Rashtriya Raksha University

School of Information Technology, Artificial Intelligence & Cyber Security (SITAICS)

At- Lavad, Dahegam, Gandhinagar, Gujarat-382305

Practical File

(Introduction to Cryptography)

Name:	Sarthak Sanay
Enrollment No:	230031101611051
Subject Name:	Introduction to Cryptography
Subject Code:	G4A19ITC
Program:	B.Tech CSE (with specialization in Cyber Security)
Year:	2nd year (Semester-IV)

This is certifying that Mr. Sarthak Sanay has satisfactorily completed all experiments in the practical work prescribed by SITAICS in the ITC laboratory.

Dr. Ashish Revar

SUBJECT INCHARGE

PRACTICAL - 1

AIM: TO IMPLEMENT CAESAR CIPHER

BRIEF :-

The Caesar cipher, named after the Roman general Julius Caesar, is one of the oldest and simplest encryption techniques ever devised.

At its core, the cipher replaces each letter in a message (the plaintext) with the letter a fixed number of positions down the alphabet. This fixed number is known as the "key." For example, with a key of 3, A becomes D, B becomes E, C becomes F, and so on; after Z it wraps around back to A. To encrypt, you add the key to each letter's position; to decrypt, you subtract the key, using modulo 26 arithmetic to handle the wrap-around.

Non-letter characters - such as spaces, digits, and punctuation—are typically left unchanged, which makes the cipher easy to implement in code or by hand. Because the only secret is the key (an integer between 1 and 25), there are only 25 possible non-trivial shifts. An attacker can therefore mount a brute-force attack—trying all possible keys - or apply simple frequency analysis to recover the original message.

Despite its historical importance and pedagogical value in introducing concepts like modular arithmetic and substitution ciphers, the Caesar cipher offers no real security by modern standards. Its ease of breaking makes it unsuitable for protecting sensitive data today, but it remains a popular example in cryptography tutorials and puzzles.

ALGORITHM / PSEUDOCODE :-

```
repeat
  print menu
  read ch
  if ch == 1 then
    read plain_text
    read key
    cipher_text = ""
    for each c in plain_text do
      if isUpper(c) then
        cipher_text += ( (c-'A'+key) mod 26 ) + 'A'
      else if isLower(c) then
        cipher_text += ( (c-'a'+key) mod 26 ) + 'a'
      else
        cipher_text += c
    end for
    print cipher_text

  else if ch == 2 then
    read cipher_text
    read key
    plain_text = ""
    for each c in cipher_text do
      if isUpper(c) then
        plain_text += ( (c-'A'-key+26) mod 26 ) + 'A'
      else if isLower(c) then
        plain_text += ( (c-'a'-key+26) mod 26 ) + 'a'
      else
        plain_text += c
    end for
    print plain_text

  else if ch == 0 then
    exit loop
  else
    print "Invalid choice"
  end if
until ch == 0
```

CODE :-

```
print("\nCaesar Cipher Encryption & Decryption Tool:-")
ch = 1

while (ch!=0):
    ch = int(input("\nEnter 1 to Encrypt. \nEnter 2 to Decrypt.
\nEnter 0 to Exit. \nEnter choice: "))

    match ch:

        case 1:
            print("\nEncrypting Caesar Cipher!\n")
            plain_text = str(input("Enter plain text: "))
            key = int(input("Enter key: "))
            cipher_text = ""

            for i in range(0, len(plain_text)):
                char = plain_text[i]

                if char == chr(32):
                    cipher_text += char
                    continue

                elif (char.isupper()):
                    cipher_text += chr((ord(char) + key-65) % 26 + 65)

                elif (char.islower()):
                    cipher_text += chr((ord(char) + key-97) % 26 + 97)

                else:
                    cipher_text += char

            print("Plain Text:  ", plain_text)
            print("Cipher Text: ", cipher_text, "\n")


        case 2:
            print("\nDecrypting Caesar Cipher!\n")
            cipher_text = str(input("Enter cipher text: "))
            key = int(input("Enter key: "))
            plain_text = ""

            for i in range(0, len(cipher_text)):
                char = cipher_text[i]

                if char == chr(32):
                    plain_text += char
                    continue
```

```
                elif (char.isupper()):
                    plain_text += chr((ord(char) - key-65) % 26 + 65)

                elif (char.islower()):
                    plain_text += chr((ord(char) - key-97) % 26 + 97)

                else:
                    plain_text += char

            print("Cipher Text: ", cipher_text)
            print("Plain Text:  ", plain_text, "\n")


        case 0:
            print("\nProgram exited successfully!")

        case _:
            print("\nEnter correct choice!\n")
```

<u>OUTPUT</u> :-

```
Caesar Cipher Encryption & Decryption Tool:

Enter 1 to Encrypt.
Enter 2 to Decrypt.
Enter 0 to Exit.
Enter choice: 1

Encrypting Caesar Cipher!

Enter plain text: Hello, My name is Sarthak
Enter key: 17
Plain Text:  Hello, My name is Sarthak
Cipher Text: Yvccf, Dp erdv zj Jrikyrb

Enter 1 to Encrypt.
Enter 2 to Decrypt.
Enter 0 to Exit.
Enter choice: 2

Decrypting Caesar Cipher!

Enter cipher text: Yvccf, Dp erdv zj Jrikyrb
Enter key: 17
Cipher Text:  Yvccf, Dp erdv zj Jrikyrb
Plain Text:   Hello, My name is Sarthak

Enter 1 to Encrypt.
Enter 2 to Decrypt.
Enter 0 to Exit.
Enter choice: 0

Program exited successfully!
```

PRACTICAL - 2

AIM: TO IMPLEMENT PLAYFAIR CIPHER

BRIEF :-

The Playfair cipher, invented by Charles Wheatstone and popularized by Lord Playfair in the mid-19th century, is an early digraph substitution cipher. Instead of substituting single letters, it operates on pairs of letters (“digraphs”), using a 5×5 key matrix (merging I/J) to determine substitutions. By encrypting in pairs and using positional rules (same row, same column, or rectangle corners), it obscures letter frequencies better than simple monoalphabetic ciphers. While it was once used by militaries for its relative ease of use and improved security, it is now considered insecure by modern standards, but remains a valuable pedagogical example of polygraphic substitution.

