Experiment No: 01

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

Objectives

1. **Understand the Caesar Cipher**: It's a simple substitution cipher where each letter in the text is "shifted" a certain number of places up or down the alphabet.
2. **Implement encryption**: Convert each letter in the plaintext by shifting it right by a specified number of positions.
3. **Implement decryption**: Convert each letter in the ciphertext by shifting it left by the same number of positions to retrieve the original text.

Theory

The **Caesar Cipher** is one of the oldest and simplest encryption techniques. It's named after Julius Caesar, who used it to protect his military communications. Here's how it works:

1. Each letter in the plaintext is replaced by a letter some fixed number of positions down or up the alphabet.
2. The shift value (or "key") determines how many places each letter in the alphabet will be shifted.
3. If the shift value is 3, then:
    - A would be replaced by D
    - B would be replaced by E
    - Y would be replaced by B, and so on, looping back to the start of the alphabet if necessary.

The formula to encrypt a letter **c** (where **c** is a letter) in Caesar Cipher is: $E(x)=(x+shift) \bmod 26$

The formula to decrypt the cipher is: $D(x)=(x-shift) \bmod 26$

Program

Here's the Python program to perform encryption and decryption using the Caesar Cipher:

```
def encrypt(text, shift):
    encrypted_text = ""
    # Traverse the text
    for char in text:
        if char.isalpha():  # Check if character is an alphabet
            # Determine offset for uppercase or lowercase letters
            offset = 65 if char.isupper() else 97
```

```
            # Encrypt character and add to result
            encrypted_text += chr((ord(char) - offset + shift) % 26 + offset)
        else:
            # Keep any non-alphabet characters as they are
            encrypted_text += char

    return encrypted_text
def decrypt(text, shift):
    decrypted_text = ""

    # Traverse the text
    for char in text:
        if char.isalpha():  # Check if character is an alphabet
            # Determine offset for uppercase or lowercase letters
            offset = 65 if char.isupper() else 97
            # Decrypt character and add to result
            decrypted_text += chr((ord(char) - offset - shift) % 26 + offset)
        else:
            # Keep any non-alphabet characters as they are
            decrypted_text += char

    return decrypted_text
# Example Usage
plaintext = "Hello, World!"
shift = 3
encrypted_message = encrypt(plaintext, shift)
decrypted_message = decrypt(encrypted_message, shift)
print("Original Message:", plaintext)
print("Encrypted Message:", encrypted_message)
print("Decrypted Message:", decrypted_message)
```

Output:

```
Original Message: Hello, World!
Encrypted Message: Khoor, Zruog!
Decrypted Message: Hello, World!
```

Experiment No: 02

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

Objectives:

1. Understanding of Cryptography Concepts: Gain practical knowledge of encryption techniques by implementing a basic symmetric encryption algorithm.
2. Exploration of Substitution Ciphers: Understand the concept of substitution ciphers, particularly the Mono-Alphabetic cipher, which involves substituting each letter of the plaintext with a corresponding letter from the cipher alphabet.

Theory:

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

In this technique, any plaintext letter can be substituted with any of the ciphertext letters, which means no ciphertext letter can not be repeated. Such an approach is referred to as monoalphabetic substitution cipher because a single cipher alphabet is used per message.
Example: Let us consider an example:
Plaintext: cryptography and computer network

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

So, the encrypted ciphertext will be: UVROASVDOKR DIB UCLOYANV INAMCVW
Ciphertext: UVROASVDOKRDIBUCLOYANVINAMCVW
Monoalphabetic cipher is easy to break because it reflects the frequency data of the original alphabet. A countermeasure is to provide multiple substitutes, known as homophones, for a single letter.
For example, the letter e could be assigned a number of different cipher symbols, such as 16, 74, 35, and 21, with each homophone assigned to a letter in rotation or randomly. If the number of symbols assigned to each letter is proportional to the relative frequency of that letter, then single-letter frequency information is completely obliterated
However, even with homophones, each element of plaintext affects only one element of ciphertext, and multiple-letter patterns (e.g., diagram frequencies) still survive in the ciphertext, making cryptanalysis relatively straightforward.

