Experiment NO: 01

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

Theory:

The caesar cipher technique is one of the earliest and simplest methods of enfryption technique. It's simply a type of substitution cipher each letter of a given text is replaced by a letter with a fixed number of positions down the alphabet.

Thus to cipher a given text we need an integer value, known as a shift which indicates the number of positions each letter of the text has been moved down.

The encryption can be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, A = 0, B = 1, ... Z = 25.
Encryption of a letter by a shift n can be described mathematically as.

For example, if the shift is 3, then the letter A would be replaced by the letter D, B would be replaced by E, C would become F and so on. The alphabet is wrapped around so that after z, it starts back at A.

Here, is an example of how to use the Caesar cipher to encryption the message "Information" with a shift of 3.

a	b	c	d	e	f	g	h	i	j	k
0	1	2	3	4	5	6	7	8	9	10

l	m	n	o	p	q	r	s	t	u	v
11	12	13	14	15	16	17	18	19	20	21

w	x	y	z.
22	23	24	25

Shift = 3

Plaintext = information

Ciphertext = lqirupdwlrq

To decrypt the message, we simply need to shift each letter back by the same number of positions.

In this case, we would shift each letter in lqirupdwlrq back by 3 positions to get the original message, "information".

Source code:

```
def caesar_encrypt(plaintext, shift):
    encrypt_text = ""
    for char in plaintext:
        if char.isalpha():
            if char.islower():
                encrypted_text += chr((ord
                (char) + shift - ord('a'))%26 +
                                    ord('a'))
            else:
                encrypted_text += chr((ord
                (char) + shift - ord('A'))%26+
                                    ord('A'))
        else:
            encrypted_text += char
    return encrypted_text
def caesar_decrypt(ciphertext, shift):
    decrypt_text = ""
```

```
        for char in ciphertext:
            if char.isalpha():
                if char.islower():
                    decrypted_text += chr((ord
                    (char)-shift-ord('a'))%26+
                                        ord('a'))
                else:
                    decrypted_text += chr((ord
                    (char)-shift-ord('A'))%26+
                                        ord('A'))
            else:
                decrypted_text += char
        return decrypted_text
plaintext = "information";
shift = 3
Ciphertext = caesar_encrypt(plaintext,
                            Shift)
Print(f"Caesar Cipher Encryption:
                        {ciphertext}")
Plaintext = caesar_decrypt(ciphertext,
                            shift)
print(f"Caesar Cipher Decryption:
                        {plaintext}")
```

Output:

Caesar Cipher Encryption : lqirupdwlrq

Caesar Cipher Decryption : information

Experiment No: 02

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

Theory:

Mono-Alphabetic 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 occurrence in that plaintext, 'A' will always get encrypted to 'D'. It uses a fixed key which consist of the 26 letters of a shuffled alphabet.

Plain text alphabet	A	B	C	D	E	F	G	H	I	J	K
Cipher text alphabet	M	U	A	O	R	V	X	B	T	C	W
plain text alphabet	L	M	N	O	P	Q	R	S	T	U	V
Cipher text alphabet	D	S	Z	G	E	H	Y	F	K	L	N
Plain text alphabet	W	X	Y	Z							
Cipher text alphabet	J	P	I	Q							

with the above key all 'A' letter in the plain text will be encoded to an 'M'.

This type of cipher is a form of symmetric encryption as the same key can be used to both encrypt and decrypt message.