Because it processes two letters at once, the Playfair cipher resists simple frequency analysis of individual letters and introduces complexity with filler characters (commonly “X”) when duplicates appear or an odd-length digraph remains. Its manual-friendly grid makes it suitable for field use in pre-computer eras, yet its fixed 5x5 matrix and predictable rules leave it vulnerable to modern cryptanalysis.

ALGORITHM / PSEUDOCODE :-

```
1. Build 5×5 Key Matrix:
   - Uppercase keyword, replace J→I, remove duplicates.
   - Fill matrix left→right, top→bottom with keyword letters.
   - Fill remaining cells with A to Z (skip J and used letters).

2. Prepare Plaintext:
   - Uppercase, replace J→I, remove spaces.
   - Insert 'X' between repeated letters in each pair.
   - If the length is odd, append 'X'.

3. Encrypt Digraphs:
   For each pair (A, B) in prepared text:
     - Find (r1,c1) for A, (r2,c2) for B in the matrix.
     - If same row: replace with letters to their right (wrap around).
     - Else if same column: replace with letters below (wrap around).
     - Else: replace with letters at the opposite corners of the rectangle.

4. Decrypt Digraphs:
   For each pair (C, D) in cipher text:
     - Find positions as above.
     - If the same row: replace with letters to their left (wrap around).
     - Else if same column: replace with letters above (wrap around).
     - Else: swap columns as in encryption.

5. Menu Loop:
   - 1: Encrypt → input plaintext & keyword → show matrix → output cipher.
   - 2: Decrypt → input ciphertext & keyword → show matrix → output plaintext.
   - 0: Exit.
```

CODE :-

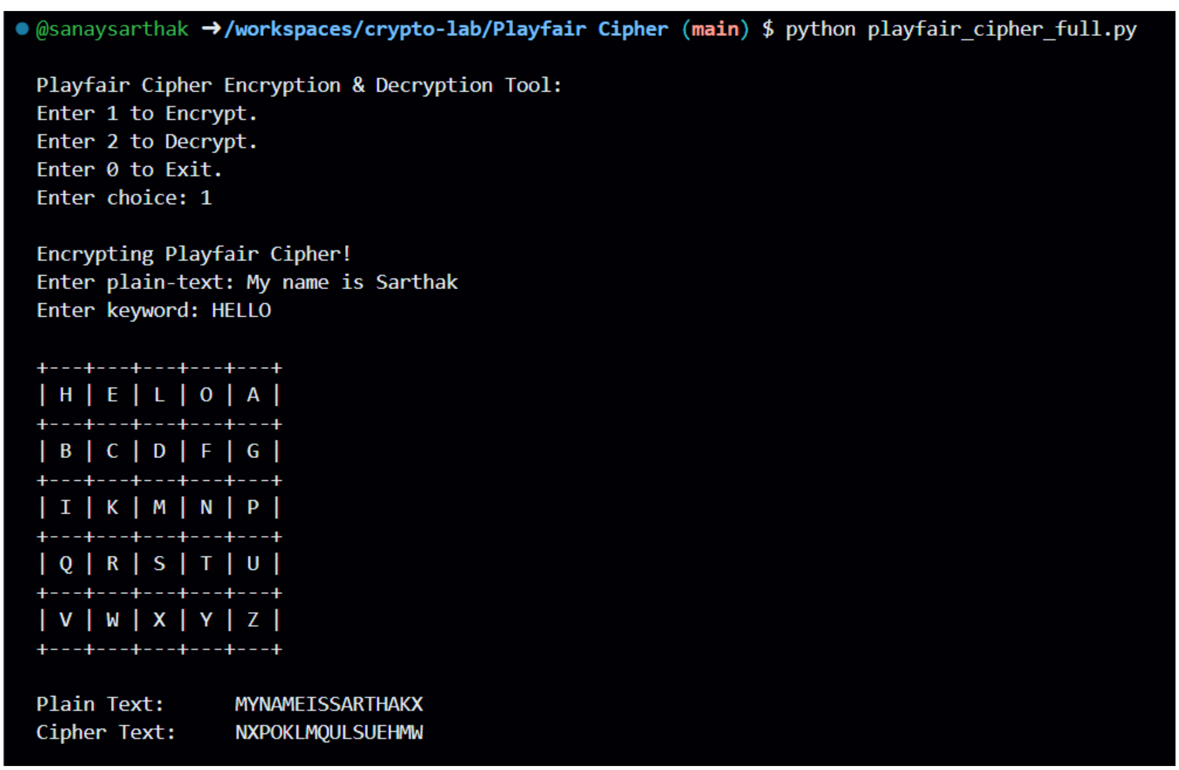
```
def create_matrix(keyword):

    keyword = keyword.upper()
    keyword = keyword.replace('J', 'I')

    matrix = [[], [], [], [], []]
    used_char = []
    alphabets = []  # ['A', 'B', 'C', ... , 'Z']
    for i in range(65, 91):
        if chr(i) == 'J':
            continue
        alphabets.append(chr(i))

    i, j = 0, 0
    for char in keyword:
        if j == 5:
            j = 0
            i += 1
        if char in used_char:
            continue
        if char == 'J':
            matrix[i].insert(j, char)
            used_char.append('I')
            j += 1
        else:
            matrix[i].insert(j, char)
            used_char.append(char)
            j += 1

    for char in alphabets:
        if j == 5:
            j = 0
            i += 1
        if char in used_char:
            continue
        if char == 'J':
            matrix[i].insert(j, char)
            used_char.append('I')
            j += 1
        else:
            matrix[i].insert(j, char)
            used_char.append(char)
            j += 1

    return matrix
```

```python
def display_matrix(matrix):
    print("\n+---+---+---+---+---+")
    for row in matrix:
        row_string = "|"
        for num in row:
            row_string += f" {num} " + "|"
        print(row_string)
        print("+---+---+---+---+---+")


def prepare_text_encrypt(plain_text):
    plain_text = plain_text.upper().replace('J', 'I').replace(" ", "")

    i = 0
    while i < len(plain_text):
        substring = plain_text[i:i+2]
        if len(substring) == 2 and substring[0] == substring[1]:
            plain_text = plain_text[:i+1] + "X" + plain_text[i+1:]
        i += 2
        if i+1 == len(plain_text):
            plain_text += "X"
            break
    return plain_text


def encrypt_playfair(plain_text, keyword):
    matrix = create_matrix(keyword)
    display_matrix(matrix)

    plain_text = prepare_text_encrypt(plain_text)

    i = 0
    cipher_text = ""

    while i < len(plain_text):
        char1 = plain_text[i]
        char2 = plain_text[i+1]

        row1 = col1 = row2 = col2 = -1
        for r in range(5):
            for c in range(5):
                if matrix[r][c] == char1:
                    row1, col1 = r, c
                if matrix[r][c] == char2:
                    row2, col2 = r, c

        if row1 == row2:
            cipher_text += matrix[row1][(col1 + 1) % 5]
            cipher_text += matrix[row2][(col2 + 1) % 5]
        elif col1 == col2:
            cipher_text += matrix[(row1 + 1) % 5][col1]
            cipher_text += matrix[(row2 + 1) % 5][col2]
```

```
        else:
            cipher_text += matrix[row1][col2]
            cipher_text += matrix[row2][col1]

        i += 2

    return plain_text, cipher_text


def decrypt_playfair(cipher_text, keyword):
    matrix = create_matrix(keyword)
    display_matrix(matrix)

    cipher_text = cipher_text.upper().replace(" ", "")

    i = 0
    plain_text = ""

    while i < len(cipher_text):
        char1 = cipher_text[i]
        char2 = cipher_text[i+1]

        row1 = col1 = row2 = col2 = -1
        for r in range(5):
            for c in range(5):
                if matrix[r][c] == char1:
                    row1, col1 = r, c
                if matrix[r][c] == char2:
                    row2, col2 = r, c

        if row1 == row2:
            plain_text += matrix[row1][(col1 - 1) % 5]
            plain_text += matrix[row2][(col2 - 1) % 5]
        elif col1 == col2:
            plain_text += matrix[(row1 - 1) % 5][col1]
            plain_text += matrix[(row2 - 1) % 5][col2]
        else:
            plain_text += matrix[row1][col2]
            plain_text += matrix[row2][col1]

        i += 2

    return plain_text


choice = None
while choice != 0:
    print("\nPlayfair Cipher Encryption & Decryption Tool:")
    print("Enter 1 to Encrypt.")
    print("Enter 2 to Decrypt.")
    print("Enter 0 to Exit.")
```

```
try:
    choice = int(input("Enter choice: "))
except ValueError:
    print("Invalid input. Please enter 1, 2, or 0.")
    continue

if choice == 1:
    print("\nEncrypting Playfair Cipher!")
    pt = input("Enter plain-text: ")
    key = input("Enter keyword: ")
    prepared, ct = encrypt_playfair(pt, key)
    print("\nPlain Text: \t", prepared)
    print("Cipher Text:\t", ct)
elif choice == 2:
    print("\nDecrypting Playfair Cipher!")
    ct_input = input("Enter cipher-text: ")
    key = input("Enter keyword: ")
    pt_out = decrypt_playfair(ct_input, key)
    print("\nCipher Text:\t", ct_input.replace(' ', '').upper())
    print("Plain Text: \t", pt_out)
elif choice == 0:
    print("\nProgram exited successfully!")
else:
    print("\nEnter correct choice!")
```

