EX NO 1: Write a program to implement the following cipher techniques to perform encryption and decryption using using ceaser cipher

AIM:

To implement a python program to implement the following cipher techniques to perform encryption and decryption using ceaser cipher.

ALGOTITHM:

Encryption:

STEP1: Take the plaintext message and the shift value.

STEP2: Initialize: Set the ciphertext variable as an empty string.

STEP 3: Iterate through the plaintext: For each character in the plaintext message, do the following:

If the character is a letter:

Convert the letter to its numerical equivalent (e.g., 'A' is 0, 'B' is 1, ..., 'Z' is 25).

Add the shift value to the numerical equivalent.

Take the result modulo 26 to handle wrapping around the alphabet.

If the result is negative, add 26 to bring it back into the range [0, 25].

STEP 4: Convert the resulting number back to the corresponding letter.

STEP 5: Append the resulting character to the ciphertext.

STEP 6: The ciphertext is the encrypted message

Decryption:

STEP 1: Take the ciphertext message and the negative of the shift value used during

encryption.

STEP 2: Set the plaintext variable as an empty string.

STEP 3: Iterate through the ciphertext: For each character in the ciphertext message, do the following:

If the character is a letter:

Convert the letter to its numerical equivalent.

Subtract the negative of the shift value from the numerical equivalent.

Take the result modulo 26 to handle wrapping around the alphabet.

If the result is negative, add 26 to bring it back into the range [0, 25].

STEP 4: Convert the resulting number back to the corresponding letter.

STEP 5: Append the resulting character to the plaintext.

STEP 6: Output: The plaintext is the decrypted message.

PROGRAM:

```
def caesar_cipher_encrypt(plain_text, shift):
encrypted_text = ""
 for char in plain_text:
 if char.isalpha():
 shifted = ord(char) + shift
 if char.islower():
  if shifted >ord('z'):
  shifted -= 26
```

```
elif shifted <ord('a'):
shifted += 26
    elif char.isupper():
     if shifted >ord('Z'):
    shifted -= 26
     elif shifted <ord('A'):
     shifted += 26
    encrypted_text += chr(shifted)
       else:
    encrypted_text += char
     return encrypted_text
       def caesar_cipher_decrypt(cipher_text, shift):
      decrypted_text = ""
   for char in cipher_text:
      if char.isalpha():
      shifted = ord(char) - shift
     if char.islower():
      if shifted >ord('z'):
      shifted -= 26
     elif shifted <ord('a'):
     shifted += 26
```

```
 elif char.isupper():
  if shifted >ord('Z'):
 shifted -= 26
 elif shifted <ord('A'):
  shifted += 26
 decrypted_text += chr(shifted)
 else:
 decrypted_text += char
 return decrypted_text
 def main():
choice = input("Enter 'e' for encryption or 'd' for decryption: ").lower()
if choice == 'e':
plain_text = input("Enter the plain text: ")
shift = int(input("Enter the shift value: "))
encrypted_text = caesar_cipher_encrypt(plain_text, shift)
print("Encrypted text:", encrypted_text)
elif choice == 'd':
 cipher_text = input("Enter the cipher text: ")
 shift = int(input("Enter the shift value: "))
   decrypted_text = caesar_cipher_decrypt(cipher_text, shift)
   print("Decrypted text:", decrypted_text)
```

```
else:
print("Invalid choice!")
if __name__ == "__main__":
main()
```

OUTPUT:

Encryption

```
Enter 'e' for encryption or 'd' for
    decryption: E
Enter the plain text: Hello, World!
Enter the shift value: 3
Encrypted text: Khoor, Zruog!
```

Decryption

```
Enter 'e' for encryption or 'd' for
    decryption: D
Enter the cipher text: Khoor, Zruog!
Enter the shift value: 3
Decrypted text: Hello, World!
```

RESULT:

Thus, the python program to implement the cipher techniques to perform encryption and decryption using using ceaser cipher has been verified and executed successfully.

EXNO: 2 Write a program to implement the following cipher techniques to perform encryption and decryption using Play fair cipher.

AIM:

To implement a python program to implement the following cipher techniques to perform encryption and decryption using Play fair cipher.

ALGORITHM:

Encryption:

STEP1: Input: Take the plaintext message and the keyword.

STEP 2: Preprocessing:

STEP 3: Remove any non-alphabetic characters from the plaintext and convert all letters to uppercase.

STEP 4: Remove any duplicate letters from the keyword and combine it with the remaining letters of the alphabet to form the Playfair matrix.