For example:

~~Shift = 4~~

Plaintext = HELLO

Ciphertext = BRDDGr

Source code:

```
Class MonalphabeticCipher:
    def__init__(self):
        self.normal_chart = ['a', 'b', 'c', 'd',
                             'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
                             'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
                             'u', 'v', 'w', 'x', 'y', 'z']
        self.codd_chart = ['M', 'U', 'A', 'O', 'R',
                           'V', 'X', 'B', 'T', 'C', 'W', 'D', 'S',
                           'Z', 'G', 'E', 'H', 'Y', 'F', 'K', 'L',
                           'N', 'J', 'P', 'I', 'Q']
```

```
def string_encryption(self,s):
    encrypted_string = ""
    for char in s:
        for i in range(26):
            if char == self.normal_char
                                    [i]:
                encrypted_string +=
                    self.coded_char[i]
                break
            elif char<'a' or char>'z':
                encrypted_string+=char
                break
    return encrypted_string
def string_decryption(self,s):
    decrypted_string = ""
    for char in s:
        for i in range(26):
            if char == self.coded_char[i]
                decrypted_string += self.
                                normal_char[i]
                break
            elif char<'A' or char>'z':
                decrypted_string+=char
                break
    return decypted_string
```

```
def main():
    cipher = MonoalphabeticCipher()
    plain_text = "HELLO"
    print("Plain text:", plain_text)
    encrypted_message = cipher.string-
        encryption(plain_text.lower())
    print("Encrypted message:",
                        encrypted_message)
    decrypted-message = cipher.string-
                    decryption(encrypted_messag
    print("Decrypted message:",
                    decrypted_message)
main()
```

Source code:

Encryted message : BRDDG

Decryted message: HELLO

Experiment NO:03

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

Theory:

The playfair Cipher encryption technique can be used to encrypt or encode a message. It operates exactly like typical encryption. The only difference is that it encrypts a digraph, or a pair of two letters, instead of a single letter.

An initial 5x5 matrix key table is created The plaintext encryption key is made out of the matrix's alphabetic characters. Be mindful that we shouldn't repeat the letters. There are 26 alphabets however, there are only 25 spaces in which we can place a letter. The matrix will delete the extra letter because there is an excess of one letter(typically J). Despite this, J is there in the plaintext before being changed to I.

Encryption rules: playfair Cipher

- Split the plaintext first into digraphs (pairs of two letters). If the plaintext has an odd number of letters, append the letter Z at the end. Even the text in the basic form of the MANGO plaintext, for instance, consists of five letters. Consequently, it is not possible to make a digraph. As a result, we will change the plaintext to MANGOZ by adding the letter Z at the end.

- Divide the plaintext into digraphs after that (pair of two letters) For any letter that appears twice, place an x there (side by side). Think about the scenario where the digraph is CO MX MU NI CA TE and the plaintext is COMMUNICATE.

- To identify the cipher (encryption) text create a 5*5 matrix or key-table and fill it with alphabetic letters as follows:

1. The first row, from left to right, should include the letters for the supplied keyword. If there are any duplicate letters in a keyword, avoid using them. This means a letter will only be taken into account once.

The following three scenarios are conceivable:

i) If two letters occur together in a row as a digraph, Each letter in the digraph, in this case, should be replaced with the letter that is immediate to its right. If there is no letter to the right, the first letter in the same row is considered to be the right letter.

ii) If a pair of letters appears in the same column, in this case, replace each letter of the digraph with the letters immediately below them. If there is no letter below, wrap around to the top of the

same column.

iii) In the event that a digraph (a pair of letters) is present in both its corresponding row and column;

To generate a cipher for a pair of letters within a 33 matrix, a 33 subset is selected from a larger 5*5 matrix. Specifically two square corners within the matrix are chosen, which are positioned on opposite sides of a square.

Decryption rules: playfair cipher

Decryption ~~prod~~ procedures are used in reverse order as encryption procedures. When decrypting, the cipher is symmetric (move left along rows and up along columns). The same key and key table that are used to decode the message are accessible to the recipient of plain text.

Playfair Cipher Example:

Plaintext = Communication

Key = computer

First create a digraph from the plaintext.

CO MX MU NI CA TI ON

Then create a matrix.