<u>OUTPUT</u> :-



```
@sanaysarthak ➜ /workspaces/crypto-lab/Playfair Cipher (main) $ python playfair_cipher_full.py

Playfair Cipher Encryption & Decryption Tool:
Enter 1 to Encrypt.
Enter 2 to Decrypt.
Enter 0 to Exit.
Enter choice: 1

Encrypting Playfair Cipher!
Enter plain-text: My name is Sarthak
Enter keyword: HELLO

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

Plain Text:      MYNAMEISSARTHAKX
Cipher Text:     NXPOKLMQULSUEHMW
```

```
Playfair Cipher Encryption & Decryption Tool:
Enter 1 to Encrypt.
Enter 2 to Decrypt.
Enter 0 to Exit.
Enter choice: 2

Decrypting Playfair Cipher!
Enter cipher-text: NXPOKLMQULSUEHMW
Enter keyword: HELLO

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

Cipher Text:     NXPOKLMQULSUEHMW
Plain Text:      MYNAMEISSARTHAKX

Playfair Cipher Encryption & Decryption Tool:
Enter 1 to Encrypt.
Enter 2 to Decrypt.
Enter 0 to Exit.
Enter choice: 0

Program exited successfully!
```

PRACTICAL - 5

AIM: TO IMPLEMENT COLUMNAR CIPHER

BRIEF :-

The Columnar Transposition Cipher is a way to hide a message by writing its letters into rows beneath a chosen keyword. You fill the grid row by row, adding extra filler characters at the end if needed so the last row is full. To make the ciphertext, you number the keyword's letters by their order in the alphabet and then read the letters down each numbered column in turn.

To decrypt, you rebuild the same grid shape using the keyword, fill in each column from the ciphertext in the right order, and then read the message off row by row, removing any filler. This simple reversal shows how the same steps in opposite order recover the original text. It's a straightforward manual method that anyone can work out with pen and paper.