Program

Here's the Python code for a Mono-Alphabetic Cipher with encryption and decryption functions. This code also generates a random substitution key each time to ensure a strong encryption.

```
import random
import string

def generate_random_key():
    alphabet = list(string.ascii_uppercase)
    shuffled_alphabet = alphabet[:]
    random.shuffle(shuffled_alphabet)

    return dict(zip(alphabet, shuffled_alphabet))


def encrypt(text, key):
    encrypted_text = ""

    # Traverse the text
    for char in text:
        if char.isalpha():  # Encrypt only alphabet characters
            # Convert to uppercase for standardization
            uppercase_char = char.upper()
            encrypted_char = key[uppercase_char]
            # Preserve original case
            encrypted_text += encrypted_char if char.isupper() else
encrypted_char.lower()
        else:
            # Keep non-alphabet characters as they are
            encrypted_text += char

    return encrypted_text


def decrypt(text, key):
    # Reverse the key to decrypt
    reverse_key = {v: k for k, v in key.items()}
    decrypted_text = ""

    # Traverse the text
    for char in text:
        if char.isalpha():  # Decrypt only alphabet characters
            # Convert to uppercase for standardization
```

```
            uppercase_char = char.upper()
            decrypted_char = reverse_key[uppercase_char]
            # Preserve original case
            decrypted_text += decrypted_char if char.isupper() else
decrypted_char.lower()
        else:
            # Keep non-alphabet characters as they are
            decrypted_text += char

    return decrypted_text

# Example Usage
plaintext = "Hello, World!"
key = generate_random_key()  # Generate a random substitution key

encrypted_message = encrypt(plaintext, key)
decrypted_message = decrypt(encrypted_message, key)

print("Original Message:", plaintext)
print("Encryption Key:", key)
print("Encrypted Message:", encrypted_message)
print("Decrypted Message:", decrypted_message)
```

Output:

```
Original Message: Hello, World!
Encryption Key: {'A': 'G', 'B': 'F', 'C': 'R', 'D': 'P', 'E': 'Q', 'F': 'Z', 'G':
'W', 'H': 'J', 'I': 'N', 'J': 'V', 'K': 'C', 'L': 'T', 'M': 'S', 'N': 'L', 'O':
'H', 'P': 'U', 'Q': 'A', 'R': 'E', 'S': 'O', 'T': 'B', 'U': 'M', 'V': 'X', 'W':
'D', 'X': 'Y', 'Y': 'K', 'Z': 'I'}
Encrypted Message: Jqtth, Dhetp!
Decrypted Message: Hello, World!
```

Experiment No: 03

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

Objectives:

1. Understanding of Cryptanalysis: Gain practical knowledge of cryptanalysis techniques, particularly brute force attacks, which involve systematically trying all possible keys until the correct one is found.
2. Algorithmic Understanding: Understand the structure and characteristics of the cipher being targeted by the brute force attack. This includes understanding the encryption algorithm, the key space, and any known vulnerabilities or weaknesses that can be exploited.
3. Security Awareness: Increase awareness of encryption techniques and their role in securing sensitive information. Explore the limitations of brute force attacks, including their computational complexity and effectiveness against different types of ciphers.

Theory:

Understanding the Brute Force Attack:

A brute force attack is an exhaustive search method that systematically tries all possible combinations until the correct solution is found.
In the context of decryption, a brute force attack involves trying every possible key to decrypt the ciphertext until the original plaintext is recovered.
Brute force attacks are often used when no other attack method is feasible, such as when the encryption key is unknown or the encryption algorithm is resistant to other types of attacks.

Key Space:
The key space refers to the total number of possible keys that can be used in the encryption algorithm. For symmetric encryption algorithms, such as Caesar cipher or Vigenère cipher, the key space represents all possible shifts or combinations of characters in the key.
The size of the key space determines the feasibility of a brute force attack. A larger key space makes brute force attacks more computationally expensive and time-consuming.

Implementation:
To implement a brute force attack for decryption, start by generating all possible keys within the key space.

For each generated key, decrypt the ciphertext using the encryption algorithm.
Compare the decrypted plaintext with known patterns or characteristics of the original plaintext.

If a match is found, the correct key has been discovered, and the decryption process can be stopped. If no match is found after trying all possible keys, the ciphertext may be incorrectly encrypted, or the key space may be too large for a brute force attack to be practical.