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

From matrix, we can write,

CO = OM

MX = RM

MU = PC

NI = SG

CA = PT

TI = BD

ON = ML

Therefore, the plaintext COMMUNICATION is encrypted using OMRMPCSGPTBDML

Source code:

```
def playfair_cipher(plaintext, key, mode):
    alphabet = 'abcdefghiklmnopqrstuvwxyz'
    key = key.lower().replace(' ', '').
                        replace('j', 'i')
    key_square =
    for letter in key + alphabet:
        if letter not in key_square:
            key_square += letter

    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[i+1]

            i+=2
        replaceplaintext += plaintext[-1] i
                if i<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)]

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
    elseif 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]
result = ''
for digraph in digraphs:
    if mode == 'encrypt':
```

```
            result += encrypt(digraph)
        elif mode == 'decrypt':
            result += decrypt(digraph)
    return result
plaintext = 'COMMUNICATION'
key = 'COMPUTER'
ciphertext = playfair_cipher(plaintext, key,
                             'encrypt')
print('Cipher Text :', ciphertext)
decrypted_text = playfair_cipher(ciphertext,
                                 key, 'decrypt')
print('Decrypted Text :', decrypted_text)
```

Output:

```
Cipher Text : omrmpcsgptbdml
Decrypted Text : comxmunication
```

Experiment No: 04

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

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 x n matrix, against modulus 26. To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption.

Example:

A	B	C	D	E	F	G	H	I	J	K	L
0	1	2	3	4	5	6	7	8	9	10	11

M	N	O	P	Q	R	S	T	U	V	W	X
12	13	14	15	16	17	18	19	20	21	22	23

Y	Z
24	25

Encryption:

Plain text = HELP

$$\text{Key, } K = \begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix}$$

$$P_1 = HE = \begin{bmatrix} H \\ E \end{bmatrix} = \begin{bmatrix} 7 \\ 4 \end{bmatrix}$$

$$C_1 = KP_1 \bmod 26$$

$$= \begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix} \times \begin{bmatrix} 7 \\ 4 \end{bmatrix} \bmod 26$$

$$= \begin{bmatrix} 21 + 12 \\ 14 + 20 \end{bmatrix} \bmod 26$$

$$= \begin{bmatrix} 33 \\ 34 \end{bmatrix} \bmod 26$$

$$= \begin{bmatrix} 7 \\ 8 \end{bmatrix}$$

$$= \begin{bmatrix} H \\ I \end{bmatrix}$$

$$= HI$$

$$P_2 = LP = \begin{bmatrix} L \\ P \end{bmatrix}$$

$$= \begin{bmatrix} 11 \\ 15 \end{bmatrix}$$

$$C_2 = KP_2 \bmod 26$$

$$= \begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix} \times \begin{bmatrix} 11 \\ 15 \end{bmatrix} \bmod 26$$

$$= \begin{bmatrix} 33 + 45 \\ 22 + 75 \end{bmatrix} \bmod 26$$

$$= \begin{bmatrix} 78 \\ 97 \end{bmatrix} \bmod 26$$

$$= \begin{bmatrix} 0 \\ 19 \end{bmatrix}$$

$$= \begin{bmatrix} A \\ T \end{bmatrix}$$

$$= AT$$

Encrypted data HIAT

Decryption:

Cipher text = HIAT

$$K = \begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix}$$

$$P = K^{-1} C \bmod 26$$

$$K^{-1} = \frac{1}{|K|} \text{adj} K$$

$$= \frac{1}{15-6} \begin{bmatrix} 5 & -3 \\ -2 & 3 \end{bmatrix}$$

$$= \frac{1}{9} \begin{bmatrix} 5 & 23 \\ 24 & 3 \end{bmatrix}$$

$$= 3 \begin{bmatrix} 5 & 23 \\ 24 & 3 \end{bmatrix}$$

$$= \begin{bmatrix} 15 & 69 \\ 72 & 9 \end{bmatrix}$$

$$= \begin{bmatrix} 15 & 69 \\ 20 & 9 \end{bmatrix}$$

$$\therefore K^{-1} = \begin{bmatrix} 15 & 17 \\ 20 & 9 \end{bmatrix}$$

$P_1 = K^{-1} C_1 \bmod 26$

$$= \begin{bmatrix} 15 & 17 \\ 20 & 9 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \end{bmatrix} \bmod 26$$