STEP 5: Fill the 5x5 Playfair matrix with the unique letters from the keyword followed by the remaining letters of the alphabet.

STEP 6: Construct Digraphs:

Break the plaintext into pairs of letters (digraphs).

If there is an odd number of letters, append an 'X' to the last letter to form a digraph.

STEP 7: If the letters are in the same row of the matrix, replace each letter with the letter to its right (wrapping around to the leftmost letter if necessary).

STEP 8: If the letters are in the same column of the matrix, replace each letter with the letter below it (wrapping around to the topmost letter if necessary).

STEP 9: If the letters are not in the same row, column, or rectangle, form a rectangle with the

Two letters and replace them with the letters on the same row but opposite corners of the rectangle.

STEP 10: The ciphertext is the result of combining all the encrypted digraphs.

Decryption:

STEP 1: Input: Take the ciphertext message and the keyword used for encryption.

STEP 2: Preprocessing:

STEP 3: Generate the Playfair matrix using the keyword, following the same process as for encryption.

STEP 4: Break the ciphertext into digraphs.

STEP 5: If the letters are in the same row of the matrix, replace each letter with the letter to its left (wrapping around to the rightmost letter if necessary).

STEP 6: If the letters are in the same column of the matrix, replace each letter with the letter above it(wrapping around to the bottommost letter if necessary).

STEP 7: If the letters form a rectangle in the matrix, replace them with the letters on the same row but opposite corners of the rectangle.

STEP 8: If the letters are not in the same row, column, or rectangle, form a rectangle with the two letters and replace them with the letters on the same row but opposite corners of the rectangle.

STEP 9: The decrypted plaintext is the result of combining all the decrypted digraphs.

PROGRAM:

```
def generate_playfair_matrix(key):
# Create a 5x5 matrix initialized with zeros
matrix = [[" for _ in range(5)] for _ in range(5)]
# Remove duplicates from the key and combine with the alphabet
key = ''.join(dict.fromkeys(key + 'abcdefghijklmnopqrstuvwxyz').keys())
# Fill the matrix with the key and remaining alphabets
```

```
alphabet = 'abcdefghiklmnopqrstuvwxyz'

i, j = 0, 0

for char in key:

if char == 'j':

char = 'i'  # In Playfair, 'i' and 'j' are usually treated as the same

matrix[i][j] = char

j += 1

if j == 5:

i += 1

j = 0

for char in alphabet:

if char == 'j':

continue

matrix[i][j] = char

j += 1

if j == 5:

i += 1

j = 0

return matrix

def get_char_position(matrix, char):

for i in range(5):
```

```
for j in range(5):

if matrix[i][j] == char:

return i, j

def playfair_encrypt(plain_text, key):

plain_text = plain_text.replace(' ', '').lower()

plain_text = plain_text.replace('j', 'i')  # Replace 'j' with 'i'

matrix = generate_playfair_matrix(key)

encrypted_text = ""

for i in range(0, len(plain_text), 2):

pair1 = plain_text[i]

pair2 = plain_text[i + 1] if i + 1 <len(plain_text) else 'x'

if pair2 == pair1:

pair2 = 'x'

encrypted_text += pair1 + pair2

continue

row1, col1 = get_char_position(matrix, pair1)

row2, col2 = get_char_position(matrix, pair2)

if row1 == row2:  # Same row

encrypted_text += matrix[row1][(col1 + 1) % 5] + matrix[row2][(col2 + 1) % 5]

elif col1 == col2:  # Same column

encrypted_text += matrix[(row1 + 1) % 5][col1] + matrix[(row2 + 1) % 5][col2]
```

```
else:  # Forming a rectangle
encrypted_text += matrix[row1][col2] + matrix[row2][col1]
return encrypted_text.upper()
def playfair_decrypt(cipher_text, key):
matrix = generate_playfair_matrix(key)
decrypted_text = ""
for i in range(0, len(cipher_text), 2):
pair1 = cipher_text[i]
pair2 = cipher_text[i + 1]
row1, col1 = get_char_position(matrix, pair1)
row2, col2 = get_char_position(matrix, pair2)
if row1 == row2:  # Same row
decrypted_text += matrix[row1][(col1 - 1) % 5] + matrix[row2][(col2 - 1) % 5]
elif col1 == col2:  # Same column
decrypted_text += matrix[(row1 - 1) % 5][col1] + matrix[(row2 - 1) % 5][col2]
else:  # Forming a rectangle
decrypted_text += matrix[row1][col2] + matrix[row2][col1]
return decrypted_text.replace('x', '')
def main():
key = input("Enter the key for Playfair cipher: ").lower()
plain_text = input("Enter the plain text: ").lower()
```

```
encrypted_text = playfair_encrypt(plain_text, key)

print("Encrypted text:", encrypted_text)

decrypted_text = playfair_decrypt(encrypted_text, key)

print("Decrypted text:", decrypted_text)

if __name__ == "__main__":

main()
```