Although the columnar cipher mixes up letters and hides obvious word patterns, it is still easy for modern programs to break. Attackers can try different key lengths or look for common words running down columns. Even so, it remains a useful teaching tool, demonstrating how moving letters (a transposition) can form the building blocks of more complex encryption systems.

ALGORITHM / PSEUDOCODE :-

```
repeat
    print menu
    read ch

    if ch == 1 then
        read plain_text
        read key

        text ← removeSpaces(plain_text)
        cols ← length(key)

        matrix ← EMPTY LIST
        for i from 0 to length(text)-1 step cols do
            append text[i..i+cols-1] as list to matrix
        end for

        print "The Matrix is as follows :-"
        print "Key:    " + join(key, " ")
        print "        " + repeat("-", 2 * cols)
        for each row in matrix do
            print "        " + join(row, " ")
        end for

        order ← getOrder(key)

        cipher_text ← ""
        for num from 1 to cols do
            colIndex ← indexOf(order, num)
            for each row in matrix do
                if colIndex < length(row) then
                    cipher_text ← cipher_text + row[colIndex]
                end if
            end for
        end for

        print "Encrypted Text: " + cipher_text

    else if ch == 0 then
        exit loop

    else
        print "Invalid choice"
    end if

until ch == 0
```

CODE :-

```
# Program in Python to implement Columnar Cipher

def encrypt(text, key):
    text = text.replace(" ", "")
    cols = len(key)
    matrix = build_matrix(text, cols)
    print_matrix(matrix, key)

    order = get_order(key)
    cipher = ""

    for num in range(1, cols + 1):
        col = order.index(num)
        for row in matrix:
            if col < len(row):
                cipher += row[col]
    return cipher

def build_matrix(text, width):
    matrix = []
    for i in range(0, len(text), width):
        matrix.append(list(text[i:i+width]))
    return matrix

def print_matrix(matrix, key):
    print("\nThe Matrix is as follows :-\n")
    print("Key:   ", " ".join(key))
    print("        " + ('-' * (len(key)*2)))
    for row in matrix:
        print("       ", " ".join(row))

def get_order(key):
    order = []
    for i, ch in enumerate(key):
        count = 1
        for j in range(i):
            if key[j] <= ch:
                count += 1
            else:
                order[j] += 1
        order.append(count)
    return order

text = input("Enter the plaintext: ")
key = input("Enter the keyword: ")
result = encrypt(text, key)
print("The Encrypted Text:", result)
```

OUTPUT :-

```
@sanaysarthak →/workspaces/crypto-lab/practicals (main) $ python columnar-cipher.py
Enter the plaintext: SARTHAKSANAY
Enter the keyword: AUDI

The Matrix is as follows :-

Key:    A U D I
        --------
        S A R T
        H A K S
        A N A Y
The Encrypted Text: SHARKATSYAAN
```

PRACTICAL - 6

AIM: TO IMPLEMENT RAILFENCE TRANSPOSITION CIPHER

BRIEF :-

The Rail Fence cipher is a simple transposition cipher that writes plaintext letters in a zig-zag pattern across a fixed number of “rails” (rows) and then reads them off row by row to form the ciphertext. By choosing a numeric depth (the key), the sender and receiver agree on how many rails to use; the plaintext is written down and up through the rails in sequence, creating a diagonal stripe. Once all letters are placed, the rows are concatenated in order to produce the encrypted message.

Decryption reverses this process by reconstructing the zig-zag matrix - marking the positions to be filled, and then refilling each rail with the appropriate ciphertext segment before reading the letters in their original zig-zag order. Though trivial to implement and useful for teaching the basics of transposition, the Rail Fence cipher offers minimal security by modern standards and can be broken easily by analyzing rail patterns.

ALGORITHM / PSEUDOCODE :-

```
repeat
print menu
read ch

if ch == 1 then
    read plain_text
    read depth
    text = remove spaces from plain_text
    // build rails
    rails = array of depth empty strings
    row = 0; dir = 1
    for each c in text do
        rails[row] += c
        if row == 0 then
            dir = 1
        else if row == depth - 1 then dir = -1
            row += dir
    end for
    cipher_text = join rails with spaces
    print cipher_text

else if ch == 2 then
    read cipher_text
    read depth
    rails_str = split cipher_text by spaces
    // reconstruct zig-zag
    length = total characters in rails_str
    mark zigzag positions in matrix[depth][length]
    fill matrix row by row from rails_str
    plain_text = read matrix in zigzag order
    print plain_text

else if ch == 0 then
    exit loop

else
    print "Invalid choice"
end if
until ch == 0
```

CODE :-

```
print("\nRail Fence Cipher Encryption & Decryption Tool:-")
ch = 1

while ch != 0:
    ch = int(input(
        "\nEnter 1 to Encrypt. \n"
        "Enter 2 to Decrypt. \n"
        "Enter 0 to Exit. \n"
        "Enter choice: "
    ))

    match ch:
        case 1:
            print("\nEncrypting Rail Fence Cipher!\n")
            plain_text = input("Enter plain text: ")
            depth = int(input("Enter depth: "))

            # remove all spaces
            text = plain_text.replace(" ", "")
            rails = [[] for _ in range(depth)]
            row, direction = 0, 1

            for char in text:
                rails[row].append(char)
                if row == 0:
                    direction = 1
                elif row == depth - 1:
                    direction = -1
                row += direction

            cipher_text = ""
            for rail in rails:
                for char in rail:
                    cipher_text += char
                cipher_text += " "

            print("Plain Text:  ", plain_text)
            print("Cipher Text: ", cipher_text, "\n")

        case 2:
            print("\nDecrypting Rail Fence Cipher!\n")
            cipher_text = input("Enter cipher text: ")
            depth = int(input("Enter depth: "))

            rails_str = cipher_text.split()
            length = sum(len(r) for r in rails_str)

            # build rail matrix
```

```
            matrix = [[''] * length for _ in range(depth)]
            row, direction = 0, 1
            for col in range(length):
                matrix[row][col] = '*'
                if row == 0:
                    direction = 1
                elif row == depth - 1:
                    direction = -1
                row += direction

            # fill characters
            index = 0
            for i in range(depth):
                for col in range(length):
                    if matrix[i][col] == '*' and index <
len(rails_str[i]):
                        matrix[i][col] = rails_str[i][index]
                        index += 1
                index = 0

            plain_text = ""
            row, direction = 0, 1
            for col in range(length):
                plain_text += matrix[row][col]
                if row == 0:
                    direction = 1
                elif row == depth - 1:
                    direction = -1
                row += direction

            print("Cipher Text: ", cipher_text)
            print("Plain Text:  ", plain_text, "\n")

        case 0:
            print("\nProgram exited successfully!")

        case _:
            print("\nEnter correct choice!\n")
```