$$= \begin{bmatrix} 241 \\ 212 \end{bmatrix} \bmod 26$$

$$= \begin{bmatrix} 7 \\ 4 \end{bmatrix}$$

$$= \begin{bmatrix} H \\ E \end{bmatrix} = HE$$

$P_2 = K^{-1} C_2 \bmod 26$

$$= \begin{bmatrix} 15 & 17 \\ 20 & 9 \end{bmatrix} \times \begin{bmatrix} 0 \\ 19 \end{bmatrix} \bmod 26$$

$$= \begin{bmatrix} 323 \\ 171 \end{bmatrix} \bmod 26$$

$$= \begin{bmatrix} 11 \\ 15 \end{bmatrix}$$

$$= \begin{bmatrix} L \\ P \end{bmatrix}$$

$= LP$

original Text = HELP

Source code:

```
import numpy as np
from egcd import egcd
alphabet = "abcdefghijklmnopqrstuvwxyz"
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)))
    det_inv = egcd(det, modulus)[1] % modulus
    matrix_modulus_inv =(
            det_inv * np.round(det * np.linalg.
            inv(matrix)).astype(int) % modulus
    )
    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]
            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 = "HELP"
message = message.lower()
k = np.matrix([[3,3],[2,5]])
```

```
kinv = matrix_mod_inv(k ~~det~~ 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: help
Encrypted message: hiat
Decrypted message: help

Experiment No: 05

Experiment Name: Write a program to implement encryption and decryption using poly-alphabetic cipher.

Theory:

A polyalphabetic cipher substitution, using multiple substitution alphabets The vigenere cipher is probably by the best known example of a poly-alphabetic cipher, through it is a Simplied special case.

We can express the vigenere cipher in the following manner. Assume a sequence of plaintext letters $P = P_0, P_1, P_2 ---- P_{n-1}$ and a key Consisting of the sequence of letters $K = K_0, K_1, K_2 --- K_{m-1}$.

where, typically $m < n$. The sequence of Ciphertext letters, $C = C_0, C_1, C_2 --- C_{n-1}$ is calculated as follows,

$C = C_0, C_1, C_2, C_3 ---- C_{n-1}$

$= E(K, P)$

$= E[(K_0, K_1, K_2 \cdots K_{m-1}), (P_0, P_1, P_2 \cdots P_{n-1})]$

$= (P_0 + K_0) \bmod 26, (P_1 + K_1) \bmod 26 \cdots$

$(P_{n-1} + K_{m-1}) \bmod 26$

Encryption:

$E_i = (P_i + K_i) \bmod 26$

Decryption:

$D_i = (E_i - K_i) \bmod 26$

Example:

Plaintext = Hello

key = ICE

= ICEIC

$E_1 = (H + I) \bmod 26$

$= (7 + 8) \bmod 26$

$= 15 \bmod 26$

$= P$

$E_2 = (E + C) \bmod 26$

$= (4 + 2) \bmod 26$

$= 6 \bmod 26$

$= G$

$E_3 = (L + E) \bmod 26$

$= (11 + 4) \bmod 26$

$= 15 \bmod 26$

$= P$

$E_4 = (L + I) \bmod 26$

$= (11 + 8) \bmod 26$

$= 19 \bmod 26$

$= T$

$E_5 = (O + C) \bmod 26$

$= (14 + 2) \bmod 26$

$= 16 \bmod 26$

$= Q$

Encrypted data = PGPTQ

Decryption:

$D_1 = (P - I) \bmod 26$

$= (15 - 8) \bmod 26$

$= 7 \bmod 26$

$= H$

$D_2 = (G - C) \bmod 26$

$= (6 - 2) \bmod 26$

$= 4 \bmod 26$

$= E$

$D_3 = (P - E) \bmod 26$

$= (15 - 4) \bmod 26$

$= 11 \bmod 26$

$= L$

$D_4 = (T - I) \bmod 26$

$= (19 - 8) \bmod 26$

$= 11 \bmod 26$

$= L$

$D_5 = (Q - C) \bmod 26$

$= (16 - 2) \bmod 26$

$= 14 \bmod 26$

$= O$

Original Text = HELLO

Source code:

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

def cipherText(string, key):
    cipher_text = []
    string = string.upper()
    for i in range(len(string)):
        x = (ord(string[i]) + ord(key[i])) %
        x += ord('A')
        cipher_text.append(chr(x))
    return("".join(cipher_text))

def originalText(cipher_text, key):
    orig_text = []
    for i in range(len(cipher_text)):
        x = (ord(cipher_text[i]) -
             ord(key[i]) + 26) % 26
```

```
            x += ord('A')
            orig_text.append(chr(x))
    return(" ".join(orig_text))

Plain_text = "HELLO"
keyword = "ICE"
keyword = keyword.upper()
key = generatekey(Plain_text, keyword)
Print(key)
Cipher_text = CipherText(plain_text, key)
print(Cipher_text)
Print("Ciphertext :", Cipher_text)
Print("Original/Decryted Text :",
        originalText(Cipher_text, key))
```