Output:

Encryption

```
Enter the key for Playfair cipher: keyword
Enter the plain text: Hide the gold in the tree stump
Encrypted text: BMNDZBXDKYBEJVDMUIXMMNUVIF
```

Decryption

```
Enter the key for Playfair cipher: keyword
Enter the cipher text: BMNDZBXDKYBEJVDMUIXMMNUVIF
Decrypted text: hidethegoldinthetreestumpx
```

RESULT:

Thus, the python program to implement the cipher techniques to perform encryption and decryption using Play fair cipher has been verified and executed successfully

EX NO: 3 Write a program to implement the following cipher techniques to perform encryption and decryption using Hill Cipher

AIM:

To implement a python program to implement the following cipher techniques to perform encryption and decryption using Hill Cipher.

ALGORITHM:

Encryption:

STEP 1: Input: Take the plaintext message and the encryption key matrix.

STEP 2: Preprocessing:

STEP 3: Convert the plaintext message into numerical values, typically using a simple mapping (e.g., A=0, B=1, ..., Z=25).

STEP 4: If necessary, pad the plaintext to ensure its length is a multiple of the key matrix size.

Encryption Rules:

STEP 5: Divide the plaintext into blocks of size n (where n is the dimension of the key matrix).

For each plaintext block:

Represent the block as a column vector.

Multiply the key matrix by the plaintext vector.

Take the result modulo 26.

STEP 6: Convert the resulting numbers back to letters using the reverse mapping.

STEP 7: The ciphertext is the concatenation of the encrypted blocks.

Decryption:

STEP 1: Input: Take the ciphertext message and the decryption key matrix.

STEP 2: Preprocessing:

STEP 3: Convert the ciphertext message into numerical values using the same mapping as for encryption.

STEP 4: Decryption Rules:

Divide the ciphertext into blocks of size n.

For each ciphertext block:

Represent the block as a column vector.

STEP 5: Compute the inverse of the decryption key matrix modulo 26 (if it exists).

STEP 6: If the inverse doesn't exist (e.g., determinant is zero), decryption cannot proceed.

Multiply the inverse matrix by the ciphertext vector.

Take the result modulo 26.

STEP 7: Convert the resulting numbers back to letters using the reverse mapping.

STEP 8: The plaintext is the concatenation of the decrypted blocks.

PROGRAM

```
import numpy as np

# Function to convert text to numbers (A=0, B=1, ..., Z=25)

def text_to_numbers(text):

    return [ord(char.lower()) - ord('a') for char in text if char.isalpha()]

# Function to convert numbers to text

def numbers_to_text(numbers):

 return ''.join(chr(num + ord('a')) for num in numbers)

# Function to generate the key matrix from the key string

def generate_key_matrix(key):

 key = key.replace(' ', '').lower()

key_len = len(key)

matrix_dim = int(key_len ** 0.5)

 if matrix_dim ** 2 != key_len:

 raise ValueError("Key length must be a perfect square")

key_matrix = np.array([ord(char) - ord('a') for char in key]).reshape((matrix_dim, matrix_dim))

    return key_matrix

# Function to encrypt plaintext using Hill Cipher
```

```
def hill_cipher_encrypt(plain_text, key_matrix):
plain_text = plain_text.replace(' ', '').lower()
plain_text = [ord(char) - ord('a') for char in plain_text if char.isalpha()]
padding = len(plain_text) % key_matrix.shape[0]
 if padding > 0:
plain_text += [0] * (key_matrix.shape[0] - padding)
plain_text_matrix = np.array(plain_text).reshape((-1, key_matrix.shape[0]))
encrypted_text_matrix = (plain_text_matrix @ key_matrix) % 26
encrypted_text = numbers_to_text(encrypted_text_matrix.flatten())
    return encrypted_text.upper()
# Function to decrypt ciphertext using Hill Cipher
def hill_cipher_decrypt(cipher_text, key_matrix):
cipher_text = cipher_text.replace(' ', '').lower()
cipher_text = [ord(char) - ord('a') for char in cipher_text if char.isalpha()]
cipher_text_matrix = np.array(cipher_text).reshape((-1, key_matrix.shape[0]))
decrypted_text_matrix = (cipher_text_matrix @ np.linalg.inv(key_matrix)) % 26
decrypted_text = numbers_to_text(decrypted_text_matrix.flatten())
    return decrypted_text.upper()
def main():
    key = input("Enter the key for Hill cipher (as a string of lowercase letters, no spaces): ")
plain_text = input("Enter the plain text: ")
```

```
    try:
key_matrix = generate_key_matrix(key)
    except ValueError as e:
        print(e)
        return
encrypted_text = hill_cipher_encrypt(plain_text, key_matrix)
print("Encrypted text:", encrypted_text)
decrypted_text = hill_cipher_decrypt(encrypted_text, key_matrix)
print("Decrypted text:", decrypted_text)
if __name__ == "__main__":
main()
```