OUTPUT :-

```
@sanaysarthak ➜ /workspaces/crypto-lab/Rail Fence Algorithm (main) $ python rail-fence-cipher-full.py

Rail Fence Cipher Encryption & Decryption Tool:-

Enter 1 to Encrypt.
Enter 2 to Decrypt.
Enter 0 to Exit.
Enter choice: 1

Encrypting Rail Fence Cipher!

Enter plain text: BIG THINGS HAVE SMALL BEGINNINGS
Enter depth: 3
Plain Text:   BIG THINGS HAVE SMALL BEGINNINGS
Cipher Text:  BHSELGI ITIGHVSALEINNS GNAMBNG


Enter 1 to Encrypt.
Enter 2 to Decrypt.
Enter 0 to Exit.
Enter choice: 2

Decrypting Rail Fence Cipher!

Enter cipher text: BHSELGI ITIGHVSALEINNS GNAMBNG
Enter depth: 3
Cipher Text:  BHSELGI ITIGHVSALEINNS GNAMBNG
Plain Text:   BIGTHINGSHAVESMALLBEGINNINGS


Enter 1 to Encrypt.
Enter 2 to Decrypt.
Enter 0 to Exit.
Enter choice: 0

Program exited successfully!
```

<u>PRACTICAL</u> - <u>7</u>

<u>AIM:</u> TO IMPLEMENT THE DES (DATA ENCRYPTION STANDARD) ALGORITHM

<u>BRIEF</u> :-

The Data Encryption Standard (DES) is a symmetric-key block cipher developed by IBM in the early 1970s and later adopted by the U.S. National Institute of Standards and Technology (NIST) as a federal encryption standard in 1977. It operates on 64-bit blocks of data using a 56-bit key (excluding 8 parity bits). DES works by dividing the plaintext into two halves and then processing them through 16 rounds of complex operations, including substitution, permutation, and bitwise logical operations based on the key. Its core structure follows the Feistel cipher model, which ensures that encryption and decryption processes are similar, enhancing efficiency.

Despite its historical importance, DES is now considered insecure due to advances in computing power. Its relatively short key length makes it vulnerable to brute-force attacks, with real-world examples successfully cracking DES-encrypted messages in under a day. As a result, DES has been largely replaced by stronger encryption standards such as Triple DES (3DES) and the Advanced Encryption Standard (AES). However, DES remains a foundational algorithm in the field of cryptography, widely studied for educational purposes and as a basis for understanding modern encryption systems.

ALGORITHM / PSEUDOCODE FOR ENCRYPTION :-

```
print menu
read plaintext
read key

if key length ≠ 8 then
    error "Key must be 8 characters"
    exit

// Pad plaintext to a multiple of 8 bytes (PKCS5 style)
pad_len = 8 - (length(plaintext) mod 8)
plaintext += chr(pad_len) × pad_len

// Generate the 16 round keys
round_keys = []
key_bits = permute(string_to_bits(key), PC1)
L, R = split(key_bits, 28)
for each shift in shift_schedule do
    L = left_shift(L, shift)
    R = left_shift(R, shift)
    round_keys.append( permute(L + R, PC2) )
end for

ciphertext = ""

// Process each 8-byte block
for each block in split(plaintext, 8 bytes) do
    bits = string_to_bits(block)
    bits = permute(bits, IP)

    L = bits[0..31]
    R = bits[32..63]

    // 16 Feistel rounds
    for i = 1 to 16 do
        E = permute(R, E_BOX)
        X = xor(E, round_keys[i])
        S = apply_sboxes(X)
        P = permute(S, P_BOX)
        L, R = R, xor(L, P)
    end for

    // Swap and final permutation
    preout = R + L
    cipher_bits = permute(preout, IP_INVERSE)
    ciphertext += bits_to_string(cipher_bits)
end for

hex_output = bytes_to_hex(ciphertext)
print hex_output
```