Output:

Plain text : HELLO

key: ICEIC

Ciphertext : PGPTQ

Original/ Decrypted Text : HELLO

Experiment NO: 06

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

Theory:

Vernam cipher is a method of encrypting alphabetic text. It is one of the substitution techniques for coverting plain text into cipher text. In this mechanism, we assign a number to each of character of the plain-text, like (a = 0, b = 1, c = 2 - - - Z = 25).

Method to take key: In the Vernam Cipher algorithm, we take a key to encrypt the plain text whose length should be equal to the length of the plain text.

Encryption Algorithm

- Assign a number to each character of the plaintext and the key according to alphabetical order.
- Bitwise XOR both the number (corresponding plaintext character and key character number).

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

Example:
Encryption:
Plaintext = O A K

Key = S O N

O = 14 = 0 1 1 1 0

S = 18 = 1 0 0 1 0

Bitwise XOR Result = 1 1 1 0 0 = 28

28 - 26 = 2 = C

A = 0 = 0 0 0 0

O = 14 = 1 1 1 0

Bitwise XOR Result : 1 1 1 0 = 14 = O

K = 10 = 1 0 1 0

N = 13 = 1 1 0 1

Bitwise XOR Result : 0 1 1 0 1 = 7 = H

Ciphertext = C O H

Decryption:

Ciphertext = C O H

key = S O N

~~C~~ = 28 = ~~10~~ ~~0 0 1 0~~ 11100

S = 18 = ~~1 0 0 1 0~~ 10010

Bitwise XOR Result = ~~10000~~ = 01110 = 14 = O

O = 14 = 01110

O = 14 = 1110

Bitwise XOR Result = 0000 = A

H = 7 = 0111

N = 13 = 1101

Bitwise XOR Result = 1010 = 10 = K

Decrypted Text = OAK

Source Code:

```
import random
def generate_key(length):
    key = ""
    for i in range(length):
        key += chr(random.randit(65,90))
    return key

def encrypt(plaintext, key):
    Ciphertext = ""
    CipherCode = []
    for i in range(len(plaintext)):
        x = ord(plaintext[i])^ord(key[i])
        CipherCode.append(x)
        Ciphertext += chr(x%26+65)
    return Ciphertext, CipherCode

def decrypt(CipherCode, key):
    plaintext = ""
    for i in range(len(CipherCode)):
        x =(CipherCode[i]^ ord(key[i]))
        plaintext += chr(x)
    return plaintext
```

```
plaintext = "OAK"
key = generate_key(len(plaintext))
Ciphertext, CipherCode = encrypt(plaintext, key)
Print("Ciphertext:", Ciphertext)
decryptedtext = decrypt(CipherCode, key)
Print("Decrypted text:", decryptedtext)
```

Output:

Ciphertext : YDM

Decrypted text : OAK

Experiment No : 07

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

Theory:

A brute-force attack is a trial and error method used by application programs to decode login information and encryption keys to use them to gain unauthorized access to systems. Using brute force is an exhaustive effort rather than employing intellectual strategies.