OUTPUT:

```
Enter the key for Hill cipher (as a string of lowercase letters, no spaces): test
Enter the plain text: hello
Encrypted text: YTNBZ
Decrypted text: HELLO
```

RESULT:

Thus, the python program to implement the cipher techniques to perform encryption and decryption using Hill Cipher has been verified and executed successfully.

EX NO:4 write a python program to implement Rail fence technique-Row major transformation

AIM:

To implement a python program to implement Rail fence technique-Row major transformation.

ALGORITHM:

STEP 1: Input: Take the ciphertext message and the number of rails (or rows) used for encryption.

STEP 2: Initialization:

Create an empty matrix with the number of rows equal to the number of rails and the number of columns equal to the length of the ciphertext.

STEP 3: Initialize variables row and col to 0 to track the current position in the matrix.

STEP 4: Initialize a boolean variable downward to true indicating that the next step will be downward.

STEP 5: Filling the Matrix:

Iterate through each position in the matrix:

STEP 6: If the position corresponds to a rail where a character should be placed according to the rail fence pattern, fill it with a special marker character (e.g., '*').

Otherwise, leave it empty.

STEP 7: Increment col to move to the next column**n**

Decryption Process:

STEP 1: Iterate through each character in the ciphertext:

STEP 2: Replace each marker character in the matrix with the corresponding character from the ciphertext.

STEP 3: Update the position according to the direction (downward or upward) and the rail fence pattern.

STEP 4: If downward is true, increment row; otherwise, decrement row.

STEP 5: If row reaches the bottom rail or the top rail, flip the direction by setting downward to its opposite value.

STEP 6: Read the characters from the matrix column by column, starting from the leftmost column to the rightmost column, to get the plaintext.

PROGRAM:

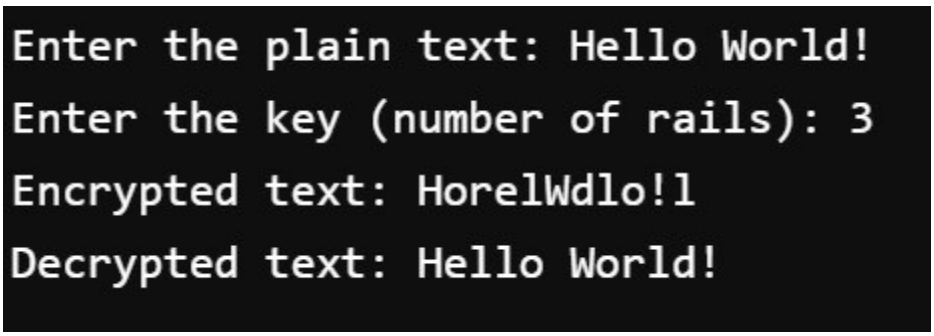
```
def rail_fence_encrypt(plain_text, key):
# Create the rail fence pattern
fence = [['\n' for _ in range(len(plain_text))] for _ in range(key)]
rail = 0
direction = False
for char in plain_text:
fence[rail][fence[rail].index('\n')] = char
if rail == 0 or rail == key - 1:
direction = not direction
if direction:
```

```
rail += 1

else:

rail -= 1

# Combine the rails into a single string

encrypted_text = ''.join([char for rail in fence for char in rail if char != '\n'])

return encrypted_text

def rail_fence_decrypt(cipher_text, key):

# Create the rail fence pattern

fence = [['\n' for _ in range(len(cipher_text))] for _ in range(key)]

rail = 0

direction = False

# Fill in the fence with placeholder characters

for i in range(len(cipher_text)):

fence[rail][i] = '*'

if rail == 0 or rail == key - 1:

direction = not direction

if direction:

rail += 1

else:

rail -= 1

# Place the characters of the ciphertext on the fence
```

```
index = 0

for i in range(key):

for j in range(len(cipher_text)):

if (fence[i][j] == '*') and (index <len(cipher_text)):

fence[i][j] = cipher_text[index]

index += 1

# Reconstruct the plaintext

rail = 0

direction = False

decrypted_text = ''

for j in range(len(cipher_text)):

decrypted_text += fence[rail][j]

if rail == 0 or rail == key - 1:

direction = not direction

if direction:

rail += 1

else:

rail -= 1

return decrypted_text

def main():

plain_text = input("Enter the plain text: ")
```

```
key = int(input("Enter the key (number of rails): "))

encrypted_text = rail_fence_encrypt(plain_text, key)

print("Encrypted text:", encrypted_text)

decrypted_text = rail_fence_decrypt(encrypted_text, key)

print("Decrypted text:", decrypted_text)

if __name__ == "__main__":

main()
```