CODE FOR ENCRYPTION :-

```
# Implementation of DES (Data Encryption Standard) Algorithm in Python

# DES Tables
ip_table = [
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17, 9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7
]

pc1_table = [
    57, 49, 41, 33, 25, 17, 9, 1,
    58, 50, 42, 34, 26, 18, 10, 2,
    59, 51, 43, 35, 27, 19, 11, 3,
    60, 52, 44, 36, 63, 55, 47, 39,
    31, 23, 15, 7, 62, 54, 46, 38,
    30, 22, 14, 6, 61, 53, 45, 37,
    29, 21, 13, 5, 28, 20, 12, 4
]

pc2_table = [
    14, 17, 11, 24, 1, 5, 3, 28,
    15, 6, 21, 10, 23, 19, 12, 4,
    26, 8, 16, 7, 27, 20, 13, 2,
    41, 52, 31, 37, 47, 55, 30, 40,
    51, 45, 33, 48, 44, 49, 39, 56,
    34, 53, 46, 42, 50, 36, 29, 32
]

e_box_table = [
    32, 1, 2, 3, 4, 5,
    4, 5, 6, 7, 8, 9,
    8, 9, 10, 11, 12, 13,
    12, 13, 14, 15, 16, 17,
    16, 17, 18, 19, 20, 21,
    20, 21, 22, 23, 24, 25,
    24, 25, 26, 27, 28, 29,
    28, 29, 30, 31, 32, 1
]

p_box_table = [
    16, 7, 20, 21, 29, 12, 28, 17,
    1, 15, 23, 26, 5, 18, 31, 10,
    2, 8, 24, 14, 32, 27, 3, 9,
```

```
    19, 13, 30, 6, 22, 11, 4, 25
]

ip_inverse_table = [
    40, 8, 48, 16, 56, 24, 64, 32,
    39, 7, 47, 15, 55, 23, 63, 31,
    38, 6, 46, 14, 54, 22, 62, 30,
    37, 5, 45, 13, 53, 21, 61, 29,
    36, 4, 44, 12, 52, 20, 60, 28,
    35, 3, 43, 11, 51, 19, 59, 27,
    34, 2, 42, 10, 50, 18, 58, 26,
    33, 1, 41, 9, 49, 17, 57, 25
]

shift_schedule = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

s_boxes = [
    # S-box 1
    [
        [14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
        [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
        [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
        [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]
    ],
    # S-box 2
    [
        [15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
        [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
        [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
        [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]
    ],
    # S-box 3
    [
        [10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
        [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
        [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
        [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]
    ],
    # S-box 4
    [
        [7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 51],
        [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
        [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
        [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]
    ],
    # S-box 5
    [
        [2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
        [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
        [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
        [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]
    ],
```

```
    # S-box 6
    [
        [12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
        [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
        [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
        [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]
    ],
    # S-box 7
    [
        [4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
        [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
        [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
        [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]
    ],
    # S-box 8
    [
        [13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
        [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
        [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
        [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]
    ]
]

def string_to_bin(text):
    return ''.join(format(ord(char), '08b') for char in text)

def bin_to_string(binary):
    return ''.join(chr(int(binary[i:i+8], 2)) for i in range(0,
len(binary), 8))

def permute(input_block, table):
    return ''.join(input_block[i-1] for i in table)

def left_shift(data, shifts):
    return data[shifts:] + data[:shifts]

def xor(a, b):
    return ''.join('1' if x != y else '0' for x, y in zip(a, b))

def apply_sbox(expanded_block):
    output = ""
    for i in range(8):
        block = expanded_block[i*6:(i+1)*6]
        row = int(block[0] + block[5], 2)
        col = int(block[1:5], 2)
        output += format(s_boxes[i][row][col], '04b')
    return output

def generate_round_keys(key):
    key = string_to_bin(key)

    key = permute(key, pc1_table)
```

```python
    left = key[:28]
    right = key[28:]

    round_keys = []

    for i in range(16):
        left = left_shift(left, shift_schedule[i])
        right = left_shift(right, shift_schedule[i])
        combined = left + right
        round_key = permute(combined, pc2_table)
        round_keys.append(round_key)

    return round_keys

def des_round(left, right, round_key):
    expanded = permute(right, e_box_table)
    xored = xor(expanded, round_key)
    substituted = apply_sbox(xored)
    permuted = permute(substituted, p_box_table)
    new_right = xor(left, permuted)
    return right, new_right

def pad_text(text):
    pad_length = 8 - (len(text) % 8)
    return text + chr(pad_length) * pad_length

def des_encrypt(plaintext, key):
    if len(key) != 8:
        raise ValueError("Key must be exactly 8 characters long")
    plaintext = pad_text(plaintext)
    round_keys = generate_round_keys(key)

    ciphertext = ""

    for i in range(0, len(plaintext), 8):
        block = plaintext[i:i+8]
        block_bin = string_to_bin(block)
        block_bin = permute(block_bin, ip_table)
        left = block_bin[:32]
        right = block_bin[32:]
        for j in range(16):
            left, right = des_round(left, right, round_keys[j])
        left, right = right, left
        combined = left + right
        encrypted_block = permute(combined, ip_inverse_table)
        ciphertext += bin_to_string(encrypted_block)
    return ciphertext

if __name__ == "__main__":
    plaintext = input("Enter plaintext: ")
    key = input("Enter 8-character key: ")
```

```
    try:
        ciphertext = des_encrypt(plaintext, key)
        hex_ciphertext = ''.join(format(ord(c), '02x') for c in ciphertext)
        print(f"\nCiphertext (hex): {hex_ciphertext}")

    except ValueError as e:
        print(f"Error: {e}")
```

OUTPUT FOR ENCRYPTION :-

```
@sanaysarthak ➜ /workspaces/crypto-lab/DES Algorithm (main) $ python des-encrypt.py
Enter plaintext: BACKDOOR
Enter 8-character key: WINDOWS
Error: Key must be exactly 8 characters long
@sanaysarthak ➜ /workspaces/crypto-lab/DES Algorithm (main) $ python des-encrypt.py
Enter plaintext: BACKDOOR
Enter 8-character key: WINDOWSS

Ciphertext (hex): 048f648984ebf2050ac787c03aa78ee5
```

ALGORITHM / PSEUDOCODE FOR DECRYPTION :-

```
BEGIN

  DISPLAY menu
  INPUT ciphertext (in hex)
  INPUT key (must be 8 characters)

  IF LENGTH(key) ≠ 8 THEN
    DISPLAY "Key must be 8 characters long"
    EXIT
  ENDIF

  raw ← hex_to_string(ciphertext)
  key_bits ← permute(string_to_bits(key), PC1)
  L, R ← split(key_bits, 28)

  round_keys ← EMPTY LIST
  FOR each shift IN shift_schedule DO
    L ← left_shift(L, shift)
    R ← left_shift(R, shift)
    round_key ← permute(L + R, PC2)
    APPEND round_key TO round_keys
  ENDFOR

  plaintext ← ""

  FOR each block IN split(raw, 8 bytes) DO
    bits ← string_to_bits(block)

    bits ← permute(bits, IP)
    L ← bits[0..31]
    R ← bits[32..63]

    FOR i FROM 15 DOWNTO 0 DO
      L, R ← des_round(L, R, round_keys[i])
    ENDFOR

    pre_output ← R + L
    decrypted_bits ← permute(pre_output, IP_INVERSE)
    plaintext ← plaintext + bits_to_string(decrypted_bits)
  ENDFOR

  plaintext ← unpad_text(plaintext)

  DISPLAY "Decrypted Plaintext: ", plaintext

END
```