A simple brute force attack commonly uses automated tools to guess all possible passwords until the correct input is ~~define~~ identified. This is odd but still effective attack method for cracking common passwords. Brute force can break weak passwords in a matter of seconds.

Source code:

```
def brute_force_decrypt(ciphertext):
    for shift in range(26):
        decrypted_text = caesar_decrypt
                         (ciphertext, shift)
        print(f"shift {shift}: {decrypted_text}")

def brute_force_encrypt(plainText):
    for shift in range(26):
        encrypted_text = caesar_encrypt
                         (plainText, shift)
        print(f" shift {shift}: {encrypted_text}")

def caesar_encrypt(plainText, shift):
    encrypted_text = ""
    for char in plainText:
        if char.isalpha():
            if char.islower():
                encrypted_text += chr((ord(cha
                +shift - ord('a'))%26+ ord('a'))
            else:
                encrypted_text += chr((ord(char)
                + shift - ord('A'))%26 + ord('A'))
        else:
            encrypted_text += char
    return encrypted_text
```

```
def caesar_decrypt(ciphertext, shift):
    decrypted_text = ""
    for char in ciphertext:
        if char.isalpha():
            if char.islower():
                decrypted_text += chr((ord(cha
                - shift - ord('a'))% 26 + ord('a'))
            else:
                decrypted_text += chr((ord(char
                - shift - ord('A'))% 26 + ord('A'))
        else:
            decrypted_text += char
    return decrypted_text

plaintext = 'hello'
print('Brute Force Encryption for Caesar
                                    Cipher:')
brute_force_encrypt(plaintext)

ciphertext = "ikmmp"
print("\nBrute Force Decryption for
                                Caesar Cipher:')
brute_force_decrypt(ciphertext)
```

~~Source code:~~ Output:

Brute Force Encryption for Caesar Cipher

Shift 0: hello

Shift 1: ifmmp

Shift 2: jgnnq

Shift 3: Khoor

Shift 4: lipps

Shift 5: mjqqt

Shift 6: nkrru

Shift 7: olssv

Shift 8: pmttw

Shift 9: ~~quuu~~ qnuux

Shift 10: rovvy

Shift 11: spwwz

Shift 12: tqxxa

Shift 13: uryyb

Shift 14: vszzc

Shift 15: wtaad

Shift 16: xubbe

Shift 17: yvccf

Shift 18: zwddg

Shift 19: axeeh

Shift 20: byffi

Shift 21: czggj
Shift 22: dahhk
Shift 23: ebiil
Shift 24: fcjjm
Shift 25: gdkkn

Brute Force Decryption for Caesar Cipher:
Shift 0: ifmmp
Shift 1: hello
Shift 2: gdkkn
Shift 3: fcjjm
Shift 4: ebiil
Shift 5: dahhk
Shift 6: czggj
Shift 7: bygfi
Shift 8: axeeh
Shift 9: zwddg
Shift 10: yvccf
Shift 11: xubbe
Shift 12: wtaad
Shift 13: vszzc
Shift 14: uryyb
Shift 15: tqxxa
Shift 16: spwwz

Shift 17 : rovvy

shift 18 : qnuux

shift 19 : pmttw

shift 20 : olssv

Shift 21 : nkrrrcu

shift 22 : mjqqt

Shift 23 : lipps

Shift 24 : khoor

Shift 25 : jgnnq