OUTPUT



RESULT:

Thus, the python program to implement Rail fence technique-Row major transformation has been verified and executed successfully.

EX NO:5 Write a python program to implement Rail Fence cipher technique with column-major transformation.

AIM:

To implement a python program to implement Rail Fence cipher technique with column major transformation.

ALGORITHM:

Encryption:

STEP 1: Input: Take the plaintext message and the number of rails (or rows) for the rail fence.

STEP 2: Initialization:

STEP 3: Create an empty matrix with the number of rows equal to the number of rails and the number of columns equal to the length of the plaintext.

STEP 4: Initialize variables row and col to 0 to track the current position in the matrix.

Initialize a boolean variable downward to true indicating that the next step will be downward.

STEP 5: Iterate through each character in the plaintext message:

STEP 6: Place the current character in the matrix at position (row, col).

STEP 7: Update the position according to the direction (downward or upward) and the rail fence pattern. If downward is true, increment row; otherwise, decrement row.

If row reaches the bottom rail or the top rail, flip the direction by setting downward to its opposite value.

Increment col to move to the next column.

STEP 8: Read the characters from the matrix column by column, starting from the leftmost column to the rightmost column, to get the ciphertext.

Decryption:

STEP 1: Take the ciphertext message and the number of rails (or rows) used for encryption.

STEP 2: Initialization:

STEP 3: Create an empty matrix with the number of rows equal to the number of rails and the number of columns equal to the length of the ciphertext.

STEP 4: Initialize variables row and col to 0 to track the current position in the matrix.

Initialize a boolean variable downward to true indicating that the next step will be downward.

STEP 5: Filling the Matrix:

Iterate through each position in the matrix:

If the position corresponds to a rail where a character should be placed according to the rail fence pattern, fill it with a special marker character (e.g., '*').

Otherwise, leave it empty.

STEP 6: Increment col to move to the next column.

Decryption Process:

STEP 1: Iterate through each character in the ciphertext:

STEP 2: Replace each marker character in the matrix with the corresponding character from the ciphertext.

STEP 3: Update the position according to the direction (downward or upward) and the rail fence pattern.

If downward is true, increment row; otherwise, decrement row.

If row reaches the bottom rail or the top rail, flip the direction by setting downward to its opposite value.

STEP 4: Read the characters from the matrix row by row, starting from the top row to the bottom row, to get the plaintext.

PROGRAM:

```
def rail_fence_encrypt(plain_text, key):
# Create the rail fence pattern
fence = [['\n' for _ in range(len(plain_text))] for _ in range(key)]
rail = 0
direction = False
for char in plain_text:
fence[rail][fence[rail].index('\n')] = char
if rail == 0 or rail == key - 1:
direction = not direction
if direction:
rail += 1
else:
rail -= 1
```

```
# Combine the rails into a single string (column-major transformation)

encrypted_text = ''.join([char for rail in fence for char in rail if char != '\n'])

return encrypted_text

def rail_fence_decrypt(cipher_text, key):

# Create the rail fence pattern

fence = [['\n' for _ in range(len(cipher_text))] for _ in range(key)]

rail = 0

direction = False

# Fill in the fence with placeholder characters

for i in range(len(cipher_text)):

fence[rail][i] = '*'

if rail == 0 or rail == key - 1:

direction = not direction

if direction:

rail += 1

else:

rail -= 1

# Place the characters of the ciphertext on the fence

index = 0

for i in range(key):

for j in range(len(cipher_text)):
```

```
if (fence[i][j] == '*') and (index <len(cipher_text)):

fence[i][j] = cipher_text[index]

index += 1

# Reconstruct the plaintext (column-major transformation)

rail = 0

direction = False

decrypted_text = ''

for j in range(len(cipher_text)):

decrypted_text += fence[rail][j]

if rail == 0 or rail == key - 1:

direction = not direction

if direction:

rail += 1

else:

rail -= 1

return decrypted_text

def main():

plain_text = input("Enter the plain text: ")

key = int(input("Enter the key (number of rails): "))

encrypted_text = rail_fence_encrypt(plain_text, key)

print("Encrypted text:", encrypted_text)
```

```
decrypted_text = rail_fence_decrypt(encrypted_text, key)

print("Decrypted text:", decrypted_text)

if __name__ == "__main__":

main()
```