CODE FOR DECRYPTION :-

```
# Implementation of DES (Data Encryption Standard) Algorithm in Python

# DES Tables
ip_table = [
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17, 9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7
]

pc1_table = [
    57, 49, 41, 33, 25, 17, 9, 1,
    58, 50, 42, 34, 26, 18, 10, 2,
    59, 51, 43, 35, 27, 19, 11, 3,
    60, 52, 44, 36, 63, 55, 47, 39,
    31, 23, 15, 7, 62, 54, 46, 38,
    30, 22, 14, 6, 61, 53, 45, 37,
    29, 21, 13, 5, 28, 20, 12, 4
]

pc2_table = [
    14, 17, 11, 24, 1, 5, 3, 28,
    15, 6, 21, 10, 23, 19, 12, 4,
    26, 8, 16, 7, 27, 20, 13, 2,
    41, 52, 31, 37, 47, 55, 30, 40,
    51, 45, 33, 48, 44, 49, 39, 56,
    34, 53, 46, 42, 50, 36, 29, 32
]

e_box_table = [
    32, 1, 2, 3, 4, 5,
    4, 5, 6, 7, 8, 9,
    8, 9, 10, 11, 12, 13,
    12, 13, 14, 15, 16, 17,
    16, 17, 18, 19, 20, 21,
    20, 21, 22, 23, 24, 25,
    24, 25, 26, 27, 28, 29,
    28, 29, 30, 31, 32, 1
]

p_box_table = [
    16, 7, 20, 21, 29, 12, 28, 17,
    1, 15, 23, 26, 5, 18, 31, 10,
    2, 8, 24, 14, 32, 27, 3, 9,
```

```
    19, 13, 30, 6, 22, 11, 4, 25
]

ip_inverse_table = [
    40, 8, 48, 16, 56, 24, 64, 32,
    39, 7, 47, 15, 55, 23, 63, 31,
    38, 6, 46, 14, 54, 22, 62, 30,
    37, 5, 45, 13, 53, 21, 61, 29,
    36, 4, 44, 12, 52, 20, 60, 28,
    35, 3, 43, 11, 51, 19, 59, 27,
    34, 2, 42, 10, 50, 18, 58, 26,
    33, 1, 41, 9, 49, 17, 57, 25
]

shift_schedule = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

s_boxes = [
    # S-box 1
    [
        [14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
        [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
        [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
        [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]
    ],
    # S-box 2
    [
        [15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
        [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
        [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
        [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]
    ],
    # S-box 3
    [
        [10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
        [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
        [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
        [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]
    ],
    # S-box 4
    [
        [7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 51],
        [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
        [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
        [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]
    ],
    # S-box 5
    [
        [2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
        [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
        [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
        [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]
    ],
```

```
    # S-box 6
    [
        [12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
        [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
        [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
        [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]
    ],
    # S-box 7
    [
        [4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
        [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
        [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
        [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]
    ],
    # S-box 8
    [
        [13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
        [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
        [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
        [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]
    ]
]

def string_to_bin(text):
    return ''.join(format(ord(char), '08b') for char in text)

def bin_to_string(binary):
    return ''.join(chr(int(binary[i:i+8], 2)) for i in range(0,
len(binary), 8))

def hex_to_string(hex_text):
    return ''.join(chr(int(hex_text[i:i+2], 16)) for i in range(0,
len(hex_text), 2))

def permute(input_block, table):
    return ''.join(input_block[i-1] for i in table)

def left_shift(data, shifts):
    return data[shifts:] + data[:shifts]

def xor(a, b):
    return ''.join('1' if x != y else '0' for x, y in zip(a, b))

def apply_sbox(expanded_block):
    output = ""
    for i in range(8):
        block = expanded_block[i*6:(i+1)*6]
        row = int(block[0] + block[5], 2)
        col = int(block[1:5], 2)
        output += format(s_boxes[i][row][col], '04b')
    return output
```

```python
def generate_round_keys(key):
    key = string_to_bin(key)
    key = permute(key, pc1_table)
    left = key[:28]
    right = key[28:]
    round_keys = []
    for i in range(16):
        left = left_shift(left, shift_schedule[i])
        right = left_shift(right, shift_schedule[i])
        combined = left + right
        round_key = permute(combined, pc2_table)
        round_keys.append(round_key)
    return round_keys

def des_round(left, right, round_key):
    expanded = permute(right, e_box_table)
    xored = xor(expanded, round_key)
    substituted = apply_sbox(xored)
    permuted = permute(substituted, p_box_table)
    new_right = xor(left, permuted)
    return right, new_right

def unpad_text(text):
    pad_value = ord(text[-1])
    if pad_value > 0 and pad_value <= 8:
        for i in range(1, pad_value + 1):
            if ord(text[-i]) != pad_value:
                return text
        return text[:-pad_value]
    return text

def des_decrypt(hex_ciphertext, key):
    if len(key) != 8:
        raise ValueError("Key must be exactly 8 characters long")
    ciphertext = hex_to_string(hex_ciphertext)
    round_keys = generate_round_keys(key)
    plaintext = ""
    for i in range(0, len(ciphertext), 8):
        block = ciphertext[i:i+8]
        block_bin = string_to_bin(block)
        block_bin = permute(block_bin, ip_table)
        left = block_bin[:32]
        right = block_bin[32:]
        for j in range(15, -1, -1):
            left, right = des_round(left, right, round_keys[j])
        left, right = right, left
        combined = left + right
        decrypted_block = permute(combined, ip_inverse_table)
        plaintext += bin_to_string(decrypted_block)
    return unpad_text(plaintext)

if __name__ == "__main__":
```

```
hex_ciphertext = input("Enter ciphertext (hex): ")
key = input("Enter 8-character key: ")
try:
    hex_ciphertext = hex_ciphertext.replace(" ", "")
    plaintext = des_decrypt(hex_ciphertext, key)
    print(f"\nPlaintext: {plaintext}")
except ValueError as e:
    print(f"Error: {e}")
```