The computational complexity of a brute force attack depends on the size of the key space.
For encryption algorithms with small key spaces, such as Caesar cipher with a shift of 1, a brute force attack can be performed quickly.
However, for encryption algorithms with large key spaces, such as modern block ciphers like AES with a 128-bit key, brute force attacks are computationally infeasible due to the vast number of possible keys.

Limitations:
Brute force attacks are not always practical, especially for encryption algorithms with large key spaces. They require significant computational resources and time to search through all possible keys. Brute force attacks may also be ineffective against encryption algorithms with additional security features, such as key stretching or password hashing.
In summary, implementing encryption and decryption using a brute force attack involves systematically trying all possible keys until the correct one is found. The feasibility of a brute force attack depends on the size of the key space and the computational resources available.

Program

```
def caesar_encrypt(text, shift):
    encrypted_text = ""

    for char in text:
        if char.isalpha():
            offset = 65 if char.isupper() else 97
            encrypted_text += chr((ord(char) - offset + shift) % 26 + offset)
        else:
            encrypted_text += char

    return encrypted_text

def brute_force_caesar(ciphertext):
    print("Attempting all possible shifts for decryption:")
    for shift in range(26):
        decrypted_text = ""

        for char in ciphertext:
            if char.isalpha():
                offset = 65 if char.isupper() else 97
                decrypted_text += chr((ord(char) - offset - shift) % 26 + offset)
            else:
                decrypted_text += char
        print(f"Shift {shift}: {decrypted_text}")

# Example usage
plaintext = "Hello, World!"
shift = 3
ciphertext = caesar_encrypt(plaintext, shift)

print("Original Message:", plaintext)
print("Encrypted Message:", ciphertext)
print("\nBrute Force Decryption Attempts:")
brute_force_caesar(ciphertext)
```

Output:

```
Original Message: Hello, World!
Encrypted Message: Khoor, Zruog!

Brute Force Decryption Attempts:
Attempting all possible shifts for decryption:
```

```
Shift 0: Khoor, Zruog!
Shift 1: Jgnnq, Yqtnf!
Shift 2: Ifmmp, Xpsme!
Shift 3: Hello, World!
Shift 4: Gdkkn, Vnqkc!
Shift 5: Fcjjm, Umpjb!
Shift 6: Ebiil, Tloia!
Shift 7: Dahhk, Sknhz!
Shift 8: Czggj, Rjmgy!
Shift 9: Byffi, Qilfx!
Shift 10: Axeeh, Phkew!
Shift 11: Zwddg, Ogjdv!
Shift 12: Yvccf, Nficu!
Shift 13: Xubbe, Mehbt!
Shift 14: Wtaad, Ldgas!
Shift 15: Vszzc, Kcfzr!
Shift 16: Uryyb, Jbeyq!
Shift 17: Tqxxa, Iadxp!
Shift 18: Spwwz, Hzcwo!
Shift 19: Rovvy, Gybvn!
Shift 20: Qnuux, Fxaum!
Shift 21: Pmttw, Ewztl!
Shift 22: Olssv, Dvysk!
Shift 23: Nkrru, Cuxrj!
Shift 24: Mjqqt, Btwqi!
Shift 25: Lipps, Asvph!
```

Experiment No: 04

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

Objectives:

1. Algorithmic Understanding: Gain a deep understanding of the Hill cipher algorithm, including matrix multiplication and modular arithmetic. Understand how the key matrix is used for encryption and decryption, and how it affects the security of the cipher.
2. Security Awareness: Increase awareness of encryption techniques and their role in securing sensitive information. Explore the strengths and weaknesses of the Hill cipher, particularly in terms of its susceptibility to attacks such as known-plaintext attacks and key size limitations.

Theory:

Hill cipher is a polygraphic substitution cipher based on linear algebra.Each letter is represented by a number modulo 26. Often the simple scheme A = 0, B = 1, …, Z = 25 is used, but this is not an essential feature of the cipher. To encrypt a message, each block of n letters (considered as an
n-component vector) is multiplied by an invertible n × n matrix, against modulus 26. To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption.
The matrix used for encryption is the cipher key, and it should be chosen randomly from the set of invertible n × n matrices (modulo 26).