OUTPUT:

```
Enter the plain text: Hello World!
Enter the key (number of rails): 3
Encrypted text: HorelWdlo!l
Decrypted text: Hello World!
```

RESULT:

Thus, the python program to implement Rail Fence cipher technique with column major transformation has been verified and executed successfully.

EX NO: 6 Write a Python Program to implement DES algorithm

AIM:

To implement a python program to implement DES Encryption Algorithm

ALGORITHM:

STEP 1: Key Generation:

STEP 2: Take the 64-bit key as input.

Perform a permutation on the key to generate a 56-bit key.

Generate 16 subkeys of 48 bits each through key shifting and permutation.

STEP 3: Initial Permutation (IP):

Permute the 64-bit plaintext according to the initial permutation table.

STEP 4: Divide the permuted plaintext into two 32-bit halves, left and right.

STEP 5: Perform 16 rounds of processing:

STEP 6: For each round, perform the following steps:

STEP 7: Expand the right half to 48 bits.

XOR the expanded right half with the subkey for the current round.

Substitute the result using S-boxes.

Permute the result using a fixed permutation table (P-box).

XOR the permuted result with the left half.

STEP 8: Swap the left and right halves.

STEP 9: Final Permutation (FP):

After the 16 rounds, swap the left and right halves.

Permute the result using the final permutation table, which is the inverse of the initial permutation.

STEP 10: For decryption, use the subkeys in reverse order.If the plaintext length is not a multiple of 64 bits, add padding to make it a multiple of 64 bits.

PROGRAM

```
pip install pycryptodome

from Crypto.Cipher import DES

from Crypto.Random import get_random_bytes

from base64 import b64encode, b64decode

def des_encrypt(key, plain_text):

    cipher = DES.new(key, DES.MODE_ECB)

cipher_text = cipher.encrypt(plain_text)

    return b64encode(cipher_text)

def des_decrypt(key, cipher_text):

    cipher = DES.new(key, DES.MODE_ECB)

decrypted_text = cipher.decrypt(b64decode(cipher_text))

    return decrypted_text

def main():

    key = get_random_bytes(8)  # Generate a random 8-byte key

plain_text = b"Hello, DES!"

cipher_text = des_encrypt(key, plain_text)
```

```
print("Encrypted text:", cipher_text)

decrypted_text = des_decrypt(key, cipher_text)

print("Decrypted text:", decrypted_text.decode())

if __name__ == "__main__":

main()
```

OUTPUT

```
Encrypted text: b'OLWdqh3OTg9GUX1MXb7TCA=='
Decrypted text: Hello, DES!
```

RESULT:

Thus, the above python program to implement DES Encryption Algorithm has been verified and executed successfull

EXNO:7 Write a python program to implement AES encryption

AIM:

To implement a python program to implement AES Encryption Algorithm.

ALGORITHM

STEP 1: Key Expansion:

Take the encryption key as input.

Expand the key into a key schedule containing round keys for each round of encryption.

STEP 2: Initial Round Key Addition

XOR the plaintext block with the first-round key

For each round, perform the following steps:

SubBytes: Substitute each byte of the state matrix with a corresponding byte from an S-box.

ShiftRows: Shift the rows of the state matrix cyclically to the left.

MixColumns (except for the last round): Mix the columns of the state matrix using a predefined matrix multiplication operation.

AddRoundKey: XOR the state matrix with the round key derived from the key schedule.

STEP 3: Final Round (no MixColumns)

Perform the final round, which consists of SubBytes, ShiftRows, and AddRoundKey.

STEP 4: The resulting state matrix after the final round represents the ciphertext.

STEP 5: Decryption is similar to encryption but involves the reverse operations.

