0, len(ciphertext), 2)]

    # Decrypt each digraph
    def decrypt_digraph(digraph):
        a, b = digraph
        row_a, col_a = divmod(key_square.index(a), 5)
        row_b, col_b = divmod(key_square.index(b), 5)

        if row_a == row_b:  # Same row
            col_a = (col_a - 1) % 5
            col_b = (col_b - 1) % 5
        elif col_a == col_b:  # Same column
            row_a = (row_a - 1) % 5
            row_b = (row_b - 1) % 5
        else:  # Rectangle swap
            col_a, col_b = col_b, col_a

        return key_square[row_a * 5 + col_a] + key_square[row_b * 5 + col_b]

    decrypted_message = "".join(decrypt_digraph(digraph) for digraph in digraphs)
    return decrypted_message

ciphertext = "omrmpcsgptbdml"
key = "computer"
decrypted_text = playfair_decrypt(ciphertext, key)
print("Decrypted Text:", decrypted_text)
plain = ""
for x in decrypted_text:
    if x == "x":
        continue
    plain += x
print("Refined Text:", plain)
```

Output:

```
Ciphertext: omrmpcsgptbdml
Key: computer


Decrypted Text: comxmunication
Refined Text: communication
```

Experiment Number: 07
Experiment Name: Write a program to implement encryption using Poly-Alphabetic cipher.

Objectives:

- To encrypt a plaintext message using the Vigenère Cipher, which employs a repeating key to apply multiple shifts to the plaintext.
- To understand the Vigenère Cipher as an example of a polyalphabetic cipher, which improves security over traditional monoalphabetic ciphers by masking letter frequency patterns.

Theory: A polyalphabetic substitution cipher, the Vigenère Cipher encrypts a message using a keyword. Blaise de Vigenère was the first to describe it in 1586. Each letter in the plaintext is shifted by a number of positions defined by a matching letter in the keyword, which is the fundamental idea underlying this cipher. Frequency analysis becomes more challenging for cryptanalysts since various shifts are applied to distinct plaintext letters because the keyword is repeated to fit the plaintext's length.

Algorithm for Vigenère Cipher Encryption:

1. Input:
    - A plaintext message (P) to be encrypted.
    - A keyword (K) used for encryption.
2. Preprocess:
    - Remove any spaces or non-alphabetic characters from the plaintext.
    - Ensure that both the plaintext and the keyword are in uppercase (or lowercase, but consistent).
3. Repeat the Keyword:
    - Repeat the keyword until its length is equal to the length of the plaintext.
4. Encrypt Each Letter:
    - For each letter P_i of the plaintext and the corresponding letter K_i from the keyword:
        - Convert P_i and K_i to their numeric values based on their position in the alphabet (A=0, B=1, ..., Z=25).
        - Calculate the encrypted letter C_i as:

$$C_i = (P_i + K_i) mod\ 26$$

        - Convert C_i back to a letter.
5. Output:
    - The resulting sequence of encrypted letters forms the ciphertext (C).

Example Steps

- **Plaintext**: "HELLO"
- **Keyword**: "KEY"

1. Remove spaces or non-alphabetic characters (if any).
    - Plaintext: "HELLO"
    - Keyword: "KEY"
2. Repeat the keyword to match the length of the plaintext.
    - Plaintext: "HELLO"
    - Keyword: "KEYKE"
3. Convert plaintext and keyword to their numeric values:
    - Plaintext: H=7, E=4, L=11, L=11, O=14
    - Keyword: K=10, E=4, Y=24, K=10, E=4
4. Apply the encryption formula for each letter:
    - (H + K) % 26 = (7 + 10) % 26 = 17 → R
    - (E + E) % 26 = (4 + 4) % 26 = 8 → I
    - (L + Y) % 26 = (11 + 24) % 26 = 9 → J
    - (L + K) % 26 = (11 + 10) % 26 = 21 → V
    - (O + E) % 26 = (14 + 4) % 26 = 18 → S
5. **Ciphertext**: "RIJVS"

Code:

```
import string

alphabetic = string.ascii_lowercase
mp = dict(zip(alphabetic, range(26)))
mp2 = dict(zip(range(26), alphabetic))

def generateKey(message, key):
   key_word = ""
   for i in range(len(message)):
      key_word += key[i % len(key)]
   return key_word

def encrypt(message, key):
   encrypted_message = ""
   for i in range(len(message)):
      encrypted_message += mp2[(mp[message[i]] + mp[key[i]]) % 26]
   return encrypted_message

# Example usage
message = "sajeeb"
key = generateKey(message, "ice")
```

```
encrypted_message = encrypt(message, key)
print("Encrypted Message: ", encrypted_message)
```

Output:

```
Message: sajeeb
Key: ice
Encrypted Message:  acnmgf
```

Experiment Number: 08
Experiment Name: Write a program to implement decryption using Poly-Alphabetic cipher.

Objectives:

- Decrypt a ciphertext encrypted using the Vigenère Cipher, reversing the encryption steps.
- Understand the concept of reversing polyalphabetic substitutions by using the same key that was used during encryption.

Theory: The Vigenère Cipher is a poly-alphabetic cipher that uses a number of distinct Caesar ciphers depending on a keyword's letters to encrypt text. To match the plaintext's length, each letter in the plaintext is moved in accordance with a matching letter in the keyword, which is repeated as necessary. As a result, the generated ciphertext is a series of Caesar shifts that change throughout the message rather than a straightforward substitution cipher.

The Vigenère Cipher's decryption procedure is basically the opposite of its encrypting procedure. The matching letter of the keyword determines how many places each letter of the plaintext is moved forward in encryption. In decryption, the original message is recovered by shifting the ciphertext's characters backward by the same amount.

Algorithm for Vigenère Cipher Decryption:

1. Input:
   - A ciphertext (C) to be decrypted.
   - A keyword (K) used during encryption.
2. Preprocess:
   - Remove any spaces or non-alphabetic characters from the ciphertext.
   - Ensure that both the ciphertext and the keyword are in uppercase (or lowercase, but consistent).
3. Repeat the Keyword:
   - Repeat the keyword to match the length of the ciphertext.
4. Decrypt Each Letter:
   - For each letter C_i of the ciphertext and the corresponding letter K_i from the keyword:
     - Convert C_i and K_i to their numeric values based on their position in the alphabet (A=0, B=1, ..., Z=25).
     - Calculate the decrypted letter P_i as:
       $$P_i = (C_i - K_i + 26)$$
     - Convert P_i back to a letter.
5. Output:
   - The resulting sequence of decrypted letters forms the plaintext (P).

<u>Example Steps:</u>

- Ciphertext: "RIJVS"
- Keyword: "KEY"

1. Remove spaces or non-alphabetic characters (if any).
    - Ciphertext: "RIJVS"
    - Keyword: "KEY"
2. Repeat the keyword to match the length of the ciphertext.
    - Ciphertext: "RIJVS"
    - Keyword: "KEYKE"
3. Convert ciphertext and keyword to their numeric values:
    - Ciphertext: R=17, I=8, J=9, V=21, S=18
    - Keyword: K=10, E=4, Y=24, K=10, E=4
4. Apply the decryption formula for each letter:
    - (R - K + 26) % 26 = (17 - 10 + 26) % 26 = 7 → H
    - (I - E + 26) % 26 = (8 - 4 + 26) % 26 = 4 → E
    - (J - Y + 26) % 26 = (9 - 24 + 26) % 26 = 11 → L
    - (V - K + 26) % 26 = (21 - 10 + 26) % 26 = 11 → L
    - (S - E + 26) % 26 = (18 - 4 + 26) % 26 = 14 → O
5. Plaintext: "HELLO"

Code:

```
import string

alphabetic = string.ascii_lowercase
mp = dict(zip(alphabetic, range(26)))
mp2 = dict(zip(range(26), alphabetic))

def generateKey(message, key):
   key_word = ""
   for i in range(len(message)):
      key_word += key[i % len(key)]
   return key_word

def decrypt(encrypted_message, key):
   decrypted_message = ""
   for i in range(len(encrypted_message)):
      decrypted_message += mp2[(mp[encrypted_message[i]] - mp[key[i]] + 26) % 26]
   return decrypted_message

# Example usage
encrypted_message = "acnmgf"
key = generateKey(encrypted_message, "ice")
```

```
decrypted_message = decrypt(encrypted_message, key)
print("Decrypted Message:", decrypted_message)
```

Output:

```
Encrypted Message:  acnmgf
Decrypted Message: sajeeb
```

Experiment Number: 09
Experiment Name: Write a program to implement encryption using Vernam cipher.

Objectives:

- To understand how bitwise XOR is used in encryption.
- To implement the Vernam Cipher decryption algorithm.

Theory: The Vernam Cipher, also known as the One-Time Pad (OTP), is a symmetric-key cipher where each character in the plaintext is XORed with a key. The cipher works by using a random key that is as long as the message itself. Each bit or character of the plaintext is combined with the corresponding bit or character of the key using the bitwise XOR (exclusive OR) operation.
The key feature of the Vernam Cipher is that it is unbreakable if the following conditions are met:

1. The key must be truly random.
2. The key must be as long as the message.
3. The key must be used only once and securely discarded after use.

The cipher achieves perfect secrecy, meaning that the ciphertext reveals no information about the plaintext without the key. However, the practical difficulty lies in generating and securely transmitting such long random keys.

Encryption Algorithm:

- Assign a number to each character of the plain text (P_i) and the key (K_i) according to alphabetical order.
- Bitwise XOR both the number (Corresponding plain-text character number and Key character number).

$$Ciphertext, C_i = P_i \oplus K_i$$

- Subtract the number from 26 if the resulting number is greater than or equal to 26, if it isn't then leave it.

Code:

```
import random
import string

alphabet = string.ascii_lowercase
mp = dict(zip(alphabet, range(26)))
mp2 = dict(zip(range(26), alphabet))

def encrypt(message, key):
  encrypted_message = ""
  encrypted_code = []
  for i in range(len(message)):
    xor = mp[message[i]] ^ mp[key[i]]
```

```
        encrypted_code.append(xor)
        encrypted_message += mp2[xor % 26]
    return encrypted_message, encrypted_code

message = "ice"
# key = "inf".lower() #random key needed
key = ""
for i in range(len(message)):
    key += chr(random.randint(65, 90))
print("Genereated Key: ", key)
key = key.lower()

encrypted_message, encrypted_code = encrypt(message, key)
print("Encrypted Message: ", encrypted_message)
print("Encrypted Code: ", encrypted_code)
```

Output:

```
Message: ice
Generated Key:  YOQ
Encrypted Message:  qmu
Encrypted Code:  [16, 12, 20]
```

Experiment Number: 10
Experiment Name: Write a program to implement decryption using Vernam cipher.

Objectives:

- To understand how XOR is used in decryption.
- To implement the Vernam Cipher decryption algorithm.

Theory: The Vernam Cipher decryption process is identical to encryption because it uses the bitwise XOR operation, which is its own inverse. When the same key is applied to the ciphertext, the original plaintext is retrieved. The key used in decryption must be exactly the same as the one used for encryption. If the key is truly random and used only once, the ciphertext becomes unreadable without the key, ensuring perfect secrecy.

Decryption Algorithm:

- Assign a number to each character of the ciphertext (C_i) and the key (K_i) based on alphabetical order.
- Perform bitwise XOR the corresponding number of the ciphertext character and the key character.

$$Plaintext, P_i = C_i \oplus K_i$$

- If the resulting number is greater than or equal to 26, subtract 26 from it to keep it within the range of the alphabet. Convert the result back to the corresponding plaintext character (P_i).

Code:

```
import string

alphabet = string.ascii_lowercase
mp = dict(zip(alphabet, range(26)))
mp2 = dict(zip(range(26), alphabet))

def decrypt(encrypted_code, key):
    decrypted_message = ""
    for i in range(len(encrypted_code)):
        xor = encrypted_code[i] ^ mp[key[i]]
        decrypted_message += mp2[xor % 26]
    return decrypted_message

# Example encrypted code and key
encrypted_code = [0, 15, 1]
key = "inf"  # The key used for encryption

decrypted_message = decrypt(encrypted_code, key)
print("Encrypted Code: ", encrypted_code)
print("Key: ", key)
print("Decrypted Message", decrypted_message)
```

Output:

```
Encrypted Code:  [0, 15, 1]
Key:  inf
Decrypted Message:  ice
```