Program

```
"""
Implementation of Hill Cipher!

Important notation:
K = Matrix which is our 'Secret Key'
P = Vector of plaintext (that has been mapped to numbers)
C = Vector of Ciphered text (in numbers)

C = E(K,P) = K*P (mod X) -- X is length of alphabet used
P = D(K,C) = inv(K)*C (mod X)  -- X is length of alphabet used

"""

import numpy as np
import string
from egcd import egcd  # pip install egcd

alphabet = string.ascii_lowercase
letter_to_index = dict(zip(alphabet, range(len(alphabet))))
index_to_letter = dict(zip(range(len(alphabet)), alphabet))


def matrix_mod_inv(matrix, modulus):
    det = int(np.round(np.linalg.det(matrix)))  # Step 1)
    det_inv = egcd(det, modulus)[1] % modulus  # Step 2)
    matrix_modulus_inv = (
            det_inv * np.round(det * np.linalg.inv(matrix)).astype(int) % modulus
    )  # Step 3)

    return matrix_modulus_inv


def encrypt(message, K):
    encrypted = ""
    message_in_numbers = []

    for letter in message:
        message_in_numbers.append(letter_to_index[letter])

    split_P = [
        message_in_numbers[i: i + int(K.shape[0])]
        for i in range(0, len(message_in_numbers), int(K.shape[0]))
    ]
```

```
    for P in split_P:
        P = np.transpose(np.asarray(P))[:, np.newaxis]

        while P.shape[0] != K.shape[0]:
            P = np.append(P, letter_to_index[" "])[:, np.newaxis]

        numbers = np.dot(K, P) % len(alphabet)
        n = numbers.shape[0]  # length of encrypted message (in numbers)

        # Map back to get encrypted text
        for idx in range(n):
            number = int(numbers[idx, 0])
            encrypted += index_to_letter[number]

    return encrypted


def decrypt(cipher, Kinv):
    decrypted = ""
    cipher_in_numbers = []

    for letter in cipher:
        cipher_in_numbers.append(letter_to_index[letter])

    split_C = [
        cipher_in_numbers[i: i + int(Kinv.shape[0])]
        for i in range(0, len(cipher_in_numbers), int(Kinv.shape[0]))
    ]

    for C in split_C:
        C = np.transpose(np.asarray(C))[:, np.newaxis]
        numbers = np.dot(Kinv, C) % len(alphabet)
        n = numbers.shape[0]

        for idx in range(n):
            number = int(numbers[idx, 0])
            decrypted += index_to_letter[number]

    return decrypted


message = "SohagHossain"
message = message.lower()
K = np.matrix([[3, 3], [2, 5]])
```

```
# K = np.matrix([[6, 24, 1], [13,16,10], [20,17,15]]) # for length of alphabet =
26
# K = np.matrix([[3,10,20],[20,19,17], [23,78,17]]) # for length of alphabet = 27
Kinv = matrix_mod_inv(K, len(alphabet))

encrypted_message = encrypt(message, K)
decrypted_message = decrypt(encrypted_message, Kinv)

print("Original message: " + message)
print("Encrypted message: " + encrypted_message)
print("Decrypted message: " + decrypted_message)
```

Output:

```
Original message: sohaghossain
Encrypted message: scvonvsockld
Decrypted message: sohaghossain
```

Experiment No: 05

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

Objectives:

1. Algorithmic Understanding: Gain a deep understanding of the Playfair cipher algorithm, including how to create and use the key matrix and the rules for encrypting and decrypting plaintext. This objective focuses on mastering the core principles of the cipher and its implementation.
2. Security Awareness: Increase awareness of encryption techniques and their role in securing sensitive information, including understanding the strengths and weaknesses of the Playfair cipher compared to other classical ciphers. This objective emphasizes understanding the security implications of using the Playfair cipher in practical scenarios.

Theory:

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

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

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

- Repeating plaintext letters that are in the same pair are separated with a filler letter, such as x, so that balloon would be treated as ba lx lo on.
- Two plaintext letters that fall in the same row of the matrix are each replaced by the letter to the right, with the first element of the row circularly following the last. For example, ar is encrypted as RM.
- Two plaintext letters that fall in the same column are each replaced by the letter beneath, with the top element of the column circularly following the last. For example, mu is encrypted as CM.
- Otherwise, each plaintext letter in a pair is replaced by the letter that lies in its own

row and the column occupied by the other plaintext letter. Thus, hs becomes BP and ea becomes IM (or JM, as the encipherer wishes).