PROGRAM

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from base64 import b64encode, b64decode
def aes_encrypt(key, plain_text):
   cipher = AES.new(key, AES.MODE_ECB)
cipher_text = cipher.encrypt(plain_text)
    return b64encode(cipher_text)
def aes_decrypt(key, cipher_text):
    cipher = AES.new(key, AES.MODE_ECB)
decrypted_text = cipher.decrypt(b64decode(cipher_text))
    return decrypted_text
def main():
    key = get_random_bytes(16)  # Generate a random 16-byte (128-bit) key for AES-128
plain_text = b"Hello, AES!"
cipher_text = aes_encrypt(key, plain_text)
print("Encrypted text:", cipher_text)
decrypted_text = aes_decrypt(key, cipher_text)
print("Decrypted text:", decrypted_text.decode())
if __name__ == "__main__":
```

main()

OUTPUT

```
Encrypted text: b'U4nr/OEqc7ax18bWJolmaw=='
Decrypted text: Hello, AES!
```

RESULT:

Thus, the above python program to implement AES Encryption Algorithm has been verified and executed successfully.

EX NO:8 Write a program to implement RSA Encryption Algorithm

AIM:

To implement a python program to implement RSA Encryption Algorithm.

ALGORITHM:

STEP 1: Key Generation:

STEP 2: Choose two large prime numbers, p and q.

STEP 3: Calculate the modulus,

STEP 4: Calculate Euler's totient function,

STEP 5: Select a public exponent

STEP 6: Compute the private exponent

STEP 7: Convert the plaintext message

STEP 8: Convert the numerical result back to the original plaintext.

PROGRAM

```
import random

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def mod_inverse(a, m):
    m0, x0, x1 = m, 0, 1
    while a > 1:
```

```
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1
def generate_keypair(p, q):
    n = p * q
    phi = (p - 1) * (q - 1)
    e = random.randrange(1, phi)
    g = gcd(e, phi)
    while g != 1:
        e = random.randrange(1, phi)
        g = gcd(e, phi)
    d = mod_inverse(e, phi)
    return ((e, n), (d, n))
def encrypt(public_key, plaintext):
    key, n = public_key
cipher_text = [pow(ord(char), key, n) for char in plaintext]
    return cipher_text
def decrypt(private_key, cipher_text):
    key, n = private_key
plain_text = [chr(pow(char, key, n)) for char in cipher_text]
```

```
    return ''.join(plain_text)

def main():
    p = 61
    q = 53
public_key, private_key = generate_keypair(p, q)
print("Public key:", public_key)
print("Private key:", private_key)
message = "Hello, RSA!"
print("Original message:", message)
encrypted_msg = encrypt(public_key, message)
print("Encrypted message:", encrypted_msg)
decrypted_msg = decrypt(private_key, encrypted_msg)
print("Decrypted message:", decrypted_msg)
if __name__ == "__main__":
main()
```

OUTPUT:

```
Public key: ((1171, 3233), (1691, 3233))
Private key: ((1691, 3233), (1171, 3233))
Original message: Hello, RSA!
Encrypted message: [3045, 2637, 1457, 1457, 2845, 2969, 2637, 1457, 2925, 2969, 1457]
Decrypted message: Hello, RSA!
```

RESULT:

Thus, the above python program to implement RSA Encryption Algorithm has been verified and executed successfully.

EX NO:9 Write a python program to implement the Diffie-HellmanKey Exchange mechanism. Consider one of the parties as Alice and the other party as Bob.

AIM:

To implement a python program to implement the Diffie-HellmanKey Exchange mechanism.

ALGORITHM:

STEP 1: Initialization:

Alice and Bob agree on a large prime number *p* and a primitive root modulo *p*,

denoted as *g*. These values are public.

Both Alice and Bob choose their own secret values:

Alice chooses a secret integer *a*.

Bob chooses a secret integer *b*.

STEP 2: Compute Public Values

STEP 3: Exchange Public Values

Alice sends her public value *A* to Bob.

Bob sends his public value *B* to Alice.

STEP 4: Finalize

Alice and Bob now both have the same shared secret key.

This shared secret key can be used as a symmetric key for encryption and decryption.

PROGRAM:

```
def mod_exp(base, exponent, modulus):
    result = 1
    base = base % modulus
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        exponent = exponent // 2
        base = (base * base) % modulus
    return result
def generate_key(prime, primitive_root):
private_key = int(input("Enter your private key: "))
public_key = mod_exp(primitive_root, private_key, prime)
    return private_key, public_key
def compute_shared_secret(private_key, other_public_key, prime):
shared_secret = mod_exp(other_public_key, private_key, prime)
    return shared_secret
def main():
prime = int(input("Enter the prime number (p): "))
primitive_root = int(input("Enter the primitive root (g): "))
print("\nAlice's side:")
```