OUTPUT FOR DECRYPTION :-

```
@sanaysarthak ➜ /workspaces/crypto-lab/DES Algorithm (main) $ python des-decrypt.py
Enter ciphertext (hex): 048f648984ebf2050ac787c03aa78ee5
Enter 8-character key: WINDOWSS

Plaintext: BACKDOOR
```

PRACTICAL - 8

AIM: TO IMPLEMENT THE DIFFIE HELLMAN KEY EXCHANGE ALGORITHM

BRIEF :-

The Diffie Hellman Key Exchange is a method that allows two people to securely share a secret key over a public channel. This means they can agree on a key for encryption without actually sending the key itself. It works by using mathematical operations based on large prime numbers and modular arithmetic. Each person picks a private number and then creates a public value using a shared base and prime. These public values are exchanged, and each person uses the other's public value along with their own private number to compute the same secret key.

This shared secret key can then be used for encrypted communication between the two parties. The strength of the Diffie-Hellman method lies in the difficulty of reversing the mathematical operations (a problem known as the Discrete Logarithm Problem). Even if someone sees the public values being shared, they can't easily figure out the secret key. However, Diffie-Hellman alone does not provide authentication, so it is often combined with other security methods to prevent attacks like man-in-the-middle.

ALGORITHM / PSEUDOCODE :-

```
print "Diffie-Hellman Key Exchange"
read sender_name
read receiver_name

read p
if p < 2 then
   error
end if

read g
if g ≤ 1 or g ≥ p or not is_primitive_root(g,p) then
    error
end if

read sender_priv
read receiver_priv
if sender_priv ≤ 0 or sender_priv ≥ p or receiver_priv ≤ 0 or receiver_priv ≥
p then
    error
end if

sender_pub -= mod_exp(g, sender_priv, p)
receiver_pub = mod_exp(g, receiver_priv, p)
sender_shared = mod_exp(receiver_pub, sender_priv, p)
receiver_shared = mod_exp(sender_pub, receiver_priv, p)

print sender_name + " pub:", sender_pub
print receiver_name + " pub:", receiver_pub
print sender_name + " shared:", sender_shared
print receiver_name + " shared:", receiver_shared

if sender_shared == receiver_shared then
    print "Shared key established"
else
    print "Error: keys mismatch"
end if

function mod_exp(b, e, m)
    return b^e mod m
end function

function is_primitive_root(g, p)
    return {g^k mod p | k←1..p-1} == {1..p-1}
end function
```

CODE :-

```
def mod_exp(base, exponent, modulus):
    return pow(base, exponent, modulus)

def is_primitive_root(g, p):
    required_set = set(num for num in range(1, p))
    actual_set = set(pow(g, powers, p) for powers in range(1, p))
    return required_set == actual_set

print("Diffie-Hellman Key Exchange :-\n")
sender_name = str(input("Enter sender's name: "))
receiver_name = str(input("Enter receiver's name: "))

p = int(input("\nEnter a large prime number (p): "))
if p < 2:
    raise ValueError("Prime number must be greater than 1.")

g = int(input(f"Enter a base number (generator) less than {p}: "))
if g <= 1 or g >= p:
    raise ValueError(f"Base number must be > 1 and < {p}.")
if not is_primitive_root(g, p):
    raise ValueError(f"{g} is not a valid generator (primitive root)
for {p}.")

sender_private = int(input(f"Enter {sender_name}'s private key (number
< p): "))
receiver_private = int(input(f"Enter {receiver_name}'s private key
(number < p): "))
if not (0 < sender_private < p and 0 < receiver_private < p):
    raise ValueError(f"Private keys must be between 1 and {p - 1}.")

sender_public = mod_exp(g, sender_private, p)
receiver_public = mod_exp(g, receiver_private, p)
sender_shared_key = mod_exp(receiver_public, sender_private, p)
receiver_shared_key = mod_exp(sender_public, receiver_private, p)

print("\nResults :-")
print(sender_name + "'s Public Key:", sender_public)
print(receiver_name + "'s Public Key:", receiver_public)
print(sender_name + "'s Shared Key:", sender_shared_key)
print(receiver_name + "'s Shared Key:", receiver_shared_key)

if sender_shared_key == receiver_shared_key:
    print("\nShared key successfully established!")
else:
    print("\nError: Shared keys do not match.")
```

OUTPUT :-

```
@sanaysarthak ➜ /workspaces/crypto-lab/practicals (main) $ python diffie-hellman.py
Diffie-Hellman Key Exchange :-

Enter sender's name: Sarthak
Enter receiver's name: Sanay

Enter a large prime number (p): 23
Enter a base number (generator) less than 23: 11
Enter Sarthak's private key (number < p): 1
Enter Sanay's private key (number < p): 15

Results :-
Sarthak's Public Key: 11
Sanay's Public Key: 10
Sarthak's Shared Key: 10
Sanay's Shared Key: 10

Shared key successfully established!
```

```
@sanaysarthak ➜ /workspaces/crypto-lab/practicals (main) $ python diffie-hellman.py
Diffie-Hellman Key Exchange :-

Enter sender's name: Elon
Enter receiver's name: Bezos

Enter a large prime number (p): 23
Enter a base number (generator) less than 23: 7
Enter Elon's private key (number < p): 11
Enter Bezos's private key (number < p): 14

Results :-
Elon's Public Key: 22
Bezos's Public Key: 2
Elon's Shared Key: 1
Bezos's Shared Key: 1

Shared key successfully established!
```