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

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

Program:

```
def playfair_cipher(plaintext, key, mode):
    # Define the alphabet, excluding 'j'
    alphabet = 'abcdefghiklmnopqrstuvwxyz'
    # Remove whitespace and 'j' from the key and convert to lowercase
    key = key.lower().replace(' ', '').replace('j', 'i')
    # Construct the key square
    key_square = ''
    for letter in key + alphabet:
        if letter not in key_square:
            key_square += letter
    # Split the plaintext into digraphs, padding with 'x' if necessary
    plaintext = plaintext.lower().replace(' ', '').replace('j', 'i')
    replaceplaintext = ''
    if mode == 'encrypt':
        it = 0
        while it < len(plaintext) - 1:
            if plaintext[it] == plaintext[it + 1]:
                replaceplaintext += plaintext[it]
                replaceplaintext += 'x'
                it += 1
            else:
                replaceplaintext += plaintext[it]
                replaceplaintext += plaintext[it + 1]
                it += 2
        replaceplaintext += plaintext[-1] if it < len(plaintext) else ''
        plaintext = replaceplaintext

    if len(plaintext) % 2 == 1:
        plaintext += 'x'
    digraphs = [plaintext[i:i + 2] for i in range(0, len(plaintext), 2)]

    # Define the encryption/decryption functions
```

```
    def encrypt(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:
            col_a = (col_a + 1) % 5
            col_b = (col_b + 1) % 5
        elif col_a == col_b:
            row_a = (row_a + 1) % 5
            row_b = (row_b + 1) % 5
        else:
            col_a, col_b = col_b, col_a
        return key_square[row_a * 5 + col_a] + key_square[row_b * 5 + col_b]

    def decrypt(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:
            col_a = (col_a - 1) % 5
            col_b = (col_b - 1) % 5
        elif col_a == col_b:
            row_a = (row_a - 1) % 5
            row_b = (row_b - 1) % 5
        else:
            col_a, col_b = col_b, col_a
        return key_square[row_a * 5 + col_a] + key_square[row_b * 5 + col_b]

    # Encrypt or decrypt the plaintext
    result = ''
    for digraph in digraphs:
        if mode == 'encrypt':
            result += encrypt(digraph)
        elif mode == 'decrypt':
            result += decrypt(digraph)

    # Return the result
    return result


# Example usage
plaintext = 'caee'
key = 'monkey'
ciphertext = playfair_cipher(plaintext, key, 'encrypt')
print('Cipher Text:', ciphertext)
```

```
decrypted_text = playfair_cipher(ciphertext, key, 'decrypt')
print('Decrypted Text:', decrypted_text)  # (Note: 'x' is added as padding)
```

Output:

```
Cipher Text: dbkzkz
Decrypted Text: caexex
```

Experiment No: 06

Experiment Name: To write a program to implement encryption and decryption using Poly-Alphabetic cipher.

Objectives:

- Algorithmic Understanding: Develop a deep understanding of the Poly-Alphabetic cipher algorithm, particularly focusing on the concept of using multiple alphabets (key streams) to encrypt plaintext. Understand how to create and manage key streams, and how they are applied during encryption and decryption.
- Security Awareness: Increase awareness of encryption techniques and their role in securing sensitive information. Explore the security properties of the Poly-Alphabetic cipher, including its resistance to frequency analysis attacks due to the variability introduced by multiple key streams. Understand its strengths and limitations compared to other classical ciphers.