```
alice_private_key, alice_public_key = generate_key(prime, primitive_root)

print("Alice's private key:", alice_private_key)

print("Alice's public key:", alice_public_key)

print("\nBob's side:")

bob_private_key, bob_public_key = generate_key(prime, primitive_root)

print("Bob's private key:", bob_private_key)

print("Bob's public key:", bob_public_key)

print("\nComputing shared secret...")

alice_shared_secret = compute_shared_secret(alice_private_key, bob_public_key, prime)

bob_shared_secret = compute_shared_secret(bob_private_key, alice_public_key, prime)

print("\nShared secret computed by Alice:", alice_shared_secret)

print("Shared secret computed by Bob:", bob_shared_secret)

  if alice_shared_secret == bob_shared_secret:

print("\nShared secret matched! Secure communication established.")

    else:

print("\nShared secret mismatch! Failed to establish secure communication.")

if __name__ == "__main__":

main()
```

OUTPUT

```
Enter the prime number (p): 23
Enter the primitive root (g): 5
Alice's side:
Enter your private key: 6
Alice's private key: 6
Alice's public key: 8
Bob's side:
Enter your private key: 15
Bob's private key: 15
Bob's public key: 19
Computing shared secret...
Shared secret computed by Alice: 2
Shared secret computed by Bob: 2
Shared secret matched! Secure communication established.
```

RESULT:

Thus, the above python program to implement the Diffie-HellmanKey Exchange mechanism has been verified and executed successfully.

EX NO:10 Write a program to calculate the message digest of a text using the SHA-1 Algorithm

AIM:

To implement a pyhton program to calculate the message digest of a text using the SHA-1 Algorithm.

ALGORITHM:

STEP 1: Padding:

Append a single '1' bit to the message.

Append '0' bits until the length of the padded message in bits is congruent to 448 mod 512.

Append the length of the original message in bits as a 64-bit big-endian integer.

STEP 2: Message Parsing

Divide the padded message into blocks of 512 bits (64 bytes).

STEP 3: Initialize Hash Values

STEP 4: Concatenate the hash values to get the final SHA-1 hash value.

PROGRAM

```
import hashlib

def calculate_sha1(text):
    sha1_hash = hashlib.sha1(text.encode()).hexdigest()
    return sha1_hash

def main():
```

```
    text = input("Enter the text to calculate SHA-1 hash: ")
    sha1_hash = calculate_sha1(text)
print("SHA-1 hash of the text:", sha1_hash)
if __name__ == "__main__":
main()
```

OUTPUT:

```
Enter the text to calculate SHA-1 hash: Hello, SHA-1!
SHA-1 hash of the text: 95e3e81b0d6f5a050bfaba65f7d8412c5042e7d2
```

RESULT:

Thus, the above python program to calculate the message digest of a text using the SHA-1 Algorithm has been verified and executed successfully

EX NO:11 Write a python program to calculate the message digest of a text using the MD- 5algorithm

AIM:

To implement a python program to calculate the message digest of a text using the MD-5 algorithm.

ALGORITHM:

STEP1: Padding:

Append a single '1' bit to the message.

Append '0' bits until the length of the padded message in bits is congruent to 448 mod 512.

Append the length of the original message in bits as a 64-bit little-endian integer.

STEP 2: Initialize Variables:

Initialize four 32-bit variables *A*, *B*, *C*, and *D* to specific constants. These are the initial hash values.

STEP 3: Process Message in 512-bit Blocks:

STEP4: Divide the padded message into blocks of 512 bits (64 bytes).

STEP 5: Break each block into sixteen 32-bit words.

STEP 6: Concatenate the hash values *A*, *B*, *C*, and *D* to get the final MD5 message digest.

PROGRAM:

```
import hashlib
def calculate_md5(text):
    md5_hash = hashlib.md5(text.encode()).hexdigest()
    return md5_hash
def main():
    text = input("Enter the text to calculate MD5 hash: ")
    md5_hash = calculate_md5(text)
print("MD5 hash of the text:", md5_hash)
if __name__ == "__main__":
main()
```

OUTPUT:

```
Enter the text to calculate MD5 hash: Hello, MD5!
MD5 hash of the text: a4d6a0bf5b7dc51d783a6d93b0003f2a
```

RESULT:

Thus, the above python program to calculate the message digest of a text using the MD-5 algorithm has been verified and executed successfully.