Theory:

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

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

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

A general equation of the encryption process is Ci = (pi + kimod m) mod 26

decryption is a generalization of Equation pi = (Ci - kimod m) mod 26

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

if the keyword is deceptive, the message “we are discovered save yourself” is encrypted as key: deceptivedeceptivedeceptive

plaintext: wearediscoveredsaveyourself

ciphertext: ZICVTWQNGRZGVTWAVZHCQYGLMGJ

Expressed numerically, we have the following result

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

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

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

Program:

```
# Poly_alphabetic cipher

alphabet = "abcdefghijklmnopqrstuvwxyz".upper()
mp = dict(zip(alphabet, range(len(alphabet))))
mp2 = dict(zip(range(len(alphabet)), alphabet))


def generateKey(plainText, keyword):
    key = ''
    for i in range(len(plainText)):
        key += keyword[i % len(keyword)]
    return key


def cipherText(plainText, key):
    cipher_text = ""
    for i in range(len(plainText)):
        shift = mp[key[i].upper()] - mp['A']
        newChar = mp2[(mp[plainText[i].upper()] + shift) % 26]
        cipher_text += newChar
    return cipher_text


def decrypt(cipher_text, key):
    plainText = ''
    for i in range(len(cipher_text)):
        shift = mp[key[i].upper()] - mp['A']
        newChar = mp2[(mp[cipher_text[i].upper()] - shift + 26) % 26]
        plainText += newChar
    return plainText


plainText = "wearediscoveredsaveyourself"
keyword = "deceptive"
key = generateKey(plainText, keyword)
cipher_text = cipherText(plainText, key)
print("Ciphertext :", cipher_text)
print("Decrypted Text :", decrypt(cipher_text, key))
```

```
Output:
Ciphertext : ZICVTWQNGRZGVTWAVZHCQYGLMGJ
Decrypted Text : WEAREDISCOVEREDSAVEYOURSELF
```

Experiment No: 07

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

Objectives:

- Algorithmic Understanding: Gain a deep understanding of the Vernam cipher algorithm, also known as the one-time pad, including how it uses a random or pseudorandom key stream to encrypt plaintext and decrypt ciphertext.
- Security Awareness: Increase awareness of encryption techniques and their role in securing sensitive information. Explore the security properties of the Vernam cipher, particularly its perfect secrecy when used with a truly random key stream of equal length as the plaintext.

Theory:

The Vernam cipher, also known as the one-time pad, is a symmetric encryption algorithm that uses the principle of the XOR (exclusive OR) operation to combine a random or pseudorandom key stream with plaintext to produce ciphertext, and vice versa for decryption.

Encryption: To encrypt a message, the Vernam cipher requires a key stream of the same length as the plaintext. Each character in the plaintext is combined with the corresponding character in the key stream using the XOR operation. The result is the ciphertext.

Decryption: To decrypt the ciphertext, the same key stream used for encryption is XORed with the ciphertext. This operation retrieves the original plaintext.

Key Generation: The security of the Vernam cipher relies on the randomness and secrecy of the key stream. Ideally, the key stream should be generated using a truly random process and should only be known to the sender and the intended recipient.

Perfect Secrecy: The Vernam cipher provides perfect secrecy when used with a truly random key stream of equal length as the plaintext. This means that even with unlimited computational resources, an attacker cannot gain any information about the plaintext from the ciphertext.

Key Reuse: One of the critical requirements of the Vernam cipher is that the key stream must never be reused for encrypting more than one message. Reusing the key stream compromises the security of the cipher and can lead to the recovery of the plaintext through statistical analysis or brute force attacks.

By implementing the Vernam cipher, learners can deepen their understanding of encryption concepts, improve their programming skills, and explore the practical applications of encryption techniques in cybersecurity and information security.

Program:

```
import random

alphabet = "abcdefghijklmnopqrstuvwxyz".upper()
mp = dict(zip(alphabet, range(len(alphabet))))
mp2 = dict(zip(range(len(alphabet)), alphabet))


def generate_key(length):
    key = ""
    for i in range(length):
        key += chr(random.randint(65, 90))  # ASCII codes for A-Z
    return key


def encrypt(plaintext, key):
    ciphertext = ""
    cipherCode = []
    for i in range(len(plaintext)):
        xor = mp[plaintext[i]] ^ mp[key[i]]
        cipherCode.append(xor)
        ciphertext += mp2[(mp['A'] + xor) % 26]
    return ciphertext, cipherCode


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


plaintext = "OAK"
plaintext = plaintext.upper()
key = generate_key(len(plaintext))
ciphertext, cipherCode = encrypt(plaintext, key)
print("Ciphertext:", ciphertext)
decryptedtext = decrypt(cipherCode, key)
print("Decrypted text:", decryptedtext)
```

Output:

```
Ciphertext: GLD
Decrypted text: OAK
```